UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Predicción de latencias en sistemas de microservicios utilizando redes neuronales de grafos heterogéneos (LQ-GNN)

Informe de Proyecto de Grado presentado por

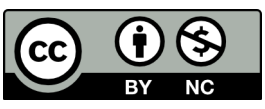
Franco Lepratti Gilles

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

Supervisor

Matías Richart y Javier Baliosian

Montevideo, 2026



Agradecimientos

Quiero expresar mi agradecimiento a mis tutores, Matías Richart y Javier Baliosian, por su orientación, disponibilidad y aportes técnicos durante todo el desarrollo de este Proyecto de Grado. Sus comentarios, correcciones y sugerencias fueron de suma importancia para conseguir el resultado obtenido, agradeciendo que se tomaran el tiempo y las molestias de juntarnos cada semana para tener un seguimiento del proyecto, al igual que integrarme al grupo de trabajo del cual es parte mi proyecto, el cual fue y es un gran grupo de trabajo donde me sentí muy cómodo.

Asimismo, agradezco al grupo de investigación MINA del Instituto de Computación por brindar el marco académico y el espacio de discusión en el que se se encuentra este trabajo. En particular, valoro el intercambio con sus integrantes y el ambiente colaborativo que facilitó el avance del proyecto.

Agradezco también a la Facultad de Ingeniería y al Instituto de Computación de la Universidad de la República por el apoyo institucional y la infraestructura provista para la experimentación. El acceso a recursos computacionales y al clúster de cómputo fue esencial para ejecutar los entrenamientos y evaluaciones reportados en este informe.

Finalmente, agradezco a mi familia y amistades por el acompañamiento, la comprensión y el apoyo brindado a lo largo de este proceso.

Resumen

Este Proyecto de Grado presenta la implementación y evaluación de un modelo de *Red Neuronal de Grafos Heterogéneos* para la predicción de latencias en sistemas basados en microservicios, Este mismo se encuentra en el marco del enfoque propuesto en *LQ-GNN: A Graph Neural Network Model for Latency Prediction of Microservice-based Applications for Elasticity Control in the Computing Continuum* (Richart y cols., 2025). El trabajo propuesto en el paper se enmarca en una línea de investigación orientada al escalado multidimensional de microservicios en el computing continuum —una infraestructura distribuida que integra recursos de cómputo desde el borde de la red hasta la nube—, donde disponer de predicciones de latencia rápidas y confiables resulta relevante para analizar alternativas de configuración, despliegue y asignación de recursos.

Las arquitecturas de microservicios ofrecen ventajas importantes en términos de escalabilidad, mantenibilidad y despliegue independiente. Sin embargo, también introducen desafíos significativos en el apartado de la performance, ya que la latencia end-to-end depende de múltiples factores interrelacionados, entre ellos la topología de dependencias, la distribución de carga, la asignación de recursos y la interacción entre servicios. En este contexto, los enfoques basados en grafos resultan especialmente adecuados, ya que permiten modelar de forma explícita la estructura relacional del sistema.

El modelo implementado representa cada aplicación como un grafo heterogéneo en el que los nodos corresponden a distintas entidades del sistema, en particular tareas, actividades y rutas de ejecución, mientras que las aristas capturan relaciones de dependencia y composición. A través de un esquema de *message passing*, el modelo aprende representaciones internas capaces de estimar la latencia asociada a una consulta bajo determinadas condiciones de carga y configuración. La implementación se desarrolló utilizando PyTorch y PyTorch Geometric, lo que permitió construir una versión funcional, reproducible y más flexible que la implementación del artículo original.

El modelo fue entrenado y evaluado con cuatro datasets (`GNN_4tier`, `GNN_mix`, `GNN_social-network` y `GNN_social-network-ut`), lo que permitió analizar su comportamiento sobre topologías distintas y estudiar su capacidad de generalización. En la configuración base (la misma con la que se entreno la implementación del paper) para predicción de latencia promedio, el mejor resultado se obtuvo en `GNN_4tier`, con un *Mean Absolute Percentage Error* (*MAPE*) de **10,85 %** en test. Para los demás datasets, se obtuvieron los siguientes va-

lores, **33,91 %** en `GNN_mix`, **21,81 %** en `GNN_social-network` y **24,46 %** en `GNN_social-network-ut`. Entonces se realizo un extencion de la evaluación para la predicción del percentil 95 de la latencia y realizar una etapa adicional de optimización, se observaron mejoras sustanciales, alcanzando valores de *MAPE* de **13,71 %** en `GNN_mix`, **7,65 %** en `GNN_social-network` y **8,62 %** en `GNN_social-network-ut`. Estos resultados muestran que la implementación desarrollada ademas de cumplir su propósito original (ser funcional), sino que también presenta un margen de mejora importante cuando se dedica esfuerzo adicional al ajuste experimental.

En conjunto, el trabajo demuestra no solo que es posible implementar el modelo LQ-GNN fuera del framework original, sino también reproducir su lógica sobre un *stack* moderno, el cual va a permitir mas maleabilidad en los detalles del modelo como pueden ser agregaciones o demás cosas que se explicaran en el informe y obtener resultados razonables en distintos escenarios. Más allá de las métricas alcanzadas, la principal contribución radica en haber construido una base experimental y de software reutilizable, osea que a posterior puede ser utilizado para ampliarlo o cambiar cosas del modelo como agregaciones o formas de envió de message passing, ademas de robustez y aplicabilidad en contextos más cercanos a producción.

Palabras clave: Redes Neuronales de Grafos, Microservicios, Predicción de Latencia, Grafos Heterogéneos, LQ-GNN, PyTorch Geometric

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación

Las arquitecturas de microservicios con los años se han convertido en el enfoque predominante para el desarrollo y operación de aplicaciones distribuidas a gran escala, al favorecer el despliegue independiente, la escalabilidad horizontal y la evolución continua del sistema. Este paradigma resulta especialmente adecuado en contextos donde la demanda varía en el tiempo y los requisitos de disponibilidad y mantenibilidad imponen condiciones estrictas de diseño y operación (Newman, 2015; Fowler y Lewis, 2014).

Sin embargo, la descomposición funcional en múltiples servicios incrementa la complejidad de la gestión de performance. En particular, la latencia end-to-end de una consulta es difícil de anticipar, ya que depende de la interacción entre dependencias de llamada, distribución de recursos, variación de carga, contención por concurrencia y retardos de comunicación, además de efectos dinámicos como la aparición de cuellos de botella (Dean y Barroso, 2013a). A diferencia que con sistemas monolíticos, donde ciertas hipótesis de estabilidad pueden hacerce de forma razonables, la heterogeneidad estructural característica de microservicios hace que aproximaciones tradicionales no resulten de manera efectiva o requieran supuestos que simplifican demasiado la realidad.

Disponer de estimaciones de latencia antes de ejecutar una consulta o aplicar una reconfiguración habilita decisiones operativas que pueden ser de gran impacto, tales como planificación de capacidad, asignación eficiente de recursos, evaluación *what-if* de alternativas de despliegue y permitir aplicar politicas de elasticidad. En escenarios de control, un predictor rápido y suficientemente preciso permite incorporar la latencia como dato de entrada para la toma de decision sin caer en el costo de evaluaciones exhaustivas en tiempo de ejecución.

En este contexto, las Redes Neuronales de Grafos (GNN) constituyen una alternativa aparentemente bastante funcional, ya que aprenden representaciones sobre estructuras relacionales y explotan explícitamente la naturaleza de grafo de las dependencias entre componentes. A través de mecanismos de *message*

passing, estos modelos capturan información local y patrones globales en grafos y se adaptan a problemas donde la estructura condiciona el comportamiento observado (Scarselli, Gori, Tsoi, Hagenbuchner, y Monfardini, 2009; Gilmer, Schoenholz, Riley, Vinyals, y Dahl, 2017).

Dentro de esta línea, el grupo de investigación MINA y, en particular, mis supervisores de este Proyecto de Grado desarrollaron previamente el enfoque LQ-GNN (Layered Queueing Network-based Graph Neural Network) (Richart y cols., 2025). Este modelo propone combinar la capacidad descriptiva de las Redes de Colas por Niveles (LQN) con la capacidad de aprendizaje de las Redes Neuronales de Grafos (GNN), representando aplicaciones de microservicios como grafos heterogéneos y utilizando una GNN especializada para predecir la latencia end-to-end en función de la topología y el estado del sistema.

En este contexto, un grafo heterogéneo es un grafo que contiene distintos tipos de nodos y/o relaciones, donde cada tipo representa una entidad o vínculo con semántica propia. En el caso de LQ-GNN, esta heterogeneidad permite distinguir, por ejemplo, entre microservicios, actividades internas y rutas de ejecución.

Dicho trabajo es la base tanto conceptual como metodológico de la que parte este informe y se enmarca en un esfuerzo de investigación orientado a habilitar mecanismos de control y elasticidad basados en un proceso de predicción.

Desde el punto de vista operativo, la entrada del modelo es una instancia de la aplicación representada como un grafo heterogéneo. En dicho grafo, los nodos describen distintas entidades del sistema, como microservicios (`task`), operaciones internas (`activity`) y rutas de ejecución (`path`), junto con atributos asociados a la configuración y al estado del sistema, por ejemplo recursos asignados, carga de entrada, tiempos de procesamiento y características de bloqueo. A partir de esta representación, el modelo aplica un proceso de *message passing* y produce como salida una predicción de latencia, que puede corresponde a el tiempo de respuesta de una aplicación. En un escenario de uso con nuevos datos, el procedimiento consiste en construir el grafo de la nueva configuración de la aplicación con el mismo formato de entrada, aplicar las mismas transformaciones de preprocesamiento utilizadas durante el entrenamiento y ejecutar el modelo entrenado para obtener una estimación de latencia. De esta forma, el predictor puede utilizarse para evaluar rápidamente configuraciones alternativas de despliegue, carga o asignación de recursos antes de aplicarlas sobre el sistema real.

A partir de esta base (el paper), el propósito central de este proyecto de grado es comprender en profundidad el modelo LQ-GNN e implementarlo de forma completa y reproducible, incluyendo el pipeline de datos, entrenamiento y evaluación experimental. El foco está puesto en asegurar fidelidad con el modelo de referencia, documentar decisiones prácticas de implementación y analizar empíricamente su desempeño en distintos datasets, de manera comparable y auditable.

Adicionalmente, este trabajo se inscribe en un proyecto de investigación más amplio, financiado por ANII, orientado a mecanismos de escalado multidimensional para microservicios en entornos distribuidos. En ese marco, las decisiones

de control pueden involucrar re ubicación, replicación y asignación dinámica de recursos. En este sentido, participé de instancias de discusión y seguimiento del proyecto en reuniones periódicas del equipo ; el desarrollo de una implementación funcional de LQ-GNN constituye un paso relevante para habilitar, en etapas posteriores, su integración como componente de soporte a decisiones dentro de dicho marco.

En resumen, este proyecto se centra en la predicción de latencias en sistemas de microservicios mediante redes neuronales de grafos heterogéneos, implementando el modelo LQ-GNN. A diferencia de trabajos cuyo objetivo es proponer un nuevo método, el propósito central de este proyecto es comprender en profundidad el modelo propuesto en el trabajo de referencia e implementarlo de forma completa incluyendo su pipeline de datos, entrenamiento y evaluación experimental, con resultados reproducibles y comparables.

1.2. Problema de investigación

El problema central abordado por este trabajo puede formularse como:

> *¿Cómo implementar de forma fiel el modelo LQ-GNN propuesto en la literatura y evaluar empíricamente su capacidad para predecir la latencia end-to-end en aplicaciones basadas en microservicios bajo diferentes configuraciones, garantizando la reproducibilidad de los resultados?*

Resolver este problema implica, entre otros aspectos: (i) comprender a fondo la representación del sistema como grafo heterogéneo y reconstruirla a partir de los datos disponibles, (ii) implementar el esquema de *message passing* por etapas y las funciones de agregación definidas por el modelo, (iii) construir un *pipeline* de entrenamiento y evaluación (preprocesamiento, partición, normalización, métricas), y (iv) validar el desempeño del modelo en distintos datasets, analizando generalización y sensibilidad a decisiones prácticas de implementación.

1.3. Objetivos

1.3.1. Objetivo general

Implementar y evaluar el modelo **LQ-GNN** para la predicción de latencias en aplicaciones basadas en microservicios, creando una implementación que permita su reutilización y validando empíricamente su desempeño mediante experimentación controlada sobre múltiples datasets.

1.3.2. Objetivos específicos

1. Revisar y sintetizar los conceptos necesarios sobre arquitecturas de microservicios, métricas de latencia y fundamentos de aprendizaje profundo en grafos, con énfasis en grafos heterogéneos.

2. Comprender y reconstruir la representación del problema utilizada por LQ-GNN como un grafo heterogéneo (tipos de nodos, tipos de relaciones y atributos), de manera consistente con el modelo de referencia.

3. Implementar el modelo LQ-GNN de forma completa (capas, *message passing* por etapas, mecanismos de agregación/atención, y *readout*) junto con el *pipeline* de datos (carga, validación, preprocesamiento/normalización y partición entrenamiento/validación/prueba) utilizando herramientas actuales de aprendizaje automático.

4. Entrenar y evaluar el modelo mediante métricas estándar de regresión (MAE, RMSE, MAPE y MSLE), analizando convergencia, estabilidad del entrenamiento y capacidad de generalización a configuraciones no observadas.

5. Documentar el proceso de desarrollo y experimentación, incluyendo decisiones de implementación, dificultades prácticas, validaciones realizadas y análisis crítico de resultados.

1.4. Hipótesis

La hipótesis principal de este trabajo establece que, al implementar con fidelidad el enfoque LQ-GNN y su representación heterogénea, el modelo es capaz de aprender representaciones efectivas de la estructura y el estado de una aplicación basada en microservicios, lo que permite predecir latencias con un buen compromiso entre precisión y costo de inferencia en los *datasets* evaluados. En consecuencia, una implementación funcional de LQ-GNN constituye un componente adecuado para ser utilizado como predictor en flujos de evaluación *what-if* y, potencialmente, en módulos de soporte a la toma de decisiones de escalado.

1.5. Alcance y limitaciones

1.5.1. Alcance

El alcance del proyecto comprende:

- Implementación integral del modelo LQ-GNN propuesto en el trabajo de referencia, incluyendo su formulación en grafos heterogéneos, *message passing* por etapas y *readout* para la predicción.

- Entrenamiento y evaluación en múltiples datasets de aplicaciones basadas en microservicios, con el objetivo de analizar desempeño, estabilidad y generalización.

- Estudio del impacto de decisiones prácticas e hiperparámetros relevantes (dimensión oculta, número de iteraciones, función de pérdida, normalización, etc.).

- Generación de un artefacto reproducible (código, configuraciones y resultados) que permita replicar los experimentos de forma consistente, orientado a su uso como componente base en desarrollos posteriores.

1.5.2. Limitaciones

Las limitaciones principales consideradas en este trabajo incluyen:

- **Generalización**: los datasets disponibles representan aplicaciones y topologías particulares; la extrapolación a sistemas con características significativamente distintas nesesita una validación adicional.
- **Entorno de evaluación**: los experimentos se realizan usando datos históricos y escenarios preprocesados; no se evalúa el comportamiento del modelo en un entorno productivo (en producción) con fallas, interferencias y cambios abruptos en tiempo real.
- **Dinámica temporal**: el modelo utiliza información agregada/estática de cada instancia; no modela explícitamente series temporales largas ni efectos transitorios del control en línea.
- **Comparación con alternativas**: el foco del proyecto es la implementación y evaluación del enfoque de referencia; por lo tanto, la comparación con otros métodos no es abordada.
- **Interpretabilidad**: el modelo prioriza la capacidad predictiva y no produce explicaciones causales directas sobre la contribución de atributos o fuentes de degradación.

1.6. Organización del documento

El informe se organiza de la siguiente manera:

- **Capítulo 2**: presenta los antecedentes y fundamentos necesarios para comprender el problema, el uso de GNN y el modelo LQ-GNN.
- **Capítulo 3**: describe la metodología, el *pipeline* de datos y la implementación del modelo.
- **Capítulo 4**: presenta la experimentación, los resultados y el análisis.
- **Capítulo 5**: expone conclusiones, contribuciones y líneas de trabajo futuro.
- **Referencias y anexos**: incluyen bibliografía utilizada y material complementario relevante.

Capítulo 2

Revisión de Antecedentes

Este capítulo presenta los conceptos y antecedentes necesarios para comprender el trabajo desarrollado. Su objetivo es construir la base conceptual nesesaria para entender e implementar correctamente el modelo LQ-GNN.

En la práctica, esta sección no fue un simple "repaso teórico": para poder implementar un modelo como LQ-GNN fue necesario construir una comprensión sólida y progresivamente compleja. En particular, se requirió entender (i) cómo se comportan sistemas distribuidos basados en microservicios, (ii) qué significa medir y predecir performance en presencia de colas, dependencias y variabilidad, y (iii) cómo las Redes Neuronales de Grafos formalizan la idea de "propagar información" sobre estructuras relacionales.

Además, fue necesario recorrer el camino conceptual desde lo más básico: antes de poder comprender una GNN, primero se estudió qué es una red neuronal *simple* (perceptrón), cómo se extiende a redes feed-forward multicapa, cómo se entrenan mediante *backpropagation* y optimización, y por qué estas redes tradicionales fallan cuando la entrada no tiene estructura fija. Recién a partir de esa base fue posible entender con claridad qué aporta una GNN y, finalmente, cómo el artículo de LQ-GNN modela el problema y organiza su *message passing*.

