



Escuela Técnica Superior de
Ingeniería Informática

TRABAJO FIN DE GRADO

Análisis y optimización de modelos de lenguaje en plataformas embebidas

Realizado por
Jesús Carrascosa Carro 

Para la obtención del título de
Grado en Ingeniería Informática - Ingeniería de Computadores

Dirigido por
Jorge Fernández Berni
Delia Velasco Montero

En el departamento de
Electrónica y Electromagnetismo

Convocatoria de julio, curso 2025/26

A Oli, Toby I, Melchor, Trilo, Toby II «El Choco», Iwi, Ufran y Noa.

Agradecimientos

A mi familia, por su apoyo a lo largo de la carrera.

A mi prima Rocío y a mi tía María Jesús, por haber leído este Trabajo de Fin de Grado innumerables veces.

A mis tutores, Jorge y Delia, por su dirección, sugerencias y correcciones, así como por la paciencia y amabilidad mostradas ante las numerosas preguntas (y correos) surgidas durante la elaboración de este trabajo.

A los docentes que me acompañaron antes de iniciar la carrera, en especial a mis profesores particulares, José Manuel e Isabel, sin cuya ayuda no habría logrado el acceso al grado; y a todo el profesorado que me ha formado durante todos estos años.

A la unidad de atención a estudiantes con discapacidad del SACU, especialmente a Mercedes de Lemus, sin cuyo apoyo probablemente no habría podido finalizar mis estudios.

A las asociaciones SUGUS e IOTUS y, en particular, a esta última, por haberme brindado la oportunidad de ejercer como presidente y vicepresidente. Espero haberos aportado tanto como vosotros me habéis aportado a mí.

A la Universidad de Shih Chien (Taiwán), al Beijing Institute of Technology (campus de Pekín y de Zhuhai) y a la Università degli Studi di Genova (Italia), por su hospitalidad y recibimiento durante mis intercambios académicos.

Resumen

La proliferación de los modelos de lenguaje grandes (LLM) ha impulsado una creciente dependencia de servicios en la nube, lo que plantea limitaciones de privacidad, latencia y conectividad en entornos con recursos restringidos. Los modelos de lenguaje pequeños (SLM), con entre mil y cuatro mil millones de parámetros, ofrecen una alternativa viable para la inferencia local en dispositivos embebidos de bajo consumo. Este trabajo estudia de forma sistemática la viabilidad y optimización de la inferencia de SLM sobre la Raspberry Pi 5 (SoC BCM2712, 8 GB LPDDR4X), evaluando seis factores experimentales mediante una figura de mérito que combina *throughput*, eficiencia energética y número de parámetros.

El trabajo realiza dos contribuciones de *software*: *MonitorSystem*, un sistema de captura de métricas de inferencia y hardware en tiempo real implementado en C++, y *monitorviz*, una librería Python que automatiza el cálculo de métricas derivadas y la generación de visualizaciones comparativas. Sobre esta infraestructura se diseñan y ejecutan seis series experimentales (E0–E5) que evalúan, respectivamente, la caracterización base de seis SLM, el nivel de cuantización, el tamaño de la ventana de contexto, el *batch size* de prefill, el motor de inferencia y el acelerador hardware Hailo-10H.

Los resultados muestran que la refrigeración activa es un requisito habilitante: su supresión reduce el *throughput* hasta en un 53 % y deja cuatro de los seis modelos por debajo del umbral de usabilidad interactiva. La cuantización a 4 bits constituye el punto de equilibrio óptimo en esta plataforma limitada por ancho de banda de memoria, mientras que 8 bits y precisión completa resultan contraproducentes. La ventana de contexto y el *batch size* de prefill tienen impacto menor o nulo en escenarios de flujo único. llama.cpp supera a Ollama en latencia de primer *token* hasta en un factor de seis, con *throughput* de decodificación equivalente. El acelerador Hailo-10H reduce el *throughput* entre un 34 y un 47 %, pero ofrece un balance favorable en el criterio combinado para Llama-3.2-3B ($\text{FoM}_{\text{red}} = 1,107$) gracias a una reducción del 57 % en el coste energético por *token*.

Palabras clave: modelos de lenguaje pequeños, LLM, SLM, inferencia en el borde, edge AI, Raspberry Pi 5, cuantización, llama.cpp, Hailo-10H, AI HAT+ 2, eficiencia energética, figura de mérito, Ollama.

Abstract

The proliferation of large language models (LLMs) has driven a growing dependence on cloud services, posing limitations in terms of privacy, latency, and connectivity for resource-constrained environments. Small language models (SLMs), ranging from one to four billion parameters, offer a viable alternative for local inference on low-power embedded devices. This work systematically investigates the feasibility and optimisation of SLM inference on the Raspberry Pi 5 (BCM2712 SoC, 8 GB LPDDR4X), evaluating six experimental factors through a figure of merit that combines throughput, energy efficiency, and parameter count.

The work delivers two software contributions: *MonitorSystem*, a real-time inference and hardware metrics capture system implemented in C++, and *monitorviz*, a Python library that automates the computation of derived metrics and the generation of comparative visualisations. On this infrastructure, six experimental series (E0–E5) are designed and executed, evaluating respectively the baseline characterisation of six SLMs, quantization level, context window size, prefill batch size, inference engine, and the Hailo-10H hardware accelerator.

Results show that active cooling is an enabling requirement: its removal reduces throughput by up to 53 % and renders four of the six models non-interactive. 4-bit quantization represents the optimal trade-off on this memory-bandwidth-bound platform, while 8-bit and full-precision formats prove counterproductive. Context window size and prefill batch size have negligible impact in single-stream scenarios. llama.cpp outperforms Ollama in time-to-first-token by up to a factor of six, with equivalent decoding throughput. The Hailo-10H accelerator reduces throughput by 34–47 %, but yields a favourable combined score for Llama-3.2-3B ($\text{FoM}_{\text{red}} = 1.107$) through a 57 % reduction in energy cost per token.

Keywords: LLM, small language models, SLM, edge inference, edge AI, Raspberry Pi 5, quantization, llama.cpp, Hailo-10H, energy efficiency, AI HAT+ 2, figure of merit, Ollama.

Índice general

1	Introducción	1
1.1.	Contexto	1
1.2.	Objetivos del TFG	3
1.3.	Estructura	3
2	Fundamentos Teóricos y Estado Del Arte	4
2.1.	Evolución histórica: de los sistemas simbólicos a los LLM	4
2.2.	El Transformer y la arquitectura de los LLM	19
2.2.1.	Estructura del Transformer	19
2.2.2.	Variaciones arquitectónicas del Transformer	23
2.2.3.	Variaciones internas del Transformer	23
2.3.	Principales arquitecturas y variantes	25
2.3.1.	Modelos monolíticos (densos)	25
2.3.2.	Modelos de razonamiento	25
2.3.3.	Mixture of Experts	25
2.3.4.	Alternativas al Transformer	26
2.4.	Formatos numéricos y cuantización	26
2.4.1.	Evolución de los formatos base de entrenamiento	27
2.4.2.	Fundamentos y tipos de cuantización	28
2.5.	Otros métodos de reducción	29
2.5.1.	Poda de parámetros (<i>Pruning</i>)	29
2.5.2.	Destilación de conocimiento (<i>knowledge distillation</i>)	29
2.6.	Métodos de actualización post-entrenamiento	29
2.6.1.	Ajuste fino eficiente en parámetros (PEFT)	30
2.6.2.	Generación aumentada por recuperación (RAG)	31
2.7.	Funcionamiento de la inferencia	33
2.7.1.	Pipeline lógico de inferencia	33
2.7.2.	La inferencia vista desde el hardware	33
2.7.3.	La inferencia desde los motores de inferencia	35
2.7.4.	Métricas de evaluación de la inferencia	36
2.8.	Estado del arte	38
2.8.1.	De los motores de inferencia	38
2.8.2.	De los modelos	40
2.8.3.	De los análisis de optimización	45
3	Metodología	48
3.1.	Objetivos metodológicos y cuestiones de investigación	48
3.1.1.	Métricas empleadas	52
3.2.	Diseño del sistema de medidas y de visualizaciones	56
3.2.1.	MonitorSystem: requisitos y arquitectura	56
3.2.2.	<i>monitoroiz</i> : requisitos y arquitectura	58
3.2.3.	Limitaciones	58
3.3.	Diseño experimental	59
3.3.1.	Plataforma hardware	59
3.3.2.	Configuración de referencia	60

3.3.3.	Protocolo de medida por serie experimental	60
3.4.	Preparación del entorno experimental	62
3.4.1.	Configuración del sistema	62
3.4.2.	Instalación y configuración de los motores de inferencia	62
3.4.3.	Preparación de los modelos	63
3.5.	Planificación y recursos	64
4	Resultados	68
4.1.	Matriz experimental	68
4.1.1.	Serie E0: línea base y refrigeración	69
4.1.2.	Serie E1: cuantización	74
4.1.3.	Serie E2: tamaño de contexto	79
4.1.4.	Serie E3: tamaño de batch de prefill	83
4.1.5.	Serie E4: motor de inferencia	88
4.1.6.	Serie E5: acelerador Hailo-10H	91
	Síntesis global	93
5	Conclusiones	94
A	Especificación de Requisitos del Software (MonitorSystem)	99
A.1.	Introducción	99
A.1.1.	Propósito	99
A.1.2.	Alcance	99
A.1.3.	Definiciones, acrónimos y abreviaturas	99
A.1.4.	Referencias	100
A.2.	Glosario de términos	100
A.3.	Objetivos del sistema (OBJ)	101
A.4.	Requisitos de información (RI)	102
A.4.1.	RI-01: Configuración de experimento	102
A.4.2.	RI-02: Métricas de inferencia por <i>prompt</i>	103
A.4.3.	RI-03: Métricas <i>hardware</i> periódicas	104
A.4.4.	RI-04: Resumen de ejecución	104
A.4.5.	RI-05: <i>Dataset</i> de <i>prompts</i>	105
A.4.6.	RI-06: Metadatos de arquitectura del modelo	105
A.5.	Reglas de negocio (RN)	106
A.6.	Requisitos funcionales (RF)	107
	RF-01 – Lectura de configuración desde fichero JSON	107
	RF-02 – Lectura de configuración desde argumento inline	107
	RF-03 – Validación de la configuración de entrada	107
	RF-04 – Ejecución con OLLAMA (TYPE_0)	107
	RF-05 – Ejecución con OLLAMA (TYPE_1)	108
	RF-06 – Ejecución con OLLAMA (TYPE_2)	108
	RF-07 – Ejecución con LLAMA	108
	RF-08 – Ejecución con HAILO_OLLAMA	108
	RF-09 – Monitorización <i>hardware</i> periódica	109
	RF-10 – Registro de métricas de inferencia	109
	RF-11 – Generación del resumen de ejecución	109
	RF-12 – Extracción de metadatos del modelo	109
	RF-13 – Descarga automática de modelos Ollama	109
	RF-14 – Creación del directorio de resultados	110
	RF-15 – Visualización de uso	110

A.7.	Requisitos no funcionales (RNF)	110
A.7.1.	Usabilidad	110
A.7.2.	Rendimiento	110
A.7.3.	Fiabilidad	110
A.7.4.	Portabilidad	111
A.7.5.	Mantenibilidad y extensibilidad	111
A.7.6.	Interoperabilidad	111
A.7.7.	Reproducibilidad científica	112
A.8.	Pruebas de aceptación (PA)	112
PA-01		112
PA-02		112
PA-03		112
PA-04		113
PA-05		113
PA-06		113
PA-07		113
PA-08		114
PA-09		114
PA-10		114
PA-11		114
PA-12		115
A.9.	Matrices de trazabilidad	115
A.9.1.	Requisitos funcionales ↔ Objetivos	115
A.9.2.	Requisitos funcionales ↔ Requisitos de información	116
A.9.3.	Pruebas de aceptación ↔ Requisitos funcionales y reglas de negocio	116
B	Especificación de Requisitos del Software (monitorviz)	117
B.1.	Introducción	117
B.1.1.	Propósito	117
B.1.2.	Alcance	117
B.1.3.	Definiciones, acrónimos y abreviaturas	117
B.1.4.	Referencias	118
B.2.	Glosario de términos del dominio	118
B.3.	Requisitos generales (objetivos)	119
B.4.	Requisitos de información	120
B.5.	Reglas de negocio	123
B.6.	Requisitos funcionales	124
B.6.1.	Carga de datos	124
B.6.2.	Transformaciones y métricas derivadas	125
B.6.3.	Filtrado e inspección de colecciones	126
B.6.4.	Visualización	126
B.7.	Requisitos no funcionales	126
B.7.1.	Usabilidad (ISO 9126: Usability)	127
B.7.2.	Rendimiento (ISO 9126: Efficiency)	127
B.7.3.	Fiabilidad (ISO 9126: Reliability)	127
B.7.4.	Compatibilidad (ISO 9126: Portability/Compatibility)	127
B.7.5.	Mantenibilidad (ISO 9126: Maintainability)	128
B.7.6.	Seguridad (ISO 9126: Security)	128
B.8.	Pruebas de aceptación	128
B.9.	Matriz de trazabilidad	129

B.9.1.	OBJ $\rightarrow$ RF, RN, PA	129
B.9.2.	PA $\rightarrow$ RF, RN	130
C	Tablas de métricas completas	131
C.1.	Serie E0 —Caracterización base (FAN y NOFAN)	131
C.2.	Serie E1 —Sensibilidad a la cuantización	133
C.3.	Serie E2 —Sensibilidad al tamaño de contexto	136
C.4.	Serie E3 —Sensibilidad al tamaño de lote	139
C.5.	Serie E4 —Sensibilidad al motor de inferencia (llama.cpp frente a Ollama) . .	142
C.6.	Serie E5 —Comparativa CPU frente a Hailo-10H	144

Índice de figuras

2.1.	Diagrama de flujo (izquierda) y tabla de reglas operacionales (derecha) del experimento IBM-Georgetown (1954), mostrando traducción ruso-inglés con 6 reglas sintácticas y códigos diacríticos. Fuente: [25].	5
2.2.	Árbol sintáctico de la frase de Chomsky “ <i>Colorless green ideas sleep furiously</i> ”, ilustrando gramática generativa correcta pero semánticamente absurda. Fuentes: [29], [30].	5
2.3.	Arquitectura de Collobert y Weston (2008) con capacidad multitarea. Conformada por las capas de <i>lookup table</i> , convoluciones TDNN, <i>max-pooling</i> , redes neuronales clásicas y salida <i>softmax</i> . Fuente: [40].	8
2.4.	Diagrama conceptual del funcionamiento de la RNN-LM de Mikolov <i>et al.</i> (2010) [41]	9
2.5.	Celda LSTM (izquierda) y arquitectura RNN-LSTM (derecha). Traducidos de [42].	10
2.6.	Word2Vec (Mikolov 2013): CBOW (contexto → palabra) vs. Skip-gram (palabra → contexto). Traducido de [44], [45].	10
2.7.	Arquitectura RNN codificador-decodificador (izquierda) y diagrama de las puertas adaptativas <i>reset/update</i> (derecha) propuesta por [46].	11
2.8.	Seq2Seq LSTM de Sutskever <i>et al.</i> (2014): <i>encoder</i> de 4 capas, <i>decoder</i> autorregresivo con inversión de fuente. Fuente: [47]	12
2.9.	Funcionamiento de la atención de Bahdanau <i>et al.</i> (2014); las <i>x</i> representan las entradas del codificador, las <i>h</i> los estados ocultos del decodificador, las <i>a</i> los pesos de la atención, y las <i>s</i> los estados ocultos del decodificador. Fuente: [48].	12
2.10.	Reformulación de múltiples tareas de PLN a un formato estándar de texto a texto procesado por el modelo T5. Fuente: [53]	14
2.11.	Arquitectura general del modelo Transformer. Fuente: [2]	20
2.12.	Composición de un <i>adapter</i> (adaptador) (izquierda) y ubicación del adaptador dentro del Transformer (derecha). Fuente: [134]	31
2.13.	Diagrama de flujo del paradigma RAG (Retrieval-Augmented Generation). .	32
2.14.	Diagrama de alto nivel del <i>pipeline</i> lógico de inferencia. Extraído de: [22]. . . .	33
2.15.	Modelo <i>Roofline</i> para la inferencia de LLMs. Se observa cómo la Generación de <i>tokens</i> está limitada por el ancho de banda de memoria (<i>memory-bound</i>), mientras que el Procesamiento del <i>Prompt</i> (<i>prefill</i>) tiende a estar limitado por la capacidad de cómputo (<i>compute-bound</i>) debido a su mayor intensidad operativa (OI). Extraído de [22].	34
2.16.	Consumo de RAM (MB) y CPU (%) a lo largo del tiempo para el modelo DeepSeek-R1-Distill-Qwen-1.5B. Las delimitaciones verticales discontinuas indican el comienzo de las fases secuenciales de carga de pesos (<i>model loading</i> , verde), procesamiento de entrada (<i>prefill</i> , naranja) y generación de <i>tokens</i> (<i>decode</i> , morado).	35
3.1.	Diagrama de Gantt del proyecto. Las fases F1–F8 se corresponden con las descritas en la Tabla 3.2.	66
4.1.	Frontera de Pareto velocidad–huella en la serie E0	70
4.2.	Descomposición de la FoM _{full} por ejes en la serie E0	71

4.3. Impacto de la refrigeración activa sobre el <i>throughput</i>	72
4.4. Desglose temporal por fases de inferencia en la serie E0	72
4.5. Mapa de calor de FoM _{full} y aceleración por cuantización	76
4.6. Figura de mérito frente a bits por peso en la serie E1	76
4.7. Perplejidad WikiText-2 por modelo y cuantización (E1)	77
4.8. Comparación entre cuantizaciones K y no-K en la serie E1	77
4.9. Huella en RAM frente al tamaño de contexto en la serie E2	79
4.10. <i>Throughput</i> frente al tamaño de contexto en la serie E2	81
4.11. Figura de mérito frente al tamaño de contexto en la serie E2	81
4.12. Latencia de primer <i>token</i> frente al tamaño de contexto en la serie E2	82
4.13. <i>Throughput</i> frente al tamaño de batch en la serie E3	83
4.14. Latencia de primer <i>token</i> frente al tamaño de batch en la serie E3	85
4.15. Coste energético frente al tamaño de batch en la serie E3	86
4.16. Figura de mérito frente al tamaño de batch en la serie E3	87
4.17. Variación de <i>throughput</i> entre motores en la serie E4	89
4.18. Utilización de <i>hardware</i> por motor en la serie E4	90

Índice de tablas

2.1. Evolución de modelos frontera	15
2.2. Parámetros de inferencia y su impacto en el rendimiento del hardware.	36
2.3. Conjunto de modelos seleccionados para el análisis experimental, con sus principales características y criterio de inclusión.	45
3.1. Matriz experimental del estudio. La configuración de referencia para todas las series con refrigeración activa es: motor llama.cpp, <i>batch size</i> = 512, contexto = 4 096 tokens, temperatura = 0,0, semilla = 42, 15 <i>prompts</i> del conjunto <i>instruction_following_eval</i> [170]. La serie E5 emplea un contexto de 2 048 tokens por restricción del compilado del modelo HEF ¹	51
3.2. Fases del proyecto y períodos de ejecución.	65
4.1. Caracterización base de los modelos en la serie E0	69
4.2. Tabla maestra de la serie E1 (cuantización)	75
4.3. Tabla selectora de cuantización en la serie E1	76
4.4. Caracterización por tamaño de contexto en la serie E2	80
4.5. Caracterización por tamaño de batch en la serie E3	84
4.6. Comparación de motores de inferencia en la serie E4	88
4.7. Comparación Hailo-10H frente a CPU en la serie E5	91
A.17. Matriz de trazabilidad: Requisitos funcionales vs. Objetivos del sistema	115
A.18. Matriz de trazabilidad: Requisitos funcionales vs. Requisitos de información .	116
C.1. E0 —Rendimiento e inferencia (FAN y NOFAN)	131
C.2. E0 —Métricas de hardware (FAN y NOFAN)	132
C.3. E0 —Eficiencia computacional (FAN y NOFAN)	132
C.4. E1 —Rendimiento por cuantización	133
C.5. E1 —Métricas de hardware por cuantización	134
C.6. E1 —Eficiencia computacional por cuantización	135
C.7. E2 —Rendimiento por contexto	136
C.8. E2 —Métricas de hardware por contexto	137
C.9. E2 —Eficiencia computacional por contexto	138
C.10. E3 —Rendimiento por batch size	139
C.11. E3 —Métricas de hardware por batch size	140
C.12. E3 —Eficiencia computacional por batch size	141
C.13. E4 —Rendimiento por engine	142
C.14. E4 —Métricas de hardware por engine	143
C.15. E4 —Eficiencia computacional por engine	144
C.16. E5 —Métricas reducidas completas (CPU vs Hailo-10H)	144
C.17. E5 —Métricas de hardware CPU	145
C.18. E5 —Eficiencia computacional CPU	145

1. Introducción

1.1. Contexto

Durante la última década, la confluencia de varios factores tecnológicos ha transformado radicalmente el panorama de la inteligencia artificial (IA). El incremento sostenido de la capacidad de cómputo, impulsado por el uso masivo de procesadores gráficos (GPU) y aceleradores especializados, junto con la disponibilidad de grandes volúmenes de datos (*big data*), conjuntos de datos estructurados a gran escala y los avances en aprendizaje automático (*machine learning*) y profundo (*deep learning*), ha permitido entrenar modelos de una complejidad sin precedentes [1]. Estos avances no solo han ampliado las capacidades de los modelos, sino también los escenarios en los que resulta viable adquirir y procesar información.

La síntesis de estos avances ha tenido un impacto especialmente disruptivo en el procesamiento del lenguaje natural (PLN), culminando en el desarrollo de los Modelos de Lenguaje de Gran Tamaño (LLM, por sus siglas en inglés). Fundamentados mayoritariamente en la arquitectura Transformer [2], y con una complejidad que escala hasta los cientos de miles de millones de parámetros, su adopción frenética en forma de servicios en la nube se precipitó con la introducción de ChatGPT [3] por parte de OpenAI. Este hito desencadenó una carrera tecnológica con la entrada de competidores como Anthropic, Google (con Gemini) y DeepSeek, entre otros. La utilidad de estos modelos ha trascendido rápidamente las tareas clásicas de traducción, generación y síntesis textual, demostrando capacidades emergentes en la generación de código, el razonamiento lógico complejo y la extracción estructurada de información en múltiples dominios del conocimiento [1], [4], [5].

Esta adopción masiva no ha estado exenta de polémicas y riesgos. A las dimensiones éticas, laborales, educativas, académicas y medioambientales se suman problemas heredados del paradigma de la algoritmización y el *big data*, como los sesgos y la falta de transparencia en su funcionamiento [6], [7]. A ello se añade una marcada dependencia tecnológica respecto de los principales actores del sector, Estados Unidos y China [8]. Esta situación ha impulsado una carrera por el desarrollo de modelos regionales, como ALIA (España) [9], [10] y Mistral (Francia) [11]. Tales iniciativas, sumadas a marcos regulatorios como la AI Act de la Unión Europea [12], buscan recuperar la soberanía tecnológica frente a desafíos como el atraso científico, la protección de la privacidad, el cumplimiento de las regulaciones vigentes, así como la censura y la desinformación.

A pesar de sus capacidades, el despliegue de estos modelos masivos, conocidos también como modelos frontera (*frontier models*), se encuentra con un obstáculo fundamental: la enorme cantidad de recursos computacionales que exigen. Esto relega su ejecución a grandes proveedores de servicios en la nube (centros de datos), lo que suscita tres problemas principales en entornos de aplicación real [13]. En primer lugar, introduce una dependencia de conectividad continua, con la latencia y la variabilidad asociadas a la red. En segundo lugar, obliga a la transmisión de datos potencialmente sensibles a infraestructuras externas, lo que amplía la superficie de exposición y plantea desafíos en materia de confidencialidad y gobernanza de la información. En tercer lugar, consolida una dependencia infraestructural respecto a servicios de terceros, con la consiguiente pérdida

de control operativo, limitaciones en la auditabilidad y posibles restricciones derivadas de políticas de uso, costes o disponibilidad.

Estos factores han motivado el desplazamiento de la inferencia hacia el propio dispositivo de adquisición, paradigma conocido como inteligencia artificial en el borde (*edge AI*), en el que el procesamiento se realiza *in situ* reenviando únicamente los resultados. Este enfoque mitiga directamente los tres problemas anteriores: elimina la dependencia de conectividad continua, reduce la latencia, preserva la privacidad al evitar la transmisión de datos en crudo y disminuye el consumo asociado a la comunicación. No obstante, trasladar modelos de lenguaje a este paradigma exige reconciliar su elevada demanda de recursos con las severas restricciones de cómputo, memoria y energía propias de los sistemas embebidos.

Es en este escenario de dependencia de servicios en la nube donde surgen los Modelos de Lenguaje Pequeños (SLM, por sus siglas en inglés) [14]. Aunque mantienen una arquitectura análoga a la de sus contrapartes masivas, su escala reducida, situada entre millones y pocos miles de millones de parámetros, permite su ejecución en dispositivos de consumo mediante herramientas de inferencia local como llama.cpp u Ollama. No obstante, el despliegue de SLM en plataformas embebidas (*edge*) sigue representando un desafío importante, dado que el coste energético y de cómputo continúa siendo elevado para los estándares de los sistemas embebidos.

La conservación de la biodiversidad constituye uno de los ámbitos donde estas necesidades resultan más acuciantes. La aceleración de la pérdida de biodiversidad y la creciente presión sobre los ecosistemas han evidenciado los límites de los enfoques de monitorización clásicos frente a la velocidad de los cambios globales, lo que ha impulsado la incorporación de tecnologías de IA para mejorar el seguimiento ecológico y la toma de decisiones basada en datos [15]-[17]. En este escenario, los entornos de implantación, situados con frecuencia en zonas remotas y de baja cobertura, hacen inviable la transmisión continua de datos y exigen su procesamiento *in situ*, lo que sitúa al paradigma del borde como condición habilitante.

Este trabajo surge en el marco de una línea de investigación desarrollada en la Universidad de Sevilla cuyo objetivo es la monitorización de especies y el seguimiento en campo de diversas métricas de biodiversidad mediante fototrampeo inteligente [18] y análisis ecoacústico [19]. En este contexto, los modelos de lenguaje desplegados localmente permitirían compilar resúmenes en lenguaje natural de los eventos más relevantes vinculados a los periodos de monitorización programados; por ejemplo, la presencia de especies de particular interés, el incremento en el número de cantos de aves como indicador de la densidad de poblaciones, o la detección de actividad furtiva. Asimismo, los gestores del medio natural, ecólogos y biólogos podrían interrogar al modelo para recabar información adicional sobre dichos periodos, en línea con el potencial que la literatura reciente atribuye a estos modelos para la síntesis de evidencia y el apoyo a la toma de decisiones en conservación [20]. En última instancia, se apunta a sistemas inteligentes desplegados en campo que resultarían casi indistinguibles de un observador humano situado de forma permanente en la misma posición, capaz de proporcionar informes recurrentes sobre lo observado y de interactuar con el usuario [21].

En este contexto, el presente trabajo tiene como objetivo analizar y desarrollar técnicas de reducción y optimización de modelos de lenguaje que permitan su implementación eficiente en dispositivos de recursos limitados y restricciones energéticas.

1.2. Objetivos del TFG

Dado el contexto anterior, el objetivo general de este Trabajo de Fin de Grado es analizar y optimizar las técnicas de reducción de modelos de lenguaje (entre ellas la cuantización, la poda y la destilación de conocimiento) para su implementación eficiente en plataformas de bajo consumo, contemplando asimismo el uso de aceleradores comerciales de inferencia. Ello implica una investigación sistemática de las compensaciones (*trade-offs*) entre eficiencia, entendida como latencia, rendimiento y consumo energético, y precisión, entendida como calidad de la inferencia, en el *hardware* de borde.

Para alcanzar este objetivo general, se definen los siguientes objetivos específicos:

- Estudiar los modelos de lenguaje del estado del arte compatibles con dispositivos embebidos, atendiendo a sus requisitos de cómputo, memoria y energía.
- Implementar una diversidad de modelos de lenguaje en dispositivos embebidos, incluyendo el uso de aceleradores *hardware* de inferencia.
- Desarrollar una aplicación de medida que registre métricas de *hardware* (consumo de recursos y estado del dispositivo, como uso de CPU, RAM, temperatura o ventilador), métricas de inferencia (tiempos de carga de los modelos, de evaluación y de generación) y su relación con los parámetros modificables de los modelos (ventana de contexto, tamaño de *batch*, temperatura, semillas de generación, entre otros).
- Visualizar las métricas anteriores mediante gráficas u otros recursos visuales que faciliten su interpretación.
- Analizar los resultados, comparar los modelos evaluados y extraer conclusiones, guías de uso y recomendaciones, fundamentadas en las métricas obtenidas, para el despliegue de modelos de lenguaje en entornos embebidos reales.

1.3. Estructura

El presente trabajo se organiza en cinco capítulos y tres anexos. El **Capítulo 2** asienta los fundamentos teóricos necesarios para contextualizar el estudio: evolución histórica de los modelos de lenguaje, arquitectura Transformer y sus variantes, técnicas de reducción de modelos (cuantización, poda y destilación), pipeline lógico de inferencia y revisión del estado del arte en motores de inferencia, modelos y análisis de optimización en plataformas embebidas. El **Capítulo 3** describe la metodología adoptada: las cuestiones de investigación e hipótesis de trabajo, el conjunto de métricas empleadas, el diseño e implementación del sistema de medida (MonitorSystem, en C++) y de la librería de visualización (monitorviz, en Python), el diseño experimental con las seis series y la configuración de referencia, la preparación del entorno, y la planificación del proyecto. El **Capítulo 4** presenta los resultados de las series experimentales E0–E5, con el análisis de los efectos de la refrigeración, el nivel de cuantización, el tamaño de contexto, el batch size de prefill, el motor de inferencia y el acelerador Hailo-10H sobre el rendimiento, la eficiencia energética y la usabilidad de los modelos evaluados. El **Capítulo 5** recoge las conclusiones del trabajo, contrasta los resultados con las hipótesis formuladas y propone líneas de investigación futura. Finalmente, los **Anexos A y B** contienen las especificaciones de requisitos del software de MonitorSystem y monitorviz respectivamente, elaboradas según la metodología IISSI; el **Anexo C** recopila las tablas de datos completas de todas las series experimentales.

2. Fundamentos Teóricos y Estado Del Arte

Este capítulo asienta los fundamentos teóricos y revisa el estado del arte que sustentan el presente trabajo. Su eje son los modelos de lenguaje de gran tamaño (LLM): sistemas que generan lenguaje natural mediante la estimación probabilística de secuencias de *tokens*¹ y que se distinguen de sus predecesores por la magnitud conjunta de sus parámetros, datos de entrenamiento y coste computacional [23]. A partir de esta noción, se recorre su evolución histórica, los principios que rigen su funcionamiento y las técnicas de optimización que hacen viable su despliegue en plataformas de recursos limitados.

2.1. Evolución histórica: de los sistemas simbólicos a los LLM

Históricamente, los fundamentos de los modelos de lenguaje han experimentado una evolución constante. Este progreso abarca desde los primeros enfoques estadísticos y las redes neuronales básicas, hasta la consolidación del aprendizaje profundo (*Deep Learning*), culminando con la arquitectura Transformer y sus modelos derivados con numerosas variaciones y enfoques recientes [24].

A continuación, se presenta un recorrido cronológico por la evolución de los grandes modelos de lenguaje y sus antecedentes, fundamentado principalmente en los trabajos de Wang *et al.* [24] y Kamath *et al.* [23]. Este análisis ilustra el cambio de paradigma desde el tránsito de la rigidez de los sistemas regidos por normas explícitas codificadas manualmente hacia arquitecturas de aprendizaje profundo capaces de inferir la estructura sintáctica y semántica directamente a partir de los datos, que se divide en 5 fases.

1950-1970: Sistemas simbólicos y primeros pasos.

Este periodo se caracterizó por el desarrollo de sistemas basados en reglas y estructuras explícitas de origen sintáctico-gramatical.

Comenzó con el Experimento IBM-Georgetown [25]-[27]. Presentado el 7 de enero de 1954, fue pionero en la traducción automática de 60 frases entre ruso e inglés, demostrando la viabilidad técnica de la traducción automática. Compuesto por un corpus² de 250 palabras en ruso latinizado pertenecientes a ámbitos especializados como la política, química o matemáticas permitía la traducción basada en un sistema de 6 reglas de “Sintaxis Operacional” y códigos metalingüísticos asociados a cada palabra codificantes de instrucciones lógicas que entre otras cosas permitía la polisemia, tal y como se puede apreciar en la Figura 2.1.

¹Unidad mínima y discreta (palabra, fragmento o carácter) en la que se divide un texto para su procesamiento numérico [22].

²En el contexto del procesamiento del lenguaje natural se puede definir como *corpus* al conjunto de palabras, textos, documentos o ejemplos lingüísticos, seleccionados, organizados y procesados de manera sistemática, que sirve como fuente de datos para el entrenamiento, validación y evaluación de modelos lingüísticos [28, acep. 2].

ELIZA(1966)[31]⁴ y SHRDLU (1971) [33], que empleaban reglas sintácticas y semánticas predefinidas [23]:

ELIZA⁵, desarrollado por Joseph Weizenbaum en el MIT, funcionaba mediante la identificación de palabras clave (*keywords*) en el texto de entrada y la aplicación de reglas de descomposición (*decomposition rules*) asociadas, que posteriormente se reconstruían mediante reglas de reensamblaje (*reassembly rules*). El sistema mantenía un script de datos (no integrado en el código) que contenía conjuntos de palabras clave, reglas de transformación y sustituciones incondicionales de pronombres, permitiendo generar respuestas aparentemente coherentes sin capacidad real de comprensión semántica [31].

Por otra parte, SHRDLU desarrollado por Terry Winograd se fundamentó en la premisa de que un sistema no puede procesar el lenguaje razonablemente sin “comprender” el tema en discusión. A diferencia de ELIZA, SHRDLU integraba sintaxis, semántica e inferencia en un sistema unificado o “vertical”, utilizando un modelo detallado de un micromundo (un robot manipulando objetos virtuales) para dar contexto al discurso. Esto permitía al sistema no solo analizar la estructura gramatical mediante Gramática Sistémica, sino representar el conocimiento en forma de procedimientos ejecutables, dotándolo de la capacidad para desambiguar oraciones, deducir hechos y dialogar sobre sus propios planes y acciones dentro de ese dominio restringido [33].

1980-2000: Modelos estadísticos y aprendizaje supervisado.

A principios de los ochenta, el estudio del procesamiento del lenguaje natural se centraba predominantemente en metodologías simbólicas, conocidas como sistemas expertos, caracterizados por reglas y ontologías establecidas manualmente que actuaban como repositorios estructurados de conocimiento específico de un dominio [23].

Pero a mediados-finales de los 80, las deficiencias de los modelos simbólicos, junto con su incapacidad para escalar y manejar ambigüedades, comenzaron a ser manifiestas. Este escenario marcó el inicio de la transición hacia enfoques estadísticos, con la premisa de emplear datos para inferir patrones en lugar de codificar explícitamente todas las reglas del lenguaje [23].

La materialización más temprana de este giro estadístico en el ámbito del procesamiento del lenguaje natural fue la adopción de los **Modelos Ocultos de Márkov** (HMM, por sus siglas en inglés). Un HMM es un proceso doblemente estocástico en el que se distinguen dos niveles: una cadena de estados latentes $\{q_t\}$ (no directamente observables) que evoluciona conforme a una distribución de probabilidad de transición, y una secuencia de observaciones $\{o_t\}$ generada por dichos estados según una distribución de emisión condicional [34]. La tarea de inferencia consiste, precisamente, en estimar la secuencia de estados más probable dadas las observaciones. Esta capacidad de modelar secuencias con dependencias ocultas resultó especialmente adecuada para tareas como el reconocimiento automático del habla (dominio en el que alcanzó su mayor madurez) y, de forma progresiva, para tareas centrales del PLN como el etiquetado morfosintáctico (*part-of-speech tagging*) o el reconocimiento de entidades nombradas. No obstante, la efectividad de estos modelos estadísticos dependía directamente de la disponibilidad de grandes volúmenes de datos anotados, lo que impulsó la construcción sistemática de corpus lingüísticos de referencia: WordNet (1985) [35], Penn

⁴Se debe matizar que en el caso ELIZA, tal y como argumenta Shrager [32], su objetivo no era desarrollar un chatbot, sino una plataforma de experimentación de la interacción humana con las IA.

⁵Accesible en Internet Archive: <https://archive.org/details/eliza.qb64>

TreeBank (1989) [36], [37] y el British National Corpus (1991) [38], que proporcionaron millones de palabras etiquetadas léxico-semántica, sintáctica y contextualmente [23], [24].

En la década de los noventa, el desarrollo de los corpus lingüísticos permitió la aparición de los primeros modelos de lenguaje estadísticos basados en el concepto de n-grama, consistentes en la predicción de la palabra i a través de las $n-i$ palabras previas, adoptando la forma de bigramas y trigramas, facilitando así las aplicaciones tempranas como el reconocimiento de voz y la traducción automática estadística. [23], [24].

2000-2014: Revolución neuronal, vectores de palabras y la arquitectura codificador-decodificador

Durante este periodo, con el trabajo fundacional de Bengio *et al.* [39], se introduce un modelado del lenguaje basado en redes neuronales, que se descompone en dos componentes fundamentales:

1. Una tabla de asignación de vectores de palabras (*lookup table C*) que mapea cada palabra a un vector de características distribuidas en $\mathbb{R}^m$ (por ejemplo, $m = 30, 60$ o 100 dimensiones), representando la semántica y similitud entre palabras.
2. Una función de red neuronal (g), que opera sobre estos vectores para estimar la probabilidad de la siguiente palabra en la secuencia.

Este modelo corrige dos limitaciones críticas de los modelos de n-gramas previos: primero, la falta de contexto más allá de una o dos palabras y segundo, la falta de un mecanismo para representar la similitud semántica entre palabras. Se consolida así el empleo de las representaciones vectoriales densas (*word embeddings*) en el procesamiento natural del lenguaje.

A finales de los 2000, Collobert y Weston [40] introdujeron una arquitectura unificada (Figura 2.3) basada en redes neuronales convolucionales profundas, con el objetivo de compartir representaciones léxicas entre múltiples tareas de PLN mediante aprendizaje multitarea. Estructurada en varias capas entrenadas de extremo a extremo mediante retropropagación, el modelo partía de una **capa de embeddings** (*lookup table*) que transformaba los índices del vocabulario en vectores densos compartidos entre tareas como el etiquetado de categorías gramaticales (*POS tagging*), *chunking*, reconocimiento de entidades nombradas, etiquetado de roles semánticos, similitud semántica y modelado del lenguaje. A continuación, una **capa de convolución** (también denominada *Time-Delay Neural Network* o TDNN) capturaba patrones locales procesando todas las ventanas de palabras de la oración de forma simultánea; una **capa de agrupamiento máximo** (*max-over-time pooling*) seleccionaba las características más relevantes a lo largo de la secuencia, generando un vector de dimensión fija independiente de la longitud de la entrada; y, finalmente, **capas totalmente conectadas** con salida **softmax** producían una distribución de probabilidad sobre las clases de cada tarea. El entrenamiento se formulaba como un objetivo discriminativo en el que el sistema debía asignar una puntuación mayor a secuencias reales que a secuencias corrompidas mediante la sustitución aleatoria de la palabra central, lo que posibilitaba el preentrenamiento de los *embeddings* sobre grandes corpus no etiquetados sin necesidad de estimar una distribución de probabilidad normalizada [40]. El aspecto fundamental del diseño residía precisamente en el **uso de la lookup table como capa compartida** entre todas las tareas, lo que forzaba al modelo a aprender representaciones de palabras más ricas y generales, consolidando el aprendizaje multitarea como estrategia de regularización implícita.

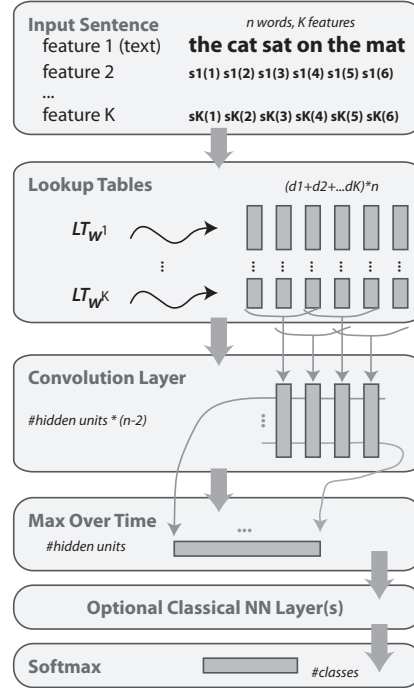


Figura 2.3: Arquitectura de Collobert y Weston (2008) con capacidad multitarea. Conformada por las capas de *lookup table*, convoluciones TDNN, *max-pooling*, redes neuronales clásicas y salida *softmax*. Fuente: [40].

Paralelamente, Mikolov *et al.* [41] introdujeron un modelo de lenguaje basado en redes neuronales recurrentes simples (*Simple Recurrent Networks*, SRN) denominado RNN-LM⁶ (Figura 2.4) que extiende la idea de Bengio *et al.* [39] al caso de contextos de longitud no acotada, evitando la restricción principal del modelo de Bengio *et al.* [39] cuya ventana de contexto se veía limitada en la práctica a las cinco o diez palabras predecesoras. En este enfoque, la probabilidad de la siguiente palabra se estima con una red neuronal cuya entrada en el instante t está formada por la concatenación del vector 1-de- N (*one-hot*) de la palabra actual y del estado del contexto previo $s(t-1)$, de modo que la capa oculta $s(t)$ actúa como una memoria recurrente que resume todo el historial previo y se retroalimenta en cada paso temporal. La salida $y(t)$ es una distribución softmax sobre el vocabulario que representa la probabilidad de la siguiente palabra dada la palabra anterior $w(t)$, y el estado de contexto $s(t-1)$, es decir, $P(w_{t+1}|w(t), s(t-1))$ [41].

La introducción de este modelo presentó dos aspectos distintivos principales:

- El modelado dinámico del contexto, donde la memoria a largo plazo no reside en la activación de las unidades de contexto (que cambian muy rápidamente de un paso temporal a otro), sino en las **sinapsis mismas** (los pesos de la red). A diferencia de los modelos basados en ventanas fijas, en la fase de test el estado recurrente permite incorporar información procedente de un historial arbitrariamente largo de palabras, permitiendo que el modelo se adapte automáticamente al dominio específico sin necesidad de reentrenamiento [41].
- Simplificación de la parametrización del contexto. A diferencia de los modelos de Bengio *et al.* [39], que requieren la especificación *ad hoc* de múltiples hiperparámetros (tamaño de la capa de proyección de palabras, tamaño de la capa oculta y longitud

⁶Recurrent neural network language model: modelo de lenguaje basado en redes neuronales recurrentes.

del contexto), el RNN-LM simplifica notablemente esta configuración al modelar el contexto de forma implícita mediante el estado recurrente, siendo principalmente necesario especificar el tamaño de la **capa oculta (contexto)** antes del entrenamiento [41].

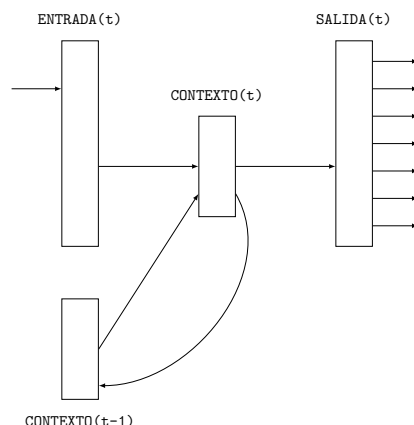


Figura 2.4: Diagrama conceptual del funcionamiento de la RNN-LM de Mikolov *et al.* (2010) [41]

Para mitigar las limitaciones de los RNN simples en el modelado de secuencias con dependencias de largo alcance (problema del desvanecimiento del gradiente), Graves [42] propuso una arquitectura de red neuronal profunda basada en LSTM (*Long Short Term Memory*) (Figura 2.5)[43], que permite la generación de secuencias mediante predicción paso a paso. Esta arquitectura se diferencia de los enfoques anteriores en que introduce una celda de memoria para mantener la coherencia estructural a través de largos períodos temporales, lo que resulta especialmente crítico cuando la red debe generar información basada en su propio historial de predicciones sin acceso a contexto futuro. Dicha arquitectura utiliza múltiples capas de LSTM (Figura 2.5) apiladas en profundidad espacio-temporal, donde cada capa recurrentemente conectada procesa la salida de la capa anterior. El núcleo de esta arquitectura es la celda de memoria LSTM, que incorpora tres mecanismos de puerta multiplicativos: la puerta de entrada (*input gate*), la puerta de olvido (*forget gate*) y la puerta de salida (*output gate*)[43]. La puerta de entrada controla qué información nueva se almacena en la celda de memoria (contexto), la puerta de olvido regula qué información previamente almacenada debe descartarse, y la puerta de salida decide cuándo exponer el contenido almacenado al resto de la red [43]. Estas puertas actúan mediante funciones sigmoideas que producen valores entre 0 y 1, modulando así el flujo de información de manera diferenciable.

En paralelo a estos avances en modelos recurrentes con memoria mejorada, Mikolov *et al.* [44], [45] presentaron en 2013 Word2Vec⁷, un enfoque que optimizó significativamente el aprendizaje de representaciones vectoriales de palabras. La principal aportación de este método consistió en un cambio conceptual: en lugar de obtener los vectores de palabras como un subproducto del modelado del lenguaje, el objetivo del entrenamiento pasa a ser aprender directamente representaciones vectoriales de palabras. Este método introdujo dos modelos, *continuous bag of words* (CBOW) y Skip-gram (Figura 2.6), ambos basados en arquitecturas log-lineales que eliminan la capa oculta no lineal presente en modelos de redes neuronales anteriores, reduciendo significativamente el costo computacional.

⁷ Accesible en <https://code.google.com/archive/p/word2vec/>

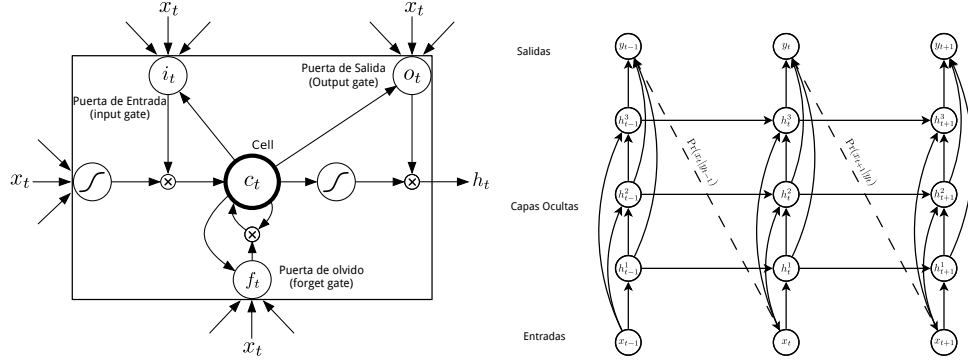


Figura 2.5: Celda LSTM (izquierda) y arquitectura RNN-LSTM (derecha). Traducidos de [42].

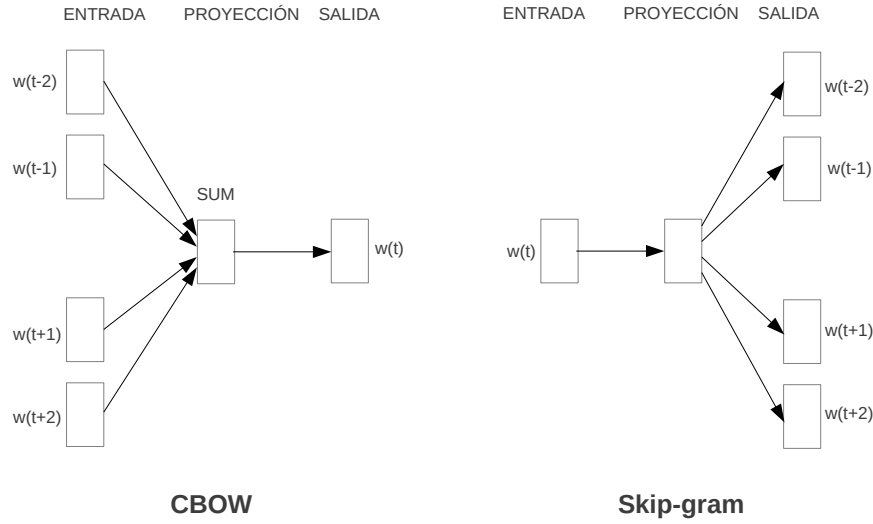


Figura 2.6: Word2Vec (Mikolov 2013): CBOW (contexto $\rightarrow$ palabra) vs. Skip-gram (palabra $\rightarrow$ contexto). Traducido de [44], [45].

El primero, CBOW(Figura 2.6 [a]), predice la palabra actual basándose en el contexto circundante. Las palabras del contexto (tanto palabras anteriores como posteriores, a diferencia de los modelos n-gramas que emplean solamente el contexto previo) se proyectan en la capa de *embeddings* compartida donde sus vectores se combinan mediante el promedio o la suma, (perdiendo así la información sobre el orden de las palabras)(Figura 2.6 [izq.]). La complejidad computacional de entrenamiento de CBOW es $Q = N \times D + D \times \log_2(V)$, donde N es el tamaño del contexto, D es la dimensionalidad de los vectores de palabras, y V es el tamaño del vocabulario [44].

El segundo modelo, Skip-gram, invierte el objetivo de CBOW: en lugar de predecir la palabra actual dado su contexto, busca predecir palabras contextuales dada una palabra de entrada(Figura 2.6 [dcha.]). Para cada palabra actual, el modelo utiliza la capa de proyección continua para estimar la probabilidad de las palabras situadas dentro de una ventana de contexto antes y después de dicha palabra. La complejidad computacional es $Q = C \times (D + D \times \log_2(V))$, donde C es la distancia máxima de palabras consideradas en el contexto.

Finalmente, en 2014 surge la arquitectura **codificador-decodificador** (*encoder-decoder*) como solución al procesamiento de secuencias de longitud variable. Cho *et al.* [46], Sutskever *et al.* [47] y Bahdanau *et al.* [48] proponen modelos que transforman secuencias fuente en representaciones vectoriales (ya sea un vector de contexto fijo o dinámico mediante atención), las cuales son decodificadas autorregresivamente para generar las secuencias objetivo.

Comenzando con la arquitectura **RNN encoder-decoder** (Figura 2.7), Cho *et al.* [46] sientan las bases de este paradigma, con redes neuronales recurrentes actuando como par codificador-decodificador. El codificador mapea una secuencia de longitud variable a un vector de contexto de dimensión fija, mientras que el decodificador mapea este vector de contexto de vuelta a una secuencia objetivo de longitud variable. Ambas redes se entrenan conjuntamente maximizando la probabilidad condicional de la secuencia objetivo dada la secuencia fuente, ya que este modelo fue propuesto originalmente en el contexto de la traducción automática, donde la secuencia fuente corresponde a una oración en el idioma de origen y la secuencia objetivo a su traducción en el idioma destino. El aspecto distintivo es la introducción de unidades recurrentes con mecanismos adaptativos, conocidas como *Gated Recurrent Units (GRU)*. Estas unidades incorporan dos puertas adaptativas (*reset gate* y *update gate*), que de forma similar a las celdas LSTM, permiten mitigar el problema del desvanecimiento del gradiente y mejoran significativamente la capacidad del modelo para capturar dependencias a largo plazo.

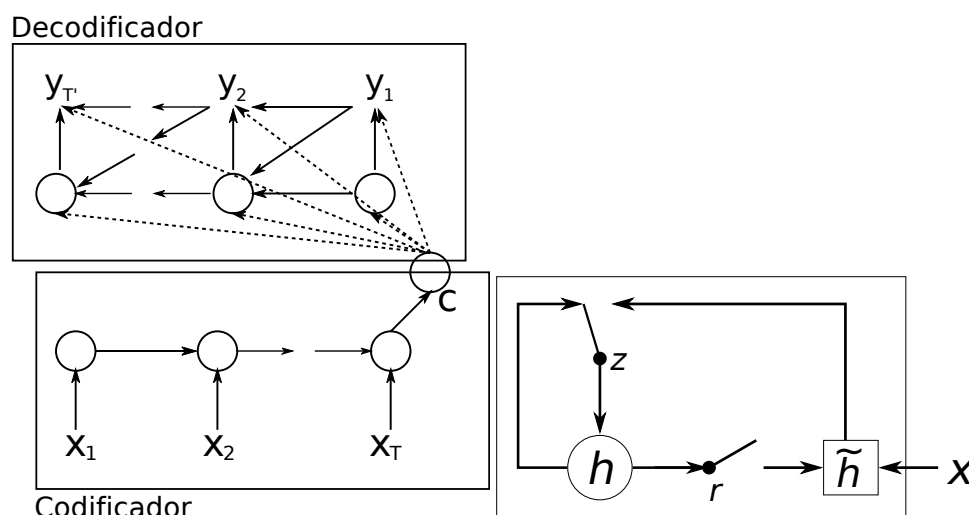


Figura 2.7: Arquitectura RNN codificador-decodificador (izquierda) y diagrama de las puertas adaptativas *reset/update* (derecha) propuesta por [46].

Por otro lado, el modelo *Sequence-2-Sequence con LSTM profundas* (Figura 2.8) propuesto por Sutskever *et al.* [47], emplea dos LSTM profundas (de cuatro capas cada una). De forma similar al modelo anterior, la primera LSTM (codificador) procesa la secuencia fuente palabra por palabra, condensando toda la información en un vector contexto de dimensión fija, que sirve como condición inicial para la segunda LSTM (decodificador). Esta última LSTM funciona como un modelo recurrente, generando la secuencia de salida de forma autorregresiva, es decir una palabra a la vez, en función del vector contexto y las palabras generadas previamente. Un hallazgo empírico que encontraron fue que la inversión del orden en la secuencia fuente mejora dramáticamente el rendimiento del modelo al reducir la distancia temporal entre las dependencias relevantes en las secuencias de entrada y

salida de corto plazo.

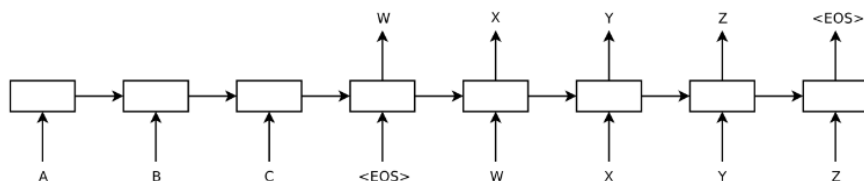


Figura 2.8: Seq2Seq LSTM de Sutskever *et al.* (2014): *encoder* de 4 capas, *decoder* autorregresivo con inversión de fuente. Fuente: [47]

Sin embargo, el uso de un único vector de contexto de dimensión fija para representar toda la secuencia de entrada constituye una limitación importante, especialmente en secuencias largas, ya que obliga al modelo a comprimir toda la información relevante en una única representación. El modelo con mecanismo de atención propuesto por Bahdanau *et al.* [48] elimina la limitación del vector de contexto de dimensión fija presente en los modelos *encoder-decoder* anteriores. En este enfoque, el decodificador usa un mecanismo de atención que le permite calcular, en cada paso temporal, una ponderación sobre los distintos estados ocultos producidos por el codificador. De esta forma puede prestar atención dinámicamente a diferentes partes de la secuencia de entrada durante cada paso del proceso de generación de la secuencia de salida. Formalmente, el mecanismo de atención computa pesos (“*scores*”) sobre los estados ocultos del codificador mediante una función de alineación aprendida. Estos pesos se normalizan para obtener una distribución de atención, a partir de la cual se construye un vector de contexto variable para cada paso de generación. Este proceso, conocido como *alineación suave* (*soft attention*), se aprende conjuntamente durante el entrenamiento sin supervisión explícita. Como resultado, el modelo mejora sustancialmente en el manejo de dependencias a largo plazo. Los años 2015

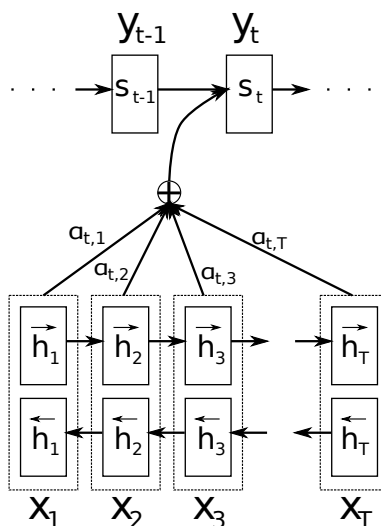


Figura 2.9: Funcionamiento de la atención de Bahdanau *et al.* (2014); las x representan las entradas del codificador, las h los estados ocultos del decodificador, las a los pesos de la atención, y las s los estados ocultos del decodificador. Fuente: [48].

y 2016 consolidaron este paradigma mediante tres contribuciones que prepararían directamente la arquitectura Transformer. En primer lugar, Luong *et al.* [49] simplificaron el modelo de atención de Bahdanau *et al.* [48] mediante dos familias (atención global y

local) y tres funciones de puntuación alternativas (*dot*, *general*, *concat*), cuya mayor eficiencia computacional anticipaba el mecanismo de *scaled dot-product attention*. En segundo lugar, He *et al.* [50] introdujeron las **conexiones residuales** (*skip connections*), que permiten a la señal fluir directamente entre capas no contiguas según la formulación $\mathcal{F}(\mathbf{x}) + \mathbf{x}$, resolviendo el desvanecimiento del gradiente en redes de gran profundidad y convirtiéndose en un componente estructural del Transformer. Finalmente, Ba *et al.* [51] propusieron la **normalización por capas** (*Layer Normalization*), que opera independientemente por muestra sin depender del tamaño del lote, lo que la hace adecuada para secuencias de longitud variable y la convertiría en el mecanismo de normalización estándar de los modelos de lenguaje posteriores.

2017-2019: Transformers y preentrenamiento masivos

En 2017, Vaswani *et al.* introdujeron el Transformer [2], un modelo basado completamente en mecanismos de atención descritos por Bahdanau *et al.* [48] y que prescinde completamente de recurrencia y convoluciones. La arquitectura utiliza capas de autoatención (*self-attention*) y redes neuronales con conexiones residuales, permitiendo mayor paralelización durante el entrenamiento comparado con modelos recurrentes [23], [24]. En cuanto al procesamiento de datos de entrada, a diferencia de los modelos secuenciales previos, el Transformer permite procesar todas las posiciones de una secuencia de forma simultánea. Este alto grado de paralelización redujo significativamente los tiempos de entrenamiento sin comprometer la calidad de los resultados en tareas de traducción automática y otras aplicaciones de procesamiento del lenguaje natural [2].

A partir de 2018 comenzaron a aparecer modelos preentrenados⁸ basados en Transformers, entre los que destacan BERT[52] y GPT-1. Ambos introdujeron el paradigma de preentrenamiento en grandes corpus de texto no etiquetado mediante objetivos autosupervisados seguido de ajuste fino (*fine-tuning*) para tareas específicas. Este enfoque demostró que el aprendizaje masivo sobre texto no etiquetado mejora significativamente el rendimiento en múltiples tareas de PLN [23], [24].

Este paradigma se consolida a finales de la década con T5 (*Text-to-Text Transfer Transformer*) [53], propone un marco unificado en el que todas las tareas de PLN, sentando las bases de lo que sería un sistema aglutinante de propósito general (Figura 2.10) y explora de forma sistemática los límites del aprendizaje por transferencia. T5 combina un preentrenamiento de tipo *denoising*⁹ sobre el corpus C4¹⁰, una arquitectura *encoder-decoder* basada en Transformers y un escalado del modelo hasta los 11 mil millones de parámetros. Este modelo marca el cierre de la etapa inicial de los Transformers preentrenados caracterizada por el aprendizaje por transferencia a gran escala.

⁸El concepto de preentrenamiento (*pretraining*) se emplea conjuntamente con el de aprendizaje por transferencia, donde los modelos son preentrenados en *datasets* masivos sin etiquetar para posteriormente ser adaptados a tareas específicas. A este tipo de modelos se les denomina *Pretrained Language Models* (modelos de lenguaje preentrenados) o *PTM* para abreviar [24].

⁹El *denoising* o reconstrucción, es un tipo de tarea de preentrenamiento donde partiendo de una frase corrupta artificialmente, el objetivo es reconstruir la versión limpia.

¹⁰C4 (*Colossal Clean Crawled Corpus*) es un corpus masivo de texto en inglés construido a partir de una instantánea (abril de 2019) de Common Crawl y sometido a diversos procesos de limpieza, filtrado (improperios, código, *lorem ipsums* ...) y eliminación de duplicados. Fue creado específicamente para el preentrenamiento de modelos T5. [53]

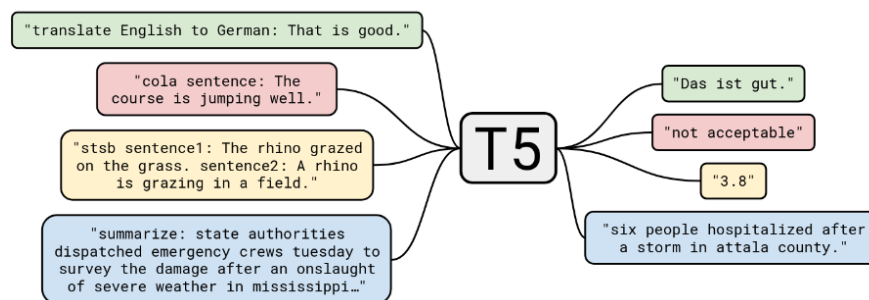


Figura 2.10: Reformulación de múltiples tareas de PLN a un formato estándar de texto a texto procesado por el modelo T5. Fuente: [53]

2020-presente: era de los grandes modelos de lenguaje (LLM)

La década de 2020 marca una transición desde modelos principalmente diseñados para tareas específicas hacia los LLM de propósito general, consolidada por el éxito de ChatGPT y superando la era de modelos especializados como T5. Este fenómeno ha generado una repercusión significativa en el campo del procesamiento del lenguaje natural analizadas aquí desde cuatro perspectivas: 1) el desarrollo de modelos frontera en la nube; 2) la evolución de las arquitecturas y técnicas de entrenamiento; 3) la democratización a través de iniciativas *open-source* y la inferencia local; y 4) la aparición de nuevos desafíos en soberanía tecnológica, ética y regulación de estos sistemas.

