ULPGC Universidad de Las Palmas de Gran Canaria | ESCUELA DE INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Optimización de Turnos Mediante Modelos Matemáticos e integración de Técnicas de Inteligencia Artificial. (ResQPlan/AI)

TITULACIÓN: Grado en Ciencia e Ingeniería de Datosl

AUTOR: Cynthia Quintana Reyes

TUTORIZADO POR:
Jose Miguel Santos Espino
Izzat Sabbagh Rodríguez

Fecha [06/2025]

ULPGC Universidad de Las Palmas de Gran Canaria | EII ESCUELA DE INGENIERÍA INFORMÁTICA

TFT04

SOLICITUD DE DEFENSA DE TRABAJO DE FIN DE TÍTULO

D./Dª Cynthia Quintana Reyes, autora del trabajo Optimización de Turnos Mediante Modelos Matemáticos e integración de Técnicas de Inteligencia Artificial. (ResQPlan/AI), correspondiente a la titulación Ciencia e Ingeniería de Datos, en colaboración con la empresa/proyecto Singular Factory

SOLICITA

que se inicie el procedimiento de defensa del mismo, para lo que se adjunta la documentación requerida, haciendo constar que

[X] se autoriza / [ ] no se autoriza la grabación en audio de la exposición y turno de preguntas.

Asimismo, con respecto al registro de la propiedad intelectual/industrial del TFT, declara que:

[ ] Se ha iniciado o hay intención de iniciarlo (defensa no pública).

[X] No está previsto.

Y para que así conste firma la presente. (fecha en firma electrónica)

El/La estudiante

Fdo.————————

A rellenar y firmar **obligatoriamente** por el/la/los/las tutores

En relación con la presente solicitud, se informa:(firmar donde corresponda)

Positivamente (en caso de detección de copia, esta firma quedará invalidada)	Negativamente (justificación en TFT05)
Fdo.————————	Fdo.————————

DIRECTOR DE LA ESCUELA DE INGENIERÍA INFORMÁTICA

Agradecimientos

A mis padres, cuyo apoyo incondicional me ha traído hasta aquí;
a mi hermana, por confiar siempre en mí;
a mi tutor José Miguel, por su orientación y ayuda constantes;
y a mi amiga y compañera Andrea, por brindarme su compañía y apoyo a lo largo de este camino.

Resumen

Este Trabajo Fin de Grado presenta ResqPlan/AI, un sistema que convierte la planificación de turnos en un proceso asistido por IA. El contexto operativo y las reglas redactadas en lenguaje natural se traducen, mediante modelos generativos, en restricciones de programación lineal entera que el solver Gurobi optimiza. Cuando las condiciones resultan incompatibles, el motor detecta la inviabilidad y relaja únicamente los requisitos, registrando cada ajuste. El objetivo es acortar drásticamente el tiempo dedicado a confeccionar horarios, eliminar errores manuales y distribuir la carga laboral de forma equitativa, garantizando al mismo tiempo la continuidad del servicio y el cumplimiento de los límites legales y de descanso.

Abstract

This Final Degree Project presents ResqPlan/AI, a system that transforms shift planning into an AI-assisted process. The operating context and rules written in natural language are translated, using generative models, into integer linear programming constraints that are optimized by the Gurobi solver. When conditions are incompatible, the engine detects infeasibility and relaxes only the requirements, recording each adjustment. The goal is to drastically reduce the time spent on schedule creation, eliminate manual errors, and distribute the workload equitably, while ensuring service continuity and compliance with legal and rest limits.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La planificación de turnos laborales constituye uno de los retos operativos más críticos en organizaciones con operaciones continuas [4], como hospitales, servicios de emergencia o centros de atención al cliente. Este problema, clasificado como NP-difícil en la literatura computacional [17], requiere conciliar múltiples factores: normativas legales, preferencias individuales y coberturas mínimas exigidas [1]. Sin embargo, los métodos manuales, aún predominantes en muchos entornos, son ineficientes para manejar fluctuaciones en la demanda o ausencias imprevistas [21]. Esto puede conducir a un ambiente laboral deficiente y, consecuentemente, a una disminución en la calidad del servicio, afectando el bienestar de los empleados [20].

En este contexto, los avances en inteligencia artificial y optimización matemática han surgido como una alternativa prometedora. En la última década, algoritmos heurísticos y técnicas de programación entera han demostrado capacidad para generar propuestas de horarios más eficientes y rápidas que los métodos tradicionales [12]. Asimismo, la integración de herramientas que interpretan restricciones en lenguaje natural ha facilitado su adopción, permitiendo a usuarios no técnicos interactuar con estos sistemas de manera intuitiva [19].