Es importante aclarar que el diseño del modelo no se definió en este proyecto: la arquitectura y el enfoque provienen del trabajo de referencia (LQ-GNN). El foco de este proyecto consistió en comprender el modelo con el nivel de detalle necesario para implementarlo con fidelidad, adaptarlo al entorno de desarrollo elegido y resolver los desafíos prácticos asociados (representación del grafo heterogéneo, construcción de relaciones, y agregaciones/mecanismos de actualización no estándar).

2.1. Sistemas de Microservicios

La arquitectura de microservicios propone construir una aplicación como un conjunto de servicios pequeños, independientes y débilmente acoplados, que se comunican mediante mecanismos livianos (típicamente HTTP/REST o mensa-

jería). A diferencia de una aplicación monolítica, donde esta se despliega como una única unidad, en microservicios cada componente puede evolucionar con mayor autonomía: se despliega, escala y versiona de forma independiente (Newman, 2015; Fowler y Lewis, 2014).

Desde el punto de vista de ingeniería, este enfoque tiene beneficios claros: desacopla equipos, habilita el escalado y suele mejorar la mantenibilidad en sistemas grandes. Sin embargo, también desplaza la complejidad hacia el dominio distribuido: las llamadas entre servicios, la coordinación, los fallos parciales y la observabilidad pasan a ser parte central del problema (Newman, 2015).

2.1.1. Complejidad distribuida y observabilidad

En un sistema de microservicios, una única solicitud percibida por el usuario rara vez se resuelve dentro de un solo componente: típicamente atraviesa una cadena de servicios que colaboran entre sí para construir la respuesta. Esta descomposición funcional introduce dependencias explícitas (por ejemplo, cuando un servicio invoca a otro de forma sincrónica y bloqueante) y también dependencias implícitas, menos visibles, asociadas a la contención de recursos compartidos y al comportamiento de colas. En particular, el desempeño observado en un servicio puede degradarse por efectos inducidos por servicios *aguas abajo* (*downstream*) que lo fuerzan a esperar, acumular solicitudes o ejecutar reintentos, generando efectos en cascada sobre la latencia end-to-end (Richart y cols., 2025; Dean y Barroso, 2013a).

Una consecuencia directa de esta dinámica es la aparición de **fallos parciales** y degradaciones graduales: un servicio puede mantenerse "en pie" desde el punto de vista de disponibilidad, pero responder más lento, producir timeouts intermitentes o generar reintentos que amplifican el tráfico y provocan efectos en cascada. Desde la perspectiva del usuario, el síntoma se manifiesta como aumento de latencia end-to-end; sin embargo, la causa puede estar varios saltos más adelante en el grafo de dependencias. Esto vuelve insuficiente una visión local del sistema y dificulta identificar qué componente explica un cambio de performance.

Por este motivo, la **observabilidad** se vuelve un requisito central en microservicios. Más allá de registrar métricas aisladas, es necesario instrumentar el sistema con métricas, logs y trazas distribuidas que permitan reconstruir el recorrido completo de una solicitud y descomponer la latencia total en contribuciones parciales. La trazabilidad distribuida, en particular, permite asociar eventos entre servicios y estimar cuánto tiempo se invierte en cada tramo (cómputo, espera, red), lo cual resulta clave tanto para el diagnóstico como para construir representaciones de datos que preserven la estructura del sistema. En el contexto de este proyecto, esta complejidad distribuida es precisamente una de las razones por las que representar explícitamente dependencias mediante grafos (en lugar de aplanar el sistema en un vector fijo) es importante para aproximar la latencia end-to-end.

2.1.2. Métricas de performance: latencia y percentiles

La latencia se define como el tiempo transcurrido entre el inicio de una operación y su finalización. En aplicaciones basadas en microservicios, esta métrica adquiere múltiples componentes que se combinan a lo largo del camino de ejecución. Por un lado, existe latencia de procesamiento dentro de cada servicio (cómputo y posibles esperas internas), y por otro lado aparece la latencia asociada a la comunicación entre servicios, que incluye transmisión, encolado en red y potenciales *overheads* de serialización y deserialización. A esto se suma un factor especialmente relevante en sistemas concurrentes: la latencia de encolamiento, es decir, el tiempo que una solicitud permanece esperando antes de ser atendida debido a límites de concurrencia (*threads*, *workers*, conexiones, *pools*) o a la saturación en etapas previas.

La métrica más importante desde el punto de vista del usuario es la latencia end-to-end, que corresponde al tiempo total desde que la solicitud ingresa al sistema hasta que se obtiene la respuesta final. En microservicios, esta latencia no se explica únicamente como una suma de promedios: su comportamiento está dominado por variabilidad (los cambios en la carga y tiempos de procesamiento a lo largo del tiempo), contención (la competencia entre solicitudes por recursos compartidos, como CPU o hilos) y colas, y depende fuertemente de la estructura de dependencias (qué servicios participan y en qué orden) y del estado instantáneo del sistema.

Por último, es importante destacar que en escenarios reales, reportar únicamente el promedio de la latencia puede no ser suficiente para determinar el rendimiento del sistema. La experiencia del usuario final está frecuentemente determinada por la *tail latency* (por ejemplo el percentil 95 o percentil 99 de la latencia), donde se manifiestan fenómenos que el promedio tiende a ocultar: esperas transitorias, congestión, *timeouts* y reintentos. Este punto es especi9almente crítico en microservicios porque basta con que un único componente presente demoras en un instante dado para que la *tail latency* se degrade significativamente, aun cuando el resto del sistema se mantenga estable (Dean y Barroso, 2013b). Por lo tanto, el análisis por percentiles no es solo una elección de reporte, sino una forma de capturar el comportamiento relevante en presencia de variabilidad y dependencias distribuidas, lo cual resulta consistente con el objetivo de construir predictores que reflejen esa complejidad.

2.1.3. Por qué predecir latencia end-to-end es difícil

Predecir la latencia end-to-end en sistemas de microservicios es desafiante porque el comportamiento observado no surge como una suma independiente de tiempos por componente, sino como el resultado de **interacciones** entre servicios, mecanismos de concurrencia y condiciones de ejecución que cambian en el tiempo. En otras palabras, la latencia de un servicio no es una propiedad fija: depende del estado local del servicio, del estado de los servicios de los que depende y de cómo el tráfico se distribuye por distintas rutas de ejecución. (Dean y Barroso, 2013b).

En primer lugar, la carga de trabajo suele ser variable y su distribución no es uniforme. Una aplicación puede ofrecer múltiples puntos de entrada (*endpoints*) o tipos de solicitud (*requests*) que disparan **rutas de ejecución** (*paths*) distintas; además, una misma solicitud puede recorrer subrutas diferentes (por ejemplo, según un *cache hit/miss*). Como consecuencia, la presión efectiva sobre cada microservicio no depende solamente de las "solicitudes por segundo" globales, sino de cómo ese tráfico se descompone y se concentra en regiones específicas de la topología, pudiendo generar cuellos de botella localizados.

En segundo lugar, las llamadas sincrónicas introducen **bloqueos** y **posesión concurrente de recursos**. Cuando un servicio realiza una llamada bloqueante a otro, mantiene recursos ocupados (por ejemplo, *threads*, conexiones o estructuras internas) mientras espera la respuesta. En este contexto, denominamos *servicio aguas abajo* (*downstream*) al servicio invocado, y *servicio aguas arriba* (*upstream*) al que inicia la llamada y depende de su respuesta. Este patrón acopla el desempeño entre componentes: una degradación en un servicio aguas abajo puede propagarse aguas arriba como aumento de colas, aun si el servicio aguas arriba no está saturado en CPU. Este tipo de acoplamiento rompe supuestos de independencia y hace que métricas locales (por ejemplo, la utilización de CPU) no siempre expliquen la latencia *end-to-end*.

En tercer lugar, pequeñas modificaciones locales de configuración pueden provocar efectos globales no obvios. Por ejemplo, aumentar *threads* o réplicas en un microservicio puede cambiar la tasa a la que se envían solicitudes a servicios vecinos, redistribuir carga, desplazar el cuello de botella y modificar la forma de las colas en diferentes puntos del sistema. En consecuencia, la relación entre "recursos asignados" y "latencia resultante" suele ser altamente no lineal y dependiente del contexto.

Finalmente, la comunicación en red agrega heterogeneidad adicional: la latencia entre servicios varía según ubicación (edge/fog/cloud), congestión y capacidad de enlace. En arquitecturas distribuidas en un continuum, dos configuraciones con recursos similares pueden exhibir latencias end-to-end muy diferentes únicamente por cambios en el posicionamiento relativo de los servicios y en sus costos de comunicación.

En conjunto, estas características motivan enfoques de modelado que preserven explícitamente la estructura del sistema y sus dependencias, en lugar de representar la aplicación como un vector de características de tamaño fijo. Para el objetivo de este proyecto, esto justifica el uso de representaciones basadas en grafos, donde la topología y las relaciones entre entidades forman parte del *input* y pueden ser explotadas por el modelo predictivo.

2.2. Redes Neuronales: fundamentos necesarios

Antes de introducir redes neuronales de grafos, fue necesario consolidar los fundamentos del aprendizaje automático, en particular el funcionamiento de redes *feed-forward* para regresión, el entrenamiento mediante descenso por gradiente y el rol de la función de pérdida. En un problema de predicción supervi-

sada se dispone de un conjunto de ejemplos $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, donde $\mathbf{x}_i$ representa el vector de atributos de entrada y y_i el valor objetivo. El propósito del modelo es aprender una función paramétrica $f_\theta(\cdot)$ que aproxime la relación entre entradas y salidas, generando predicciones $\hat{y}_i = f_\theta(\mathbf{x}_i)$ para instancias no observadas. En este trabajo, la salida corresponde a una magnitud continua asociada a *performance* (latencia end-to-end), cuya relación con las variables de entrada suele ser altamente no lineal debido a efectos de contención, encolamiento, variabilidad de carga y dependencias entre componentes. (Goodfellow, Bengio, y Courville, 2016)

2.2.1. Redes *feed-forward* y representaciones internas

En su formulación más simple, una red neuronal multicapa (MLP) construye una representación interna de los datos mediante una composición de transformaciones lineales y no lineales. Dada una entrada $\mathbf{h}^{(0)} = \mathbf{x}$, una capa l produce:

$$\mathbf{h}^{(l)} = f^{(l)}\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right),$$

donde $\mathbf{W}^{(l)}$ y $\mathbf{b}^{(l)}$ son parámetros entrenables y $f^{(l)}(\cdot)$ es una función de activación no lineal (por ejemplo ReLU, LeakyReLU o GELU). La no linealidad es esencial: sin ella, la composición de capas colapsaría a una única transformación lineal, incapaz de capturar patrones complejos. En modelos de regresión, la última capa suele ser lineal (sin activación) para permitir valores reales sin restricciones, salvo que se desee imponer positividad u otras propiedades (por ejemplo mediante *softplus*).

Desde el punto de vista conceptual, las capas intermedias aprenden representaciones latentes que reexpresan la información de entrada en un espacio donde la tarea de predicción resulta más simple. En problemas de *performance*, esto equivale a aprender combinaciones no lineales de señales que, individualmente, pueden ser poco informativas (por ejemplo métricas locales), pero que en conjunto capturan efectos emergentes del sistema (por ejemplo saturación, desplazamiento de cuellos de botella y propagación de demoras).

2.2.2. Función de pérdida como objetivo de aprendizaje

El entrenamiento de la red consiste en ajustar $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}$ minimizando una función de pérdida $\mathcal{L}(y, \hat{y})$ que cuantifica el desacuerdo entre el valor real y y la predicción $\hat{y}$. En términos empíricos, se minimiza el riesgo empírico sobre el conjunto de entrenamiento:

$$\min_{\theta} \ \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(y_i, f_\theta(\mathbf{x}_i)).$$

Para tareas de regresión, opciones típicas incluyen MSE (error cuadrático medio), MAE (error absoluto medio) o pérdidas robustas como Huber. Más allá de su interpretación estadística, la pérdida determina la señal de entrenamiento: es

el criterio que guía la actualización de parámetros y define qué tipos de errores se penalizan más fuertemente. Esto es particularmente relevante en latencias, donde la distribución del objetivo suele presentar asimetría y colas pesadas: errores grandes pueden corresponder a episodios raros (picos) pero operativamente importantes.

2.2.3. Descenso por gradiente y *backpropagation*

Para minimizar la pérdida se utiliza descenso por gradiente (o variantes estocásticas). Dado un batch $\mathcal{B}$, se calcula el gradiente $\nabla_\theta \mathcal{L}$ mediante *backpropagation*, aplicando la regla de la cadena a lo largo de la composición de capas. Intuitivamente, el *forward* produce predicciones y la pérdida; el *backward* propaga "responsabilidad" hacia atrás indicando cuánto contribuye cada parámetro al error. Luego se actualizan los parámetros:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L},$$

donde η es la tasa de aprendizaje. En la práctica se utiliza mini-batch gradient descent: el gradiente se estima sobre subconjuntos de datos, lo cual reduce costo computacional y suele introducir ruido beneficioso que actúa como regularización implícita.

En este contexto, las redes neuronales profundas han demostrado ser una herramienta efectiva para modelar relaciones complejas y no lineales en distintos dominios. Estas arquitecturas, compuestas por múltiples capas de transformación, permiten aprender representaciones jerárquicas de los datos, capturando tanto patrones locales como globales. Esta capacidad de aprendizaje resulta especialmente valiosa en sistemas donde el comportamiento emerge de la interacción entre múltiples componentes, como ocurre en aplicaciones basadas en microservicios.

En redes profundas, la dinámica de entrenamiento está influida por múltiples factores: la escala de los datos, la elección de activaciones, la inicialización de pesos y la magnitud de gradientes. Problemas como *exploding gradients* (es una técnica para limitar el valor de los gradientes durante el entrenamiento y evitar problemas como *exploding gradients* —son gradientes excesivamente grandes que desestabilizan el aprendizaje— o *vanishing gradients* — son gradientes muy pequeños que dificultan la actualización de los parámetros) pueden comprometer la convergencia, por lo que resulta habitual incorporar técnicas complementarias como normalización (sobre entradas o representaciones) y *gradient clipping* (es la técnica que limita los gradientes para evitar valores demasiado grandes o demasiado pequeños que afecten el entrenamiento) en caso de inestabilidades. En este proyecto, la estabilidad numérica fue un aspecto central debido a la combinación de un objetivo con gran rango dinámico —es decir, latencias que pueden variar significativamente entre valores bajos y altos— y un modelo con múltiples módulos de propagación.

2.2.4. Optimizadores adaptativos y regularización

Si bien el descenso por gradiente simple es conceptualmente suficiente, en redes modernas suele preferirse el uso de optimizadores adaptativos. *Adam* (Kingma y Ba, 2015) mantiene estimaciones móviles del primer y segundo momento del gradiente y adapta el tamaño del paso por parámetro, lo cual resulta especialmente útil cuando distintos componentes del modelo presentan escalas de gradiente muy dispares. Esto es frecuente en arquitecturas compuestas (por ejemplo, embeddings + capas de propagación + readout), donde diferentes módulos aprenden a ritmos distintos.

En este trabajo se utilizó *AdamW* (Loshchilov y Hutter, 2019), que separa explícitamente la regularización por *weight decay* del término de gradiente. Esta separación evita que el efecto del *decay* quede "mezclado" con la adaptación del learning rate de Adam, y suele mejorar la generalización en la práctica. El *weight decay* puede interpretarse como una penalización a la magnitud de los pesos, incentivando soluciones más simples y reduciendo sobreajuste, particularmente cuando el modelo tiene capacidad suficiente para memorizar patrones espurios del conjunto de entrenamiento.

2.2.5. Pérdidas robustas y particularidades de latencias

En problemas de regresión de latencia es común observar: (i) distribuciones fuertemente asimétricas, (ii) presencia de valores extremos asociados a colas y episodios de alta variabilidad, y (iii) un rango amplio de valores que puede diferir significativamente entre topologías o escenarios. En este contexto, la elección de la pérdida impacta directamente sobre la estabilidad del entrenamiento.

La pérdida cuadrática (MSE) penaliza de manera proporcional al cuadrado del error, lo que puede hacer que un número pequeño de outliers domine la señal de gradiente y desplace el aprendizaje hacia esos casos extremos, deteriorando el ajuste global. Por su parte, MAE reduce la sensibilidad a outliers, pero puede producir gradientes menos informativos cerca del óptimo y dificultar convergencia suave. Por este motivo, en este trabajo se utilizó la *Huber loss* (Huber, 1964), definida (para un umbral δ) como:

$$\mathcal{L}_\delta(e) = \begin{cases} \frac{1}{2}e^2, & \text{si } |e| \le \delta, \\ \delta\left(|e| - \frac{1}{2}\delta\right), & \text{si } |e| > \delta, \end{cases}$$

donde $e = y - \hat{y}$. Esta función se comporta como MSE para errores pequeños (favoreciendo una convergencia precisa alrededor del mínimo) y como MAE para errores grandes (reduciendo la influencia de outliers). En la práctica, este compromiso resulta adecuado cuando se busca estabilidad y robustez en presencia de colas pesadas (siendo estas distribuciones con alta probabilidad de valores extremos, donde eventos poco frecuentes pero muy grandes influyen significativamente en el comportamiento), sin perder precisión en el régimen "típico" del sistema.