Desarrollo de modelos frontera en la nube

El panorama de los modelos frontera durante la década de 2020 ha estado marcado por una fuerte competencia entre laboratorios de investigación y corporaciones tecnológicas para liderar la capacidad de cómputo y las capacidades emergentes. Este ciclo comenzó con la liberación de GPT-3 en mayo de 2020, el cual demostró que el simple escalado de parámetros hasta los 175 mil millones permitía realizar tareas para las que el modelo no había sido entrenado explícitamente [5]. Sin embargo, la verdadera disrupción social y técnica se produjo en noviembre de 2022 con el lanzamiento de ChatGPT por parte de OpenAI¹¹, que alcanzó los 100 millones de usuarios en tan solo dos meses [3]. Este hito estableció la visión de los Modelos como Servicio (MaaS), impulsada por técnicas de alineamiento¹² como el Aprendizaje por Refuerzo a partir de la Retroalimentación Humana (RLHF), que permitió que los modelos fueran más útiles, honestos y seguros [55].

A medida que avanzaba la década, el ecosistema se diversificó con la aparición de competidores de alto nivel. Anthropic, fundada con un enfoque en la seguridad de la IA, introdujo la familia Claude 2 [56] al público general en julio de 2023, destacando por su técnica de IA Constitucional [57], [58], que consiste en alinear el comportamiento del modelo a través de un conjunto explícito de principios o valores, reduciendo la dependencia de la supervisión humana directa, y por su capacidad para manejar ventanas de contexto cada vez más amplias [56]. Por su parte, Google presentó la serie Gemini 1.5 [59] en mayo de 2024, integrando capacidades multimodales nativas que permiten

¹¹<https://openai.com/es-ES/index/chatgpt/>

¹²Se puede definir el alineamiento (*alignment*) como el conjunto de técnicas orientadas a ajustar el comportamiento del modelo para que sus respuestas sean útiles, honestas e inofensivas, en consonancia con los valores e intenciones humanas [54].

procesar texto, imagen, audio y vídeo de forma simultánea, y alcanzando hitos de contexto de hasta 1 millón de *tokens* en sus versiones Pro [59]. En julio de 2024 Meta lanzó la familia Llama3 [60] de modelos multimodales de pesos abiertos con una arquitectura refinada de 405 mil millones de parámetros en su versión más potente, consolidándose como el estándar de la industria para el ecosistema de código abierto ¹³ (*open-weights*) y convirtiéndose así en un referente en la investigación académica.

Es a inicios de 2025, con la irrupción de la empresa china DeepSeek, cuando el mercado experimenta un cambio de paradigma. Sus modelos DeepSeek-V3 (arquitectura MoE de 671 mil millones de parámetros) [62] y, especialmente, la variante de razonamiento DeepSeek-R1 [63], lograron alcanzar un rendimiento comparable a los modelos más avanzados de OpenAI (o1) y Anthropic, pero con una fracción del coste de entrenamiento (estimado en menos de 6 millones de dólares frente a los cientos de millones de sus competidores occidentales) [62].

Tabla 2.1: Evolución de modelos frontera

Modelo	Alineamiento / Entrenamiento	Arquitectura / Características	Impacto
GPT-3 (2020)	Preentrenamiento autoregresivo	Transformer (175B params)	Capacidades Emergentes.
ChatGPT (2022)	RLHF (PPO)	MaaS	Adopción masiva.
Claude 2 (2023)	IA Constitucional	Optimización para contexto amplio	Mejora en seguridad y ética.
Gemini 1.5 (2024)	RLHF Multimodal	MoE Nativa	Ventana de 1M tokens.
Llama 3 (2024)	SFT / DPO	<i>Open-weights</i>	Estándar académico.
DeepSeek (2025)	RL Puro	Razonamiento	Eficiencia de costes.

Avances en arquitectura y razonamiento

El desarrollo técnico ha evolucionado desde el paradigma inicial de “más es mejor” (*Bigger is Better*) [5] referidas al número de parámetros hacia arquitecturas optimizadas para el razonamiento y la eficiencia. Las Leyes de Escala (Scaling Laws) propuestas en 2020 por Kaplan *et al.* [1] confirmaron que el rendimiento mejora de forma predecible con el aumento de parámetros, datos y cómputo. No obstante, el escalado de modelos hasta cientos de miles de millones de parámetros ha motivado la adopción de arquitecturas más eficientes, como *Mixture-of-Experts*¹⁴ (MoE). El modelo MoE se compone de múltiples subredes especializadas (*experts*), de las cuales solo una fracción se activa para cada entrada, lo que permite mantener una alta capacidad representacional reduciendo el coste computacional efectivo durante la inferencia. En paralelo, un avance fundamental ha sido una nueva generación de modelos optimizados para el razonamiento complejo, como OpenAI o1 [64] y DeepSeek-R1 [63], que utilizan técnicas de Cadena de Pensamiento ¹⁵ (*Chain-of-Thought*) [65] durante la inferencia para resolver problemas matemáticos y lógicos complejos que superan las capacidades de los modelos autorregresivos estándar. Por otra parte, la integración de agentes de IA permite que estos modelos no solo generen texto, sino que razonen sobre subtarefas y utilicen herramientas externas de forma autónoma, presentando así un enfoque híbrido que combina modelos de lenguaje con mecanismos de control y acceso a recursos externos.

¹³Al margen de los matices entre código abierto (*open-source*) y *software* libre, la “apertura” puede aplicarse de forma parcial a componentes concretos del modelo (pesos, arquitectura, datos de entrenamiento, etc.) o de forma integral a la totalidad de ellos; solo en este último caso puede hablarse de un modelo verdaderamente *open-source* [61].

¹⁴Véase la Sección 2.3.3 para más detalles.

¹⁵Véase la Sección 2.3.2 para más detalles.

Democratización de acceso a través del *Open Source* y la inferencia local

La democratización de la Inteligencia Artificial se ha acelerado notablemente gracias a la convergencia de modelos de pesos abiertos (*open-weights*) y herramientas de inferencia local. Proyectos pioneros como llama.cpp han permitido la ejecución de modelos avanzados en *hardware* de consumo (incluyendo CPUs y GPUs domésticas) mediante técnicas de cuantización (ver Sección 2.4.2). Estas técnicas reducen la precisión de los pesos (por ejemplo, a 4 bits) para minimizar el uso de memoria RAM sin incurrir en una degradación crítica del rendimiento [66]. Paralelamente, herramientas como Ollama [67] permiten el despliegue de modelos en entornos locales de forma casi instantánea [66].

Esta tendencia ha impulsado el auge de los Modelos de Lenguaje Pequeños (SLM). A diferencia de sus homólogos masivos, los SLM cuentan con un número reducido de parámetros, generalmente entre 1 y 7 miles de millones (*billions*), destacando familias como Phi (Microsoft), Gemma (Google), Llama (Meta), Salamandra (Barcelona Supercomputing Center, BSC) o Mistral (Mistral AI) [14]; estos modelos permiten el funcionamiento con una menor capacidad de cómputo y huella en memoria. Es precisamente esta eficiencia, sumada a la disponibilidad, y optimización del *software* de inferencia local, lo que permite el despliegue en entornos locales (*on-premises*). Entre sus ventajas, destacan:

- **Democratización y accesibilidad de adopción:** Reduce las barreras de entrada económicas al eliminar la dependencia de infraestructuras en la nube costosas o *hardware* de grado empresarial, pudiéndose desplegar en dispositivos *edge* o de bajo coste [14].
- **Personalización:** La naturaleza abierta de estos modelos facilita el ajuste fino (*fine-tuning*) mediante técnicas como LoRA (*Low Rank Adaptation*, véase la Sección 2.6.1). Esto permite adaptar el modelo a dominios científicos específicos o lenguajes técnicos sin necesidad de reentrenamientos masivos, pudiendo resultar en un mayor rendimiento y precisión [14].
- **Control y privacidad de los datos y despliegue en áreas sin conexión:** El procesamiento local garantiza la confidencialidad de datos sensibles o sujetos a propiedad intelectual. Además, permite el despliegue en entornos críticos sin conexión a internet (*air-gapped*), asegurando la continuidad operativa y el cumplimiento de normativas como el RGPD [13], [14].
- **Reducción de la latencia:** al evitar la comunicación con servidores remotos, la inferencia local puede disminuir significativamente los tiempos de respuesta, lo que resulta crítico en aplicaciones de tiempo real.
- **Reproducibilidad científica:** El acceso a modelos abiertos y su ejecución en entornos locales ejecutables permite la auditoria de resultados y replicabilidad de experimentos, garantizando más transparencia en la investigación.

Desafíos en soberanía, ética y regulación

El desarrollo de la inteligencia artificial (IA) se ha consolidado como un factor tecnoestratégico que define la autonomía política y económica en el escenario global. En la actualidad, existe una marcada asimetría en la producción de modelos de frontera: durante 2024, instituciones con sede en Estados Unidos produjeron 40 modelos de IA notables, frente a los 15 desarrollados en China y únicamente 3 en el conjunto de la Unión

Europea [8]. Esta concentración de capacidades técnicas y financieras ha generado una dependencia estructural hacia ecosistemas tecnológicos dominantes, del oligopolio sino-estadounidense, lo que conlleva riesgos asociados a la transferencia de sesgos culturales y lingüísticos, censura [68] y reivindicaciones políticas [69] y dependencias regulatorio-legislativas externas.

Esto ha desembocado en el concepto de IA soberana, enmarcado dentro de la soberanía digital. Este concepto se refiere a la capacidad de una nación-estado para desarrollar tecnologías basadas en inteligencia artificial sobre infraestructura doméstica propia, utilizando datos y recursos humanos locales, sin interferencias ni dependencias de plataformas extranjeras. Esto incluye el ciclo de vida completo del desarrollo de la IA: desde la recolección de datos y el entrenamiento hasta el despliegue y la regulación de los sistemas [70].

En Europa, dentro de la fase del ciclo de vida del desarrollo (recolección de datos y entrenamiento), en España destaca el proyecto público ALIA, en colaboración con el Barcelona Supercomputing Center y financiado con fondos de la Unión Europea – NextGenerationEU. Este proyecto tiene como objetivo proporcionar una infraestructura pública de recursos de IA, ofreciendo tanto un corpus de datos multilingüe en las lenguas oficiales y cooficiales del Estado, como modelos (entre los que destacan Salamandra y ALIA [71]), herramientas de integración, demostradores y guías de adopción [9].

Por otra parte, incluyendo la fase de despliegue, en el ámbito industrial la compañía francesa Mistral AI, se ha consolidado como un actor relevante en el desarrollo de modelos abiertos con su familia de modelos Mixtral. Fundada en 2023, esta startup ha desarrollado una amplia red de colaboraciones con entidades gubernamentales a nivel global, posicionándose como referente europeo en IA soberana para el sector público. Entre sus iniciativas destacadas se encuentran el programa “AI for Citizens” lanzado en julio de 2025, orientado a gobiernos e instituciones públicas con principios de apertura, colaboración, autonomía tecnológica y control de datos [11]; un acuerdo marco con el Ministerio de las Fuerzas Armadas francesas en enero de 2026 para suministrar modelos y servicios a defensa, CEA, ONERA y SHOM sobre infraestructura soberana [72]; su integración en el asistente interministerial de IA en febrero de 2026; y alianzas como la firmada en noviembre de 2025 con Francia, Alemania, Mistral AI y SAP para una IA soberana en administración pública europea, enfocada en ERP inteligente, gestión financiera y agentes digitales. Estas colaboraciones se extienden a Luxemburgo (junio 2025) [73], Singapur (marzo 2025, para defensa) [74], Armenia (febrero 2025) [75] y la Comunidad de Madrid (diciembre 2025, para servicios públicos [76]) [11], [77], [78].

Desde el punto de vista regulatorio, destaca el Reglamento (UE) 2024/1689 (Ley de IA) [12], cuya implementación alcanza hitos críticos en 2026. Este marco pionero establece un enfoque basado en el riesgo, prohibiendo prácticas inaceptables (como la puntuación social o la identificación biométrica en tiempo real [12, Art. 30]) desde febrero de 2025, y haciendo obligatorios los requisitos para sistemas de “alto riesgo” a partir de agosto de 2026 [12, Art. 179]. El reglamento no solo busca mitigar riesgos de seguridad o de vulneraciones de derechos [12, Arts. 27 a 29], sino que impone obligaciones de transparencia a los modelos de propósito general (incluyendo regulación adicionales para aquellos cuyo entrenamiento supere las 10^{25} operaciones en punto flotante, clasificándolos como de alto impacto [12, Arts. 110 a 115]), exigiendo que respeten la legislación europea de derechos de autor y detallen los datos utilizados en su entrenamiento [12, Arts. 104 a 109]. De igual forma que el RGPD, cuenta con un marcado ámbito de extraterritorialidad, al no solo aplicarse dentro de las empresas situadas o que operen en la Unión Europea sino a

cualquier empresa que actúe sobre ciudadanos de la Unión [12, Arts.21 a 24].

Fuera del marco europeo, en Estados Unidos la regulación está emergiendo a nivel estatal con enfoques sectoriales específicos. Destaca la ley de Colorado, “*Consumer Protections for Artificial Intelligence*” (SB24-205) [79], cuya entrada en vigor el 30 de enero de 2026 marca un hito al imponer un ‘deber de cuidado’ (*duty of care*) para evitar sesgos algorítmicos. Esta norma pone especial énfasis en la protección de los usuarios frente a la discriminación algorítmica en decisiones de alto impacto, como el empleo, la vivienda o los servicios financieros. Por otro lado, la “*California AI Transparency Act*” (SB 942) [80] se centra en la integridad del ecosistema digital, obligando a los proveedores de sistemas de IA generativa de gran escala a implementar marcas de agua y sistemas de detección para que los usuarios puedan identificar contenido sintético, combatiendo así la desinformación y el fraude de identidad.

Por último, un caso peculiar de regulación es el de China, donde las Medidas Interinas para la Gestión de Servicios de IA Generativa (CAC, 10/07/2023)[81], [82, Arts. 4 y 7 a 9] en conjunto con la enmienda de Ley de Ciberseguridad [83], [84], exigen alineación con los valores del PCC, generando censura y propaganda [68], [85], [86] al prohibir contenido subversivo con registro obligatorio de algoritmos. Como consecuencia, en 2025 diversos gobiernos como el de Taiwán [87] han tomado medidas contra la compañía china DeepSeek limitando o directamente prohibiendo su uso [88].

Finalmente, cabe señalar la emergencia de diversos movimientos sociales y colectivos profesionales que cuestionan los métodos de desarrollo de la IA generativa desde dos vertientes principales: el uso masivo de datos sin consentimiento previo (*web scraping*) y el impacto medioambiental asociado al ciclo de vida de los modelos.

En lo relativo a la dimensión medioambiental, la literatura académica ha documentado el elevado consumo energético y las emisiones de gases de efecto invernadero derivados del entrenamiento de modelos de gran tamaño [89], a lo que se suma la demanda energética sostenida de su inferencia a escala de servicio en centros de datos [90]. Esta preocupación refuerza el interés por estrategias de optimización y despliegue eficiente orientadas a reducir la huella computacional de dichos sistemas.

En el ámbito organizativo, destacan colectivos como Arte es Ética [91]. En la vertiente jurídica, son relevantes las demandas colectivas (*class actions*) interpuestas contra entidades como Anthropic [92], [93] y OpenAI [94], basadas en la presunta vulneración de derechos de autor al emplear conjuntos de datos que contienen obras protegidas y repositorios de contenido no autorizado sin consentimiento de los autores. En esta misma línea, organizaciones internacionales de derechos humanos como Amnistía Internacional han señalado que el entrenamiento de estos sistemas a partir de *web scraping* a gran escala resulta incompatible con el derecho internacional de los derechos humanos, al sustentarse en una vulneración sistemática del derecho a la privacidad, hasta el punto de reclamar su prohibición [7].

En respuesta, han emergido herramientas técnicas orientadas a la protección de la propiedad intelectual frente a su uso no autorizado. Aunque los ejemplos más conocidos se sitúan en el dominio de la imagen (como Glaze [95] y Nightshade [96], que introducen modificaciones imperceptibles en los píxeles para dificultar su empleo en entrenamientos no consentidos), existen estrategias análogas en el ámbito textual y del rastreo web. Es el caso de Anubis [97], una utilidad que interpone desafíos a las conexiones entrantes con el fin de obstaculizar la recolección automatizada de datos destinada al entrenamiento de modelos.

2.2. El Transformer y la arquitectura de los LLM

Tal y como se menciona en la cronología de la Sección 2.1 en 2017 Vaswani *et al.* publican el artículo “*Attention is All you Need*” [2], en el que introducen la arquitectura Transformer, generalizando la idea de los mecanismos de auto-atención propuestos por Bahdanau *et al.* [48], prescindiendo completamente de la recurrencia y convolución, eliminando así las dos principales limitaciones intrínsecas a dichas arquitecturas.

La primera limitación corresponde con el procesamiento secuencial de las palabras durante el entrenamiento, limitación que restringe el paralelismo computacional aumentando los tiempos de entrenamiento. Como señalan los autores, “la naturaleza inherentemente secuencial de los modelos recurrentes impide la paralelización dentro de ejemplos de entrenamiento, que se vuelve crítica en secuencias más largas, ya que las restricciones de memoria limitan el agrupamiento entre ejemplos” [2].

La segunda limitación es el desvanecimiento del gradiente: la degradación de la información en las dependencias a largo alcance entre *tokens* en secuencias prolongadas. En los modelos recurrentes, la propagación a través de múltiples pasos temporales dificulta el aprendizaje de relaciones lejanas, limitación que el mecanismo de autoatención supera al conectar directamente cualquier par de posiciones de la secuencia.

Al reducir el camino máximo entre dependencias a $\mathcal{O}(1)$ operaciones secuenciales, en contraposición al $\mathcal{O}(n)$ propio de las RNNs, y permitir el procesamiento completamente paralelo, el Transformer se ha consolidado como la piedra angular de los modelos de lenguaje modernos [2].

2.2.1. Estructura del Transformer

El Transformer (Figura 2.11) en la versión original propuesta por Vaswani *et al.* [2] adopta una arquitectura modelo codificador-decodificador (del inglés *encoder-decoder*) basada en mecanismos de atención, donde ambos componentes desempeñan un rol específico en la transformación de secuencias de entrada en secuencias de salida. Antes de describir ambos bloques, es necesario introducir los tipos de atención presentes en la arquitectura.

La atención en el Transformer:

El núcleo de esta arquitectura es la atención escalada (*scaled dot-product attention*) y su extensión multicabeza (*multi-head attention*). Este mecanismo se formula en términos de vectores de consultas (Q), claves (K) y valores (V) obtenidos mediante proyecciones lineales de la entrada X :

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \quad (2.1)$$

donde W^Q, W^K, W^V son matrices de pesos entrenables¹⁶[2]. La atención se calcula como:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2.2)$$

¹⁶Algunas variaciones del Transformer incluyen un sesgo de proyección b por cada tipo de vector.

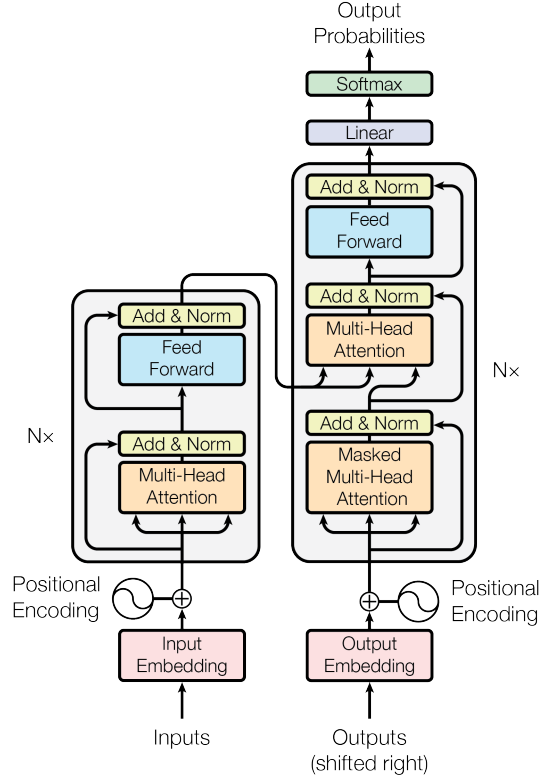


Figura 2.11: Arquitectura general del modelo Transformer. Fuente: [2]

donde la función *softmax* transforma las similitudes entre las consultas Q y las claves K en una distribución de probabilidad que actúa como pesos de atención sobre los valores V . El término $\sqrt{d_k}$, siendo d_k la dimensión de los vectores clave, opera como factor de escalado que estabiliza el entrenamiento al evitar que los productos escalares alcancen magnitudes que saturan la función *softmax*. Para capturar diferentes relaciones en paralelo, el Transformer emplea atención multicabeza, que concatena h proyecciones independientes:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O \quad (2.3)$$

donde W^O es la matriz de pesos de salida y cada cabeza se define como $\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$. Este diseño permite al modelo aprender de forma simultánea diferentes tipos de relaciones, ya sean sintácticas, semánticas o posicionales.

En la arquitectura Transformer se distinguen tres tipos principales de mecanismos de autoatención:

- **Autoatención (*self-attention*):** Donde los vectores Q, K, V provienen de la misma secuencia de entrada. Permite modelar las dependencias internas de la secuencia.
- **Atención enmascarada (*masked self attention*):** Variante de la autoatención que aplica una máscara para impedir que una posición atienda a *tokens* futuros, garantizando el carácter autorregresivo del modelo.
- **Atención cruzada (*cross-attention*):** Donde las consultas Q provienen de la secuencia generada en el decodificador, mientras que las claves K y los valores V provienen del codificador.

Codificador

El codificador transforma la secuencia de entrada en una representación contextual comprimida para el decodificador [98]. Su arquitectura se compone en N capas idénticas (originalmente $N = 6$ [2]) donde cada capa se descompone a su vez en 2 subcapas principales:

1. **Mecanismo de Atención Multicabeza (*Multihead Self-attention*)**: permite que el modelo atienda a todas las partes de la secuencia de entrada simultáneamente, capturando diversas relaciones y dependencias en paralelo.
2. **Red Neuronal *feed-forward* posicional (*Position-Wise Feed-Forward Neural Network*)**: es una red neuronal completamente conectada que se aplica de manera independiente a cada posición dentro de la secuencia de entrada. Esta subcapa está compuesta por dos transformaciones lineales separadas por una función de activación no lineal, específicamente la función de Unidad Lineal Rectificada (ReLU, por sus siglas en inglés), y se expresa matemáticamente como:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (2.4)$$

donde W_1, W_2, b_1, b_2 son parámetros entrenables del modelo. La primera capa lineal transforma la entrada x mediante la multiplicación por la matriz de pesos W_1 y la adición del sesgo b_1 . Seguidamente, se aplica la función de activación ReLU, que introduce no linealidad al sistema al establecer en cero todos los valores negativos y preservar los valores positivos. Finalmente, la segunda capa lineal proyecta la salida resultante a un espacio de dimensión adecuada mediante la multiplicación por W_2 y la adición del sesgo b_2 [2], [98].

Esta disposición modular permite que la red capture y modele relaciones complejas y no lineales entre las representaciones contextuales generadas por las capas de atención. Esta transformación se aplica de manera idéntica e independiente a cada *token* (*position-wise*). Además, la inclusión de la función ReLU facilita el entrenamiento de la red al mitigar problemas comunes en redes profundas, como el desvanecimiento del gradiente, ya que proporciona gradientes más estables durante la retropropagación [99].

Esta subcapa enriquece las representaciones aprendidas dentro de cada posición en la secuencia sin mezclar información entre *tokens* diferentes, manteniendo así la independencia posicional característica del diseño del Transformer. En el modelo Transformer original, la dimensión de entrada y salida es $d_{model} = 512$, mientras que la capa interna tiene una dimensionalidad de $d_{ff} = 2048$, lo que permite una expansión temporal del espacio de características antes de la proyección final [2].

Cada una de estas dos subcapas va seguida de un mecanismo de **suma residual y normalización**, tal como se representa en la Figura 2.11 y de manera análoga al decodificador. La suma (o conexión) residual, denominada x' , consiste en la conexión directa entre la entrada x de la subcapa y su salida. En otros términos, si una subcapa (como la de atención o la de retroalimentación) se representa como $Sublayer(x)$, la salida residual es la suma de la entrada y el resultado de la subcapa:

$$x' = x + Sublayer(x) \quad (2.5)$$

Estas conexiones facilitan la transmisión de la información de entrada a través de múltiples capas sin atenuación o distorsión significativa. Como resultado, y en conjunción con la capa ReLU, se mitiga el problema de desvanecimiento del gradiente y, consecuentemente, se favorece la convergencia de modelos más profundos [50], otorgando robustez y consistencia al flujo de información durante el entrenamiento [98].

Después de la suma residual, las representaciones se someten a una normalización por capas (*LayerNorm*), que se encarga de normalizar sus valores para asegurar que la media y la varianza de cada característica permanezcan dentro de rangos controlados. Este proceso normaliza cada elemento de la secuencia de forma independiente. Matemáticamente, para una representación $x = (x_1, x_2, \dots, x_H)$, la normalización se define como:

$$LayerNorm(x) = \gamma \left(\frac{x - \mu}{\sigma} \right) + \beta \quad (2.6)$$

donde $\mu = \frac{1}{H} \sum_{i=1}^H x_i$ es la media aritmética de todos los valores de la capa, $\sigma = \sqrt{\frac{1}{H} \sum_{i=1}^H (x_i - \mu)^2}$ es la desviación estándar y γ, β son parámetros entrenables que permiten escalar y trasladar la representación normalizada [98], [100].

Este procedimiento evita que las activaciones alcancen valores extremos que puedan comprometer la estabilidad numérica del modelo [51]. Su aplicación combinada con la suma residual da lugar a la siguiente expresión funcional para la salida de cada subcapa [98]:

$$Output = LayerNorm(x + Sublayer(x)) \quad (2.7)$$

Decodificador

El decodificador toma como entrada las representaciones contextuales producidas por el codificador y genera de forma autorregresiva la secuencia de salida, es decir, generando cada *token* en función de los *tokens* previamente generados y de la información de la secuencia de entrada [2], [98]. Al igual que el codificador, está formado por N capas idénticas (originalmente $N = 6$), pero cada capa del decodificador incorpora además un submódulo de atención sobre las salidas del codificador.

De forma análoga al codificador, las tres subcapas en las que se descompone el decodificador están seguidas por el mecanismo de suma residual y normalización:

1. **Atención Multicabeza Enmascarada (*Masked Multi-head Attention*)**: esta subcapa aplica el mismo mecanismo de atención multicabeza descrito para el codificador, pero restringido mediante una máscara que impide que la posición t atienda a posiciones futuras ($t' > t$) de la secuencia de salida [2]. De este modo se preserva la naturaleza autorregresiva del modelo, garantizando que la predicción de un *token* solo dependa de los *tokens* anteriores.
2. **Atención Multicabeza Codificador-Decodificador (*encoder-decoder Multi-Head Attention*)**: la segunda subcapa permite que el decodificador atienda explícitamente a las representaciones producidas por el codificador (lo que se denomina atención cruzada o *cross-attention*). En este caso, las consultas (Q) se derivan de las activaciones del decodificador, mientras que las claves (K) y los valores (V) proceden de las salidas del codificador [2].

3. **Red Neuronal *feed-forward* Posicional:** es idéntica a la red *feed-forward* posicional descrita en el codificador (ítem 2), aplicada de forma independiente a cada posición de la secuencia generada [2], [98].

2.2.2. Variaciones arquitectónicas del Transformer

A partir de la arquitectura básica codificador-decodificador, la práctica actual se ha materializado en tres familias de modelos Transformer, según el uso que se hace del codificador y del decodificador [23], [101]:

- **Codificador-Decodificador:** que mantienen la estructura completa original, orientada a tareas de transformación secuencia-a-secuencia, como la traducción automática (p. ej., T5) [98], [101].
- **Solo codificador:** emplean exclusivamente el bloque codificador para obtener representaciones contextuales bidireccionales (no enmascaradas) de la entrada. Se utilizan principalmente en tareas de comprensión como clasificación o etiquetado. Destaca el modelo BERT [52], [98], [101].
- **Solo decodificador:** se emplea únicamente el bloque decodificador, eliminando la atención cruzada al prescindir del codificador. Estos modelos utilizan autoatención enmascarada (causal) y están diseñados para la generación autorregresiva de secuencias. Destacan los modelos de la familia GPT [102] y la gran mayoría de modelos modernos [98], [101].

A partir de este punto, dada su relevancia y amplio uso y salvo mención explícita de lo contrario, se asume por defecto el empleo de la arquitectura “solo decodificador” en el análisis subsiguiente de este TFG.

2.2.3. Variaciones internas del Transformer

Más allá de las anteriormente citadas variaciones modales, se han propuesto y empleado múltiples variaciones internas del Transformer con el objetivo de mejorar la eficiencia computacional, capacidad de modelado o estabilidad [101]. En este apartado se describen brevemente algunas de las más relevantes.

Atención eficiente El mecanismo de autoatención estándar presenta una complejidad cuadrática $\mathcal{O}(T^2)$ con la longitud de la secuencia T , lo que limita su aplicación directa en contextos de gran longitud [101]. Para mitigar este problema se han propuesto diferentes estrategias:

- **Atención dispersa (*sparse attention*):** en contraposición a la atención densa, cada *token* atiende únicamente a un subconjunto estructurado de posiciones (dilatada, de banda, global, de bloque local o aleatoria), reduciendo así el número de productos escalares requeridos [101].
- **Atención lineal:** reformula el cálculo de la atención mediante mapas kernel aproximados para evitar la construcción explícita de la matriz $T \times T$, logrando complejidades efectivas cercanas a $\mathcal{O}(T)$ [101].
- **Optimización de la caché clave-valor:** en los modelos autorregresivos, las claves (K) y los valores (V) generados en pasos previos se almacenan para evitar

recomputaciones. De forma análoga a la paginación del sistema operativo, técnicas como *PagedAttention* (atención paginada) [103] almacenan dicha caché en bloques segmentados en memoria, permitiendo una recuperación más eficiente durante el cálculo de la atención [104].

- **Compresión de la caché KV:** reduce el número de posiciones efectivamente almacenadas. Métodos como el introducido por Wang *et al.* [105] se fundamentan en la observación de que una pequeña parte de los *tokens* (denominados *Heavy-Hitters*) que concentran la mayor parte de la contribución al cálculo de la atención, disminuyendo así el uso de memoria y de ancho de banda durante la generación autorregresiva sin degradar significativamente la calidad de la salida [104].¹⁷
- **Cuantización de la caché KV:** en lugar de reducir el número de *tokens* almacenados, disminuye la precisión numérica de estos, reduciendo el tamaño de cada entrada. Dado que la caché KV suele estar limitada por el ancho de banda, esta técnica permite una mayor aceleración de la inferencia.

Codificaciones posicionales avanzadas Dado que el Transformer carece de redes de recurrencia o convolución que distingan el orden de los *tokens*, la incorporación de información posicional en la autoatención resulta esencial [2]. Además de las codificaciones sinusoidales originales y de los *embeddings* posicionales aprendidos, han cobrado relevancia alternativas más avanzadas, como las codificaciones posicionales relativas y las *Rotary Position Embeddings* (RoPE).

Las codificaciones posicionales relativas [106] modelan las distancias entre pares de posiciones en lugar de posiciones absolutas [101], partiendo de la intuición de que, en la autoatención, la relación posicional por parejas de los elementos de entrada (dirección y distancia) puede resultar más beneficiosa que sus posiciones absolutas [101]. Por su parte, las RoPE introducen la información posicional mediante rotaciones en subespacios bidimensionales del espacio de *embeddings*, combinando la codificación de posición absoluta con una dependencia explícita de la posición relativa. De este modo, dotan al modelo de un decaimiento progresivo de las interacciones a medida que aumenta la distancia relativa y de una mayor flexibilidad en el tratamiento de secuencias largas [101], [107].

Normalización y conexiones residuales La combinación de conexiones residuales y normalización por capas (*Layer Normalization* o LN) es clave para estabilizar el entrenamiento y mitigar la degradación del gradiente. Existen dos esquemas principales según su colocación [101]:

- **Post-LN:** el esquema original donde la LN se sitúa entre los bloques residuales (después de la suma). Aunque puede ofrecer mejor rendimiento final, suele requerir un calentamiento (*warm-up*) cuidadoso de la tasa de aprendizaje para evitar inestabilidades [101].
- **Pre-LN:** la normalización se aplica antes de cada subcapa (atención o FFN) dentro de la conexión residual. Este diseño facilita un entrenamiento más estable y permite prescindir de fases de *warm-up* prolongadas, siendo el estándar en modelos modernos [101].

¹⁷Quizás el término más adecuado sería podado o cribado de la caché KV.

Modificaciones en las capas feed-forward Las capas feed-forward posicionales representan una fracción significativa de los parámetros del Transformer, al escalar típicamente su dimensión interna a un factor de $4 \times d_{\text{model}}$ [2]. Entre las variaciones recopiladas por Lin *et al.* [101] para mejorar la eficiencia, se encuentran el uso de funciones de activación alternativas (como GELU [108] o Swish [109]) y arquitecturas basadas en unidades con compuertas (p. ej., bloques tipo GLU [110]), la reducción del ancho de estas capas o adaptación en función de la profundidad, o directamente la eliminación de estas [111].

2.3. Principales arquitecturas y variantes

Tras el establecimiento del Transformer como el estándar de facto en el procesamiento del lenguaje natural, la investigación ha evolucionado hacia arquitecturas que buscan balancear los costes computacionales con potenciar capacidades emergentes como el razonamiento complejo. En este contexto, es fundamental distinguir entre los modelos densos (monolíticos) y las nuevas variantes arquitectónicas.

2.3.1. Modelos monolíticos (densos)

Los modelos monolíticos, también denominados densos, activan todos sus parámetros durante la inferencia, lo que produce representaciones coherentes, pero implica un consumo computacional elevado debido a la complejidad $\mathcal{O}(n^2)$ respecto a la longitud de secuencia. Esta arquitectura, representativa de los primeros Transformers como BERT o GPT iniciales, limita su aplicabilidad en dispositivos por las demandas de memoria y latencia, aunque técnicas como la cuantización post-entrenamiento permiten mitigaciones parciales.

2.3.2. Modelos de razonamiento

Una tendencia inicialmente introducida por Wei *et al.* [65] es el escalado en el tiempo de prueba (*test-time scaling*) mediante técnicas de razonamiento multipaso o *Chain-of-Thought* (CoT). Esta variante no modifica necesariamente la arquitectura neuronal interna, sino que induce al modelo a generar pasos intermedios de razonamiento antes de emitir la respuesta final. Esta estrategia ha demostrado mejoras significativas en tareas de razonamiento lógico, matemático y simbólico en modelos de gran escala [65], [66]. Modelos como DeepSeek-R1 [63] o la serie OpenAI o1 [64] ejemplifican esta tendencia. Sin embargo, desde la perspectiva de la computación embebida, el CoT incrementa significativamente el número de *tokens* generados, lo que convierte a la fase de decodificación autorregresiva en el cuello de botella dominante de latencia y consumo energético [66].

2.3.3. Mixture of Experts

La arquitectura *Mixture of Experts* (MoE), basada en la propuesta por Jacobs *et al.* [112], [113], representa una variante de los Transformers en la que los bloques feed-forward estándar se sustituyen por conjuntos de subredes especializadas denominadas *experts*. Cada *expert* $E(x)$ posee sus propios parámetros, mientras que un

mecanismo de enrutamiento (*router*), implementado mediante una función de *gating* $G(x)$, selecciona dinámicamente un subconjunto de *experts* para procesar cada *token* de entrada x [98], [100]. Expresado de forma matemática para n *experts*:

$$y = \sum_{i=0}^{n-1} G_i(x) \cdot E_i(x) \quad (2.8)$$

De este modo, se incrementa la capacidad total del modelo (número de parámetros) sin un aumento proporcional en el coste computacional durante la inferencia, dado que cada *token* se procesa únicamente por un subconjunto limitado de *experts* activados de forma dispersa. Sin embargo, el coste en memoria se mantiene, ya que todos los expertos han de estar cargados para su eventual uso [98], [100]. Desde su propuesta original por Jacobs *et al.* [112], [113], las estrategias en modelos MoE han evolucionado notablemente en múltiples dimensiones, incluyendo la transición desde arquitecturas internas basadas en LSTM [114] a Transformers [115], el incremento en profundidad de las capas [116], la mejora en estabilidad durante el entrenamiento con GShard [116], y el desarrollo de estrategias de routing avanzadas entre las que se encuentran *top-k* (selección de los k mejores expertos) [114] y *hard routing* (solo se selecciona el mejor experto) [98], [100].

2.3.4. Alternativas al Transformer

Como respuesta a la ineficiencia cuadrática del mecanismo de atención de los Transformers, han emergido los Modelos de Espacio de Estados Estructurados (SSM) [117], orientados a mejorar la eficiencia en el procesamiento de secuencias largas. Destaca la arquitectura Mamba entre estos modelos. A diferencia de los Transformers, que deben atender a toda la historia de la secuencia, la premisa fundamental de los SSM es la capacidad de comprimir la historia de una secuencia de longitud L en un estado oculto interno de tamaño fijo t . Se procesa la secuencia de entrada h_t mediante una secuencia de estados internos x_t de la siguiente forma:

$$\begin{aligned} x_t &= Ax_{t-1} + Bh_t \\ y_t &= Cx_t \end{aligned} \quad (2.9)$$

donde A , B y C son las matrices de transición aprendidas. Esto permite una computación y un uso de memoria que escala de forma lineal $\mathcal{O}(n)$ con la longitud de la secuencia, facilitando el procesamiento de contextos extremadamente largos en *hardware* restringido [98], [100]. Existen también variantes híbridas como Jamba [118] o IBM Granite 4.0 [119], que combinan capas de atención y bloques MoE con capas SSM para maximizar la eficiencia.

2.4. Formatos numéricos y cuantización

La eficiencia en el entrenamiento y despliegue de modelos de lenguaje de gran escala (LLM) depende críticamente de la representación numérica de sus pesos y activaciones. Esta sección analiza la evolución de los formatos de punto flotante y las técnicas de cuantización empleadas para optimizar el uso de memoria y la velocidad de cómputo.

2.4.1. Evolución de los formatos base de entrenamiento

La transición entre formatos de punto flotante ha sido impulsada por la necesidad de mitigar fenómenos de desbordamiento (*overflow*) y subdesbordamiento (*underflow*) de gradientes sin incurrir en los costes de memoria y velocidad de cómputo asociados al formato FP32.

- **FP32 (*Single Precision*)**: basado en el estándar IEEE 754 [120], utiliza 1 bit para el signo, 8 para el exponente y 23 para la mantisa. Aunque proporciona un rango dinámico de aproximadamente 10^{-38} a $3,4 \times 10^{38}$, su uso en modelos de miles de millones de parámetros resulta ineficiente por su alta demanda de ancho de banda.
- **FP16 (*Half Precision*)**: utiliza 1 bit de signo, 5 bits para el exponente y 10 bits para la mantisa, ofreciendo un rango dinámico aproximado de ± 65.504 . Debido a su rango limitado, se emplea junto con técnicas de entrenamiento en precisión mixta (*Mixed Precision training*), donde los pesos y gradientes se mantienen en FP32 (resolución completa), mientras que los pasos hacia delante y hacia atrás se ejecutan en FP16, complementado con *loss scaling* para evitar *underflow* en gradientes pequeños [121].
- **BF16 (*Brain Floating Point*)**: desarrollado por Google Brain, este formato reasigna los 16 bits para mantener 8 bits de exponente y solo 7 de mantisa, manteniendo así el rango dinámico (pero no la precisión) del formato FP32, lo que además de facilitar las conversiones desde dicho formato, elimina la necesidad de usar *loss-scaling* presente en FP16, pudiendo representar los gradientes pequeños directamente [122].

A pesar de su popularidad y haberse convertido en prácticamente el estándar *de facto*¹⁸, el formato BF16 introduce desafíos críticos en la reproducibilidad y estabilidad numérica, especialmente en modelos de razonamiento complejo. Tal y como señalan Yuan *et al.* [123] estos problemas se derivan de:

- **No asociatividad aritmética**: La raíz del problema reside en la naturaleza no asociativa de la aritmética de punto flotante bajo precisión limitada. Esto significa que el orden de las operaciones (determinado por la programación de hilos en la GPU) altera el resultado final:

$$(a + b) + c \neq a + (b + c)$$

[123]

- **Sensibilidad a la configuración del sistema**: El rendimiento y las respuestas del modelo bajo BF16 son altamente sensibles a factores externos como el número de versión del *hardware* y el tamaño de lote (*batch size*) de evaluación. Cambios en estos parámetros pueden provocar variaciones de hasta un 9 % en la precisión y diferencias de hasta 9,000 *tokens* en la longitud de la respuesta en modelos como DeepSeek-R1-Distill-Qwen-7B [123].
- **Efecto cascada en modelos de razonamiento**: Debido a que BF16 tiene solo 7 bits de mantisa (frente a los 23 de FP32), los pequeños errores de redondeo en los primeros *tokens* se acumulan. En cadenas de pensamiento largas (*Chain of Thought*), esto provoca que las probabilidades de los *tokens* top-1 y top-2 se inviertan, desviando por completo la lógica del modelo [123].
- **Inconsistencia en el greedy decoding**: Aunque teóricamente el greedy decoding debería ser determinista al escoger siempre el *token* con mayor probabilidad, el uso de

¹⁸Véase que las últimas familias de modelos como Salamandra y Llama3.3 utilizan como formato base el BF16.

BF16 introduce una inestabilidad sustancial que impide obtener resultados idénticos en diferentes entornos de ejecución, lo que dificulta la validación científica de nuevos métodos [123].

Aun así, la demanda de mayor velocidad y eficiencia energética en el entrenamiento es tal que la investigación está derivando hacia la búsqueda de representaciones numéricas más pequeñas, llegando a formatos de 1.58 bit [124], [125].

2.4.2. Fundamentos y tipos de cuantización

La cuantización es la técnica principal para acelerar la inferencia de los LLM al disminuir el ancho de bits de los pesos, las activaciones o ambos [126]. Su principal reto es maximizar la eficiencia computacional sin degradar significativamente la precisión del modelo original [126].

Como expresan Husom *et al.* en [127] y Li *et al.* en [128] la cuantización se puede clasificar multidimensionalmente en los siguientes ejes:

- **Según la estrategia de entrenamiento:**
 - **Cuantización post-entrenamiento (PTQ):** se aplica después que un modelo haya sido entrenado sin modificar el proceso de entrenamiento original.
 - **Cuantización consciente del entrenamiento (QAT):** integra la cuantización directamente en el proceso de entrenamiento, simulando la pérdida de precisión de la cuantización en las propagaciones hacia adelante y hacia atrás.
- **Según la ubicación y granularidad:**
 - **Solo Pesos:** solo se cuantizan los tensores de pesos W de cada capa lineal.
 - **Pesos y Activaciones:** cuantizan tanto las activaciones (entradas) X y los pesos W de cada capa lineal.
 - **Caché KV:** cuantiza los estados de atención (claves K y valores V).
 - **Granularidad:** define el nivel de detalle con el que se aplican los parámetros (como factores de escala, eg. conversión flotante a entera) de la cuantización[128]:
 - **Por tensor:** se aplica un único conjunto de parámetros a cada tensor.
 - **Por canal:** se asignan parámetros de forma individuales para cada canal del tensor, capturando mejor sus características específicas.
 - **Por grupo:** donde múltiples canales o capas son cuantizados con los mismos parámetros de cuantización.
- **El ancho de bit de destino:** Siendo típicamente 8 en formato flotante o entero o 4 en formato entero. También puede darse el caso (sobre todo en SMLs) que se use una precisión mixta para preservar la sensibilidad de ciertas capas o proteger *outliers* de forma similar a los *heavy-hitters* descritos en la Sección 2.2.3 [128].

2.5. Otros métodos de reducción

Más allá de las variaciones modales de la arquitectura Transformer (Sección 2.2.2) y el empleo de anchos de bits reducidos (Sección 2.4), existen técnicas complementarias que permiten una optimización del rendimiento del modelo bien por una modificación de la topología del modelo (poda) o una compresión del conocimiento (destilación).

2.5.1. Poda de parámetros (*Pruning*)

La poda consiste en la eliminación de componentes redundantes del modelo (pesos, neuronas o canales) que no contribuyen significativamente a la precisión final. Siguiendo los principios de Han *et al.* [129] el objetivo es inducir dispersión (*sparsity*) en las matrices de pesos para reducir el almacenamiento y el coste computacional.

En el contexto de los Transformers y LLM, como señalan Guo *et al.* [100], se distinguen dos enfoques principales:

- **Poda No estructurada:** elimina los pesos individuales basándose en su magnitud (los más cercanos a cero). Aunque permite tasas de compresión teóricas muy altas, su aceleración en *hardware* comercial es limitada debido a que las matrices dispersas resultantes no suelen alinearse con las capacidades de procesamiento de los núcleos de cálculo estándar.
- **Poda Estructurada:** elimina unidades completas, como cabezas de atención o capas de la red generando arquitecturas más pequeñas y densas que pueden ser aceleradas directamente por las unidades de cómputo. Sin embargo, es difícil mantener un equilibrio entre la compresión y rendimiento adecuado.

2.5.2. Destilación de conocimiento (*knowledge distillation*)

La destilación de conocimiento (*knowledge distillation*), originalmente propuesta por Hinton *et al.* [130], es una técnica situada a medio camino entre el (post)entrenamiento y la optimización, en la que un modelo pequeño y optimizado, denominado alumno, se entrena para reproducir el comportamiento de un modelo más grande y complejo, denominado maestro, comprimiendo de este modo el conocimiento del maestro en el alumno.

En función del acceso disponible a los parámetros internos del modelo maestro, y tal como describen Yang *et al.* [131], la destilación se puede clasificar en dos grandes grupos: de caja blanca (*white box*) y de caja negra (*black box*). En la destilación de caja blanca se dispone de acceso a la información interna del modelo, mientras que en la de caja negra únicamente se tiene acceso a las salidas del maestro (p. ej., a través de una API).

2.6. Métodos de actualización post-entrenamiento

Los métodos de actualización post-entrenamiento permiten adaptar modelos de lenguaje preentrenados a dominios o tareas específicas sin incurrir en los costes prohibitivos de un entrenamiento completo desde cero. Esta adaptación, aunque no abordada en el presente TFG, resulta particularmente relevante en los modelos de tamaño reducido, cuyas capacidades generales son más limitadas que las de los modelos frontera

[4], [5] y que, por tanto, requieren una especialización dirigida para alcanzar un rendimiento competitivo en tareas concretas. Aunque el ajuste fino completo (*Full Fine-Tuning*, FFT) actualiza la totalidad de los parámetros del modelo, las tendencias actuales se orientan hacia la eficiencia paramétrica, el alineamiento robusto y la integración de conocimiento dinámico.

2.6.1. Ajuste fino eficiente en parámetros (PEFT)

El paradigma PEFT surge como una respuesta a la crisis de escalabilidad del FFT, en la que almacenar múltiples instancias de modelos con miles de millones de parámetros resulta inviable. Para resolver este problema, el enfoque se centra en modificar solo una fracción muy pequeña del total de parámetros. Estos métodos se agrupan, principalmente, en tres¹⁹ tipos de arquitecturas: reparametrización, aditivas, selectivas [132].

Métodos de reparametrización: LoRA

La Adaptación de Bajo Rango (LoRA) [133] se fundamenta en la hipótesis de que las actualizaciones de pesos durante el aprendizaje poseen una “dimensión intrínseca” baja. Para una matriz de pesos preentrenada $W_0 \in \mathbb{R}^{d \times k}$, LoRA propone representar la actualización ΔW mediante el producto de dos matrices de bajo rango $B \in \mathbb{R}^{d \times r}$ (inicializada a 0) y $A \in \mathbb{R}^{r \times k}$ (con inicialización Gaussiana aleatoria), donde $r \ll \min(d, k)$. La salida se expresa como:

$$h = W_0 x + \Delta W_x = W_0 x + \frac{\alpha}{r} B A_x \quad (2.10)$$

donde $\frac{\alpha}{r}$ escala la contribución de la adaptación.

LoRA destaca por su eficiencia en el entrenamiento, reduciendo hasta 3 veces la barrera de *hardware* al optimizar solo las matrices de bajo rango sin calcular gradientes para la mayoría de parámetros. Su diseño lineal permite fusionar las matrices entrenables con pesos congelados al desplegar, sin latencia extra en inferencia frente a *fine-tuning* completo. Además, es ortogonal a otros métodos y se combina fácilmente, manteniendo alto rendimiento con bajo consumo de recursos [133].

Métodos aditivos: *Adapters* y *Prefix Tuning*

Los métodos aditivos insertan nuevos módulos en la arquitectura Transformer, manteniendo congelados los pesos originales del modelo preentrenado. Estos métodos permiten una adaptación eficiente a tareas específicas con un número mínimo de parámetros adicionales, agrupándose principalmente en *adapters* y *prefix-tuning* [132], [134], [135].

Adapters Los *adapters* o adaptadores, propuestos originalmente por Houlsby *et al.* [134], se definen como módulos de cuello de botella (*bottleneck*) integrados secuencialmente dentro de la arquitectura del Transformer. Su implementación se realiza en dos puntos específicos de cada capa: inmediatamente después de la subcapa de atención multi-cabeza (*Multi-Head Attention*) y tras la subcapa de red *feed-forward*. Estructuralmente, como se

¹⁹La fuente [132] empleada como referencia incluye dos tipos adicionales, que por considerarse derivados de los aquí expuestos no se han incluido (híbridos y cuantización).

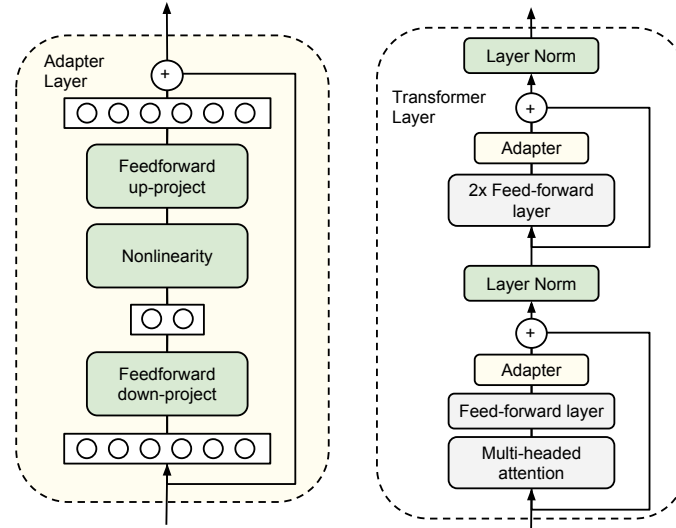


Figura 2.12: Composición de un *adapter* (adaptador) (izquierda) y ubicación del adaptador dentro del Transformer (derecha). Fuente: [134]

puede apreciar en la Figura 2.12, un *adapter* opera mediante una proyección lineal que reduce la dimensión del vector de características d a un espacio latente de menor dimensionalidad m ($m \ll d$), para posteriormente aplicar una función de activación no lineal (frecuentemente ReLU) y, finalmente, proyecta los datos de vuelta a la dimensión original d . El módulo incorpora una conexión residual que garantiza que, mediante una inicialización cercana a cero, el comportamiento inicial del modelo sea equivalente a la identidad. Durante el proceso de transferencia de aprendizaje, los parámetros del modelo preentrenado permanecen congelados, optimizándose únicamente los *adapters* y las capas de normalización [134].

Prefix-Tuning El *prefix-tuning*, introducido por Li. *et al* [135], constituye una alternativa ligera al ajuste fino convencional que no modifica los pesos internos del modelo de lenguaje (LM), sino que optimiza una secuencia de vectores continuos específicos para la tarea, denominada *prefix*. A diferencia de los métodos de inserción de capas, el *prefix-tuning* actúa directamente sobre el flujo de activaciones: el prefijo se concatena al inicio de la secuencia de entrada y se propaga en todas las capas de la arquitectura Transformer [135].

Métodos selectivos

Los métodos selectivos se distinguen por identificar y actualizar única y específicamente un subconjunto de los parámetros preexistentes del modelo, manteniendo el resto congelado. Destaca BitFit propuesto por Zaken *et al.* [136], que actualiza exclusivamente los términos de sesgo (*bias*) de las transformaciones lineales del Transformer (normalización, *feed-forward* y atención multicabeza si esta emplea sesgos en las proyecciones). La operación se define matemáticamente como $h = f(Wx + b)$, donde solo b es optimizable.

2.6.2. Generación aumentada por recuperación (RAG)

La Generación Aumentada por Recuperación (RAG, por sus siglas en inglés) [137] es un paradigma híbrido de post-entrenamiento orientado a mejorar la generación de texto en

tareas intensivas en conocimiento mediante mecanismos de recuperación externa. Integra modelos generativos paramétricos preentrenados (memoria paramétrica, p. ej., un LLM) con una memoria no paramétrica externa (un índice denso de documentos, como Wikipedia), accesible bajo demanda mediante recuperadores densos (o dispersos) preentrenados (p. ej., DPR con bi-encoders BERT) [137]-[139].

Su propósito principal consiste en mitigar limitaciones inherentes de los LLMs puros, como las alucinaciones factuales, el conocimiento desactualizado y la ausencia de procedencia, fundamentando las respuestas en evidencia recuperada y actualizada, indexada mediante variaciones del algoritmo aproximado del vecino más cercano (ANNS), lo que produce salidas más factuales, específicas y diversas [100].

En su funcionamiento, tal y como se aprecia en la Figura 2.13, intervienen 3 elementos:

1. **Memoria No Paramétrica (base de datos de conocimiento):** repositorio externo de información donde los documentos se dividen en fragmentos (*chunks*). Estos segmentos se transforman en representaciones vectoriales (*embeddings*) y se indexan en una base de datos vectorial utilizando estructuras de datos optimizadas para ANNS ((eg. HNSW para búsquedas jerárquicas rápidas, DiskANN para *datasets* grandes en disco, Faiss con aceleración GPU ... [100])).
2. **Recuperador (Mecanismo de Selección):** actúa como la interfaz entre la consulta del usuario y el corpus. Ante una entrada (*query*), el recuperador (típicamente un bi-encoder) codifica la pregunta en el mismo espacio latente que los documentos. Mediante el cálculo de similitud semántica, extrae los k fragmentos más relevantes.
3. **Modelo Paramétrico o Generador (LLM):** es el componente encargado de la síntesis final, recibe del recuperador un *prompt* aumentado que concatena la consulta original con los fragmentos recuperados. El modelo procesa este contexto para producir una respuesta que no solo es coherente lingüísticamente, sino que está anclada en la evidencia externa, permitiendo además la atribución de fuentes y la procedencia de la información.

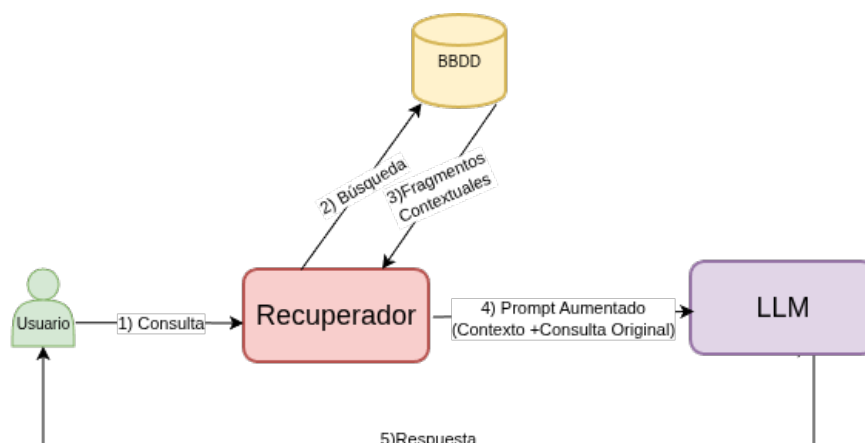


Figura 2.13: Diagrama de flujo del paradigma RAG (Retrieval-Augmented Generation).

2.7. Funcionamiento de la inferencia

2.7.1. Pipeline lógico de inferencia

Tal y como expone Whatmough en su artículo “*Generative Artificial Intelligence on Edge Devices: Models, Hardware, and Systems: A Tutorial*” [22], y complementando lo explicado en la Sección 2.2.1, el funcionamiento lógico de la inferencia en los modelos de solo decodificador (*decoder-only*) puede descomponerse en siete etapas, tal como muestra la Figura 2.14:

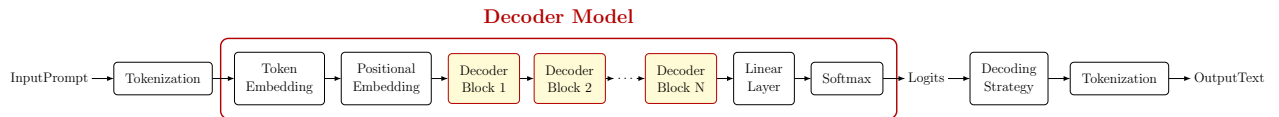


Figura 2.14: Diagrama de alto nivel del *pipeline* lógico de inferencia. Extraído de: [22].

1. **Tokenización:** división del texto de entrada en unidades llamadas *tokens*, que pueden corresponder a palabras completas, fragmentos de palabra, espacios o signos de puntuación.
2. **Embedding de token:** transformación de cada *token* en un vector numérico en un espacio latente, que mantiene proximidad semántica.
3. **Embedding posicional:** incorporación de información relativa a la posición secuencial del *token* en la oración, esencial para conservar el contexto y el sentido del texto.
4. **N-Bloques decodificadores:** la secuencia de *embeddings* enriquecida con la información posicional se procesa a través de *N* bloques decodificadores apilados, estructurados de forma análoga a los descritos en la Sección 2.2.1. Cada bloque combina un mecanismo de atención multicabeza y una red feed-forward posicional, ambos envueltos por conexiones residuales y capas de normalización, lo que permite refinar progresivamente las representaciones contextuales de los *tokens* y capturar dependencias de largo alcance en la secuencia
5. **Capa lineal:** tras procesar la información a través de los bloques decodificadores, se aplica una capa lineal que proyecta las representaciones internas a un espacio dimensional equiparable al tamaño del vocabulario para la predicción.
6. **Decodificación:** esta etapa aplica la función softmax sobre la salida de la capa lineal, convirtiendo los logits en una distribución de probabilidad sobre los posibles *tokens* a generar.
7. **(Re)tokenización:** finalmente, las probabilidades generadas son transformadas nuevamente en texto legible.

2.7.2. La inferencia vista desde el hardware

Desde la perspectiva de recursos *hardware*, la inferencia según [22] se puede dividir en 2 etapas diferenciadas: el procesamiento del *prompt* o *prefill*, y la generación de *tokens* o *decode*. En *prefill*, se procesan lotes de *tokens* de entrada reutilizando los pesos del modelo, lo que minimiza el coste computacional por *token* y la limitación principalmente viene de la capacidad de cómputo (*compute-bound*), generando las cachés de pares clave-valor (KV

cache) para atención eficiente incrementando el uso de la memoria RAM, pero no *tokens* de salida [22].

Por contraste, la fase de *decode*, encargada de la emisión de *tokens* de salida, genera cada nuevo *token* mediante una pasada completa del modelo, que se encadena a la secuencia de *tokens* de entrada *ad nauseam* hasta alcanzar un *token* de fin de secuencia o el límite máximo de longitud de secuencia. Esto exige cargar el modelo entero por cada *token*, haciendo que el coste en ancho de banda de memoria supere al computacional, denominándose así *memory-bound* [22].

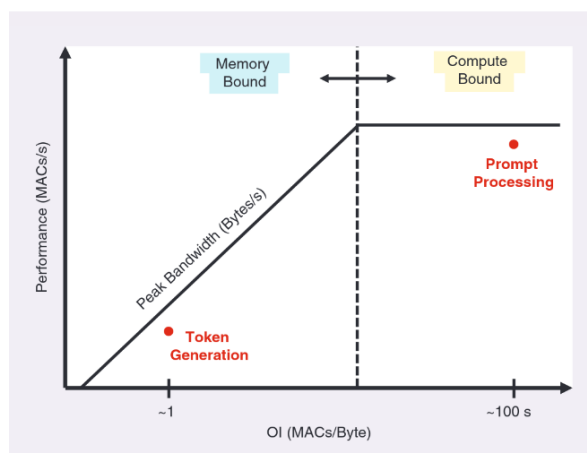


Figura 2.15: Modelo *Roofline* para la inferencia de LLMs. Se observa cómo la Generación de *tokens* está limitada por el ancho de banda de memoria (*memory-bound*), mientras que el Procesamiento del *Prompt* (*prefill*) tiende a estar limitado por la capacidad de cómputo (*compute-bound*) debido a su mayor intensidad operativa (OI). Extraído de [22].

En la Figura 2.15, se ilustra mediante un modelo del tejado (intensidad operativa OI en MACs/byte vs. rendimiento en MACs/s) la dicotomía entre las cargas de trabajo de cada etapa, el *prefill* en la región derecha (limitada por el pico del cómputo), y *decode* en la izquierda (por el techo diagonal de ancho de banda DRAM).

Sin embargo, al trasladar estos conceptos teóricos a una traza de ejecución real, como la mostrada en la Figura 2.16, es necesario identificar primero la cronología operativa de la inferencia para evitar interpretaciones erróneas. En la gráfica se distingue inicialmente una pendiente lineal creciente de consumo de memoria (línea roja) que no corresponde al llenado de cachés KV del *prefill*, sino a la fase de carga del modelo (*Model Loading*), una operación dominada por la transferencia de los pesos desde el almacenamiento secundario. Una vez cargado el modelo en RAM, la actividad de la CPU (línea azul) se dispara, abarcando en un único bloque continuo de alta utilización tanto el procesamiento del *prompt* (*prefill*) como la generación de la respuesta (*decode*).

Esta homogeneidad visual es coherente con la naturaleza de las métricas monitorizadas (consumo de RAM y CPU), ya que la Figura 2.16 representa la ocupación de memoria (en MB) y no el ancho de banda efectivo. Por tanto, el cambio de régimen teórico —de limitado por cómputo a limitado por memoria— no se traduce en una variación perceptible²⁰ en la curva de consumo, dado que el modelo permanece residente en RAM y el procesador se mantiene saturado independientemente de si el cuello de botella es aritmético-lógico o de transferencia. En consecuencia, para identificar el punto exacto de

²⁰ Concretamente, las variaciones en la capacidad de cómputo y en el ancho de banda disponible sí repercuten en la duración de las fases de *prefill* y *decode*, aunque no en el nivel absoluto de ocupación de RAM y CPU.

transición entre la evaluación del *prompt* y la generación de *tokens*, resulta imprescindible correlacionar la traza de *hardware* con los eventos de registro internos del motor de inferencia.