Este Trabajo Fin de Grado, desarrollado en colaboración con la empresa Singular Factory, se centra en la intersección entre estas innovaciones tecnológicas y las necesidades organizativas. La motivación de Singular Factory para este proyecto radica en su compromiso por ofrecer soluciones innovadoras que optimicen la gestión de recursos humanos en sus clientes, mejorando la eficiencia operativa y el bienestar de los empleados. Su objetivo es desarrollar un sistema automatizado que combine la expresividad del lenguaje natural con algoritmos de optimización, ofreciendo a los gestores una herramienta que reduzca el tiempo de diseño, minimice errores y garantice equidad en la distribución de turnos, especialmente en entornos donde los fallos tienen consecuencias críticas.

1.1. Contexto y relevancia del trabajo

Aunque los sistemas automatizados para la gestión de turnos han avanzado mucho, buscando optimizar operaciones en diversos sectores, aún enfrentan desafíos importantes al implementarse en organizaciones con entornos dinámicos. Por ejemplo, algunas bibliotecas de

optimización muy usadas en la industria, como *IBM ILOG CP Optimizer*, tienen limitaciones documentadas para adaptarse a lugares de trabajo cambiantes, a menudo por lo complejos que son sus modelos y lenguajes de programación [10].

Estas dificultades no son solo técnicas, sino que impactan directamente en la adopción y eficacia de estas herramientas. Un problema principal es su limitada capacidad para manejar restricciones dinámicas y múltiples objetivos. Muchos modelos fallan al adaptarse a cambios en los requisitos o en la plantilla, ya que sus algoritmos a menudo se basan en entornos estáticos, a pesar de que la realidad operativa de muchas organizaciones es naturalmente cambiante. Esta rigidez dificulta la rápida integración de nuevas necesidades, como cambios inesperados en la demanda de personal, ausencias imprevistas o la necesidad de incorporar nuevas normativas o preferencias de los empleados. En consecuencia, la implementación de estos sistemas puede resultar costosa y lenta, y su valor real disminuye si no pueden evolucionar al ritmo de las operaciones diarias.

1.2. Motivaciones

La necesidad de optimizar la planificación de turnos ha sido el punto de partida fundamental para este proyecto, impulsado por Singular Factory. Esta exigencia se hizo particularmente patente al abordar la gestión de los turnos para los retenes de emergencia de La Palma. En este contexto, el carácter crítico de las operaciones y la gran variabilidad de los recursos exigían un método de gestión ágil y, sobre todo, fiable. Se ha observado que los coordinadores se enfrentaban regularmente a cambios de última hora, que resultaban extremadamente complejos y engorrosos de manejar con metodologías manuales o herramientas tradicionales, generando solapamientos, cuellos de botella operativos y un impacto negativo en el bienestar del equipo.

El reconocimiento del gran potencial de un sistema capaz de traducir automáticamente las restricciones en lenguaje natural a formulaciones de optimización impulsó la idea inicial de aplicar esta solución al caso de La Palma. Sin embargo, dado el amplio espectro de necesidades de planificación en diversos sectores y el compromiso de Singular Factory con soluciones de impacto general, se decidió evolucionar hacia un enfoque donde el procesamiento de lenguaje natural (PLN) fuera la vía principal para definir reglas y requisitos. Así, se ha diseñado una plataforma genérica que puede adaptarse a una gran variedad de sectores y necesidades de planificación de turnos. Un aspecto clave de esta propuesta es garantizar la facilidad de uso para quienes no tienen conocimientos de programación. La integración del PLN con las técnicas de optimización no solo mejora significativamente el cálculo de horarios, sino que también ofrece una flexibilidad esencial para incorporar nuevas reglas o modificar operaciones de forma dinámica. Además, esto favorece una distribución más justa de las cargas de trabajo y reduce el tiempo dedicado a ajustes manuales. Se considera que estas mejoras consolidan la herramienta como un asistente fiable que optimiza tanto la eficiencia operativa como el bienestar del personal, sea cual sea el contexto de aplicación.

1.3. Objetivos

1.3.1. Objetivos del proyecto

El propósito fundamental de este trabajo es el desarrollo de una herramienta de planificación de turnos que sea capaz de transformar de manera automática y transparente texto en lenguaje natural en un modelo de optimización resoluble mediante el *solver* Gurobi. Para lograr este fin, se diseñará una estructura de datos flexible que será alimentada por el módulo de *PLN*. Dicha estructura permitirá la representación dinámica de elementos clave como días, franjas horarias y recursos disponibles, facilitando la definición de variables y restricciones sin que se requiera la programación manual de cada nuevo escenario.

Para conseguirlo, se procederá a la implementación de un módulo de procesamiento de lenguaje natural (*PLN*). Este módulo tendrá la función de extraer e interpretar la información relevante contenida en los requisitos operativos formulados en lenguaje natural, y de generar a partir de esta las variables y el código de restricción correspondiente para el entorno Gurobi. Se espera que esta automatización empodere a usuarios sin conocimientos previos en programación para que puedan formular sus propias reglas, observando cómo estas se traducen de manera inmediata al modelo matemático subyacente que el *solver* Gurobi ejecutará. Adicionalmente, el trabajo final incluirá una interfaz web para la comunicación con el usuario, desarrollada por **Andrea Hernando González** como parte de su Trabajo Fin de Grado.