Finalmente, además de la elección de la pérdida, la escala numérica del objetivo influye fuertemente en la estabilidad del aprendizaje. En latencias, dife-

rencias de orden de magnitud pueden inducir gradientes dominados por rangos altos, dificultando aprender patrones en rangos bajos. Por ello, en el pipeline experimental de este trabajo se adoptó una transformación logarítmica del objetivo durante el entrenamiento (según se detalla en el capítulo metodológico), con el fin de comprimir el rango dinámico, suavizar la influencia de valores extremos y facilitar una optimización más estable.

2.3. Métricas de evaluación para regresión

La evaluación de modelos de predicción de latencia requiere métricas que capturen distintos aspectos del error. En este trabajo se reportan cuatro métricas complementarias: MAE, RMSE, MAPE y MSLE. Sean y_i los valores reales y $\hat{y}_i$ las predicciones para N instancias. El *Mean Absolute Error* (MAE) se define como:

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i| \,,$$

y cuantifica el error medio en unidades de la variable objetivo. En contraste, el *Root Mean Squared Error* (RMSE) se define como:

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2},$$

y penaliza con mayor fuerza los errores grandes, por lo que resulta útil cuando interesa reflejar el impacto de outliers o predicciones muy desviadas.

Para comparar desempeño cuando el rango de latencias varía entre escenarios, se utiliza el *Mean Absolute Percentage Error* (MAPE):

$$\mathrm{MAPE} = \frac{100}{N}\sum_{i=1}^{N} \left| \frac{y_i - \hat{y}_i}{y_i} \right| ,$$

que expresa el error en términos relativos (porcentaje). Esta métrica facilita comparar conjuntos con escalas distintas, aunque requiere cuidado cuando existen valores de y_i muy cercanos a cero.

Finalmente, se reporta el *Mean Squared Logarithmic Error* (MSLE):

$$\mathrm{MSLE} = \frac{1}{N}\sum_{i=1}^{N} \left(\log(1 + y_i) - \log(1 + \hat{y}_i)\right)^2 ,$$

que mide discrepancia en el espacio logarítmico y tiende a priorizar errores relativos, siendo especialmente útil cuando la distribución del objetivo presenta colas pesadas. Dado que las latencias en sistemas distribuidos suelen exhibir alta asimetría, MSLE y MAPE aportan una perspectiva complementaria a MAE/RMSE. En conjunto, estas métricas permiten caracterizar desempeño tanto en error absoluto (interpretación directa en tiempo) como en error relativo (comparabilidad entre rangos).

2.4. Redes Neuronales de Grafos (GNN)

En el contexto de este trabajo, el límite conceptual que empuja hacia Redes Neuronales de Grafos (GNN) es que las redes neuronales tradicionales asumen entradas de estructura fija. Si representamos un sistema distribuido "aplanándolo", se pierde información estructural: quién depende de quién, qué rutas existen, y por dónde se propagan demoras. En microservicios, esa estructura *es* parte del fenómeno que queremos predecir.

Las Redes Neuronales de Grafos son modelos de aprendizaje diseñados para operar directamente sobre estructuras relacionales, preservando explícitamente la información de **nodos**, **aristas** y **atributos** asociados. Su motivación central es que, en muchos dominios, el comportamiento que se desea predecir depende tanto de las características individuales de cada entidad como de *cómo* esas entidades se relacionan entre sí. Este punto es clave para el contexto de este trabajo: en una arquitectura de microservicios, las dependencias de llamadas y los rutas de ejecución condicionan de forma directa la latencia, por lo que el grafo no es una representación incidental sino parte del fenómeno a modelar.

En términos generales, las GNN siguen el paradigma de **message passing**, donde cada nodo mantiene un estado latente (embedding) que se actualiza iterativamente al intercambiar información con sus vecinos (Scarselli y cols., 2009; Gilmer y cols., 2017). Tras varias iteraciones, cada nodo incorpora información de un vecindario cada vez mayor, lo que permite capturar dependencias indirectas (por ejemplo, efectos aguas abajo en una cadena de llamadas).

2.4.1. *Message passing*: idea y formalización

En una formulación típica, en la iteración t cada nodo v recibe mensajes desde sus vecinos $u \in \mathcal{N}(v)$ y los combina en una representación agregada:

$$\mathbf{m}_v^{(t)} = \mathrm{AGG}\Big(\{\phi(\mathbf{h}_v^{(t)}, \mathbf{h}_u^{(t)}, \mathbf{e}_{uv}) : u \in \mathcal{N}(v)\}\Big),$$

$$\mathbf{h}_v^{(t+1)} = \psi(\mathbf{h}_v^{(t)}, \mathbf{m}_v^{(t)}),$$

donde $\mathbf{h}_v^{(t)}$ es el *embedding* del nodo v en la iteración t, $\mathbf{e}_{uv}$ son atributos (opcionales) de la arista (u, v), ϕ es una función que define cómo se construye un mensaje, AGG es una función de agregación y ψ es una función que actualiza el estado del nodo. Un aspecto fundamental es que AGG debe ser *invariante al orden* (por ejemplo, suma, promedio o máximo), ya que un grafo no tiene un orden natural de vecinos. Esta propiedad es crítica para que el modelo sea consistente frente a permutaciones en la representación interna de las aristas.

En términos intuitivos, este esquema implementa un mecanismo de "propagación" de información: cada nodo resume su contexto local, y ese resumen se va refinando iteración a iteración. A su vez, el número de iteraciones controla el alcance: luego de k iteraciones, un nodo puede incorporar información de nodos a distancia k en el grafo. En problemas donde la influencia se propaga a través de cadenas de dependencia, esto permite capturar efectos indirectos relevantes; al

mismo tiempo, demasiadas iteraciones pueden introducir dificultades prácticas como *over-smoothing* (*embeddings* que se vuelven demasiado similares) o degradación del gradiente en modelos profundos, por lo que el número de iteraciones suele elegirse como hiperparámetro.

Arquitecturas como GCN (Kipf y Welling, 2017) y GAT (Veličkovi'c y cols., 2018) pueden entenderse como instancias particulares de este esquema general de *message passing*. En ambos casos, el objetivo es combinar información de los vecinos para actualizar el embedding de cada nodo, pero difieren en *cómo* definen la construcción del mensaje y su agregación.

En GCN, la actualización se implementa mediante una agregación **normalizada** de las representaciones vecinas (incluyendo típicamente auto-conexiones), lo que equivale a una operación análoga a una "convolución" sobre el grafo. La normalización (por ejemplo, basada en el grado) evita que nodos con vecindarios grandes dominen la magnitud de los *embeddings* y contribuye a estabilizar el entrenamiento. En términos del esquema general, puede interpretarse como un caso donde ϕ aplica una transformación lineal compartida a los vecinos, AGG realiza una suma/promedio ponderado por normalización, y ψ aplica una no linealidad para producir el nuevo estado del nodo.

En GAT, en cambio, la agregación incorpora un mecanismo de **atención** que aprende pesos distintos para cada vecino, permitiendo ponderar de manera adaptativa qué conexiones son más relevantes para la actualización. Esto resulta útil cuando la contribución de los vecinos no es uniforme: el modelo puede asignar mayor importancia a ciertos nodos o relaciones según el contexto de la instancia. Bajo el formalismo anterior, ϕ produce mensajes transformados, AGG computa una combinación ponderada por coeficientes de atención aprendidos y ψ integra el resultado para actualizar el *embedding*.

En general, existen múltiples elecciones razonables para estas funciones: (i) AGG suele ser una operación invariante al orden como suma, promedio o máximo; (ii) ϕ puede ser una transformación lineal o una MLP aplicada al *embedding* del vecino (posiblemente incorporando atributos de arista); y (iii) ψ suele implementarse como una MLP o una celda recurrente (por ejemplo, GRU) que combina el estado previo del nodo con la información agregada. Estas variantes comparten el mismo principio: la predicción final se apoya en representaciones aprendidas que incorporan progresivamente información estructural del grafo.

2.4.2. De representaciones locales a predicciones globales

Además de tareas a nivel de nodo (por ejemplo, clasificación de nodos), muchas aplicaciones requieren predecir propiedades del *grafo completo*. En este caso, el objetivo del modelo no es producir una etiqueta por entidad, sino obtener un valor global, como la latencia end-to-end de una consulta. Para ello, se utiliza un **readout** o pooling global que resume el conjunto de embeddings de nodos en una representación única del grafo:

$$\mathbf{h}_G = \mathrm{POOL}(\{\mathbf{h}_v\}_{v \in V}),$$

y luego una **red neuronal multicapa** (*Multi-Layer Perceptron*, MLP) transforma $\mathbf{h}_G$ en la predicción. Una MLP es una red *feed-forward* compuesta por una secuencia de capas lineales intercaladas con funciones de activación no lineales, que actúa como un aproximador paramétrico capaz de mapear la representación global $\mathbf{h}_G$ a una salida de interés (por ejemplo, una predicción escalar de latencia). La función POOL debe ser invariante a permutaciones por la misma razón que la agregación local: el conjunto de nodos no tiene un orden canónico. Las opciones más comunes incluyen suma, promedio o máximo, aunque también pueden emplearse mecanismos de atención global para aprender qué nodos son más relevantes para la predicción.

Esta separación "propagación local + pooling global" es especialmente adecuada en problemas donde existe una interacción entre fenómenos locales y métricas globales. En el caso de microservicios, la latencia end-to-end está compuesta por contribuciones locales (servicios y actividades) que, al combinarse a través de rutas y dependencias, generan un comportamiento global. El readout permite que el modelo aprenda cómo sintetizar esa información distribuida en una señal agregada útil.

2.5. Predicción de performance en sistemas distribuidos

Históricamente, la predicción de performance se ha abordado mediante: (i) modelos analíticos (teoría de colas), (ii) simulación, y (iii) modelos basados en datos.

Los **modelos analíticos** son interpretables, pero requieren supuestos fuertes para ser tratables. En sistemas con llamadas sincrónicas y dependencia entre capas, técnicas como *Layered Queueing Networks* (LQN) capturan la idea de "colas por capas" y la posesión concurrente de recursos (Franks, Al-Omari, Woodside, Das, y Derisavi, 2009). Sin embargo, su realismo depende de qué tan bien representen el comportamiento del sistema, ya que suelen apoyarse en hipótesis simplificadoras sobre el tráfico. Por ejemplo, el paper de LQ-GNN (Richart y cols., 2025) señala que este tipo de enfoques suele asumir llegadas de solicitudes según un proceso de Poisson, lo que muchas veces no modela adecuadamente escenarios reales de aplicaciones basadas en microservicios.

La **simulación** puede ser muy precisa, pero su costo computacional suele ser alto, lo que dificulta evaluar muchas configuraciones rápidamente. En microservicios, se han propuesto simuladores como µqSim (Zhang, Gan, y Delimitrou, 2019) para estudiar performance a escala.

Finalmente, los enfoques de **aprendizaje automático** buscan aprender directamente a partir de datos, reduciendo la necesidad de realizar supuestos analíticos sobre el comportamiento del sistema. En el estado del arte aparecen modelos como Sinan (Zhang, Hua, Zhou, Suh, y Delimitrou, 2021) y enfoques basados en grafos como GRAF (Park, Choi, Lee, y Han, 2021) y PERT-GNN (Tam, Liu, Xu, Xie, y Lau, 2023), que explotan explícitamente la estructura de

dependencias entre microservicios. Sin embargo, una debilidad común de muchos de estos enfoques es su limitada capacidad de generalización, ya que los modelos aprendidos suelen estar fuertemente ligados a la aplicación específica utilizada durante el entrenamiento.

2.6. LQ-GNN: predicción de latencia con grafos heterogéneos inspirados en LQN

Esta sección resume el modelo *LQ-GNN: A Graph Neural Network Model for Latency Prediction of Microservice-based Applications for Elasticity Control in the Computing Continuum* (Richart y cols., 2025) y su formulación para predicción de latencia en aplicaciones basadas en microservicios. El objetivo de este Proyecto de Grado no es proponer una arquitectura nueva, sino comprender el diseño existente e implementar el modelo de forma fiel y funcional. Para ello, fue necesario asimilar en detalle tanto la representación del sistema como el flujo de información que el modelo utiliza para producir una predicción de latencia end-to-end.

A diferencia de enfoques que "aplanan" el sistema en un vector de características, LQ-GNN parte de la premisa de que la *estructura* del sistema (dependencias entre servicios, composición de operaciones y rutas de ejecución) es parte esencial del fenómeno a predecir. Por lo tanto, el modelo construye una representación explícita del sistema como un grafo heterogéneo y aplica un esquema de *message passing* por etapas, iterado múltiples veces, para capturar cómo el estado de recursos, la carga y las dependencias se reflejan en la latencia observada.

2.6.1. Representación del sistema

LQ-GNN modela una aplicación de microservicios como un grafo heterogéneo, donde distintos tipos de nodos y relaciones representan entidades con semánticas diferentes dentro de la ejecución de una consulta. Esta elección es clave: la latencia end-to-end no depende únicamente de "qué microservicios existen", sino también de *cómo* se ejecutan internamente las solicitudes y *por dónde* circulan dentro del sistema.

En términos generales, el grafo utiliza tres tipos principales de nodos:

1. **Task** Representa un microservicio (o unidad de despliegue). Sus atributos reflejan aspectos de capacidad y configuración, por ejemplo parámetros vinculados a concurrencia, recursos asignados, réplicas y métricas agregadas de carga o utilización. La intención es que el nodo `task` capture el "estado operativo" del microservicio, que condiciona los tiempos de respuesta que ese microservicio puede entregar bajo determinada carga.

2. **Activity** Representa pasos internos de procesamiento dentro de una tarea. Esta capa intermedia permite modelar que un mismo microservicio

puede ejecutar diferentes operaciones con costos distintos, y que las dependencias sincrónicas y el bloqueo pueden aparecer a nivel de operación. Por eso, las `activities` incluyen características relacionadas con demoras de procesamiento y con el tipo de interacción (por ejemplo, llamadas que pueden bloquear la atención de nuevas solicitudes).

3. **Path** Representa rutas de ejecución relevantes (end-to-end o subrutas) compuestas por secuencias de actividades. Este nodo explicita que la carga no se distribuye uniformemente: distintas rutas pueden tener intensidades de tráfico diferentes y, por lo tanto, inducir presiones distintas sobre las actividades y tareas involucradas. El nodo `path` incorpora atributos de carga que permiten contextualizar la predicción en el escenario operativo considerado.

La estructura del grafo conecta estas entidades mediante relaciones tipadas que reflejan pertenencia (actividades contenidas en tareas), composición (actividades que integran un camino) y dependencias entre actividades. Esto permite representar, de forma explícita, el hecho de que el estado de una actividad depende tanto del microservicio que la ejecuta como del flujo end-to-end en el que participa, y que la latencia global emerge como resultado de múltiples dependencias interconectadas.

2.6.2. Message passing por etapas

Una vez construido el grafo heterogéneo, el modelo aplica un proceso iterativo de actualización de estados latentes (embeddings) mediante propagación de mensajes por etapas. La motivación es que existe una dependencia circular entre las entidades: los caminos dependen del comportamiento de las actividades que contienen, las actividades están condicionadas por la carga de los caminos que las atraviesan, y las tareas resumen el comportamiento conjunto de sus actividades. Para capturar ese acoplamiento, el modelo alterna actualizaciones de `path`, `activity` y `task` durante múltiples iteraciones.

El proceso comienza proyectando los atributos originales de cada tipo de nodo a un espacio latente común de dimensión fija. A partir de ahí, cada iteración aplica tres actualizaciones conceptualmente distintas:

1. **Actualización de paths desde activities.** Cada `path` agrega información proveniente de las actividades que lo componen para construir una representación del flujo end-to-end bajo el estado actual del sistema. Dado que un camino puede verse como una secuencia de pasos, la actualización está diseñada para capturar dependencias a lo largo del recorrido y no solo una combinación "en bolsa" de actividades.

2. **Actualización de activities desde múltiples fuentes.** Cada `activity` combina mensajes provenientes de (i) su `task` asociada, (ii) los `paths` que

la incluyen y (iii) actividades vecinas que modelan dependencias de ejecución. Esta etapa es particularmente relevante porque concentra la integración entre capacidad local (estado del servicio), presión global (carga por rutas) y estructura de dependencias. Además, el modelo pondera contribuciones de mensajes según su relevancia, evitando asumir que todos los vecinos influyen por igual en todos los escenarios.

3. **Actualización de tasks desde activities.** Cada `task` agrega información de sus actividades internas para obtener una representación del microservicio coherente con el contexto de carga y dependencias vigente. De esta forma, el estado de una tarea no es una descripción estática, sino una representación aprendida que refleja cómo sus operaciones internas se ven afectadas por el resto del sistema.

Tras completar T iteraciones, el modelo aplica un mecanismo de *readout* para transformar representaciones latentes en predicciones. Conceptualmente, esto permite producir una estimación final que resume el estado del sistema y su estructura de dependencias, conservando la trazabilidad de "por dónde" se propagó la información en el grafo a lo largo de las etapas de actualización. En este Proyecto de Grado, internalizar este flujo de información fue determinante para implementar el modelo con fidelidad: aquí la GNN no es solo "una red aplicada a un grafo", sino un procedimiento iterativo que refleja cómo el desempeño se acopla y se compone entre entidades con semánticas distintas.

2.6.3. Grafos heterogéneos

En muchos problemas reales, no todos los nodos representan la misma clase de entidad ni todas las aristas representan el mismo tipo de relación. En grafos heterogéneos existen múltiples tipos de nodos y/o múltiples tipos de aristas. Esto exige que el *message passing* respete la semántica de cada relación: no es razonable aplicar la misma transformación a un mensaje "microservicio $\rightarrow$ actividad" que a un mensaje "actividad $\rightarrow$ path", porque codifican dependencias diferentes.