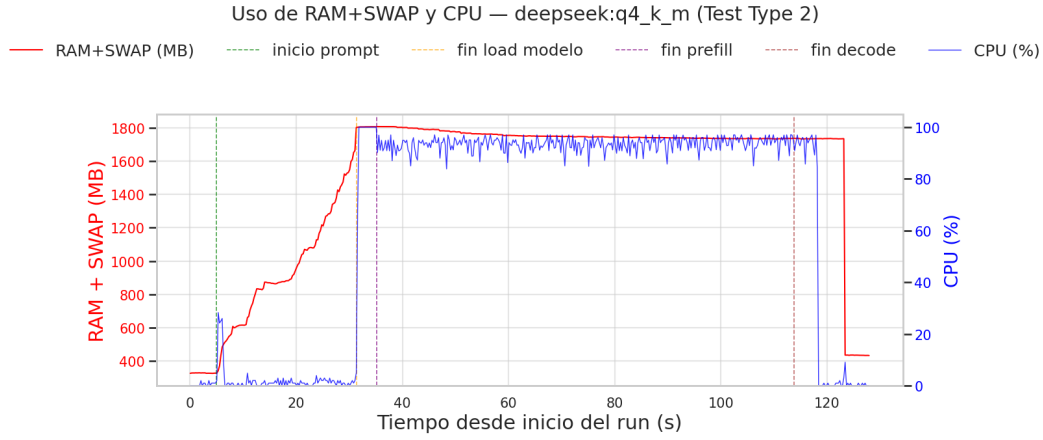


Figura 2.16: Consumo de RAM (MB) y CPU (%) a lo largo del tiempo para el modelo DeepSeek-R1-Distill-Qwen-1.5B. Las delimitaciones verticales discontinuas indican el comienzo de las fases secuenciales de carga de pesos (*model loading*, verde), procesamiento de entrada (*prefill*, naranja) y generación de *tokens* (*decode*, morado).

2.7.3. La inferencia desde los motores de inferencia

La inferencia vista desde la perspectiva de los motores de inferencia se puede dividir en tres fases²¹ diferenciadas:

1. **Carga del modelo (*model loading*):** en esta fase se trasladan los pesos desde el almacenamiento masivo a la memoria RAM o VRAM. Este proceso está dominado por el ancho de banda del bus de memoria. Además, se realiza una reserva estratégica de la caché KV, cuyo tamaño en bytes²² puede estimarse, según la expresión adaptada de Chen *et al.* [140], como:

$$\text{Tamaño KV}_{\text{Bytes}} = B_{\text{prompts}} \times 2 \times L \times S \times (n_{kv_cabeza} \times d_{cabeza}) \times Q \quad (2.11)$$

Donde:

- B_{prompts} : corresponde al número de secuencias (*prompts*) procesadas simultáneamente. También se le denomina en la literatura como *batch size*, pero para evitar confusión con el *batch size* de *prefill* que controla el número de *tokens* que se procesan simultáneamente en dicha fases. Por esta razón se ha optado por denominarlo *prompt batch size* y reservar *batch size* para el de *prefill*.
- 2: factor que se corresponde al almacenamiento de los 2 principales elementos: claves(*keys*) y valores(*values*).
- L : número de capas del modelo.
- S : tamaño de la secuencia en *tokens*. También denominada contexto máximo o ventana de contexto.

²¹Dependiendo del motor empleado, estas fases pueden recibir distintos nombres a los aquí utilizados.

²²La huella en memoria ocupada por la caché KV puede verse afectada por las técnicas de optimización de la atención, como la *PagedAttention*, entre otros factores.

- n_{kv_cabeza} : número de cabezas de KV. Puede variar según las técnicas de atención empleadas como en GQA(*Grouped Query attention*) o MQA (*Multi-Query Attention*) donde el número de cabezas se corresponde con la división del número de cabezas entre un factor de grupo G ($n_{kv_cabeza} = \frac{n_{cabezas}}{G}$) o 1 respectivamente.
 - d_{cabeza} : Dimensión de la cabeza de atención. Definida como la división de la dimensión del modelo entre el número total de cabezas de atención del modelo: $\frac{d_{modelo}}{n_{cabezas}}$.
 - Q : precisión de los datos en Bytes(eg. 2 para BF16).
2. **Procesamiento del *prompt*(*prompt evaluation* o *prefill*)**: en esta fase el motor procesa B secuencias de entrada de tamaño S de forma paralela, generando y almacenando los pares clave-valor generados. Al procesarse todos los *tokens* en un único pase, esta etapa está típicamente limitada por la capacidad de cómputo (*compute-bound*), tal y como muestra la Figura 2.15. Como se puede intuir, a mayores valores de S y $B_{prompts}$ más aumenta el uso de memoria.
 3. **Generación de la respuesta (*evaluation* o *decode*)**:el comportamiento de la generación está modulado adicionalmente por la temperatura, que controla la aleatoriedad de la distribución softmax; el parámetro *top-k*, que restringe el muestreo a los k *tokens* más probables; y el parámetro *top-p* o muestreo de núcleo (nucleus sampling), que selecciona dinámicamente el subconjunto mínimo de *tokens* cuya probabilidad acumulada supera un umbral p . Estos parámetros, junto con la semilla (*seed*), afectan directamente a la reproducibilidad experimental.

Tabla 2.2: Parámetros de inferencia y su impacto en el rendimiento del hardware.

Parámetro	Fase Dominante	Cuello de Botella	Impacto en el Sistema
Cuantización (Q)	Carga / Prefill	Memory-bound	Reduce ancho de banda y RAM
Prompt Batch Size ($B_{prompts}$)	Prefill	Compute-bound	Aumenta el TFLOPS/utilización
Prompt Batch Size ($B_{prompts}$)	Decode	Memory-bound	Mejora el uso del bus de memoria
Contexto (S)	Prefill	Compute-bound	Escalado cuadrático ($O(S^2)$)
Contexto (S)	Decode	Memory-bound	Escalado lineal de la caché KV
Temperatura/ <i>Seed</i> / <i>Top-k</i> , p	Decode (Sampling)	Ninguno	Reproducibilidad de los resultados

2.7.4. Métricas de evaluación de la inferencia

La caracterización del rendimiento de un sistema de inferencia de LLM requiere un conjunto de métricas que capturen tanto la dimensión temporal como la energética y la de calidad. A continuación se definen las principales empleadas en el presente trabajo, junto con las denominaciones alternativas habituales en la literatura. El sistema de medida completo y el desglose por fase se detallan en el Capítulo 3.

- **Latencia:** métricas temporales que caracterizan el tiempo de respuesta del sistema durante el proceso de inferencia.
 - **Tiempo de carga del modelo (TTLM, *Time To Load Model*)**: intervalo transcurrido desde la invocación del motor de inferencia hasta que el modelo queda completamente cargado en memoria y listo para procesar el primer *prompt*:

$$TTLM = t_{\text{modelo listo}} - t_{\text{invocación}} \quad (2.12)$$

- **Tiempo hasta el primer token (TTFT, Time To First Token):** tiempo transcurrido desde la recepción del *prompt* hasta la emisión del primer *token* de respuesta [22]:

$$\text{TTFT} = t_{\text{primer token}} - t_{\text{prompt}} \quad (2.13)$$

Esta métrica está dominada por la fase de *prefill* (Sección 2.7.2) y resulta especialmente relevante en aplicaciones interactivas.

- **Tiempo total de generación:** intervalo transcurrido desde la recepción del *prompt* hasta la emisión del último *token* de la respuesta:

$$T_{\text{gen}} = t_{\text{último token}} - t_{\text{prompt}} \quad (2.14)$$

Esta magnitud integra las fases de *prefill* y *decode*, proporcionando una medida global de la latencia de inferencia.

■ Rendimiento:

- **Tasa de generación (TPS, Tokens Per Second o Throughput):** número de *tokens* generados por unidad de tiempo durante la fase de *decode* [127], [140]:

$$\text{TPS} = \frac{N_{\text{tokens}}}{T_{\text{gen}} - \text{TTFT}} \quad (2.15)$$

donde N_{tokens} es el número de *tokens* generados en la respuesta. Esta métrica depende principalmente del ancho de banda de memoria disponible (Sección 2.7.2). Adicionalmente, el rendimiento se expresará en palabras por segundo (*words per second*, WPS). Esta representación reduce la influencia de las diferencias entre tokenizadores, permitiendo una comparación más homogénea entre modelos. Asimismo, facilita la comparación con la capacidad de percepción humana durante la lectura, estimada aproximadamente entre 3 y 5 palabras por segundo [22].

- **Tiempo por token de salida (TPOT, Time Per Output Token):** tiempo medio entre la emisión de dos *tokens* consecutivos durante la fase de *decode* [140]:

$$\text{TPOT} = \frac{T_{\text{gen}} - \text{TTFT}}{N_{\text{tokens}}} = \frac{1}{\text{TPS}} \quad (2.16)$$

Es la inversa de la tasa de generación y expresa la latencia asociada a cada *token* generado.

■ Consumo energético:

- **Potencia media (W):** valor medio de la potencia eléctrica consumida por el sistema durante una fase de inferencia determinada [141], [142]:

$$\bar{P} = \frac{1}{\Delta t} \int_{t_0}^{t_0 + \Delta t} P(t) dt \approx \frac{1}{n} \sum_{i=1}^n P_i \quad (2.17)$$

donde Δt es la duración de la fase considerada y P_i las muestras obtenidas por el sistema de monitorización. A partir de esta magnitud se calcula la energía consumida:

$$E = \bar{P} \cdot \Delta t \quad (2.18)$$

- **Eficiencia energética (*tokens/J*):** número de *tokens* generados por julios consumidos durante el run [127], [128]:

$$\varepsilon = \frac{\text{tokens/s}}{\bar{P}_{run}} \quad (2.19)$$

Esta métrica²³ normaliza el consumo energético respecto a la producción útil del sistema, permitiendo comparar modelos y configuraciones.

■ **Métricas de calidad:**

- **Perplejidad (*Perplexity*):** mide la incertidumbre del modelo al predecir una secuencia de *tokens* [140]:

$$PP(W) = \exp\left(-\frac{1}{N} \sum_{i=1}^N \log P(w_i|w_1, \dots, w_{i-1})\right) \quad (2.20)$$

Valores menores indican una mayor capacidad predictiva.

- **Exactitud (*Accuracy*):** proporción de respuestas correctas obtenidas en conjuntos de evaluación estandarizados (*benchmarks*) [127]:

$$\text{Acc} = \frac{N_{\text{correctas}}}{N_{\text{total}}} \quad (2.21)$$

2.8. Estado del arte

A continuación se ofrece una visión general del estado del arte dividida en tres enfoques.

2.8.1. De los motores de inferencia

El ecosistema de motores de inferencia para LLM ha experimentado una expansión significativa desde 2023, impulsada por la necesidad de ejecutar modelos de forma eficiente tanto en infraestructuras de centro de datos como en dispositivos de recursos limitados. Tal y como sintetiza la *survey* de Park *et al.* [66], estos motores pueden clasificarse en primera instancia según su grado de especialización en modelos de lenguaje en dos grandes familias: los *frameworks* de propósito general y los motores de inferencia especializados.

Los *frameworks* de propósito general, entre los que destacan PyTorch [143] y TensorFlow [144], están diseñados para soportar prácticamente cualquier tipo de red neuronal. Aunque incorporan optimizaciones, estas son de carácter amplio y generalista, quedando las optimizaciones específicas para LLM relegadas a extensiones opcionales de ser compatibles. Son comunes en análisis que estudian aplicaciones no-LLM o las combinan con LLM [141], [142], [145], pero su empleo en inferencia de LLM se traduce por lo general en un rendimiento inferior y un mayor consumo de recursos [66].

En contraposición, los motores de inferencia especializados, que incorporan capacidades no presentes en los *framework* generalistas, como la paginación de la atención [103], *streaming* o paginación por lotes. No obstante, cada motor responde a unos objetivos de desarrollo

²³En el presente TFG se denota como ε en lugar de η , notación habitual en la literatura, para evitar confusión con la eficiencia computacional de la CPU, η_{CPU} , definida en la Ecuación 3.5.

propio que condicionan su plataforma *hardware* de destino (heterogeneidad), su ámbito de aplicación y escalabilidad (*edge* o *cloud*) y el grado de facilidad de uso o de aplicación de las medidas de optimización [66]. Dentro del ámbito de la inferencia local por su uso en análisis de optimización (véase 2.8.3), destacan tres motores directamente relevantes para el presente TFG.

- llama.cpp: proyecto de código abierto iniciado por Georgi Gerganov y mantenido por la comunidad, es una librería C++ para inferencia LLM en CPU (x86/ARM) y GPUs NVIDIA, con cuantización agresiva (1.5-8 bits) y formato GGUF²⁴ unificado. Ofrece kernels *hardware*-específicos (OpenBlas), FlashAttention opcional y servidores HTTP compatibles con OpenAI API. Optimizado para single-node con multithreading básico, carece de soporte nativo multi-nodo. Su usabilidad es extendida mediante proyectos como llama-run y llama-simple [66], [147].
- Ollama: desarrollado por Ollama Inc. y la comunidad, escrito en Go, está diseñado para despliegue local de LLM en una sola GPU, usando llama.cpp como *backend* en una arquitectura cliente-servidor. Soporta modelos como Llama, Mistral y DeepSeek en formatos GGUF/Safetensors²⁵ a través de *Modelfiles*, con CLI y REST API para chat y generación. Prioriza facilidad de uso (instalación en una línea, integraciones como Open WebUI) sobre optimizaciones avanzadas: sin multi-GPU, sin batching continuo ni gestión de memoria especializada. Compatible con NVIDIA/AMD/ARM [66], [67].
- vLLM: desarrollado originalmente por investigadores de la Universidad de California Berkeley y actualmente mantenido como proyecto de código abierto con amplia participación industrial, es un servidor de alto rendimiento para LLM con *PagedAttention* (KV-cache paginado), *continuous batching*, FlashAttention-3 y paralelismo tensorial. Soporta GPU/Google TPU/Inferentia, AsyncLLM para concurrencia y *cascade inference* para prefijos compartidos[66]. Su orientación a entornos de servidor de grado empresarial lo hace, sin embargo, inviable para el despliegue en plataformas embebidas de recursos limitados por lo que no se abordará en este TFG.

Un aspecto crítico para el presente TFG, no abordado por Park *et al.* [66], es la compatibilidad de estos motores con los aceleradores *hardware* comerciales disponibles para plataformas embebidas. Tras el análisis de la documentación y los repositorios de desarrollo de los tres motores considerados, se constata que ninguno implementa actualmente conectividad con los aceleradores ampliamente utilizados y propuestos para este TFG como son Google Coral [149] ni Hailo-8 (Raspberry Pi AI Kit [150]). Los nuevos aceleradores específicos para LLM, como el Hailo-10H (Raspberry Pi AI HAT+ 2 [151]) o el AI-8850 LLM [152], emplean runtimes con APIs propias, sin integración oficial en los motores anteriores²⁶. Esta limitación implica que no es posible establecer una comparativa en igualdad de condiciones entre inferencia CPU pura e inferencia con acelerador empleando un único motor de referencia.

²⁴GGUF: formato de archivo binario, sucesor de GGML, que almacena en un único fichero los tensores del modelo junto con sus metadatos y parámetros de cuantización, optimizado para la inferencia local de modelos de lenguaje [146]. Es el formato empleado por llama.cpp y herramientas derivadas [146].

²⁵Safetensors: formato de archivo para almacenar tensores de modelos de manera segura y eficiente, desarrollado por Hugging Face como alternativa a los formatos basados en pickle (como .bin o .pth). Evita los riesgos de ejecución de código arbitrario de estos y permite una carga rápida mediante copia cero (zero-copy) y carga diferida (lazy loading)[148]. A diferencia de GGUF, almacena únicamente los tensores y no incorpora metadatos del modelo ni cuantización [148].

²⁶véase <https://github.com/ggml-org/llama.cpp/issues/11603#issuecomment-3947847152>

2.8.2. De los modelos

El ecosistema de SLM ha experimentado una expansión y diversificación aceleradas desde 2024, impulsado por las necesidades de soberanía tecnológica y las inversiones en proyectos de desarrollo expuestas en la Sección 2.1, la madurez de las plataformas de inferencia local descritas en la Sección 2.8.1, la creciente demanda de aplicaciones de IA en entornos restringidos, y las mejoras arquitectónicas descritas en las secciones 2.2.3 y 2.3. No obstante, este estado del arte queda acotado por las restricciones de memoria de la plataforma objetivo²⁷: la huella en memoria de un modelo cuantizado puede estimarse como $Q \times n_{\text{parámetros}}$, a lo que deben sumarse la memoria reservada por el sistema operativo, el *overhead* del motor de inferencia y la caché KV, cuyo crecimiento con la ventana de contexto queda descrito por la Ecuación 2.11. Estas restricciones sitúan, en la práctica, el tamaño máximo de los modelos utilizables en torno a 10 B de parámetros en cuantización de 4 bits, y a menos de 3 B en precisión de 16 bits.

A continuación se describen las principales familias de modelos consideradas, para posteriormente exponer los criterios de inclusión y descarte que han determinado el conjunto experimental definitivo.

Familia Llama 3.x

La familia Llama, desarrollada por Meta, constituye uno de los principales referentes en modelos *open-weights* para inferencia local. Basada en arquitectura Transformer de solo decodificador, ha evolucionado desde Llama [153] hacia versiones progresivamente más eficientes y robustas. Para este TFG se han considerado las variantes Llama 3.2 [154], Llama 3.3 [155] y la versión multimodal Llama 4 [156], si bien únicamente la primera resulta adecuada para la inferencia en *hardware* de bajo consumo, dadas sus variantes de 1 B y 3 B de parámetros.

Según la documentación oficial [154], Llama 3.2 incluye modelos de lenguaje generativos preentrenados e *instruction-tuned* disponibles en tamaños de 1 B (1,23 B parámetros efectivos) y 3 B (3,21 B). Ambas variantes emplean GQA, lo que mejora la escalabilidad en inferencia, y admiten una ventana de contexto de hasta 128 k *tokens*. El ajuste fino combina SFT, *Rejection Sampling* (RS) y DPO, mientras que el preentrenamiento se realizó sobre un máximo de 9 billones de *tokens* de datos públicos, incorporando además destilación de conocimiento a partir de los *logits* de los modelos Llama 3.1 8 B y 70 B para compensar la reducción de capacidad derivada de la poda. El conocimiento de los modelos tiene fecha de corte en diciembre de 2023.

Un aspecto diferencial de Llama 3.2 respecto a Llama 3.1 es su orientación explícita al despliegue en entornos con recursos muy limitados, como dispositivos móviles o placas embebidas. Con este fin, Meta proporciona variantes cuantizadas oficiales mediante dos técnicas: SpinQuant (basada en rotación ortogonal del espacio de pesos junto con cuantización GPTQ) y QLoRA, que combina cuantización de 4 bits por grupo con adaptadores LoRA entrenados mediante *Quantization-Aware Training* (QAT) [154].

Familia Gemma 3²⁸

²⁷Raspberry Pi 5 con 8 GB de memoria RAM compartida, cuyas características técnicas completas se describen en el Capítulo 3.

²⁸Durante la elaboración de este TFG, la familia Gemma 3 fue sucedida por Gemma 4 [157].

La familia Gemma, desarrollada por Google DeepMind, comprende modelos ligeros de código abierto contruidos sobre la misma base tecnológica que los modelos Gemini. En el rango considerado en este trabajo (de hasta 10 mil millones de parámetros), las variantes relevantes son Gemma 3 1B y 4B, y los modelos Gemma 3n E2B y E4B, ambas líneas diseñadas explícitamente para ejecutarse en *hardware* de bajo consumo.

Los modelos Gemma 3 1B y 4B [158] comparten una arquitectura Transformer de solo decodificador con GQA y normalización QK-norm. Su principal innovación arquitectónica es la proporción 5:1 entre capas de atención local y global, con ventana deslizante de 1.024 *tokens*, que reduce la sobrecarga de la caché KV de hasta un 60 % a menos del 15 % con un impacto mínimo en la perplejidad [158]. El modelo 1 B se limita al procesamiento de texto con una ventana de contexto de 32 k *tokens*, mientras que el 4 B incorpora el codificador visual SigLIP y extiende el contexto a 128 k *tokens* mediante reajuste de RoPE.

Por otro lado, los modelos Gemma 3n E2B y E4B introducen un enfoque distinto basado en la activación selectiva de parámetros mediante la arquitectura MatFormer, que anida el modelo E2B dentro del E4B, permitiendo escalar dinámicamente el número de parámetros activos según los recursos disponibles. Esta capacidad se complementa con el *Per-Layer Embedding* (PLE) caching (que mantiene las incrustaciones por capa fuera de la memoria operativa) y la carga condicional de los módulos de visión y audio. El resultado práctico es que el modelo E2B, con más de 5 B parámetros totales, puede operar con una carga efectiva de 1,91 B. Ambas variantes soportan entrada multimodal de texto, imagen y audio, con cobertura de más de 140 idiomas y una ventana de contexto de 32 k *tokens*.

Destilados de DeepSeek-R1

El modelo DeepSeek-R1 [63] propone un enfoque basado en RL puro para desarrollar capacidades de razonamiento en LLM, prescindiendo en gran medida de la supervisión humana propia del RLHF. Este método de entrenamiento facilita la aparición de mecanismos analíticos en el modelo, tales como la autorreflexión, la verificación de respuestas y la adaptación de estrategias. Sin embargo, el volumen de parámetros de la versión original (671 B parámetros totales) excede la capacidad de la plataforma objetivo.

Para posibilitar la ejecución local de este tipo de sistemas, se recurre a técnicas de destilación de conocimiento (Sección 2.5.2). A continuación se describen las variantes destiladas basadas en las familias Qwen2.5 y Llama 3.1, que se ajustan a las restricciones de recursos de este trabajo.

- **Variantes basadas en Qwen (1,5B y 7B):** las variantes destiladas construidas sobre la arquitectura fundacional de la familia Qwen2.5 (específicamente la versión matemática) se distribuyen en tamaños de 1,5 y 7 mil millones de parámetros. El modelo de 1,5B presenta unos requerimientos de memoria muy reducidos, compatibles con la plataforma objetivo incluso en precisión de 16 bits. Por su parte, la variante de 7B ofrece un mayor rendimiento en tareas de razonamiento analítico, aunque sus dimensiones requieren la aplicación de cuantización de 4 bits para reducir su huella en memoria a un rango de 4 a 5 GB [159].
- **Variante basada en Llama 3.1 (8B):** la metodología de destilación de DeepSeek-R1 se ha aplicado asimismo sobre el modelo Llama 3.1 de 8 mil millones de parámetros. Esta arquitectura, muy similar a la descrita en la Sección 2.8.2, consiste en un Transformer autorregresivo con GQA y soporte nativo para ventanas de contexto de hasta 128.000 *tokens* [160]. Dado que este modelo se aproxima al umbral superior de la plataforma

objetivo, su despliegue requiere de forma indispensable la aplicación de cuantización a 4 bits.

Familia EuroLLM

El proyecto EuroLLM [161], al igual que las iniciativas nacionales descritas en la Sección 2.1, constituye una iniciativa respaldada por la Unión Europea cuyo propósito es desarrollar una familia de LLM de pesos abiertos (licencia Apache 2.0) altamente optimizados para la comprensión y generación de texto en las lenguas oficiales de la Unión Europea, mitigando así la dependencia de los modelos anglocéntricos. En el contexto de este TFG, acotado por las restricciones de memoria de la plataforma objetivo, se analizan tres variantes de esta familia: EuroLLM-1.7B [162], EuroMoE-2.6B [163] y EuroLLM-9B [164].

- **EuroLLM-1.7B:** la variante de 1,7 mil millones de parámetros emplea una arquitectura Transformer densa tradicional de solo decodificador. Cuenta con un total de 1,657 mil millones de parámetros (1,133 mil millones excluyendo los *embeddings*), distribuidos en 24 capas. El mecanismo de atención utiliza GQA con 16 cabezas de consulta y 8 de clave-valor, incorporando RoPE con una frecuencia base ($\Theta = 10,000$). Emplea SwiGLU y normalización RMSNorm previa a cada capa, con una longitud de secuencia nativa de 4.096 *tokens*. El modelo fue preentrenado sobre 4 billones de *tokens*. Debido a su reducido número de parámetros, puede ejecutarse con cierta holgura en la plataforma objetivo, permitiendo su despliegue en formato de 16 bits sin comprometer el espacio reservado para el sistema operativo ni la caché KV [161], [162].
- **EuroMoE-2.6B:** consta de 2,61 mil millones de parámetros totales, estructurados en 24 capas con un esquema de 64 expertos totales, de los cuales únicamente se activan 8 por cada *token*. En consecuencia, los parámetros activos durante el pase hacia adelante se reducen a tan solo 0,6 mil millones, disminuyendo drásticamente la latencia y la demanda computacional. Arquitectónicamente, incorpora GQA (8 cabezas de consulta y 2 de clave-valor), RMSNorm, SwiGLU y RoPE con un valor Θ ajustado a 500.000, con una ventana de contexto de 4.096 *tokens*. El modelo fue preentrenado con un volumen de 8 billones de *tokens*. Su elevada dispersión lo convierte en una opción de interés para el despliegue en *hardware* restringido [163].
- **EuroLLM-9B:** el modelo de mayor envergadura es un Transformer denso compuesto por 9,154 mil millones de parámetros (8,105 mil millones excluyendo incrustaciones), distribuidos en 42 capas con una dimensión oculta (*hidden size*) en sus redes *feed-forward* de 12.288. Implementa GQA con 32 cabezas de consulta y 8 de clave-valor, RoPE ($\Theta = 10,000$), SwiGLU y RMSNorm, con una longitud de secuencia de 4.096 *tokens*. Su entrenamiento base se ejecutó sobre 4 billones de *tokens* [164]. Dado su elevado número de parámetros, requiere la aplicación de cuantización previa a su ejecución en la plataforma objetivo.

Familia Salamandra (ALIA)

Los modelos Salamandra emplean una estructura Transformer autorregresiva de solo decodificador a la que incorporan mejoras sustanciales respecto al diseño clásico. Entre ellas destacan la omisión de los términos de sesgo (*biases*) para mejorar la estabilidad del entrenamiento, el uso de RoPE con una frecuencia base de 10.000, la sustitución de ReLU por SwiGLU como función de activación, y la normalización RMSNorm. Asimismo,

integran *Flash Attention* para optimizar la eficiencia computacional y mitigar el consumo de memoria. En lo relativo a la atención, el modelo de 2 B utiliza atención multicabezal estándar, mientras que las versiones de 7 B y 40 B implementan GQA con 8 grupos, lo que favorece una inferencia más rápida y reduce la sobrecarga de la caché KV [71].

Respecto a su ventana de contexto, los modelos de 2 B y 7 B admiten hasta 8.192 *tokens*, mientras que la variante de 40 B se limita a 4.096 *tokens*. Un rasgo distintivo de la arquitectura es la adopción de un tokenizador basado en *Byte-Pair Encoding* (BPE) con un extenso vocabulario de 256.000 *tokens*. Este sobredimensionamiento busca potenciar las capacidades multilingües del modelo, permitiendo una codificación más eficiente del texto y mitigando el riesgo de sobresegmentación en lenguas de bajos recursos [71].

Como ya se mencionó en la Sección 2.1, el preentrenamiento de la familia Salamandra se realizó desde cero empleando un corpus altamente curado y extraído exclusivamente de fuentes de acceso abierto, que abarca texto en 35 idiomas europeos así como código de programación. De forma complementaria a los modelos fundacionales, el BSC proporciona versiones *instruction-tuned* afinadas con datos de dominio público y optimizadas para casos de uso conversacionales, aunque se señala que, al igual que con EuroLLM, estas versiones aún no han sido alineadas exhaustivamente con preferencias humanas²⁹ [71].

Familia Ministral 3

La serie Ministral 3 [165], desarrollada por Mistral AI, constituye una familia de modelos de lenguaje densos y eficientes en parámetros, diseñados específicamente para aplicaciones con restricciones de cómputo y memoria. Esta colección se distribuye bajo la licencia permisiva Apache 2.0 y se ofrece en tres tamaños: 3, 8 y 14 mil millones de parámetros. Dentro de las restricciones de la plataforma objetivo, las variantes de 3 B y 8 B resultan adecuadas para la ejecución de inferencia local, dejando margen suficiente para la caché KV y los recursos del sistema operativo.

La familia se basa en una arquitectura Transformer de solo decodificador que integra GQA con 32 cabezas de consulta y 8 de clave-valor, RoPE, SwiGLU y normalización RMSNorm. Una característica notable es el soporte para una extensa ventana de contexto de hasta 256.000 *tokens*, lograda mediante la técnica YaRN y un escalado de temperatura *softmax* basado en la posición. Para la variante de 3 B se emplean además *embeddings* de entrada y salida vinculados (*tied embeddings*), con el fin de evitar que los parámetros de incrustación dominen el tamaño total de la red. Todos los modelos comparten un vocabulario de 131.000 *tokens* e incorporan un codificador visual (*Vision Transformer* de 410 M de parámetros) que dota a la familia de capacidades multimodales nativas para la comprensión de imágenes [165].

El proceso de preentrenamiento destaca por el uso de *Cascade Distillation*, un enfoque iterativo que combina poda y destilación de *logits* a partir de un modelo maestro de mayor escala (Mistral Small 3.1 de 24 B parámetros). Este método permite que las versiones compactas de 3 B y 8 B hereden las capacidades del modelo original con un presupuesto computacional sensiblemente menor, sobre un conjunto de entre 1 y 3 billones de *tokens* [165].

Mistral AI ofrece tres variantes para cada tamaño: un modelo *base* preentrenado para

²⁹En el caso del proyecto ALIA, la fase de alineamiento comenzó en forma de concurso durante la elaboración del TFG: https://alia.gob.es/content/dam/alia/pdf/Bases_y_proteccion_de_datos%E2%80%9393Desafio_ALIA.pdf.

propósitos generales, una versión *instruct* afinada para el seguimiento de instrucciones, y un modelo *reasoning* orientado a la resolución de problemas complejos. La fase de postentrenamiento combina SFT, *Online Direct Preference Optimization* (ODPO) y, en el caso de las variantes de razonamiento, aprendizaje por refuerzo mediante *Group Relative Policy Optimization* (GRPO) [165].

Familia Granite 4.0 (Variantes Híbridas)³⁰

La familia Granite 4.0, desarrollada por IBM y liberada bajo licencia permisiva Apache 2.0, introduce una evolución arquitectónica orientada a maximizar la eficiencia en entornos empresariales y despliegues en el borde (*edge computing*). Estos modelos destacan además por ser los primeros de pesos abiertos en obtener la certificación ISO/IEC 42001:2023 sobre sistemas de gestión de inteligencia artificial, garantizando altos estándares de gobernanza, seguridad y trazabilidad mediante la firma criptográfica de sus pesos [119].

El aspecto técnico más relevante de esta generación es la adopción de una arquitectura híbrida que combina bloques de atención Transformer estándar con capas SSM, específicamente Mamba-2 [167]. Mientras que los Transformers procesan secuencias completas en paralelo, las capas Mamba-2 procesan la información de manera secuencial (*token a token*), comprimiendo el estado y mitigando el crecimiento cuadrático del coste computacional y de la caché KV (Ecuación 2.11). Adicionalmente, esta arquitectura omite el uso de codificación posicional explícita (*No Positional Encoding*, NoPE), lo que reduce la carga computacional y permite soportar ventanas de contexto de hasta 128.000 *tokens* de manera eficiente. Esto supone una reducción en el consumo de memoria superior al 70 % frente a modelos densos de tamaño equivalente [119].

Dentro de las variantes híbridas de Granite 4.0, se distinguen aquellas basadas en MoE y aquellas con arquitectura híbrida densa. Su aplicabilidad en la plataforma objetivo se detalla a continuación:

- **Granite 4.0 H Small (32B total / 9B activos):** este modelo MoE cuenta con 32 mil millones de parámetros totales, de los cuales activa 9 mil millones durante la inferencia. A pesar de su eficiencia computacional, alojar la totalidad de sus pesos en memoria requiere aproximadamente 16 GB incluso bajo cuantización de 4 bits, por lo que excede los límites físicos de la plataforma objetivo.
- **Granite 4.0 H Tiny (7B total / 1B activos):** con cuantización de 4 bits, el modelo ocupa en torno a 4 GB de RAM. Gracias a su naturaleza MoE, activa únicamente 1 mil millones de parámetros por *token*, minimizando la latencia. El uso de capas Mamba-2 comprime adicionalmente la huella de la caché KV, dejando margen suficiente en la plataforma objetivo.
- **Granite 4.0 H Micro (3B):** se trata de un modelo híbrido denso de 3 mil millones de parámetros que, dada su compacidad, puede cargarse en memoria aplicando una cuantización ligera o en precisión de 16 bits en función del sistema operativo subyacente, constituyendo una alternativa eficiente para tareas de baja latencia o llamadas a funciones (*function calling*).
- **Serie Nano (Granite 4.0 H 1B y H 350M):** estas variantes híbridas densas (1,5 mil millones y 350 millones de parámetros, respectivamente) presentan requerimientos

³⁰ Durante la elaboración de este TFG, la familia Granite 4.0 fue sucedida por Granite 4.1 [166].

de memoria muy reducidos, ajustándose de forma nativa a la plataforma objetivo sin necesidad de cuantización.

A partir de las familias descritas, la conformación del conjunto experimental definitivo atiende a dos criterios complementarios. El primero es la **cobertura del espacio arquitectónico**: se ha procurado incluir al menos un representante de cada paradigma vigente en la literatura de SLM, de modo que el análisis posterior permita identificar los *trade-offs* intrínsecos a cada diseño. El segundo es la **restricción operativa del conjunto experimental**: los modelos en torno a los 8 mil millones de parámetros han sido excluidos con el objeto de acotar la duración total de los experimentos y mantener el número de configuraciones en un rango manejable, con independencia de su arquitectura.

Aplicados ambos criterios, el conjunto experimental comprende seis modelos organizados por categoría arquitectónica (véase la Tabla 2.3): Llama 3.2 1B y Llama 3.2 3B, como representantes de la arquitectura densa tradicional de referencia en la literatura (con el valor añadido de que la variante de 1 B admite ejecución sin cuantización, permitiendo explorar el efecto de los pesos nativos); Gemma 3n E2B, como representante de la arquitectura MatFormer con activación selectiva de parámetros, concebido explícitamente para *hardware* de bajo consumo; DeepSeek-R1-Distill-Qwen-1.5B, como representante de los modelos de razonamiento de arquitectura densa y exponente de la soberanía tecnológica china; Ministral 3B, seleccionado por su representatividad en el rango de parámetros de interés y como exponente de la soberanía tecnológica francesa; y Granite 4.0 H Micro, como representante de la arquitectura híbrida SSM+Transformer.

Las familias EuroLLM y Salamandra (ALIA) han sido excluidas del conjunto experimental por no haber completado, en el momento de realización de los experimentos, la fase de alineamiento de sus versiones *instruction-tuned* con preferencias humanas, lo que comprometería la comparabilidad de sus respuestas en las tareas de evaluación previstas.

Tabla 2.3: Conjunto de modelos seleccionados para el análisis experimental, con sus principales características y criterio de inclusión.

Modelo	Desarrollador	Parámetros (activos)	Peso orig.	Arquitectura	Contexto máx.	Criterio selección	de
Llama 3.2 1B	Meta	1,23 B	BF16	Densa	128 k	Referencia científica; pesos nativos	
Llama 3.2 3B	Meta	3,21 B	BF16	Densa	128 k	Referencia científica	
Gemma 3n E2B	Google DeepMind	5 B (1,91 B)	BF16	MatFormer	32 k	Diseño activación selectiva	<i>edge</i> ;
DeepSeek-R1-Distill-Qwen-1.5B	DeepSeek	1,5 B	BF16	Densa (razonamiento)	128 k	Razonamiento; soberanía china	
Ministral 3B	Mistral AI	3 B	BF16	Densa	256 k	Soberanía francesa	
Granite 4.0 H Micro	IBM	3 B	BF16	SSM + Transformer	128 k	Diversidad arquitectónica	

2.8.3. De los análisis de optimización

El análisis del rendimiento de sistemas de inferencia en dispositivos de recursos limitados tiene raíces previas a la era LLM. Los trabajos de Velasco-Montero *et al.* [141], [142], centrados en la inferencia visual para sistemas IoT, establecen un marco

metodológico de referencia para la selección de *software* y modelos en *hardware* embebido, identificando las métricas *hardware* relevantes (frecuencia, voltaje, temperatura de operación, y uso de memoria, CPU y energía), cuya monitorización resulta determinante para la caracterización rigurosa del comportamiento del sistema. En la misma línea, Benoit-Cattin *et al.* [168] demuestran empíricamente sobre una Raspberry Pi 4B que el efecto térmico (*thermal throttling*) aparece en cuestión de segundos bajo carga computacional elevada, y que la temperatura ambiente es un parámetro crítico para la reproducibilidad cuando no se dispone de refrigeración activa. Aunque el trabajo versa sobre inferencia visual con CNNs, su relevancia metodológica para el presente TFG es directa al compartir la misma plataforma *hardware* (aunque una versión inferior), poniendo de manifiesto que el control térmico es un requisito ineludible para garantizar experimentos reproducibles en inferencia prolongada.

En el ámbito específico de los LLM, los análisis de optimización presentan una naturaleza multidimensional que dificulta la comparación entre trabajos. Los principales ejes de variación son el *hardware* empleado, las técnicas de optimización aplicadas, las métricas de evaluación consideradas (o la combinación de estas) y, de forma crítica, la disponibilidad del *software* experimental para su reproducción. Desde una perspectiva *hardware* amplia, Li *et al.* [128] proporcionan una revisión sistemática del impacto de distintos tipos de aceleradores (GPU, FPGA, ASIC, PIM) sobre la eficiencia de la inferencia en términos absolutos de rendimiento (tokens/segundo) como energéticos (tokens/julio), si bien no menciona los motores de inferencia empleados.

En lo que respecta a los análisis empíricos en *hardware* edge, Chen *et al.* [140] proponen ELIB, el marco comparativo más relevante en cuanto a la metodología y objetivos de este TFG. Su contribución principal es la introducción de dos métricas complementarias que capturan las limitaciones intrínsecas de la inferencia descritas en la Sección 2.7.2: una limitada por la capacidad de cómputo (FLOPS) y otra limitada por el ancho de banda de memoria (MBU, *Memory Bandwidth Utilization*), definida como el cociente entre el ancho de banda empleado y el máximo ancho de banda teórico. Junto a estas, el marco incorpora métricas temporales clásicas (tokens/segundo, TTFT y TTLM) y de precisión mediante el índice de perplejidad. Empleando este marco, los autores concluyen que para maximizar el MBU se requiere la combinación de tres estrategias: el aumento del *prompt batch size*, la optimización de la ventana de contexto y la reducción del tamaño de la caché KV. En cuanto a su reproducibilidad, pese a mencionarse como proyecto de código abierto de libre compartición, el *software* no se encuentra disponible públicamente; durante el desarrollo de este trabajo se localizó una versión en progreso³¹ que fue posteriormente retirada por los propios autores tras ser contactados al respecto, sin que se haya podido obtener el código fuente completo. A pesar de esta limitación para su implementación práctica, el estudio resulta muy exhaustivo en su explicación conceptual, lo que permite comprender la lógica del sistema desde una perspectiva de alto nivel. En cuanto a su alcance, el trabajo no contempla el impacto energético en términos de *tokens/julio*, no analiza el efecto de los distintos motores de inferencia especializados, restringe la evaluación de calidad al índice de perplejidad sin considerar la exactitud de las respuestas, y no monitoriza variables *hardware* adicionales como temperatura, frecuencia de operación o uso de SWAP.

Arya y Simmhan [169] realizan una evaluación sistemática de la inferencia de LLM sobre la NVIDIA Jetson Orin AGX (64 GB de RAM compartida CPU/GPU), variando el *batch size*, la longitud de secuencia, el nivel de cuantización y los modos de potencia sobre cuatro modelos de entre 2,7B y 32,8B parámetros. Entre sus hallazgos más relevantes destaca que, a diferencia

³¹<https://github.com/TFG-yisuscc/ELIB>

de lo observado en GPUs de servidor, la cuantización INT8 en modelos pequeños incrementa la latencia respecto a FP16 debido al overhead de descuantización, mientras que en modelos de mayor tamaño el impacto sobre la latencia es mínimo. El trabajo mide además el consumo energético en términos absolutos, si bien no lo normaliza por *token* generado. No obstante, emplea PyTorch sin motores de inferencia especializados, lo que limita la comparabilidad de sus resultados con los obtenidos mediante motores dedicados como Ollama o llama.cpp.

En una línea similar, pero con una perspectiva mixta, Sobhani *et al.* [145] caracterizan el rendimiento y consumo energético de redes neuronales y SLM sobre dos plataformas *edge* representativas (Raspberry Pi y NVIDIA Jetson Nano) y dos aceleradores no aptos para LLM (Intel Neural Compute Stick y Google Coral USB), analizando las compensaciones entre puntuación F1, tiempo de inferencia, potencia y uso de memoria. Sin embargo, el trabajo emplea LiteRT como framework de referencia sin considerar motores de inferencia especializados para LLM, y no diversifica suficientemente la selección de modelos de lenguaje evaluados (solamente TinyBert y phi-2), lo que limita la generalidad de sus conclusiones para el caso específico de los SLM cuantizados.

Finalmente, el antecedente empírico más directo del presente TFG es el trabajo de Husom *et al.* [127], que evalúa 28 modelos cuantizados del repositorio de Ollama desplegados sobre una Raspberry Pi 4 con 4 GB de RAM, midiendo simultáneamente eficiencia energética (tokens/julio), latencia de inferencia y precisión de salida sobre cinco conjuntos de datos estandarizados. Es el único trabajo que combina métricas energéticas normalizadas con evaluación de calidad basada en tareas reales. Sin embargo, se circunscribe a un único motor de inferencia sobre CPU, no monitoriza variables *hardware* adicionales durante la inferencia (frecuencia, temperatura, uso de SWAP ...) y no contempla el impacto de factores térmicos sobre la reproducibilidad, aspecto que, como se mencionó anteriormente, se ha demostrado ser determinante en este tipo de evaluaciones [141], [142], [168].

Del análisis conjunto de estos trabajos se desprende una clara fragmentación en la literatura existente. Ningún estudio integra simultáneamente el uso de motores de inferencia especializados, la monitorización exhaustiva de variables *hardware* (incluyendo temperatura y frecuencia), la evaluación energética normalizada y el efecto de aceleradores comerciales. A ello se añade una laguna transversal a todos los trabajos revisados: ninguno caracteriza el coste temporal ni energético de la fase de carga del modelo, esto es, el intervalo comprendido entre la invocación del motor y el momento en que el modelo queda listo para procesar el primer *prompt*. Este dato resulta de especial interés en plataformas de recursos limitados, donde el elevado consumo de memoria de los SLM hace que la carga bajo demanda, con liberación del modelo cuando no se encuentra en uso, constituya una estrategia potencialmente relevante cuya evaluación requiere conocer dicho coste. Esta carencia evidencia la necesidad de un enfoque como el planteado en el presente TFG, que permita caracterizar de forma rigurosa el comportamiento de los SLM en entornos *edge* bajo condiciones realistas.

3. Metodología

En este capítulo se presenta la metodología que se ha seguido para abordar el problema de estudio de manera sistemática y objetiva. En primer lugar, se detallan las cuestiones de investigación planteadas y agrupadas en experimentos. En segundo lugar, se describen las métricas empleadas ampliando lo expuesto en la Sección 2.7.4. Tras ello, se abordan el diseño de los sistemas de medida y visualización. Después se presentará el diseño experimental seguido en la elaboración de este trabajo y finalmente se detallará la planificación y los recursos requeridos para este proyecto.

3.1. Objetivos metodológicos y cuestiones de investigación

El presente estudio adopta un enfoque empírico de carácter cuantitativo, basado en el diseño y la ejecución de experimentos controlados sobre una Raspberry Pi 5 de 8 GB como representativa de una plataforma de computación en el borde. El objeto de análisis es el comportamiento del conjunto de modelos de lenguaje pequeños (descritos en la Tabla 2.3) durante la inferencia, caracterizado mediante la medición sistemática de variables de rendimiento, consumo energético, temperatura y estado térmico, y aprovechamiento de recursos *hardware*. Las variables independientes son el modelo empleado, el nivel de cuantización, el tamaño de la ventana de contexto, el *batch size* de *prefill*, el motor de inferencia y la condición de refrigeración. Las variables dependientes o métricas se describen posteriormente en la Sección 3.1.1.

El punto de partida para la formulación de las cuestiones de investigación es el conjunto de carencias detectadas en la literatura revisada en la Sección 2.8.3. Tal y como se concluye en dicha sección, ninguno de los trabajos previos analizados integra simultáneamente el uso de motores de inferencia especializados para LLM, la monitorización exhaustiva de variables *hardware* durante la inferencia, la evaluación energética normalizada y la medición del coste energético de la fase de carga del modelo. Las cuestiones de investigación que se formulan a continuación se articulan precisamente en torno a estas lagunas:

- **CI-01.** ¿Cómo varía el desempeño comparativo de los SLM evaluados sobre la Raspberry Pi 5 en condiciones nominales de operación, en términos de rendimiento temporal, eficiencia energética y consumo y aprovechamiento de los recursos *hardware*?
- **CI-02.** ¿En qué medida el nivel de cuantización de un modelo afecta simultáneamente al rendimiento temporal, la eficiencia energética, la calidad de las respuestas y el consumo y aprovechamiento de los recursos *hardware*, en particular de la memoria? ¿Es posible identificar un nivel de cuantización que constituya un equilibrio entre eficiencia computacional y calidad generativa en plataformas embebidas de recursos limitados?
- **CI-03.** ¿Cómo afecta el tamaño de la ventana de contexto al rendimiento temporal, el consumo de memoria y la eficiencia energética de la inferencia, en los distintos modelos evaluados?
- **CI-04.** ¿Cómo afecta el *batch size* de *prefill* a la duración, el consumo de memoria y la

eficiencia energética de dicha fase, y en qué medida su ajuste permite optimizar el *throughput* global de la inferencia en condiciones de recursos limitados?

- **CI-05.** ¿Qué diferencias de rendimiento, consumo de cómputo y eficiencia energética introduce el motor Ollama frente a llama.cpp en condiciones de configuración equivalentes?
- **CI-06.** ¿Qué ganancia de rendimiento y eficiencia energética aporta el acelerador *hardware* Hailo-10H , operado mediante hailo-ollama, frente a la ejecución exclusiva en CPU mediante llama.cpp, para los modelos compatibles con la plataforma?
- **CI-07.** ¿Qué diferencias cuantificables introduce la refrigeración activa, frente a la operación en convección natural, sobre el rendimiento, la eficiencia energética y el consumo y aprovechamiento de cómputo y memoria?

Con carácter transversal al conjunto de series experimentales, se plantea además la siguiente cuestión orientadora, que guía la arquitectura del propio sistema de medida descrito en la Sección 3.2.1:

- **CI-08.** ¿Cuál es el coste temporal y energético de la fase de carga del modelo a lo largo de las distintas configuraciones evaluadas (exceptuando el uso de acelerador)?
- **CI-09.** ¿Es posible definir un marco experimental unificado y reproducible que integre de forma simultánea métricas temporales, energéticas, y térmicas, para la evaluación comparativa de SLM en plataformas de computación en el borde?

La vinculación entre cada cuestión de investigación y la serie experimental correspondiente queda recogida en la Tabla 3.1. En dicha tabla se incluye también la configuración de referencia común a todas las series, cuya justificación se detalla en la Sección 3.3.2.

Hipótesis de trabajo

A partir de las cuestiones de investigación anteriores y del estado del arte revisado en la Sección 2.8, se formulan las siguientes hipótesis de trabajo, contrastadas empíricamente en el Capítulo 4.

H-E0a (CI-01). Los seis modelos evaluados serán utilizables en modo interactivo bajo refrigeración activa, con rendimientos heterogéneos determinados principalmente por el tamaño del modelo y la presión sobre el ancho de banda de memoria.

H-E0b (CI-07). La ausencia de refrigeración activa provocará estrangulamiento térmico sostenido que reducirá el *throughput* de forma significativa y expulsará del rango de usabilidad a los modelos de mayor coste computacional.

H-E1 (CI-02). La cuantización a 4 bits por peso constituye el punto de equilibrio óptimo en la plataforma: alivia el cuello de botella de ancho de banda de memoria maximizando el *throughput* con una degradación de calidad inferior al 3–5 % [126], mientras que la cuantización a 8 bits o superior resulta contraproducente al incrementar la presión de memoria sin mejora proporcional de la figura de mérito.

H-E2 (CI-03). Ampliar la ventana de contexto producirá una caída suave y monótona del *throughput* sin afectar a la huella de RAM ni al TTFT, dado que el coste de la caché KV es marginal frente al de los pesos cuantizados en los rangos evaluados.

- H-E3 (CI-04).** Incrementar el batch size de prefill mejorará el *throughput* global al explotar el paralelismo de cómputo de los cuatro núcleos del BCM2712, cuya fase de prefill está limitada por capacidad de cómputo (*compute-bound*).
- H-E4 (CI-05).** llama.cpp ofrecerá una latencia de primer token significativamente inferior a la de Ollama, cuya arquitectura cliente-servidor introduce sobrecarga de inicialización por petición, con *throughput* de decodificación comparable entre ambos motores.
- H-E5 (CI-06).** El acelerador Hailo-10H proporcionará un *speedup* significativo en *throughput* y una mejora de eficiencia energética por token respecto a la CPU, gracias a su arquitectura de cómputo dedicada y su memoria independiente.

Tabla 3.1: Matriz experimental del estudio. La configuración de referencia para todas las series con refrigeración activa es: motor llama.cpp, *batch size* = 512, contexto = 4 096 tokens, temperatura = 0,0, semilla = 42, 15 *prompts* del conjunto *instruction_following_eval* [170]. La serie E5 emplea un contexto de 2 048 tokens por restricción del compilado del modelo HEF¹.

Serie CI		Variable estudiada	Niveles evaluados	Runs	Refrigeración
E0	CI-01, CI-07	Modelo; condición de refrigeración	6 modelos $\times$ 2 condiciones (con/sin ventilador)	12	Con y sin ventilador
E1	CI-02	Nivel de cuantización	Q3_K_M, Q4_0, Q4_K_M, Q5_K_M, Q8_0; BF16 adicional para Llama 3.2 1B	31	Con ventilador
E2	CI-03	Tamaño de contexto	512, 1 024, 2 048, 4 096, 5 120 tokens	30	Con ventilador
E3	CI-04	<i>Batch size</i>	128, 256, 512, 1 024, 2 048	30	Con ventilador
E4	CI-05	Motor de inferencia	llama.cpp vs. Ollama (paralela a E0)	12	Con ventilador
E5	CI-06	Acelerador hardware	llama.cpp (CPU) vs. hailo-ollama (NPU Hailo-10H)	2	Con ventilador

¹Los modelos desplegados en el Hailo-10H se compilan en formato HEF (*Hailo Executable Format*) con los parámetros de contexto y *batch* fijados en tiempo de compilación. El valor de 2 048 tokens corresponde al empleado en las guías de referencia del fabricante [171]. En consecuencia, los parámetros *batch_size* y *context_size* configurados en el sistema de medida se registran en los metadatos del experimento pero no tienen efecto en tiempo de ejecución cuando se utiliza hailo-ollama (véase RN-09 en el documento de requisitos de MonitorSystem).

3.1.1. Métricas empleadas

La caracterización del comportamiento de los modelos se sustenta en un conjunto de métricas que abarca cinco dimensiones: rendimiento de inferencia, coste energético, eficiencia computacional, aprovechamiento de memoria y calidad. Para facilitar la comparación entre modelos, se propone además una Figura de Mérito que integra las cuatro primeras dimensiones anteriormente citadas. Las métricas de uso extendido en la literatura ya fueron definidas formalmente en la Sección 2.7.4, por lo que aquí se enuncian de forma sintética; las métricas propias de este trabajo se detallan con su definición formal.

Métricas de rendimiento de inferencia.

- **Throughput de generación** (tokens/s): número de *tokens* generados por unidad de tiempo en la fase de decodificación. Definida en la Sección 2.7.4.
- **TTFT** (*Time To First Token*, s): latencia entre el envío del *prompt* y la emisión del primer *token*. Definida en la Sección 2.7.4.
- **TTLM** (*Time To Load Model*, s): tiempo de carga del modelo, comprendido entre la invocación del motor y el momento en que el modelo queda listo para procesar el primer *prompt*. Definida en la Sección 2.7.4.

Métricas de coste energético. La potencia se obtiene del sensor interno de la Raspberry Pi 5 mediante el comando `vcgencmd pmic_read_adc`, muestreada con período `hardware_period`. Esta lectura no contempla el consumo de los HAT ni de los periféricos USB, lo que constituye una limitación discutida en la Sección 3.2.3. En la serie E5 se prevé complementar esta medida con un instrumento de pared externo.

- **Potencia media del run** ($\bar{P}_{run}$, W): media aritmética de las muestras de potencia interna registradas a lo largo de la ejecución completa, incluidas las fases de carga, *prefill*, decodificación e intervalos entre *prompts*.
- **Energía total del run** (E_{run} , J): producto de la potencia media por la duración total de la ejecución,

$$E_{run} = \bar{P}_{run} \cdot T_{run} \quad (3.1)$$

donde T_{run} es el intervalo entre los instantes de inicio y fin del *run*.

- **Eficiencia energética** (tokens/J): número de *tokens* generados por julios consumidos, calculada como

$$\varepsilon = \frac{\text{tokens}}{J} = \frac{\text{tokens/s}}{\bar{P}_{run}} \quad (3.2)$$

esto es, el cociente entre el *throughput* medio y la potencia media del *run*.

- **Coste por fase** (J y W; absoluto y relativo): desglose del coste energético entre las fases de *prefill* y decodificación. La separación temporal de ambas fases se obtiene de los tiempos reportados por el motor (`prompt_eval_duration` y `eval_duration`). Dado que la duración del *prefill* es habitualmente inferior al período de muestreo *hardware*, la resolución del sensor resulta insuficiente para integrar la potencia instantánea sobre dicha fase con fiabilidad. Por ello, la energía de cada fase se estima de forma proporcional a su duración, empleando la potencia media del *run* como aproximación,

$$E_{\text{fase}} \approx \bar{P}_{run} \cdot T_{\text{fase}}, \quad \text{fase} \in \{\text{prefill}, \text{decode}, \text{load}\} \quad (3.3)$$

y se reporta tanto en valor absoluto como en porcentaje sobre el total. Esta aproximación se asume de forma explícita como limitación metodológica derivada de la granularidad del sensor interno.

Métricas de eficiencia computacional. Las métricas anteriores no distinguen entre un *throughput* elevado por mérito del modelo y uno penalizado por la infrautilización de la CPU debida al estrangulamiento térmico (*throttling*) o a la limitación por memoria. Las tres métricas siguientes, propias de este trabajo, cuantifican el cómputo efectivamente realizado por la CPU, ajustado por throttling.

- **Trabajo CPU efectivo** (W_{CPU} , core-s): cómputo consumido medido en núcleos-segundo equivalentes a frecuencia nominal. Es análogo al concepto de persona-hora,

$$W_{CPU} = \int_0^T \frac{f(t)}{f_{nom}} \cdot \frac{U(t)}{100} \cdot N_{cores} dt \quad (3.4)$$

donde $f(t)$ es la frecuencia instantánea por muestra *hardware*, $f_{nom} = 2,4$ GHz es la frecuencia nominal del SoC de la Raspberry Pi 5, $U(t)$ es el porcentaje de uso de CPU y $N_{cores} = 4$ es el número de núcleos. Un *run* de 100 s con los cuatro núcleos al 100 % a frecuencia nominal produce $W_{CPU} = 400$ core-s; el mismo *run* bajo throttling sostenido a la mitad de la frecuencia nominal produce $W_{CPU} = 200$ core-s.

- **Eficiencia CPU** (η_{CPU} , adimensional): cociente entre el trabajo efectivo realizado y el máximo teórico durante el *run*,

$$\eta_{CPU} = \frac{W_{CPU}}{N_{cores} \cdot T} \quad \eta_{CPU} \in [0, 1] \quad (3.5)$$

Un valor de 1,0 indica que los cuatro núcleos operaron al 100 % a frecuencia nominal durante todo el *run*; un valor de 0,5 indica que solo se aprovechó la mitad del cómputo disponible, por efecto del throttling, los periodos de inactividad o la contención de memoria.

- **Trabajo por token** (W_{token} , core-s/token): normaliza el trabajo efectivo por el número de *tokens* generados, como análogo consciente del cómputo a la latencia por *token*,

$$W_{token} = \frac{W_{CPU}}{N_{tokens}} \quad (3.6)$$

Métrica de aprovechamiento de memoria

- **MBU** (*Memory Bandwidth Utilization*, adimensional): Propuesta por Chen *et al.* [140]. Se define como la fracción del ancho de banda de memoria teórico aprovechada durante la decodificación, calculada como

$$MBU = \frac{BW_{achieved}}{BW_{peak}} \quad BW_{achieved} = \frac{M_{model} + M_{KV}}{TPOT} \quad (3.7)$$

donde $TPOT = t_{decode}/N_{tokens}$ es el tiempo medio por *token* de salida, $BW_{peak} = 17$ GB/s es el ancho de banda teórico máximo de la Raspberry Pi 5 [172], $M_{model} = N_{params} \times \text{bpw}/8$ es el tamaño en bytes de los pesos del modelo para la precisión bpw (bits por peso), y M_{KV} es el tamaño de la caché KV descrito según la Ecuación 2.11. Por su definición, esta métrica asume el comportamiento típico de un modelo denso: en cada paso de decodificación se transfieren a memoria todos los

pesos del modelo y la caché KV acumulada. Esta hipótesis no se cumple necesariamente en modelos cuya arquitectura diverge de las densas tradicionales, por lo que los valores de MBU obtenidos para dichos modelos deben interpretarse con cautela. Es el caso de Granite 4.0-H-Micro (híbrido SSM+Transformer) y Gemma 3n (MatFormer).

Para corregir estas desviaciones se introducen las siguientes modificaciones a la Ecuación 3.7:

$$\text{MBU}_{\text{corr}} = \frac{1}{\text{BW}_{\text{peak}}} \cdot \frac{M_{\text{model}}^{\text{act}} + M_{\text{KV}}^{\text{attn}} + M_{\text{state}}}{\text{TPOT}} \quad (3.8)$$

donde cada término modela el tráfico de memoria realmente transferido por *token* en una arquitectura no densa:

- $M_{\text{model}}^{\text{act}} = N_{\text{params}}^{\text{act}} \times \text{bpw}/8$ emplea únicamente los parámetros realmente activados por *token* ($N_{\text{params}}^{\text{act}} \leq N_{\text{params}}$), relevante en modelos dispersos como las variantes de Granite con mezcla de expertos (MoE) o Gemma 3n con MatFormer. En modelos densos $N_{\text{params}}^{\text{act}} = N_{\text{params}}$, como es el caso de Granite 4.0-H-Micro.
- $M_{\text{KV}}^{\text{attn}} = 2 L_{\text{attn}} n_{\text{kv}} d_h S b_{\text{kv}}$ es la caché KV, contabilizada *solo* sobre las L_{attn} capas de atención del modelo (y no sobre el total de capas), donde n_{kv} es el número de cabezas clave-valor, d_h la dimensión por cabeza, S la longitud de contexto y b_{kv} los bytes por elemento de la caché. Este término crece linealmente con S .
- $M_{\text{state}} = L_{\text{ssm}} (d_{\text{inner}} d_{\text{state}} + d_{\text{conv}} (d_{\text{inner}} + 2 n_{\text{groups}} d_{\text{state}})) \times \text{bpw}_{\text{state}}/8$ es el tamaño del estado recurrente de las L_{ssm} capas de espacio de estados (Mamba-2 en Granite), donde d_{inner} es la dimensión interna del bloque, d_{state} el tamaño del estado, d_{conv} el núcleo convolucional, n_{groups} el número de grupos y $\text{bpw}_{\text{state}}$ la precisión del estado. El primer sumando corresponde al estado del modelo de espacio de estados y el segundo a la ventana convolucional causal, que opera sobre las proyecciones x , B y C concatenadas. A diferencia de la caché KV, este estado es de tamaño *fijo* e independiente de la longitud de contexto, lo que constituye la principal ventaja de las arquitecturas híbridas en escenarios de contexto largo.²

Esta descomposición aditiva sustituye la caché KV monolítica de la Ecuación 3.7 por los dos mecanismos de memoria que coexisten en una arquitectura híbrida: una caché KV reducida (al provenir de un número menor de capas de atención) y un estado recurrente de coste constante. En el caso límite de un modelo denso con atención completa ($L_{\text{ssm}} = 0$, $L_{\text{attn}} = L$ y $N_{\text{params}}^{\text{act}} = N_{\text{params}}$), se tiene $M_{\text{state}} = 0$ y la Ecuación 3.8 se reduce a la definición original de Chen *et al.*

Cabe señalar que, debido a limitaciones en la instrumentación del banco de pruebas, el término M_{state} no se registró durante la ejecución, sino que se calculó *a posteriori* a partir de los parámetros arquitectónicos del modelo extraídos de su fichero GGUF ($L_{\text{ssm}} = 36$, $d_{\text{inner}} = 4096$, $d_{\text{state}} = 128$, $d_{\text{conv}} = 4$, $n_{\text{groups}} = 1$) y de la precisión del estado verificada en los registros del motor de inferencia. Al ser M_{state} un valor fijo, independiente tanto del contexto como de la cuantización, este cálculo posterior es

²La precisión del estado recurrente es independiente de la cuantización de los pesos. Verificado en los registros de carga de llama.cpp: los tensores del estado SSM (`ssm_a`, `ssm_d`, `ssm_conv1d`, `ssm_dt`) se mantienen en f32 ($\text{bpw}_{\text{state}} = 32$) en las cuantizaciones Q4_K_M, Q4_0 y bf16; por ello M_{state} es constante entre cuantizaciones y solo $M_{\text{model}}^{\text{act}}$ varía con bpw .

exacto y no introduce incertidumbre adicional en la métrica. Para Granite 4.0-H-Micro, $M_{\text{state}} \approx 74,4$ MiB.

Métricas de calidad generativa.