Finalmente, se contempla la validación rigurosa del sistema mediante la aplicación a datos reales. Inicialmente, se priorizará el caso de los retenes de emergencia de La Palma, para posteriormente ampliar su ámbito de aplicación a otros sectores operativos. En el marco de estas pruebas, se realizará un análisis exhaustivo de los tiempos de cálculo, la calidad de las soluciones propuestas y la facilidad con la que el sistema permite la incorporación de cambios dinámicos. El objetivo último es garantizar que la plataforma resultante sea no solo precisa en sus optimizaciones, sino también altamente adaptable a futuros escenarios de planificación.

1.3.2. Objetivos académicos

En el plano estrictamente universitario el proyecto persigue los siguientes fines:

1. Diseñar un sistema de planificación de turnos que permita poner en práctica metodologías formales de modelado y optimización, partiendo de requisitos operativos expresados en lenguaje natural.

2. Integrar técnicas avanzadas de programación lineal y de Procesamiento del Lenguaje Natural para demostrar la viabilidad de soluciones híbridas — IA + investigación operativa — dentro de un único flujo de trabajo.

3. Desarrollar una prueba de concepto que contemple el ciclo completo: captura de requisitos, generación automática de variables y restricciones, resolución con un *solver* comercial y

presentación de resultados a usuarios no técnicos.

4. Fomentar la capacidad de análisis crítico mediante la comparación de distintos dominios (emergencias, docencia, logística) y la evaluación de su impacto en tiempos de cómputo, calidad de solución y mantenimiento del sistema.

5. Aplicar una metodología iterativa que combine entregas frecuentes, validación con casos reales y retroalimentación continua, favoreciendo la adaptación a requisitos cambiantes.

1.3.3. Competencias desarrolladas

ED4. *Capacidad de identificar y analizar problemas y diseñar, desarrollar, implementar, verificar y documentar soluciones* ***software*** *en ámbitos de aplicación de la Inteligencia Artificial en Ciencia e Ingeniería de Datos.*

- ✔ El proyecto obliga a identificar las variables críticas de tres dominios distintos, formalizarlas en un modelo matemático y documentar la transformación automática desde texto libre hasta código Gurobi.
- ✔ La verificación se materializa en la generación de IIS, la relajación de viabilidad y las pruebas con datos reales, todo ello registrado en informes reproducibles.

ED1. *Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación.*

- ✔ Se estudian los fundamentos de los *Large Language Models*, su integración mediante API y su orquestación con un *solver* de programación lineal.
- ✔ El diseño modular (captura de texto, generación de variables, traducción de restricciones, optimización y visualización) demuestra cómo combinar IA y métodos clásicos para construir un servicio completo adaptable a múltiples ámbitos.

1.4. Alcances y limitaciones

El presente trabajo se enfoca en el diseño e implementación de una plataforma genérica para la planificación de turnos, con la capacidad de adaptarse a diversos contextos organizativos. Para la fase de validación inicial, se ha seleccionado el caso de los retenes de emergencia de La Palma. En su configuración actual, el sistema soporta exclusivamente escenarios con horarios fijos y recursos previamente categorizados (por ejemplo, diferenciación entre pilotos, copilotos o auxiliares), asumiendo que la información relativa a la disponibilidad del personal y los requisitos reglamentarios se encuentra completa y ha sido definida con precisión.

La funcionalidad de traducción automática de texto en lenguaje natural a restricciones de optimización se implementa mediante la integración con modelos avanzados de lenguaje a

través de la API de OpenAI. Esto implica que la capacidad de interpretar y procesar las redacciones de los usuarios depende directamente de la robustez y comprensión del modelo de IA subyacente. Consecuentemente, formulaciones excesivamente divergentes del lenguaje habitual o que contengan ambigüedades intrínsecas pueden derivar en interpretaciones erróneas o incompletas del requisito. Además, la calidad de los resultados obtenidos depende directamente de la coherencia y exhaustividad de las especificaciones introducidas por el usuario; la inclusión de reglas parciales o la presencia de contradicciones en los requisitos pueden ocasionar que el *solver* no identifique una solución válida o que relaje restricciones de manera no deseada.

En lo que respecta al rendimiento computacional y la escalabilidad, es importante destacar que este proyecto no tiene como objetivo principal desarrollar un sistema que sea directamente escalable para cualquier carga de trabajo o tamaño de plantilla desde su fase inicial. Se reconoce que la complejidad del problema de optimización de turnos puede crecer exponencialmente con el número de variables y restricciones. Sin embargo, se buscará evaluar y comprender las capacidades de escalabilidad del modelo desarrollado para establecer sus límites y entender la viabilidad del producto en diferentes escenarios reales. La arquitectura del sistema ha sido concebida de manera modular, permitiendo futuras extensiones y optimizaciones para abordar necesidades de mayor escala en desarrollos posteriores.