Una forma común de manejar heterogeneidad es definir transformaciones específicas por tipo de relación (*relation-specific*), o bien funciones de actualización específicas por tipo de nodo, combinando luego los mensajes provenientes de diferentes relaciones mediante agregaciones o mecanismos de atención. En este proyecto, esta capacidad es central porque el modelo LQ-GNN representa entidades distintas (`task`, `activity` y `path`) con roles y atributos diferentes, y su interacción (por etapas) es parte explícita del diseño propuesto en el paper. En consecuencia, el grafo heterogéneo no es un "detalle de implementación" sino una decisión estructural que habilita representar el problema con mayor fidelidad.

2.6.4. Implementación práctica: TensorFlow vs PyTorch y elección de PyTorch Geometric

Además de comprender los fundamentos teóricos de las GNN, una parte relevante del trabajo fue seleccionar un stack tecnológico que permitiera *implementar* el modelo definido en el paper con fidelidad al diseño y con un flujo de desarrollo que facilitara iteración y depuración. En particular, al trabajar con grafos heterogéneos y con message passing por etapas, los errores frecuentes suelen ser sutiles (dimensiones incompatibles, relaciones mal indexadas, inconsistencias entre tipos de nodos), por lo que la experiencia práctica del framework tiene impacto directo en la velocidad de desarrollo.

En este proyecto se consideraron principalmente dos ecosistemas: TensorFlow (y sus librerías para grafos) y PyTorch (en particular PyTorch Geometric). Aunque ambos permiten entrenar redes neuronales profundas, en la práctica difieren en el estilo de programación, en el nivel de "ergonomía" para grafos heterogéneos y en la facilidad para implementar capas no estándar.

Criterios prácticos de elección

La elección del framework no se basó únicamente en rendimiento o popularidad, sino en criterios directamente vinculados al tipo de modelo a implementar:

- **Soporte natural para grafos heterogéneos**: LQ-GNN opera sobre múltiples tipos de nodos y relaciones (por ejemplo `task`, `activity`, `path` y sus vínculos).
- **Capacidad de implementar message passing no estándar**: el paper propone una actualización por etapas (paths → activities → tasks), y en particular una combinación de mensajes en `activity` que no coincide con una capa "clásica" lista para usar.
- **Facilidad de depuración e iteración**: al prototipar y ajustar el modelo, resulta crucial inspeccionar tensores intermedios y verificar rápidamente shapes y flujos de datos.

PyTorch Geometric y `HeteroData`

PyTorch Geometric (PyG) ofrece una representación directa para grafos heterogéneos mediante `HeteroData`, que permite almacenar datos separados por tipo de nodo y tipo de arista (atributos, índices de aristas y atributos por relación). Esta estructura calza de forma particularmente natural con LQ-GNN, ya que cada bloque conceptual del modelo (`task`, `activity`, `path`) se representa como un tipo explícito y las relaciones entre ellos se mantienen diferenciadas (*torch_geometric.data.HeteroData*, 2025).

En términos prácticos, esto facilita:

- construir el grafo como un conjunto de componentes tipados, evitando ambigüedades;

- implementar capas por relación (o por subconjunto de relaciones) de manera modular;
- depurar validando dimensiones y consistencia por tipo, lo que reduce el costo de errores.

Agregación personalizada: por qué PyTorch se volvió la opción más cómoda

Durante la implementación, una dificultad central fue que la agregación propuesta en el paper no coincide con agregaciones estándar (*sum/mean/max*) aplicadas de forma uniforme. En particular, la actualización de `activity` recibe mensajes de tres fuentes conceptualmente distintas (la tarea a la que pertenece, los paths que la atraviesan y actividades vecinas) que luego deben combinarse, en el diseño original, mediante un mecanismo de atención.

En PyTorch Geometric, la clase base `MessagePassing` está diseñada para construir capas personalizadas, permitiendo definir la lógica de mensajes, la agregación y la actualización de manera explícita (PyTorch Geometric, 2025). Esto fue clave para replicar la lógica del paper sin tener que "forzar" el modelo a encajar en una capa predefinida.

Además, PyTorch resultó práctico para iterar porque el forward se expresa como una secuencia clara de operaciones sobre tensores, lo que facilita inspeccionar estados intermedios y validar hipótesis durante la depuración. Esta característica fue especialmente útil en un proyecto centrado en implementación, donde el principal desafío no era proponer una nueva arquitectura, sino llevar a código (y hacer funcionar) una arquitectura ya definida, respetando su comportamiento.

TensorFlow (y TensorFlow GNN) como alternativa considerada

TensorFlow cuenta con soporte para GNN mediante TensorFlow GNN, que representa grafos con esquema heterogéneo a través de `GraphTensor` y herramientas asociadas (Google LLC, 2024). Sin embargo, para este proyecto el factor decisivo no fue la posibilidad de representar grafos heterogéneos, sino la combinación de rapidez de iteración, facilidad para implementar message passing particular y una experiencia de depuración más directa durante el proceso de implementación.

Decisión adoptada

En síntesis, para este proyecto se optó por **PyTorch junto con PyTorch Geometric**, ya que esta combinación ofreció el mejor equilibrio entre soporte para grafos heterogéneos, flexibilidad para implementar mecanismos de *message passing* no estándar y facilidad de depuración durante el desarrollo. Esta elección resultó especialmente adecuada para reproducir con fidelidad la arquitectura de LQ-GNN y construir un pipeline experimental controlable y reproducible.

En particular, esta elección condicionó de forma directa la implementación posterior del modelo, ya que permitió representar explícitamente las entidades `task`, `activity` y `path`, así como sus relaciones tipadas, y facilitó la construcción de capas personalizadas alineadas con el esquema de propagación definido en LQ-GNN.

Capítulo 3

Metodología e Implementación

Este capítulo describe, de forma técnica y detallada, la metodología seguida y la implementación del modelo LQ-GNN utilizada en este proyecto. Es importante aclarar el encuadre: la arquitectura del modelo no se diseñó desde cero en este trabajo, sino que se implementó un modelo ya propuesto en la literatura, reproduciendo su lógica y adaptándola a un stack moderno (PyTorch/PyTorch Geometric) y a un *pipeline* reproducible de datos/entrenamiento. Por lo tanto, las decisiones documentadas en este capítulo se concentran en cómo se llevó esa arquitectura a código, cómo se representaron los datos para que el modelo pudiera operar y qué criterios se utilizaron para entrenar y evaluar de forma consistente.

Repositorio del código fuente. El código fuente desarrollado para este proyecto se encuentra disponible en el repositorio: `https://gitlab.fing.edu.uy/elasticon/gnn`. Allí se incluye la implementación del modelo, los scripts de entrenamiento y evaluación, los archivos de configuración utilizados y las utilidades necesarias para procesar los datasets. Los resultados experimentales reportados en este informe se corresponden con la versión del código disponible en la rama `main` del repositorio.

3.1. Metodología

3.1.1. Enfoque de trabajo

El proyecto se desarrolló con una metodología que combina desarrollo de software y validación experimental. La meta fue obtener un artefacto funcional: una implementación completa del modelo LQ-GNN capaz de entrenar, validar y evaluar predicciones de latencia sobre grafos heterogéneos derivados de aplicaciones basadas en microservicios.

En la práctica, el trabajo se organizó en ciclos iterativos ("implementación → verificación → experimento → corrección"). Este enfoque fue necesario porque, en modelos de grafos heterogéneos, los errores más frecuentes no son únicamente de optimización (hiperparámetros), sino de la implementación de la representación (qué significa cada tipo de nodo, cómo se conectan, cómo se agrupan en batches), de consistencia (alineación de índices, dimensiones y normalización) y, sobre todo, de correcciones del funcionamiento de las agregaciones.

3.2. Modelado del problema

3.2.1. Definición del grafo heterogéneo

La aplicación se representa como un grafo heterogéneo $G = (V, E)$ con partición disjunta de nodos:

$$V = V_{\text{task}} \cup V_{\text{activity}} \cup V_{\text{path}}.$$

Cada subconjunto corresponde a una entidad con una semántica distinta: microservicios, operaciones internas y rutas de ejecución. Las aristas E se separan por tipo de relación, representando dependencias entre entidades (pertenencia, flujo, composición y vecindarios operacionales).

Este modelado es coherente con el objetivo de predicción de latencia end-to-end: la latencia global emerge como composición de actividades a lo largo de *paths*, y esa composición depende del estado de *tasks* (recursos, carga, concurrencia) y de interacciones locales entre actividades.

3.2.2. Variable objetivo

El objetivo del modelo es predecir una métrica global asociada al grafo que representa una instancia de ejecución/configuración de la aplicación. En este trabajo, la variable objetivo y corresponde al tiempo de respuesta end-to-end promedio del sistema, ademas de el percentil 95 o y el percentil 99 (dependiendo cual es usada para entrenar) y se almacenan en el dataset en milisegundos, sin transformaciones. Durante el entrenamiento, el objetivo se transforma mediante la función $\log(1 + y)$ únicamente para el cálculo de la pérdida, manteniendo la unidad original. Para el reporte de resultados, las predicciones se retransforman a la escala original y se convierten a segundos dividiendo por 1000, con el fin de facilitar su interpretación.

En el dataset, cada instancia (grafo) está asociada a resultados de simulación que incluyen estadísticas de desempeño. En particular, para cada escenario se dispone de medidas del tiempo de respuesta (por ejemplo, promedio y percentiles altos) y otras señales agregadas del experimento. Siguiendo el foco principal del modelo y del caso reportado en este informe, se adopta como objetivo primario la **latencia promedio end-to-end**, ya que captura el comportamiento esperado del sistema bajo la configuración dada y permite comparar directamente con la referencia.

Operativamente, el modelo recibe como entrada el grafo heterogéneo (tipos de nodos, *atributos* y relaciones) y produce una **predicción escalar** $\hat{y}$ por instancia. Esta predicción se interpreta como una estimación del tiempo de respuesta global del sistema para el escenario representado por el grafo. Dado que distintas topologías y configuraciones pueden inducir distribuciones de latencia muy diferentes, la definición del objetivo como métrica global (y no a nivel de nodo) es consistente con el uso esperado del predictor en análisis *what-if* y soporte a decisiones: comparar rápidamente escenarios alternativos y estimar su impacto en la latencia.

3.3. Datos y preparación

3.3.1. Dataset y estructura general

El dataset utilizado en este trabajo representa instancias de sistemas de microservicios como grafos dirigidos en formato JSON. Cada instancia reúne tres tipos de información. En primer lugar, incluye metadatos a nivel de grafo, entre ellos una marca de validez (`valid`), el escenario de despliegue (`deployment`) y métricas globales de latencia end-to-end provistas por el simulador, como la latencia promedio (`total_avg_rt`), el percentil 95 (`total_p95_rt`) y el percentil 99 (`total_p99_rt`).

En segundo lugar, cada instancia contiene nodos tipados de tres clases: `task`, `activity` y `path`. Cada uno de estos tipos cumple un rol distinto dentro de la representación del sistema y, por lo tanto, posee atributos específicos. Los nodos `task` incluyen variables de configuración y carga del microservicio, como `num_threads`, `num_replicas`, `num_cores`, `cpu_util` y `num_requests`, además de métricas de latencia a nivel de microservicio (`ms_avg_rt`, `ms_p95_rt`, `ms_p99_rt`). Los nodos `activity` contienen atributos como `proc_delay` y `blocking`, mientras que los nodos `path` incluyen la carga del camino (`load`) y métricas de latencia asociadas a ese recorrido (`avg_rt`, `p95_rt`, `p99_rt`).

Por último, cada instancia incorpora enlaces dirigidos entre entidades. Estos enlaces codifican tanto la pertenencia entre tareas y actividades como las relaciones de precedencia entre actividades y la asociación entre caminos y las actividades que los componen.

La Tabla 3.1 resume los principales componentes del dataset y los atributos más relevantes de cada uno. En este trabajo, las variables objetivo a nivel de grafo corresponden a la latencia end-to-end del sistema, en particular la latencia promedio (`total_avg_rt`) y el percentil 95 (`total_p95_rt`), ambas expresadas en milisegundos (ms).

A nivel operativo, el dataset se utiliza dividido en subconjuntos de entrenamiento, validación y test, de forma de estimar el desempeño fuera de muestra y controlar el sobreajuste durante el entrenamiento.

Tabla 3.1: Resumen de la estructura del dataset y de sus atributos principales.

Componente	**Descripción y atributos principales**
Metadatos del grafo	Marca de validez (`valid`), escenario de despliegue (`deployment`) y métricas globales de latencia end-to-end: `total_avg_rt`, `total_p95_rt`, `total_p99_rt`.
Nodos `task`	Representan microservicios. Incluyen atributos de configuración y carga como `num_threads`, `num_replicas`, `num_cores`, `cpu_util`, `num_requests`, además de métricas de latencia a nivel de microservicio: `ms_avg_rt`, `ms_p95_rt`, `ms_p99_rt`.
Nodos `activity`	Representan operaciones internas dentro de una tarea. Incluyen atributos como `proc_delay` y `blocking`.
Nodos `path`	Representan caminos de ejecución. Incluyen la carga (`load`) y métricas de latencia por camino: `avg_rt`, `p95_rt`, `p99_rt`.
Enlaces dirigidos	Codifican la pertenencia entre tareas y actividades, las relaciones de precedencia entre actividades y la asociación entre caminos y las actividades que los componen.

3.3.2. Conversión a grafo heterogéneo (HeteroData)

Para poder entrenar LQ-GNN con PyTorch Geometric, cada JSON se transforma a una instancia de `HeteroData`. Esta conversión es uno de los puntos más importantes del pipeline, porque:

- define explícitamente qué es cada entidad (tipo de nodo) y cómo se conecta (tipo de arista);
- determina el formato en el que el modelo recibirá atributos, índices de aristas y (cuando aplica) atributos asociados a aristas;
- condiciona la posibilidad de aplicar message passing por relaciones, tal como requiere un grafo heterogéneo.

En la implementación, se construyen tres matrices de atributos iniciales:

- **Task** (`task.x`): vector con atributos de configuración/carga del microservicio.
- **Activity** (`activity.x`): vector con atributos de delay y un indicador de bloqueo.
- **Path** (`path.x`): vector con la carga del path.

Luego, se generan estructuras `edge_index` por tipo de relación. El modelo considera, como mínimo, relaciones entre:

- `activity` → `path` (con un atributo `label` asociado),
- `path` → `activity`,
- `task` → `activity`,
- `activity` → `task`,
- `activity` → `activity` (con un atributo `in_out` asociado).

Esta separación por relaciones es esencial para respetar la semántica heterogénea del problema: no todas las aristas representan lo mismo, ni deberían mezclarse bajo una única regla de agregación.

3.3.3. Normalización y transformaciones de escala

En modelos de aprendizaje, la escala numérica de los atributos y de la variable objetivo puede dominar la estabilidad del entrenamiento y la magnitud de los gradientes. Por este motivo, durante el desarrollo se implementó soporte experimental para distintas estrategias de normalización, seleccionables mediante un único parámetro de configuración.

Se consideran tres modos: *ninguna* (`none`), *logaritmo más uno* (`log1p`) y *estandarización z-score* (`zscore`). En el modo `none`, tanto los atributos de los nodos como la variable objetivo se mantienen en su escala original (tras una conversión de unidades opcional). En `log1p` se aplica la transformación $\log(1 + x)$ a valores no negativos, reduciendo el rango de magnitudes y atenuando el efecto de valores muy grandes. En `zscore`, cada variable se transforma según $(x - \mu)/(\sigma + \varepsilon)$, donde μ y σ son la media y la desviación estándar estimadas únicamente sobre el conjunto de entrenamiento, y ε es una constante pequeña fija que evita divisiones por cero cuando σ es nula.

Las estadísticas (μ, σ) se calculan una sola vez sobre el conjunto de entrenamiento para cada tipo de atributo de nodo (*task*, *activity*, *path*) y para la variable objetivo cuando aplica. Esas mismas estadísticas se aplican después a los conjuntos de entrenamiento, validación y prueba, de modo que no haya filtración de información desde conjuntos no vistos. La variable objetivo (por ejemplo, latencia P95) puede expresarse en milisegundos en los datos; se realiza una conversión interna a una unidad común para que las métricas sean comparables entre conjuntos. Cuando se usa z-score en la salida, la predicción puede desnormalizarse antes de calcular errores en la escala original para reportar métricas interpretables.

En resumen, la normalización es opcional, coherente con la práctica de ajustar transformaciones solo con datos de entrenamiento y permite comparar de forma controlada el efecto de distintas escalas (sin normalizar, logarítmica o estandarizada) sobre la estabilidad y el rendimiento del modelo.

3.4. Arquitectura implementada

Esta sección describe una de las partes mas importantes de la implementación: la arquitectura del modelo LQ-GNN. Constituye el aporte central de este trabajo —reproducir con fidelidad el diseño del modelo sobre un stack moderno y controlable— por lo que se presenta con el detalle necesario para entender qué hace cada bloque y cómo se encadenan. Aquí se detallan la proyección inicial de los atributos de cada tipo de nodo (embeddings), el esquema de *message passing* por etapas que actualiza en cada iteración los estados de `path`, `activity` y `task`, los mecanismos de agregación y actualización (GRU y atención) que distinguen al modelo de una GNN estándar, el *readout* que produce la predicción escalar de latencia, y las decisiones de regularización (dropout) adoptadas para estabilizar el entrenamiento. El resto del capítulo —datos, stack tecnológico y protocolo de entrenamiento— está al servicio de alimentar, ejecutar y evaluar esta arquitectura.