- **Perplejidad** (*perplexity*, PPL): medida de calidad generativa definida en la Sección 2.7.4. Cuantifica la incertidumbre media del modelo al predecir una secuencia de tokens donde una perplejidad menor indica que el modelo asigna mayor probabilidad a la secuencia, interpretándose como mejor ajuste al corpus.

La perplejidad es inherentemente dependiente del tokenizador: dos modelos con vocabularios distintos producen valores no comparables entre sí, ya que la longitud de la secuencia en *tokens* y la distribución de probabilidades están condicionadas por la segmentación léxica de cada familia de modelos. Por ello, esta métrica es válida únicamente para comparaciones dentro de modelos que comparten el mismo tokenizador, como las distintas cuantizaciones de un mismo modelo en la serie E1. Esta limitación es la razón por la que la perplejidad no se incorpora como eje de la FoM (Ecuación 3.10) y se analiza de forma independiente.

Figuras de mérito Con el fin de sintetizar las cuatro dimensiones de eficiencia en un único valor escalar que permita ordenar y comparar configuraciones, se definen dos figuras de mérito de estructura común: una *completa*, aplicable a las series E0–E4, y una *reducida*, aplicable a la serie E5, donde la ausencia de ejecución en CPU impide calcular la MBU y la η_{CPU} . La perplejidad, al ser autorreferencial y no comparable entre modelos distintos, se analiza como dimensión independiente en los resultados de la serie E1.

Ambas FoM emplean la *media geométrica ponderada* de sus ejes normalizados, cuya propiedad clave es que, si cualquier eje tiende a cero, el valor agregado tiende también a cero, penalizando desempeños desequilibrados. Cada eje se normaliza respecto a una configuración de referencia fija, lo que hace la FoM estable ante la incorporación de nuevas configuraciones. Para corregir el sesgo intrínseco de los ejes de rendimiento y eficiencia energética a favor de modelos pequeños, ambos se ponderan linealmente por el número de parámetros activos N^{act} , transformando tokens/s y tokens/J en parámetros·token/s y parámetros·token/J respectivamente. MBU y η_{CPU} , al ser cocientes adimensionales acotados en $[0, 1]$, no se ponderan por N .

FoM completa (series E0–E4). La configuración de referencia es la ejecución con llama.cpp del modelo Llama 3.2 3B, cuantización Q4_K_M, contexto 4096 *tokens*, *batch size* 512, semilla 42 y temperatura 0,0. Los cuatro ejes normalizados (y por ende adimensionales) son:

$$\tilde{T} = \frac{N^{act} \cdot T}{N_{\text{ref}}^{act} \cdot T_{\text{ref}}}, \quad \widetilde{\text{MBU}} = \frac{\text{MBU}}{\text{MBU}_{\text{ref}}}, \quad \tilde{\eta} = \frac{\eta_{CPU}}{\eta_{CPU,\text{ref}}}, \quad \tilde{\varepsilon} = \frac{N^{act} \cdot \varepsilon}{N_{\text{ref}}^{act} \cdot \varepsilon_{\text{ref}}} \quad (3.9)$$

donde T se expresa en $\text{tokens} \cdot \text{s}^{-1}$ (véase la Ec. 2.15), ε en $\text{tokens} \cdot \text{J}^{-1}$ (véase la Ec. 2.19), y tanto MBU como η_{CPU} son de por sí adimensionales y acotadas en $[0, 1]$ (véanse las Ecs. 3.8 y 3.5). La ponderación por N^{act} en $\tilde{T}$ y $\tilde{\varepsilon}$ corrige el sesgo a favor de modelos de menor tamaño y no altera la adimensionalidad del cociente.

La figura de mérito completa, definida mediante un conjunto de pesos $\{w_i\}_{i=1}^4$ que permiten ajustar la relevancia relativa de cada eje, adopta en este trabajo pesos iguales ($w_i = 1 \forall i$):

$$\text{FoM}_{\text{full}} = \left(\tilde{T}^{w_1} \cdot \widetilde{\text{MBU}}^{w_2} \cdot \tilde{\eta}^{w_3} \cdot \tilde{\varepsilon}^{w_4} \right)^{1/\sum w_i} \xrightarrow{w_i=1} \sqrt[4]{\tilde{T} \cdot \widetilde{\text{MBU}} \cdot \tilde{\eta} \cdot \tilde{\varepsilon}} \quad (3.10)$$

FoM reducida (serie E5). La normalización se ancla para cada modelo a su propia ejecución en CPU con llama.cpp, de modo que la FoM reducida cuantifica el factor de aceleración del Hailo-10H respecto a la CPU. El factor N se cancela en la comparación intra-modelo pero se conserva en la definición para permitir derivar a posteriori la comparación ponderada entre los dos modelos evaluados:

$$\text{FoM}_{\text{red}} = \left(\tilde{T}_5^{w_1} \cdot \tilde{\varepsilon}_5^{w_2} \right)^{1/(w_1+w_2)} \xrightarrow{w_1=w_2} \sqrt{\tilde{T}_5 \cdot \tilde{\varepsilon}_5}, \quad \text{donde} \quad \tilde{T}_5 = \frac{T}{T_{\text{cpu},m}}, \quad \tilde{\varepsilon}_5 = \frac{\varepsilon}{\varepsilon_{\text{cpu},m}} \quad (3.11)$$

Bajo este anclaje, un valor $\text{FoM}_{\text{red}} > 1$ indica ventaja global del acelerador y un valor $\text{FoM}_{\text{red}} < 1$ indica ventaja de la CPU. A modo ilustrativo: si el acelerador alcanza $\tilde{T}_5 = 1,20$ (un 20 % más rápido que la CPU) y $\tilde{\varepsilon}_5 = 1,25$ (un 25 % más eficiente energéticamente), la FoM resulta $\sqrt{1,20 \times 1,25} \approx 1,22 > 1$, con ventaja para el acelerador. Si, por el contrario, el acelerador es más lento ($\tilde{T}_5 = 0,53$) pero más eficiente energéticamente ($\tilde{\varepsilon}_5 = 1,30$), la FoM resulta $\sqrt{0,53 \times 1,30} \approx 0,83 < 1$, con ventaja para la CPU pese a la mejora energética parcial del acelerador³

Criterio de usabilidad. De forma complementaria, se define un indicador binario que señala si una configuración supera un umbral mínimo de usabilidad en uso interactivo. A partir de una velocidad de lectura de referencia de 3 palabras/s [22] y un factor de conversión palabra→token calculado empíricamente sobre el corpus de respuestas de cada modelo, se obtiene un umbral T_{hum} en tokens/s. Una configuración se marca como *usable* si $T \geq T_{\text{hum}}$. Este criterio no interviene en el cálculo de la FoM sino que se emplea como filtro de viabilidad en el análisis de resultados.

3.2. Diseño del sistema de medidas y de visualizaciones

3.2.1. MonitorSystem: requisitos y arquitectura

La obtención de las métricas descritas requiere una herramienta capaz de ejecutar experimentos de inferencia reproducibles y de registrar de forma simultánea las métricas de inferencia y el estado del *hardware*. Las herramientas de *benchmarking* de LLM existentes no cubren este doble registro sobre la plataforma objetivo, por lo que se desarrolló una herramienta propia, MonitorSystem⁴, implementada en C++. Su especificación se elaboró siguiendo la metodología de ingeniería de requisitos de la asignatura Introducción a la Ingeniería del Software y Sistemas de Información (IISI); el documento completo se recoge en el Anexo A y en esta sección se sintetizan sus aspectos esenciales.

³Datos extraídos de la Sección 4.1.6.

⁴Código y binarios disponibles en: <https://github.com/TFG-yisusc/MonitorSystemCplusplus>

Objetivos del sistema. MonitorSystem se concibe como una herramienta de línea de comandos orientada al investigador, cuyos objetivos se resumen en seis historias de usuario: ejecutar de forma automatizada un conjunto de *prompts* sobre un LLM en la Raspberry Pi 5 (OBJ-01); monitorizar el estado *hardware* durante la inferencia (OBJ-02); permitir la ejecución equivalente sobre distintos motores de inferencia (OBJ-03); parametrizar los experimentos mediante hiperparámetros configurables (OBJ-04); persistir los resultados en un formato estructurado y estándar para su análisis externo (OBJ-05); y, en conjunto, evaluar la viabilidad del despliegue de LLM en *hardware* de bajo coste (OBJ-06). El componente de visualización y análisis de los resultados se implementa como una herramienta independiente en Python, descrita en la Sección 3.2.2, lo que desacopla la captura de datos de su posterior tratamiento (OBJ-05).

Soporte de motores de inferencia. El sistema abstrae tres motores tras una interfaz común: i) llama.cpp, integrado como biblioteca para la inferencia en CPU sobre modelos en formato GGUF; ii) Ollama, servidor local accesible mediante API HTTP; iii) y hailo-ollama, servidor de inferencia propio de Hailo AI para el acelerador Hailo-10H, que opera sobre modelos en formato HEF y expone una API REST similar a la de Ollama, sin integración ni dependencia funcional entre ambos. Esta abstracción permite ejecutar un mismo experimento sobre distintos motores variando únicamente el campo *inference_engine* de la configuración, lo que sustenta las comparativas de las series E4 y E5. El motor hailo-ollama presenta dos limitaciones impuestas por su API: no expone los recuentos de *tokens* ni los tiempos por fase, que se registran con valor nulo (RN-07), y opera con los parámetros de *batch* de *prefill* y contexto fijados en tiempo de compilación del modelo HEF, por lo que dichos campos se almacenan como metadato, pero no surten efecto en ejecución (RN-09).

Tipos de experimento. Cada experimento se ejecuta en uno de tres modos. El modo TYPE_0 registra únicamente las métricas de inferencia, sin monitorización *hardware*. El modo TYPE_1 añade la monitorización *hardware* en paralelo. El modo TYPE_2 replica el comportamiento de TYPE_1 e introduce una pausa de cinco segundos antes del primer *prompt* y entre *prompts* consecutivos, con el fin de facilitar la distinción de cada inferencia.

Arquitectura y concurrencia. La monitorización *hardware* se ejecuta en un hilo independiente que muestrea el estado de la Raspberry Pi 5 con período *hardware_period* y vuelca cada muestra al fichero correspondiente de forma continua durante el experimento (RF-09). El hilo principal gestiona la inferencia secuencial de los *prompts*, y tras cada inferencia construye y persiste de inmediato el registro de métricas, forzando el vaciado del *buffer* para garantizar la integridad de los datos ante interrupciones (RF-10). El conjunto de *prompts* de evaluación, procedente del *dataset instruction_following_eval* de Google Research [170], se embebe en el binario en tiempo de compilación, lo que elimina dependencias de ficheros externos en ejecución (RI-05).

Salidas y reproducibilidad. Cada ejecución genera un directorio identificado por su marca temporal de inicio en nanosegundos (RN-10), que contiene tres artefactos: un fichero JSONL con las métricas de inferencia por *prompt*, un fichero JSONL con las muestras *hardware* periódicas, y un fichero resumen.json con los metadatos completos del experimento, incluida la configuración original y los metadatos de arquitectura del modelo. La reproducibilidad se garantiza mediante la fijación de la semilla de aleatoriedad: bajo idénticas condiciones de *hardware* y versión del modelo, una semilla fija produce salidas

idénticas (RN-08). Esta propiedad es la que permite fijar $seed = 42$ y $temperature = 0,0$ en la configuración de referencia del estudio.

3.2.2. *monitorviz*: requisitos y arquitectura

El análisis y la visualización de los artefactos⁵ producidos por MonitorSystem se implementan en una librería Python independiente, *monitorviz*⁶, cuya especificación formal, elaborada siguiendo la misma metodología IISSI, se recoge en el Anexo B. Esta separación desacopla la captura de datos, ejecutada sobre la Raspberry Pi 5 en tiempo real, de su tratamiento posterior en el equipo de análisis, donde se dispone de recursos computacionales más amplios y de un entorno interactivo Jupyter.

Capacidades principales. La librería cubre cuatro funcionalidades. La primera es la carga y validación de colecciones de ejecuciones mediante `load_collection`, con tolerancia controlada ante datos ausentes o corruptos. La segunda es el cálculo automático de las métricas descritas en la Sección 3.1.1: perplexidad, MBU naïve y corregida, trabajo CPU efectivo, eficiencia energética y figuras de mérito. La tercera es la exportación de resultados a DataFrames de pandas con esquema estable entre versiones, de modo que los *notebooks* no requieran modificación ante actualizaciones de la librería. La cuarta es la generación de visualizaciones estáticas mediante matplotlib, entre las que se incluyen series temporales de *hardware*, gráficas comparativas entre configuraciones, frontera de Pareto y mapas de calor de throttling.

Arquitectura. El paquete se estructura en cuatro capas sin dependencias ascendentes: `models/` (modelos de datos validados con Pydantic v2), `io/` (*parsers* de los artefactos de MonitorSystem), `transforms/` (cálculo de métricas derivadas) y `viz/` (generación de figuras). La librería requiere Python 3.11 o superior, se instala mediante `uv pip install -e .` y opera desde JupyterLab 4, de modo que los *notebooks* de análisis actúan directamente sobre los datos experimentales versionados en el repositorio, garantizando la trazabilidad completa entre los resultados presentados y las ejecuciones que los originan.

3.2.3. Limitaciones

Medición energética (series E0–E4). El sensor interno del SoC mide únicamente el consumo de la placa base, excluyendo el *Active Cooler* y los periféricos USB; la serie E5 subsana esta limitación mediante un vatímetro externo, lo que introduce una asimetría de comparabilidad entre series. Adicionalmente, dado que la duración del *prefill* es habitualmente inferior al período de muestreo (0,25 s), la energía por fase se estima de forma proporcional a la duración usando la potencia media del *run*, aproximación que se asume como limitación metodológica explícita.

Métrica MBU. La fórmula MBU asume transferencia completa de pesos en cada paso de decodificación; en Granite 4.0-H-Micro y Gemma 3n, cuyas arquitecturas divergen del modelo denso, los valores constituyen cotas superiores. El término M_{state} de la corrección

⁵Los *datasets* generados se encuentran en el siguiente repositorio: <https://github.com/TFG-yisuscc/TFG-DATA>

⁶Código disponible en: <https://github.com/TFG-yisuscc/MonitorSystemVisualizer>

para Granite se calculó a posteriori desde los metadatos del GGUF, y el valor de $N_{\text{params}}^{\text{act}}$ para modelos dispersos se fija manualmente a partir de la documentación arquitectónica publicada, al no ser extraíble de forma automática del fichero de cuantización.

Fase de carga en la serie E4 (Ollama). Ollama reporta `load_duration` no nulo y variable en cada *prompt* aunque el modelo permanezca en memoria [173], a diferencia de llama.cpp. El TTLM de la serie E4 es representativo únicamente en el primer *prompt* de cada *run*.

Serie E5: acelerador Hailo-10H. La ausencia de un procedimiento público para compilar modelos en formato HEF restringe la serie E5 a los modelos distribuidos por Hailo AI, impidiendo evaluar el mismo conjunto que en las series anteriores. Además, los parámetros de contexto y *batch size* quedan fijados en tiempo de compilación del HEF, lo que imposibilita las variaciones paramétricas de las series E2 y E3.

Métricas de calidad generativa. La perplejidad calculada por MonitorSystem es autorreferencial y no comparable entre modelos con tokenizadores distintos; no constituye por sí sola un indicador de idoneidad para una tarea concreta. El cálculo sobre corpus estandarizados se realiza mediante una herramienta auxiliar independiente (PerplexitySystem) fuera del alcance de las series principales.

Generalización. MonitorSystem ha sido validado exclusivamente sobre la Raspberry Pi 5 (BCM2712); las métricas que incorporan parámetros físicos de la plataforma (MBU, eficiencia CPU, métricas energéticas) requieren recalibración para cualquier otro SoC.

3.3. Diseño experimental

Esta sección describe la plataforma *hardware* empleada, la configuración de referencia común a todas las series y el protocolo de medida específico de cada una de ellas.

3.3.1. Plataforma hardware

La plataforma de referencia del estudio es una Raspberry Pi 5 con 8 GB de memoria LPDDR4X-4267 [174]. Su SoC, el Broadcom BCM2712, integra cuatro núcleos Cortex-A76 con frecuencia nominal de 2,4 GHz, 512 kB de caché L2 por núcleo y 2 MB de caché L3 compartida [172]. La interfaz de memoria de 32 bits proporciona un ancho de banda teórico de 17 GB/s [172]; por lo tanto en el contexto del cálculo de la MBU (Sección 3.1.1), el parámetro BW_{peak} se fija en 17 GB/s para esta plataforma. El sistema se alimenta mediante la fuente de alimentación oficial Raspberry Pi 27 W USB-C [175], que proporciona 5,1 V a 5 A mediante USB Power Delivery, garantizando una entrega de potencia estable durante las cargas computacionales de la inferencia y habilitando el límite de corriente USB extendido de 1,6 A. Como almacenamiento se emplea una tarjeta microSD SanDisk Extreme UHS-I de 128 GB [176], con velocidades secuenciales de hasta 190 MB/s en lectura y 90 MB/s en escritura, cuyo rendimiento condiciona directamente el tiempo de carga del modelo (TTLM) en los modelos de mayor tamaño. La unidad dispone de refrigeración activa mediante el *Active Cooler* oficial de Raspberry Pi [177], cuyo ventilador opera de forma controlada por el firmware en función de la temperatura del SoC.

Todas las series experimentales se ejecutan con refrigeración activa, a excepción de la condición sin ventilador de la serie E0, en la que el *Active Cooler* se desconecta físicamente con el fin de caracterizar el impacto del efecto térmico sobre las métricas de rendimiento (CI-07).

Para la serie E5, la plataforma se amplía con el Raspberry Pi AI HAT+ 2, que incorpora el acelerador neuronal Hailo-10H con una capacidad de cómputo de 40 TOPS (INT4) y 8 GB de memoria LPDDR4X dedicada [151]. Los modelos desplegados en este acelerador se obtienen directamente a través de la API de hailo-ollama ya compilados previamente en formato HEF con los parámetros de *batch* y contexto fijados en tiempo de compilación, tal como se indicó en la Sección 3.2.1. La medición del consumo energético en la serie E5 se realiza mediante un vatímetro de pared externo⁷, cuya medida contempla el consumo total del sistema, incluyendo el Raspberry Pi AI HAT+ 2 y las pérdidas de la fuente de alimentación, a diferencia del sensor interno empleado en las series E0–E4.

3.3.2. Configuración de referencia

Con el fin de garantizar la comparabilidad entre series, se define una configuración de referencia común que permanece constante en todos los experimentos salvo en el parámetro bajo estudio (*ceteris paribus*). Los valores adoptados son los siguientes: motor de inferencia llama.cpp; *batch size* de *prefill* igual a 512 *tokens*; ventana de contexto de 4096 *tokens*; temperatura de muestreo 0,0; semilla de aleatoriedad 42; los 15 primeros *prompts* del conjunto *instruction_following_eval*; y período de muestreo *hardware* de 0,25 s.

La temperatura 0,0 y la semilla 42 se fijan para garantizar la reproducibilidad de las salidas del modelo: bajo condiciones idénticas de *hardware* y versión del modelo, la respuesta generada para cada *prompt* es determinista (RN-08). La elección del motor llama.cpp como referencia obedece a que constituye la interfaz de inferencia de más bajo nivel disponible en la plataforma, sin la sobrecarga de un servidor intermediario, lo que proporciona la línea base de rendimiento más favorable para la comparativa.

La excepción a esta configuración es la serie E5, en la que la ventana de contexto se reduce a 2048 *tokens* por restricción del compilado de los modelos HEF empleados por hailo-ollama [171].

3.3.3. Protocolo de medida por serie experimental

Todas las series se ejecutan en modo TYPE_1 de MonitorSystem, con monitorización *hardware* activa durante la inferencia y período de muestreo de 0,25 s. Cada *run* genera los tres artefactos descritos en la Sección 3.2.1: métricas de inferencia por *prompt*, muestras *hardware* periódicas y fichero de resumen. La Tabla 3.1 recoge el número total de *runs* por serie.

E0: Impacto de la refrigeración. Se ejecuta la configuración de referencia sobre los seis modelos seleccionados (Tabla 2.3) en dos condiciones de refrigeración: con *Active Cooler* activo y con ventilador desconectado. Esta serie proporciona la línea base para las cuestiones CI-01 y CI-07, y sus resultados con refrigeración activa sirven de comparativa directa para las series E1, E2, E3 y E4, que comparten la misma configuración base en el punto de referencia de cada parámetro estudiado.

⁷Especificaciones del instrumento: precisión $\pm 2\%$; corriente de trabajo máxima 16 A (3680 W); rango de medida 0,00–65,00 A, 0–9999 W, 180–250 VAC, 0–9999 Hz.

E1: nivel de cuantización. Sobre cada uno de los seis modelos se evalúan cinco niveles de cuantización en formato GGUF: Q3_K_M, Q4_0, Q4_K_M, Q5_K_M y Q8_0, *ceteris paribus*. De forma adicional, el modelo Llama 3.2 1B se evalúa también en precisión BF16, por ser el único modelo del conjunto cuyo tamaño lo hace viable en este formato dentro de los límites de memoria de la plataforma. Esta serie aborda la cuestión CI-02.

La evaluación de la perplejidad de referencia sobre corpus estandarizado (en contraposición a la perplejidad sobre las respuestas generadas que registra MonitorSystem) se ha realizado mediante un programa auxiliar independiente denominado **PerplexitySystem**, empleando los 15 000 primeros *tokens* del conjunto *WikiText-2 raw* [178] con un tamaño de lote y de contexto de 512 *tokens*. Dado que la perplejidad es dependiente del tokenizador (véase la Sección 3.1.1), los resultados solo son comparables únicamente entre cuantizaciones del mismo modelo base o entre aquellos modelos que compartan un mismo tokenizador.

E2: tamaño de ventana de contexto. Se varía el tamaño de la ventana de contexto entre cinco valores: 512, 1 024, 2 048, 4 096 y 5 120 *tokens*, *ceteris paribus*. Esta serie aborda la cuestión CI-03.

E3: batch size de prefill. Se evalúan cinco valores del parámetro *batch size*: 128, 256, 512, 1 024 y 2 048 *tokens*, sobre los seis modelos, *ceteris paribus*. Esta serie aborda la cuestión CI-04.

E4: motor de inferencia. Se replica la configuración de referencia de E0 (refrigeración activa) sustituyendo el motor llama.cpp por Ollama, sobre los seis modelos, *ceteris paribus*. La comparación directa entre los resultados de E4 y los correspondientes de E0 permite cuantificar la sobrecarga introducida por el servidor HTTP de Ollama bajo condiciones de configuración equivalentes, abordando la cuestión CI-05.

E5: acelerador hardware Hailo-10H . Se ejecutan los modelos compatibles con el compilado HEF disponible, concretamente Llama3.2:3B⁸ y el destilado de Deepseek R1 [171] descargados directamente a través de la API de hailo-ollama, con la plataforma ampliada con el Raspberry Pi AI HAT+ 2. La ventana de contexto se reduce a 2 048 *tokens* por restricción de compilación; el resto de parámetros se mantiene en los valores de referencia, si bien los campos *batch size* y contexto se registran como metadatos, pero no surten efecto en ejecución (RN-09). La comparativa frente a los resultados de E0 con llama.cpp para los mismos modelos permite cuantificar la ganancia real del acelerador, abordando la cuestión CI-06. Dado que hailo-ollama no expone los tiempos por fase ni las log-probabilidades por *token* (RN-07), las métricas de perplejidad y de coste por fase no están disponibles para esta serie, en consecuencia, la FoM completa descrita en la Sección 3.1.1 tampoco, optándose por la versión reducida (Ec. 3.11).

⁸En las versiones posteriores de hailo-ollama para otras plataformas, este modelo ha sido reemplazado por la versión de 1B parámetros, no obstante, se puede seguir accediendo a él mediante el histórico del repositorio: https://github.com/hailo-ai/hailo_model_zoo_genai/commit/66901fc6c4108005bc9868e7f59adb8b2a2637d2

3.4. Preparación del entorno experimental

Esta sección describe el proceso de instalación y configuración del entorno *software* sobre la plataforma de destino, previo a la ejecución de las series experimentales.

3.4.1. Configuración del sistema

La plataforma de destino se configura con Raspberry Pi OS Lite de 64 bits en su versión Trixie (basada en Debian 13, imagen de fecha 2026-04-21), por ser la distribución oficial de referencia para la Raspberry Pi 5 y contar con soporte nativo para los controladores del acelerador Hailo-10H. Se selecciona la variante Lite, sin entorno gráfico, con el fin de minimizar la carga de procesos en segundo plano que pudiera interferir en las mediciones.

Tras la primera inicialización y actualización del sistema, se habilita el modo PCIe Gen 3 en la EEPROM del firmware, requisito necesario para que el Raspberry Pi AI HAT+ 2 opere a su rendimiento máximo. A continuación se instala el paquete `hailo-h10-all`, que proporciona los controladores y el *runtime* del acelerador Hailo-10H. El acelerador permanece inactivo hasta que es inicializado por el servidor `hailo-ollama` en la serie E5.

Para el acceso remoto al sistema durante el desarrollo y la ejecución de experimentos se emplea Tailscale 1.98.2 [179], que establece una red privada virtual entre la Raspberry Pi 5 y el equipo de desarrollo, permitiendo la conexión SSH sin depender de la red local. Esta infraestructura sustenta también la compilación remota de MonitorSystem mediante CMake y `vcpkg` desde CLion.

3.4.2. Instalación y configuración de los motores de inferencia

llama.cpp. MonitorSystem integra `llama.cpp` (versión 9030) [147] como biblioteca mediante `vcpkg` [180] con enlazado estático, lo que elimina dependencias dinámicas en tiempo de ejecución y simplifica el despliegue del binario en la Raspberry Pi 5. La construcción se realiza con CMake a través de la cadena de compilación remota configurada en CLion, con perfiles diferenciados para el equipo de desarrollo y la plataforma de destino.

Ollama. Ollama 0.24.0 se instala mediante el script de instalación oficial, que configura automáticamente el servicio del sistema y habilita la API HTTP en el puerto 11434. No se modifica la configuración por defecto del servidor, de modo que las diferencias de rendimiento observadas en la serie E4 respecto a `llama.cpp` son atribuibles únicamente a la sobrecarga inherente al motor, no a ningún ajuste adicional.

hailo-ollama. El servidor `hailo-ollama` específico para Raspberry Pi 5 (versión 5.11) se instala a partir de la página oficial de desarrolladores de Hailo⁹ y expone una API HTTP compatible con la de Ollama. Su inicialización activa el controlador del acelerador Hailo-10H y reserva los recursos del NPU para la inferencia.

⁹<https://hailo.ai/developer-zone/>

3.4.3. Preparación de los modelos

Modelos para llama.cpp y Ollama. Cuatro de los seis modelos ¹⁰seleccionados, Gemma 3n [181], Llama3.2 [182], [183], Deepseek R1-Qwen [184], se descargan en formato safetensors desde Hugging Face en precisión BF16. A partir de estos ficheros se generan los GGUF mediante la herramienta `hf_to_gguf` del repositorio de llama.cpp [147] que posteriormente se cuantizarán a los diversos niveles de cuantización empleados a través de llama-quantize (automatizado mediante el script `quantize.sh` [185]). Los dos modelos restantes, Granite 4.0 micro-h [186] y Ministral 3B [187], disponen de GGUF oficiales precompilados en Hugging Face por lo que para estos se utilizan directamente los artefactos disponibles en sus repositorios correspondientes siendo solo necesaria su cuantización a los diversos niveles empleados mediante la herramienta descrita anteriormente.

Para Ollama, la plantilla de cada Modelfile se extrae a partir del modelo base ya importado en el servidor mediante el comando `ollama show --modelfile`. Este procedimiento garantiza que los parámetros de arquitectura y la plantilla de *prompt* son los correctos para cada familia de modelos. Posteriormente, se aplica un *script* de automatización (`apply_modelfiles`), disponible en el repositorio de scripts del proyecto [185], que registra en Ollama cada modelo asociando la plantilla con el GGUF correspondiente. De este modo se asegura la coherencia a lo largo de todos los modelos evaluados en la serie E4.

Modelos para hailo-ollama (serie E5). Los modelos compatibles con el Hailo-10H , Deepseek-R1-Qwen [171] y Llama 3.2:3b⁸, se descargan directamente a través de la API de hailo-ollama en formato HEF. A diferencia del flujo descrito anteriormente, este proceso no requiere conversión manual: hailo-ollama gestiona la obtención y el registro del modelo de forma análoga al comando `ollama pull` en Ollama estándar. Los parámetros de *batch* y contexto quedan fijados en tiempo de compilación del fichero HEF.

Automatización de la ejecución experimental. La ejecución de cada serie experimental se automatiza mediante un *script* de Bash disponible en el repositorio de scripts del proyecto [185]. El *script* `testReboots.sh` en conjunción a la herramienta `cron` [188] orquesta la secuencia completa de *runs*: tras cada ejecución, reinicia el sistema y espera cinco minutos antes de invocar MonitorSystem con el fichero de configuración JSON correspondiente al siguiente *run*, lo que permite lanzar una serie completa de forma desatendida sin intervención manual entre experimentos consecutivos. Como excepción, en el experimento E5, durante el último minuto de espera previo al lanzamiento del *run*, el *script* arranca en paralelo el servidor hailo-ollama, de modo que este quede operativo en el momento en que MonitorSystem comienza su ejecución.

Configuración del sistema para la ejecución. Antes de iniciar las series experimentales se establece una partición de intercambio (*swap*) de 2 GB, necesaria para que los modelos de mayor tamaño puedan completar su carga en condiciones de memoria ajustada, y se activa el inicio de sesión automático para permitir la ejecución desatendida de los experimentos.

¹⁰Todos los modelos seleccionados, así como sus cuantizaciones y modelfiles se encuentran en la siguiente dirección: <https://huggingface.co/buckets/jescarcas5/MODELOS-TFG>

3.5. Planificación y recursos

El presente trabajo se enmarca simultáneamente en dos estructuras formales. Por un lado, constituye el Trabajo de Fin de Grado de la titulación, con una carga asignada de 12 ECTS (equivalentes a 300 horas de trabajo del estudiante según el criterio de 25 h/ECTS). Por otro, se desarrolla íntegramente en el contexto de una beca de colaboración con el departamento de Electrónica y Electromagnetismo, dotada de 420 horas distribuidas en 7 meses y medio. Cabe señalar que el TFG se inició de forma no oficial un semestre antes del comienzo formal de la beca, de modo que la fase de definición del proyecto y los primeros pasos de documentación precedieron al encuadre institucional de la colaboración. A partir del inicio de la beca, ambos marcos se solapan de forma completa: la totalidad del trabajo recogido en esta memoria es simultáneamente producto de la beca y del TFG. El alcance de la beca es ligeramente más amplio en tanto que contempla la posibilidad de extensión hacia líneas de investigación futuras no abordadas en el presente trabajo.

Fases del proyecto

El desarrollo del proyecto no siguió una planificación rígida predefinida, sino un enfoque iterativo condicionado por la naturaleza exploratoria del trabajo y por una serie de contingencias técnicas y organizativas descritas más adelante. La Tabla 3.2 resume las fases identificadas a posteriori, con sus períodos aproximados de ejecución y la Figura 3.1 las visualiza como diagrama de Gantt.

Tabla 3.2: Fases del proyecto y períodos de ejecución.

F	Descripción	Período	Actividades principales
F1	Definición del proyecto	Feb. 2025	Primera reunión con los tutores; definición del alcance y de los objetivos iniciales.
F2	Formación y revisión bibliográfica inicial	Mar. – Ago. 2025	Estudio del funcionamiento de los LLM; familiarización con los motores de inferencia; inicio de la recopilación bibliográfica; 2ª Reunión con los tutores (online).
F3	Prototipado en Python	Jun. – Oct. 2025	Implementación de versiones de prueba del sistema de monitorización y del visualizador en Python, con el fin de validar el enfoque y adquirir experiencia con las interfaces de los motores.
F4	Compilación bibliográfica y redacción del cap. 2	Sep. 2025 – Ene. 2026	Búsqueda sistemática en bases de datos académicas; organización de referencias en Zotero; redacción del estado del arte; 3ª Reunión con los tutores.
F5	Diseño e implementación de MonitorSystem (C++)	Feb. – Abr. 2026	Especificación de requisitos; implementación del sistema de monitorización en C++ con llama.cpp (vcpkg, enlazado estático); configuración del entorno de compilación remota.
F6	Configuración del entorno experimental	May. 2026	Instalación del SO y dependencias en la Raspberry Pi 5; conversión y registro de los modelos; validación del entorno; 4ª Reunión con los tutores (informal).
F7	Ejecución de las series experimentales	May. – Jun. 2026	Ejecución de las series E0–E5; supervisión y validación de los resultados.
F8	Análisis de resultados y redacción	May. – Jun. 2026	Análisis de los datos con el visualizador; redacción de la memoria.

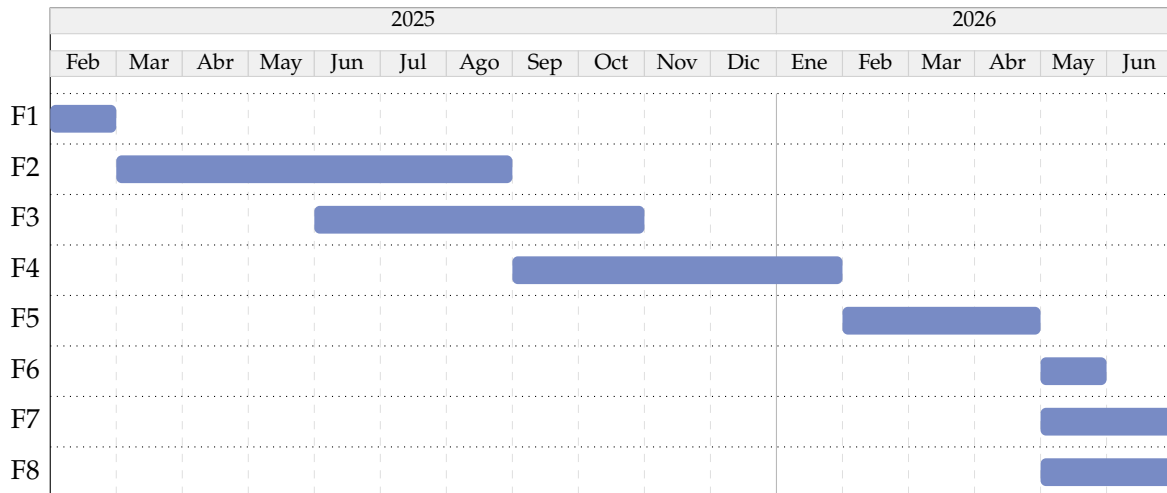


Figura 3.1: Diagrama de Gantt del proyecto. Las fases F1–F8 se corresponden con las descritas en la Tabla 3.2.

Desviaciones y contingencias

La planificación real se vio afectada por varias contingencias de índole técnica y organizativa que motivaron retrasos apreciables respecto a la estimación inicial.

Entre las de carácter técnico, cabe destacar las siguientes. El acelerador original disponible (Raspberry Pi AI Kit de primera generación) resultó ser incompatible con los motores de inferencia de LLM, lo que obligó a esperar la disponibilidad del Raspberry Pi AI HAT+ 2 y retrasó de forma significativa el inicio de la serie E5. La configuración del entorno de compilación remota en CLion sobre la Raspberry Pi 5 mediante SSH y Tailscale presentó sucesivos problemas de compatibilidad con CMake y vcpkg, que requirieron el uso provisional de Visual Studio Code hasta que la cadena de compilación quedó estabilizada. La detección de errores de programación en MonitorSystem con posterioridad a la ejecución de las primeras series obligó a repetir un conjunto de pruebas. Por último, la aparición de errores de *timeout* de comportamiento errático en MonitorSystem supuso el obstáculo técnico de mayor impacto: la ausencia de un patrón reproducible durante la fase inicial de diagnóstico imposibilitó su localización durante más de un mes, hasta que el error se volvió sistemático y pudo ser identificado y corregido.

A estas contingencias técnicas se sumaron causas de carácter organizativo, derivadas de compromisos con la Asociación IOTUS y de circunstancias personales, que redujeron la disponibilidad efectiva de trabajo en determinados períodos.

Recursos

Hardware. Raspberry Pi 5 de 8 GB con *Active Cooler*, fuente de alimentación oficial de 27 W y tarjeta microSD SanDisk Extreme de 128 GB, como plataforma de destino. Para la serie E5, el Raspberry Pi AI HAT+ 2 con acelerador Hailo-10H . Con el fin de mantener separados el entorno de desarrollo y el entorno experimental, se empleó una segunda tarjeta microSD SanDisk Extreme Pro de 64 GB [189] dedicada exclusivamente al desarrollo sobre la Raspberry Pi 5, de modo que el entorno experimental permaneciera aislado de cualquier modificación derivada de la actividad de desarrollo. Como equipo de

desarrollo se emplearon dos portátiles en fases sucesivas: un Slimbook Pro X 15 (2019), utilizado durante la fase inicial del proyecto con Fedora 41 a 43, y un Lenovo ThinkPad L14 Gen 6 con procesador AMD Ryzen AI 340 Pro y 64 GB de RAM, empleado en la fase final bajo Fedora 43. Ambos equipos se conectaron de forma remota a la Raspberry Pi 5 mediante Tailscale.

Software. Raspberry Pi OS Lite 64 bits (Trixie); llama.cpp (biblioteca, vcpkg); Ollama 0.24.0; hailo-ollama; MonitorSystem (desarrollo propio, C++); visualizador (desarrollo propio, Python); LaTeX mediante Overleaf para la redacción de la memoria; Zotero para la gestión bibliográfica; GitHub¹¹ para el control de versiones del código; Python para el prototipado inicial y el análisis de resultados.

Metodología de desarrollo *software*. El desarrollo de los componentes *software* propios (MonitorSystem y visualizador) siguió un modelo de cascada iterativa (*iterative waterfall*): las fases clásicas de análisis de requisitos, diseño, implementación y validación se ejecutaron de forma secuencial dentro de cada iteración, pero los resultados de la validación realimentaron el inicio de la siguiente, permitiendo incorporar correcciones y nuevos requisitos detectados durante la experimentación. La viabilidad de este enfoque se apoyó de forma determinante en las versiones de prototipado rápido desarrolladas previamente en Python (fase F3): la experimentación con dichos prototipos permitió identificar los requisitos funcionales, las reglas de negocio y las restricciones de la plataforma con suficiente claridad como para abordar la especificación formal recogida en el documento de requisitos y, posteriormente, la implementación definitiva en C++.

¹¹Los repositorios empleados se encuentran en la organización TFG-yisuscc: <https://github.com/TFG-yisuscc>

4. Resultados

En este capítulo se presentan los resultados obtenidos a partir de la evaluación de los distintos modelos de lenguaje en el dispositivo seleccionado (RPi5). El objetivo ha sido analizar su comportamiento en términos de rendimiento, consumo de recursos y calidad de las respuestas generadas con el fin de identificar las configuraciones más óptimas.

Para ello, se han realizado las distintas ejecuciones experimentales planteadas en el apartado anterior organizadas en distintos experimentos que se detallarán a continuación.

Los datos obtenidos permiten establecer una visión del impacto de las distintas variables en el proceso de inferencia en los dispositivos *edge* proporcionando una base objetiva para determinar qué estrategia de optimización son más efectivas en entornos en el borde.

4.1. Matriz experimental

A continuación, se muestran los resultados de los seis experimentos diseñados distribuidos según las variables modificadas.

4.1.1. Serie E0: línea base y refrigeración

La serie E0 establece la línea base de comportamiento de los seis modelos sobre la Raspberry Pi 5 en su configuración de referencia. Este experimento busca determinar qué modelo es más eficiente en condiciones normales, sirviendo además para dar respuesta a las cuestiones de investigación CI-01 y CI-07.

Para lograrlo, se realizaron seis runs con refrigeración activa (ventilador) y seis runs sin ella, cuyos resultados con ventilador constituyen el punto de comparación común para las series posteriores (E1–E4).

Tabla 4.1: Caracterización de los seis modelos en la configuración de referencia (Q4_K_M, contexto 4096, *batch* 512) sobre Raspberry Pi 5, con refrigeración activa (✓) y ausente (✗). Se reportan tamaño, huella en RAM, rendimiento temporal (tok/s, TTFT), coste energético (J/token), utilización corregida de ancho de banda (MBU_{corr}), eficiencia de cómputo (η_{CPU}) y figura de mérito (FoM_{full}). La columna *Usable* indica si el *throughput* supera el umbral de lectura humana T_{hum} . Para Gemma 3n se adopta el recuento de 1,91 B parámetros activos, cota inferior de su configuración MatFormer. Filas ordenadas por usabilidad y FoM_{full} descendente. Por pulcritud, las desviaciones típicas de todas las métricas se recogen en las Tablas C.1–C.3

Modelo	Ref.	N (B)	RAM (GB)	tok/s	TTFT (ms)	J/tok	MBU (%)	η_{CPU}	FoM	Us.
<i>Refrigeración activa (FAN)</i>										
Ministral 3B	✓	3.43	3.01	4.7	1929	1.440	66.1	0.980	1.058	✓
DeepSeek-R1 1.5B	✓	1.78	1.51	9.0	951	0.799	63.7	0.975	1.025	✓
Llama 3.2 3B (ref.)	✓	3.21	2.91	4.6	1941	1.469	62.5	0.987	1.000	✓
Llama 3.2 1B	✓	1.24	1.36	11.2	673	0.577	58.8	0.975	0.961	✓
Granite 4.0-H-Micro	✓	3.19	2.46	4.4	2348	1.519	53.0	0.968	0.934	✓
Gemma 3n	✓	1.91	2.16	5.7	2613	1.214	55.5	0.946	0.818	✓
<i>Refrigeración ausente (NOFAN)</i>										
Llama 3.2 1B	✗	1.24	1.36	7.2	1214	0.654	37.3	0.440	0.610	✓
DeepSeek-R1 1.5B	✗	1.78	1.51	4.2	2391	1.129	30.4	0.337	0.497	✓
Ministral 3B	✗	3.43	3.01	2.7	3703	1.784	38.0	0.429	0.618	✗
Granite 4.0-H-Micro	✗	3.19	2.46	2.9	4227	1.722	34.3	0.422	0.593	✗
Llama 3.2 3B (ref.)	✗	3.21	2.91	2.6	3718	1.857	36.2	0.446	0.585	✗
Gemma 3n	✗	1.91	2.16	3.1	5165	1.559	29.6	0.402	0.455	✗

En condiciones nominales con refrigeración activa, los seis modelos superan el umbral de usabilidad interactiva (sin refrigeración activa, cuatro de los seis modelos quedan por debajo de dicho umbral). La figura de mérito sitúa a Ministral 3B a la cabeza ($FoM_{full} = 1,058$), seguido muy de cerca por el destilado DeepSeek-R1 1.5B (1,025), ambos por encima del modelo de referencia Llama 3.2 3B (1,000 por definición). La proximidad de las puntuaciones en la zona alta indica que ningún modelo domina de forma absoluta: la elección óptima queda condicionada por la restricción que prime el integrador. El destilado DeepSeek-R1 1.5B constituye el compromiso más equilibrado para integración con recursos ajustados, pues alcanza 9,0 tok/s (el segundo mayor *throughput* del conjunto) con la menor latencia de primer *token* (951 ms) y el segundo menor coste energético (0,80 J/token), ocupando una huella de apenas 1,51 GB. En el extremo de velocidad bruta, Llama 3.2 1B lidera con 11,2 tok/s y la menor huella (1,36 GB), si bien su FoM (0,961) se ve moderada

por la ponderación según el número de parámetros. Para Gemma 3n se adopta deliberadamente la cota inferior de parámetros activos (1,91 B), coherente con su arquitectura MatFormer de activación selectiva.

E0 — Frontera de Pareto: velocidad vs huella RAM
(tamaño burbuja $\propto$ FoM_{full})

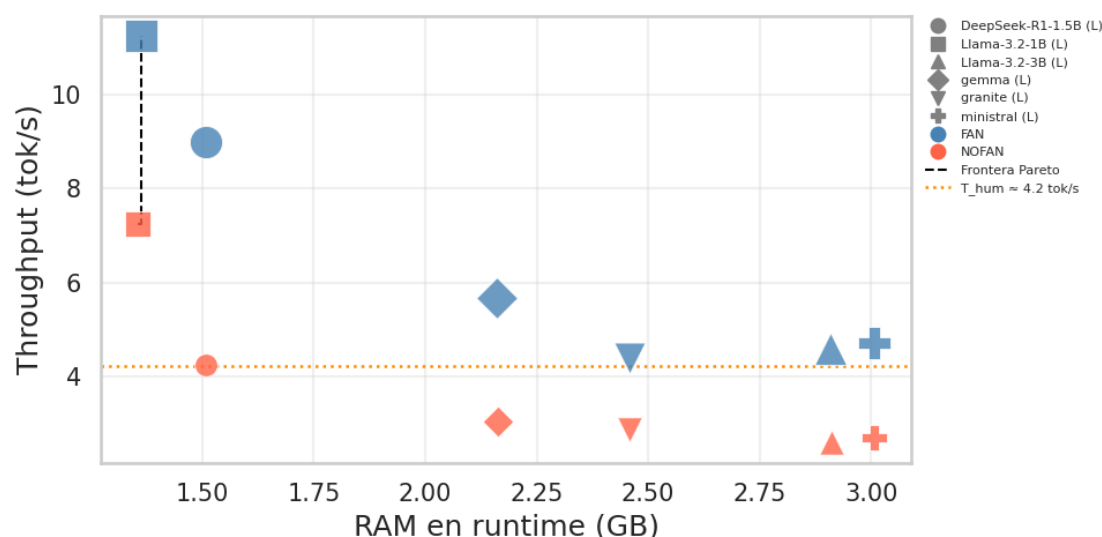


Figura 4.1: Frontera de Pareto en el plano *throughput* (tok/s) – huella en RAM (GB). El tamaño de cada burbuja es proporcional a la FoM_{full} y la línea vertical discontinua marca el umbral de usabilidad T_{hum} . Configuración de referencia, refrigeración activa.

Los modelos compactos dominan la frontera de Pareto (véase la Figura 4.1). Llama 3.2 1B y el destilado DeepSeek-R1 1.5B ocupan la región superior-izquierda (alto *throughput*, baja huella), constituyendo las opciones de referencia bajo restricción severa de memoria. El tamaño de burbuja matiza la lectura: Ministral 3B, pese a su menor velocidad (4,7 tok/s) y la mayor huella de memoria del conjunto (3,01 GB), presenta la mayor FoM (1,058), reflejo de su elevada eficiencia de cómputo y aprovechamiento de ancho de banda una vez ponderados por tamaño. Gemma 3n presenta la FoM más baja (0,818) pese a una huella moderada (2,16 GB), lo que apunta a una subutilización relativa del ancho de banda para su arquitectura MatFormer.

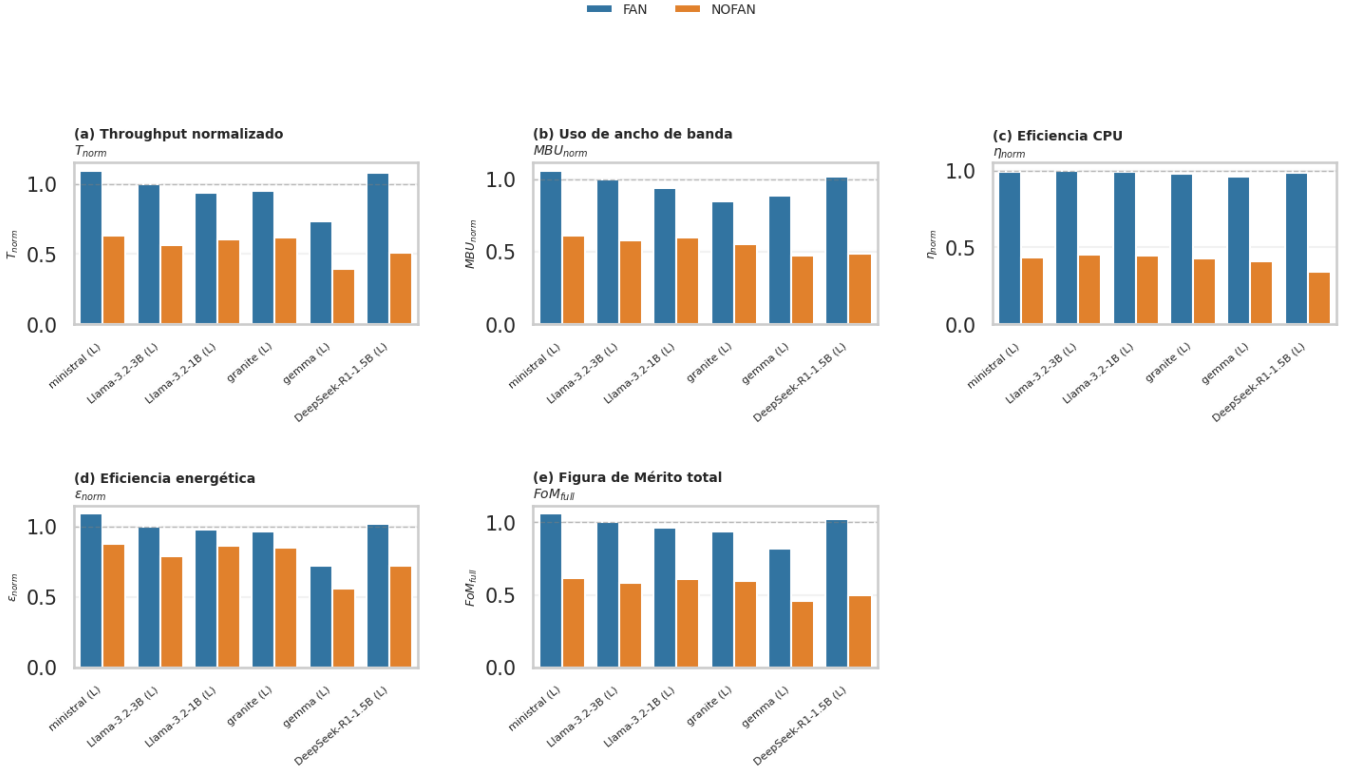


Figura 4.2: Descomposición de la FoM_{full} en sus cuatro ejes normalizados ($\tilde{T}$, $\widehat{MBU}$, $\tilde{\eta}$, $\tilde{\epsilon}$) por modelo, en la configuración de referencia con refrigeración activa. Cada eje se normaliza respecto a Llama 3.2 3B.

Con refrigeración activa la eficiencia de cómputo η_{CPU} es prácticamente homogénea entre modelos (0,95–0,99), por lo que no actúa como eje discriminante en estas condiciones; la diferenciación procede de la utilización de ancho de banda corregida y de la eficiencia energética ponderada (véase la Figura 4.2). Esta homogeneidad de η_{CPU} contrasta de forma marcada con el régimen de refrigeración ausente, donde se desploma hasta el rango 0,34–0,45, según se aprecia en la Tabla 4.1.

E0 — Mejora de throughput con ventilador activo

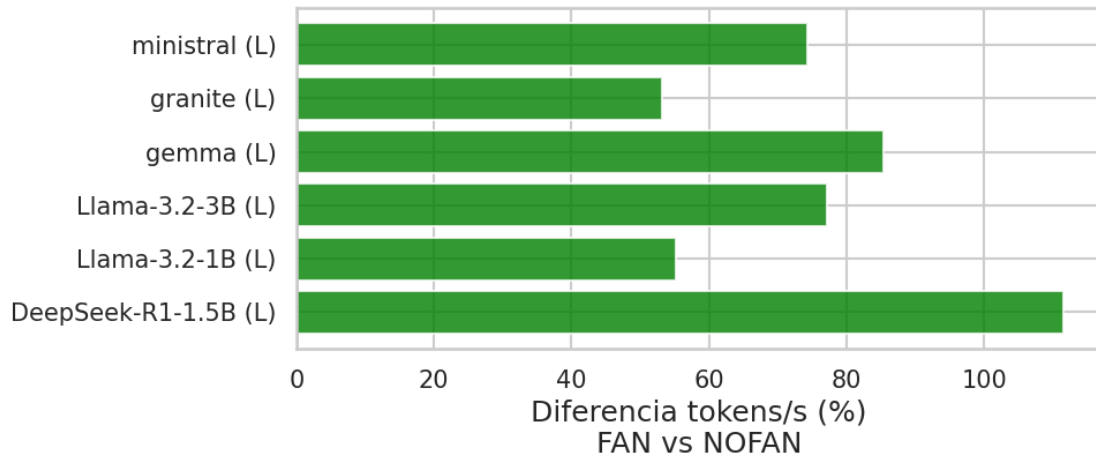


Figura 4.3: Incremento porcentual de *throughput* al activar la refrigeración (FAN frente a NOFAN) por modelo, en la configuración de referencia.

El efecto de la refrigeración activa es sustancial y desigual entre modelos (véase la Figura 4.3). El destilado DeepSeek-R1 1.5B es el más sensible, con un incremento del 111,5 % al activar el ventilador, frente al 55,2 % de Llama 3.2 1B. El resultado de mayor relevancia para la integración es el cambio de estado de usabilidad: con refrigeración ausente, cuatro de los seis modelos (Minstral, Granite, Llama 3.2 3B y Gemma 3n) caen por debajo del umbral T_{hum} y dejan de ser utilizables en modo interactivo. La refrigeración activa no es, por tanto, una optimización marginal, sino un requisito habilitante para la mayoría de configuraciones evaluadas. Asimismo, el desplome de η_{CPU} en convección natural (de $\approx 0,97$ a $\approx 0,40$) confirma que la causa es el efecto térmico y no una limitación intrínseca del modelo.

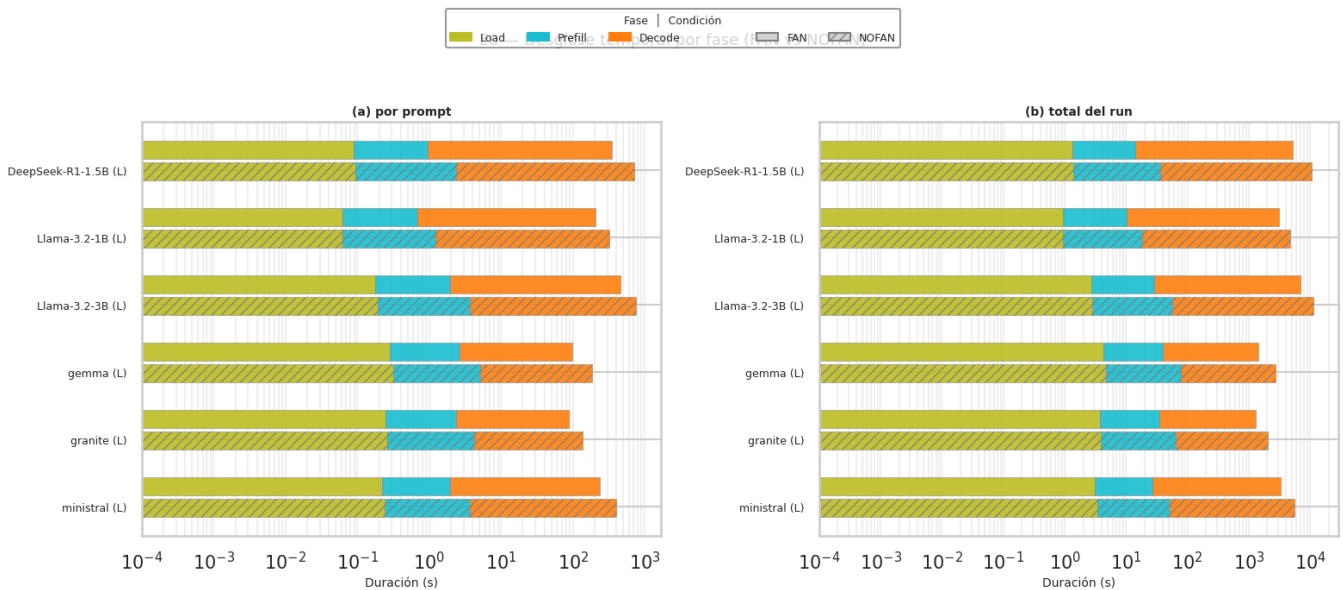


Figura 4.4: Reparto del tiempo de ejecución (escala logarítmica) entre las fases de carga del modelo (*load*), *prefill* y decodificación (*decode*) por modelo, en la configuración de referencia con refrigeración activa.

El desglose temporal (Figura 4.4, escala logarítmica) confirma que la decodificación concentra la práctica totalidad del coste de inferencia. La fase de carga del modelo es despreciable en régimen de operación (entre 0,06 y 0,29 s por *prompt*), lo que respalda la estrategia de mantener el modelo residente en memoria entre peticiones (CI-08). El *prefill* representa una fracción reducida y estable (0,6–2,3 s por *prompt*), mientras que la decodificación domina con holgura: oscila entre los 85 s por *prompt* de Granite y los 345 s de DeepSeek-R1 1.5B, cuyo mayor coste se atribuye a la naturaleza extensa de sus respuestas con razonamiento. Esta distribución justifica que el *throughput* de generación, y no la latencia de carga, sea la métrica temporal determinante para la experiencia interactiva, y motiva el foco de las series E2 y E3 en los parámetros que afectan a dichas fases.

La Figura 2.16, obtenida mediante un run independiente con Test Type 2¹, ofrece la perspectiva *hardware* de este mismo patrón para DeepSeek-R1-1.5B Q4_K_M: la rampa de carga de pesos eleva la RAM hasta su valor estacionario y da paso a un plateau de *decode* sostenido con la CPU a prácticamente 100 %. Al ser el *decode* la fase dominante, cualquier reducción de frecuencia por efecto térmico se traduce en una caída proporcional del *throughput* global, lo que explica que la supresión de refrigeración activa derrumbe η_{CPU} y expulse a cuatro modelos del rango de usabilidad.

Síntesis de hallazgos

La serie E0 establece la línea base de comportamiento de los seis modelos sobre la Raspberry Pi 5 en la configuración de referencia y responde a las cuestiones CI-01 y CI-07. Respecto a la caracterización nominal (CI-01), se concluye que los seis modelos son utilizables en modo interactivo siempre que operen con refrigeración activa, y que ninguno domina de forma absoluta: las figuras de mérito se agrupan de forma estrecha en la zona alta (Ministral 3B, 1,058; DeepSeek-R1 1.5B, 1,025; Llama 3.2 3B, 1,000), de modo que la elección óptima queda condicionada por la restricción que prime el integrador. Cuando el factor crítico es la huella de memoria o la latencia de respuesta, el destilado DeepSeek-R1 1.5B emerge como el compromiso más equilibrado, al combinar el segundo mayor *throughput* (9,0 tok/s), la menor latencia de primer *token* (951 ms) y una huella de 1,51 GB; cuando prima la velocidad bruta, Llama 3.2 1B es la opción de referencia (11,2 tok/s, 1,36 GB).

Respecto al impacto de la refrigeración (CI-07), se demuestra que la refrigeración activa no constituye una mejora marginal, sino un requisito habilitante: su ausencia provoca incrementos de tiempo por efecto térmico que expulsan del rango de usabilidad a cuatro de los seis modelos, y reduce la eficiencia de cómputo η_{CPU} desde valores próximos a 0,97 hasta el entorno de 0,40. La magnitud del efecto es desigual y mayor en los modelos de razonamiento con generación extensa, alcanzando un incremento de *throughput* del 111,5 % en DeepSeek-R1 1.5B al activar el ventilador.

Finalmente, el desglose por fases revela que la decodificación concentra la práctica totalidad del coste temporal y energético, mientras que la carga del modelo resulta despreciable en operación en cuanto se promedia por *prompt*. Este resultado orienta el resto del estudio: las series E2 y E3 se centran en los parámetros que gobiernan las fases de *prefill* y decodificación, y la línea base aquí establecida con refrigeración activa actúa como punto de comparación común para las series E1–E4.

¹A diferencia del Test Type 1 usado en todas las series experimentales, el Type 2 incluye pausas de cinco segundos antes de la carga, entre *prompt* y entre la finalización y la descarga del modelo, con el fin de facilitar visualmente la distinción entre las fases.

4.1.2. Serie E1: cuantización

La serie E1 evalúa el efecto del nivel de cuantización, expresado en bits por peso (bpw), sobre el rendimiento, la huella de memoria y la eficiencia de los seis modelos, manteniendo fija la configuración de referencia en el resto de parámetros. Constituye la respuesta a la cuestión CI-02 y es la serie de mayor valor práctico para un integrador, pues determina qué nivel de cuantización ofrece el mejor compromiso en la plataforma.

Nota sobre la perplexidad y runs excluidos. La calidad generativa bajo cuantización se evalúa mediante *PerplexitySystem*, sistema auxiliar que calcula la perplexidad en modo *teacher-forcing* sobre los primeros 512 tokens del corpus WikiText-2 *raw* (PPL_{wk}), en condiciones idénticas para todas las cuantizaciones del mismo modelo. Esta métrica es comparable entre cuantizaciones del mismo modelo base con tokenizador compartido; no es comparable entre modelos con tokenizadores distintos. Los runs de DeepSeek-R1-1.5B en Q4_0 y Q8_0 se excluyen de *todos* los análisis de la serie por colapso del modelo (generación de secuencias vacías o incoherentes).

El resultado más relevante para la integración es que la cuantización agresiva no solo reduce la huella de memoria, sino que *mejora* la figura de mérito en todos los modelos (véase la Tabla 4.2). La FoM_{full} crece de forma sistemática al descender de Q8_0 hacia Q4 y Q3, alcanzando su máximo en Q4_0 o Q3_K_M según el modelo. La causa es doble: la menor huella reduce el tráfico de memoria por *token* (factor limitante en esta plataforma *memory-bound*) y eleva el *throughput*, a la vez que disminuye el coste energético por *token*. Como contrapartida, la configuración Q8_0 cae por debajo del umbral de usabilidad T_{hum} en cuatro de los seis modelos, al no alcanzar la velocidad de generación mínima para interacción fluida.

La Tabla 4.3 resume la decisión según la prioridad del integrador. Cuando prima el equilibrio global, Ministral-3B en Q4_0 ofrece la mayor figura de mérito del conjunto (1,118). Si el factor crítico es la velocidad de respuesta, Llama-3.2-1B en Q4_K_M alcanza 11,7 tok/s con una huella de 1,36 GB. Y bajo restricción severa de memoria, Llama-3.2-1B en Q3_K_M opera en 0,97 GB conservando una velocidad equivalente.

La Figura 4.5 sintetiza la decisión en una sola vista. La banda de mejor desempeño se concentra en las columnas Q4_0 y Q3_K_M, lo que confirma que el punto óptimo de cuantización para esta plataforma se sitúa en torno a los 4–5 bits por peso.