1.5. Organización del documento

El presente documento se estructura en ocho capítulos, procurando una progresión lógica que va de los aspectos más generales a los más específicos del trabajo. Tras esta Introducción, en la que se ha enmarcado el problema, justificado su relevancia, y descrito las motivaciones y objetivos del estudio, se ofrece una descripción detallada de los capítulos subsiguientes.

El Capítulo 2 se dedica a la revisión del estado del arte en la planificación de turnos, así como en la aplicación de técnicas de procesamiento de lenguaje natural al modelado de restricciones. En este apartado se evalúan tanto las bibliotecas clásicas de optimización como los enfoques más recientes que combinan el PLN con los *solvers* matemáticos. Posteriormente, el Capítulo 3 incorpora la elección de herramientas y se comparan las alternativas más relevantes existentes (Gurobi, OR-Tools y PuLP.), estableciendo los criterios de rendimiento, licenciamiento y compatibilidad con el módulo de PLN que motivan la adopción final de Gurobi.

El Capítulo 4 describe la metodología adoptada y la planificación del proyecto, detallando las fases clave que incluyen la comprensión del problema, el diseño de la arquitectura del sistema, su desarrollo y las iteraciones de prueba.

En el Capítulo 5 se expone la formulación matemática del modelo de optimización. Esto abarca la definición precisa de las variables, los parámetros y las restricciones, así como el esquema de datos que alimenta al *solver* Gurobi. A continuación, el Capítulo 6 presenta en detalle el módulo de procesamiento de lenguaje natural, explicando el mecanismo mediante el cual se extraen automáticamente las entidades y las reglas desde el texto en lenguaje natural para su posterior traducción a código interpretable por Gurobi. Seguidamente, el Capítulo 7

muestra los resultados de la validación del sistema, incluyendo los casos de prueba iniciales con los retenes de La Palma, la ampliación a otros escenarios, las métricas de tiempo de cálculo y la calidad de las soluciones obtenidas, y ejemplos de uso relevantes.

Finalmente, el Capítulo 8 contiene las conclusiones derivadas de este trabajo, una reflexión crítica sobre las limitaciones identificadas durante el desarrollo y una propuesta de futuras líneas de investigación y evolución del sistema.

Capítulo 2

Estado del arte

La optimización de la planificación de turnos ha recorrido un largo camino, evolucionando desde simples esquemas manuales hasta sofisticados modelos matemáticos capaces de gestionar cientos de variables y restricciones. Sin embargo, a pesar de estos avances, experiencias recientes —como las dificultades en la programación del personal sanitario durante la pandemia de COVID-19 [9] o los problemas en la gestión de turnos de tripulaciones aéreas ante disrupciones operativas [15]— evidencian que, por muy potentes que sean, los *solvers* tradicionales todavía carecen de la flexibilidad y la capacidad de adaptación requeridas en entornos dinámicos.

En respuesta a esta realidad, este capítulo ofrece una panorámica de las soluciones actualmente disponibles, así como de los desafíos que persisten. Primero, se presentarán los fundamentos de los principales modelos matemáticos empleados en la planificación de turnos, como la *Programación Lineal* (LP), la *Programación Entera Mixta* (MILP) y diversas *metaheurísticas*. A continuación, se realizará un estudio crítico entre las tres herramientas más consolidadas en este ámbito, con el fin de establecer los criterios que guiarán la elección de la plataforma más adecuada para el desarrollo de este proyecto.

Asimismo, se analizará cómo el *Procesamiento de Lenguaje Natural* (PLN) y las técnicas de *machine learning* están siendo progresivamente incorporadas a estos sistemas, con el objetivo de hacerlos más interactivos y adaptativos. Finalmente, se revisarán aplicaciones reales y estudios previos que han integrado la optimización matemática con la inteligencia artificial, evaluando sus resultados y las lecciones aprendidas. Esta visión integradora permitirá situar con precisión el aporte de este trabajo, así como justificar las decisiones de diseño y las herramientas adoptadas.

2.1. Modelos Matemáticos en la Optimización de Turnos

La optimización de turnos es un campo clave de la Investigación Operativa que busca asignar personal a tareas en franjas horarias específicas, considerando múltiples restricciones. La elección del modelo matemático adecuado depende de la naturaleza del problema y la escala de la operación.

2.1.1. Programación Lineal (LP)

La Programación Lineal (LP) es una técnica fundamental que se aplica a problemas donde las relaciones entre variables son lineales. Aunque es útil para ciertos escenarios, su aplicación directa en la asignación de personal es limitada, ya que los problemas de turnos suelen requerir que las variables tomen valores enteros.