3.4.1. Vista general del flujo

El flujo del modelo puede resumirse en cuatro bloques encadenados. En primer lugar, los atributos crudos de cada tipo de nodo (`task`, `activity`, `path`) se proyectan a un espacio común de dimensión fija mediante capas lineales (embeddings iniciales). A continuación, se ejecutan T iteraciones de *message passing* por etapas: en cada iteración se actualizan en orden los estados de `path` (a partir de las actividades que componen cada camino), luego los de `activity` (integrando mensajes de la tarea, los paths y las actividades vecinas) y por último los de `task` (agregando la información de sus actividades). Los estados resultantes de cada iteración son la entrada de la siguiente; de este modo, la información se propaga a lo largo de las T etapas sin producir aún una predicción escalar. Tras las T iteraciones, se aplica un pooling global sobre los nodos `task` para obtener una representación del grafo, que una red *readout* transforma en la predicción escalar de latencia $\hat{y}$. Esta salida $\hat{y}$ se obtiene una sola vez por forward. Durante el entrenamiento se compara con el objetivo para calcular la pérdida y actualizar los parámetros del modelo; cada nueva época repite el mismo flujo sobre los datos con los parámetros ya actualizados (véase el capítulo de entrenamiento y evaluación).

3.4.2. Embeddings iniciales

El primer paso del modelo consiste en proyectar los atributos crudos de cada tipo de nodo a un espacio latente común de dimensión fija `hidden_dim`. Esta decisión responde a dos necesidades: (i) cada tipo de nodo tiene distinta cantidad y significado de atributos (por ejemplo, `task` incluye variables de configuración y carga del microservicio, `activity` incluye demora y un indicador de bloqueo, y `path` incluye la carga del camino), por lo que no puede asumirse una representación única; (ii) el *message passing* posterior opera sobre vectores de dimensión

uniforme, de modo que las transformaciones y agregaciones por relación estén bien definidas.

En la implementación se utilizan tres capas lineales independientes, una por tipo de nodo. Cada capa mapea el vector de atributos crudos del tipo correspondiente a un vector de dimensión `hidden_dim`. Así se obtienen los estados iniciales $h_k^{(0)}$ para nodos `task`, $h_a^{(0)}$ para `activity` y $h_p^{(0)}$ para `path`, que constituyen la entrada del primer paso de *message passing*. No se comparten pesos entre tipos: la proyección de cada entidad es específica, de modo que se preserve la semántica distinta de cada una antes de mezclar información en las etapas siguientes. En la implementación, las dimensiones de entrada son 5 para `task`, 2 para `activity` y 1 para `path`, coherentes con el número de atributos por tipo de nodo definidos en el dataset.

3.4.3. Message passing por etapas e iteraciones

La arquitectura LQ-GNN se implementa como un procedimiento iterativo de T iteraciones (en el código, `num_layers`), donde en cada iteración se ejecuta una secuencia fija de tres etapas ordenadas de actualización. La motivación es capturar una dependencia circular natural del sistema: los `paths` dependen de las `activities` que recorren; las `activities` se ven afectadas tanto por el contexto del `path` (carga, recorrido) como por el estado de su `task` (capacidad) y por dependencias con otras `activities`; y finalmente cada `task` resume el comportamiento agregado de sus `activities`. Los estados resultantes de cada iteración son la entrada de la siguiente, de modo que tras T pasadas cada entidad incorpora información propagada desde vecindarios cada vez más amplios en el grafo.

Estados latentes y relaciones utilizadas

Luego de embebir los atributos crudos en un espacio común, el modelo mantiene tres conjuntos de estados: $h_k^{(t)}$ para nodos `task`, $h_a^{(t)}$ para nodos `activity` y $h_p^{(t)}$ para nodos `path`. En cada iteración del `forward` se utilizan explícitamente las siguientes relaciones tipadas, representadas como `edge_index` (y atributos de arista cuando corresponden) en PyG:

- `activity` → `path`: composición del camino a partir de actividades, con un atributo de arista (`label`) utilizado para mantener consistencia con el orden/rol de la actividad dentro del path.
- `path` → `activity`: retroalimentación desde paths hacia actividades (contexto de carga y flujo).
- `task` → `activity`: pertenencia/contención (una actividad está asociada a una tarea).
- `activity` → `task`: agregación desde actividades hacia la tarea.

- `activity` → `activity`: dependencias operacionales entre actividades, con atributos de arista (por ejemplo, un indicador direccional `in_out` en la implementación).

Etapa 1: actualización de `path` desde `activity`

En la primera etapa de la iteración t, se actualiza $h_p^{(t)}$ utilizando mensajes provenientes de las actividades conectadas a cada path. Conceptualmente, el objetivo es construir una representación del flujo end-to-end que refleje tanto los costos internos de las actividades como su composición dentro del camino. En la implementación, esta actualización se realiza mediante un módulo específico (`PathUpdateLayer`) que recibe el estado actual de los paths, los estados de las activities y las aristas `activity` → `path`, además de las etiquetas (`label`) cuando están disponibles. La agregación no es una suma o promedio indiferenciado: los mensajes se ordenan según el destino y el `label` (orden de la actividad en el path) y se procesan con una GRU, de modo que se respete la estructura secuencial del recorrido.

Etapa 2: actualización de `activity` integrando múltiples fuentes

La segunda etapa es la más rica del modelo: actualiza $h_a^{(t)}$ integrando información desde tres fuentes principales: (i) el estado de la tarea que contiene a la actividad (capacidad/configuración), (ii) el estado de los paths que atraviesan la actividad (presión de carga y contexto end-to-end), y (iii) actividades vecinas (dependencias de ejecución y acoplamientos internos).

En la implementación, esta integración se concentra en `ActivityUpdateLayer`, que recibe explícitamente $h_a^{(t)}$, $h_k^{(t)}$, $h_p^{(t)}$ junto con las relaciones `path` → `activity`, `task` → `activity` y `activity` → `activity`, incluyendo atributos de arista cuando aplican. A nivel conceptual, esta etapa busca capturar el efecto de "propagación" de degradaciones: una actividad puede volverse lenta no solo por su costo local, sino porque un path con alta carga o una dependencia aguas abajo afecta su comportamiento.

Etapa 3: actualización de `task` desde `activity`

En la tercera etapa, cada `task` se actualiza agregando información de sus actividades internas mediante la relación `activity` → `task`. Esta actualización (en `TaskUpdateLayer`) produce un estado $h_k^{(t+1)}$ que resume el comportamiento del microservicio en el contexto actual del sistema, incorporando indirectamente información de paths y dependencias a través de las actividades.

Por qué repetir T iteraciones

El ciclo completo (paths → activities → tasks) se repite T veces para permitir que la información se propague a través de dependencias indirectas. Con pocas iteraciones, cada entidad solo incorpora contexto local inmediato; con

más iteraciones, una actividad (o una task) puede capturar efectos inducidos por rutas y dependencias más alejadas en el grafo. Esto es relevante en microservicios, donde los cuellos de botella y la contención se manifiestan como efectos en cascada y no necesariamente como un fenómeno estrictamente local.

Salida de las iteraciones y lectura global

Tras las T iteraciones se obtienen los estados finales $h_k^{(T)}$, $h_a^{(T)}$, $h_p^{(T)}$. En este punto aún no existe una predicción escalar: la lectura global consiste en aplicar un pooling global sobre uno de los conjuntos de estados (en la implementación, *mean pooling* sobre los nodos `task` mediante `global_mean_pool`) para obtener un vector h_G que representa el grafo completo. Ese vector es la entrada del bloque de *readout* (subsección 3.4.5), que lo transforma en la predicción escalar $\hat{y}$. La separación entre (i) message passing iterativo y (ii) pooling + readout refuerza que primero se construyen estados contextualizados por entidad, y después el readout aprende a traducir la representación agregada del grafo en tiempo de respuesta.

3.4.4. Mecanismos de actualización: GRU y atención

La implementación replica los mecanismos centrales propuestos por LQ-GNN:

- **Agregación ordenada con GRU** en las etapas donde el modelo trata una colección de mensajes como una secuencia (por ejemplo, para resumir contribuciones de múltiples actividades a un path, o de múltiples actividades a una task). En este caso, la GRU funciona como un agregador que puede mantener memoria y capturar patrones no lineales en la combinación de mensajes.
- **Atención sobre vecinos** en la actualización de activities, donde confluyen mensajes desde múltiples fuentes (tasks, paths y actividades vecinas). La atención permite ponderar qué mensajes son más relevantes para el estado actual, lo cual es consistente con la idea de "camino crítico" o "vecinos más influyentes" en términos de latencia.

En la práctica, estas decisiones no se tratan como elecciones arbitrarias de arquitectura dentro del proyecto, sino como parte del comportamiento del modelo que se buscó reproducir con fidelidad.

3.4.5. Readout y predicción global

Luego de completar las T iteraciones, el modelo obtiene estados finales para cada entidad. Para producir una predicción única por grafo (latencia end-to-end), se aplica un esquema de pooling global sobre los nodos `task` para obtener una representación h_G del grafo completo (como se describió en el apartado anterior). El *readout* es una red feed-forward que transforma h_G en la salida

escalar $\hat{y}$. En la implementación, el módulo de readout (`ReadoutPathHead`) está formado por una MLP de dos capas ocultas de dimensión 32 con activación ReLU y capas de dropout entre ellas, terminando en una salida lineal de dimensión 1. Esta estructura permite que el modelo aprenda una función no lineal desde la representación global del grafo hasta el valor de latencia, al tiempo que el dropout (detallado en la siguiente subsección) reduce el sobreajuste en esta etapa final.

Esta separación entre (i) propagación local iterativa y (ii) lectura global final permite que el modelo capture dependencias internas del sistema antes de condensarlas en una predicción de latencia.

3.4.6. Regularización: dropout

Para estabilizar el entrenamiento y reducir el sobreajuste, se aplica *dropout* en el bloque de readout. En la implementación, el parámetro `dropout` (por defecto 0,15) se utiliza en las capas intermedias de la MLP del readout: tras cada capa oculta y antes de la siguiente transformación lineal. De este modo, durante el entrenamiento una fracción de las unidades se desactiva aleatoriamente en cada forward, lo que actúa como regularización y evita que el readout dependa de un subconjunto reducido de dimensiones de h_G. En un modelo con múltiples iteraciones de message passing y una representación global que condensa toda la información del grafo, regularizar el readout resulta especialmente relevante para mejorar la generalización fuera de muestra.

3.4.7. Organización del código y módulos del proyecto

El pipeline de implementación se estructura en módulos con responsabilidades bien delimitadas, de modo que datos, modelo, entrenamiento y utilidades puedan mantenerse y modificarse de forma independiente. A continuación se resume el rol de cada componente principal; el repositorio de código permite inspeccionar los detalles de implementación.

- `config`: centraliza la configuración del experimento: hiperparámetros de entrenamiento (`num_epochs`, `batch_size`, `learning_rate`, `hidden_dim`, `num_layers`, `dropout`, `weight_decay`, `early_stopping_patience`), rutas a los directorios de datos (train, validación, test), modo de normalización (`normalization_mode`, `zscore_eps`), unidad del objetivo en los JSON (`target_unit_in_json`), semilla para reproducibilidad y rutas de salida (modelo y logs). Los argumentos de línea de comandos pueden sobrescribir estos valores sin modificar el archivo.

- `dataset`: se encarga de la carga y conversión de datos. Expone compute_feature_stats(train_path) para calcular, sobre el conjunto de entrenamiento, las estadísticas de normalización (media y desviación estándar por tipo de nodo) que se aplican luego a todos los grafos; `convert_to_hetero_data(graph_json)` para transformar un grafo en formato JSON en una instancia `HeteroData`

de PyG (atributos por tipo de nodo, `edge_index` por tipo de relación, aplicación de normalización cuando corresponde); y `create_dataloader(path, batch_size, shuffle)` para construir el `DataLoader` que entrega batches de grafos heterogéneos al bucle de entrenamiento.

- `train`: orquesta el flujo de entrenamiento y evaluación. En `main()` se cargan las estadísticas de normalización desde train, se crean los tres `DataLoader` (train, validación, test), se instancia el modelo LQGNN y el optimizador (AdamW) con la función de pérdida (Huber) y el scheduler (ReduceLROnPlateau). El bucle por épocas ejecuta `train_epoch` (forward, pérdida, backprop, actualización de pesos), `validate` (evaluación sin gradientes y cálculo de métricas en escala original) y actualiza el learning rate según la pérdida de validación. Se mantiene el mejor estado del modelo según `val_loss` y se aplica early stopping si no hay mejora durante `early_stopping_patience` épocas. Al finalizar, se carga el mejor checkpoint, se evalúa sobre test y se guardan los pesos en disco.

- `utils`: proporciona normalización y desnormalización de atributos y del objetivo (`FEATURE_STATS`, `set_norm_stats`, `normalize_node_features`, `normalize_features`, `denormalize_features`) y el cálculo de las métricas de evaluación (MAE, RMSE, MAPE, MSLE) sobre predicciones y etiquetas en escala original.

- `models/`: contiene la arquitectura del modelo. `lqgnn.py` define la clase `LQGNN` (`nn.Module`) que en el `forward` aplica los embeddings por tipo de nodo, el bucle de T iteraciones llamando a las capas de actualización, el pooling global sobre `task` y el readout. `message_passing.py` implementa `PathUpdateLayer`, `ActivityUpdateLayer` y `TaskUpdateLayer`, que realizan las tres etapas de message passing (agregación con GRU donde corresponde, atención sobre mensajes en `ActivityUpdateLayer`). `layers.py` define los bloques auxiliares: `StateUpdateGRU`, `MessageAttention`, `ReadoutPathHead` y la MLP para atributos de arista en la relación activity–activity (`ActivityToActivityMLP`).

Esta separación permite modificar un componente (por ejemplo, la normalización o el criterio de early stopping) sin alterar el resto del pipeline, y facilita la trazabilidad y la reproducción de los experimentos.

Capítulo 4

Experimentación

Este capítulo presenta la configuración experimental utilizada, la evolución del proceso de entrenamiento y los resultados obtenidos al evaluar la implementación de LQ-GNN sobre distintos datasets. Además, se discute la relación entre los resultados logrados en este trabajo y los reportados por los autores del modelo, cuya implementación original se realizó sobre el framework *Ignnition* (Barcelona Neural Networking Center (BNN-UPC), 2026). En todos los casos, el objetivo de esta etapa experimental es doble: (i) verificar empíricamente que la implementación reproduce el comportamiento esperado del modelo y (ii) cuantificar su desempeño en escenarios variados.

4.1. Stack tecnológico

La implementación del modelo se realizó en Python utilizando PyTorch como framework principal de aprendizaje y PyTorch Geometric (PyG) como biblioteca especializada para el procesamiento de grafos. Esta elección se justifica por dos requerimientos concretos del problema: (i) la necesidad de operar sobre grafos heterogéneos con múltiples tipos de nodos y relaciones, y (ii) la necesidad de implementar *message passing* con reglas específicas (incluyendo agregaciones no estándar) de forma controlable y depurable.

PyTorch provee un modelo de ejecución imperativo que facilita inspeccionar tensores intermedios y depurar el flujo del `forward`, lo cual es especialmente útil cuando se trabaja con estructuras heterogéneas donde errores de índices, dimensiones o alineación de aristas pueden ser difíciles de detectar. Por su parte, PyTorch Geometric aporta una representación explícita para grafos heterogéneos mediante `HeteroData`, permitiendo almacenar atributos por tipo de nodo y `edge_index` por tipo de relación, lo que calza naturalmente con la estructura del problema (`task`, `activity`, `path` y relaciones entre ellos).

4.1.1. Entorno de ejecución y reproducibilidad

Los experimentos reportados en este informe se ejecutaron en ClusterUY, la plataforma de computación científica de alto desempeño del Centro Nacional de Supercomputación de Uruguay, utilizando un flujo de trabajo basado en SLURM (SchedMD, 2026). El entrenamiento del modelo se realizó con aceleración por GPU, solicitando un dispositivo `NVIDIA P100` mediante la directiva `-gres=gpu:p100:1`, con `4` CPUs asignadas al proceso, `64 GB` de memoria y un tiempo máximo de `8 horas` por corrida. Para reducir el overhead de E/S durante el entrenamiento, cada ejecución se copió a un directorio temporal en `/scratch`, desde donde se corrió el entrenamiento; al finalizar, los artefactos (logs y modelos) se sincronizaron de vuelta al directorio del proyecto en `$HOME`.

El entorno de software se gestionó mediante `conda`, activando el entorno `lqgnn` al inicio de cada job. El script de ejecución (Listing 6.1) establece además un modo estricto de bash (`set -euo pipefail`) para evitar ejecuciones parciales silenciosas ante errores, y registra información básica del nodo y del dispositivo visible para CUDA.

Entorno local (desarrollo). Además de las ejecuciones en el clúster, se utilizó un entorno local en PC para tareas de desarrollo, depuración y validaciones rápidas del pipeline. En este entorno local se verificaron versiones de dependencias dentro del entorno `conda` (`lqgnn`). En particular, se observó:

- Python: `3.11.13` (Anaconda, Windows).
- PyTorch: `2.4.0+cpu` (sin soporte CUDA en esa instalación).
- PyTorch Geometric: `2.5.3`.