La Figura 4.6 revela una tendencia común: la curva de FoM_{full} crece de forma monótona al aumentar la cuantización (derecha a izquierda) hasta alcanzar una rodilla en torno a Q4, a partir de la cual la mejora se atenúa o se revierte ligeramente en Q3 por el incremento de la latencia de primer *token*. El caso de BF16 en Llama-3.2-1B ilustra el extremo opuesto: la precisión completa hunde el *throughput* a 1,6 tok/s y expulsa a la configuración del rango de usabilidad, evidenciando que en una plataforma *memory-bound* el exceso de precisión es contraproducente.

La Figura 4.7 muestra que la degradación de calidad al pasar de Q8_0 a Q4_K_M es modesta en todos los modelos: inferior al 5 % (Ministral-3B: +1,7 %; Llama-3.2-3B: +3,2 %; Llama-3.2-1B: +3,5 %; Granite: +4,3 %). La variante Q5_K_M es virtualmente sin pérdidas (< 1,4 % en todos los modelos). Q3_K_M introduce degradaciones más apreciables (8–19 % según el modelo), con una inflexión clara respecto a los niveles superiores. Cabe destacar que Q4_0, pese a compartir el nivel de bits con Q4_K_M, presenta mayor degradación de calidad en todos los modelos (hasta un +13,4 % en Llama-3.2-1B), lo que refuerza la superioridad de las variantes K también en esta dimensión. Estos resultados son coherentes con los de Kurtic

Tabla 4.2: Métricas de inferencia, eficiencia energética y calidad por modelo y nivel de cuantización (serie E1). La perplejidad (PPL_{wk}) se calcula sobre los primeros 15000 tokens (con un contexto de 512) del corpus WikiText-2 en modo *teacher-forcing* mediante PerplexitySystem; el error estándar entre *chunks* se indica entre paréntesis. En negrita, la configuración con mayor FoM_{full} por modelo. † Q4_0 y Q8_0 de DeepSeek-R1-1.5B excluidos por colapso del modelo. ✗ configuración no interactiva ($< T_{hum}$). Por pulcritud, las desviaciones típicas de todas las métricas se recogen en las Tablas C.4–C.6.

Cuant.	BPW	RAM (GB)	tok/s	TTFT (ms)	J/tok	MBU (%)	PPL_{wk}	FoM_{full}	Us.
<i>DeepSeek-R1-1.5B</i> †									
Q5_K_M	5.8	1.65	8.7	1076	0.885	69.9	41.75 (1.28)	1.017	✓
Q4_K_M	5.0	1.51	8.9	952	0.796	63.2	43.66 (1.34)	1.026	✓
Q3_K_M	4.2	0.98	10.0	1810	0.760	59.6	46.06 (1.44)	1.051	✓
<i>Llama-3.2-1B</i>									
BF16 ^a	16.1	0.56	1.6	21514	4.595	24.6	14.69 (0.34)	0.286	✗
Q8_0	8.6	1.87	7.6	951	0.825	63.7	14.71 (0.34)	0.818	✓
Q5_K_M	5.9	1.46	11.2	752	0.648	66.7	14.86 (0.34)	0.964	✓
Q4_K_M	5.2	1.36	11.7	674	0.553	60.7	15.23 (0.35)	0.992	✓
Q4_0	5.0	1.32	11.2	458	0.554	56.2	17.27 (0.41)	0.964	✓
Q3_K_M	4.5	0.97	11.4	1305	0.619	51.2	17.17 (0.40)	0.919	✓
<i>Llama-3.2-3B</i>									
Q8_0	8.5	4.34	3.2	2830	2.052	71.5	10.89 (0.24)	0.869	✗
Q5_K_M	5.8	3.21	4.3	2194	1.718	67.7	10.98 (0.24)	0.966	✓
Q4_K_M	5.0	2.91	4.6	1936	1.487	62.4	11.24 (0.25)	1.000	✓
Q4_0	4.8	2.81	5.0	1197	1.306	64.1	11.52 (0.26)	1.060	✓
Q3_K_M	4.2	1.82	5.5	3562	1.331	62.6	12.07 (0.27)	1.074	✓
<i>Gemma-3n (2B activos)</i> ‡									
Q8_0	8.6	3.18	3.8	3309	1.708	51.3	35.54 (1.27)	0.670	✗
Q5_K_M	6.3	2.36	5.2	2933	1.462	53.2	33.83 (1.19)	0.758	✓
Q4_K_M	5.9	2.16	5.6	2693	1.238	55.0	36.06 (1.28)	0.814	✓
Q4_0	5.8	2.10	5.7	2219	1.175	55.1	37.78 (1.33)	0.832	✓
Q3_K_M	5.5	1.43	6.2	3864	1.205	57.0	33.00 (1.14)	0.850	✓
<i>Granite-4.0-H-Micro</i>									
Q8_0	8.5	3.92	2.8	3480	2.280	58.1	9.96 (0.24)	0.776	✗
Q5_K_M	5.7	2.79	3.9	2747	1.940	54.7	10.10 (0.24)	0.863	✗
Q4_K_M	4.9	2.46	4.4	2365	1.546	52.9	10.39 (0.25)	0.932	✓
Q4_0	4.7	2.37	4.5	1616	1.418	52.2	10.08 (0.23)	0.956	✓
Q3_K_M	3.9	1.09	5.2	4532	1.486	50.4	11.88 (0.29)	0.968	✓
<i>Ministral-3B</i>									
Q8_0 ^b	8.5	4.51	3.0	3016	2.236	69.7	8.43 (0.17)	0.858	✗
Q5_K_M	5.8	3.33	4.2	2342	1.848	67.8	8.47 (0.17)	0.971	✗
Q4_K_M	5.0	3.01	4.9	1874	1.427	71.2	8.57 (0.17)	1.091	✓
Q4_0	4.8	2.90	5.1	1164	1.308	68.4	8.90 (0.18)	1.118	✓
Q3_K_M	4.2	1.85	5.5	3630	1.434	67.0	9.13 (0.18)	1.108	✓

† Q4_0 y Q8_0 de DeepSeek-R1-1.5B excluidos por colapso del modelo (generación de caracteres especiales y respuestas vacías). ‡ Se adopta la cota inferior de parámetros activos (1,91 B), coherente con la arquitectura MatFormer de activación selectiva; la FoM_{full} se calcula sobre esta cota. ^a BF16: pesos accedidos vía mmap; la huella refleja únicamente buffers anónimos (activaciones y KV-cache). ^b Ministral Q8_0: uso marginal de swap (0,02 GB).

Tabla 4.3: Configuración recomendada según la prioridad del integrador, con huella en RAM de cada opción. El criterio 1 maximiza la figura de mérito global; el criterio 2 maximiza el *throughput*; el criterio 3 minimiza la huella en RAM entre configuraciones usables.

Criterio	Modelo	Cuant.
Mejor FoM_{full}	Ministral-3B	Q4_0
Máx. velocidad (usable)	Llama-3.2-1B	Q4_K_M
Mín. huella RAM	Llama-3.2-1B	Q3_K_M

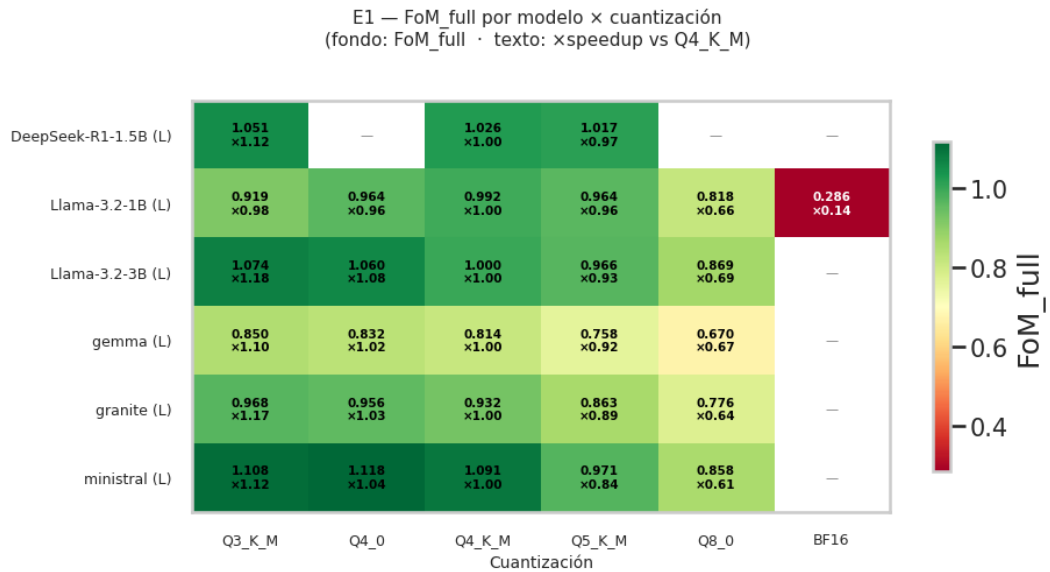


Figura 4.5: Mapa de calor de la FoM_{full} por modelo y nivel de cuantización. Cada celda muestra la FoM_{full} (métrica principal) y, como métrica complementaria, la aceleración de *throughput* relativa a la cuantización de referencia Q4_K_M de cada modelo. La escala cromática (verde) destaca las configuraciones de mejor desempeño.

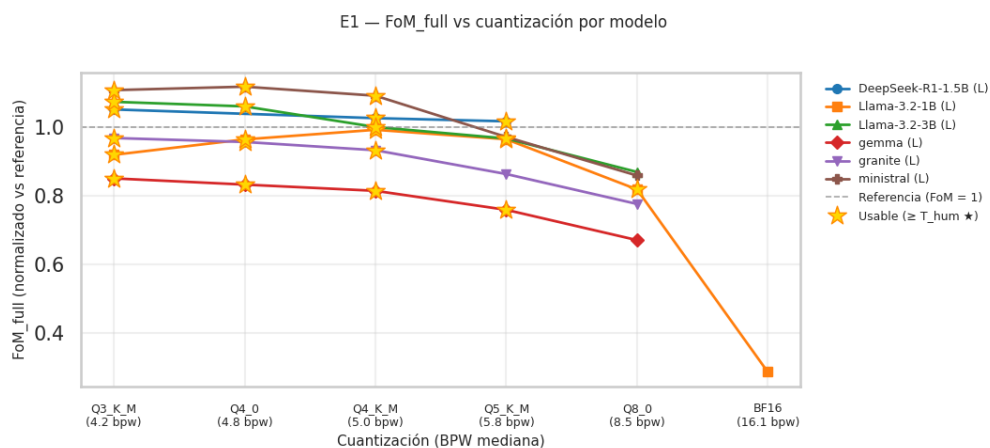


Figura 4.6: FoM_{full} en función de los bits por peso (bpw), una línea por modelo; valores menores de bpw (mayor nivel de cuantización) hacia la izquierda. La línea horizontal discontinua marca el nivel de usabilidad asociado al umbral T_{hum} . Configuración de referencia, refrigeración activa.

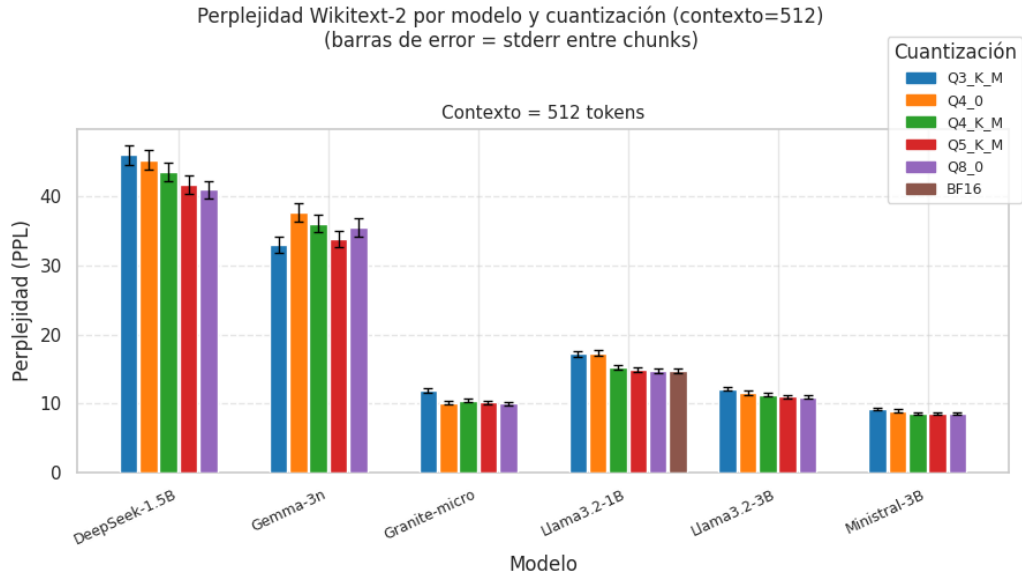


Figura 4.7: Perplejidad sobre WikiText-2 (PPL_{wk} , contexto 512 tokens, *teacher-forcing*) por modelo y nivel de cuantización. Las bandas de error corresponden al error estándar entre *chunks*. Los runs de DeepSeek-R1-1.5B en Q4_0 y Q8_0 se excluyen por colapso del modelo.

et al. [126] y validan, en el contexto de inferencia en el borde, que la cuantización a 4 bits conserva la calidad generativa dentro de márgenes operativamente aceptables.

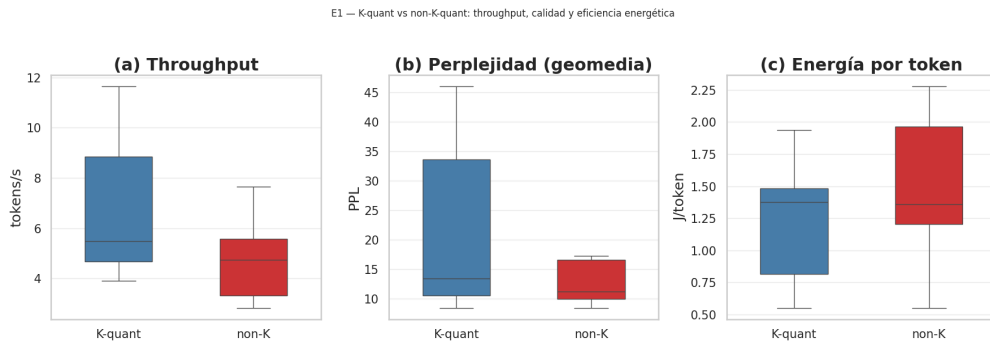


Figura 4.8: Comparación agregada entre las familias de cuantización K (*K-quants*) y no-K, en *throughput* (tok/s) y coste energético (J/token), sobre el conjunto de modelos y cuantizaciones. Se reportan mediana y dispersión.

La Figura 4.8 muestra que la comparación entre familias de cuantización no arroja una ventaja sistemática de las variantes K sobre las no-K en *throughput* ni en coste energético: las medianas son próximas (5,5 frente a 4,8 tok/s; 1,38 frente a 1,36 J/token) y los rangos se solapan ampliamente. En cambio, en la dimensión de calidad (PPL_{wk} , véase la Figura 4.7) las variantes K muestran una ventaja sistemática: Q4_K_M obtiene una perplejidad inferior a Q4_0 en todos los modelos evaluados (hasta un 13,4 % menor en Llama-3.2-1B). La elección entre familias resulta, por tanto, determinante en calidad aunque no en rendimiento, lo que hace preferibles las variantes K en todos los escenarios salvo restricción extrema de latencia de carga.

Síntesis de hallazgos

La serie E1 responde a la cuestión CI-02 y confirma la hipótesis H-E1 (véase la Sección 3.1), aportando la guía de cuantización para la integración en dos dimensiones complementarias.

En la dimensión de **rendimiento**, la cuantización a 4–5 bits por peso constituye el régimen óptimo en la Raspberry Pi 5 (Figura 4.6): maximiza la figura de mérito, reduce la huella de memoria y el coste energético por *token*, sin sacrificar la usabilidad. Q4_K_M se consolida como el punto de equilibrio recomendable por defecto, usable en los seis modelos con baja latencia de primer *token*; Q4_0 y Q3_K_M ofrecen mejoras adicionales en escenarios con mayor presión de recursos. Q8_0 expulsa del rango de usabilidad a cuatro de los seis modelos (Llama-3.2-3B, Gemma-3n, Granite y Ministral); también Granite Q5_K_M y Ministral Q5_K_M resultan no usables (véase la Tabla 4.2). BF16 degrada el *throughput* de Llama-3.2-1B hasta 1,6 tok/s, haciéndolo inviable.

En la dimensión de **calidad** (PPL_{wk} , Figura 4.7), la evaluación con PerplexitySystem sobre WikiText-2 de forma similar a los resultados obtenidos por Kurtic *et al.* [126], confirma que Q4_K_M es competitiva con Q8_0, siendo la degradación inferior al 5 % en todos los modelos. Q5_K_M es virtualmente idéntica a Q8_0 ($< 1,4$ %); Q3_K_M introduce degradaciones apreciables (8–19 % según el modelo). Las variantes K son sistemáticamente superiores a las no-K en calidad (véase la Figura 4.8), lo que hace preferibles las variantes K en todos los escenarios. Estos resultados son robustos por cuanto se sustentan en métricas independientes y complementarias: métricas físicas para el rendimiento y *teacher-forcing* sobre corpus estandarizado para la calidad.

4.1.3. Serie E2: tamaño de contexto

La serie E2 analiza el efecto del tamaño de la ventana de contexto (512–5 120 *tokens*) sobre el rendimiento, la huella de memoria y la eficiencia, manteniendo fijos el resto de parámetros de referencia. Responde a la cuestión CI-03 y permite al integrador dimensionar el contexto sin degradar la experiencia de uso.

Los resultados expuestos en la Tabla 4.4 confirman la hipótesis H-E2 (véase la Sección 3.1) y arrojan tres observaciones de interés práctico. En primer lugar, la huella en RAM crece de forma suave y predecible con el contexto, reflejo del coste creciente de la caché KV: el incremento varía entre el 2 % de Granite SSM (arquitectura de estado fijo, sin caché KV creciente) y el 21 % de Llama 3.2 3B al pasar de 512 a 5 120 *tokens*. En valores absolutos ningún modelo supera los 3,2 GB, por lo que el contexto no constituye un factor de presión de memoria frente al límite de 8 GB de la plataforma. En segundo lugar, el TTFT se mantiene prácticamente plano, lo que indica que ampliar la ventana de contexto no penaliza la latencia de primer *token* mientras el *prompt* efectivo no la ocupe. En tercer lugar, el *throughput* decrece de forma suave y monótona al aumentar el contexto (más acusada en los modelos de razonamiento, como DeepSeek-R1-1.5B, que pasa de 11,1 a 8,7 tok/s entre 512 y 5 120), con la consiguiente caída moderada de la figura de mérito.

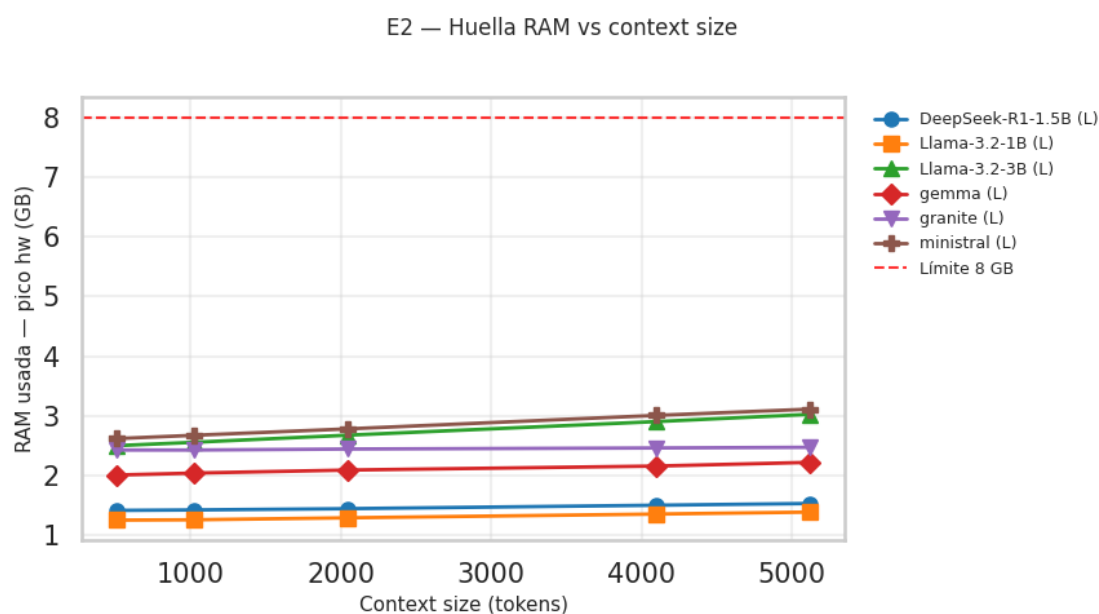


Figura 4.9: Huella en RAM medida frente al tamaño de contexto, una línea por modelo. Refleja la contribución de la caché KV al crecer la ventana de contexto. Refrigeración activa.

La Figura 4.9 muestra el crecimiento de la huella en RAM con el contexto: los modelos de atención estándar (Llama, DeepSeek, Ministral) presentan incrementos del 8–21 % entre los extremos del rango evaluado, mientras que Granite SSM permanece prácticamente constante (+2 %) al no requerir una caché KV creciente. En ningún caso se supera el límite de 8 GB de la plataforma, por lo que el contexto no es un factor de presión de memoria sino un parámetro de ajuste fino del *throughput*.

Tabla 4.4: Caracterización de cada modelo según el tamaño de contexto, sobre Raspberry Pi 5 con refrigeración activa. Se reportan huella en RAM, *throughput* (tok/s), latencia de primer token (TTFT), coste energético (J/token), utilización corregida de ancho de banda (MBU_{corr}), eficiencia de cómputo (η_{CPU}) y figura de mérito (FoM_{full}). La columna *Us.* indica si se supera el umbral de usabilidad T_{hum} . La huella en RAM varía con el contexto por el coste creciente de la caché KV; Granite SSM permanece prácticamente constante por su arquitectura de estado fijo. Por pulcritud, las desviaciones típicas de todas las métricas se recogen en las Tablas C.7–C.9.

Ctx	RAM (GB)	tok/s	TTFT (ms)	J/tok	MBU (%)	η_{CPU}	FoM	Us.
<i>Llama-3.2-1B</i>								
512	1.26	13.1	652	0.535	63.2	0.942	1.024	✓
1024	1.26	12.9	649	0.536	63.4	0.954	1.023	✓
2048	1.30	12.4	651	0.547	62.9	0.964	1.008	✓
4096	1.36	11.3	671	0.571	59.3	0.975	0.963	✓
5120	1.39	10.8	690	0.588	56.9	0.979	0.938	✓
<i>Llama-3.2-3B</i>								
512	2.50	5.4	1887	1.338	66.1	0.959	1.070	✓
1024	2.56	5.2	1862	1.355	65.3	0.971	1.056	✓
2048	2.68	5.0	1856	1.409	64.6	0.979	1.031	✓
4096	2.91	4.6	1936	1.443	62.4	0.987	1.000	✓
5120	3.03	4.5	1972	1.487	61.3	0.989	0.983	✓
<i>DeepSeek-R1-1.5B</i>								
512	1.42	11.1	924	0.651	73.3	0.945	1.164	✓
1024	1.43	10.8	919	0.650	72.3	0.959	1.156	✓
2048	1.45	10.3	927	0.681	70.5	0.967	1.124	✓
4096	1.51	9.1	951	0.767	64.3	0.974	1.036	✓
5120	1.54	8.7	948	0.813	62.1	0.977	1.002	✓
<i>Ministral-3B</i>								
512	2.62	5.1	2011	1.398	64.8	0.956	1.068	✓
1024	2.68	4.9	1907	1.423	64.3	0.967	1.057	✓
2048	2.78	4.8	1900	1.452	64.5	0.973	1.048	✓
4096	3.01	4.7	1921	1.431	66.1	0.980	1.055	✓
5120	3.11	4.7	1939	1.442	66.8	0.983	1.055	✓
<i>Gemma-3n</i>								
512	2.01	5.7	2710	1.223	48.3	0.934	0.784	✓
1024	2.04	5.6	2702	1.209	48.8	0.946	0.789	✓
2048	2.10	5.7	2605	1.201	51.3	0.946	0.802	✓
4096	2.16	5.7	2612	1.232	55.3	0.946	0.811	✓
5120	2.22	5.7	2613	1.223	57.5	0.946	0.821	✓
<i>Granite-4.0-H-Micro</i>								
512	2.43	4.4	2416	1.541	52.1	0.965	0.921	✓
1024	2.43	4.4	2394	1.520	52.4	0.968	0.927	✓
2048	2.45	4.4	2348	1.506	52.7	0.968	0.931	✓
4096	2.46	4.5	2349	1.509	53.5	0.968	0.936	✓
5120	2.48	4.4	2361	1.530	53.1	0.968	0.928	✓

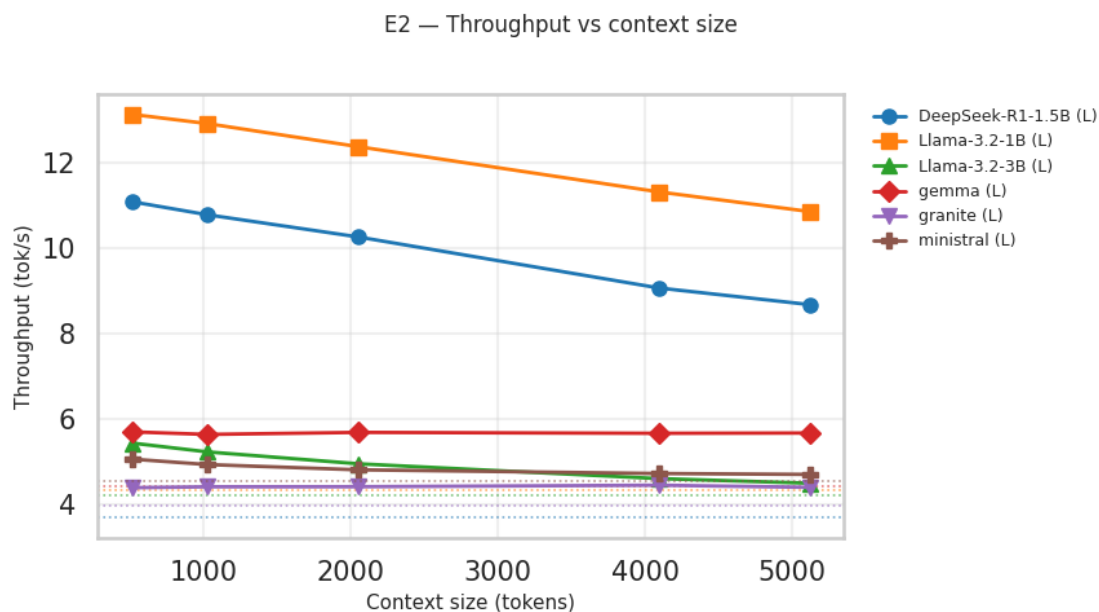


Figura 4.10: *Throughput* (tok/s) frente al tamaño de contexto, una línea por modelo. La línea horizontal discontinua marca el umbral de usabilidad T_{hum} . Refrigeración activa.

El *throughput* desciende de forma monótona y apreciable al ampliar el contexto (véase la Figura 4.10), sin que ningún modelo cruce el umbral de usabilidad en el rango evaluado. La caída es especialmente pronunciada en los modelos con mayor MBU (DeepSeek, -22% ; Llama-3.2-1B, -18%): a mayor saturación del ancho de banda, mayor sensibilidad al tráfico creciente de la caché KV por *token* generado. Gemma 3n y Granite SSM, con MBU inferior al 55% , permanecen prácticamente insensibles al contexto.

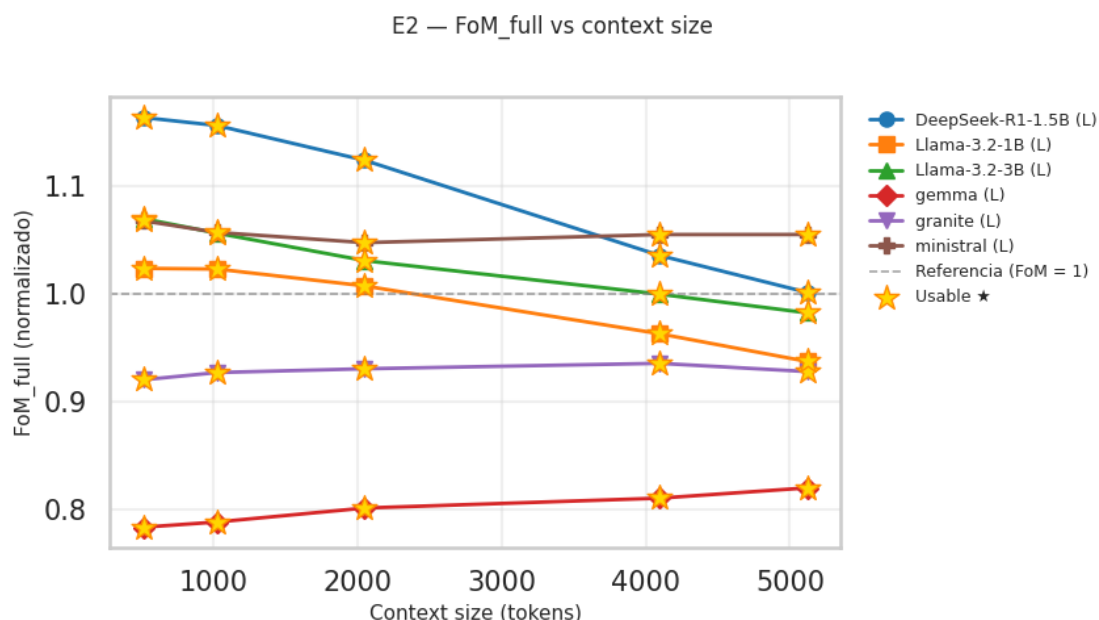


Figura 4.11: FoM_{full} frente al tamaño de contexto, una línea por modelo. Refrigeración activa.

La figura de mérito decrece de forma progresiva con el contexto en los modelos de mayor

MBU (véase la Figura 4.11): DeepSeek cae de 1,164 a 1,002 (−14 %) y Llama-3.2-3B de 1,070 a 0,983 (−8 %), lo que refuerza la recomendación de dimensionar el contexto al menor valor que cubra las necesidades de la aplicación. Gemma 3n y Granite SSM, cuya FoM permanece prácticamente plana, confirman que a MBU inferior al 55 % el contexto deja de ser el factor limitante.

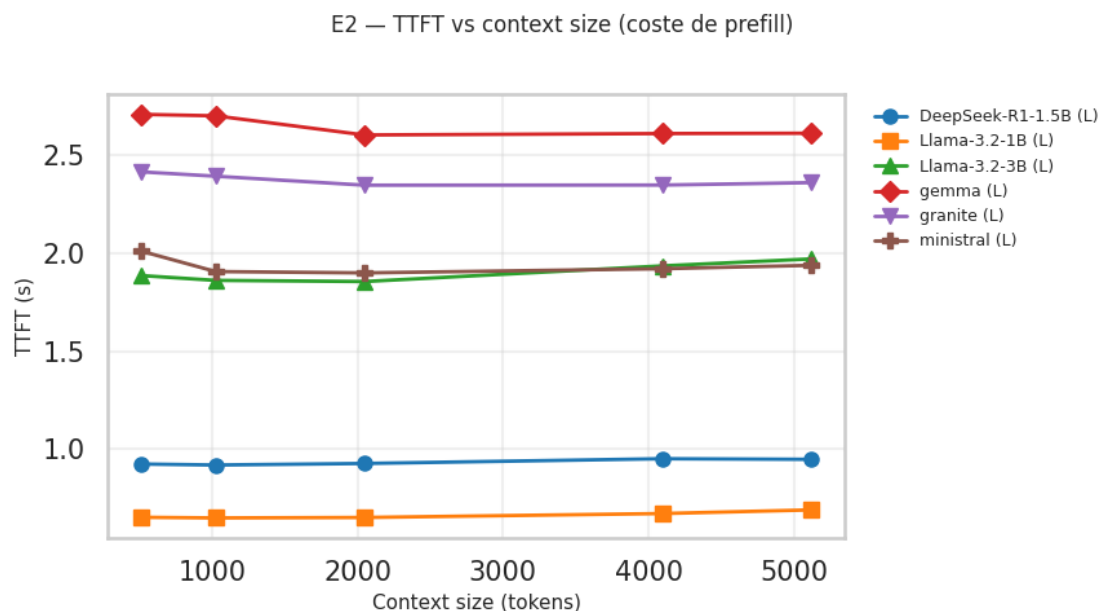


Figura 4.12: Latencia de primer *token* (TTFT) frente al tamaño de contexto, una línea por modelo. Refrigeración activa.

El TTFT se mantiene estable a lo largo de todo el rango de contextos (véase la Figura 4.12), confirmando la hipótesis de que la capacidad de la ventana de contexto, por sí sola, no encarece el *prefill* mientras el *prompt* no la ocupe. El coste de *prefill* depende de la longitud real del *prompt*, no de la capacidad reservada.

Síntesis de hallazgos

La serie E2 responde a la cuestión CI-03 y confirma la hipótesis H-E2: ampliar la ventana de contexto incrementa de forma marginal la huella de memoria (por el bajo coste de la caché KV en estos rangos) y no penaliza la latencia de primer *token*, a costa de una reducción suave y predecible del *throughput*. En consecuencia, la recomendación para la integración es directa: dimensionar el contexto al menor valor que cubra las necesidades de la aplicación, ya que un contexto holgado no aporta ventajas y reduce levemente el rendimiento. El contexto no constituye, en esta plataforma, un factor crítico de memoria ni de latencia, sino un parámetro de ajuste fino del *throughput*.

4.1.4. Serie E3: tamaño de batch de prefill

La serie E3 evalúa el efecto del tamaño de *batch* de *prefill* (128–2048) sobre el rendimiento y la latencia, con el resto de parámetros en su valor de referencia. Responde a la cuestión CI-04 y determina si el ajuste de este parámetro ofrece margen de optimización en la plataforma.

El resultado de E3, con los valores expuestos en la Tabla 4.5 es, por sí mismo, un hallazgo: el tamaño de *batch* de *prefill* no tiene un efecto apreciable sobre ninguna de las métricas. La RAM, el *throughput*, el TTFT, el coste energético y la figura de mérito permanecen esencialmente constantes a lo largo de todo el rango evaluado, con variaciones inferiores al margen de ruido experimental. Este comportamiento refuta parcialmente la hipótesis H-E3 (véase la Sección 3.1), que anticipaba una mejora de *throughput* con *batch* mayores, y tiene una explicación directa: en un escenario de inferencia de un único flujo conversacional, sin peticiones concurrentes que agrupar, el *batching* de *prefill* no encuentra paralelismo adicional que explotar sobre los cuatro núcleos de la plataforma. El *prefill* de un *prompt* individual ya satura los recursos disponibles con independencia del tamaño de *batch* configurado.

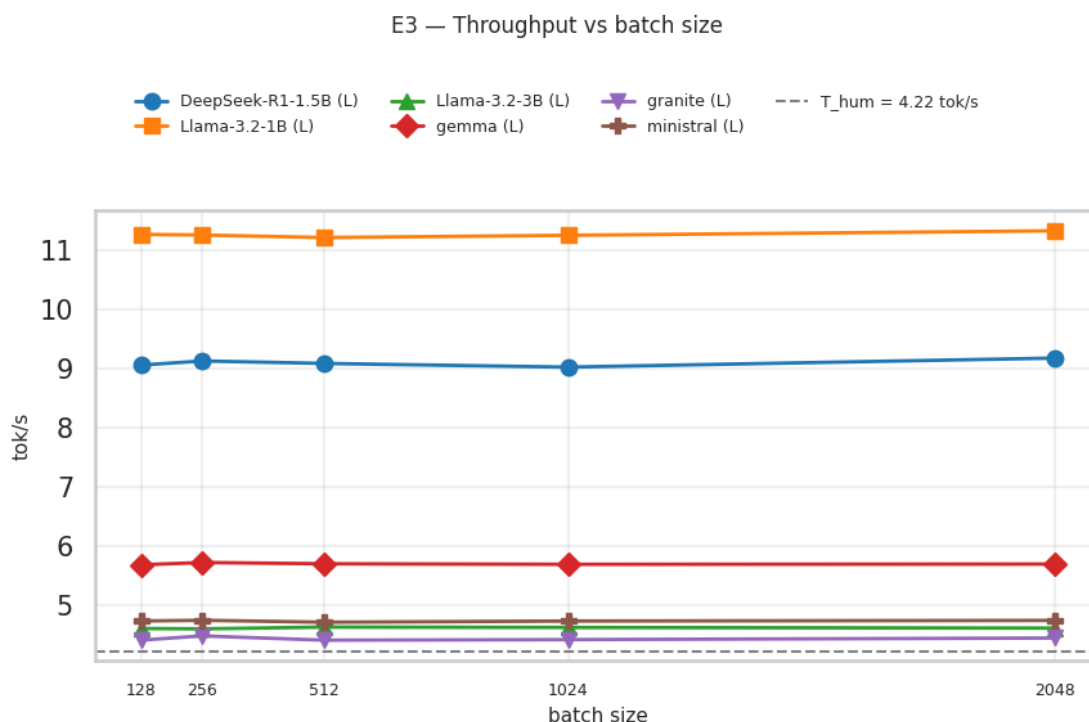


Figura 4.13: *Throughput* (tok/s) frente al tamaño de *batch* de *prefill*, una línea por modelo. La línea horizontal discontinua marca el umbral de usabilidad T_{hum} . Refrigeración activa.

Las curvas de *throughput* son planas para los seis modelos (véase la Figura 4.13): el tamaño de *batch* no modifica la velocidad de generación en inferencia de un solo flujo. Todos los modelos se mantienen en su nivel de usabilidad de referencia con independencia de este parámetro.

Tabla 4.5: Caracterización de cada modelo según el tamaño de *batch* de *prefill*, sobre Raspberry Pi 5 con refrigeración activa. Se reportan huella en RAM, *throughput* (tok/s), latencia de primer token (TTFT), coste energético (J/token), utilización corregida de ancho de banda (MBU_{corr}), eficiencia de cómputo (η_{CPU}) y figura de mérito (FoM_{full}). Todas las configuraciones resultan usables ($\checkmark$). Por pulcritud, las desviaciones típicas de todas las métricas se recogen en las Tablas C.10–C.12.

Batch	RAM (GB)	tok/s	TTFT (ms)	J/tok	MBU (%)	η_{CPU}	FoM
<i>Llama-3.2-1B</i>							
128	1.36	11.26	673	0.582	58.97	0.975	0.956
256	1.36	11.25	675	0.573	59.18	0.975	0.960
512	1.36	11.21	675	0.584	58.74	0.975	0.953
1024	1.36	11.25	675	0.583	58.92	0.975	0.955
2048	1.36	11.32	670	0.579	59.49	0.975	0.961
<i>Llama-3.2-3B</i>							
128	2.90	4.60	1942	1.450	62.45	0.987	0.998
256	2.91	4.59	1940	1.462	62.41	0.987	0.996
512	2.91	4.62	1932	1.453	62.77	0.987	1.000
1024	2.91	4.62	1941	1.437	62.53	0.987	1.002
2048	2.91	4.61	1934	1.442	62.56	0.987	1.000
<i>DeepSeek-R1-1.5B</i>							
128	1.51	9.05	950	0.779	64.25	0.974	1.031
256	1.51	9.12	942	0.761	64.83	0.974	1.041
512	1.51	9.08	950	0.763	64.51	0.974	1.038
1024	1.51	9.02	952	0.766	63.98	0.974	1.033
2048	1.51	9.17	948	0.758	65.31	0.974	1.045
<i>Ministral-3B</i>							
128	3.00	4.73	1927	1.433	66.12	0.980	1.054
256	3.01	4.74	1923	1.416	66.32	0.979	1.059
512	3.01	4.71	1926	1.420	65.90	0.980	1.054
1024	3.01	4.73	1925	1.433	66.06	0.980	1.054
2048	3.01	4.74	1931	1.412	66.22	0.980	1.059
<i>Gemma-3n</i>							
128	2.15	5.68	2611	1.220	55.41	0.946	0.813
256	2.16	5.72	2597	1.207	55.77	0.946	0.818
512	2.17	5.69	2612	1.224	55.57	0.946	0.814
1024	2.16	5.68	2610	1.221	55.48	0.946	0.814
2048	2.16	5.69	2611	1.215	55.52	0.946	0.815
<i>Granite-4.0-H-Micro</i>							
128	2.44	4.40	2352	1.519	52.89	0.968	0.928
256	2.45	4.48	2328	1.489	53.74	0.968	0.941
512	2.46	4.40	2355	1.534	52.88	0.968	0.926
1024	2.46	4.41	2360	1.526	52.99	0.968	0.928
2048	2.46	4.44	2344	1.522	53.30	0.968	0.931

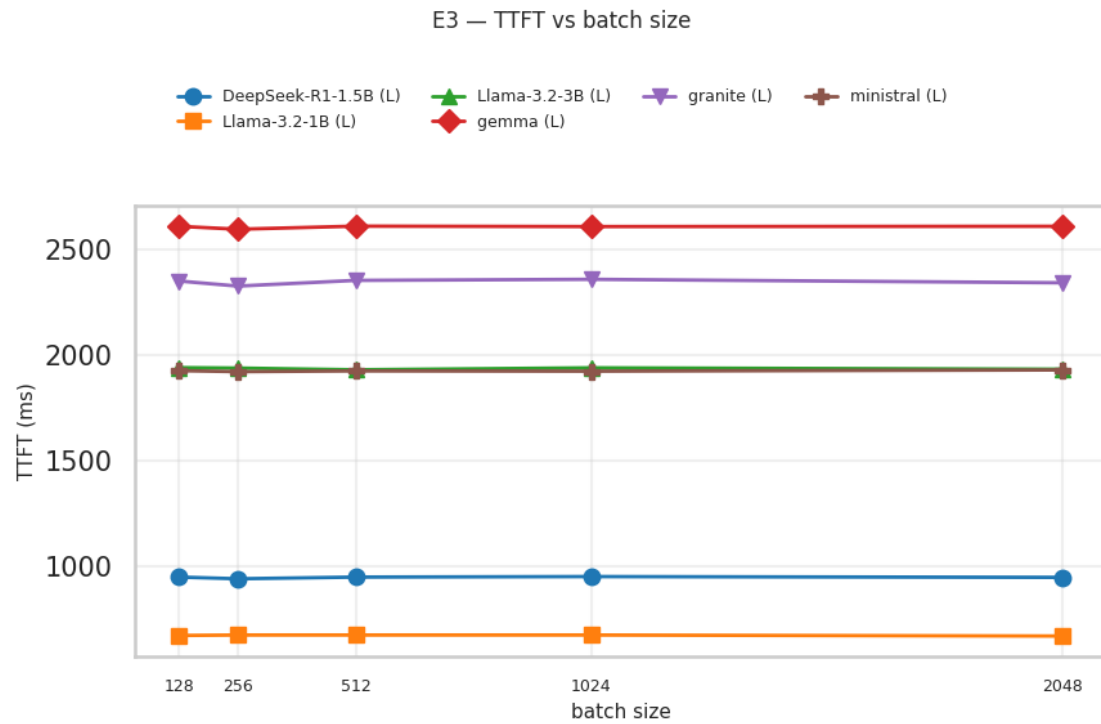


Figura 4.14: Latencia de primer *token* (TTFT) frente al tamaño de *batch* de *prefill*, una línea por modelo. Refrigeración activa.

El TTFT tampoco se ve afectado por el tamaño de *batch* (véase la Figura 4.14), lo que confirma que, en ausencia de concurrencia, el coste de *prefill* de un *prompt* individual es independiente de la capacidad de agrupación configurada.

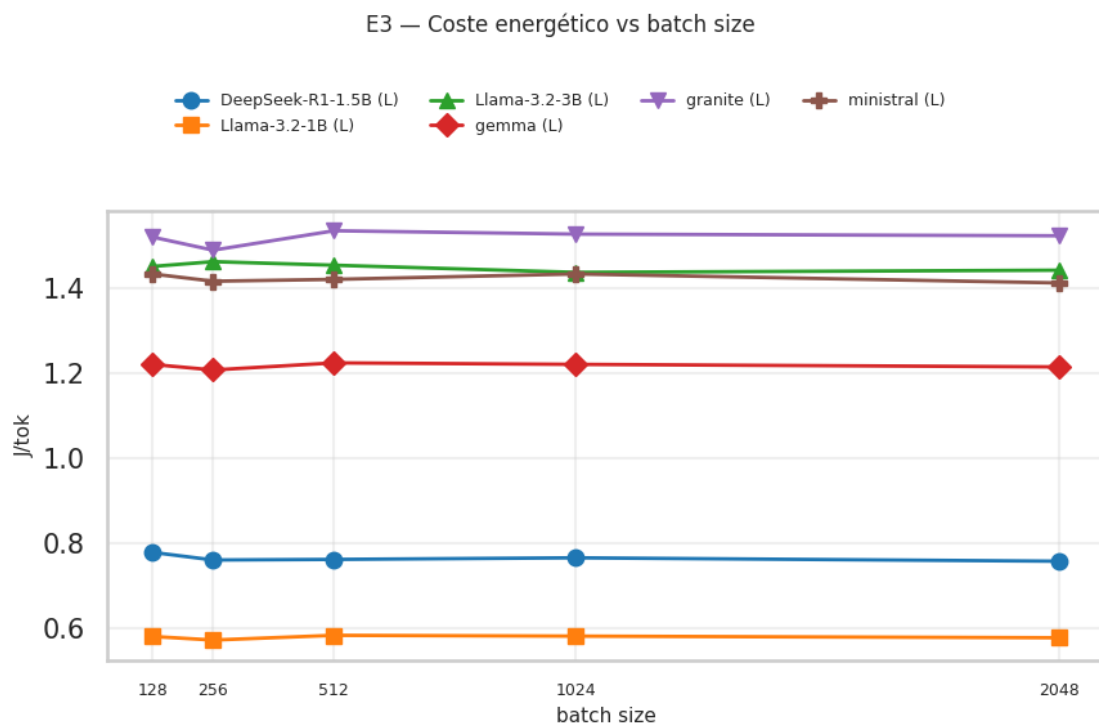


Figura 4.15: Coste energético por *token* (J/token) frente al tamaño de *batch* de *prefill*, una línea por modelo. Refrigeración activa.

El coste energético por *token* permanece estable en todo el rango (véase la Figura 4.15), en coherencia con la invariancia del *throughput*: al no alterarse el patrón de cómputo, tampoco lo hace el consumo por *token* generado.

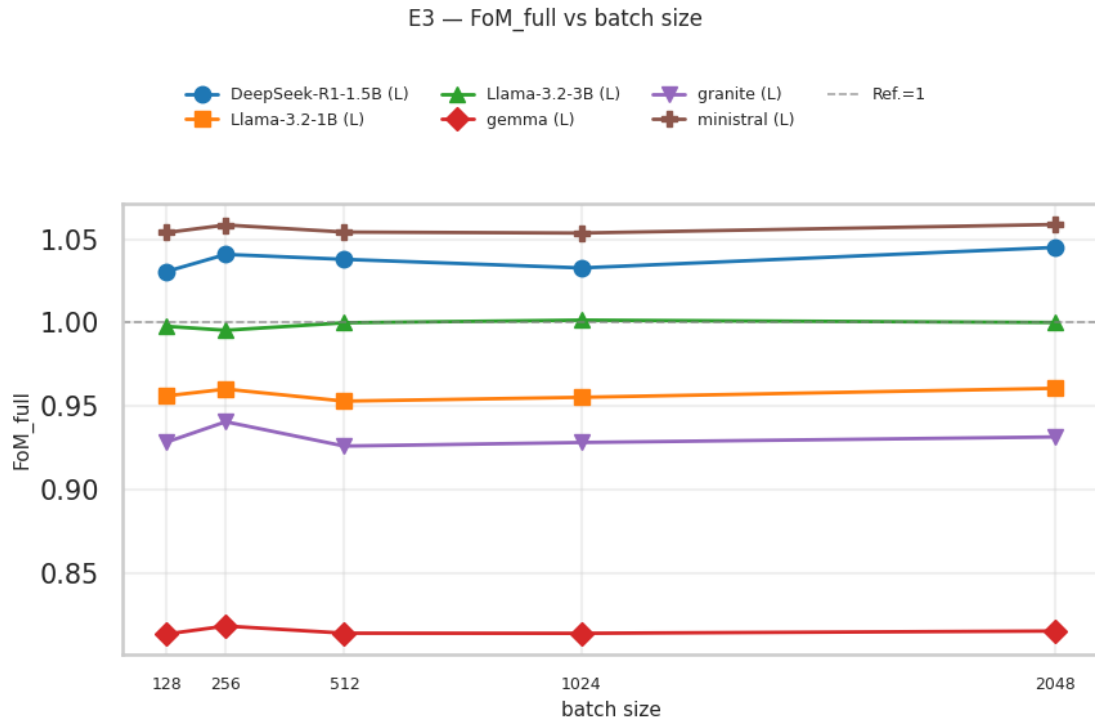


Figura 4.16: FoM_{full} frente al tamaño de *batch* de *prefill*, una línea por modelo. Refrigeración activa.

La figura de mérito refleja la misma invariancia (véase la Figura 4.16): las diferencias entre tamaños de *batch* quedan dentro del ruido experimental. El parámetro no constituye, por tanto, una palanca de optimización en el régimen de uso evaluado.

Síntesis de hallazgos

La serie E3 responde a la cuestión CI-04 con un resultado claro: el tamaño de *batch* de *prefill* es indiferente para el rendimiento, la latencia y la eficiencia en inferencia de un único flujo conversacional sobre la Raspberry Pi 5. La hipótesis H-E3, que anticipaba una mejora de *throughput* con *batch* mayores, se refuta para este escenario, al no existir peticiones concurrentes cuyo *prefill* pueda agruparse y paralelizarse. La implicación práctica para el integrador es que este parámetro puede mantenerse en su valor por defecto sin penalización alguna, y que cualquier beneficio del *batching* solo podría emerger en despliegues con concurrencia (varios *prompts* simultáneos), un escenario fuera del alcance de este trabajo y que se señala como posible extensión futura.

4.1.5. Serie E4: motor de inferencia

La serie E4 compara dos motores de inferencia (llama.cpp y Ollama) ejecutando los mismos modelos en idéntica configuración, para aislar la sobrecarga introducida por la capa de *runtime*. Responde a la cuestión CI-05 y orienta la elección de la pila de *software* de despliegue.

Tabla 4.6: Comparación entre llama.cpp (l.cpp) y Ollama por modelo, sobre Raspberry Pi 5 con refrigeración activa. Se reportan *throughput* (tok/s), latencia de primer token (TTFT), coste energético (J/token) y figura de mérito (FoM_{full}) para cada motor, junto con la variación porcentual de *throughput* y de FoM de Ollama respecto a llama.cpp. Valores positivos de Δ indican ventaja de Ollama. Por pulcritud, las desviaciones típicas de todas las métricas se recogen en las Tablas C.13–C.15.

Modelo	tok/s		TTFT (ms)		J/tok		FoM		Δ %	Δ %
	l.cpp	Oll.	l.cpp	Oll.	l.cpp	Oll.	l.cpp	Oll.	tok/s	FoM
DeepSeek-R1-1.5B	8.98	9.49	951	3111	0.799	0.748	1.025	1.035	+5,7	+1,0
Llama-3.2-1B	11.23	11.16	673	2552	0.577	0.616	0.961	0.906	−0,6	−5,7
Llama-3.2-3B	4.61	5.08	1941	6563	1.469	1.495	1.000	1.030	+10,3	+3,0
Gemma-3n	5.68	5.54	2613	7236	1.214	1.255	0.818	0.751	−2,5	−8,2
Granite-4.0-H	4.41	4.32	2348	9932	1.519	1.688	0.934	0.883	−2,0	−5,4
Ministral-3B	4.73	3.55	1929	12848	1.440	2.011	1.058	0.826	−25,0	−21,9

La comparación epuesta en la Tabla 4.6 revela que llama.cpp ofrece, en conjunto, un rendimiento igual o superior a Ollama, con una diferencia especialmente marcada en la latencia de primer *token*. El *throughput* es comparable entre motores en la mayoría de modelos (con ventajas puntuales de Ollama en Llama-3.2-3B (+10,3 %) y DeepSeek (+5,7 %), y una penalización severa en Ministral-3B (−25 %)), pero el TTFT de Ollama es sistemáticamente varios órdenes de magnitud mayor: frente a los menos de 2 s de llama.cpp, Ollama llega a superar los 12 s en Ministral-3B. Esta sobrecarga de inicialización por petición lastra la experiencia interactiva y penaliza la figura de mérito en cuatro de los seis modelos.

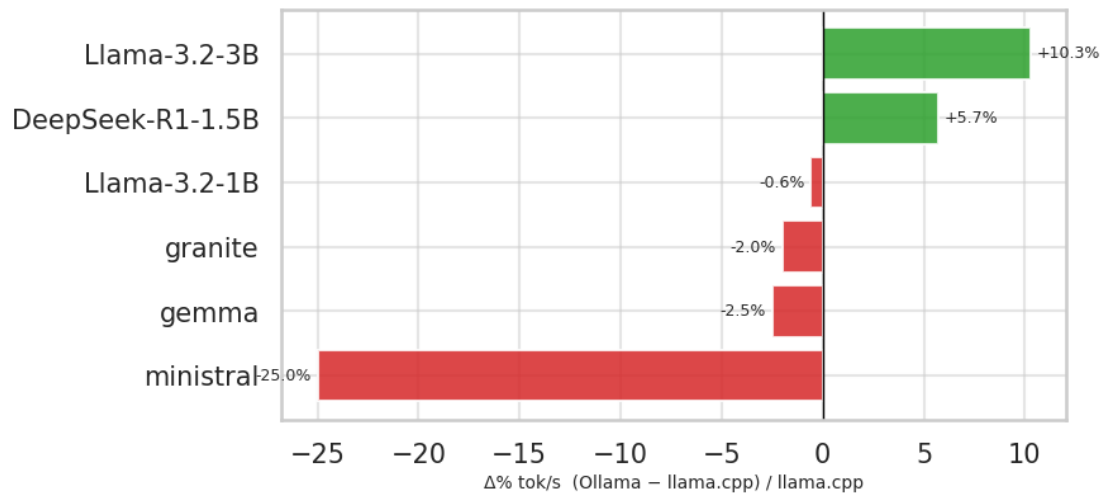


Figura 4.17: Variación porcentual del *throughput* de Ollama respecto a llama.cpp por modelo (barras divergentes desde cero). Valores negativos indican ventaja de llama.cpp. Refrigeración activa.

La variación de *throughput* no presenta un patrón uniforme (véase la Figura 4.17): Ollama mejora en dos modelos y empeora en cuatro, con un caso de penalización pronunciada (Ministral-3B). En *throughput* puro, por tanto, ningún motor domina de forma concluyente, y la elección no puede basarse solo en esta métrica. La figura de mérito (véase la Tabla 4.6) inclina la balanza hacia llama.cpp, que supera o iguala a Ollama en cuatro de los seis modelos, con diferencias notables en Ministral-3B y Gemma-3n; solo Llama-3.2-3B y DeepSeek-R1-1.5B obtienen una FoM ligeramente superior con Ollama, insuficiente para compensar su penalización de latencia. Cabe señalar que el run de Ministral-3B con Ollama registró una generación vacía ($n = 1$), lo que se atribuye a la sobrecarga de inicialización que puede provocar un *timeout* interno del motor antes de emitir el primer token; este incidente no altera la conclusión general pero refuerza la idoneidad de llama.cpp para este modelo.

La diferencia determinante se concentra en el TTFT (véase la Tabla 4.6): Ollama introduce una sobrecarga de inicialización por petición que multiplica la latencia de primer *token* respecto a llama.cpp en todos los modelos, alcanzando un factor superior a seis en el peor caso (Ministral-3B: 12 848 ms frente a 1 929 ms). El coste energético por *token* es comparable entre motores, con ligera ventaja de llama.cpp en los modelos donde la sobrecarga de Ollama es mayor.

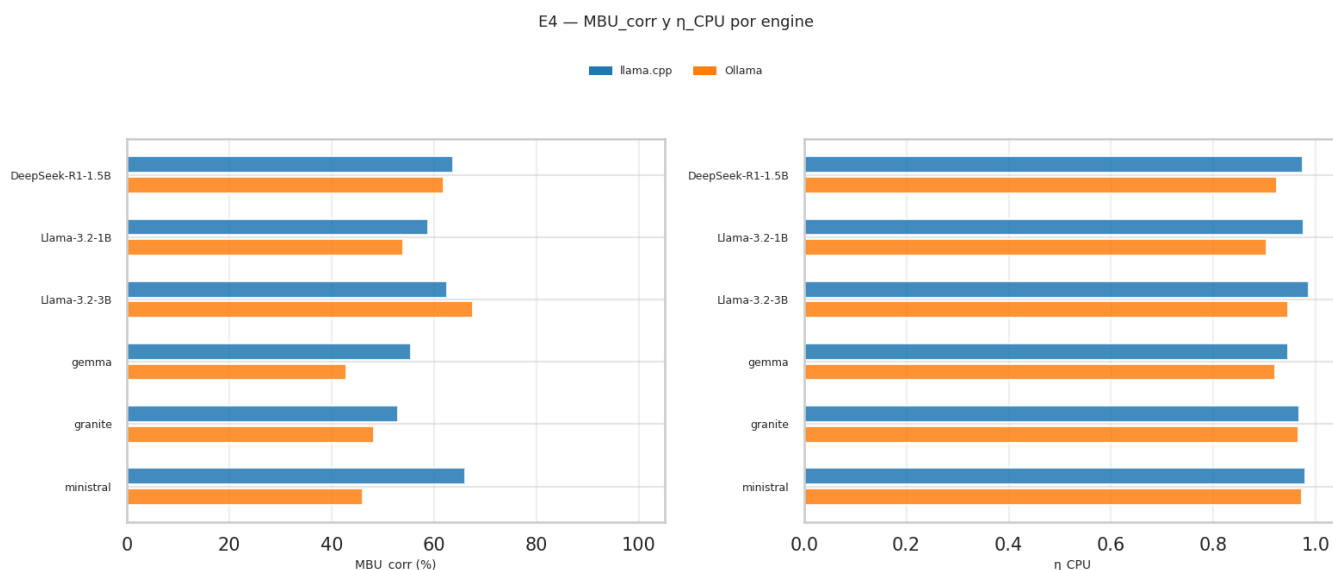


Figura 4.18: Utilización corregida de ancho de banda (MBU_{corr}) y eficiencia de cómputo (η_{CPU}) comparadas entre llama.cpp y Ollama, por modelo. Refrigeración activa.

El aprovechamiento de *hardware* es similar entre motores en utilización de ancho de banda y eficiencia de cómputo durante la fase de generación (véase la Figura 4.18), lo que indica que la diferencia de comportamiento no procede del núcleo de inferencia (compartido) sino de la capa de orquestación y de la gestión de la petición, donde Ollama añade sobrecarga [173]. Una excepción destacable es la huella de RAM de Gemma 3n, que en Ollama prácticamente se duplica (4,30 GB frente a 2,16 GB en llama.cpp), lo que apunta a una gestión de memoria menos eficiente, posiblemente debida a un soporte parcial de la arquitectura MatFormer.

En términos de usabilidad, 10 de los 12 runs resultan usables; los 2 no usables corresponden ambos a Ollama: Ministral-3B por *throughput* insuficiente (3,55 tok/s < T_{hum}) y DeepSeek-R1-1.5B por una tasa tokens/palabra anómalamente elevada en el run de Ollama, que eleva el umbral T_{hum} hasta 13,8 tok/s, por encima del *throughput* alcanzado.

Síntesis de hallazgos

La serie E4 responde a la cuestión CI-05 y confirma la hipótesis H-E4 (véase la Sección 3.1): llama.cpp es el motor preferible para el despliegue en la Raspberry Pi 5. Aunque el *throughput* de generación es comparable entre ambos motores (e incluso favorable a Ollama en dos modelos), llama.cpp ofrece una latencia de primer *token* drásticamente menor, hasta más de seis veces inferior, y una figura de mérito igual o superior en la mayoría de los casos. La sobrecarga de inicialización por petición de Ollama, que en el peor caso supera los 12 s, lo hace inadecuado para interacción conversacional fluida, pese a la comodidad operativa que aporta su capa de gestión. La recomendación para la integración es, por tanto, emplear llama.cpp como motor de inferencia cuando la latencia de respuesta sea un requisito, reservando Ollama para escenarios donde prime la facilidad de despliegue sobre la inmediatez.

4.1.6. Serie E5: acelerador Hailo-10H

La serie E5 evalúa el acelerador de inferencia Hailo-10H frente a la ejecución en CPU, ejecutando el modelo completo sobre el acelerador. Responde a la cuestión CI-06 y determina si la aceleración por *hardware* aporta ventajas de rendimiento o de eficiencia energética en la plataforma. A diferencia de las series anteriores, el banco de pruebas de Hailo no expone las fases de inferencia, el ancho de banda de memoria ni las probabilidades de los *tokens*, por lo que no es posible calcular la MBU, la eficiencia de cómputo (η_{CPU}) ni la perplejidad. En consecuencia, la evaluación se realiza con la **figura de mérito reducida** $FoM_{red} = \sqrt{T_{norm} \cdot \varepsilon_{norm}}$, media geométrica del *throughput* y la eficiencia energética, ambos ponderados por el número de parámetros y normalizados respecto a la ejecución en CPU del mismo modelo. Bajo este anclaje, un valor $FoM_{red} > 1$ indica ventaja del acelerador y < 1 , ventaja de la CPU.

Nota sobre el número de parámetros. Dado que el *backend* de Hailo no expone el número de parámetros del modelo, este se obtiene del fichero `resumen.json` de la ejecución equivalente en CPU (`llama.cpp`), y se emplea el mismo valor en ambas ramas. Como el anclaje de la FoM reducida es la propia ejecución en CPU del modelo, el factor de parámetros se cancela en los dos ejes y la FoM reducida expresa exclusivamente la ganancia relativa Hailo/CPU.