2.1.2. Programación Entera Mixta (MILP)

La Programación Entera Mixta (MILP) extiende la LP al permitir variables de decisión enteras, binarias o continuas. Esta capacidad la convierte en una herramienta muy potente y ampliamente usada en la optimización de turnos, ya que permite modelar la naturaleza discreta de las asignaciones de personal. Con MILP se pueden incorporar restricciones realistas como los límites de horas de trabajo, la disponibilidad del personal, la secuencia de turnos y descansos, la cobertura mínima requerida y la distribución equitativa de la carga laboral. Su eficacia ha sido demostrada en diversos contextos, como la optimización de turnos en hospitales, donde ha contribuido a reducir costos operativos y mejorar la distribución del personal.

2.1.3. Metaheurísticas

Cuando los problemas de optimización de turnos son demasiado grandes o complejos para resolverse de manera exacta con métodos como MILP en un tiempo razonable, se utilizan las metaheurísticas. Estas son estrategias de búsqueda aproximadas que exploran grandes espacios de soluciones para encontrar resultados de buena calidad de forma eficiente. Son especialmente útiles en problemas donde la rapidez de la solución es crucial. Entre las metaheurísticas más aplicadas en la optimización de turnos se encuentran los Algoritmos Genéticos (GA), que, inspirados en la evolución biológica, refinan soluciones candidatas mediante operadores de selección, cruce y mutación. Otra técnica es la Búsqueda Tabú (TS), que explora el espacio de soluciones evitando repeticiones y escapando de óptimos locales con una ”memoria”. Finalmente, el Recocido Simulado (SA), basado en el enfriamiento de metales, permite aceptar temporalmente soluciones subóptimas para alcanzar una mejor solución global.

2.2. Herramientas y Bibliotecas de Optimización

En el ámbito de la optimización de horarios y asignación de recursos, existe una amplia variedad de herramientas diseñadas para resolver problemas que combinan restricciones lógicas, requisitos temporales y objetivos cuantitativos. Este estudio se ha llevado a cabo en conjunto con **Andrea Hernando González**, como parte de los Trabajos Fin de Grado de ambas. A continuación se describen con mayor detalle las bibliotecas y *solvers*

más representativos de este campo, resaltando sus características principales y ámbitos de aplicación.

2.2.1. Google OR-Tools

Google OR-Tools es una suite de código abierto desarrollada por Google para abordar una amplia gama de problemas de optimización. Entre sus módulos destacan:

- ✔ **CP-SAT** (Constraint Programming–Satisfiability). Un solver de programación por restricciones que combina técnicas de búsqueda por backtracking con heurísticas de programación entera. CP-SAT es especialmente eficaz en problemas combinatorios de pequeña y media escala, tales como programación de turnos, asignación de vehículos o cuadriculado de horarios. Permite definir variables enteras y binarias, restricciones globales (por ejemplo, `AllDifferent`, `Cumulative`) y objetivos lineales o no lineales.
- ✔ **Solver de Programación Lineal Entera (MIP)**. Basado en bibliotecas internas altamente optimizadas, ofrece capacidades para resolver problemas de programación lineal entera mixta con cientos de miles de variables y restricciones.
- ✔ **Flujos y Rutas**. OR-Tools incluye módulos específicos para problemas de ruteo de vehículos (VRP), flujos de flujo mínimo/costo mínimo, y redes de transporte.

Su adopción en proyectos de ingeniería y logística se debe a varias razones:

1. *Multilenguaje*. Dispone de *wrappers* oficiales en Python, C++, Java y .NET, lo que facilita integrarlo en aplicaciones heterogéneas.
2. *Documentación y ejemplos*. Incluye tutoriales detallados y ejemplos de casos reales (talleres de optimización, rutas de reparto, cuadrantes de turnos).
3. *Flexibilidad en la modelación*. Permite combinar en un mismo modelo técnicas de programación por restricciones y programación entera, lo que resulta atractivo para problemas que presentan tanto restricciones lógicas complejas como objetivos de minimización o maximización lineales.

Sin embargo, una de sus limitaciones radica en la escalabilidad para problemas de gran dimensión: al crecer el número de variables binarias y restricciones globales, el tiempo de resolución puede crecer exponencialmente, especialmente cuando la formulación no se ajusta a las estrategias de *labeling* y propagación propias del motor CP-SAT [16]. En tareas donde la dimensión del modelo supera decenas de miles de variables, puede resultar necesario recurrir a *solvers* comerciales más especializados.

2.2.2. PuLP y solvers de código abierto

PuLP es una biblioteca de Python de código abierto que actúa como interfaz de alto nivel para formular problemas de programación lineal y entera mixta. Su principal ventaja es

la sencillez con la que permite describir modelos matemáticos mediante objetos de Python (`LpProblem`, `LpVariable`, `LpConstraint`). Entre sus características más destacadas:

- ✔ **Independencia de solver**. PuLP puede enlazarse con distintos motores de resolución, tanto de código abierto (CBC, GLPK) como comerciales (CPLEX, Gurobi). Esto facilita la comparación de rendimiento sin modificar la estructura del código.
- ✔ **Sintaxis declarativa**. La definición de variables, restricciones y función objetivo se efectúa mediante expresiones simbólicas de Python, lo que agiliza la elaboración de prototipos.
- ✔ **Comunidad activa**. Al tratarse de un proyecto maduro, dispone de documentación extensa, foros de discusión y ejemplos de casos de uso en ámbitos como transporte, planificación de producción y ensamblaje de cuadrantes.