Dado que PyTorch fue instalado en modalidad `+cpu` en el entorno local, la disponibilidad de CUDA en dicho equipo es `False`. Esto no afecta a los resultados reportados en el capítulo de experimentación, ya que el entrenamiento final fue ejecutado en el clúster con GPU.

Protocolo de ejecución en el clúster. Cada corrida se ejecutó como un job independiente con SLURM, utilizando el script del Listing 6.1. El procedimiento seguido en cada ejecución fue:

1. Activar el entorno `conda` (`lqgnn`).
2. Crear un directorio de ejecución en `/scratch/$USER` asociado al `job id`.
3. Copiar el proyecto (código + estructura de datos) a `/scratch` para minimizar latencias de E/S.
4. Ejecutar el entrenamiento (`train.py`) registrando logs y guardando el modelo entrenado.

5. Copiar resultados (logs y pesos del modelo) de vuelta a `$HOME` en una carpeta por job.

Este esquema permite aislar cada corrida, preservar artefactos de salida por `job id` y facilitar auditoría posterior de resultados.

Selección de modelos y replicación. Para evaluar el desempeño fuera de muestra, en cada dataset se seleccionó el *checkpoint* asociado a la menor pérdida de validación (*val loss*) durante el entrenamiento, y dicho modelo fue posteriormente evaluado sobre el conjunto de test. El pipeline registra los hiperparámetros relevantes y los resultados de cada corrida en archivos versionados por identificador de *job*, lo que permite replicar el protocolo experimental bajo el mismo entorno y particiones de datos.

Las ejecuciones en el clúster se realizaron mediante scripts de envío a SLURM que automatizan la activación del entorno, la copia del proyecto a `/scratch`, la ejecución del entrenamiento y la sincronización de resultados al directorio de trabajo permanente. Un ejemplo representativo del script utilizado se incluye en el Anexo 6.

Tiempo de ejecución. El tiempo de ejecución de los entrenamientos varió según el dataset utilizado y la configuración experimental considerada. En los entrenamientos realizados con la configuración base, las ejecuciones de mayor duración alcanzaron aproximadamente las 8 horas. Este valor dependió del dataset utilizado: el dataset *mix* fue el que presentó los mayores tiempos de entrenamiento, mientras que el dataset *4tier* fue el que demandó menos tiempo.

Por otro lado, al utilizar la configuración optimizada, los entrenamientos más extensos llegaron a demorar aproximadamente 16 horas. Estas ejecuciones se realizaron utilizando aceleración por GPU y procesamiento en paralelo dentro de cada *batch*. Este costo computacional fue uno de los factores que limitó la posibilidad de realizar una búsqueda más amplia de configuraciones óptimas, ya que cada nueva combinación de hiperparámetros implicaba un tiempo de entrenamiento considerable.

4.1.2. Implementación de referencia y limitaciones del framework original

Un aspecto práctico importante del proyecto fue que los autores del trabajo en el que se basa este informe contaban con una implementación previa del modelo en un framework de experimentación (utilizado como prototipo). Dicha implementación fue valiosa como referencia funcional para contrastar comportamientos generales, pero presentaba limitaciones relevantes: al operar dentro de una abstracción de alto nivel, no permitía controlar con precisión ciertos detalles del *message passing* ni reproducir algunas decisiones del modelo con fidelidad.

En particular, una de las restricciones más relevantes era la imposibilidad de implementar una agregación secuencial que respetara el orden de las actividades dentro de cada *path*. En LQ-GNN, la actualización de `path` se beneficia

de procesar los mensajes provenientes de `activity` mediante una GRU en un orden coherente con el recorrido del camino de ejecución. Esto es conceptualmente importante porque la latencia end-to-end no solo depende del conjunto de actividades involucradas, sino también de cómo se componen en un flujo; tratar esos mensajes como un conjunto sin orden (por ejemplo, mediante suma o promedio) pierde esa información secuencial.

La implementación desarrollada en este proyecto, basada en código PyTorch/PyTorch Geometric, permitió sortear estas limitaciones e implementar la agregación con GRU respetando el orden definido por el *path*, además de habilitar modificaciones finas necesarias para depuración y verificación. En consecuencia, la implementación previa se utilizó como punto de comparación cualitativo (sanity check), mientras que la versión desarrollada aquí se orientó a maximizar la fidelidad del comportamiento del modelo y la trazabilidad del pipeline completo.

4.1.3. Datasets y partición de datos.

Para la evaluación experimental se utilizaron datasets generados a partir de tres topologías de aplicaciones basadas en microservicios: una aplicación web de tipo *4-tier* y dos variantes de una aplicación *Social Network*. Además, se consideró un cuarto dataset denominado *mix*, construido a partir de instancias seleccionadas de las tres topologías anteriores, con el objetivo de evaluar la capacidad de generalización del modelo frente a diferentes estructuras de aplicación.

En cada caso, las muestras fueron generadas variando distintas características de configuración, como la cantidad de hilos, la cantidad de núcleos asignados, el despliegue de los microservicios y la carga de entrada de la aplicación. Luego del proceso de filtrado de simulaciones no válidas, cada topología quedó representada por aproximadamente 100.000 muestras. Estas muestras fueron divididas en tres subconjuntos: 80 % para entrenamiento, 10 % para validación y 10 % para test. El conjunto de entrenamiento se utilizó para ajustar los parámetros del modelo, el de validación para monitorear su desempeño durante el entrenamiento y el de test para realizar la evaluación final sobre instancias no vistas previamente.

4.2. Entrenamiento y evaluación

Se consideraron dos líneas de experimentación. La primera replica la configuración del paper: optimizador Adam (sin *weight decay*), función de pérdida error cuadrático medio (MSE), y el resto de hiperparámetros alineados con el trabajo de referencia; en esta línea se ejecutaron experimentos con variable objetivo igual al promedio de latencia y con el P95 (percentil 95), para comparar ambos blancos de predicción. La segunda línea explora una posible mejora del entrenamiento: se probaron optimización de hiperparámetros, funciones de pérdida más robustas frente a valores atípicos, como Huber, y el optimizador AdamW. A grandes rasgos, mientras que MSE penaliza cuadráticamente los errores gran-

des y por eso es más sensible a *outliers*, Huber combina un comportamiento cuadrático para errores pequeños con uno aproximadamente lineal para errores grandes, lo que suele volver el entrenamiento más estable. Por su parte, AdamW puede verse como una variante de Adam que incorpora regularización mediante *weight decay* desacoplado de la actualización adaptativa de los gradientes, favoreciendo una mejor capacidad de generalización. Junto con ello, se evaluaron dos esquemas de normalización—`zscore` (estandarización por media y desviación estándar) y `log1p` (transformación $\log(1 + y)$)—para analizar su impacto en el desempeño.

4.2.1. Métricas

La evaluación del desempeño se realiza con métricas de regresión complementarias: **MAE**, **RMSE**, **MAPE** y **MSLE**. Estas capturan perspectivas distintas del error: magnitud absoluta (MAE), penalización de valores extremos (RMSE), error relativo (MAPE) y sensibilidad en escala logarítmica (MSLE). Según el experimento, la variable objetivo puede ser el promedio de latencia o el P95; en todos los casos las predicciones y etiquetas se retransforman a la escala original mediante $\exp(\cdot) - 1$ cuando corresponde, y las métricas se reportan en segundos (MAE, RMSE) o en porcentaje (MAPE) para facilitar la interpretación.

4.2.2. Estructura del pipeline de entrenamiento

El sistema se organiza en módulos con responsabilidades claras:

- **Carga y conversión de datos**: lectura de JSON, conversión a `HeteroData`, construcción de `DataLoader` con batching eficiente.
- **Modelo**: definición de embeddings, bloques de message passing por etapas/iteraciones y readout final.
- **Entrenamiento/validación/test**: bucles separados para entrenamiento y evaluación, registro de pérdidas, guardado de modelo y exportación de resultados.
- **Utilidades**: normalización/desnormalización (z-score o log1p, según `normalization_mode`) y cálculo de métricas.

Este diseño modular mejora la mantenibilidad, y ademas facilita el trabajo experimental: permite alternar entre la configuración del paper (Adam, MSE, target promedio o P95) y las variantes con pérdida robusta y AdamW, así como entre normalizaciones z-score y log1p, sin reescribir todo el pipeline, y favorece la trazabilidad de resultados al mantener una separación explícita entre datos, modelo y entrenamiento.

4.3. Configuración experimental y Resultados

Los experimentos reportados se apoyan en un proceso iterativo previo de depuración del pipeline (representación, normalización, unidades, batching). En modelos con grafos heterogéneos y múltiples relaciones, los primeros resultados suelen estar dominados por errores de ingeniería más que por el ajuste fino de hiperparámetros. En esta etapa final se priorizó documentar corridas coherentes con una configuración base explícita y comparaciones claras entre objetivo de latencia promedio y percentil 95 (P95).

4.3.1. Evolución del proceso experimental

El trabajo no se redujo a un ajuste de hiperparámetros, sino a validar el pipeline completo (conversión de datos → `HeteroData` → entrenamiento → métricas). Se realizaron comprobaciones de coherencia numérica: pérdida que desciende, predicciones no degeneradas, rango de salidas plausible y correspondencia cualitativa entre objetivo y predicción.

Se prestó atención a las unidades en los registros de salida: en los archivos de log las unidades usadas fueron (`ms`) . Para el presente informe (para coherencia con el paper), las unidades de las metricas se unifican en segundos (s).

La distribución del objetivo en latencia suele ser asimétrica; según la versión del entrenamiento, pueden aplicarse transformaciones de escala en el objetivo o normalización de características (por ejemplo z-score). Las métricas en test se calculan tras invertir dichas transformaciones en la medida en que el código de evaluación lo implemente, de modo que MAE/RMSE reflejen error en la escala física acordada arriba.

4.3.2. Configuración Base

Configuración base (target promedio y target P95 no optimizado)

Las corridas con target promedio y las de target P95 con la misma configuración base utilizaron los siguientes hiperparámetros:

- **Semilla**: `seed` = 5234.
- **Modelo**: `hidden_dim` = 8, `num_layers` = 8.
- **Entrenamiento**: `batch_size` = 16, `learning_rate` = 10^{-3}, `num_epochs` = 50, optimizador Adam ($\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$), `weight_decay` = 0.
- **Normalización de entrada**: log(1+x) (`normalization_mode` = `log1p`) `detect_binary` = `True`.
- **Pérdida**: MSE (`loss_name` = `mse`).
- **Particiones**: `train`, `train_valid`, `test` según cada dataset.

El entrenamiento se ejecutó en GPU (CUDA). La única diferencia deliberada entre la configuracion con target `avg` y `p95` es el atributo objetivo en el JSON (promedio vs. P95), manteniendo el resto de la configuración y conjuntos de atributos alineados.

Datasets evaluados en esta etapa

Se reportan resultados para tres conjuntos, de modo que la comparación promedio vs. P95 sea directa:

- `GNN_mix`,
- `GNN_social-network`,
- `GNN_social-network-ut`.
- `GNN_4tier`.

Resultados : Target promedio y comparación con los resultados del paper

En las secciones previas ya se presentó el marco de LQ-GNN y el paper de referencia; aquí solo se recogen, de forma explícita, las cifras reportadas en el artículo para poder compararlas con las corridas de este trabajo. La Tabla 4.1 corresponde a la predicción del tiempo de respuesta promedio end-to-end de la aplicación completa (frente al simulador en el paper); la Tabla 4.3, al percentil 95. En ambas tablas del paper, MAE y RMSE están en segundos (s); MAPE y MSLE son adimensionales.

Tabla 4.1: Resultados reportados en el paper (Tabla I): predicción del tiempo de respuesta promedio de la aplicación completa, error respecto del simulador. MAE y RMSE en s.

Escenario (App)	**MAPE (%)**	**MAE (s)**	**MSLE**	**RMSE (s)**
App 1 (4-tier / topología 1)	6.9	0.239	0.0230	1.633
App 2 (Social Network, topología 2)	4.6	0.388	0.0118	2.191
App 3 (Social Network UT, topología 3)	4.7	0.383	0.0090	2.046
Mix (entrenamiento cruzado)	9.5	0.268	0.0908	2.934

En esa tabla, el escenario Mix del artículo concentra un MAPE de 9.5 % (promedio), frente a valores algo menores cuando el modelo se entrena por aplicación individual; en P95 (tabla mas adelante), MAE y RMSE son claramente mayores que en promedio, como es esperable al predecir la cola.

Resultados propios con target promedio y comparación con el paper

La Tabla 4.2 resume los resultados obtenidos en este trabajo al entrenar el modelo para predecir la latencia promedio end-to-end. Estas corridas corresponden a la línea experimental que replica la configuración base del paper, manteniendo la misma estructura general del modelo y los mismos tipos de atributos de entrada considerados en la propuesta original. En este sentido, la comparación resulta útil como referencia para analizar en qué medida la implementación desarrollada reproduce el comportamiento esperado del enfoque LQ-GNN.

Tabla 4.2: Evaluación en test con target promedio. MAE y RMSE en segundos (s).

Escenario (App)	**MAPE (%)**	**MAE (s)**	**MSLE**	**RMSE (s)**
App 1 (4-Tier)	10.85	0.485	0.0335	2.314
App 2 (Social Network)	21.81	1.594	0.0735	4.105
App 3 (Social Network-UT)	24.46	1.840	0.0926	4.763
Mix	33.91	1.837	0.1473	4.811

Comparación con los resultados reportados en el paper. La comparación con el trabajo original debe interpretarse con cautela. El paper reporta resultados obtenidos a partir de una implementación en *Ignnition*, utilizando su propio pipeline de generación de datos, sus particiones de entrenamiento/validación/test y un protocolo experimental específico basado en simulación. En cambio, este proyecto implementa LQ-GNN en un stack diferente, basado en PyTorch y PyTorch Geometric, con un pipeline propio de carga, conversión a grafos heterogéneos, entrenamiento y evaluación. Por lo tanto, no corresponde leer esta comparación como una equivalencia estricta "punto a punto", sino como una referencia para ubicar el comportamiento de la implementación desarrollada respecto del modelo original.

Aun con esa salvedad, la comparación resulta informativa. En el paper, el escenario Mix para predicción de la latencia promedio reporta un MAPE de 9.5 %, mientras que en este trabajo el dataset GNN_mix obtiene un MAPE de 33.91 %, lo que muestra una diferencia clara en desempeño bajo la configuración base. Sin embargo, no todos los resultados quedan igual de alejados: en el caso de GNN_4tier, el modelo alcanza un MAPE de 10.85 %, valor que se ubica en el mismo orden de magnitud que el reportado por el paper para el escenario Mix. Por su parte, los datasets GNN_social-network y GNN_social-network-ut presentan errores relativos mayores, del orden de 21–24 %, lo que sugiere que, bajo esta configuración inicial, esos escenarios resultan más difíciles de modelar.

Interpretación de estas diferencias. Que los resultados de esta etapa sean, en general, inferiores a los del paper no implica que la implementación haya fallado. El objetivo principal de este Proyecto de Grado no fue superar cuan-

titativamente al trabajo original en una primera instancia, sino implementar el modelo de forma fiel, funcional y reproducible, verificando que el pipeline completo —desde la construcción del grafo heterogéneo hasta la evaluación final— operara correctamente. En otras palabras, la meta central fue contar con una implementación propia de LQ-GNN que permitiera reproducir su lógica, validar su funcionamiento empírico y, sobre todo, habilitar una plataforma más flexible para futuras modificaciones y estudios.

Este punto es particularmente importante porque la implementación original en *Ignnition*, si bien fue suficiente para presentar el modelo, impone restricciones prácticas a la hora de inspeccionar, modificar o extender ciertos componentes internos. En cambio, la implementación realizada en este trabajo ofrece un mayor nivel de control sobre la arquitectura y sobre el pipeline experimental, permitiendo modelar con más detalle aspectos específicos del algoritmo y facilitando futuras variantes del modelo. Además, esta implementación permitió incorporar un aspecto relevante que no había podido resolverse en *Ignnition*: el procesamiento ordenado de los mensajes que recibe cada nodo `path` desde sus nodos `activity`, respetando el orden de llegada dentro del camino. Esto resulta especialmente valioso porque permite representar con mayor fidelidad la naturaleza secuencial de esas interacciones. Desde esa perspectiva, el valor principal de esta etapa no está solo en los números obtenidos, sino en haber demostrado que el modelo puede ser reconstruido, entrenado y extendido exitosamente fuera del framework original.

Alcance de esta comparación dentro del proyecto. Por lo tanto, esta comparación con el paper debe leerse como una validación de contexto: muestra que la implementación desarrollada produce resultados razonables y consistentes, aunque todavía no optimizados al máximo. Más adelante en este capítulo se presentan experimentos adicionales donde se ajustan hiperparámetros, función de pérdida, optimizador y estrategias de normalización. Esos resultados muestran que, cuando se dedica más esfuerzo a la optimización, el modelo presenta un margen de mejora importante. En consecuencia, los valores reportados en la Tabla 4.2 representan una línea base funcional sobre la cual luego se construyen variantes más fuertes, y no el techo de desempeño del modelo implementado.

Target P95 con la misma configuración base

Sustituyendo solo el target por el percentil 95, se obtiene la Tabla 4.4. Es esperable que el error empeore respecto del promedio: el P95 captura la cola alta de la distribución de latencias y suele ser intrínsecamente más difícil de predecir con el mismo presupuesto de modelo y entrenamiento "no optimizado".