Nota metodológica sobre la medición energética. A diferencia de las series E0–E4, donde el coste energético se estima a partir de la potencia media del sensor interno del SoC ($\bar{P}_{run} \cdot T_{run}$), la serie E5 emplea la lectura acumulada del vatímetro externo (`total_kwh` del `resumen.json`) para *ambos* sustratos —CPU y Hailo—, con objeto de incluir el consumo del HAT+ y las pérdidas de la fuente de alimentación. Esta lectura abarca el encendido del sistema más aproximadamente cinco minutos de funcionamiento en reposo previos a la ventana de inferencia propiamente dicha, por lo que los valores de J/token reportados en la Tabla 4.7 constituyen **cotas superiores** del coste real de inferencia. Puesto que el protocolo de medida es idéntico en ambas ramas, el sesgo es sistemático y se cancela parcialmente en el cociente ε_{norm} , lo que hace los valores de ε_{norm} y FoM_{red} *comparables entre sí* dentro de la serie E5. Los valores absolutos de J/token no son, en cambio, directamente comparables con los reportados en las series E0–E4.

Tabla 4.7: Comparación del acelerador Hailo-10H frente a la ejecución en CPU (`llama.cpp`) por modelo. Se reportan *throughput* (tok/s) y coste energético (J/token) en ambos sustratos, los factores normalizados de *throughput* (T_{norm}) y de eficiencia energética (ε_{norm}), la figura de mérito reducida (FoM_{red}) y la usabilidad en cada sustrato. Valores normalizados respecto a la ejecución en CPU del mismo modelo. ⁽¹⁾ J/token obtenido del vatímetro externo (`total_kwh`); incluye encendido y ~ 5 min de reposo previos a la inferencia: cota superior, comparable entre sustratos, no con las series E0–E4.

Modelo	tok/s		J/token ⁽¹⁾		T_{norm}	ε_{norm}	FoM_{red}	Us.
	CPU	Hailo	CPU	Hailo				
DeepSeek-R1-1.5B	10.25	6.78	1.139	1.114	0.662	1.023	0.822	✓/✓
Llama-3.2-3B	5.03	2.65	2.462	1.061	0.528	2.320	1.107	✗/✓

Los resultados refutan la hipótesis previa en lo que respecta al *throughput*: el acelerador Hailo-10H *no* proporciona aceleración, sino que lo reduce de forma sustancial en ambos

modelos, un 34 % en DeepSeek-R1-1.5B (de 10,3 a 6,8 tok/s) y un 47 % en Llama-3.2-3B (de 5,0 a 2,7 tok/s). Sin embargo, el juicio global varía al incorporar la dimensión energética. La figura de mérito reducida es inferior a la unidad en DeepSeek-R1-1.5B ($FoM_{red} = 0,822$), lo que confirma la ventaja de la CPU para ese modelo; sin embargo, alcanza 1,107 en Llama-3.2-3B, superando la paridad y evidenciando que, en balance combinado de velocidad y eficiencia energética, el acelerador resulta preferible a la CPU para el modelo de mayor tamaño.

No obstante, el comportamiento en la dimensión energética difiere cualitativamente entre modelos. En DeepSeek-R1-1.5B el acelerador es más lento y no ofrece ventaja energética apreciable: el coste por *token* es prácticamente idéntico en ambos sustratos ($\varepsilon_{norm} = 1,023$, diferencia inferior al 3 %), de modo que el acelerador es inferior a la CPU en los dos ejes y su FoM reducida queda en 0,822. En cambio, en Llama-3.2-3B la ganancia energética del Hailo-10H es muy marcada: pese a la pérdida de velocidad, el coste energético por *token* se reduce en un 57 % ($\varepsilon_{norm} = 2,320$, de 2,462 a 1,061 J/tok). Esta ventaja compensa con creces la penalización de velocidad y eleva la FoM reducida de Llama-3.2-3B hasta 1,107, superando la paridad con la CPU.

En términos de usabilidad, la ralentización del acelerador tiene consecuencias prácticas (véase la Tabla 4.7): Llama-3.2-3B, usable en CPU, cae por debajo del umbral T_{hum} al ejecutarse en Hailo-10H, mientras que DeepSeek-R1-1.5B permanece usable en ambos sustratos aunque con menor margen. El acelerador puede, por tanto, comprometer la interactividad de modelos que en CPU sí la alcanzaban.

Síntesis de hallazgos

La serie E5 responde a la cuestión CI-06 y refuta parcialmente la hipótesis de partida. El acelerador Hailo-10H no aporta aceleración de *throughput* (lo reduce entre un 34 y un 47 %) en ninguno de los dos modelos evaluados. Sin embargo, el balance energético es radicalmente diferente según el modelo: en DeepSeek-R1-1.5B el acelerador es prácticamente neutro en términos energéticos ($\varepsilon_{norm} = 1,023$, diferencia < 3 %) y su FoM reducida queda en 0,822, siendo la CPU preferible; en Llama-3.2-3B, en cambio, el Hailo-10H reduce el consumo por *token* en un 57 % ($\varepsilon_{norm} = 2,320$), lo que compensa la pérdida de velocidad y eleva la FoM reducida hasta 1,107, situando al acelerador por encima de la CPU en el criterio combinado. La restricción de usabilidad matiza esta ventaja: Llama-3.2-3B deja de ser interactiva sobre el Hailo-10H al caer por debajo del umbral T_{hum} , por lo que la superioridad del acelerador en FoM reducida es relevante únicamente en aplicaciones donde la latencia de respuesta no sea un requisito prioritario. Este resultado debe interpretarse en el contexto de madurez del ecosistema de *software* para inferencia generativa sobre Hailo, considerablemente menos optimizado que la pila de CPU empleada en el resto del trabajo; la evolución de dicho ecosistema, así como la caracterización de un mayor número de modelos sobre el acelerador, se identifican como líneas de trabajo futuro. En el estado actual, el Hailo-10H se recomienda únicamente en escenarios con restricción energética estricta, tolerancia a la pérdida de interactividad y uso de modelos de tamaño suficiente (Llama-3.2-3B o superiores) para que la ventaja de eficiencia compense la penalización en velocidad.

Síntesis global

Las seis series experimentales convergen en una imagen coherente del comportamiento de los SLM sobre la Raspberry Pi 5, articulada en torno a tres ejes transversales.

El ancho de banda de memoria es el factor limitante dominante. La plataforma opera en régimen *memory-bound* en la fase de decodificación, con valores de MBU_{corr} entre el 50 y el 72 % en condiciones nominales (véase la Figura 4.2). Este cuello de botella explica de forma unificada los resultados de cuatro series: en E1, reducir la cuantización alivia el tráfico de memoria y eleva el *throughput* de forma monótona hasta llegar a aproximadamente 4 bits (Figura 4.6); en E2, ampliar la ventana de contexto incrementa el tráfico de la caché KV y degrada el *throughput* de forma apreciable, con pérdidas de hasta el 22 % en los modelos con mayor dependencia del MBU (Figura 4.10); en E3, el batch size de prefill resulta indiferente porque la decodificación no encuentra paralelismo adicional en flujo único (Figura 4.13); y en E4, la arquitectura sin capa de servidor de llama.cpp reduce la latencia de primer *token* sin alterar el *throughput* de decodificación (véase la Tabla 4.6).

La configuración de referencia óptima queda bien definida. La combinación de refrigeración activa, cuantización Q4_K_M y motor llama.cpp constituye la configuración recomendada para despliegues interactivos sobre la plataforma: cubre la usabilidad en los seis modelos, minimiza la latencia de primer *token* y ofrece el mejor equilibrio entre *throughput*, huella de memoria y coste energético (Figura 4.1). La evaluación de calidad con PerplexitySystem sobre WikiText-2 (Figura 4.7) confirma que esta configuración degrada la perplejidad menos de un 5 % respecto a Q8_0, validando que el coste en calidad es operativamente aceptable. Q4_0 y Q3_K_M pueden sustituir a Q4_K_M bajo restricción severa de recursos, con la salvedad de que Q4_0 muestra mayor degradación de calidad que Q4_K_M al mismo nivel de bits (véase la Tabla 4.2).

El acelerador hardware introduce un balance asimétrico y dependiente del modelo. La serie E5 aporta la única desviación respecto a la configuración CPU de referencia: el Hailo-10H no mejora el *throughput*, sino que lo reduce entre un 34 y un 47 % (véase la Tabla 4.7)—pero ofrece una ventaja energética sustancial para modelos de tamaño suficiente (véase la Tabla 4.7). Para Llama-3.2-3B, la reducción del 57 % en el coste energético por *token* compensa la pérdida de velocidad y eleva la figura de mérito reducida por encima de la paridad ($FoM_{red} = 1,107$, véase la Tabla 4.7). Para DeepSeek-R1-1.5B, el balance es neutro en energía ($\varepsilon_{norm} = 1,023$) y desfavorable en velocidad, por lo que la CPU es preferible ($FoM_{red} = 0,822$). Esta asimetría, condicionada por la madurez del ecosistema hailo-ollama, indica que el hardware de aceleración puede ser rentable en escenarios con restricción energética y modelos de tamaño medio o superior, pero no sustituye a la CPU en el caso general con la configuración actual.

En conjunto, los resultados responden a las nueve cuestiones de investigación planteadas y confirman que la Raspberry Pi 5 es una plataforma viable para la inferencia local de SLM en modo interactivo, con la configuración adecuada y dentro de los límites de tamaño y cuantización identificados en este trabajo.

5. Conclusiones

En el presente TFG se ha abordado el análisis y la optimización de modelos de lenguaje pequeños (SLM) sobre la Raspberry Pi 5, evaluando de forma sistemática el impacto de seis factores experimentales (refrigeración, cuantización, tamaño de contexto, *batch* de *prefill*, motor de inferencia y acelerador hardware) sobre el rendimiento, la eficiencia energética y la usabilidad interactiva. El trabajo se articula en torno a dos contribuciones principales: el sistema de medida *MonitorSystem* (C++), que integra la captura de métricas de inferencia y hardware en tiempo real, y la librería de análisis *monitorviz* (Python), que automatiza el cálculo de métricas derivadas y la generación de visualizaciones.

Cumplimiento de objetivos

Los resultados obtenidos permiten afirmar que los cinco objetivos específicos definidos en la Sección 1.2 se han alcanzado satisfactoriamente.

El primer objetivo, el estudio de los SLM del estado del arte compatibles con dispositivos embebidos, se aborda en el Capítulo 2, que revisa la evolución de los LLM, las arquitecturas Transformer y sus variantes, y las técnicas de reducción de modelos, culminando en la selección razonada de seis modelos representativos evaluados en el Capítulo 4.

El segundo objetivo, la implementación de modelos sobre dispositivos embebidos incluyendo aceleradores hardware, se materializa en el Capítulo 3, específicamente en las series E0–E5, que cubren los dos motores de inferencia (llama.cpp y Ollama) sobre los seis modelos seleccionados, y la evaluación del acelerador Hailo-10H en la serie E5 mediante hailo-ollama.

El tercer y cuarto objetivos, el desarrollo y la visualización del sistema de medida, se satisfacen con *MonitorSystem* y *monitorviz* respectivamente, cuyas especificaciones de requisitos se recogen en los Anexos A y B. Ambas herramientas son reutilizables de forma independiente de los experimentos concretos de este trabajo.

El quinto objetivo, el análisis de resultados y la extracción de conclusiones y recomendaciones, se desarrolla a lo largo del Capítulo 4 y se sintetiza en la presente sección.

Conclusiones principales

Las siguientes conclusiones responden a las cuestiones de investigación CI-01–CI-09 planteadas en la Sección 3.1.

- **La refrigeración activa es un requisito habilitante, no una optimización (CI-01, CI-07).** Con ventilador activo, los seis modelos superan el umbral de usabilidad interactiva T_{hum} ; sin él, cuatro de los seis quedan por debajo. El efecto térmico reduce la eficiencia de cómputo η_{CPU} desde $\approx 0,97$ hasta $\approx 0,40$ y provoca incrementos de *throughput* de hasta un 111,5 % al activar el ventilador. En condiciones nominales, ningún modelo

domina de forma absoluta: Ministral-3B maximiza la figura de mérito global ($FoM_{full} = 1,058$), mientras que DeepSeek-R1-1.5B ofrece el mejor compromiso entre velocidad, huella de memoria y latencia de primer *token*.

- **La cuantización a 4 bits es el punto óptimo en una plataforma *memory-bound* (CI-02).** La FoM_{full} crece de forma monótona al descender desde Q8_0 hacia Q4 y Q3, con máximos en Q4_0 o Q3_K_M según el modelo. Q8_0 cae por debajo de T_{hum} en cuatro modelos; también Granite-micro y Ministral-3B quedan fuera del rango de usabilidad en Q5_K_M. La precisión BF16, viable solo en Llama-3.2-1B, hunde el *throughput* hasta 1,6 tok/s y expulsa al modelo del rango de usabilidad. La evaluación de calidad con PerplexitySystem sobre WikiText-2 confirma que Q4_K_M degrada la perplejidad menos de un 5 % respecto a Q8_0 en todos los modelos (+1,7 % en Ministral-3B, +3,2 % en Llama-3.2-3B, +3,5 % en Llama-3.2-1B, +4,3 % en Granite), mientras que Q5_K_M es virtualmente sin pérdidas (< 1,4 %). Q4_K_M resulta por tanto el punto de equilibrio óptimo en ambas dimensiones: rendimiento y calidad.
- **El tamaño de contexto es un parámetro de ajuste fino, no un factor crítico (CI-03).** El TTFT permanece esencialmente constante en el rango 512–5 120 *tokens*. La huella en RAM crece de forma suave y predecible (entre un 8 y un 21 % según el modelo, por el coste creciente de la caché KV), sin que ningún modelo supere los 4 GB ni comprometa el límite de la plataforma. El *throughput* decrece de forma apreciable con el contexto, con pérdidas de hasta el 22 % en los modelos con mayor dependencia del MBU. La recomendación práctica es dimensionar el contexto al mínimo que cubra el caso de uso.
- **El batch size de prefill es indiferente en inferencia de flujo único (CI-04).** En ausencia de peticiones concurrentes, el *batching* de prefill no encuentra paralelismo adicional que explotar, y el *throughput*, el TTFT, el consumo energético y la FoM permanecen constantes en todo el rango evaluado (128–2 048 *tokens*).
- **llama.cpp es el motor preferible cuando la latencia importa (CI-05).** El *throughput* de decodificación es comparable entre llama.cpp y Ollama, pero la sobrecarga de inicialización por petición de Ollama multiplica el TTFT hasta en un factor de seis, llegando a superar los 12 s en Ministral-3B, lo que lo hace inadecuado para interacción conversacional fluida. Se recomienda llama.cpp para despliegues interactivos y Ollama para escenarios donde prime la facilidad operativa sobre la inmediatez.
- **El acelerador Hailo-10H ofrece un balance dependiente del modelo (CI-06).** No proporciona aceleración de *throughput* (lo reduce entre un 34 y un 47 %) pero su balance varía según el modelo: para DeepSeek-R1-1.5B el rendimiento combinado es inferior al de la CPU ($FoM_{red} = 0,822$); para Llama-3.2-3B la ventaja energética del 57 % compensa la pérdida de velocidad y eleva la FoM_{red} hasta 1,107, superando a la CPU en balance global. La evaluación está condicionada por la madurez del ecosistema hailo-ollama, considerablemente menos optimizado que llama.cpp.
- **La fase de decodificación concentra el coste temporal y energético (CI-08).** La carga del modelo es despreciable en operación (0,06–0,29 s por *prompt*), lo que respalda la estrategia de mantener el modelo residente en memoria entre peticiones.
- **Un marco experimental unificado y reproducible es factible en esta plataforma (CI-09).** MonitorSystem integra métricas temporales, energéticas y térmicas en un único artefacto JSON determinista, y monitorviz automatiza su análisis comparativo, proporcionando el andamiaje necesario para extender el estudio a nuevas plataformas y modalidades.

Limitaciones

Las siguientes limitaciones acotan el alcance de las conclusiones anteriores y deben tenerse en cuenta al generalizar los resultados.

- **Plataforma única.** Todos los experimentos se ejecutan exclusivamente sobre la Raspberry Pi 5 (SoC BCM2712, 8 GB LPDDR4X). Los resultados no son directamente transferibles a otras plataformas de computación en el borde (Jetson Orin Nano, Orange Pi, RISC-V, etc.), cuyas características de ancho de banda de memoria, jerarquía de caché y eficiencia energética difieren de forma sustancial.
- **Cobertura parcial de la medición energética en las series E0–E4.** La potencia se obtiene del sensor interno del SoC mediante `vcgencmd pmic_read_adc`, que mide únicamente el consumo de la placa base, excluyendo el Active Cooler y los periféricos USB. Los valores de J/token reportados en E0–E4 son por tanto cotas inferiores del consumo real del sistema (véase la Sección 3.2.3). La serie E5 emplea un vatímetro externo que sí contempla el consumo total, lo que introduce una asimetría que impide comparar valores absolutos de J/token entre E5 y las series E0–E4.
- **Cobertura limitada de aceleradores y modelos en E5.** La serie E5 queda restringida a los dos modelos distribuidos por Hailo AI en formato HEF precompilado (DeepSeek-R1-1.5B y Llama-3.2-3B). La ausencia de un flujo público para compilar modelos propios en formato HEF impide evaluar el acelerador con el conjunto completo de seis modelos ni controlar los parámetros de contexto y batch en compilación.
- **Escenario de flujo único.** Todos los experimentos corresponden a inferencia de un único flujo conversacional sin concurrencia. Los resultados del batch size (E3) y de la comparativa de motores (E4) no son directamente aplicables a despliegues con múltiples usuarios simultáneos, donde el batching de prefill y la capa de gestión de Ollama sí pueden ofrecer ventajas.
- **Alcance de la evaluación de calidad.** La perplejidad estándar sobre WikiText-2 (PerplexitySystem) es comparable entre cuantizaciones del mismo modelo base con tokenizador compartido, pero no entre modelos distintos ni entre arquitecturas. No constituye por tanto una medida de calidad absoluta ni captura idoneidad para tareas específicas del dominio objetivo. Esta limitación motiva la línea de trabajo futuro relativa al desarrollo de un conjunto de pruebas específico de biodiversidad.
- **Modelos de texto exclusivamente.** El estudio se circunscribe a SLM generativos de texto. No se evalúan modelos multimodales (visión, audio), cuya relevancia para el dominio de aplicación objetivo es directa.
- **Solo cuantización post-entrenamiento en formato GGUF.** No se evalúan experimentalmente otras técnicas de compresión contempladas en el estado del arte (cuantización con conocimiento del entrenamiento, poda, destilación de conocimiento, adaptadores LoRA), cuyo impacto en la plataforma queda fuera del alcance de este trabajo.
- **Dataset de evaluación no específico del dominio.** Los 15 prompts de *instruction_following_eval* son de propósito general y no representan los casos de uso del dominio de aplicación previsto (monitorización de biodiversidad), lo que impide extraer conclusiones sobre la idoneidad de los modelos evaluados para dicho dominio.

Líneas de trabajo futuro

El presente trabajo abre varias líneas de continuación directa, enmarcadas en la beca de colaboración asociada al TFG y en la línea de investigación del grupo en sistemas embebidos para inteligencia artificial aplicada a la monitorización de la biodiversidad.

- **Ampliación de la cobertura experimental.** Extender las series E0–E4 a un conjunto más amplio de modelos en el rango 0,5–8 B de parámetros, incluyendo variantes con diversas arquitecturas y optimizaciones arquitectónicas (atención agrupada, mezcla de expertos, modelos de estado recurrente), versiones ajustadas para instrucción en el idioma del dominio objetivo, y modelos multimodales (visión, audio y LLM multimodales) orientados a la detección y clasificación de especies. Una vez que el ecosistema Hailo AI permita compilar modelos propios en formato HEF, ampliar igualmente la serie E5 al conjunto completo de modelos y controlar los parámetros de contexto y batch en compilación.
- **Extensión de MonitorSystem y monitorviz.** Ampliar el sistema de medida para instrumentar modelos multimodales (audio, texto), incorporando métricas de latencia, consumo y calidad específicas de estas modalidades, y para soportar nuevas plataformas hardware más allá de la Raspberry Pi 5 (p. ej., Jetson Orin Nano).
- **Técnicas de compresión y adaptación del conocimiento.** Explorar experimentalmente técnicas de compresión más allá de la cuantización post-entrenamiento en formato GGUF (poda, destilación de conocimiento) y técnicas de adaptación al dominio (ajuste fino supervisado, adaptadores LoRA), evaluando su impacto sobre el triángulo rendimiento–eficiencia–calidad con el mismo marco de métricas desarrollado en este trabajo.
- **Evaluación en escenarios avanzados.** Extender E3 y E4 a despliegues con múltiples flujos conversacionales simultáneos, donde el batching de prefill y la arquitectura de servidor de Ollama pueden aportar ventajas significativas respecto al escenario de flujo único evaluado. Integrar y evaluar además mecanismos de *tool calling* que permitan al modelo solicitar la activación de sensores y actuadores (cámara, micrófono, GPS, sensores ambientales) y coordinarse con otros sistemas de monitorización inteligente en el borde (fototrampeo, bioacústica) de forma autónoma, avanzando hacia sistemas de monitorización adaptativa en el borde.
- **Evaluación de idoneidad para el dominio de biodiversidad.** Esta línea requiere tres pasos secuenciales:
 1. *Corpus de conocimiento del dominio.* Compilar un corpus textual específico de biodiversidad que incluya informes de campo, artículos científicos, libros de texto, guías de especies y documentación técnica relevante, con el que adaptar los modelos al vocabulario, los conceptos y el razonamiento propio del dominio mediante ajuste fino o técnicas RAG. En el caso de modelos multimodales, el corpus deberá complementarse con datos procedentes de sistemas de clasificación de fauna, flora y ruido ambiental.
 2. *Suite de evaluación funcional.* Diseñar un conjunto de pruebas que mida la adecuación funcional del modelo al caso de uso real: generación de reportes en lenguaje natural a partir de observaciones sensoriales, respuesta a consultas sobre fauna y flora local e interpretación de datos de monitorización en campo.
 3. *Integración en MonitorSystem.* Incorporar la suite como módulo de evaluación de

idoneidad, completando el marco con una capa de calidad orientada a la aplicación que complemente las métricas de rendimiento puro desarrolladas en este trabajo.

A. Especificación de Requisitos del Software (MonitorSystem)

Sistema	MonitorSystem
Versión	1.0
Fecha	2026-06-03
Metodología	Ingeniería de Requisitos (IISSI – Universidad de Sevilla)
Estado	Completado

A.1. Introducción

A.1.1. Propósito

El presente documento especifica los requisitos del sistema **MonitorSystem**, herramienta de *benchmarking* desarrollada en C++ para la medición y registro del rendimiento de modelos de lenguaje de gran tamaño (LLM) ejecutados sobre *hardware* de computación en el borde (*edge computing*). El documento se elabora en el marco del Trabajo de Fin de Grado (TFG) del autor y sigue la metodología de Ingeniería de Requisitos del IISSI de la Universidad de Sevilla.

A.1.2. Alcance

El sistema se encarga de:

- Lanzar experimentos de inferencia sobre LLMs utilizando distintos motores de inferencia.
- Recoger métricas de rendimiento de inferencia (tiempos, recuentos de *tokens*, probabilidades por *token*) durante la ejecución de un conjunto de *prompts*.
- Recoger métricas de estado *hardware* (temperatura, frecuencia, potencia, memoria, *throttling*) de forma simultánea a la inferencia.
- Persistir los resultados en un formato estructurado (JSON/JSONL) para su análisis posterior por herramientas externas.

Quedan fuera del alcance de este documento:

- El componente de visualización y análisis de resultados (repositorio Python separado, en desarrollo).
- La gestión y despliegue de los servidores de inferencia (Ollama, hailo-ollama).
- La compilación cruzada para plataformas distintas de Raspberry Pi 5.

A.1.3. Definiciones, acrónimos y abreviaturas

Véase la Sección [A.2](#).

A.1.4. Referencias

- README del proyecto (README.md y README_eng.md).
- *Dataset de prompts: instruction_following_eval*, Google Research. Disponible en: https://github.com/google-research/google-research/tree/master/instruction_following_eval
- API Ollama: documentación oficial de Ollama. Disponible en: <https://ollama.com>
- llama.cpp: repositorio ggml-org/llama.cpp. Disponible en: <https://github.com/ggml-org/llama.cpp>
- hailo-ollama: repositorio hailo-ai/hailo_model_zoo_genai. Disponible en: https://github.com/hailo-ai/hailo_model_zoo_genai

A.2. Glosario de términos

Término	Definición
LLM	<i>Large Language Model</i> . Modelo de lenguaje de gran tamaño basado en redes neuronales transformadoras.
Inferencia	Proceso de generación de texto por parte de un LLM a partir de un <i>prompt</i> de entrada.
Token	Unidad mínima de procesamiento de texto en un LLM. Puede corresponder aproximadamente a una palabra o subpalabra.
Prompt	Texto de entrada suministrado al LLM para obtener una respuesta.
Log-probabilidad	Logaritmo neperiano de la probabilidad asignada por el modelo a cada <i>token</i> generado. Permite analizar la distribución de probabilidad de la salida.
Motor de inferencia (<i>inference engine</i>)	Componente <i>software</i> que ejecuta la inferencia. El sistema soporta OLLAMA, LLAMA (llama.cpp) y HAILO_OLLAMA.
OLLAMA	Servidor de inferencia local que gestiona modelos LLM mediante una API HTTP REST.
llama.cpp	Biblioteca de inferencia en C/C++ para modelos en formato GGUF, con soporte para CPU.
HAILO_OLLAMA	Servidor de inferencia para acelerador <i>hardware</i> Hailo, accesible mediante API HTTP similar a Ollama.
Hailo	Acelerador <i>hardware</i> neuronal (HAT para Raspberry Pi) que permite acelerar la inferencia de modelos LLM.
GGUF	Formato de fichero binario para almacenar pesos y metadatos de modelos LLM compatibles con llama.cpp.
Batch size	Número de <i>tokens</i> procesados en paralelo durante la fase de <i>prefill</i> (evaluación del <i>prompt</i>).
Context size	Tamaño de la ventana de contexto del modelo, en <i>tokens</i> .
Temperatura	Hiperparámetro de muestreo que controla la aleatoriedad de la generación de texto.
Semilla (<i>seed</i>)	Valor entero que inicializa el generador de números pseudoaleatorios para garantizar la reproducibilidad.
Throttling	Reducción de la frecuencia de funcionamiento del procesador por causas térmicas o de tensión.

Término	Definición
JSONL	Formato de fichero donde cada línea es un objeto JSON independiente (<i>JSON Lines</i>).
Benchmark	Experimento de medición de rendimiento bajo condiciones controladas y reproducibles.
Edge computing	Paradigma de computación en el que el procesamiento se realiza en dispositivos de bajo consumo cercanos a la fuente de datos, sin depender de la nube.
RPi5	Raspberry Pi 5, plataforma <i>hardware</i> de referencia del sistema.
TFG	Trabajo de Fin de Grado. Proyecto académico en el que se enmarca este sistema.
Prefill	Fase de inferencia en la que se procesa el <i>prompt</i> de entrada y se genera el estado KV-cache.
Decode	Fase de inferencia en la que se generan los <i>tokens</i> de respuesta de forma autoregresiva.
resumen.json	Fichero JSON de metadatos de una ejecución, generado al finalizar cada experimento.
hw_metrics	Fichero JSONL con las métricas <i>hardware</i> muestreadas periódicamente durante la ejecución.
prompt_metrics	Fichero JSONL con las métricas de inferencia por <i>prompt</i> .

A.3. Objetivos del sistema (OBJ)

Los objetivos se expresan como historias de usuario épicas en formato Mike Cohn.

ID	Historia de usuario
OBJ-01	Como investigador, quiero ejecutar de forma automatizada un conjunto de <i>prompts</i> sobre un LLM en Raspberry Pi 5, para obtener métricas de inferencia reproducibles que permitan el análisis comparativo entre experimentos.
OBJ-02	Como investigador, quiero monitorizar el estado <i>hardware</i> de la Raspberry Pi 5 durante la inferencia, para caracterizar el consumo energético y térmico asociado a la ejecución de LLMs en <i>hardware</i> de bajo coste.
OBJ-03	Como investigador, quiero ejecutar experimentos equivalentes con distintos motores de inferencia (OLLAMA, llama.cpp, HAILO_OLLAMA), para comparar su rendimiento relativo bajo las mismas condiciones experimentales.
OBJ-04	Como investigador, quiero parametrizar los experimentos mediante hiperparámetros configurables (temperatura, <i>batch size</i> , <i>context size</i> , semilla) y distintos aceleradores hardware, para evaluar el impacto de cada configuración en el rendimiento y el comportamiento del modelo.
OBJ-05	Como analista de datos, quiero acceder a los resultados de los experimentos en un formato estructurado y estándar (JSON/JSONL), para poder procesarlos, visualizarlos y analizarlos con herramientas externas sin modificar el sistema de monitorización.

(continuación de objetivos)

ID	Historia de usuario
OBJ-06	Como investigador, quiero estudiar la viabilidad del despliegue de LLMs en <i>hardware</i> de computación en el borde de bajo coste, para determinar qué modelos y configuraciones son aplicables en escenarios reales con recursos limitados.

A.4. Requisitos de información (RI)

A.4.1. RI-01: Configuración de experimento

Descripción: Parámetros que definen completamente un experimento de *benchmarking*. Se suministran al sistema como fichero JSON o cadena JSON *inline*.

Campo	Tipo	Oblig.	Descripción
inference_engine	."LLAMA" "LLAMA" "HAILO_OLLAMA"	Obligatorio	Motor de inferencia a utilizar.
test_type	"TYPE_0" "TYPE_1" "TYPE_2"	Obligatorio	Tipo de experimento a ejecutar.
model_path_or_name	string	Obligatorio	Nombre del modelo (Ollama/Hailo) o ruta al fichero GGUF (llama.cpp).
batch_size	entero ≥ 0	Obligatorio	Tamaño de <i>batch</i> de procesado de <i>tokens</i> .
context_size	entero ≥ 0	Obligatorio	Tamaño de la ventana de contexto en <i>tokens</i> .
seed	entero	Obligatorio	Semilla de aleatoriedad. El valor -1 implica semilla aleatoria.
num_prompts	entero [1, 541]	Obligatorio	Número de <i>prompts</i> a ejecutar.
temperature	real ≥ 0	Obligatorio	Temperatura de muestreo.
hardware_period	real > 0	Obligatorio	Período entre muestras <i>hardware</i> , en segundos.
annotations	string u objeto JSON	Opcional	Descripción libre del experimento o metadatos adicionales.
ollama_url	string (URL)	Opc. (OLLAMA)	URL del servidor Ollama. Por defecto: <code>http://localhost:11434</code> .
hailo_server_host	string	Opc. (HAILO)	Host del servidor hailo-ollama. Por defecto: <code>localhost</code> .

(continuación de RI-01)

Campo	Tipo	Oblig.	Descripción
hailo_server_port	entero > 0	Opc. (HAILO)	Puerto del servidor hailo-ollama. Por defecto: 8000.

A.4.2. RI-02: Métricas de inferencia por *prompt*

Descripción: Registro de los resultados y tiempos de una inferencia individual. Se almacena una entrada por *prompt* en el fichero *_prompt_metrics_*.jsonl.

Campo	Tipo	Oblig.	Descripción
start_timestamp_ns	entero (ns)	Obligatorio	Timestamp de inicio de la solicitud (cliente).
finish_timestamp_ns	entero (ns)	Obligatorio	Timestamp de fin de la solicitud (cliente).
total_duration_ns	entero (ns)	Obligatorio	Duración total reportada por el motor.
prompt_eval_duration_ns	entero (ns)	Obligatorio	Duración de la fase de <i>prefill</i> . 0 si HAILO_OLLAMA.
eval_duration_ns	entero (ns)	Obligatorio	Duración de la fase de <i>decode</i> . 0 si HAILO_OLLAMA.
load_duration_ns	entero (ns)	Obligatorio	Tiempo de carga del modelo. 0 si HAILO_OLLAMA.
prompt_eval_count	entero	Obligatorio	Número de <i>tokens</i> del <i>prompt</i> . 0 si HAILO_OLLAMA.
eval_count	entero	Obligatorio	Número de <i>tokens</i> generados.
answer	string	Obligatorio	Texto de la respuesta generada.
tokenProb	string (JSON)	Obligatorio	Log-probabilidades por <i>token</i> (Ollama/llama.cpp) o "NONE" si HAILO_OLLAMA.
engine	string	Obligatorio	Motor de inferencia utilizado.
model	string	Obligatorio	Nombre o ruta del modelo.
prompt_id	entero	Obligatorio	Índice del <i>prompt</i> en el <i>dataset</i> (0-based).

A.4.3. RI-03: Métricas *hardware* periódicas

Descripción: Estado *hardware* de la Raspberry Pi 5 muestreado con período configurable. Se almacena una entrada por muestra en el fichero *_hw_metrics_*.jsonl.

Campo	Tipo	Oblig.	Descripción
timestamp__	entero (ms)	Obligatorio	Timestamp de la muestra.
temperature__	real (°C)	Obligatorio	Temperatura de la CPU.
frequency__	array real (GHz)	Obligatorio	Frecuencia por núcleo.
voltage__	real (V)	Obligatorio	Tensión del núcleo.
fan_speed__	real (RPM)	Obligatorio	Velocidad del ventilador.
internalpower__	real (W)	Obligatorio	Potencia interna medida por el sensor de la RPi5. No incluye HATs ni periféricos USB.
throttling__	objeto JSON	Obligatorio	Estado de <i>throttling</i> (under_voltage, freq_capped, throttled, soft_throttled y sus <i>flags</i> de ocurrencia histórica).
mem_total__	entero (bytes)	Obligatorio	Memoria RAM total.
mem_used__	entero (bytes)	Obligatorio	Memoria RAM usada.
mem_percent__	real (%)	Obligatorio	Porcentaje de memoria RAM usada.
swap_total__	entero (bytes)	Obligatorio	Memoria <i>swap</i> total.
swap_used__	entero (bytes)	Obligatorio	Memoria <i>swap</i> usada.
swap_percent__	real (%)	Obligatorio	Porcentaje de <i>swap</i> usada.
cpu_usage__	real (%)	Obligatorio	Porcentaje de uso de CPU global.
cpu_ticks__	objeto JSON	Obligatorio	Contadores de tiempo de CPU por tipo (user, nice, system, idle, iowait, irq, softirq, steal).

A.4.4. RI-04: Resumen de ejecución

Descripción: Metadatos completos de un experimento, generados al finalizar la ejecución. Se almacena en el fichero resumen.json.

Campo	Tipo	Oblig.	Descripción
inference_engine	string	Obligatorio	Motor utilizado.
test_type	string	Obligatorio	Tipo de test.
batch_size	entero	Obligatorio	<i>Batch size</i> configurado.
context_size	entero	Obligatorio	Tamaño de contexto configurado.
seed	entero	Obligatorio	Semilla utilizada.

(continuación de RI-04)

Campo	Tipo	Oblig.	Descripción
num_prompts	entero	Obligatorio	Número de <i>prompts</i> ejecutados.
temperature	real	Obligatorio	Temperatura de muestreo.
model_path_or_name	string	Obligatorio	Identificador del modelo.
hardware_period	real	Obligatorio	Período de muestreo <i>hardware</i> .
timestamp_run_start	entero (ns)	Obligatorio	Timestamp de inicio de la ejecución.
timestamp_run_end	entero (ns)	Obligatorio	Timestamp de fin de la ejecución.
annotations	objeto JSON	Obligatorio	Metadatos libres del experimento, incluyendo el subcampo <code>model_info</code> .
annotations. model_info	objeto JSON	Condicional	Metadatos de arquitectura del modelo (<code>n_params</code> , <code>architecture</code> , <code>embedding_length</code> , <code>n_layers</code> , <code>max_context</code> , <code>quantization</code>). Puede ser null si no se pudo obtener.
og_config_json	string (JSON)	Obligatorio	Configuración de entrada original serializada.
hailo_server_host	string	Solo HAILO	Host del servidor hailo-ollama.
hailo_server_port	entero	Solo HAILO	Puerto del servidor hailo-ollama.

A.4.5. RI-05: *Dataset de prompts*

Descripción: Conjunto de *prompts* de evaluación embebido en el binario en tiempo de compilación.

Atributo	Valor
Origen	<i>Dataset instruction_following_eval</i> , Google Research.
Cardinalidad	541 <i>prompts</i> .
Formato fuente	JSONL (campo <code>prompt</code> por línea).
Disponibilidad	Embebido en el binario; no requiere fichero externo en tiempo de ejecución.
Selección	Se utilizan los primeros <code>num_prompts</code> <i>prompts</i> en orden de aparición.

A.4.6. RI-06: Metadatos de arquitectura del modelo

Descripción: Información sobre la arquitectura interna del modelo utilizado, extraída al inicio de cada ejecución.

Campo	Tipo	Oblig.	Descripción
n_params	entero	Opcional	Número de parámetros del modelo.
architecture	string	Opcional	Arquitectura base del modelo (p. ej. "llama", "granite").
embedding_length	entero	Opcional	Dimensión del espacio de <i>embeddings</i> .
n_layers	entero	Opcional	Número de capas transformadoras.
max_context	entero	Opcional	Tamaño máximo de contexto soportado por el modelo.
quantization	string	Opcional	Nivel de cuantización aplicado al modelo (p. ej. "Q4_K_M").

A.5. Reglas de negocio (RN)

ID	Descripción
RN-01	El número de <i>prompts</i> (num_prompts) debe ser un entero comprendido en el intervalo cerrado [1, 541].
RN-02	El período de muestreo <i>hardware</i> (hardware_period) debe ser un valor real estrictamente positivo (> 0).
RN-03	La temperatura de muestreo (temperature) no puede ser negativa (≥ 0).
RN-04	El batch_size y el context_size no pueden ser negativos (≥ 0).
RN-05	El motor de inferencia OTHER no está soportado y debe producir un error de configuración.
RN-06	Para el motor HAILO_OLLAMA, el puerto del servidor (hailo_server_port) debe ser un entero estrictamente positivo (> 0).
RN-07	Para el motor HAILO_OLLAMA, los campos prompt_eval_count, prompt_eval_duration_ns, eval_duration_ns y load_duration_ns se fijan al valor 0 por diseño, al no estar disponibles en la API del servidor. El campo tokenProb se fija al valor "NONE".
RN-08	Una semilla (seed) de valor -1 indica semilla aleatoria; cualquier otro valor entero produce resultados reproducibles bajo las mismas condiciones de <i>hardware</i> y versión de modelo.
RN-09	Para el motor HAILO_OLLAMA, los parámetros batch_size y context_size se registran en el resumen pero no tienen efecto en tiempo de ejecución, ya que los modelos Hailo se compilan con dichos parámetros fijados en tiempo de compilación del modelo HEF.
RN-10	Cada ejecución genera un directorio de resultados con nombre igual al <i>timestamp</i> de inicio en nanosegundos, bajo la ruta results/.
RN-11	Los metadatos de arquitectura del modelo (model_info) se obtienen antes del primer <i>prompt</i> . Si no se pueden obtener, el campo se registra como null en el resumen sin interrumpir la ejecución.
RN-12	Para el motor OLLAMA, si el modelo especificado no está disponible localmente, el sistema debe descargarlo automáticamente antes de iniciar el experimento.

(continuación de reglas de negocio)

ID	Descripción
RN-13	La potencia reportada en las métricas <i>hardware</i> corresponde únicamente al sensor interno de la Raspberry Pi y no incluye el consumo de HATs ni periféricos USB conectados.

A.6. Requisitos funcionales (RF)

RF-01: Lectura de configuración desde fichero JSON

Descripción:	El sistema debe aceptar la configuración de un experimento mediante un fichero JSON cuya ruta se pasa como primer argumento de línea de comandos.
Actores:	Investigador.
Precondiciones:	El fichero existe y es legible.
Postcondiciones:	La configuración queda cargada en memoria y validada.
Relaciones:	OBJ-01, OBJ-04; RI-01.

RF-02: Lectura de configuración desde argumento *inline*

Descripción:	El sistema debe aceptar la configuración de un experimento mediante una cadena JSON pasada como argumento <code>-json '<json_string>'</code> en la línea de comandos.
Actores:	Investigador.
Postcondiciones:	La configuración queda cargada en memoria y validada.
Relaciones:	OBJ-01, OBJ-04; RI-01.

RF-03: Validación de la configuración de entrada

Descripción:	El sistema debe validar todos los campos de configuración antes de iniciar el experimento, produciendo mensajes de error descriptivos ante valores incorrectos o ausentes.
Actores:	Sistema.
Relaciones:	RN-01, RN-02, RN-03, RN-04, RN-05, RN-06.

RF-04: Ejecución de experimento con motor OLLAMA (TYPE_0)

Descripción:	El sistema debe ejecutar <code>num_prompts prompts</code> de forma secuencial contra el servidor Ollama y registrar las métricas de inferencia de cada uno, sin monitorización <i>hardware</i> .
Actores:	Investigador.
Precondiciones:	El servidor Ollama está activo y accesible en <code>ollama_url</code> .

Postcondiciones: Se genera el fichero *_prompt_metrics_*.jsonl con una entrada por *prompt*.
Relaciones: OBJ-01, OBJ-03; RI-02, RI-05; RN-12.

RF-05: Ejecución de experimento con motor OLLAMA (TYPE_1)

Descripción: El sistema debe ejecutar num_prompts *prompts* de forma secuencial contra el servidor Ollama con monitorización *hardware* paralela.
Actores: Investigador.
Precondiciones: El servidor Ollama está activo. La plataforma es Raspberry Pi 5.
Postcondiciones: Se generan los ficheros *_prompt_metrics_*.jsonl y *_hw_metrics_*.jsonl.
Relaciones: OBJ-01, OBJ-02, OBJ-03; RI-02, RI-03.

RF-06: Ejecución de experimento con motor OLLAMA (TYPE_2)

Descripción: Idéntico a RF-05, con una pausa de 5 segundos antes del primer *prompt* y entre cada par de *prompts* consecutivos.
Actores: Investigador.
Relaciones: OBJ-01, OBJ-02; RI-02, RI-03.

RF-07: Ejecución de experimento con motor LLAMA (TYPE_0, TYPE_1, TYPE_2)

Descripción: El sistema debe cargar el modelo GGUF especificado mediante llama.cpp y ejecutar los *prompts* con la misma lógica de tipos que para OLLAMA (RF-04, RF-05, RF-06). El tiempo de carga del modelo se incluye en las métricas del primer *prompt*.
Actores: Investigador.
Precondiciones: La ruta al fichero GGUF es válida y el fichero es accesible.
Postcondiciones: Idénticas a RF-04/RF-05/RF-06 según el tipo de test.
Relaciones: OBJ-01, OBJ-02, OBJ-03; RI-02, RI-03; RN-09.

RF-08: Ejecución de experimento con motor HAILO_OLLAMA (TYPE_0, TYPE_1, TYPE_2)

Descripción: El sistema debe ejecutar los *prompts* contra el servidor hailo-ollama con la misma lógica de tipos que para OLLAMA. Los campos no reportados por la API de hailo-ollama se fijan a 0.
Actores: Investigador.
Precondiciones: El servidor hailo-ollama está activo y accesible en hailo_server_host: hailo_server_port.
Postcondiciones: Idénticas a RF-04/RF-05/RF-06 según el tipo de test.
Relaciones: OBJ-01, OBJ-02, OBJ-03; RI-02, RI-03; RN-07, RN-09.

RF-09: Monitorización *hardware* periódica en hilo paralelo

- Descripción:** El sistema debe lanzar un hilo de ejecución que, con período *hardware_period*, muestrea las métricas *hardware* de la RPi5 y las escribe en el fichero **_hw_metrics_*.jsonl* de forma continua durante el experimento.
- Actores:** Sistema.
- Relaciones:** OBJ-02; RI-03; RN-02, RN-13.

RF-10: Registro de métricas de inferencia por *prompt*

- Descripción:** Tras cada inferencia, el sistema debe construir un objeto de métricas (RI-02) y escribirlo como línea JSON en el fichero de métricas de *prompts*, realizando *flush* inmediato para garantizar la persistencia ante interrupciones.
- Actores:** Sistema.
- Relaciones:** OBJ-01, OBJ-05; RI-02.

RF-11: Generación del resumen de ejecución

- Descripción:** Al finalizar el experimento, el sistema debe generar el fichero *resumen.json* con la configuración completa, los *timestamps* de inicio y fin, y los metadatos del modelo (RI-04).
- Actores:** Sistema.
- Postcondiciones:** El fichero *resumen.json* existe en el directorio de resultados y es JSON válido.
- Relaciones:** OBJ-05; RI-04, RI-06.

RF-12: Extracción de metadatos del modelo

- Descripción:** Al inicio de cada ejecución, el sistema debe intentar obtener los metadatos de arquitectura del modelo (RI-06) e incorporarlos al resumen como subcampo *model_info* dentro de *annotations*. Si la extracción falla, se registra *null* y la ejecución continúa.
- Actores:** Sistema.
- Relaciones:** OBJ-04; RI-04, RI-06; RN-11.

RF-13: Descarga automática de modelos Ollama

- Descripción:** Si el modelo especificado para el motor OLLAMA no está disponible localmente, el sistema debe descargarlo antes de iniciar el experimento, informando al usuario del proceso.
- Actores:** Sistema, Investigador.
- Relaciones:** OBJ-01; RN-12.

RF-14: Creación del directorio de resultados

Descripción:	Al inicio de cada ejecución, el sistema debe crear el directorio <code>results/<timestamp_ns>/</code> donde se almacenarán todos los ficheros de salida del experimento.
Actores:	Sistema.
Relaciones:	OBJ-05; RN-10.

RF-15: Visualización de uso por línea de comandos

Descripción:	Si el sistema se invoca sin argumentos o con argumentos incorrectos, debe mostrar un mensaje de uso con los campos requeridos y las opciones disponibles.
Actores:	Investigador.
Relaciones:	OBJ-01.

A.7. Requisitos no funcionales (RNF)

A.7.1. Usabilidad

ID	Descripción
RNF-01	El sistema debe proporcionar mensajes de error descriptivos e identificar el campo problemático ante cualquier configuración inválida, sin requerir inspección del código fuente.
RNF-02	La interfaz de uso debe limitarse a la línea de comandos, con dos modos de entrada equivalentes (fichero JSON o argumento <i>inline</i>), sin necesidad de instalación adicional de herramientas externas.

A.7.2. Rendimiento

ID	Descripción
RNF-03	El hilo de monitorización <i>hardware</i> debe respetar el período <code>hardware_period</code> con una desviación máxima inferior al 10 % del período configurado, para garantizar la precisión temporal de las muestras.
RNF-04	La sobrecarga introducida por el hilo de monitorización <i>hardware</i> sobre el tiempo de inferencia no debe ser estadísticamente significativa frente a la duración natural de una inferencia en la plataforma objetivo.

A.7.3. Fiabilidad

ID	Descripción
RNF-05	Cada línea de métricas debe escribirse y sincronizarse a disco (<i>flush</i>) de forma inmediata tras su generación, de modo que una interrupción abrupta no provoque pérdida de más de una muestra.
RNF-06	Un fallo en la obtención de metadatos del modelo o en la conexión con el servidor de inferencia no debe corromper los ficheros de salida ya generados en la misma ejecución.

A.7.4. Portabilidad

ID	Descripción
RNF-07	El binario compilado debe enlazarse estáticamente de forma que pueda copiarse y ejecutarse en cualquier Raspberry Pi 5 sin instalar dependencias adicionales (salvo el servidor de inferencia si corresponde).
RNF-08	El código fuente debe compilarse con C++20 y CMake $\geq 3.25.1$ mediante vcpkg como gestor de dependencias.
RNF-09	Las interfaces de métricas <i>hardware</i> son específicas de Raspberry Pi (vcgencmd, /sys/class/thermal, /proc/meminfo, etc.) y el sistema no garantiza su funcionamiento en otras plataformas.

A.7.5. Mantenibilidad y extensibilidad

ID	Descripción
RNF-10	La arquitectura debe permitir la incorporación de nuevos motores de inferencia mediante la adición de una nueva clase cliente y el correspondiente valor en el enumerado InferenceEngines, sin modificar la lógica principal del sistema.
RNF-11	La arquitectura debe permitir la incorporación de nuevos tipos de test (TestType) sin modificar las clases de cliente existentes.

A.7.6. Interoperabilidad

ID	Descripción
RNF-12	Todos los ficheros de salida deben estar en formato JSON estándar (RFC 8259), con codificación UTF-8, para garantizar su consumo por herramientas externas como el visualizador Python.
RNF-13	El formato de los ficheros de salida (RI-02, RI-03, RI-04) debe mantenerse estable entre versiones del sistema, o bien documentarse explícitamente cualquier cambio que rompa la compatibilidad.

A.7.7. Reproducibilidad científica

ID	Descripción
RNF-14	El uso de una semilla determinista ($\text{seed} \neq -1$) debe garantizar que dos ejecuciones con la misma configuración, el mismo modelo y el mismo motor de inferencia produzcan respuestas idénticas, salvo diferencias en las métricas temporales derivadas de variaciones del sistema operativo.
RNF-15	El fichero resumen.json debe incluir la configuración de entrada original completa (og_config.json) para garantizar la trazabilidad completa del experimento.

A.8. Pruebas de aceptación (PA)

PA-01: Configuración desde fichero JSON válido

Requisitos asociados: RF-01, RF-03

Precondición: Existe el fichero config_example.json con campos correctos.

Procedimiento:

1. Ejecutar: `./MonitorSystem config_example.json`

Criterio de aceptación: El sistema inicia la ejecución sin mensajes de error; se crea el directorio results/<timestamp>/.

PA-02: Configuración mediante argumento *inline*

Requisitos asociados: RF-02, RF-03

Procedimiento:

1. Ejecutar el siguiente comando (como cadena JSON *inline*):
`./MonitorSystem --json '{"inference_engine":"OLLAMA",
 "test_type":"TYPE_0","batch_size":512,"context_size":1024,
 "seed":42,"num_prompts":1,"temperature":0.7,
 "model_path_or_name":"<modelo>","hardware_period":0.5}'`

Criterio de aceptación: El sistema inicia la ejecución correctamente.

PA-03: Rechazo de configuración inválida —num_prompts fuera de rango

Requisitos asociados: RF-03; RN-01

Procedimiento:

1. Ejecutar con `num_prompts: 0` o `num_prompts: 542`.

Criterio de aceptación: El sistema termina con código de error $\neq 0$ y muestra un mensaje que identifica el campo num_prompts y los límites válidos [1, 541].

PA-04: Rechazo de configuración inválida —temperatura negativa

Requisitos asociados: RF-03; RN-03

Procedimiento:

1. Ejecutar con temperature: -0.1.

Criterio de aceptación: El sistema termina con código de error $\neq 0$ y muestra un mensaje que identifica el campo temperature.

PA-05: Rechazo de configuración inválida —hardware__period no positivo

Requisitos asociados: RF-03; RN-02

Procedimiento:

1. Ejecutar con hardware__period: 0.

Criterio de aceptación: El sistema termina con código de error $\neq 0$ y muestra un mensaje que identifica el campo hardware__period.

PA-06: Generación correcta de prompt__metrics (TYPE_0, OLLAMA)

Requisitos asociados: RF-04, RF-10; RI-02

Precondición: Servidor Ollama activo con el modelo especificado disponible.

Procedimiento:

1. Ejecutar con test__type: "TYPE_0", inference__engine: ".OLLAMA", num__prompts: 3.

Criterio de aceptación:

- Existe el fichero *_prompt__metrics_*.jsonl en el directorio de resultados.
- El fichero contiene exactamente 3 líneas JSON válidas.
- Cada línea contiene los campos obligatorios de RI-02.
- Se cumple start__timestamp__ns < finish__timestamp__ns en todas las entradas.

PA-07: Generación correcta de hw__metrics (TYPE_1, OLLAMA)

Requisitos asociados: RF-05, RF-09; RI-03

Precondición: Servidor Ollama activo; plataforma es Raspberry Pi 5.

Procedimiento:

1. Ejecutar con test__type: "TYPE_1", hardware__period: 1.0, num__prompts: 2.

Criterio de aceptación:

- Existen los ficheros *_prompt__metrics_*.jsonl y *_hw__metrics_*.jsonl.
- El fichero hw__metrics contiene al menos 1 línea por cada segundo de duración total del experimento (tolerancia: $\pm 10\%$).
- Cada línea del fichero hw__metrics contiene los campos obligatorios de RI-03.

PA-08: Generación correcta de resumen.json

Requisitos asociados: RF-11, RF-12; RI-04

Procedimiento:

1. Completar cualquier ejecución válida.

Criterio de aceptación:

- Existe el fichero resumen.json en el directorio de resultados.
- Es JSON válido con todos los campos obligatorios de RI-04.
- Se cumple `timestamp_run_start < timestamp_run_end`.
- El campo `og_config_json` es parseable como JSON y contiene los campos de la configuración original.

PA-09: Pausa entre *prompts* en TYPE_2

Requisitos asociados: RF-06; RI-02

Procedimiento:

1. Ejecutar con `test_type: "TYPE_2"`, `num_prompts: 2`.
2. Calcular la diferencia entre `start_timestamp_ns` del *prompt* 1 y `finish_timestamp_ns` del *prompt* 0.

Criterio de aceptación: La diferencia es ≥ 5 segundos (5×10^9 ns).

PA-10: Campos HAILO_OLLAMA fijados a 0

Requisitos asociados: RF-08; RN-07

Precondición: Servidor hailo-ollama accesible.

Procedimiento:

1. Ejecutar con `inference_engine: "HAILO_OLLAMA"`, `test_type: "TYPE_0"`, `num_prompts: 1`.
2. Inspeccionar el fichero `*_prompt_metrics_*.jsonl`.

Criterio de aceptación: Los campos `prompt_eval_count`, `prompt_eval_duration_ns`, `eval_duration_ns` y `load_duration_ns` son 0; `tokenProb` es "NONE".

PA-11: Reproducibilidad con semilla fija (OLLAMA)

Requisitos asociados: RNF-14; RI-02

Precondición: Servidor Ollama activo con `temperature = 0`.

Procedimiento:

1. Ejecutar dos veces con la misma configuración y `seed  $\neq$  -1`.
2. Comparar el campo `answer` de cada *prompt* en ambas ejecuciones.

Criterio de aceptación: Todos los campos `answer` son idénticos entre ambas ejecuciones.

PA-12: Mensaje de uso ante invocación incorrecta

Requisitos asociados: RF-15

Procedimiento:

1. Ejecutar ./MonitorSystem sin argumentos.

Criterio de aceptación: El sistema imprime un mensaje de uso que lista los campos requeridos y termina con código de error $\neq 0$.

A.9. Matrices de trazabilidad

A.9.1. Requisitos funcionales $\leftrightarrow$ Objetivos

Tabla A.17: Matriz de trazabilidad: Requisitos funcionales vs. Objetivos del sistema

RF \ OBJ	OBJ-01	OBJ-02	OBJ-03	OBJ-04	OBJ-05	OBJ-06
RF-01	✓			✓		
RF-02	✓			✓		
RF-03	✓			✓		
RF-04	✓		✓			✓
RF-05	✓	✓	✓			✓
RF-06	✓	✓	✓			✓
RF-07	✓	✓	✓	✓		✓
RF-08	✓	✓	✓	✓		✓
RF-09		✓				
RF-10	✓				✓	
RF-11				✓	✓	
RF-12				✓	✓	
RF-13	✓					
RF-14					✓	
RF-15	✓					

A.9.2. Requisitos funcionales ↔ Requisitos de información

Tabla A.18: Matriz de trazabilidad: Requisitos funcionales vs. Requisitos de información

RF \ RI	RI-01	RI-02	RI-03	RI-04	RI-05	RI-06
RF-01	✓					
RF-02	✓					
RF-03	✓					
RF-04		✓			✓	
RF-05		✓	✓		✓	
RF-06		✓	✓		✓	
RF-07		✓	✓		✓	
RF-08		✓	✓		✓	
RF-09			✓			
RF-10		✓				
RF-11				✓		✓
RF-12				✓		✓
RF-13						
RF-14				✓		
RF-15	✓					

A.9.3. Pruebas de aceptación ↔ Requisitos funcionales y reglas de negocio

PA	RF asociados	RN/RNF asociadas
PA-01	RF-01, RF-03	—
PA-02	RF-02, RF-03	—
PA-03	RF-03	RN-01
PA-04	RF-03	RN-03
PA-05	RF-03	RN-02
PA-06	RF-04, RF-10	RN-12
PA-07	RF-05, RF-09	RN-02, RN-13
PA-08	RF-11, RF-12	RN-10, RN-11
PA-09	RF-06	—
PA-10	RF-08	RN-07
PA-11	—	RNF-14
PA-12	RF-15	—

B. Especificación de Requisitos del *Software* (*monitorviz*)

Sistema	<i>monitorviz</i>
Versión	1.0
Fecha	2026-06-13
Metodología	Ingeniería de Requisitos (IISSI – Universidad de Sevilla)
Estado	Completado

B.1. Introducción

B.1.1. Propósito

El presente documento constituye la Especificación de Requisitos del *Software* (ERS) del sistema *monitorviz*, una librería Python para el análisis y la visualización de datos de rendimiento de la inferencia de modelos de lenguaje grandes (LLMs) sobre plataformas de computación embebida. El documento sigue la metodología de Ingeniería de Requisitos IISSI de la Universidad de Sevilla y tiene como destinatarios al investigador principal (autor del TFG), al director del trabajo y a posibles investigadores externos que reutilicen la librería.

B.1.2. Alcance

monitorviz consume datos producidos por el binario C++ MonitorSystemCplusplus, que monitoriza la inferencia de LLMs sobre una Raspberry Pi 5, y ofrece:

1. Carga y validación de conjuntos de ejecuciones de inferencia.
2. Cálculo automático de métricas derivadas de rendimiento, eficiencia energética y calidad de generación.
3. Generación de visualizaciones estáticas (series temporales, distribuciones, Pareto, etc.).
4. Integración nativa con entornos Jupyter para análisis exploratorio y reproducible.

Quedan fuera del alcance:

- Interfaz de línea de comandos (CLI).
- Paneles interactivos (dashboards web tipo Dash/Streamlit).
- Soporte para plataformas de *hardware* distintas a la Raspberry Pi 5.
- Ejecución del proceso de inferencia (responsabilidad de MonitorSystemCplusplus).

B.1.3. Definiciones, acrónimos y abreviaturas

Véase el Glosario (Sección [B.2](#)).

B.1.4. Referencias

- ISO/IEC 9126-1:2001 — Ingeniería del *Software*: Calidad del producto.
- Metodología IISSI, Universidad de Sevilla.
- ERS de MonitorSystem (Anexo A).

B.2. Glosario de términos del dominio

Término	Definición
Run	Ejecución completa del proceso de inferencia: carga del modelo, evaluación de un conjunto de <i>prompts</i> y, opcionalmente, muestreo de <i>hardware</i> . Se almacena en <code>results/<timestamp_ns>/</code> .
Prompt	Texto de entrada al modelo LLM. Genera una respuesta (<i>answer</i>) junto con tiempos de fase y probabilidades de <i>token</i> .
Engine	Motor de inferencia. Valores: OLLAMA, LLAMA (llama.cpp), HAILO_OLLAMA (Motor para HAILO con interfaz similar a Ollama).
Generación vacía	Respuesta con <code>eval_count ≤ 1</code> y <code>answer=</code> . Comportamiento legítimo de modelos base no ajustados por instrucción; las métricas derivadas son indefinidas (None/NaN).
Perplexidad (PPL)	Medida de calidad de generación: $\exp\left(-\frac{1}{N} \sum_{i=1}^N \log p_i\right)$, donde p_i son las probabilidades de cada <i>token</i> generado.
Throughput TTFT	Tasa de generación expresada en tokens/s o palabras/s. <i>Time To First Token</i> : latencia desde el inicio de la petición hasta el primer <i>token</i> de respuesta.
TPOT	<i>Time Per Output Token</i> : tiempo medio por <i>token</i> generado durante la fase de decodificación.
MBU	<i>Memory Bandwidth Utilization</i> : fracción del ancho de banda de memoria pico consumida durante la inferencia, calculada mediante la fórmula ELIB.
FoM	<i>Figure of Merit</i> : métrica compuesta que agrega <i>throughput</i> , MBU, eficiencia CPU y eficiencia energética en un único valor normalizado respecto a una ejecución de referencia.
FoM_full	Media geométrica de cuatro ejes normalizados: T , MBU, η_{CPU} y ε .
FoM_red	Media geométrica de <i>throughput</i> y eficiencia energética normalizados respecto a la CPU base del mismo modelo. Usada en la serie E5 para comparar aceleradores.
Fase de inferencia	Subdivisión temporal de un <i>prompt</i> : <i>load</i> (carga del modelo), <i>prefill</i> (evaluación del contexto de entrada) y <i>decode</i> (generación <i>token a token</i>).
Hardware period	Intervalo de muestreo de métricas <i>hardware</i> , en segundos. Codificado dentro del campo anidado <code>og_config_json</code> de <code>resumen.json</code> .
Throttling	Limitación automática de la frecuencia o del voltaje del SoC por protección térmica o de alimentación. Se reporta mediante flags booleanas de estado actual e histórico.

Término	Definición
TokenProb	Probabilidad (o log-probabilidad) de cada <i>token</i> generado. Serializada como cadena JSON; el formato interno difiere según el engine (véase RN-03 y RN-04).
RunCollection	Conjunto de múltiples <i>runs</i> en memoria sobre el que se aplican filtros y se generan tablas resumen agregadas.
summary_df	DataFrame con una fila por run que consolida métricas de rendimiento, <i>hardware</i> , energía y propiedades del modelo. Tabla principal de análisis comparativo.
t_rel_s	Tiempo relativo al inicio del run, en segundos. Eje X estándar de las series temporales.
RunMeta	Metadatos registrados manualmente: estado del ventilador, presencia de acelerador externo, temperatura ambiente y notas libres.
GGUF	Formato de cuantización de modelos para llama.cpp. El nombre del fichero codifica arquitectura y esquema de cuantización (p. ej. Q4_K_M).
Raspberry Pi 5	Plataforma objetivo: SoC BCM2712 (Cortex-A76, 4 núcleos, 2,4 GHz nominal), 8 GB LPDDR4X-4267 32 bit (ancho de banda pico: 17,1 GB/s).
Hailo-10h	Acelerador neuronal externo (NPU) conectado a la RPi5 vía PCIe; evaluado en la serie E5.

B.3. Requisitos generales (objetivos)

Los objetivos se expresan como historias de usuario épicas en formato Mike Cohn.