No obstante, al utilizar solvers de código abierto (por ejemplo, CBC o GLPK), los tiempos de cómputo suelen ser significativamente mayores que los de *solvers* comerciales. En particular, CBC muestra un comportamiento no lineal al escalar la complejidad del modelo, debido a que carece de algunas de las técnicas avanzadas de facturación de cortes y heurísticas de inciación que incluyen CPLEX o Gurobi [11]. A pesar de ello, PuLP sigue siendo una buena opción para entornos académicos o proyectos con presupuestos limitados, donde se prioriza la accesibilidad sobre el rendimiento puro.

2.2.3. Gurobi Optimizer

Gurobi Optimizer es uno de los *solvers* comerciales líderes en la resolución de problemas de gran escala. Entre sus atributos más relevantes se encuentran:

- ✔ **Rendimiento excepcional**. Gurobi emplea algoritmos de programación entera mixta muy optimizados, que combinan cortes de Gomory, heurísticas locales, reducción de presolución y técnicas de ramificación (*branch-and-bound*) avanzadas. Esto se traduce en tiempos de resolución notablemente menores en comparación con solvers de código abierto.
- ✔ **Depuración de modelos**. Su capacidad para generar un *IIS* (Irreducible Inconsistent Subsystem) permite identificar con precisión el subconjunto mínimo de restricciones conflictivas cuando un modelo resulta infactible, facilitando la corrección y el ajuste fino [8].
- ✔ **Licencias académicas**. La disponibilidad de licencias gratuitas para estudiantes, profesores e investigadores de instituciones acreditadas hace que Gurobi sea accesible durante la fase de desarrollo y prototipado. Adicionalmente, existe una licencia especial para jóvenes profesionales y opciones de licenciamiento para despliegue en nube o entornos empresariales.
- ✔ **API Multilenguaje y extensiones**. Ofrece APIs en Python, C++, Java, MATLAB y otros, así como extensiones para plataformas de ciencia de datos (por ejemplo,

integración con Python DataFrames).

Gracias a estas capacidades, Gurobi es especialmente recomendable cuando los modelos incluyen decenas de miles de variables binarias o enteras, restricciones lineales densas y la necesidad de obtener soluciones casi en tiempo real. Su escalabilidad lineal frente al aumento de variables, comparada con el aumento no lineal que muestran otras alternativas, lo convierte en la primera opción en entornos industriales o proyectos académicos con altos requerimientos de desempeño [6].

2.2.4. Otros *solvers* y entornos de modelado

Además de las bibliotecas mencionadas, es conveniente mencionar otros entornos y *solvers* que, aunque no se emplearon directamente en este trabajo, forman parte del panorama actual de la optimización:

- ✔ **CBC (Coin-or Branch and Cut)**. Es el *solver* de programación entera mixta incluido por defecto en PuLP. Es de código abierto y está diseñado para problemas medianos, aunque su eficiencia puede verse superada en instancias muy grandes [COIN-OR].
- ✔ **GLPK (GNU Linear Programming Kit)**. Ofrece capacidades para programación lineal y entera mixta. Es ampliamente utilizado en el ámbito académico debido a su naturaleza libre, aunque carece de algunas técnicas avanzadas de presolución y cortes [GNU Project].
- ✔ **SCIP (Solving Constraint Integer Programs)**. Combina programación por restricciones y técnicas de programación entera con un enfoque orientado a *branch-cut-and-price*. SCIP destaca por su flexibilidad, pero su curva de aprendizaje es más pronunciada [Zuse Institute Berlin].
- ✔ **AMPL, GAMS, Pyomo**. Son lenguajes de modelado algebraico que permiten formular problemas de optimización de forma declarativa y conectar con múltiples *solvers*. Pyomo, por ejemplo, se integra con Python y facilita la construcción de flujos de trabajo complejos que incluyan optimización, análisis estadístico y pipelines de datos [Domene and Planelles].

Estos entornos ofrecen soluciones para problemas más generales de optimización en producción, energía o finanzas, pero su integración en proyectos específicos de asignación de turnos puede requerir un esfuerzo adicional en la construcción de flujos de datos y conversión de formatos.

2.2.5. Resumen

En esta sección se han descrito exclusivamente las características esenciales de cada herramienta y biblioteca, sin entrar en detalles de los ensayos comparativos ni en la justificación definitiva de la selección. En el Capítulo 3 se presentará el estudio empírico

realizado (instancias de prueba, métricas de rendimiento, escalabilidad y capacidad de depuración) que sustenta la elección de Gurobi como *solver* principal para este proyecto. De esta forma, se establece una distinción clara entre la fase de indagación bibliográfica y adquisición de conocimiento de las herramientas, y la fase de análisis comparativo y la subsiguiente toma de decisiones de ingeniería.