Tabla 4.3: Resultados reportados en el paper (Tabla II): predicción del percentil 95 del tiempo de respuesta end-to-end. MAE y RMSE en s.

Escenario (App)	**MAPE (%)**	**MAE (s)**	**MSLE**	**RMSE (s)**
App 1	7.4	1.747	0.0210	12.533
App 2	6.9	2.539	0.0374	15.050
App 3	7.3	2.171	0.0213	13.838
Mix	10	1.798	0.0924	23.793

Tabla 4.4: Evaluación en test con target P95, utilizando la misma configuración base que en el caso de latencia promedio. MAE y RMSE en segundos (s).

Escenario (App)	**MAPE (%)**	**MAE (s)**	**MSLE**	**RMSE (s)**
App 1 (4-Tier / topología 1)	10.77	0.4402	0.0163	2.1841
App 2 (Social Network / topología 2)	26.63	7.7330	0.1485	22.3630
App 3 (Social Network UT / topología 3)	25.07	7.4320	0.1322	23.5260
Mix (entrenamiento cruzado)	33.81	7.7340	0.2583	25.8860

En los tres conjuntos, MAE y RMSE son claramente mayores que en el escenario de promedio (Tabla 4.2), lo que ilustra la mayor dificultad del objetivo P95 bajo configuración idéntica. Cabe denotar que el dataset de la app 1 (4 tier) tuvo resultados prácticamente iguales con una leve tendencia a la mejora, pero casi impercetible.

4.3.3. Configuración mejorada - Búsqueda de mejores hiperparámetros (P95 optimizado)

Para mostrar que los resultados anteriores no agotan el potencial del modelo, se realizó una ronda adicional.

Configuración optimizada (target P95)

Esta configuracion se centrada en el target P95. En la cual se utilizaron los siguientes hiperparámetros:

- **Modelo**: `hidden_dim` = 64, `num_layers` = 6, `dropout` = 0,2.
- **Entrenamiento**: `batch_size` = 64, `learning_rate` = 3×10^{-4}, `num_epochs` = 200, optimizador **AdamW**, `weight_decay` = 5×10^{-4}.
- **Pérdida**: Huber (`loss_name` = `huber`) con `huber_delta` = 0,5.
- **Regularización y estabilidad**: *early stopping*, recorte de gradientes y *ReduceLROnPlateau*.

- **Unidad del target**: consistente con la codificación presente en el JSON de cada dataset.

Bajo corridas dentro de este espacio de hiperparámetros, las métricas de test mejoraron de forma sustancial respecto de las obtenidas con la configuración base reportada en la Tabla 4.4. Esto sugiere que existe un margen importante de mejora cuando se introducen una función de pérdida más robusta, regularización explícita y una búsqueda de hiperparámetros más amplia. La Tabla 4.5 resume, por dataset, las métricas finales de test obtenidas a partir de los logs de esta configuración.

Datasets evaluados en esta etapa

Se reportan resultados para tres conjuntos:

- `GNN_mix`,
- `GNN_social-network`,
- `GNN_social-network-ut`.

Para el dataset `GNN_4tier`, la experimentación se realizó únicamente con el target de latencia promedio (*avg*), ya que este conjunto mostró resultados satisfactorios desde las primeras corridas. Además, por las restricciones de tiempo del proyecto ya discutidas en esta sección, se priorizó concentrar el esfuerzo experimental adicional en los demás datasets y configuraciones.

Resultados Obtenidos

Tabla 4.5: Evaluación en test para P95 con configuración optimizada (AdamW + Huber). MAE y RMSE en segundos (s).

Escenario (App)	**MAPE (%)**	**MAE (s)**	**MSLE**	**RMSE (s)**
App 2 (Social Network / topología 2)	7.65	0.720	0.0180	2.893
App 3 (Social Network UT/ topología 3)	8.62	2.537	0.0131	11.099
Mix (entrenamiento cruzado)	13.71	3.792	0.0406	20.205

Comparación con el paper. Estos resultados optimizados muestran una mejora clara respecto de la configuración base presentada en la Tabla 4.4 y permiten una comparación más favorable con los valores reportados en el paper para P95 (Tabla 4.3). En particular, para `GNN_mix` el modelo optimizado reduce el MAPE a 13.71 %, acercándose al valor de 10 % reportado por el escenario Mix del artículo, aunque todavía por encima de este. En los datasets `GNN_social-network` y `GNN_social-network-ut`, los valores obtenidos (7.65 % y 8.62 %, respectivamente) quedan mucho más próximos a los reportados en el

paper para las aplicaciones individuales de social network, cuyos MAPE se ubican entre 6.9 % y 7.3 %. Esto sugiere que, una vez ajustados los hiperparámetros y la estrategia de entrenamiento, la implementación desarrollada en este trabajo no solo mejora sustancialmente respecto de su propia línea base, sino que también se aproxima de forma razonable a los resultados del trabajo de referencia en la tarea de predicción de P95.

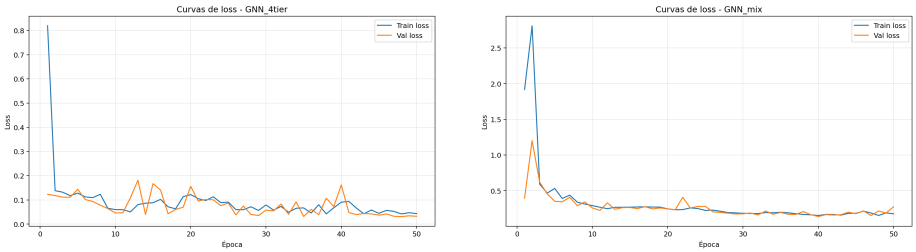
4.4. Comparación con la implementación original en *Ignnition*

Bajo la configuración base, los resultados obtenidos en este trabajo quedan claramente por debajo de los reportados en el paper. En la predicción de latencia promedio, por ejemplo, el escenario *Mix* del artículo reporta un MAPE de 9.5 %, mientras que la implementación desarrollada aquí alcanza 33.91 %. En P95 ocurre algo similar: frente a los valores del paper para *Mix* (10 %) y para las topologías individuales de Social Network (4.7–7.3 %), la configuración base de este trabajo obtiene errores bastante mayores, del orden de 24.46–25.07 %. Estos resultados muestran que una implementación funcional del modelo no garantiza por sí sola alcanzar el desempeño del trabajo original sin una etapa adicional de ajuste experimental.

Sin embargo, la comparación cambia de forma importante cuando se consideran las corridas optimizadas. En esa etapa, los resultados para P95 mejoran de manera sustancial: Social Network y Social Network UT pasan a 7.65 % y 8.62 % de MAPE, respectivamente, y el caso *Mix* baja a 13.71 %. Aunque estos valores todavía no igualan completamente a los del paper, sí muestran que la brecha puede reducirse de forma considerable cuando se introducen una función de pérdida más robusta, regularización explícita y una búsqueda de hiperparámetros más cuidadosa. En particular, para las topologías individuales de Social Network, la implementación desarrollada en este proyecto queda ya en un rango cercano al reportado por el trabajo original.

Desde esa perspectiva, el principal aporte de esta implementación no está únicamente en la comparación numérica directa con *IGNNITION*, sino en haber reconstruido el modelo en un entorno más flexible y controlable, capaz de reproducir su lógica, habilitar inspección detallada y permitir variantes de entrenamiento que en la implementación original resultaban más difíciles de explorar. En ese sentido, la experiencia experimental sugiere que el valor de esta implementación no es solo reproducir LQ-GNN, sino también ofrecer una base más adecuada para futuras extensiones y mejoras.

(a) `GNN_4tier` **(b) `GNN_mix`**



(c) `GNN_social-network` **(d) `GNN_social-network-ut`**

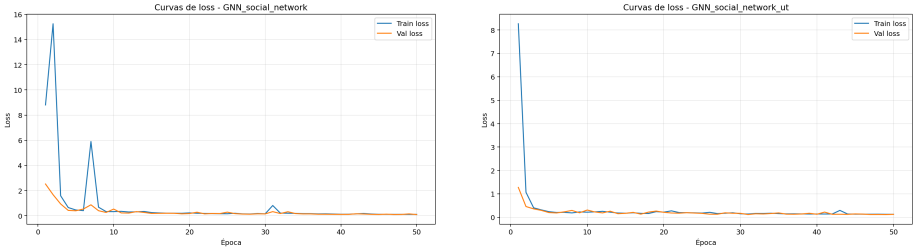


Figura 4.1: Curvas de *train loss* y *val loss* para los cuatro datasets evaluados en la configuración base con target promedio.

4.5. Análisis cualitativo de las curvas de pérdida (Configuración Base - AVG)

Las curvas de *train loss* y *val loss* permiten complementar la lectura de las métricas finales, ya que muestran cómo evoluciona el entrenamiento y hasta qué punto el modelo logra estabilizarse en cada dataset. En términos generales, en los cuatro conjuntos se observa una caída pronunciada de la pérdida durante las primeras épocas, lo que indica que el modelo aprende rápidamente patrones básicos de la relación entre la estructura del grafo y la latencia objetivo. Sin embargo, la forma posterior de las curvas no es igual en todos los casos y revela diferencias importantes en estabilidad y generalización.

Caso `GNN_4tier`. La curva de `GNN_4tier` muestra un descenso inicial rápido y luego una evolución relativamente baja en entrenamiento, pero con oscilaciones visibles en validación. La *val loss* presenta varios picos intermedios antes de estabilizarse en valores pequeños hacia el final, mientras que la *train loss* se mantiene en una banda baja pero todavía con cierta variabilidad. Esto sugiere que, aunque el modelo logra aprender el problema, la validación en este dataset es más sensible a pequeñas variaciones durante el entrenamiento. En otras palabras, no parece tratarse de un sobreajuste clásico y sostenido, sino más bien de un escenario donde la partición de validación resulta inestable o particularmente

exigente. Aun así, el comportamiento final de ambas curvas es consistente con el hecho de que este dataset obtuvo el mejor resultado de test para el target promedio dentro de la configuración base.

Caso `GNN_mix`. En `GNN_mix` se observa una etapa inicial mucho más inestable, con pérdidas altas al comienzo tanto en entrenamiento como en validación, seguida por una caída gradual hasta una zona más estable a partir de aproximadamente la mitad del entrenamiento. Luego de esa etapa, ambas curvas quedan relativamente cercanas, aunque la validación conserva cierta variabilidad y un leve deterioro al final. Esto indica que el entrenamiento converge, pero sobre un paisaje más difícil que en `GNN_4tier`. La interpretación es coherente con las métricas finales: si bien el modelo aprende y estabiliza la pérdida, el error de test permanece alto, lo que sugiere que el desafío en este dataset no está tanto en la optimización como en la dificultad intrínseca del problema bajo la configuración base.

Caso `GNN_social-network`. La curva de `GNN_social-network` presenta el comportamiento más abrupto en las primeras épocas, con pérdidas iniciales muy elevadas y algunos picos notorios de *train loss* incluso después del descenso inicial. A pesar de eso, la *val loss* muestra una tendencia general descendente y termina en valores bastante menores que los observados al comienzo. Esto sugiere que el entrenamiento atraviesa fases de inestabilidad local, pero consigue recuperar una trayectoria de convergencia aceptable. La existencia de esos picos en entrenamiento puede interpretarse como evidencia de una optimización más sensible al dataset o al batch, aun cuando la validación mantiene una tendencia razonablemente controlada. El resultado final es, por tanto, el de un entrenamiento funcional pero menos suave que en otros conjuntos.

Caso `GNN_social-network-ut`. En `GNN_social-network-ut` también se observa una caída muy fuerte durante las primeras épocas, seguida por una etapa de estabilización donde las curvas de entrenamiento y validación quedan bastante próximas entre sí. En comparación con `GNN_social-network`, este conjunto muestra una evolución algo más regular después del arranque inicial, con oscilaciones moderadas y sin una separación persistente entre entrenamiento y validación. Esto sugiere un comportamiento más estable del proceso de aprendizaje y una generalización razonable dentro de la configuración utilizada. Aun así, las métricas finales indican que esa estabilidad no se traduce automáticamente en el menor error absoluto o relativo, lo que refuerza la idea de que una curva de pérdida "prolija" no necesariamente implica mejor desempeño final si el dataset en sí mismo presenta mayor complejidad.

Lectura global. Tomadas en conjunto, las curvas muestran que el pipeline implementado es capaz de entrenar el modelo de forma consistente en los cuatro datasets, sin colapso de optimización ni divergencia sostenida. Las diferencias aparecen más bien en el grado de estabilidad y en la dificultad relativa de cada

conjunto: `GNN_4tier` combina buen desempeño final con validación oscilante, `GNN_mix` exhibe convergencia pero con error final alto, y los datasets de social network muestran descensos marcados con distinta sensibilidad a picos de entrenamiento. En consecuencia, estas curvas respaldan la interpretación general del capítulo: la implementación es funcional, pero el comportamiento del entrenamiento y la calidad de generalización dependen de forma importante del dominio y del dataset considerado.

4.6. Discusión

En conjunto, los resultados obtenidos muestran que la implementación desarrollada es funcional y capaz de entrenar el modelo de manera consistente en los datasets considerados. Bajo la configuración base y utilizando como objetivo la latencia promedio, el modelo alcanza desempeños razonables, aunque en general por debajo de los valores reportados en el paper de referencia. En este sentido, el objetivo principal del proyecto —contar con una implementación propia, reproducible y operativa de LQ-GNN— puede considerarse cumplido, aun cuando la primera configuración evaluada no reproduzca todavía el mejor desempeño cuantitativo del trabajo original.

Al sustituir el objetivo promedio por el percentil 95 sin modificar el resto de la configuración, los errores aumentan de forma sistemática. Este comportamiento es esperable, ya que los resultados del P95 del paper tambien fueron peores que los del AVG del mismo. Los resultados de esta etapa sugieren que la configuración base resulta suficiente para validar el funcionamiento general del pipeline, pero no necesariamente para capturar con precisión objetivos más exigentes como los percentiles altos.

Sin embargo, los experimentos con la configuración optimizada muestran que ese desempeño inicial no representa un límite del modelo implementado. La combinación de AdamW, pérdida Huber, regularización explícita, ajuste de la tasa de aprendizaje y una búsqueda más cuidadosa de hiperparámetros permitió reducir de forma marcada los errores en la predicción de P95. Esto indica que una parte importante de la brecha observada en la configuración base no responde exclusivamente a limitaciones de la arquitectura, sino también a decisiones de entrenamiento y calibración experimental.

En consecuencia, la principal lectura de esta etapa es doble. Por un lado, confirma que la implementación desarrollada reproduce de forma estable la lógica del modelo y permite obtener resultados coherentes en distintos escenarios. Por otro, muestra que el desempeño final depende de manera sensible de la configuración de entrenamiento, en particular cuando se trabaja con objetivos más complejos como la latencia en cola. A partir de esto, una línea natural de trabajo futuro consiste en profundizar la búsqueda sistemática de hiperparámetros y estrategias de regularización, e incluso explorar ajustes específicos por dataset para mejorar la capacidad de generalización del modelo.

Capítulo 5

Conclusiones y Trabajo Futuro

Este Proyecto de Grado tuvo como objetivo principal comprender, implementar y validar empíricamente el modelo LQ-GNN para la predicción de latencias en aplicaciones basadas en microservicios. El aporte central del trabajo no consistió en proponer una arquitectura nueva, sino en construir una implementación propia, funcional y reproducible del modelo, acompañada de un pipeline completo de datos, entrenamiento y evaluación experimental.

Además, este trabajo se desarrolló en el marco de una línea de investigación más amplia orientada al modelado y control de desempeño en sistemas de microservicios. En ese contexto, la implementación obtenida no debe verse únicamente como el resultado puntual de este informe, sino también como una base reutilizable para iteraciones futuras sobre el modelo, nuevos experimentos y posibles extensiones hacia mecanismos de soporte a decisiones.

5.1. Resultados alcanzados

El resultado principal del proyecto es una implementación end-to-end de LQ-GNN en Python, utilizando PyTorch y PyTorch Geometric para representar grafos heterogéneos, construir el pipeline de entrenamiento y evaluar el comportamiento del modelo sobre múltiples datasets. Esta implementación cubre todo el flujo necesario: carga y validación de los datos, construcción de instancias `HeteroData`, entrenamiento en GPU, cálculo de métricas y almacenamiento de resultados.

Desde el punto de vista experimental, los resultados obtenidos muestran que la implementación es funcional y que logra aprender patrones útiles de los datos, aunque con un desempeño que depende fuertemente tanto del dataset como del target considerado. Para la predicción de latencia promedio, la mejor configuración base se observó en `GNN_4tier`, donde el modelo alcanzó un MAPE de 10.85 % en test. En cambio, `GNN_mix` presentó un error considerablemente ma-

yor (33.91 %), mientras que `GNN_social-network` y `GNN_social-network-ut` se ubicaron en 21.81 % y 24.46 %, respectivamente. Estos resultados indican que la dificultad del problema no es uniforme entre dominios y que una misma configuración experimental no produce el mismo nivel de generalización en todos los conjuntos.

En el caso del target P95, la configuración base mostró un aumento consistente del error respecto del promedio, algo esperable dado que el percentil 95 representa la cola de la distribución de latencias. En este escenario, los valores de test fueron 33.81 % de MAPE para `GNN_mix`, 26.63 % para `GNN_social-network` y 25.07 % para `GNN_social-network-ut`. Sin embargo, una vez incorporadas variantes de mejora en el entrenamiento, el modelo mostró un margen de mejora muy significativo: los resultados optimizados para P95 descendieron a 13.71 % en `GNN_mix`, 7.65 % en `GNN_social-network` y 8.62 % en `GNN_social-network-ut`. Esto confirma que los resultados iniciales de la configuración base no representaban el techo del modelo, sino una línea de partida a partir de la cual el desempeño podía mejorar sustancialmente con una etapa adicional de ajuste.