- OBJ-01** *Como investigador*, quiero cargar y unificar datos de múltiples ejecuciones de inferencia LLM en un único entorno Python, *para* disponer de un punto de acceso uniforme a los resultados experimentales independientemente del motor utilizado.
- OBJ-02** *Como investigador*, quiero calcular automáticamente métricas derivadas de rendimiento y eficiencia a partir de los datos brutos, *para* caracterizar el comportamiento de los modelos LLM en la Raspberry Pi 5 sin reimplementar los cálculos en cada notebook.
- OBJ-03** *Como investigador*, quiero visualizar las series temporales de métricas de *hardware* durante la inferencia, *para* detectar eventos de throttling, picos térmicos y patrones de consumo energético en cada ejecución.
- OBJ-04** *Como investigador*, quiero comparar el rendimiento y la eficiencia de diferentes combinaciones de modelo, motor y configuración experimental, *para* identificar la configuración óptima para inferencia LLM en *hardware* embebido.
- OBJ-05** *Como director del TFG*, quiero reproducir los notebooks de análisis a partir de los datos originales versionados, *para* verificar la validez de los resultados y las conclusiones presentadas.
- OBJ-06** *Como investigador externo*, quiero importar *monitorviz* como librería Python

estándar, *para* reutilizar los modelos de datos, transformaciones y visualizaciones en estudios de inferencia LLM sobre *hardware* embebido similar.

B.4. Requisitos de información

RI-01 Ejecución de inferencia (*Run*)

Fuente: resumen.json en el directorio de run.

Dato	Tipo	Restricción	Carácter
Identificador (run_id)	Cadena	Nombre del directorio	Obligatorio
Motor (inference_engine)	Enumerado	OLLAMA, LLAMA, HAILO_OLLAMA	Obligatorio
Modelo (model_path_or_name)	Cadena		Obligatorio
Tipo de test (test_type)	Enumerado	TYPE_0, TYPE_1, TYPE_2	Obligatorio
Tamaño de lote (batch_size)	Entero ≥ 1		Obligatorio
Tamaño de contexto (context_size)	Entero ≥ 1	Tokens	Obligatorio
Número de prompts (num_prompts)	Entero ≥ 0		Obligatorio
Semilla (seed)	Entero	Reproducibilidad	Obligatorio
Temperatura (temperature)	Real ≥ 0		Obligatorio
Período de hardware (hardware_period_s)	Real > 0	Segundos; anidado en og_config_json	Obligatorio
Inicio del run (timestamp_run_start_ns)	Entero	Nanosegundos	Obligatorio
Fin del run (timestamp_run_end_ns)	Entero	Nanosegundos	Obligatorio
Energía total (total_kwh)	Real ≥ 0	kWh; medición externa Hailo	Opcional
Potencia mínima (watt_min)	Real ≥ 0	W; medición externa Hailo	Opcional
Potencia máxima (watt_max)	Real ≥ 0	W; medición externa Hailo	Opcional

RI-02 Métricas por *prompt* (*PromptMetric*)

Fuente: *_prompt_metrics_*.jsonl.

Dato	Tipo	Restricción	Carácter
Inicio del prompt	Entero	Nanosegundos	Obligatorio

Dato	Tipo	Restricción	Carácter
Fin del <i>prompt</i>	Entero	Nanosegundos	Obligatorio
Duración total (total_duration_ns)	Entero ≥ 0	Nanosegundos	Obligatorio
Duración de carga (load_duration_ns)	Entero ≥ 0	Nanosegundos	Obligatorio
Duración de <i>prefill</i> (prompt_eval_duration_ns)	Entero ≥ 0	Nanosegundos	Obligatorio
Duración de <i>decode</i> (eval_duration_ns)	Entero ≥ 0	0 para HAILO_OLLAMA; véase RN-12	Obligatorio
Tokens de entrada (prompt_eval_count)	Entero ≥ 0		Obligatorio
Tokens de salida (eval_count)	Entero ≥ 0		Obligatorio
Respuesta (answer)	Cadena	Puede ser vacía	Obligatorio
Motor (engine)	Enumerado		Obligatorio
Modelo (model)	Cadena		Obligatorio
Tipo de probabilidad (prob_type)	Entero	0 = log-prob; 1 = prob cruda	Obligatorio
Probabilidades de token (tokens)	Lista[TokenProb]	Serializada como cadena JSON	Obligatorio

Campos derivados calculados por el sistema: perplexity, tokens_per_second, words_per_second, time_to_first_token_ns, is_empty_generation.

RI-03 Muestra de *hardware* (*HwSample*)

Fuente: *_hw_metrics_.jsonl (presente únicamente en TYPE_1 y TYPE_2).

Dato	Tipo	Restricción	Carácter
Marca temporal (timestamp_ms)	Entero	Milisegundos (13 dígitos); véase RN-05	Obligatorio
Temperatura (temperature_c)	Real	Grados Celsius	Obligatorio
Frecuencia por núcleo (frequency_ghz)	Lista[Real]	GHz; véase RN-08	Obligatorio
RPM del ventilador (fan_rpm)	Real ≥ 0		Obligatorio
Voltaje (voltage_v)	Real ≥ 0	Voltios	Obligatorio
Potencia interna (internal_power_w)	Real ≥ 0	W; excluye el chip Hailo	Obligatorio
Uso de CPU (cpu_usage_pct)	Real [0,100]	Porcentaje	Obligatorio
Ticks de CPU (cpu_ticks)	Diccionario	Por estado (user, system, idle, ...)	Obligatorio
Memoria usada (mem_used_bytes)	Entero ≥ 0	Bytes	Obligatorio

Dato		Tipo	Restricción	Carácter
Memoria	total	Entero > 0	Bytes	Obligatorio
(mem_total_bytes)				
Porcentaje de memoria		Real $[0,100]$		Obligatorio
(mem_pct)				
Swap	usada	Entero ≥ 0	Bytes	Obligatorio
(swap_used_bytes)				
Swap	total	Entero ≥ 0	Bytes	Obligatorio
(swap_total_bytes)				
Porcentaje de swap		Real $[0,100]$		Obligatorio
(swap_pct)				
Flags de throttling		ThrottlingFlags	Véase RI-04	Obligatorio

RI-04 Flags de throttling (*ThrottlingFlags*)

Nota: el JSON de origen usa el tipo “ocurred” (una ‘c’); el sistema acepta ambas gráficas mediante alias (RN-07).

Dato	Tipo	Restricción	Carácter
Under-voltage activo	Booleano	Estado actual	Obligatorio
Frecuencia limitada activa	Booleano	Estado actual	Obligatorio
Throttled activo	Booleano	Estado actual	Obligatorio
Soft-throttled activo	Booleano	Estado actual	Obligatorio
Under-voltage ocurrido	Booleano	Histórico	Obligatorio
Frecuencia limitada ocurrida	Booleano	Histórico	Obligatorio
Throttled ocurrido	Booleano	Histórico	Obligatorio
Soft-throttled ocurrido	Booleano	Histórico	Obligatorio

RI-05 Probabilidades de *token* (*TokenProb*)

Dato	Tipo	Restricción	Carácter
Texto del <i>token</i> (token)	Cadena	Vacío para LLAMA	Obligatorio
Log-probabilidad	Real ≤ 0	$-\infty$ si $p \leq 0$	Obligatorio
(logprob)			
Bytes del <i>token</i> (bytes)	Lista[Entero]	Solo OLLAMA	Opcional

RI-06 Metadatos experimentales (*RunMeta*)

Fuente: meta.yaml en el directorio de run (ausente en runs sin anotación manual).

Dato	Tipo	Restricción	Carácter
Ventilador activo (fan)	Booleano		Opcional

Dato		Tipo	Restricción	Carácter
Acelerador (accelerator)	externo	Booleano		Opcional
Temperatura (ambient_temperature_c)	ambiente	Real	Grados Celsius	Opcional
Notas (notes)		Cadena		Opcional
Campos adicionales (other)		Arbitrario	Extensible	Opcional

RI-07 Colección de ejecuciones (*RunCollection*)

Una colección agrupa entre 1 y 200 runs. Puede filtrarse por cualquier combinación de factores: `run_id`, `engine`, `model_short`, `test_type`, `batch_size`, `context_size`, `seed`, `temperature`, `hardware_period_s`, `fan` y `accelerator`.

RI-08 Tabla resumen (*summary_df*)

Tabla derivada con una fila por run que consolida:

- Factores experimentales (todos los de RI-07).
- Agregados de *prompts*: `n_prompts`, `n_empty_generations`, `tokens_per_s_mean/p50/p95`, `words_per_s_mean`, `perplexity_geomean/p50`, `latency_ms_mean/p95`, `ttft_ms_mean`, `tpot_mean_s`.
- Agregados de *hardware* (NaN si TYPE_0): `temp_max/mean_c`, `power_mean/max_w`, `throttled_ratio`, `throttled_any_ever`.
- Eficiencia energética: `energy_per_token_j`, `energy_total_j`, `cpu_work_core_s`, `cpu_efficiency`, `cpu_work_per_token`, `cpu_work_per_joule`.
- Propiedades del modelo: `model_size_gb`, `n_params`, `mbu_naive_pct`, `mbu_corr_pct`, `mbu_is_upper_bound`, `mbu_corr_notes`.

B.5. Reglas de negocio

- RN-01 Generaciones vacías legítimas.** Un *prompt* con `eval_count` ≤ 1 y `answer` = no constituye un error. Las métricas derivadas *perplexidad*, *tokens/s* y *palabras/s* deben ser None/NaN para dichos *prompts*.
- RN-02 Tolerancia off-by-one en TokenProb.** La diferencia $|\text{len}(\text{tokens}) - \text{eval_count}|$ puede ser 1 sin emitir error. Solo se registra una advertencia si la diferencia es ≥ 2 .
- RN-03 TokenProb serializado como cadena.** El campo `tokenProb` en el JSONL de *prompts* es una cadena JSON que debe deserializarse con `json.loads()` antes de su procesamiento.
- RN-04 Dos formatos de TokenProb según engine.**
- *probType* = 0 (OLLAMA): lista de objetos {token, logprob, bytes}.
 - *probType* = 1 (LLAMA): lista de probabilidades crudas (no logarítmicas). El sistema las convierte mediante $\log(p)$; si $p \leq 0$, asigna $-\infty$; si $p > 1$, registra advertencia y usa 0,0.

- RN-05 Timestamps de hw_metrics en milisegundos.** El campo `timestamp_` de las muestras de *hardware* está en milisegundos (13 dígitos), no en nanosegundos. La conversión a `t_rel_s` debe aplicarse correctamente para alinear con las ventanas de los *prompts*.
- RN-06 hardware_period anidado.** El período de muestreo de *hardware* no es un campo directo de `resumen.json`; está codificado dentro de la cadena JSON del campo `og_config.json` y debe extraerse deserializando dicha cadena.
- RN-07 Typo en flags de throttling.** Los campos históricos de throttling en el JSON de origen usan la grafía “ocurred” (una ‘c’). El sistema debe aceptar tanto ‘ocurred’ como “occurred” mediante alias, preservando la grafía original.
- RN-08 Frecuencia de CPU en GHz.** El campo `frequency_` del binario C++ está en GHz. El sistema no deberá convertirlo a Hz salvo indicación explícita.
- RN-09 Referencia del FoM_full.** El FoM completo se normaliza respecto a la ejecución: Llama-3.2-3B, cuantización Q4_K_M, engine LLAMA, `context_size=4096`, `batch_size=512`. Si la referencia no está presente en la colección, el cómputo no produce resultados.
- RN-10 MBU diferenciada por arquitectura.**
- Modelos densos: fórmula ELIB estándar.
 - Híbridos SSM (Granite): añade M_{state} (74,4 MiB fijo), reduce M_{KV} a las capas de atención.
 - Activación dispersa (Gemma 3n MatFormer, MoE): usa $N_{\text{params,act}}$ para M_{model} ; el resultado es una cota inferior.
- RN-11 Nombre corto del modelo (model_short).** Para LLAMA: basename del fichero GGUF sin extensión. Para OLLAMA y HAILO_OLLAMA: nombre registrado en el servidor Ollama.
- RN-12 HAILO_OLLAMA: fallback de duración de decode.** El engine HAILO_OLLAMA reporta `eval_duration_ns=0`. Para el cálculo de *throughput*, el sistema utilizará `total_duration_ns` como duración efectiva.
- RN-13 Umbral de usabilidad conversacional.** El *throughput* mínimo para uso conversacional se define como $T_{\text{hum}} = r_{t/w} \times W_{\text{min}}$, donde $r_{t/w}$ es la tasa tokens/palabra del modelo derivada de los *prompts* de la colección y $W_{\text{min}} = 3$ palabras/s.

B.6. Requisitos funcionales

B.6.1. Carga de datos

- RF-01 Carga de un run individual.** El sistema proporcionará `load_run(run_dir)`, que, dado el directorio de un run, descubrirá automáticamente los ficheros `resumen.json`, `*_prompt_metrics*.jsonl`, `*_hw_metrics*.jsonl` y `meta.yaml`, los validará y retornará un objeto Run completamente poblado.
- RF-02 Carga de una colección de runs.** El sistema proporcionará `load_collection(root)`, que localizará recursivamente todos los directorios de run bajo una raíz dada. Con `skip_errors=True` (valor por defecto), los runs corruptos se omitirán registrando un aviso; con `skip_errors=False` se propagará la primera excepción.

- RF-03** **Detección de ficheros múltiples.** Cuando existan varios ficheros que coincidan con el mismo patrón en un directorio de run, el sistema registrará una advertencia y usará el primero.
- RF-04** **Normalización de TokenProb.** El sistema normalizará el campo tokenProb (cadena JSON) a una lista de objetos TokenProb aplicando las conversiones de RN-03 y RN-04, con manejo defensivo de valores malformados.

B.6.2. Transformaciones y métricas derivadas

- RF-05** **Cálculo de perplexidad.** El sistema calculará la perplexidad por *prompt* como $\exp(-\frac{1}{N} \sum \log p_i)$. Para generaciones vacías (RN-01) el valor será None.
- RF-06** **Cálculo de *throughput*.** El sistema calculará tokens/s, palabras/s, TTFT y TPOT para cada *prompt* no vacío, aplicando el fallback de RN-12 para HAILO_OLLAMA.
- RF-07** **DataFrame de *prompts*.** El sistema proporcionará `prompt_metrics_to_df(run)`: una fila por *prompt* con todos los factores experimentales y las métricas derivadas.
- RF-08** **DataFrame de *hardware*.** El sistema proporcionará `hw_metrics_to_df(run)`: una fila por muestra de *hardware* con las métricas expandidas y `t_rel_s`. Para runs sin datos de *hardware* retornará un DataFrame vacío con el esquema correcto.
- RF-09** **DataFrame de *tokens* en formato largo.** El sistema proporcionará `tokens_to_df(run)`: una fila por *token* con columnas `prompt_id`, `token_idx`, `token` y `logprob`.
- RF-10** **DataFrame de frecuencia por núcleo.** El sistema proporcionará `hw_freq_long(run)`: formato largo con columna `core_id` para análisis per-núcleo con seaborn.
- RF-11** **Tabla resumen de colección.** El sistema proporcionará `RunCollection.summary_df()` que consolidará todas las métricas de RI-08 con una fila por run.
- RF-12** **Cálculo de MBU.** El sistema calculará la Utilización de Ancho de Banda de Memoria en versión naïve y corregida por arquitectura según RN-10, requiriendo `model_info` en `RunSummary`.
- RF-13** **Trabajo y eficiencia CPU.** El sistema calculará W_{CPU} , η_{CPU} y $W_{CPU}/token$, así como su desglose por fase (*load*, *prefill*, *decode*) mediante `cpu_work_by_phase`.
- RF-14** **Energía por *token*.** El sistema calculará `energy_per_token_j` y `energy_total_j`, usando la medición externa de potencia para HAILO_OLLAMA cuando esté disponible.
- RF-15** **FoM completo.** El sistema proporcionará `compute_fom_full(df)` que calculará los cuatro ejes normalizados y su media geométrica según RN-09.
- RF-16** **FoM reducido.** El sistema proporcionará `compute_fom_red(df)` para comparación intra-modelo respecto a la CPU base (véase RN-09).
- RF-17** **Usabilidad conversacional.** El sistema proporcionará `compute_usability(summary_df, prompt_df)` que calculará T_{hum} por modelo y marcará los runs que superen dicho umbral (RN-13).

B.6.3. Filtrado e inspección de colecciones

- RF-18 Filtrado de colección.** El sistema proporcionará `RunCollection.filter(**kwargs)` que retornará una nueva colección con los runs que coincidan con todos los valores indicados, admitiendo filtrado simultáneo por múltiples factores.
- RF-19 Matriz de experimentos.** El sistema proporcionará `RunCollection.experiment_matrix()` que retornará una tabla de cobertura con el número de runs por combinación (modelo $\times$ engine $\times$ fan $\times$ accelerator).

B.6.4. Visualización

- RF-20 Series temporales de *hardware*.** El sistema generará figuras matplotlib con la evolución temporal de cualquier métrica de *hardware* sobre el eje `t_rel_s`, con anotación opcional de spans de *prompts* y marcadores de *throttling*.
- RF-21 Figuras de doble eje Y.** El sistema generará figuras con dos ejes Y para los pares temperatura/potencia y temperatura/frecuencia sobre el mismo eje temporal.
- RF-22 Panel de distribuciones de *hardware*.** El sistema generará un panel con histogramas o KDE de todas las métricas de *hardware* de una colección.
- RF-23 Barras comparativas.** El sistema generará barplots con intervalos de error para comparar métricas de rendimiento entre runs de una colección.
- RF-24 Frontera de Pareto.** El sistema generará diagramas de dispersión con la frontera de Pareto para dos métricas cualesquiera de la tabla resumen, con codificación por color según el factor seleccionado.
- RF-25 Log-probabilidad por posición de *token*.** El sistema generará diagramas de log-probabilidad en función de la posición del *token* en la respuesta.
- RF-26 Desglose de fases por run.** El sistema generará barras apiladas con las duraciones de las fases *load*, *prefill* y *decode* por run de la colección.
- RF-27 Heatmap de throttling.** El sistema generará un mapa de calor de la tasa de throttling por combinación modelo $\times$ engine.
- RF-28 Heatmap de correlaciones.** El sistema generará una matriz de correlación entre las métricas numéricas de la tabla resumen.
- RF-29 Gráfica de sensibilidad.** El sistema generará diagramas de líneas que muestren la variación de una métrica en función de un factor experimental (p.ej. `tokens/s` vs. `context_size`).
- RF-30 Diagramas de metodología.** El sistema generará diagramas de arquitectura del sistema, de los ejes del FoM y de la matriz de experimentos para su inclusión en la memoria del TFG.

B.7. Requisitos no funcionales

Los requisitos se clasifican según la norma ISO/IEC 9126-1.

B.7.1. Usabilidad (ISO 9126: Usability)

- RNF-01** **API consistente de primitivas de visualización.** Todas las funciones del módulo viz/primitives.py seguirán la convención func(ax, data, ...) ->ax, permitiendo su composición directa sobre cualquier objeto Axes de matplotlib.
- RNF-02** **Comportamiento seguro por defecto en la carga.** El parámetro skip_errors de load_collection tendrá valor True por defecto, evitando que un único run corrupto interrumpa el análisis de una colección completa.
- RNF-03** **Estabilidad del esquema de DataFrames.** Los nombres de las columnas de los DataFrames producidos por el módulo transforms se mantendrán estables entre versiones menores de la librería, de modo que los notebooks existentes no requieran modificación al actualizar el paquete.
- RNF-04** **Documentación de unidades.** Cada función de transformación pública incluirá una cadena de documentación que especifique las unidades de cada columna del DataFrame retornado.

B.7.2. Rendimiento (ISO 9126: Efficiency)

- RNF-05** **Tiempo de carga de colección.** load_collection procesará hasta 200 runs en un tiempo máximo de 60 segundos en la máquina de análisis del investigador.
- RNF-06** **Tiempo de generación de summary_df.** El método summary_df() sobre una colección de 200 runs se completará en menos de 10 segundos.
- RNF-07** **Tiempo de generación de figuras compuestas.** Cada función del módulo viz/composite.py generará su figura en menos de 5 segundos para colecciones de hasta 200 runs.

B.7.3. Fiabilidad (ISO 9126: Reliability)

- RNF-08** **Validación en los límites de entrada.** Todos los modelos de datos serán validados mediante Pydantic v2 en el momento de la carga. Las anomalías toleradas (RN-01, RN-02, RN-07) se registrarán como advertencias en el logger estándar de Python, sin elevar excepciones.
- RNF-09** **Degradación controlada ante datos ausentes.** La ausencia de ficheros opcionales (*_hw_metrics_*.jsonl, meta.yaml) no impedirá la carga del run; las métricas dependientes de dichos ficheros serán NaN/None en los DataFrames.
- RNF-10** **Cobertura de pruebas.** La cobertura automática (pytest-cov) sobre el módulo src/monitorviz será superior al 80 %.

B.7.4. Compatibilidad (ISO 9126: Portability/Compatibility)

- RNF-11** **Versión mínima de Python.** La librería será compatible con Python 3.11 o superior.
- RNF-12** **Instalación reproducible.** La instalación se realizará mediante uv pip install -e . sin requerir configuración adicional del entorno.

- RNF-13** **Sistema operativo.** La librería funcionará correctamente en Linux (verificado en Fedora 43).
- RNF-14** **Entorno Jupyter.** La librería será utilizable desde JupyterLab 4 con ipykernel 6 o superiores.

B.7.5. Mantenibilidad (ISO 9126: Maintainability)

- RNF-15** **Conformidad con el linter.** La ejecución de `ruff check src/` con la configuración del repositorio no producirá errores ni advertencias.
- RNF-16** **Separación de capas.** La arquitectura del paquete mantendrá la separación en cuatro capas: `models/` (Pydantic), `io/` (parsers), `transforms/` (cálculos) y `viz/` (visualización), sin dependencias ascendentes entre capas.

B.7.6. Seguridad (ISO 9126: Security)

- RNF-17** **Acceso de solo lectura.** La librería no escribirá, modificará ni eliminará ficheros del directorio de datos. Su operación se limitará a la lectura de los ficheros de runs.
- RNF-18** **Ausencia de transmisión de datos.** La librería no realizará llamadas de red ni transmitirá datos a servicios externos durante su operación normal.

B.8. Pruebas de aceptación

ID	Descripción	Resultado esperado	Requisitos asociados
PA-01	Cargar run OLLAMA TYPE_1 con <i>prompts</i> no vacíos.	<code>perplexity</code> , <code>tokens_per_second</code> y <code>words_per_second</code> son valores numéricos positivos para todos los <i>prompts</i> .	RF-01, RF-05, RF-06
PA-02	Cargar run LLAMA con primera generación vacía (<code>eval_count ≤ 1</code> , <code>answer =</code>).	<code>is_empty_generation=True</code> y métricas derivadas son <code>None</code> para ese <i>prompt</i> ; los demás <i>prompts</i> tienen valores normales.	RF-01, RF-05, RF-06, RN-01
PA-03	Cargar run con <code>len(tokens) = eval_count - 1</code> (off-by-one).	La carga se completa sin excepción; se registra una advertencia en el log.	RF-04, RN-02
PA-04	<code>load_collection</code> con un run con <code>resumen.json</code> malformado y varios runs válidos, <code>skip_errors=True</code> .	Retorna <code>RunCollection</code> con solo los runs válidos; se registra una advertencia por el run omitido.	RF-02, RNF-02, RNF-08

ID	Descripción	Resultado esperado	Requisitos asociados
PA-05	filter(engine="LLAMA", batch_size=1) sobre colección mixta.	La nueva colección contiene únicamente runs con engine=LLAMA y batch_size=1.	RF-18
PA-06	summary_df() sobre colección con model_info completo.	mbu_naive_pct y mbu_corr_pct tienen valores no nulos para todos los runs.	RF-11, RF-12
PA-07	hw_timeline(run) sobre un run TYPE_1.	Figura matplotlib generada sin excepción; eje X muestra t_rel_s, eje Y la métrica seleccionada.	RF-20, RNF-01
PA-08	pareto_panel(coll, x="tokens_per_s_mean", y=.energy_per_token_j") con ≥ 5 runs.	Figura con la frontera de Pareto correctamente identificada y visualizada.	RF-24
PA-09	compute_fom_full(df) sobre colección que incluye el run de referencia.	fom_full $\approx 1,0$ para el run de referencia y > 0 para los demás.	RF-15, RN-09
PA-10	Medir tiempo de load_collection con 100 runs válidos.	Tiempo total inferior a 60 segundos.	RF-02, RNF-05
PA-11	cpu_work_by_phase sobre un run TYPE_1 con fases bien definidas.	DataFrame con tres filas por <i>prompt</i> (load, <i>prefill</i> , <i>decode</i>); cpu_work_core_s no negativo y coherente con el total.	RF-13, RN-05
PA-12	Cargar run HAILO_OLLAMA y verificar <i>throughput</i> .	tokens_per_second calculado usando total_duration_ns (no eval_duration_ns=0).	RF-06, RN-12
PA-13	compute_usability(summary_prompt_df) con min_words_per_s=3.	Columna usable refleja correctamente qué runs superan T_{hum} por modelo.	RF-17, RN-13
PA-14	ruff check src/ en el repositorio.	Ningún error ni advertencia de linting.	RNF-15

B.9. Matriz de trazabilidad

B.9.1. OBJ $\rightarrow$ RF, RN, PA

OBJ	RF asociados	RN que aplican	PA de verificación
OBJ-01	RF-01, RF-02, RF-03, RF-04	RN-03, RN-04, RN-06, RN-07	PA-01, PA-02, PA-03, PA-04
OBJ-02	RF-05, RF-06, RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13, RF-14, RF-15, RF-16, RF-17	RN-01, RN-02, RN-05, RN-08, RN-09, RN-10, RN-11, RN-12, RN-13	PA-01, PA-02, PA-06, PA-09, PA-11, PA-12, PA-13
OBJ-03	RF-20, RF-21, RF-22, RF-26, RF-27, RF-28, RF-29	RN-05, RN-08	PA-07, PA-11
OBJ-04	RF-11, RF-15, RF-16, RF-18, RF-19, RF-23, RF-24, RF-25, RF-29, RF-30	RN-09, RN-10, RN-13	PA-05, PA-06, PA-08, PA-09, PA-13
OBJ-05	RF-01, RF-02, RF-11	—	PA-04, PA-10
OBJ-06	RF-01, RF-02, RF-07, RF-08, RF-09, RF-10, RF-11	—	PA-01, PA-10

B.9.2. PA → RF, RN

PA	Descripción breve	RF verificados	RN verificadas
PA-01	Run OLLAMA TYPE_1 con métricas derivadas	RF-01, RF-05, RF-06	—
PA-02	Generación vacía LLAMA	RF-01, RF-05, RF-06	RN-01
PA-03	Off-by-one en TokenProb	RF-04	RN-02
PA-04	load_collection con run corrupto	RF-02	—
PA-05	Filtrado por engine y batch_size	RF-18	—
PA-06	summary_df con MBU	RF-11, RF-12	RN-10
PA-07	hw_timeline TYPE_1	RF-20	RN-05
PA-08	Frontera de Pareto	RF-24	—
PA-09	FoM_full con run de referencia	RF-15	RN-09
PA-10	Tiempo de carga 100 runs	RF-02	—
PA-11	cpu_work_by_phase	RF-13	RN-05
PA-12	Throughput HAILO_OLLAMA	RF-06	RN-12
PA-13	Usabilidad (compute_usability)	RF-17	RN-13
PA-14	Conformidad ruff	—	—

C. Tablas de métricas completas

Este anexo recoge las tablas de métricas detalladas generadas a partir de cada serie experimental. Los resultados resumidos se presentan en el Capítulo 4; aquí se incluyen todas las columnas para facilitar la reproducibilidad y el análisis independiente.

C.1. Serie E0 —Caracterización base (FAN y NOFAN)

Tabla C.1: Serie E0 —Caracterización base, rendimiento e inferencia. Configuración: Q4_K_M, contexto 4096, lote 512. Ref. = condición de refrigeración (✓ FAN, × NOFAN). TTFT: tiempo al primer token (ms); σ : desviación típica intra-run entre prompts. TTLM: tiempo medio de carga del modelo por prompt (s). $\bar{P}$: potencia media del run (W); E : energía total (kJ). RAM: uso medio de memoria RAM en runtime (mean(mem_used_bytes)/1e9, GB métricos); σ_{RAM} : desviación típica.

Modelo	Ref.	RAM (GB)	σ_{RAM} (GB)	tok/s	σ	TTFT (ms)	σ (ms)	TTLM (s)	J/tok	tok/J	$\bar{P}$ (W)	E (kJ)
<i>DeepSeek-R1-1.5B</i>												
✓ FAN	✓	1.51	0.02	8.98	1.29	951	554	0.09	0.799	1.251	7.18	38.3
× NOFAN	×	1.51	0.01	4.25	0.42	2391	924	0.09	1.129	0.886	4.79	52.9
<i>Llama-3.2-1B</i>												
✓ FAN	✓	1.36	0.02	11.23	2.47	673	363	0.06	0.577	1.733	6.48	20.4
× NOFAN	×	1.36	0.01	7.24	2.05	1214	440	0.06	0.654	1.529	4.73	23.5
<i>Llama-3.2-3B</i>												
✓ FAN	✓	2.91	0.04	4.61	1.11	1941	1202	0.18	1.469	0.681	6.77	47.5
× NOFAN	×	2.91	0.03	2.60	0.52	3718	1630	0.19	1.857	0.538	4.83	55.9
<i>gemma (Gemma 3n)*</i>												
✓ FAN	✓	2.16	0.06	5.68	0.09	2613	1885	0.29	1.214	0.824	6.90	10.6
× NOFAN	×	2.16	0.04	3.07	0.38	5165	2247	0.31	1.559	0.642	4.78	13.7
<i>granite (Granite SSM)[†]</i>												
✓ FAN	✓	2.46	0.07	4.41	0.06	2348	1611	0.25	1.519	0.658	6.70	9.1
× NOFAN	×	2.46	0.06	2.88	0.30	4227	1893	0.26	1.722	0.581	4.96	10.4
<i>ministral</i>												
✓ FAN	✓	3.01	0.07	4.73	0.71	1929	1310	0.22	1.440	0.694	6.81	22.7
× NOFAN	×	3.01	0.05	2.71	0.46	3703	1476	0.24	1.784	0.560	4.84	27.4

Tabla C.2: Serie E0 —Métricas de hardware durante inferencia. Misma configuración que la Tabla C.1. $T_{\text{máx}}$: temperatura máxima registrada ($^{\circ}\text{C}$). $\bar{T}$: temperatura media; σ_T : desviación típica ($^{\circ}\text{C}$). Throttle: porcentaje de tiempo con throttling térmico (%); 0 % en todos los runs FAN, ≈ 100 % en NOFAN. Mem: uso medio de RAM durante inferencia activa (%). CPU: uso medio de CPU durante inferencia (%); σ_{CPU} : desviación típica (%). $\bar{P}$: potencia media (W); σ_P : desviación típica (W).

Modelo	Ref.	$T_{\text{máx}}$ ($^{\circ}\text{C}$)	$\bar{T}$ ($^{\circ}\text{C}$)	σ_T ($^{\circ}\text{C}$)	Throttle (%)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)
<i>DeepSeek-R1-1.5B</i>										
✓ FAN	✓	75.5	72.0	1.6	0.0	17.8	97.8	1.9	7.18	0.86
× NOFAN	×	96.3	94.5	0.9	100.0	17.8	97.3	2.2	4.79	0.57
<i>Llama-3.2-1B</i>										
✓ FAN	✓	70.0	66.0	1.8	0.0	16.1	98.0	2.3	6.48	0.93
× NOFAN	×	93.0	90.1	1.0	99.9	16.0	97.5	2.0	4.73	0.52
<i>Llama-3.2-3B</i>										
✓ FAN	✓	73.2	67.2	2.0	0.0	34.4	99.1	2.2	6.77	0.77
× NOFAN	×	93.0	90.3	1.0	99.9	34.4	99.0	1.9	4.83	0.42
<i>gemma (Gemma 3n)</i>										
✓ FAN	✓	72.2	68.6	2.2	0.0	25.6	97.2	3.9	6.90	1.04
× NOFAN	×	95.8	91.8	2.3	99.8	25.6	97.1	3.5	4.78	0.51
<i>granite (Granite SSM)</i>										
✓ FAN	✓	69.4	65.9	1.8	0.0	29.1	98.8	4.3	6.70	0.81
× NOFAN	×	95.8	92.6	2.4	99.7	29.1	98.8	3.5	4.96	0.42
<i>ministral</i>										
✓ FAN	✓	73.2	67.6	2.4	0.0	35.6	98.9	3.0	6.81	0.83
× NOFAN	×	93.6	90.7	1.2	99.9	35.6	98.8	2.6	4.84	0.42

Tabla C.3: Serie E0 —Caracterización base, eficiencia computacional. Misma configuración que la Tabla C.1. W_{CPU} : trabajo CPU efectivo (core·s); η_{CPU} : eficiencia de cómputo [0,1]; W_{tok} : trabajo por token (core·s/tok); MBU_{corr} : utilización de ancho de banda corregida por arquitectura (%); FoM_{full} : figura de mérito compuesta (Llama-3.2-3B FAN = 1,000). *cota inferior de parámetros activos (MatFormer); † corrección SSM.

Modelo	Ref.	W_{CPU} (core·s)	η_{CPU}	W_{tok} (cs/tok)	MBU_{corr} (%)	FoM_{full}	Usable
<i>DeepSeek-R1-1.5B</i>							
✓ FAN	✓	20812.1	0.975	0.484	63.7	1.025	✓
× NOFAN	×	14906.2	0.337	0.347	30.4	0.497	✓
<i>Llama-3.2-1B</i>							
✓ FAN	✓	12274.6	0.975	0.461	58.8	0.961	✓
× NOFAN	×	8719.6	0.440	0.328	37.3	0.610	✓
<i>Llama-3.2-3B</i>							
✓ FAN	✓	27700.8	0.987	1.214	62.5	1.000	✓
× NOFAN	×	20617.6	0.446	0.904	36.2	0.585	×

(continúa en la página siguiente)

Modelo	Ref.	W_{CPU} (core·s)	η_{CPU}	W_{tok} (cs/tok)	MBU_{corr} (%)	FoM _{full}	Usable
<i>gemma (Gemma 3n)*</i>							
✓ FAN	✓	5819.4	0.946	0.731	55.5	0.818	✓
× NOFAN	×	4613.3	0.402	0.579	29.6	0.455	×
<i>granite (Granite SSM)[†]</i>							
✓ FAN	✓	5234.2	0.968	0.930	53.0	0.934	✓
× NOFAN	×	3544.5	0.422	0.630	34.3	0.593	×
<i>ministral</i>							
✓ FAN	✓	13078.9	0.980	1.124	66.1	1.058	✓
× NOFAN	×	9724.0	0.429	0.836	38.0	0.618	×

C.2. Serie E1 —Sensibilidad a la cuantización

Tabla C.4: Serie E1 —Sensibilidad a la cuantización, rendimiento de inferencia. Condición fija: FAN activo, engine LLAMA, contexto 4096, lote 512. BPW: bits por peso (efectivos). TTFT: tiempo al primer token (ms). σ : desviación típica intra-run entre prompts. J/tok: energía por token; tok/J: su inversa. PPL: perplexidad Wikitext-2 (contexto 512, llama.cpp). [†]Q4_0 y Q8_0 de DeepSeek-R1-1.5B excluidos (colapso de generación por cuantización uniforme en pesos con outliers). RAM: uso medio de memoria RAM en runtime (mean(mem_used_bytes)/1e9, GB métricos).

Cuant.	BPW	RAM (GB)	σ_{RAM} (GB)	tok/s	σ	TTFT (ms)	J/tok	tok/J	PPL	Usable
<i>DeepSeek-R1-1.5B[†]</i>										
Q3_K_M	4.2	0.98	0.01	9.99	1.61	1810	0.760	1.316	46.06	✓
Q4_K_M	5.0	1.51	0.01	8.92	1.30	952	0.796	1.257	43.66	✓
Q5_K_M	5.8	1.65	0.02	8.65	1.06	1076	0.885	1.131	41.75	✓
<i>Llama-3.2-1B</i>										
Q3_K_M	4.5	0.97	0.01	11.39	3.28	1305	0.619	1.616	17.17	✓
Q4_0	5.0	1.32	0.01	11.24	2.69	458	0.554	1.805	17.27	✓
Q4_K_M	5.2	1.36	0.02	11.66	2.61	674	0.553	1.809	15.23	✓
Q5_K_M	5.9	1.46	0.03	11.17	1.50	752	0.648	1.543	14.86	✓
Q8_0	8.6	1.87	0.03	7.65	1.08	951	0.825	1.211	14.71	✓
BF16	16.1	0.56	0.01	1.61	0.05	21514	4.595	0.218	14.69	×
<i>Llama-3.2-3B</i>										
Q3_K_M	4.2	1.82	0.02	5.46	1.46	3562	1.331	0.751	12.07	✓
Q4_0	4.8	2.81	0.04	4.98	1.24	1197	1.306	0.766	11.52	✓
Q4_K_M	5.0	2.91	0.04	4.61	1.12	1936	1.487	0.673	11.24	✓
Q5_K_M	5.8	3.21	0.05	4.29	0.68	2194	1.718	0.582	10.98	✓
Q8_0	8.5	4.34	0.09	3.18	0.30	2830	2.052	0.487	10.89	×
<i>gemma (Gemma 3n)*</i>										
Q3_K_M	5.5	1.43	0.02	6.21	0.33	3864	1.205	0.830	33.00	✓
Q4_0	5.8	2.10	0.03	5.75	0.24	2219	1.175	0.851	37.78	✓
Q4_K_M	5.9	2.16	0.06	5.64	0.09	2693	1.238	0.808	36.06	✓

(continúa en la página siguiente)

Cuant.	BPW	RAM (GB)	σ_{RAM} (GB)	tok/s	σ	TTFT (ms)	J/tok	tok/J	PPL	Usable
Q5_K_M	6.3	2.36	0.06	5.19	0.06	2933	1.462	0.684	33.83	✓
Q8_0	8.6	3.18	0.08	3.80	0.04	3309	1.708	0.585	35.54	×
<i>granite (Granite SSM)[†]</i>										
Q3_K_M	3.9	1.09	0.02	5.17	0.04	4532	1.486	0.673	11.88	✓
Q4_0	4.7	2.37	0.05	4.54	0.05	1616	1.418	0.705	10.08	✓
Q4_K_M	4.9	2.46	0.07	4.41	0.05	2365	1.546	0.647	10.39	✓
Q5_K_M	5.7	2.79	0.07	3.92	0.02	2747	1.940	0.515	10.10	×
Q8_0	8.5	3.92	0.09	2.83	0.02	3480	2.280	0.439	9.96	×
<i>ministral</i>										
Q3_K_M	4.2	1.85	0.03	5.52	0.74	3630	1.434	0.697	9.13	✓
Q4_0	4.8	2.90	0.05	5.11	0.71	1164	1.308	0.764	8.90	✓
Q4_K_M	5.0	3.01	0.09	4.93	0.26	1874	1.427	0.701	8.57	✓
Q5_K_M	5.8	3.33	0.08	4.17	0.31	2342	1.848	0.541	8.47	×
Q8_0	8.5	4.51	0.12	3.00	0.13	3016	2.236	0.447	8.43	×

Tabla C.5: Serie E1 —Métricas de hardware durante inferencia. Misma configuración que la Tabla C.4. $T_{\text{máx}}$: temperatura máxima registrada (°C). $\bar{T}$: temperatura media; σ_T : desviación típica (°C). $\bar{P}$: potencia media activa (W); σ_P : desviación típica (W). E : energía total consumida por el run completo (kJ). Ningún run registró throttling térmico (todos con FAN activo).

Cuant.	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	σ_T (°C)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)	E (kJ)
<i>DeepSeek-R1-1.5B</i>									
Q3_K_M	76.5	72.6	1.7	11.6	97.6	1.7	7.59	1.26	39.4
Q4_K_M	74.3	70.7	1.6	17.9	97.8	1.9	7.10	0.86	38.2
Q5_K_M	76.0	72.4	1.6	19.5	97.9	2.0	7.65	0.95	30.2
<i>Llama-3.2-1B</i>									
Q3_K_M	74.3	69.1	2.5	11.5	97.9	1.9	7.05	1.40	30.6
Q4_0	68.8	64.6	1.3	15.6	98.0	1.9	6.23	0.80	22.5
Q4_K_M	70.0	65.2	1.9	16.1	98.0	2.4	6.44	0.93	18.2
Q5_K_M	74.3	69.9	2.7	17.3	97.9	2.9	7.24	1.18	11.9
Q8_0	68.3	65.3	1.4	22.1	98.5	2.2	6.31	0.61	21.3
BF16	75.5	71.5	2.3	6.6	99.6	0.9	7.41	0.51	85.4
<i>Llama-3.2-3B</i>									
Q3_K_M	76.5	70.3	2.5	21.5	99.1	1.9	7.26	1.28	43.4
Q4_0	71.0	66.9	1.4	33.2	99.2	1.8	6.50	0.60	40.6
Q4_K_M	75.5	69.4	2.1	34.4	99.1	2.2	6.85	0.78	48.2
Q5_K_M	77.1	71.5	2.0	38.0	99.1	2.5	7.38	0.98	32.9
Q8_0	69.4	66.8	1.3	51.3	99.2	2.8	6.53	0.46	27.2
<i>gemma (Gemma 3n)</i>									
Q3_K_M	76.0	72.7	1.8	16.9	97.3	2.3	7.48	1.30	24.6
Q4_0	72.2	69.7	1.6	24.8	97.5	2.4	6.75	0.92	18.9
Q4_K_M	74.3	70.5	2.4	25.5	97.3	3.8	6.98	1.02	10.8

(continúa)

Cuant.	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	σ_T (°C)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)	E (kJ)
Q5_K_M	76.0	72.3	2.3	27.9	97.5	3.7	7.58	1.14	11.6
Q8_0	68.8	66.6	1.5	37.6	98.1	3.4	6.50	0.71	14.3
<i>granite (Granite SSM)</i>									
Q3_K_M	75.5	71.8	2.5	12.9	98.8	3.1	7.68	1.32	8.3
Q4_0	70.0	66.1	1.7	28.0	98.9	3.1	6.44	0.67	7.3
Q4_K_M	72.2	67.8	2.1	29.1	98.8	4.4	6.81	0.83	9.2
Q5_K_M	76.0	71.9	2.8	33.0	99.0	4.0	7.61	0.85	10.9
Q8_0	68.3	66.0	1.4	46.3	99.2	3.6	6.45	0.49	14.3
<i>ministral</i>									
Q3_K_M	76.5	72.8	2.0	21.9	98.6	2.6	7.92	1.43	20.3
Q4_0	73.8	68.1	2.1	34.4	98.9	2.4	6.68	0.73	18.6
Q4_K_M	74.9	70.6	2.3	35.6	98.4	4.2	7.04	0.85	11.7
Q5_K_M	76.5	72.4	1.8	39.4	98.8	3.3	7.70	0.93	19.3
Q8_0	70.5	67.8	1.4	53.3	99.0	3.5	6.71	0.50	18.7

Tabla C.6: Serie E1 —Eficiencia computacional y figura de mérito. Misma configuración que la Tabla C.4. W_{CPU} : trabajo CPU efectivo (core·s); W_{tok} : trabajo por token (core·s/tok); η_{CPU} : eficiencia de cómputo [0,1]; MBU_{corr} : utilización de ancho de banda corregida por arquitectura (%); FoM_{full} : figura de mérito compuesta (referencia: Llama-3.2-3B Q4_K_M = 1,000).

Cuant.	W_{CPU} (core·s)	W_{tok} (cs/tok)	η_{CPU}	MBU_{corr} (%)	FoM_{full}	Usable
<i>DeepSeek-R1-1.5B</i>						
Q3_K_M	20201.1	0.438	0.973	59.6	1.051	✓
Q4_K_M	20976.7	0.488	0.975	63.2	1.026	✓
Q5_K_M	15361.0	0.518	0.974	69.9	1.017	✓
<i>Llama-3.2-1B</i>						
Q3_K_M	16933.4	0.448	0.977	51.2	0.919	✓
Q4_0	14105.5	0.458	0.976	56.2	0.964	✓
Q4_K_M	11037.3	0.465	0.975	60.7	0.992	✓
Q5_K_M	6376.6	0.456	0.971	66.7	0.964	✓
Q8_0	13235.1	0.634	0.979	63.7	0.818	✓
BF16	45773.3	2.641	0.993	24.6	0.286	×
<i>Llama-3.2-3B</i>						
Q3_K_M	23576.5	1.135	0.987	62.6	1.074	✓
Q4_0	24685.2	1.201	0.987	64.1	1.060	✓
Q4_K_M	27800.3	1.219	0.987	62.4	1.000	✓
Q5_K_M	17566.3	1.255	0.984	67.7	0.966	✓
Q8_0	16371.2	1.494	0.981	71.5	0.869	×
<i>gemma (Gemma 3n)*</i>						
Q3_K_M	12644.5	0.705	0.962	57.0	0.850	✓
Q4_0	10745.1	0.748	0.961	55.1	0.832	✓
Q4_K_M	5870.7	0.737	0.946	55.0	0.814	✓

(continúa en la página siguiente)

Cuant.	W_{CPU} (core·s)	W_{tok} (cs/tok)	η_{CPU}	MBU_{corr} (%)	FoM_{full}	Usable
Q5_K_M	5800.3	0.802	0.947	53.2	0.758	✓
Q8_0	8374.3	1.083	0.954	51.3	0.670	×
<i>granite (Granite SSM)[†]</i>						
Q3_K_M	4212.7	0.822	0.970	50.4	0.968	✓
Q4_0	4395.8	0.898	0.969	52.2	0.956	✓
Q4_K_M	5237.1	0.931	0.968	52.9	0.932	✓
Q5_K_M	5568.0	1.047	0.969	54.7	0.863	×
Q8_0	8602.0	1.444	0.972	58.1	0.776	×
<i>ministral</i>						
Q3_K_M	10017.5	0.871	0.977	67.0	1.108	✓
Q4_0	10909.3	1.081	0.979	68.4	1.118	✓
Q4_K_M	6428.9	0.857	0.966	71.2	1.091	✓
Q5_K_M	9760.9	1.050	0.975	67.8	0.971	×
Q8_0	10853.0	1.400	0.973	69.7	0.858	×

C.3. Serie E2 —Sensibilidad al tamaño de contexto

Tabla C.7: Serie E2 —Sensibilidad al tamaño de contexto, rendimiento de inferencia. Condición fija: FAN activo, engine LLAMA, lote 512. Ctx: tamaño de contexto (tokens). TTFT: tiempo al primer token (ms); σ : desviación típica intra-run entre prompts. TTLM: tiempo de carga del modelo (s). Todos los runs completaron con Usable = ✓. RAM: uso medio de memoria RAM en runtime (mean(mem_used_bytes)/1e9, GB métricos).

Ctx (tok)	tok/s	σ	TTFT (ms)	TTLM (s)	J/tok	tok/J	RAM (GB)	σ_{RAM} (GB)	Usable
<i>DeepSeek-R1-1.5B</i>									
512	11.08	0.12	924	0.09	0.651	1.535	1.42	0.04	✓
1024	10.78	0.13	919	0.09	0.650	1.539	1.43	0.03	✓
2048	10.26	0.34	927	0.09	0.681	1.469	1.45	0.02	✓
4096	9.06	1.25	951	0.09	0.767	1.303	1.51	0.02	✓
5120	8.68	1.63	948	0.09	0.813	1.230	1.54	0.01	✓
<i>Llama-3.2-1B</i>									
512	13.12	0.26	652	0.06	0.535	1.869	1.26	0.04	✓
1024	12.91	0.49	649	0.06	0.536	1.867	1.26	0.03	✓
2048	12.37	1.03	651	0.06	0.547	1.830	1.30	0.02	✓
4096	11.31	2.44	671	0.06	0.571	1.750	1.36	0.02	✓
5120	10.85	2.95	690	0.06	0.588	1.700	1.39	0.01	✓
<i>Llama-3.2-3B</i>									
512	5.44	0.05	1887	0.18	1.338	0.747	2.50	0.08	✓
1024	5.23	0.19	1862	0.18	1.355	0.738	2.56	0.07	✓
2048	4.96	0.54	1856	0.18	1.409	0.710	2.68	0.05	✓
4096	4.61	1.12	1936	0.18	1.443	0.693	2.91	0.04	✓
5120	4.50	1.31	1972	0.18	1.487	0.672	3.03	0.04	✓

(continúa)

Ctx (tok)	tok/s	σ	TTFT (ms)	TTLT (s)	J/tok	tok/J	RAM (GB)	σ_{RAM} (GB)	Usable
<i>gemma (Gemma 3n)*</i>									
512	5.70	0.05	2710	0.29	1.223	0.818	2.01	0.06	✓
1024	5.64	0.07	2702	0.29	1.209	0.827	2.04	0.05	✓
2048	5.69	0.10	2605	0.29	1.201	0.833	2.10	0.06	✓
4096	5.67	0.08	2612	0.29	1.232	0.812	2.16	0.05	✓
5120	5.68	0.09	2613	0.29	1.223	0.818	2.22	0.06	✓
<i>granite (Granite SSM)†</i>									
512	4.40	0.04	2416	0.25	1.541	0.649	2.43	0.06	✓
1024	4.42	0.06	2394	0.25	1.520	0.658	2.43	0.07	✓
2048	4.42	0.06	2348	0.25	1.506	0.664	2.45	0.07	✓
4096	4.46	0.07	2349	0.25	1.509	0.663	2.46	0.06	✓
5120	4.41	0.04	2361	0.25	1.530	0.653	2.48	0.07	✓
<i>ministral</i>									
512	5.07	0.07	2011	0.22	1.398	0.715	2.62	0.09	✓
1024	4.94	0.17	1907	0.22	1.423	0.703	2.68	0.07	✓
2048	4.82	0.42	1900	0.22	1.452	0.689	2.78	0.07	✓
4096	4.73	0.71	1921	0.22	1.431	0.699	3.01	0.06	✓
5120	4.71	0.81	1939	0.22	1.442	0.693	3.11	0.06	✓

Tabla C.8: Serie E2 —Métricas de hardware durante inferencia. Misma configuración que la Tabla C.7. $T_{\text{máx}}$: temperatura máxima (°C); $\bar{T}$: temperatura media; σ_T : desviación típica (°C). Throttle: porcentaje de tiempo con throttling térmico (%); 0 % en todos los casos (FAN activo). Mem: uso medio de RAM durante inferencia activa (%). $\bar{P}$: potencia media del run (W); σ_P : desviación típica (W).

Ctx (tok)	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	σ_T (°C)	Throttle (%)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)
<i>DeepSeek-R1-1.5B</i>									
512	74.3	70.3	2.9	0.0	16.8	96.9	4.7	7.22	1.03
1024	71.0	67.6	2.1	0.0	16.9	97.1	3.3	7.00	0.96
2048	70.0	67.2	1.4	0.0	17.1	97.3	2.5	6.98	0.92
4096	73.8	68.6	1.6	0.0	17.8	97.7	1.9	6.95	0.86
5120	74.3	70.1	1.9	0.0	18.2	97.9	1.8	7.05	0.88
<i>Llama-3.2-1B</i>									
512	73.2	68.6	2.9	0.0	14.9	96.8	5.2	7.02	0.99
1024	68.8	65.6	1.9	0.0	14.9	97.0	4.2	6.91	0.96
2048	68.8	65.7	1.4	0.0	15.3	97.2	3.2	6.76	0.90
4096	68.8	65.0	1.4	0.0	16.1	97.9	2.4	6.46	0.93
5120	71.6	66.7	2.0	0.0	16.5	98.2	2.1	6.38	0.94
<i>Llama-3.2-3B</i>									
512	74.9	71.0	2.6	0.0	29.6	98.4	4.9	7.28	0.83
1024	70.5	66.9	1.5	0.0	30.3	98.6	3.8	7.09	0.79
2048	73.8	68.9	2.4	0.0	31.7	98.8	3.0	6.99	0.77
4096	71.6	66.1	1.8	0.0	34.4	99.1	2.2	6.65	0.75

(continúa)

Ctx (tok)	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	σ_T (°C)	Throttle (%)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)
5120	75.5	68.3	2.3	0.0	35.8	99.3	1.9	6.70	0.76
<i>gemma (Gemma 3n)</i>									
512	73.8	70.4	2.6	0.0	23.8	97.2	4.6	6.97	1.03
1024	70.5	67.7	2.1	0.0	24.2	97.3	3.9	6.82	1.03
2048	69.4	66.5	1.9	0.0	24.8	97.3	3.9	6.83	1.00
4096	71.6	68.3	2.3	0.0	25.6	97.3	3.9	6.98	1.01
5120	73.8	70.1	2.3	0.0	26.3	97.3	3.9	6.94	1.03
<i>granite (Granite SSM)</i>									
512	72.2	67.9	2.0	0.0	28.7	98.8	4.6	6.78	0.82
1024	69.4	65.3	1.7	0.0	28.7	98.8	4.3	6.72	0.82
2048	68.8	64.5	1.6	0.0	28.9	98.8	4.3	6.66	0.79
4096	70.0	64.4	1.8	0.0	29.2	98.8	4.4	6.73	0.83
5120	71.6	67.4	1.8	0.0	29.3	98.8	4.4	6.74	0.82
<i>ministral</i>									
512	72.7	68.1	2.5	0.0	31.0	98.3	5.0	7.08	0.89
1024	71.0	67.1	1.8	0.0	31.7	98.5	4.1	7.03	0.85
2048	74.3	70.0	1.9	0.0	32.9	98.6	3.6	6.99	0.82
4096	72.7	66.5	2.1	0.0	35.6	98.9	3.0	6.77	0.81
5120	76.0	69.5	2.9	0.0	36.8	99.0	2.8	6.79	0.84

Tabla C.9: Serie E2 —Sensibilidad al tamaño de contexto, eficiencia computacional. Misma configuración que la Tabla C.7. W_{CPU} (core-s); η_{CPU} [0,1]; W_{tok} (core-s/tok); MBU_{corr} (%); FoM_{full} (referencia Llama-3.2-3B ctx=4096 = 1,000). *cota inferior de parámetros activos (MatFormer); †corrección SSM.

Ctx (tok)	W_{CPU} (core·s)	η_{CPU}	W_{tok} (cs/tok)	MBU_{corr} (%)	FoM_{full}
<i>DeepSeek-R1-1.5B</i>					
512	2362.7	0.945	0.371	73.3	1.164
1024	5015.2	0.959	0.377	72.3	1.156
2048	10054.6	0.967	0.398	70.5	1.124
4096	20536.7	0.974	0.478	64.3	1.036
5120	26247.4	0.977	0.523	62.1	1.002
<i>Llama-3.2-1B</i>					
512	1678.1	0.942	0.314	63.2	1.024
1024	2892.2	0.954	0.321	63.4	1.023
2048	5382.3	0.964	0.349	62.9	1.008
4096	12115.6	0.975	0.455	59.3	0.963
5120	16738.6	0.979	0.528	56.9	0.938
<i>Llama-3.2-3B</i>					
512	4291.9	0.959	0.756	66.1	1.070
1024	7596.4	0.971	0.788	65.3	1.056
2048	12991.7	0.979	0.889	64.6	1.031
4096	27794.7	0.987	1.218	62.4	1.000

(continúa)

Ctx (tok)	W_{CPU} (core·s)	η_{CPU}	W_{tok} (cs/tok)	MBU_{corr} (%)	FoM _{full}
5120	38216.0	0.989	1.420	61.3	0.983
<i>gemma (Gemma 3n)*</i>					
512	4019.1	0.934	0.734	48.3	0.784
1024	5808.2	0.946	0.735	48.8	0.789
2048	5816.3	0.946	0.730	51.3	0.802
4096	5819.9	0.946	0.731	55.3	0.811
5120	5826.3	0.946	0.732	57.5	0.821
<i>granite (Granite SSM)[†]</i>					
512	4481.8	0.965	0.937	52.1	0.921
1024	5192.2	0.968	0.930	52.4	0.927
2048	5218.2	0.968	0.927	52.7	0.931
4096	5181.3	0.968	0.921	53.5	0.936
5120	5235.5	0.968	0.930	53.1	0.928
<i>ministral</i>					
512	4227.2	0.956	0.814	64.8	1.068
1024	6485.5	0.967	0.837	64.3	1.057
2048	8731.5	0.973	0.911	64.5	1.048
4096	13069.5	0.980	1.123	66.1	1.055
5120	16213.6	0.983	1.281	66.8	1.055

C.4. Serie E3 —Sensibilidad al tamaño de lote

Tabla C.10: Serie E3 —Sensibilidad al tamaño de lote, rendimiento de inferencia. Condición fija: FAN activo, engine LLAMA, contexto 4096 tokens. Batch: tamaño de lote (tokens). TTFT: tiempo al primer token (ms); σ : desviación típica intra-run entre prompts. TTLM: tiempo de carga (s). RAM: memoria media ocupada en runtime ($\overline{\text{mem_used_bytes}}/10^9$, en GB). Todos los runs con Usable = ✓.

Batch (tok)	tok/s	σ	TTFT (ms)	TTLM (s)	J/tok	tok/J	$\bar{P}$ (W)	RAM (GB)	σ_{RAM} (GB)
<i>DeepSeek-R1-1.5B</i>									
128	9.05	1.26	950	0.09	0.779	1.284	7.05	1.51	0.01
256	9.12	1.22	942	0.09	0.761	1.314	6.94	1.51	0.01
512	9.08	1.23	950	0.09	0.763	1.311	6.92	1.51	0.02
1024	9.02	1.27	952	0.09	0.766	1.305	6.91	1.51	0.01
2048	9.17	1.14	948	0.09	0.758	1.319	6.96	1.51	0.02
<i>Llama-3.2-1B</i>									
128	11.26	2.47	673	0.06	0.582	1.719	6.55	1.36	0.02
256	11.25	2.36	675	0.06	0.573	1.744	6.45	1.36	0.02
512	11.21	2.43	675	0.06	0.584	1.711	6.55	1.36	0.01
1024	11.25	2.45	675	0.06	0.583	1.717	6.55	1.36	0.02
2048	11.32	2.40	670	0.06	0.579	1.728	6.55	1.36	0.02
<i>Llama-3.2-3B</i>									

(continúa)

Batch (tok)	tok/s	σ	TTFT (ms)	TTLM (s)	J/tok	tok/J	$\bar{P}$ (W)	RAM (GB)	σ_{RAM} (GB)
128	4.60	1.10	1942	0.18	1.450	0.689	6.67	2.90	0.04
256	4.59	1.09	1940	0.18	1.462	0.684	6.71	2.91	0.04
512	4.62	1.10	1932	0.18	1.453	0.688	6.72	2.91	0.05
1024	4.62	1.12	1941	0.18	1.437	0.696	6.63	2.91	0.04
2048	4.61	1.10	1934	0.18	1.442	0.694	6.64	2.91	0.04
<i>gemma (Gemma 3n)*</i>									
128	5.68	0.08	2611	0.29	1.220	0.819	6.93	2.15	0.06
256	5.71	0.11	2597	0.29	1.207	0.828	6.90	2.16	0.05
512	5.69	0.08	2612	0.29	1.224	0.817	6.97	2.17	0.05
1024	5.68	0.08	2610	0.29	1.221	0.819	6.94	2.16	0.06
2048	5.69	0.09	2611	0.29	1.215	0.823	6.91	2.16	0.06
<i>granite (Granite SSM)[†]</i>									
128	4.40	0.05	2352	0.25	1.519	0.658	6.69	2.44	0.07
256	4.47	0.07	2328	0.25	1.489	0.672	6.66	2.45	0.07
512	4.40	0.05	2355	0.25	1.534	0.652	6.75	2.46	0.07
1024	4.41	0.05	2360	0.25	1.526	0.655	6.73	2.46	0.07
2048	4.44	0.08	2344	0.25	1.522	0.657	6.76	2.46	0.07
<i>ministral</i>									
128	4.73	0.70	1927	0.22	1.433	0.698	6.77	3.00	0.06
256	4.74	0.70	1923	0.22	1.416	0.706	6.71	3.01	0.06
512	4.71	0.69	1926	0.22	1.420	0.704	6.68	3.01	0.06
1024	4.73	0.71	1925	0.22	1.433	0.698	6.77	3.01	0.06
2048	4.74	0.71	1931	0.22	1.412	0.708	6.69	3.01	0.07

Tabla C.11: Serie E3 —Métricas de hardware durante inferencia. Misma configuración que la Tabla C.10. $T_{\text{máx}}$: temperatura máxima (°C); $\bar{T}$: temperatura media; σ_T : desviación típica (°C). Throttle: 0 % en todos los casos (FAN activo). Mem: uso medio de RAM (%); $\bar{P}$: potencia media (W); σ_P : desviación típica (W).

Batch (tok)	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	σ_T (°C)	Throttle (%)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)
<i>DeepSeek-R1-1.5B</i>									
128	74.3	70.8	1.5	0.0	17.8	97.8	1.9	7.05	0.88
256	70.0	66.1	1.4	0.0	17.8	97.8	1.9	6.94	0.88
512	71.0	67.5	1.3	0.0	17.8	97.8	1.9	6.92	0.87
1024	70.0	67.3	1.2	0.0	17.8	97.8	1.9	6.91	0.86
2048	70.5	67.2	1.2	0.0	17.8	97.7	1.9	6.96	0.87
<i>Llama-3.2-1B</i>									
128	71.0	66.1	1.8	0.0	16.0	98.0	2.4	6.55	0.95
256	69.4	65.0	1.5	0.0	16.0	97.9	2.4	6.45	0.94
512	71.0	66.0	1.7	0.0	16.1	98.0	2.3	6.55	0.95
1024	70.0	65.9	1.7	0.0	16.1	97.9	2.4	6.55	0.95
2048	69.4	64.9	1.4	0.0	16.0	97.9	2.4	6.55	0.95
<i>Llama-3.2-3B</i>									

(continúa)

Batch (tok)	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	σ_T (°C)	Throttle (%)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)
128	72.7	66.9	2.1	0.0	34.3	99.1	2.2	6.67	0.75
256	71.6	66.1	1.8	0.0	34.4	99.1	2.2	6.71	0.76
512	72.2	65.7	1.4	0.0	34.5	99.2	2.2	6.72	0.75
1024	71.0	66.0	1.8	0.0	34.4	99.1	2.2	6.63	0.76
2048	70.5	65.7	1.5	0.0	34.4	99.1	2.2	6.64	0.74
<i>gemma (Gemma 3n)</i>									
128	72.2	69.1	2.0	0.0	25.4	97.3	3.9	6.93	0.99
256	68.8	66.2	1.5	0.0	25.5	97.3	3.9	6.90	1.02
512	71.6	68.2	1.9	0.0	25.6	97.3	3.9	6.97	1.03
1024	71.0	67.9	1.9	0.0	25.5	97.2	3.9	6.94	1.02
2048	72.2	68.5	2.5	0.0	25.6	97.3	3.9	6.91	1.01
<i>granite (Granite SSM)</i>									
128	68.8	65.5	1.8	0.0	28.9	98.8	4.4	6.69	0.82
256	70.0	64.4	1.6	0.0	29.0	98.8	4.4	6.66	0.81
512	70.0	65.7	1.6	0.0	29.1	98.8	4.3	6.75	0.83
1024	68.8	65.4	1.6	0.0	29.1	98.8	4.4	6.73	0.81
2048	70.5	65.3	1.8	0.0	29.1	98.8	4.4	6.76	0.85
<i>ministral</i>									
128	74.3	68.6	2.7	0.0	35.5	98.9	3.0	6.77	0.81
256	71.6	66.5	1.8	0.0	35.6	98.9	3.0	6.71	0.81
512	71.6	66.5	1.8	0.0	35.6	98.9	3.0	6.68	0.79
1024	72.2	66.9	2.0	0.0	35.6	98.9	3.0	6.77	0.81
2048	71.6	66.2	1.8	0.0	35.6	98.9	3.0	6.69	0.80

Tabla C.12: Serie E3 —Sensibilidad al tamaño de lote, eficiencia computacional. Misma configuración que la Tabla C.10. W_{CPU} (core-s); η_{CPU} [0,1]; W_{tok} (core-s/tok); MBU_{corr} (%); FoM_{full} (referencia Llama-3.2-3B bs=512 = 1,000). *cota inferior de parámetros activos (MatFormer); †corrección SSM.