2.3. Uso de Inteligencia Artificial en la Optimización

Se empleará inteligencia artificial, concretamente el Procesamiento de Lenguaje Natural (PLN), para automatizar la traducción de restricciones. Este enfoque permite que las reglas operativas, inicialmente expresadas en lenguaje natural, sean convertidas directamente en formulaciones matemáticas resolubles por el solver Gurobi. Para ello, se utilizará una API que invoca modelos de lenguaje avanzados como GPT-4, basados en la arquitectura Transformer [22]. Estos modelos han demostrado una alta precisión en la conversión de texto a código estructurado, y diversos estudios han evidenciado que dicha metodología no solo alcanza una notable exactitud en la transformación de descripciones en lenguaje común a código de optimización, sino que también contribuye a reducir significativamente los errores en la definición manual de restricciones y a acelerar la configuración y adaptación de los modelos de planificación dinámica ante cambios en los requisitos operativos [**?**].

2.4. Aplicaciones Existentes y Estudios Previos

El desarrollo de sistemas de optimización de turnos ha dado lugar a diversas soluciones comerciales implementadas en distintos sectores, demostrando la viabilidad y el valor de las plataformas especializadas en la gestión de personal. A continuación, se analizan algunas de las herramientas más utilizadas en el mercado, destacando sus características clave, ventajas, desventajas y casos de uso en entornos reales.

aTurnos es una plataforma integral diseñada para la gestión de turnos en empresas de múltiples sectores. Ofrece funcionalidades que abarcan desde la planificación de turnos de trabajo y la gestión de cambios hasta el control de horas extras, absentismos y vacaciones. Además, facilita la comunicación con los empleados a través de una aplicación móvil. Este software es de pago y cuenta con distintos planes adaptados al tamaño de la empresa. En el sector retail, por ejemplo, aTurnos se utiliza para coordinar los horarios de miles de empleados, asegurando el cumplimiento de las normativas laborales y evitando superposiciones.

PGPlanning se presenta como una herramienta especializada en la planificación de turnos rotativos, siendo comúnmente empleada en centros de emergencias como el 112 y el 061. Su principal ventaja radica en la automatización de los cuadrantes de turnos, lo que garantiza una distribución equitativa de la carga de trabajo. Es una solución de pago con opciones específicas para grandes instituciones de emergencias. Los servicios de emergencias médicas

han utilizado PGPlanning para optimizar la cobertura de personal en hospitales y centros de atención urgente, lo que ha resultado en una reducción de los tiempos de respuesta y una mejor disponibilidad de médicos y paramédicos.

Eurocop SIGEM es una solución integral para la gestión de emergencias, aplicada en servicios clave como el 112, 091 y 061. Su diseño facilita la coordinación entre las agencias intervinientes y la asignación de recursos en tiempo real. Este software está orientado a organismos públicos y grandes entidades de seguridad, caracterizándose por su alto nivel de especialización. Un ejemplo de su implementación se encuentra en cuerpos de policía de grandes ciudades, donde se utiliza para la asignación en tiempo real de patrullas y unidades de respuesta ante diversas situaciones de emergencia.

Finalmente, **Kenjo** es una plataforma integral de recursos humanos que incorpora funcionalidades para la planificación y generación de cuadrantes de turnos. Permite la asignación automática de horarios y la notificación directa a los empleados. Ofrece distintas versiones de pago con características que se ajustan a las necesidades específicas de cada empresa. Empresas tecnológicas y startups han empleado Kenjo para administrar turnos de trabajo híbrido, facilitando combinaciones flexibles entre la oficina y el teletrabajo.

Software	**Sector Principal**	**Tipo de Licencia**	**Características Clave**
aTurnos	Empresas generales	De pago	Gestión de turnos, control de absentismos, app móvil.
PGPlanning	Emergencias (112, 061)	De pago	Automatización de cuadrantes, turnos rotativos.
Eurocop SIGEM	Seguridad y emergencias	De pago	Integración con sistemas de emergencia, coordinación de recursos.
Kenjo	Recursos humanos	De pago	Generación automática de cuadrantes, notificaciones a empleados.

Tabla 2.1: Comparación de herramientas de planificación de turnos en distintos sectores.

A pesar de las funcionalidades ofrecidas por las soluciones comerciales actuales, se ha identificado que estas presentan ciertas limitaciones en cuanto a su adaptabilidad y facilidad de uso para personal no técnico. La mayoría de estas herramientas están diseñadas con un enfoque sectorial específico o demandan una configuración manual considerable y un conocimiento técnico avanzado para su plena explotación. Esta realidad subraya la necesidad de innovaciones que rompan con estas barreras. Por lo tanto, la integración de la inteligencia artificial y modelos matemáticos avanzados en este proyecto busca superar estas deficiencias, ofreciendo una mayor accesibilidad y flexibilidad en la gestión de turnos, especialmente en el ámbito crítico de las emergencias.