En conjunto, estos experimentos permiten afirmar que el trabajo logró su meta principal: reconstruir el modelo, hacerlo entrenable y demostrar empíricamente su comportamiento en varios escenarios. También muestran que la capacidad predictiva de la implementación depende de manera importante de las decisiones experimentales y del esfuerzo dedicado a la optimización, especialmente cuando se trabaja sobre métricas más exigentes como el P95.

5.2. Aporte de la implementación y valor frente al framework original

Uno de los aportes más importantes del trabajo fue trasladar el modelo a un stack con mayor nivel de control y flexibilidad. La implementación original de referencia fue realizada sobre *Ignnition*, un framework útil para prototipado, pero que impone restricciones prácticas cuando se busca inspeccionar, modificar o extender componentes específicos del *message passing*. En cambio, la combinación de PyTorch y PyTorch Geometric permitió construir una versión más transparente y controlable del modelo.

Esta diferencia no fue solo una cuestión de comodidad de desarrollo. La implementación realizada en este proyecto permitió incorporar un aspecto relevante que no había podido resolverse de forma satisfactoria en el framework original: el procesamiento ordenado de los mensajes que recibe cada nodo `path` desde sus nodos `activity`, respetando el orden del camino mediante una agregación secuencial basada en GRU. Este punto es importante porque en LQ-GNN la estructura de un *path* no debe interpretarse simplemente como un conjunto de actividades, sino como una secuencia cuya composición afecta la latencia end-to-end.

Desde esta perspectiva, el valor de la implementación desarrollada no radica únicamente en las métricas obtenidas, sino también en haber construido una

versión del modelo que resulta más adecuada para investigación posterior. La nueva base permite modificar funciones de agregación, explorar variantes de optimización, estudiar nuevas transformaciones de escala y realizar ajustes finos que serían mucho más difíciles de implementar sobre una abstracción rígida.

5.3. Relación con el paper de referencia

La comparación con el trabajo original debe interpretarse como una referencia de contexto y no como una competencia estricta. El paper reporta, para la predicción del promedio, valores de MAPE entre aproximadamente 5 % y 7 % cuando el modelo se entrena por aplicación individual, y alrededor de 9.5 % en el escenario Mix. Para el caso de P95, los valores reportados son cercanos a 6.9 %–7.4 % para aplicaciones individuales y 10 % para Mix.

Frente a esa referencia, la configuración base de este trabajo produce en general resultados peores, especialmente en `GNN_mix` y en los experimentos iniciales con P95. No obstante, el caso de `GNN_4tier` con target promedio alcanza un MAPE de 10.85 %, que se ubica en el mismo orden de magnitud que el escenario Mix reportado en el paper. Más aún, al optimizar el entrenamiento para P95, los resultados en `GNN_social-network` (7.65 %) y `GNN_social-network-ut` (8.62 %) se aproximan de forma razonable a los valores reportados en el trabajo original para aplicaciones individuales.

Esto sugiere que la implementación desarrollada conserva la capacidad predictiva esencial del diseño LQ-GNN y que, aunque la configuración inicial no reproduce de inmediato los mejores resultados del paper, el modelo sí muestra potencial real de acercarse a ellos cuando se dedica una etapa más cuidadosa de optimización. En consecuencia, el principal valor del trabajo no está en haber superado la referencia, sino en haber construido una implementación fiel, funcional y extensible del modelo.

5.4. Contribuciones específicas

Las contribuciones concretas del proyecto pueden resumirse en los siguientes puntos:

- Se desarrolló una implementación de LQ-GNN en un stack moderno y flexible, basada en PyTorch y PyTorch Geometric.
- Se construyó un pipeline completo de datos, entrenamiento y evaluación, capaz de operar sobre grafos heterogéneos y producir métricas comparables entre distintos escenarios.
- Se validó empíricamente el modelo sobre varios datasets y targets, mostrando tanto el comportamiento de una configuración base como el potencial de mejora mediante optimización.

- Se incorporó una mejora relevante respecto del prototipo original: el tratamiento secuencial ordenado de los mensajes hacia los nodos `path`.
- Se dejó una base de software adecuada para realizar extensiones futuras, estudios comparativos y variantes metodológicas del modelo.

5.5. Trabajo futuro

A partir de los resultados obtenidos, se abren varias líneas de trabajo futuro.

Una primera línea natural consiste en profundizar la optimización del entrenamiento, Los resultados sobre P95 muestran que existe un margen de mejora importante al ajustar hiperparámetros, regularización, optimizador y función de pérdida. Por ello, una continuación directa del proyecto sería realizar una búsqueda sistemática más amplia, incluyendo tamaño de batch, dimensión oculta, número de iteraciones de *message passing*, tasas de aprendizaje, *weight decay*, *dropout*, criterios de *early stopping* y pérdidas robustas.

Una segunda línea es estudiar con más detalle el impacto de las transformaciones de escala y estrategias de normalización. Dado que la predicción de latencias es especialmente sensible al rango dinámico del target y a la presencia de colas pesadas, un análisis más profundo de estas decisiones podría contribuir a estabilizar el entrenamiento y mejorar la generalización.

Una tercera línea, de carácter más aplicado, es avanzar hacia la integración del predictor en un ciclo de decisión. En particular, el modelo podría utilizarse para evaluar escenarios *what-if* asociados a cambios de configuración, despliegue o asignación de recursos, actuando como componente de soporte dentro de mecanismos de elasticidad o control.

También resulta relevante explorar validaciones más cercanas a entornos reales, incluyendo posibles problemas de adaptación de dominio y *drift*, entendido como la degradación del desempeño del modelo cuando las condiciones del sistema o la distribución de los datos cambian con el tiempo. En ese contexto, podrían estudiarse estrategias de *fine-tuning*, reentrenamiento periódico o actualización continua del modelo.

Por último, la implementación desarrollada deja abierta la posibilidad de investigar variantes arquitectónicas del propio LQ-GNN, así como incorporar mecanismos de interpretabilidad o incluso formulaciones multitarea que permitan predecir no solo latencia end-to-end, sino también otras métricas de desempeño relevantes.

5.6. Cierre

En síntesis, este Proyecto de Grado entrega una implementación completa, funcional y reproducible de LQ-GNN, junto con una evaluación experimental que muestra tanto sus limitaciones iniciales como su potencial de mejora. El trabajo demuestra que el modelo puede ser reconstruido exitosamente fuera del framework original, que puede entrenarse de forma controlada sobre distintos

datasets y que, con una etapa de optimización apropiada, puede acercarse de manera razonable a los resultados reportados en la literatura.

Por ello, la principal contribución del proyecto no es únicamente haber obtenido determinadas métricas, sino haber dejado una base sólida para investigación futura: una implementación más controlable, más extensible y más adecuada para estudiar con detalle el comportamiento del modelo y sus posibles evoluciones en el contexto de predicción de latencias para sistemas de microservicios.

5.7. Reflexión

Más allá de los resultados numéricos obtenidos, este proyecto me dejó un aprendizaje que considero especialmente valioso: implementar un modelo publicado no es simplemente "llevar una idea a código", sino atravesar un proceso mucho más profundo de comprensión, validación y cuestionamiento. A lo largo del trabajo, muchas veces quedó en evidencia que leer un paper y entender su propuesta a nivel general no alcanza para reproducirlo de forma fiel. Recién al enfrentarse a los detalles concretos de la implementación —la representación de los datos, la semántica de cada relación, las decisiones de agregación, la estabilidad numérica del entrenamiento y la interpretación de los resultados— aparece una comprensión real del modelo y de sus dificultades.

En ese sentido, uno de los aspectos más enriquecedores del proyecto fue comprobar que una parte importante del trabajo de investigación no pasa solamente por proponer ideas nuevas, sino también por reconstruir, validar y volver reutilizable conocimiento existente. Poder implementar LQ-GNN fuera de su framework original implicó no solo estudiar el modelo con profundidad, sino también tomar decisiones prácticas, resolver ambigüedades y construir una base que otros puedan continuar usando. Desde esta perspectiva, el valor del trabajo no está únicamente en las métricas alcanzadas, sino en haber transformado una propuesta teórica en un artefacto funcional, trazable y extensible.

También, en un plano más personal, haber podido participar —gracias a mis tutores— de reuniones semanales con un grupo de investigación fue una de las experiencias más valiosas de todo el proyecto. Para mí eso fue realmente oro. No solo por lo que aprendí en términos técnicos, sino porque me permitió ver más de cerca cómo se discuten ideas, cómo se formulan preguntas y cómo se construye conocimiento de manera colectiva. Fue una instancia que disfruté mucho y que terminó siendo una parte muy importante de lo que me llevo de esta etapa.

Además, este proyecto me permitió experimentar de forma muy concreta la diferencia entre "hacer que algo funcione" y "entender por qué funciona, cuándo deja de funcionar y cómo podría mejorarse". Ese cambio de mirada fue, probablemente, uno de los aportes personales más importantes del proceso. Me obligó a prestar atención no solo al resultado final, sino también a la calidad del pipeline, a la coherencia de las decisiones experimentales y a la forma en que cada modificación impacta en el comportamiento del modelo.

Finalmente, este trabajo reafirmó mi interés por el área de investigación en la intersección entre sistemas distribuidos y aprendizaje automático. Haber participado en un proyecto vinculado a una línea de trabajo real, con potencial continuidad y aplicación, hizo que este Proyecto de Grado no fuera solamente una instancia de cierre de la carrera, sino también una experiencia formativa que me permitió acercarme a la investigación de una manera mucho más práctica, crítica y concreta.

Capítulo 6

Anexo - Scripts de ejecución en el clúster

En este anexo se incluye un ejemplo del script utilizado para ejecutar entrenamientos en el clúster mediante SLURM.

Listing 6.1: Script SLURM utilizado para ejecutar entrenamientos en el clúster (ejemplo para `GNN_4tier`).

```
#!/bin/bash

#SBATCH --job-name=gnn_data
#SBATCH --partition=normal
#SBATCH --qos=gpu
#SBATCH --gres=gpu:p100:1
#SBATCH --cpus-per-task=4
#SBATCH --mem=64G
#SBATCH --time=08:00:00
#SBATCH --output=slurm-%j.out

set -euo pipefail

# 1) Activar conda
source ~/miniconda3/etc/profile.d/conda.sh
conda activate lqgnn

# 2) Trabajar en /scratch (I/O rapido)
RUN_DIR=/scratch/$USER/gnn_runs/job_$SLURM_JOB_ID
mkdir -p "$RUN_DIR"

# Copiar proyecto desde GNN_data
rsync -a --delete ~/GNN/GNN_data/ "$RUN_DIR"/

cd "$RUN_DIR"
mkdir -p outputs models
```

```
# Verificar que data/ existe (importante)
if [ ! -d "$RUN_DIR/data" ]; then
    echo "ERROR: Carpeta data/ no encontrada en $RUN_DIR"
    exit 1
fi

echo "Nodo: $(hostname)"
echo "CUDA_VISIBLE_DEVICES=$CUDA_VISIBLE_DEVICES"
echo "Directorio de trabajo: $RUN_DIR"
echo "Proyecto: GNN_data"
echo "=== Rutas de datos ==="
echo "train_path: $RUN_DIR/data/train"
echo "val_path: $RUN_DIR/data/train_valid"
echo "test_path: $RUN_DIR/data/test"
echo "======================"

# 3) Ejecutar
python train.py "$@" \
  --output_log_path   "outputs/run_${SLURM_JOB_ID}.txt" \
  --output_model_path "models/model_${SLURM_JOB_ID}.pth"

# 4) Traer resultados a HOME (carpeta por job)
mkdir -p ~/GNN/GNN_data/outputs/job_$SLURM_JOB_ID
mkdir -p ~/GNN/GNN_data/models/job_$SLURM_JOB_ID
rsync -a outputs/ ~/GNN/GNN_data/outputs/job_$SLURM_JOB_ID/
rsync -a models/ ~/GNN/GNN_data/models/job_$SLURM_JOB_ID/
    2>/dev/null || true

echo " Resultados copiados a ~/GNN/GNN_data/outputs/
    job_$SLURM_JOB_ID/"
```

Referencias

Barcelona Neural Networking Center (BNN-UPC). (2026). *Ignnition: Fast prototyping of graph neural networks.* https://ignnition.org/. (Accessed: 2026-04-27)

Dean, J., y Barroso, L. A. (2013a). The tail at scale. *Communications of the ACM*, *56*(2), 74–80. doi: 10.1145/2408776.2408794

Dean, J., y Barroso, L. A. (2013b). The tail at scale. *Communications of the ACM*, *56*(2), 74–80. Descargado de https://www.barroso.org/publications/TheTailAtScale.pdf (Open PDF) doi: 10.1145/2408776.2408794

Fowler, M., y Lewis, J. (2014). *Microservices.* martinfowler.com. Descargado de https://martinfowler.com/articles/microservices.html (Accessed: 2026-01-29)

Franks, G., Al-Omari, T., Woodside, M., Das, O., y Derisavi, S. (2009, marzo). Enhanced Modeling and Solution of Layered Queueing Networks. *IEEE Transactions on Software Engineering*, *35*(2), 148–161. Descargado 2023-12-13, de http://ieeexplore.ieee.org/document/4620121/ doi: 10.1109/TSE.2008.74

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., y Dahl, G. E. (2017). Neural message passing for quantum chemistry. En *Proceedings of the 34th international conference on machine learning* (Vol. 70, pp. 1263–1272). PMLR. Descargado de https://proceedings.mlr.press/v70/gilmer17a.html

Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep learning.* MIT Press. (Available online: https://www.deeplearningbook.org/)

Google LLC. (2024). *tensorflow-gnn: Tensorflow gnn.* Python Package Index (PyPI). Descargado de https://pypi.org/project/tensorflow-gnn/ (Version 1.0.3. Accessed: 2026-01-29)

Huber, P. J. (1964). Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, *35*(1), 73–101. doi: 10.1214/aoms/1177703732

Kingma, D. P., y Ba, J. (2015). Adam: A method for stochastic optimization. En *International conference on learning representations.*

Kipf, T. N., y Welling, M. (2017). Semi-supervised classification with graph convolutional networks. En *International conference on learning representations.* Descargado de https://openreview.net/forum?id=SJU4ayYgl

Loshchilov, I., y Hutter, F. (2019). Decoupled weight decay regularization. En *International conference on learning representations*. Descargado de https://openreview.net/forum?id=Bkg6RiCqY7

Newman, S. (2015). *Building microservices*. O'Reilly Media.

Park, J., Choi, B., Lee, C., y Han, D. (2021, diciembre). GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. En *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies* (pp. 154–167). Virtual Event Germany: ACM. Descargado 2023-11-14, de https://dl.acm.org/doi/10.1145/3485983.3494866 doi: 10.1145/3485983.3494866

PyTorch Geometric. (2025). *Messagepassing api*. PyTorch Geometric Documentation. Descargado de https://pytorch-geometric.readthedocs.io/en/2.7.0/generated/torch_geometric.nn.conv.MessagePassing.html (Accessed: 2026-01-29)

Richart, M., Gorricho, J.-L., Baliosian, J., Contreras, L. M., Muñiz, A., y Serrat, J. (2025). Lq-gnn: A graph neural network model for response time prediction of microservice-based applications in the computing continuum. *IEEE Transactions on Parallel and Distributed Systems*, *36*(12), 2566-2577. Descargado de https://www.fing.edu.uy/inco/grupos/mina/lqgnn/index.html doi: 10.1109/TPDS.2025.3564214

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., y Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, *20*(1), 61–80. doi: 10.1109/TNN.2008.2005605

SchedMD. (2026). *Slurm workload manager documentation*. https://slurm.schedmd.com/documentation.html. (Accessed: 2026-04-27)

Tam, D. S. H., Liu, Y., Xu, H., Xie, S., y Lau, W. C. (2023, agosto). PERT-GNN: Latency Prediction for Microservice-based Cloud-Native Applications via Graph Neural Networks. En *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (pp. 2155–2165). Long Beach CA USA: ACM. Descargado 2023-08-23, de https://dl.acm.org/doi/10.1145/3580305.3599465 doi: 10.1145/3580305.3599465

torch_geometric.data.heterodata. (2025). PyTorch Geometric Documentation. Descargado de https://pytorch-geometric.readthedocs.io/en/2.7.0/generated/torch_geometric.data.HeteroData.html (Accessed: 2026-01-29)

Veličkovi'c, P., Cucurull, G., Casanova, A., Romero, A., Li'o, P., y Bengio, Y. (2018). Graph attention networks. En *International conference on learning representations*. Descargado de https://openreview.net/forum?id=rJXMpikCZ

Zhang, Y., Gan, Y., y Delimitrou, C. (2019). uqsim: Scalable and validated simulation of cloud microservices. *arXiv preprint*. Descargado de https://arxiv.org/abs/1911.02122 (arXiv:1911.02122)

Zhang, Y., Hua, W., Zhou, Z., Suh, G. E., y Delimitrou, C. (2021, abril). Sinan: ML-based and QoS-aware resource management for cloud microservices. En *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 167–

181). Virtual USA: ACM. Descargado 2023-08-23, de https://dl.acm.org/doi/10.1145/3445814.3446693 doi: 10.1145/3445814.3446693