Batch (tok)	W_{CPU} (core·s)	η_{CPU}	W_{tok} (cs/tok)	MBU_{corr} (%)	FoM_{full}
<i>DeepSeek-R1-1.5B</i>					
128	20587.8	0.974	0.479	64.2	1.031
256	20357.0	0.974	0.474	64.8	1.041
512	20470.0	0.974	0.477	64.5	1.038
1024	20687.1	0.974	0.482	64.0	1.033
2048	20138.3	0.974	0.469	65.3	1.045
<i>Llama-3.2-1B</i>					
128	12234.1	0.975	0.460	59.0	0.956
256	12083.8	0.975	0.454	59.2	0.960
512	12256.8	0.975	0.461	58.7	0.953
1024	12229.1	0.975	0.460	58.9	0.955
2048	12049.6	0.975	0.453	59.5	0.961
<i>Llama-3.2-3B</i>					

(continúa)

Batch (tok)	W_{CPU} (core·s)	η_{CPU}	W_{tok} (cs/tok)	MBU _{corr} (%)	FoM _{full}
128	27638.9	0.987	1.212	62.4	0.998
256	27621.7	0.987	1.211	62.4	0.996
512	27493.2	0.987	1.205	62.8	1.000
1024	27726.0	0.987	1.215	62.5	1.002
2048	27567.1	0.987	1.208	62.6	1.000
<i>gemma (Gemma 3n)*</i>					
128	5821.3	0.946	0.731	55.4	0.813
256	5790.7	0.946	0.727	55.8	0.818
512	5803.2	0.946	0.729	55.6	0.814
1024	5812.2	0.946	0.730	55.5	0.814
2048	5810.9	0.946	0.730	55.5	0.815
<i>granite (Granite SSM)†</i>					
128	5249.5	0.968	0.933	52.9	0.928
256	5143.2	0.968	0.914	53.7	0.941
512	5241.0	0.968	0.931	52.9	0.926
1024	5237.9	0.968	0.931	53.0	0.928
2048	5189.6	0.968	0.922	53.3	0.931
<i>ministral</i>					
128	13040.9	0.980	1.121	66.1	1.054
256	12975.7	0.979	1.115	66.3	1.059
512	13025.6	0.980	1.120	65.9	1.054
1024	13086.0	0.980	1.125	66.1	1.054
2048	13046.5	0.980	1.121	66.2	1.059

C.5. Serie E4 —Sensibilidad al motor de inferencia (llama.cpp frente a Ollama)

Tabla C.13: Serie E4 —Comparativa de engine de inferencia, rendimiento. Condición fija: FAN activo, contexto 4096, lote 512. TTFT incluye cold-start en Ollama; σ : desviación típica intra-run entre prompts. TTLM: tiempo de carga por prompt (s); en Ollama incluye *keep-alive*. $\bar{P}$: potencia media del run (W); E : energía total (kJ). FoM_{full}: referencia Llama-3.2-3B llama.cpp = 1,000. Usable: supera T_{hum} (≥ 3 palabras/s). RAM: uso medio de memoria RAM en runtime (mean(mem_used_bytes)/1e9, GB métricos).

Modelo	Engine	RAM (GB)	σ_{RAM} (GB)	tok/s	σ	TTFT (ms)	TTLM (s)	J/tok	tok/J	$\bar{P}$ (W)	E (kJ)
<i>DeepSeek-R1-1.5B</i>											
llama.cpp	LLAMA	1.51	0.02	8.98	1.29	951	0.09	0.799	1.251	7.18	38.3
Ollama	OLLAMA	1.72	0.08	9.49	0.25	3111	1.11	0.748	1.337	7.10	9.3
<i>Llama-3.2-1B</i>											
llama.cpp	LLAMA	1.36	0.02	11.23	2.47	673	0.06	0.577	1.733	6.48	20.4
Ollama	OLLAMA	1.45	0.09	11.16	0.81	2552	0.95	0.616	1.623	6.88	3.6

(continúa)

Modelo	Engine	RAM (GB)	σ_{RAM} (GB)	tok/s	σ	TTFT (ms)	TTLM (s)	J/tok	tok/J	$\bar{P}$ (W)	E (kJ)
<i>Llama-3.2-3B</i>											
llama.cpp	LLAMA	2.91	0.04	4.61	1.11	1941	0.18	1.469	0.681	6.77	47.5
Ollama	OLLAMA	2.98	0.23	5.08	0.22	6563	1.90	1.495	0.669	7.60	9.0
<i>gemma (Gemma 3n)*</i>											
llama.cpp	LLAMA	2.16	0.06	5.68	0.09	2613	0.29	1.214	0.824	6.90	10.6
Ollama	OLLAMA	4.30	0.36	5.54	0.19	7236	3.05	1.255	0.797	6.95	10.9
<i>granite (Granite SSM)[†]</i>											
llama.cpp	LLAMA	2.46	0.07	4.41	0.06	2348	0.25	1.519	0.658	6.70	9.1
Ollama	OLLAMA	2.52	0.17	4.32	0.05	9932	1.63	1.688	0.592	7.29	9.5
<i>ministral</i>											
llama.cpp	LLAMA	3.01	0.07	4.73	0.71	1929	0.22	1.440	0.694	6.81	22.7
Ollama	OLLAMA	3.50	0.13	3.55	0.42	12848	2.36	2.011	0.497	7.14	32.0

Tabla C.14: Serie E4 —Métricas de hardware durante inferencia. Misma configuración que la Tabla C.13. $T_{\text{máx}}$: temperatura máxima (°C); $\bar{T}$: temperatura media; σ_T : desviación típica (°C). Throttle: 0 % en todos los casos (FAN activo). Mem: uso medio de RAM (%); $\bar{P}$: potencia media (W); σ_P : desviación típica (W).

Modelo	Engine	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	σ_T (°C)	Throttle (%)	Mem (%)	CPU (%)	σ_{CPU} (%)	$\bar{P}$ (W)	σ_P (W)
<i>DeepSeek-R1-1.5B</i>										
llama.cpp	LLAMA	75.5	72.0	1.6	0.0	17.8	97.8	1.9	7.18	0.86
Ollama	OLLAMA	76.5	70.9	3.7	0.0	20.4	92.5	9.9	7.10	1.75
<i>Llama-3.2-1B</i>										
llama.cpp	LLAMA	70.0	66.0	1.8	0.0	16.1	98.0	2.3	6.48	0.93
Ollama	OLLAMA	72.2	67.5	3.9	0.0	17.1	90.5	13.3	6.88	1.78
<i>Llama-3.2-3B</i>										
llama.cpp	LLAMA	73.2	67.2	2.0	0.0	34.4	99.1	2.2	6.77	0.77
Ollama	OLLAMA	76.0	71.6	4.4	0.0	35.3	94.7	14.0	7.60	1.62
<i>gemma (Gemma 3n)</i>										
llama.cpp	LLAMA	72.2	68.6	2.2	0.0	25.6	97.2	3.9	6.90	1.04
Ollama	OLLAMA	73.8	69.8	4.4	0.0	50.9	92.1	15.2	6.95	1.76
<i>granite (Granite SSM)</i>										
llama.cpp	LLAMA	69.4	65.9	1.8	0.0	29.1	98.8	4.3	6.70	0.81
Ollama	OLLAMA	73.8	69.8	3.9	0.0	29.9	96.7	12.8	7.29	1.27
<i>ministral</i>										
llama.cpp	LLAMA	73.2	67.6	2.4	0.0	35.6	98.9	3.0	6.81	0.83
Ollama	OLLAMA	74.3	70.9	2.4	0.0	41.4	97.3	7.9	7.14	1.23

Tabla C.15: Serie E4 —Comparativa de engine, eficiencia computacional. Misma configuración que la Tabla C.13. W_{CPU} (core-s); η_{CPU} [0,1]; W_{tok} (core-s/tok); MBU_{corr} (%); FoM_{full} (referencia Llama-3.2-3B llama.cpp = 1,000). *cota inferior de parámetros activos (MatFormer); †corrección SSM.

Modelo	Engine	W_{CPU} (core-s)	η_{CPU}	W_{tok} (cs/tok)	MBU_{corr} (%)	FoM_{full}	Usable
<i>DeepSeek-R1-1.5B</i>							
llama.cpp	LLAMA	20812.1	0.975	0.484	63.7	1.025	✓
Ollama	OLLAMA	4836.9	0.923	0.429	61.9	1.035	×
<i>Llama-3.2-1B</i>							
llama.cpp	LLAMA	12274.6	0.975	0.461	58.8	0.961	✓
Ollama	OLLAMA	1907.7	0.904	0.377	53.9	0.906	✓
<i>Llama-3.2-3B</i>							
llama.cpp	LLAMA	27700.8	0.987	1.214	62.5	1.000	✓
Ollama	OLLAMA	4495.3	0.946	0.849	67.6	1.030	✓
<i>gemma (Gemma 3n)*</i>							
llama.cpp	LLAMA	5819.4	0.946	0.731	55.5	0.818	✓
Ollama	OLLAMA	5753.2	0.920	0.759	42.9	0.751	✓
<i>granite (Granite SSM)†</i>							
llama.cpp	LLAMA	5234.2	0.968	0.930	53.0	0.934	✓
Ollama	OLLAMA	5031.7	0.966	1.029	48.2	0.883	✓
<i>ministral</i>							
llama.cpp	LLAMA	13078.9	0.980	1.124	66.1	1.058	✓
Ollama	OLLAMA	17427.0	0.972	1.334	46.1	0.826	×

C.6. Serie E5 —Comparativa CPU frente a Hailo-10H

Tabla C.16: Serie E5 —Comparativa CPU (llama.cpp) frente a acelerador Hailo-10H. Condición fija: FAN activo, contexto 4096, lote 512. $T_{norm} = tok/s_{Hailo}/tok/s_{CPU}$ (por modelo); $\varepsilon_{norm} = (J/tok)_{CPU}/(J/tok)_{Hailo}$; $FoM_{red} = \sqrt{T_{norm} \cdot \varepsilon_{norm}}$ (> 1 : Hailo supera el balance rendimiento/energía de la CPU). Todas las magnitudes eléctricas provienen del vatímetro externo¹ (resolución $\pm 2\%$): P_{min}/P_{max} : potencia mínima/máxima instantánea (W); E : energía total (kJ). J/tok calculado desde vatímetro en ambos sustratos para comparabilidad directa¹.

Modelo	Sustrato	tok/s	J/tok	tok/J	P_{min} (W)	P_{max} (W)	E (kJ)	T_{norm}	ε_{norm}	FoM_{red}
<i>DeepSeek-R1-1.5B</i>										
CPU	LLAMA	10.253	1.139	0.878	3.9	11.5	28.8	1.000	1.000	1.000
Hailo	HAILO_OLLAMA	6.783	1.114	0.898	3.9	6.0	7.2	0.662	1.023	0.822
<i>Llama-3.2-3B</i>										
CPU	LLAMA	5.025	2.462	0.406	3.9	11.8	36.0	1.000	1.000	1.000
Hailo	HAILO_OLLAMA	2.653	1.061	0.942	3.9	6.0	7.2	0.528	2.320	1.107

Tabla C.17: Serie E5 —Métricas de hardware CPU durante inferencia (vcgencmd). Solo runs CPU (llama.cpp); Hailo no reporta hw vía vcgencmd. $T_{\text{máx}}$: temperatura máxima (°C); $\bar{T}$: temperatura media (°C). Throttle: porcentaje de tiempo con throttling térmico (%). Mem: uso medio de RAM (%).

Modelo	$T_{\text{máx}}$ (°C)	$\bar{T}$ (°C)	Throttle (%)	Mem (%)	$\bar{P}$ (W)
DeepSeek-R1-1.5B	74.3	70.5	0.0	17.1	7.19
Llama-3.2-3B	69.4	65.9	0.0	31.7	6.96

Tabla C.18: Serie E5 —Eficiencia computacional CPU durante inferencia. Misma configuración que la Tabla C.16. W_{CPU} (core·s); η_{CPU} [0,1]; W_{tok} (core·s/tok); MBU_{corr} (%). Solo runs CPU; Hailo ejecuta en acelerador dedicado.

Modelo	W_{CPU} (core·s)	η_{CPU}	W_{tok} (cs/tok)	MBU_{corr} (%)
DeepSeek-R1-1.5B	10051.6	0.967	0.398	70.5
Llama-3.2-3B	12793.2	0.979	0.875	65.5

Bibliografía

- [1] J. Kaplan, S. McCandlish, T. Henighan et al., *Scaling Laws for Neural Language Models*, arXiv:2001.08361 [cs], ene. de 2020. doi:10.48550/arXiv.2001.08361.
- [2] A. Vaswani, N. Shazeer, N. Parmar et al., *Attention Is All You Need*, en, arXiv:1706.03762 [cs], ago. de 2023. doi:10.48550/arXiv.1706.03762.
- [3] K. Hu y K. Hu, «ChatGPT sets record for fastest-growing user base - analyst note,» en, *Reuters*, feb. de 2023. dirección: <https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/>.
- [4] B. Zoph, C. Raffel, D. Schuurmans et al., «Emergent abilities of large language models,» *TMLR*, 2022.
- [5] T. B. Brown, B. Mann, N. Ryder et al., «Language models are few-shot learners,» en *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ép. NIPS '20, Red Hook, NY, USA: Curran Associates Inc., 2020, págs. 1877-1901, ISBN: 978-1-7138-2954-6. dirección: <https://dl.acm.org/doi/10.5555/3495724.3495883>.
- [6] C. O'Neil, *Armas de destrucción matemática: cómo el Big Data aumenta la desigualdad y amenaza la democracia*, spa. Madrid: Capitán Swing, 2018, OCLC: 1050272580, ISBN: 978-84-947408-4-8.
- [7] Amnesty International, «Unlawful by Design: Exposing the Human Rights Costs of Generative AI,» Amnesty International, Research Briefing POL 40/0996/2026, mayo de 2026. dirección: <https://www.amnesty.org/en/documents/pol40/0996/2026/en/>.
- [8] N. Maslej, L. Fattorini, R. Perrault et al., *Artificial Intelligence Index Report 2025*, arXiv:2504.07139 [cs], nov. de 2025. doi:10.48550/arXiv.2504.07139.
- [9] Agencia Española de Supervisión de Inteligencia Artificial (AESIA), *ALIA - Agencia Española de Supervisión de Inteligencia Artificial*. dirección: <https://alia.gob.es/>.
- [10] España. Consorcio Barcelona Supercomputing Center-Centro Nacional de Supercomputación, *Resolución de 3 de junio de 2025, del Consorcio Barcelona Supercomputing Center-Centro Nacional de Supercomputación, por la que se publica el Convenio con las entidades participantes para la realización de actuaciones ligadas al desarrollo del Plan Nacional de Tecnologías del Lenguaje y del Ecosistema Empresarial de las Tecnologías del Lenguaje*, Boletín Oficial del Estado, núm. 143, sec. III, pp. 78483-78516, Referencia BOE-A-2025-12053. Disponible en https://www.boe.es/diario_boe/txt.php?id=BOE-A-2025-12053, jun. de 2025.
- [11] Mistral AI, *AI for Citizens*. dirección: <https://mistral.ai/news/ai-for-citizens>.
- [12] Unión Europea, *Reglamento (UE) 2024/1689 del Parlamento Europeo y del Consejo, de 13 de junio de 2024, sobre la inteligencia artificial (Ley de Inteligencia Artificial) y por el que se modifican los Reglamentos (CE) n.º 302/2013 y (UE) n.º 1024/2012 y la Directiva (UE) 2019/770*, Diario Oficial de la Unión Europea, 2024. dirección: <https://eur-lex.europa.eu/legal-content/ES/TXT/?uri=celex%3A32024R1689>.

- [13] O. Friha, M. Amine Ferrag, B. Kantarci, B. Cakmak, A. Ozgun y N. Ghoualmi-Zine, «LLM-Based Edge Intelligence: A Comprehensive Survey on Architectures, Applications, Security and Trustworthiness,» *IEEE Open Journal of the Communications Society*, vol. 5, págs. 5799-5856, 2024. [doi:10.1109/OJCOMS.2024.3456549](https://doi.org/10.1109/OJCOMS.2024.3456549).
- [14] U. Kamath, K. Keenan, G. Somers y S. Sorenson, «LLMs: Evolution and New Frontiers,» en *Large Language Models: A Deep Dive*, Cham: Springer Nature Switzerland, 2024, págs. 423-438. [doi:10.1007/978-3-031-65647-7_10](https://doi.org/10.1007/978-3-031-65647-7_10).
- [15] W. M. Adams, «Geographies of conservation II: Technology, surveillance and conservation by algorithm,» EN, *Progress in Human Geography*, vol. 43, n.º 2, págs. 337-350, abr. de 2019. [doi:10.1177/0309132517740220](https://doi.org/10.1177/0309132517740220).
- [16] R. Kwok, «AI empowers conservation biology,» en, *Nature*, vol. 567, n.º 7746, págs. 133-134, mar. de 2019. [doi:10.1038/d41586-019-00746-1](https://doi.org/10.1038/d41586-019-00746-1).
- [17] O. R. Wearn, R. Freeman y D. M. P. Jacoby, «Responsible AI for conservation,» en, *Nature Machine Intelligence*, vol. 1, n.º 2, págs. 72-73, feb. de 2019. [doi:10.1038/s42256-019-0022-7](https://doi.org/10.1038/s42256-019-0022-7).
- [18] D. Velasco-Montero, J. Fernández-Berni y A. Rodríguez-Vázquez, «A Case Study: Remote Animal Monitoring,» en, en *Visual Inference for IoT Systems: A Practical Approach*, Cham: Springer International Publishing, 2022, págs. 125-159. [doi:10.1007/978-3-030-90903-1_6](https://doi.org/10.1007/978-3-030-90903-1_6).
- [19] D. Velasco-Montero, C. Lozano-Pons, J. Fernández-Berni y G. Bastianelli, «On-site acoustic identification of bird species based on a shallow neural network,» *Ecological Informatics*, vol. 94, págs. 103-117, 2026. [doi:https://doi.org/10.1016/j.ecoinf.2026.103687](https://doi.org/10.1016/j.ecoinf.2026.103687).
- [20] S. A. Reynolds, S. Beery, N. Burgess et al., «The potential for AI to revolutionize conservation: a horizon scan,» en, *Trends in Ecology & Evolution*, vol. 40, n.º 2, págs. 191-207, feb. de 2025. [doi:10.1016/j.tree.2024.11.013](https://doi.org/10.1016/j.tree.2024.11.013).
- [21] Z. Miao, Y. Zhang, Z. Fabian et al., «New frontiers in artificial intelligence for biodiversity research and conservation with multimodal language models,» en, *Methods in Ecology and Evolution*, vol. 17, n.º 2, págs. 238-256, 2026, eprint: <https://besjournals.onlinelibrary.wiley.com/doi/pdf/10.1111/2041-210x.70120>. [doi:10.1111/2041-210x.70120](https://doi.org/10.1111/2041-210x.70120).
- [22] P. N. Whatmough, «Generative Artificial Intelligence on Edge Devices: Models, Hardware, and Systems: A tutorial,» *IEEE Solid-State Circuits Magazine*, vol. 17, n.º 3, págs. 32-37, 2025. [doi:10.1109/MSSC.2025.3579704](https://doi.org/10.1109/MSSC.2025.3579704).
- [23] U. Kamath, K. Keenan, G. Somers y S. Sorenson, «Large Language Models: An Introduction,» en, en *Large Language Models: A Deep Dive*, Cham: Springer Nature Switzerland, 2024, págs. 1-27. [doi:10.1007/978-3-031-65647-7_1](https://doi.org/10.1007/978-3-031-65647-7_1).
- [24] Z. Wang, Z. Chu, T. V. Doan, S. Ni, M. Yang y W. Zhang, «History, development, and principles of large language models: an introductory survey,» en, *AI and Ethics*, vol. 5, n.º 3, págs. 1955-1971, jun. de 2025. [doi:10.1007/s43681-024-00583-7](https://doi.org/10.1007/s43681-024-00583-7).
- [25] *Computers and Automation 1954-02: Vol 3 Iss 2*, English. Berkeley Enterprises, feb. de 1954. dirección: http://archive.org/details/sim_computers-and-people_1954-02_3_2.
- [26] J. Hutchins, «The first public demonstration of machine translation: the Georgetown-IBM system, 7th January 1954,» en, dirección: <https://open.unive.it/hitrust/books/HutchinsFirst.pdf>.

- [27] W. J. Hutchins, «The Georgetown-IBM experiment demonstrated in January 1954,» en *Proceedings of the 6th Conference of the Association for Machine Translation in the Americas: Technical Papers*, R. E. Frederking y K. B. Taylor, eds., Washington, USA: Springer, sep. de 2004, págs. 102-114. dirección: <https://aclanthology.org/2004.amta-papers.12/>.
- [28] R. ASALE y RAE, *Corpus, corpus | Diccionario de la lengua española*, 23.^a ed., [versión 23.8.1 en línea]. es. dirección: <https://dle.rae.es/corpus>.
- [29] N. Chomsky, *Syntactic Structures*, eng, 1. Aufl. s.l.: Mouton de Gruyter, 2002. doi:10.1515/9783110218329.
- [30] K. D. Cruz, *English: Tree diagram for Chomsky's 'colorless green ideas sleep furiously'*. Abr. de 2022. dirección: https://commons.wikimedia.org/wiki/File:Tree_Diagram_for_Chomsky%27s_Sentence.png.
- [31] J. Weizenbaum, «ELIZA—a computer program for the study of natural language communication between man and machine,» *Commun. ACM*, vol. 9, n.º 1, págs. 36-45, 1966. doi:10.1145/365153.365168.
- [32] J. Shrager, *ELIZA Reinterpreted: The world's first chatbot was not intended as a chatbot at all*, en, arXiv:2406.17650 [cs], sep. de 2024. doi:10.48550/arXiv.2406.17650.
- [33] T. Winograd, «Procedures as a Representation for Data in a Computer Program for Understanding Natural Language,» inglés, ene. de 1971, Accepted: 2004-10-20T20:29:48Z. dirección: <https://dspace.mit.edu/handle/1721.1/7095>.
- [34] L. Rabiner y B. Juang, «An introduction to hidden Markov models,» *IEEE ASSP Magazine*, vol. 3, n.º 1, págs. 4-16, ene. de 1986. doi:10.1109/MASSP.1986.1165342.
- [35] Princeton University, ed., *Wordnet 1.6*, eng. Cambridge, Mass: MIT Press, 1998, ISBN: 978-0-262-56116-7.
- [36] M. Marcus, G. Kim, M. A. Marcinkiewicz et al., «The Penn Treebank: Annotating Predicate Argument Structure,» en *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*, 1994. dirección: <https://aclanthology.org/H94-1020/>.
- [37] M. Marcus, «Building a Large Annotated Corpus of English: The Penn Treebank:» en, Defense Technical Information Center, Fort Belvoir, VA, inf. téc., abr. de 1993. doi:10.21236/ADA273556.
- [38] *English Today 1993-01: Vol 9 Iss 1*, English. Cambridge University Press, ene. de 1993. dirección: http://archive.org/details/sim_english-today_1993-01_9_1.
- [39] Y. Bengio, R. Ducharme, P. Vincent y C. Janvin, «A neural probabilistic language model,» *J. Mach. Learn. Res.*, vol. 3, n.º null, págs. 1137-1155, mar. de 2003, issn: 1532-4435.
- [40] R. Collobert y J. Weston, «A unified architecture for natural language processing: deep neural networks with multitask learning,» en *Proceedings of the 25th international conference on Machine learning*, ép. ICML '08, New York, NY, USA: Association for Computing Machinery, jul. de 2008, págs. 160-167. doi:10.1145/1390156.1390177.
- [41] T. Mikolov, M. Karafiát, L. Burget, J. Černocký y S. Khudanpur, «Recurrent neural network based language model,» en, ISCA, sep. de 2010, págs. 1045-1048. doi:10.21437/Interspeech.2010-343.
- [42] A. Graves, *Generating Sequences With Recurrent Neural Networks*, arXiv:1308.0850 [cs], jun. de 2014. doi:10.48550/arXiv.1308.0850.

- [43] S. Hochreiter y J. Schmidhuber, «Long Short-Term Memory,» en, *Neural Computation*, vol. 9, n.º 8, págs. 1735-1780, nov. de 1997. [doi:10.1162/neco.1997.9.8.1735](https://doi.org/10.1162/neco.1997.9.8.1735).
- [44] T. Mikolov, K. Chen, G. Corrado y J. Dean, *Efficient Estimation of Word Representations in Vector Space*, en, arXiv:1301.3781 [cs], sep. de 2013. [doi:10.48550/arXiv.1301.3781](https://doi.org/10.48550/arXiv.1301.3781).
- [45] T. Mikolov, I. Sutskever, K. Chen, G. Corrado y J. Dean, *Distributed Representations of Words and Phrases and their Compositionality*, arXiv:1310.4546 [cs], oct. de 2013. [doi:10.48550/arXiv.1310.4546](https://doi.org/10.48550/arXiv.1310.4546).
- [46] K. Cho, B. van Merriënboer, C. Gulcehre et al., «Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation,» en *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang y W. Daelemans, eds., Doha, Qatar: Association for Computational Linguistics, oct. de 2014, págs. 1724-1734. [doi:10.3115/v1/D14-1179](https://doi.org/10.3115/v1/D14-1179).
- [47] I. Sutskever, O. Vinyals y Q. V. Le, «Sequence to sequence learning with neural networks,» en *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ép. NIPS'14, vol. 2, Cambridge, MA, USA: MIT Press, 2014, págs. 3104-3112.
- [48] D. Bahdanau, K. Cho e Y. Bengio, *Neural Machine Translation by Jointly Learning to Align and Translate*, Version Number: 7, 2014. [doi:10.48550/ARXIV.1409.0473](https://doi.org/10.48550/ARXIV.1409.0473).
- [49] T. Luong, H. Pham y C. D. Manning, «Effective Approaches to Attention-based Neural Machine Translation,» en *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, L. Márquez, C. Callison-Burch y J. Su, eds., Lisbon, Portugal: Association for Computational Linguistics, sep. de 2015, págs. 1412-1421. [doi:10.18653/v1/D15-1166](https://doi.org/10.18653/v1/D15-1166).
- [50] K. He, X. Zhang, S. Ren y J. Sun, «Deep Residual Learning for Image Recognition,» en *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, págs. 770-778. [doi:10.1109/CVPR.2016.90](https://doi.org/10.1109/CVPR.2016.90).
- [51] J. L. Ba, J. R. Kiros y G. E. Hinton, *Layer Normalization*, arXiv:1607.06450 [stat], jul. de 2016. [doi:10.48550/arXiv.1607.06450](https://doi.org/10.48550/arXiv.1607.06450).
- [52] J. Devlin, M.-W. Chang, K. Lee y K. Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, en, arXiv:1810.04805 [cs], mayo de 2019. [doi:10.48550/arXiv.1810.04805](https://doi.org/10.48550/arXiv.1810.04805).
- [53] C. Raffel, N. Shazeer, A. Roberts et al., «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,» *Journal of Machine Learning Research*, vol. 21, n.º 140, págs. 1-67, 2020. dirección: <http://jmlr.org/papers/v21/20-074.html>.
- [54] A. Askell, Y. Bai, A. Chen et al., *A General Language Assistant as a Laboratory for Alignment*, arXiv:2112.00861 [cs.CL], dic. de 2021. [doi:10.48550/arXiv.2112.00861](https://doi.org/10.48550/arXiv.2112.00861).
- [55] L. Ouyang, J. Wu, X. Jiang et al., «Training language models to follow instructions with human feedback,» en *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho y A. Oh, eds., vol. 35, Curran Associates, Inc., 2022, págs. 27730-27744. dirección: https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.
- [56] Anthropic, «Claude 2 Model Card,» Anthropic, inf. téc., jul. de 2023, Accessed: February 17, 2026. dirección: <https://www-cdn.anthropic.com/bd2a28d2535bfb0494cc8e2a3bf135d2e7523226/Model-Card-Claude-2.pdf>.

- [57] Constitutional.ai, *Constitutional AI | Tracking Anthropic's AI Revolution*, en. dirección: <https://constitutional.ai/>.
- [58] Y. Bai, S. Kadavath, S. Kundu et al., *Constitutional AI: Harmlessness from AI Feedback*, arXiv:2212.08073 [cs.CL], dic. de 2022. doi:10.48550/arXiv.2212.08073.
- [59] Gemini Team, «Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context,» *arXiv preprint arXiv:2403.05530*, 2024. dirección: <https://arxiv.org/abs/2403.05530>.
- [60] A. Grattafiori, A. Dubey, A. Jauhri et al., *The Llama 3 Herd of Models*, arXiv:2407.21783 [cs], nov. de 2024. doi:10.48550/arXiv.2407.21783.
- [61] *The Open Source AI Definition –1.0*, en-US, oct. de 2024. dirección: <https://opensource.org/ai/open-source-ai-definition>.
- [62] DeepSeek-AI, A. Liu, B. Feng et al., *DeepSeek-V3 Technical Report*, 2025. arXiv: 2412.19437 [cs.CL]. dirección: <https://arxiv.org/abs/2412.19437>.
- [63] DeepSeek-AI, *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*, 2025. arXiv: 2501 . 12948 [cs.CL]. dirección: <https://arxiv.org/abs/2501.12948>.
- [64] OpenAI, *OpenAI o1 System Card*, <https://openai.com/index/openai-o1-system-card/>, 2024.
- [65] J. Wei, X. Wang, D. Schuurmans et al., «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models,» en, *Advances in Neural Information Processing Systems*, vol. 35, págs. 24 824-24 837, dic. de 2022. dirección: https://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.
- [66] S. Park, S. Jeon, C. Lee, S. Jeon, B.-S. Kim y J. Lee, *A Survey on Inference Engines for Large Language Models: Perspectives on Optimization and Efficiency*, arXiv:2505.01658 [cs], nov. de 2025. doi:10.48550/arXiv.2505.01658.
- [67] O. contributors, *Ollama*, 2023. dirección: <https://github.com/ollama/ollama>.
- [68] P. Qiu, S. Zhou y E. Ferrara, «Information suppression in large language models: Auditing, quantifying, and characterizing censorship in DeepSeek,» en, *Information Sciences*, vol. 724, págs. 122 702, ene. de 2026. doi:10.1016/j.ins.2025.122702.
- [69] H. Waight, E. Yang, Y. Yuan et al., «State media control influences large language models,» en, *Nature*, págs. 1-9, mayo de 2026. doi:10.1038/s41586-026-10506-7.
- [70] R. Dale, «Sovereign AI in 2025,» en, *Natural Language Processing*, vol. 31, n.º 5, págs. 1312-1321, sep. de 2025. doi:10.1017/nlp.2025.10007.
- [71] A. Gonzalez-Agirre, M. Pàmies, J. Llop et al., *Salamandra Technical Report*, 2025. arXiv: 2502.08489 [cs.CL]. dirección: <https://arxiv.org/abs/2502.08489>.
- [72] L. Marchandon y L. Marchandon, «France's armed forces ministry awards Mistral AI framework agreement,» en, *Reuters*, ene. de 2026. dirección: <https://www.reuters.com/business/frances-armed-forces-ministry-awards-mistral-ai-framework-agreement-2026-01-08/>.
- [73] Ministry of the Economy, Luxembourg, *Mistral AI and Luxembourg enter into a strategic partnership*, https://meco.gouvernement.lu/en/actualites.gouvernement2024+en+actualites+toutes__actualites+communiques+2025+06-juin+17-frieden-mistral-ai.html, Comunicado oficial del Gobierno de Luxemburgo sobre la alianza estratégica con Mistral AI, jun. de 2025.

- [74] Ministry of Defence Singapore (MINDEF), Defence Science and Technology Agency (DSTA) y DSO National Laboratories, *MINDEF, DSTA and DSO partner Mistral AI to advance generative AI for defence applications*, <https://www.dso.org.sg/media-article/mindef-dsta-and-dso-partner-mistral-ai-to-advance-generative-ai-for-defence-applications>, Comunicado oficial sobre la colaboración con Mistral AI para desarrollar modelos generativos de IA aplicados a la defensa de Singapur, mar. de 2025.
- [75] Ministry of High-Tech Industry of the Republic of Armenia, *Ministry of HighTech Industry of Republic of Armenia and Mistral AI team up to empower the Armenian AI ecosystem*, <https://hightech.gov.am/en/tegekatvakan-kentron/ayl/norutyunner/ai-partnership-launched>, Comunicado oficial sobre el acuerdo de colaboración con Mistral AI para reforzar la infraestructura de IA, soluciones GovTech y servicios públicos en Armenia, feb. de 2025.
- [76] C. d. Madrid, *Díaz Ayuso acuerda con la startup francesa Mistral nuevos proyectos de Inteligencia Artificial para los servicios públicos madrileños*, es, Section: Actualidad, dic. de 2025. dirección: <https://www.comunidad.madrid/noticias/2025/12/08/diaz-ayuso-acuerda-startup-francesa-mistral-nuevos-proyectos-inteligencia-artificial-servicios-publicos-madrilenos>.
- [77] *Solutions for government AI strategies and initiatives | Mistral AI*, en. dirección: <https://mistral.ai/solutions/ai-for-citizens>.
- [78] *Announcing AI for Citizens | Mistral AI*, fr. dirección: <https://mistral.ai/fr/news/ai-for-citizens>.
- [79] Colorado General Assembly. «SB24-205: Consumer Protections for Artificial Intelligence.» (2024), dirección: <https://leg.colorado.gov/bills/sb24-205>.
- [80] California State Legislature. «California Senate Bill 942 (2023–2024 Regular Session).» (2024), dirección: https://leginfo.legislature.ca.gov/faces/billStatusClient.xhtml?bill_id=202320240SB942.
- [81] China Law Translate. «Interim Measures for the Management of Generative Artificial Intelligence Services.» (2023), dirección: <https://www.chinalawtranslate.com/en/generative-ai-interim/>.
- [82] Cyberspace Administration of China et al. «生成式人工智能服务管理暂行办法.» Promulgado el 10 jul2023; en vigor desde 15 ago 2023; (2023), dirección: https://www.moj.gov.cn/pub/sfbgw/flfggz/flfggzbmzg/202401/t20240109_493171.html.
- [83] L. Zhang. «China: Amended Cybersecurity Law Takes Effect.» Global Legal Monitor, Library of Congress. (2026), dirección: <https://www.loc.gov/item/global-legal-monitor/2026-01-08/china-amended-cybersecurity-law-takes-effect/>.
- [84] Standing Committee of the National People's Congress. «中华人民共和国网络安全法（修正）» 由第十四届全国人民代表大会常务委员会于 2025 年 10 月 28 日通过，自 2026 年 1 月 1 日起施行. (2025), dirección: https://www.npc.gov.cn/npc/c2/c30834/202510/t20251028_449048.html.
- [85] M. Bastian, *Deepseek's V3 is the latest example of state-controlled censorship in Chinese LLMs*, en-US, dic. de 2024. dirección: <https://the-decoder.com/deepseeks-v3-is-the-latest-example-of-state-controlled-censorship-in-chinese-llms/>.
- [86] A. Naseh, H. Chaudhari, J. Roh, M. Wu, A. Oprea y A. Houmansadr, *R1dacted: Investigating Local Censorship in DeepSeek's R1 Language Model*, en, arXiv:2505.12625 [cs], mayo de 2025. doi:10.48550/arXiv.2505.12625.

- [87] «Taiwan bans government departments from using DeepSeek AI,» en, *Reuters*, feb. de 2025. dirección: <https://www.reuters.com/technology/taiwan-bans-government-departments-using-deepseek-ai-2025-02-03/>.
- [88] «Governments, regulators increase scrutiny of DeepSeek,» en, *Reuters*, jun. de 2025. dirección: <https://www.reuters.com/legal/litigation/governments-regulators-increase-scrutiny-deepseek-2025-06-27/>.
- [89] E. Strubell, A. Ganesh y A. McCallum, «Energy and Policy Considerations for Deep Learning in NLP,» en *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, A. Korhonen, D. Traum y L. Màrquez, eds., Florence, Italy: Association for Computational Linguistics, jul. de 2019, págs. 3645-3650. doi:10.18653/v1/P19-1355.
- [90] F. Jeanquartier, C. Jean-Quartier, P. Rieder et al., «Assessing the carbon footprint of language models: Towards sustainability in AI,» *Resources, Conservation and Recycling*, vol. 226, pág. 108 670, feb. de 2026. doi:10.1016/j.resconrec.2025.108670.
- [91] *Arte es Ética –Artivismo y Autorías de Habla Hispana*, es-AR. dirección: <https://arteesetica.org/>.
- [92] R. Staff, «Anthropic agrees to pay \$1.5 billion to settle author class action,» *Reuters*, 5 de sep. de 2025, Acceso el 19 de febrero de 2026. dirección: <https://www.reuters.com/sustainability/boards-policy-regulation/anthropic-agrees-pay-15-billion-settle-author-class-action-2025-09-05/>.
- [93] *Bartz et al. v. Anthropic PBC*, Acuerdo de \$1.5 billones aprobado preliminarmente en 2025, ago. de 2024. dirección: <https://www.courtlistener.com/docket/69058235/bartz-v-anthropic-pbc/>.
- [94] *Authors Guild v. OpenAI Inc.* 19 de sep. de 2023. dirección: <https://www.courtlistener.com/docket/67810584/authors-guild-v-openai-inc/>.
- [95] S. Shan, J. Cryan, E. Wenger, H. Zheng, R. Hanocka y B. Y. Zhao, «Glaze: protecting artists from style mimicry by text-to-image models,» en *Proceedings of the 32nd USENIX Conference on Security Symposium*, ép. SEC '23, Anaheim, CA, USA: USENIX Association, 2023, ISBN: 978-1-939133-37-3.
- [96] S. Shan, W. Ding, J. Passananti, S. Wu, H. Zheng y B. Y. Zhao, «Nightshade: Prompt-Specific Poisoning Attacks on Text-to-Image Generative Models,» en *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, págs. 807-825. doi:10.1109/SP54263.2024.00207.
- [97] Techaro, *Anubis: Web AI Firewall Utility*, <https://anubis.techaro.lol/>, Repositorio: <https://github.com/TecharoHQ/anubis>. Accedido: 2026-06-03, 2026.
- [98] U. Kamath, K. Keenan, G. Somers y S. Sorenson, «Language Models Pre-training,» en, en *Large Language Models: A Deep Dive*, Cham: Springer Nature Switzerland, 2024, págs. 29-82. doi:10.1007/978-3-031-65647-7_2.
- [99] X. Glorot, A. Bordes e Y. Bengio, «Deep Sparse Rectifier Neural Networks,» en *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson y M. Dudík, eds., ép. Proceedings of Machine Learning Research, vol. 15, Fort Lauderdale, FL, USA: PMLR, nov. de 2011, págs. 315-323. dirección: <https://proceedings.mlr.press/v15/glorot11a.html>.
- [100] C. Guo, F. Cheng, Z. Du et al., «A Survey: Collaborative Hardware and Software Design in the Era of Large Language Models,» *IEEE Circuits and Systems Magazine*, vol. 25, n.º 1, págs. 35-57, 2025. doi:10.1109/MCAS.2024.3476008.

- [101] T. Lin, Y. Wang, X. Liu y X. Qiu, «A survey of transformers,» en *AI Open*, vol. 3, págs. 111-132, 2022. doi:10.1016/j.aiopen.2022.10.001.
- [102] G. Yenduri, R. M. C. S. G et al., *Generative Pre-trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions*, en, arXiv:2305.10435 [cs], mayo de 2023. doi:10.48550/arXiv.2305.10435.
- [103] W. Kwon, Z. Li, S. Zhuang et al., «Efficient Memory Management for Large Language Model Serving with PagedAttention,» en *Proceedings of the 29th Symposium on Operating Systems Principles*, ép. SOSp '23, Koblenz, Germany: Association for Computing Machinery, 2023, págs. 611-626. doi:10.1145/3600006.3613165.
- [104] Y. Huang, L. J. Wan, H. Ye et al., *New Solutions on LLM Acceleration, Optimization, and Application*, arXiv:2406.10903 [cs], jun. de 2024. doi:10.48550/arXiv.2406.10903.
- [105] F. Wang, Z. Zhang, X. Zhang et al., «A Comprehensive Survey of Small Language Models in the Era of Large Language Models: Techniques, Enhancements, Applications, Collaboration with LLMs, and Trustworthiness,» en, *ACM Transactions on Intelligent Systems and Technology*, vol. 16, n.º 6, págs. 1-87, dic. de 2025. doi:10.1145/3768165.
- [106] P. Shaw, J. Uszkoreit y A. Vaswani, «Self-Attention with Relative Position Representations,» en *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, M. Walker, H. Ji y A. Stent, eds., New Orleans, Louisiana: Association for Computational Linguistics, jun. de 2018, págs. 464-468. doi:10.18653/v1/N18-2074.
- [107] J. Su, M. Ahmed, Y. Lu, S. Pan, W. Bo e Y. Liu, «RoFormer: Enhanced transformer with Rotary Position Embedding,» en, *Neurocomputing*, vol. 568, pag. 127 063, feb. de 2024. doi:10.1016/j.neucom.2023.127063.
- [108] D. Hendrycks y K. Gimpel, *Gaussian Error Linear Units (GELUs)*, arXiv:1606.08415 [cs], jun. de 2023. doi:10.48550/arXiv.1606.08415.
- [109] P. Ramachandran, B. Zoph y Q. V. Le, «Searching for Activation Functions,» en, feb. de 2018. dirección: <https://openreview.net/forum?id=Hkuq2EkPf>.
- [110] Y. N. Dauphin, A. Fan, M. Auli y D. Grangier, «Language Modeling with Gated Convolutional Networks,» en *Proceedings of the 34th International Conference on Machine Learning*, D. Precup e Y. W. Teh, eds., ép. Proceedings of Machine Learning Research, vol. 70, PMLR, ago. de 2017, págs. 933-941. dirección: <https://proceedings.mlr.press/v70/dauphin17a.html>.
- [111] Y. Yang, L. Wang, S. Shi, P. Tadepalli, S. Lee y Z. Tu, «On the Sub-layer Functionalities of Transformer Decoder,» en *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He e Y. Liu, eds., Online: Association for Computational Linguistics, nov. de 2020, págs. 4799-4811. doi:10.18653/v1/2020.findings-emnlp.432.
- [112] M. I. Jordan y R. A. Jacobs, «Hierarchical Mixtures of Experts and the EM Algorithm,» *Neural Computation*, vol. 6, n.º 2, págs. 181-214, 1994. doi:10.1162/neco.1994.6.2.181.
- [113] R. A. Jacobs, M. I. Jordan, S. J. Nowlan y G. E. Hinton, «Adaptive Mixtures of Local Experts,» *Neural Computation*, vol. 3, n.º 1, págs. 79-87, 1991. doi:10.1162/neco.1991.3.1.79.

- [114] N. Shazeer, A. Mirhoseini, K. Maziarz et al., «Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer,» en *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, OpenReview.net, 2017. dirección: <https://openreview.net/forum?id=B1ckMDqlg>.
- [115] W. Fedus, B. Zoph y N. Shazeer, «Switch transformers: scaling to trillion parameter models with simple and efficient sparsity,» *J. Mach. Learn. Res.*, vol. 23, n.º 1, 120:5232-120:5270, 2022, ISSN: 1532-4435. dirección: <https://dl.acm.org/doi/10.5555/3586589.3586709>.
- [116] D. Lepikhin, H. Lee, Y. Xu et al., «GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding,» en *International Conference on Learning Representations*, OpenAlex: <https://openalex.org/W3122317902>, mayo de 2021.
- [117] A. Gu, K. Goel y C. Ré, «Efficiently Modeling Long Sequences with Structured State Spaces,» en *The International Conference on Learning Representations (ICLR)*, <https://arxiv.org/abs/2111.00396>, 2022.
- [118] B. Lenz, O. Lieber, A. Arazi et al., «Jamba: Hybrid Transformer-Mamba Language Models,» en *International Conference on Learning Representations*, Y. Yue, A. Garg, N. Peng, F. Sha y R. Yu, eds., vol. 2025, 2025, págs. 67 959-67 984. dirección: https://proceedings.iclr.cc/paper_files/paper/2025/file/a9ed43fa31dc8b4a7d7a673d713dcb5f-Paper-Conference.pdf.
- [119] IBM® Granite 4.0: modelos híbridos hipereficientes y de alto rendimiento para empresas, es-es, oct. de 2025. dirección: <https://www.ibm.com/es-es/new/announcements/ibm-granite-4-0-hyper-efficient-high-performance-hybrid-models>.
- [120] *IEEE Standard for Floating-Point Arithmetic*. doi:10.1109/IEEESTD.2019.8766229.
- [121] P. Micikevicius, S. Narang, J. Alben et al., «Mixed Precision Training,» en, feb. de 2018. dirección: <https://openreview.net/forum?id=r1gs9JgRZ>.
- [122] *BFloat16: The secret to high performance on Cloud TPUs*, en-US. dirección: <https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus>.
- [123] J. Yuan, H. Li, X. Ding et al., *Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference*, arXiv:2506.09501 [cs], oct. de 2025. doi:10.48550/arXiv.2506.09501.
- [124] H. Wang, S. Ma, L. Ma et al., «BitNet: 1-bit Pre-training for Large Language Models,» *Journal of Machine Learning Research*, vol. 26, n.º 125, págs. 1-29, 2025. dirección: <http://jmlr.org/papers/v26/24-2050.html>.
- [125] S. Ma, H. Wang, L. Ma et al., *The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits*, arXiv:2402.17764 [cs], feb. de 2024. doi:10.48550/arXiv.2402.17764.
- [126] E. Kurtic, A. N. Marques, S. Pandit, M. Kurtz y D. Alistarh, «“Give Me BF16 or Give Me Death” ? Accuracy-Performance Trade-Offs in LLM Quantization,» en *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova y M. T. Pilehvar, eds., Vienna, Austria: Association for Computational Linguistics, jul. de 2025, págs. 26 872-26 886. doi:10.18653/v1/2025.acl-long.1304.
- [127] E. J. Husom, A. Goknil, M. Astekin et al., «Sustainable LLM Inference for Edge AI: Evaluating Quantized LLMs for Energy Efficiency, Output Accuracy, and Inference Latency,» *ACM Trans. Internet Things*, vol. 6, n.º 4, nov. de 2025. doi:10.1145/3767742.

- [128] J. Li, J. Xu, S. Huang et al., *Large Language Model Inference Acceleration: A Comprehensive Hardware Perspective*, arXiv:2410.04466 [cs], jun. de 2025. doi:10.48550/arXiv.2410.04466.
- [129] S. Han, J. Pool, J. Tran y W. J. Dally, «Learning both weights and connections for efficient neural networks,» en *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, ép. NIPS'15, Montreal, Canada: MIT Press, 2015, págs. 1135-1143.
- [130] G. Hinton, O. Vinyals y J. Dean, «Distilling the Knowledge in a Neural Network,» en *NIPS Deep Learning and Representation Learning Workshop*, 2015. dirección: <http://arxiv.org/abs/1503.02531>.
- [131] C. Yang, Y. Zhu, W. Lu et al., «Survey on Knowledge Distillation for Large Language Models: Methods, Evaluation, and Application,» *ACM Trans. Intell. Syst. Technol.*, vol. 16, n.º 6, nov. de 2025. doi:10.1145/3699518.
- [132] L. Wang, S. Chen, L. Jiang et al., «Parameter-efficient fine-tuning in large language models: a survey of methodologies,» en *Artificial Intelligence Review*, vol. 58, n.º 8, pág. 227, mayo de 2025. doi:10.1007/s10462-025-11236-4.
- [133] E. J. Hu, Y. Shen, P. Wallis et al., *LoRA: Low-Rank Adaptation of Large Language Models*, arXiv:2106.09685 [cs], oct. de 2021. doi:10.48550/arXiv.2106.09685.
- [134] N. Houlsby, A. Giurgiu, S. Jastrzebski et al., «Parameter-Efficient Transfer Learning for NLP,» en *Proceedings of the 36th International Conference on Machine Learning*, PMLR, mayo de 2019, págs. 2790-2799. dirección: <https://proceedings.mlr.press/v97/houlsby19a.html>.
- [135] X. L. Li y P. Liang, «Prefix-Tuning: Optimizing Continuous Prompts for Generation,» en *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li y R. Navigli, eds., Online: Association for Computational Linguistics, ago. de 2021, págs. 4582-4597. doi:10.18653/v1/2021.acl-long.353.
- [136] E. Ben Zaken, Y. Goldberg y S. Ravfogel, «BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models,» en *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, S. Muresan, P. Nakov y A. Villavicencio, eds., Dublin, Ireland: Association for Computational Linguistics, mayo de 2022, págs. 1-9. doi:10.18653/v1/2022.acl-short.1.
- [137] P. Lewis, E. Perez, A. Piktus et al., «Retrieval-augmented generation for knowledge-intensive NLP tasks,» en *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ép. NIPS '20, Red Hook, NY, USA: Curran Associates Inc., 2020, págs. 9459-9474, ISBN: 978-1-7138-2954-6. dirección: <https://dl.acm.org/doi/10.5555/3495724.3496517>.
- [138] A. Brown, M. Roman y B. Devereux, «A Systematic Literature Review of Retrieval-Augmented Generation: Techniques, Metrics, and Challenges,» en *Big Data and Cognitive Computing*, vol. 9, n.º 12, dic. de 2025. doi:10.3390/bdcc9120320.
- [139] W. Fan, Y. Ding, L. Ning et al., «A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models,» en *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ép. ACM Conferences, ago. de 2024, págs. 6491-6501. doi:10.1145/3637528.3671470.

- [140] H. Chen, C. Tian, Z. He, B. Yu, Y. Liu y J. Cao, *Inference performance evaluation for LLMs on edge devices with a novel benchmarking framework and metric*, en, arXiv:2508.11269 [cs], ago. de 2025. doi:10.48550/arXiv.2508.11269.
- [141] D. Velasco-Montero, J. Fernández-Berni y A. Rodríguez-Vázquez, «Relevant Hardware Metrics for Performance Evaluation,» en, en *Visual Inference for IoT Systems: A Practical Approach*, D. Velasco-Montero, J. Fernández-Berni y A. Rodríguez-Vázquez, eds., Cham: Springer International Publishing, 2022, págs. 61-88. doi:10.1007/978-3-030-90903-1_4.
- [142] D. Velasco-Montero, J. Fernández-Berni y A. Rodríguez-Vázquez, «Optimal Selection of Software and Models for Visual Inference,» en, en *Visual Inference for IoT Systems: A Practical Approach*, D. Velasco-Montero, J. Fernández-Berni y A. Rodríguez-Vázquez, eds., Cham: Springer International Publishing, 2022, págs. 31-60. doi:10.1007/978-3-030-90903-1_3.
- [143] A. Paszke, S. Gross, F. Massa et al., «PyTorch: An Imperative Style, High-Performance Deep Learning Library,» en *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox y R. Garnett, eds., Curran Associates, Inc., 2019, págs. 8024-8035. dirección: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [144] Martín Abadi, Ashish Agarwal, Paul Barham et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. dirección: <https://www.tensorflow.org/>.
- [145] G. Sobhani, M. M. A. Ifath, T. Sharma e I. Haque, *On the Sustainability of AI Inferences in the Edge*, arXiv:2507.23093, jul. de 2025. doi:10.48550/arXiv.2507.23093.
- [146] ggml-org, *GGUF File Format Specification*, <https://github.com/ggml-org/ggml/blob/master/docs/gguf.md>, Accedido: 11 de junio de 2026, 2023.
- [147] G. Gerganov et al., *llama.cpp*, 2023. dirección: <https://github.com/ggml-org/llama.cpp>.
- [148] Hugging Face, *Safetensors: A Simple, Safe Way to Store and Distribute Tensors*, <https://github.com/huggingface/safetensors>, Accedido: 11 de junio de 2026, 2022.
- [149] Google, *Coral: A platform for built-in AI*, <https://coral.ai/>, 2019.
- [150] Raspberry Pi, *Raspberry Pi AI Kit: Product brief*, <https://datasheets.raspberrypi.com/ai-kit/raspberry-pi-ai-kit-product-brief.pdf>.
- [151] Raspberry Pi, *Raspberry Pi AI HAT Plus 2: Product brief*, <https://pip-assets.raspberrypi.com/categories/1319-raspberry-pi-ai-hat-2/documents/RP-009655-MM.pdf>.
- [152] M5Stack, *LLM-8850 AI Accelerator Module*, <https://openlab.io/a/s/products/m5stack-ai-8850-llm-accelerator-kit>, Módulo acelerador de IA basado en Axera AX8850, 2025.
- [153] H. Touvron, T. Lavril, G. Izacard et al., *LLaMA: Open and Efficient Foundation Language Models*, arXiv:2302.13971 [cs], feb. de 2023. doi:10.48550/arXiv.2302.13971.
- [154] Meta AI, *Llama 3.2 Model Card*, GitHub Repository, Llama 3.2 multilingual LLM family (1B & 3B) optimized for dialogue, retrieval, summarization and lightweight deployments, with detailed training data and benchmark metrics, 2024. dirección: https://github.com/meta-llama/llama-models/blob/main/models/llama3_2/MODEL_CARD.md.

- [155] Meta AI, *Llama 3.3 Model Card*, GitHub Repository, Meta Llama 3.3: a pretrained & instruction-tuned 70B multilingual text generation model with Grouped-Query Attention and diverse benchmarks, 2024. dirección: https://github.com/meta-llama/llama-models/blob/main/models/llama3_3/MODEL_CARD.md.
- [156] Meta AI, *Llama 4 Model Card*, GitHub Repository, Llama 4 Scout and Llama 4 Maverick models: multimodal mixture-of-experts architecture, pretrained on publicly available and Meta datasets, with extended context lengths and benchmarks for text and image tasks, 2025. dirección: https://github.com/meta-llama/llama-models/blob/main/models/llama4/MODEL_CARD.md.
- [157] Google DeepMind, *Gemma 4*, <https://ai.google.dev/gemma/docs/core>, Accedido: 12 de junio de 2026, 2026.
- [158] G. Team, «Gemma 3,» 2025. dirección: <https://arxiv.org/abs/2503.19786>.
- [159] A. Yang, B. Zhang, B. Hui et al., «Qwen2.5-Math Technical Report: Toward Mathematical Expert Model via Self-Improvement,» *arXiv preprint arXiv:2409.12122*, 2024.
- [160] M. AI, *Llama 3.1 Model Card*, https://github.com/meta-llama/llama-models/blob/main/models/llama3_1/MODEL_CARD.md, 2024.
- [161] P. H. Martins, P. Fernandes, J. Alves et al., *EuroLLM: Multilingual Language Models for Europe*, arXiv:2409.16235 [cs], sep. de 2024. doi:10.48550/arXiv.2409.16235.
- [162] *utter-project/EuroLLM-1.7B-Instruct · Hugging Face*, feb. de 2026. dirección: <https://huggingface.co/utter-project/EuroLLM-1.7B-Instruct>.
- [163] *utter-project/EuroMoE-2.6B-A0.6B-Instruct-2512 · Hugging Face*, feb. de 2026. dirección: <https://huggingface.co/utter-project/EuroMoE-2.6B-A0.6B-Instruct-2512>.
- [164] *utter-project/EuroLLM-9B-Instruct · Hugging Face*, feb. de 2026. dirección: <https://huggingface.co/utter-project/EuroLLM-9B-Instruct>.
- [165] A. H. Liu, K. Khandelwal, S. Subramanian et al., *Ministral 3*, 2026. arXiv: 2601.08584 [cs.CL]. dirección: <https://arxiv.org/abs/2601.08584>.
- [166] IBM Research, *Granite 4.1 Language Models*, <https://huggingface.co/blog/ibm-granite/granite-4-1>, Accessed: 2026-04-28, 2026.
- [167] T. Dao y A. Gu, «Transformers are SSMs: generalized models and efficient algorithms through structured state space duality,» en *Proceedings of the 41st International Conference on Machine Learning*, ép. ICML'24, Vienna, Austria: JMLR.org, 2024.
- [168] T. Benoit-Cattin, D. Velasco-Montero y J. Fernández-Berni, «Impact of Thermal Throttling on Long-Term Visual Inference in a CPU-Based Edge Device,» en, *Electronics*, vol. 9, n.º 12, pág. 2106, dic. de 2020. doi:10.3390/electronics9122106.
- [169] M. Arya e Y. Simmhan, «Understanding the Performance and Power of LLM Inferencing on Edge Accelerators,» en *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2025, págs. 1108-1111. doi:10.1109/IPDPSW66978.2025.00173.
- [170] J. Zhou, T. Lu, S. Mishra et al., *Instruction-Following Evaluation for Large Language Models*, arXiv:2311.07911 [cs], nov. de 2023. doi:10.48550/arXiv.2311.07911.
- [171] Hailo Technologies Ltd, *DeepSeek-R1-Distill-Qwen-1.5B — Hailo Model Explorer*, Accessed: Jun. 9, 2026, 2026. dirección: <https://hailo.ai/products/hailo-software/model-explorer/generative-ai/deepseek-r1-distill-qwen-1-5b/>.

- [172] Raspberry Pi Ltd, *Processors — Raspberry Pi Documentation*, Section: BCM2712, Accessed: Jun. 9, 2026, 2026. dirección: <https://www.raspberrypi.com/documentation/computers/processors.html>.
- [173] *Different load times for an already loaded model, Why?* · Issue #12443 · ollama/ollama, en. dirección: <https://github.com/ollama/ollama/issues/12443>.
- [174] Raspberry Pi Ltd, «Raspberry Pi 5 Product Brief,» Raspberry Pi Ltd, inf. téc. RP-008348-DS-6, abr. de 2026, Accessed: Jun. 9, 2026. dirección: <https://pip-assets.raspberrypi.com/categories/892-raspberry-pi-5/documents/RP-008348-DS-6-raspberry-pi-5-product-brief.pdf>.
- [175] Raspberry Pi Ltd., «Raspberry Pi 27W USB-C Power Supply,» Raspberry Pi Ltd., Product Brief RP-008245-DS-1, 2023, Accedido: jun. 2026. <https://pip-assets.raspberrypi.com/categories/898-raspberry-pi-27w-usb-c-power-supply/documents/RP-008245-DS-1-27w-usb-c-power-supply-product-brief.pdf>.
- [176] Western Digital Corporation, «SanDisk Extreme UHS-I microSD Memory Card,» Western Digital Corporation, Datasheet, 2022, 128 GB. Accedido: jun. 2026. https://documents.sandisk.com/content/dam/asset-library/en_us/assets/public/sandisk/product/memory-cards/extreme-uhs-i-microsd/data-sheet-extreme-uhs-i-microsd-190mbps.pdf.
- [177] Raspberry Pi Ltd, «Raspberry Pi Active Cooler for Raspberry Pi 5,» Raspberry Pi Ltd, Product Brief RP-008188-DS-2, abr. de 2024, Accessed: Jun. 9, 2026. dirección: <https://pip-assets.raspberrypi.com/categories/993-raspberry-pi-active-cooler/documents/RP-008188-DS-2-raspberry-pi-active-cooler-product-brief.pdf>.
- [178] S. Merity, C. Xiong, J. Bradbury y R. Socher, *Pointer Sentinel Mixture Models*, 2016. arXiv: 1609.07843 [cs.CL].
- [179] Tailscale Inc., *Tailscale*, <https://tailscale.com>, v1.98.2. Accedido: jun. 2026.
- [180] Microsoft, *vcpkg — C++ Library Manager*, <https://github.com/microsoft/vcpkg>, Accedido: jun. 2026.
- [181] Google DeepMind, *Gemma-3n-E2B-it*, Hugging Face model repository, 2025. dirección: <https://huggingface.co/google/gemma-3n-E2B-it>.
- [182] Meta AI, *Llama-3.2-1B-Instruct*, Hugging Face model repository, 2024. dirección: <https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct>.
- [183] Meta AI, *Llama-3.2-3B-Instruct*, Hugging Face model repository, 2024. dirección: <https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct>.
- [184] DeepSeek AI, *DeepSeek-R1-Distill-Qwen-1.5B*, Hugging Face model repository, 2025. dirección: <https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B>.
- [185] J. Carrascosa Carro, *TFG-yisuscc/scripts*, original-date: 2025-02-06T11:55:16Z, mayo de 2026. dirección: <https://github.com/TFG-yisuscc/scripts>.
- [186] IBM Granite Team, *Granite 4.0 H-Micro GGUF*, Hugging Face model repository, 2025. dirección: <https://huggingface.co/ibm-granite/granite-4.0-h-micro-GGUF>.
- [187] Mistral AI, *Ministral-3-3B-Instruct-2512-GGUF*, Hugging Face model repository, 2025. dirección: <https://huggingface.co/mistralai/Ministral-3-3B-Instruct-2512-GGUF>.
- [188] *cron(8) - Linux manual page*. dirección: <https://www.man7.org/linux/man-pages/man8/cron.8.html>.

- [189] SanDisk Corporation, «SanDisk Extreme PRO microSDXC UHS-I Card Datasheet,» SanDisk Corporation, Datasheet, 2024, Accessed: Jun. 9, 2026. dirección: https://documents.sandisk.com/content/dam/asset-library/en_us/assets/public/sandisk/product/memory-cards/extreme-pro-microsd-uhs-i/data-sheet-sandisk-extreme-pro-uhs-i-microsd.pdf.