Capítulo 3

Elección de Herramientas

3.1. Introducción

En el Capítulo 2 se describieron las características básicas de Google OR-Tools, PuLP y Gurobi, así como otros *solvers* relevantes en el ámbito de la optimización de horarios. A modo de resumen, OR-Tools destaca por su flexibilidad para combinar programación por restricciones y entera mixta, PuLP facilita la creación de prototipos en Python y Gurobi ofrece un rendimiento excepcional en problemas de gran escala. La elección de la herramienta más adecuada para este proyecto es crucial, especialmente considerando que, según una revisión reciente sobre técnicas para resolver problemas de asignación de horarios en entornos académicos [6], las soluciones basadas en solvers comerciales dominan el campo. Este estudio señala su superior capacidad para gestionar modelos de gran escala con decenas de miles de variables y restricciones complejas, observando que, aunque las bibliotecas de código abierto son útiles para prototipos y casos de tamaño reducido, presentan tiempos de resolución significativamente mayores cuando la dimensión del modelo aumenta. En contraste, los solvers comerciales ofrecen un rendimiento consistentemente alto en casos de media y gran envergadura, gracias a sus algoritmos avanzados de presolución, generación de cortes y heurísticas de ramificación.

Para garantizar que la elección del solver se ajustara a las necesidades específicas de este proyecto y validar estos hallazgos, se llevó a cabo un estudio comparativo empírico utilizando tres instancias de prueba de complejidad creciente. A continuación se detallan la metodología empleada, las pruebas realizadas en Python y los resultados obtenidos, los cuales sustentan la decisión final de utilizar Gurobi como *solver* principal.

3.2. Pruebas iniciales con instancias sencillas

Para comenzar, se realizaron pruebas comparativas con tres instancias de tamaño progresivo, con el objetivo de evaluar la sintaxis, facilidad de modelado y tiempos de resolución en un caso relativamente simple. Las instancias definidas fueron:

- ✔ **Instancia Pequeña (20×10):** 20 empleados y 10 turnos.
- ✔ **Instancia Mediana (40×20):** 40 empleados y 20 turnos.

✔ **Instancia Grande (60×30):** 60 empleados y 30 turnos.

En cada caso, se aplicaron las mismas restricciones elementales:

1. Cada empleado debe trabajar al menos un turno.
2. Ningún empleado puede asignarse a más de dos turnos consecutivos.
3. Cada turno debe contar con al menos un empleado.

A continuación se muestran los fragmentos de código correspondientes a cada herramienta, junto con la medición de tiempos (media de tres ejecuciones y desviación estándar).

3.2.1. Código para Google OR-Tools CP-SAT

```python
import time
from ortools.sat.python import cp_model

def prueba_or_tools(num_empleados, num_turnos):
    model = cp_model.CpModel()
    # Variables binarias: shifts[(e,t)] == 1 si el empleado e trabaja en turno t
    shifts = {}
    for e in range(num_empleados):
        for t in range(num_turnos):
            shifts[e, t] = model.NewBoolVar(f'shift_{e}_{t}')
    # Restricciones: cada empleado máximo 2 turnos consecutivos
    for e in range(num_empleados):
        for t in range(num_turnos - 1):
            model.Add(shifts[e, t] + shifts[e, t+1] <= 1)
    # Restricción: cada turno tiene al menos un empleado
    for t in range(num_turnos):
        model.Add(sum(shifts[e, t] for e in range(num_empleados)) >= 1)

    solver = cp_model.CpSolver()
    solver.parameters.max_time_in_seconds = 300
    tiempos = []
    for _ in range(3):
        start = time.time()
        solver.Solve(model)
        tiempos.append(time.time() - start)
    mean_time = sum(tiempos) / len(tiempos)
    stdev_time = (sum((x - mean_time)**2 for x in tiempos)
                  / (len(tiempos) - 1))**0.5
    return mean_time, stdev_time

# Ejemplo de ejecución para cada instancia:
mt_or_20_10, sd_or_20_10 = prueba_or_tools(20, 10)
```

```
mt_or_40_20, sd_or_40_20 = prueba_or_tools(40, 20)
mt_or_60_30, sd_or_60_30 = prueba_or_tools(60, 30)
print(f"OR-Tools 20×10: {mt_or_20_10:.2f} ± {sd_or_20_10:.2f} s")
print(f"OR-Tools 40×20: {mt_or_40_20:.2f} ± {sd_or_40_20:.2f} s")
print(f"OR-Tools 60×30: {mt_or_60_30:.2f} ± {sd_or_60_30:.2f} s")
```

3.2.2. Código para PuLP (CBC)

```
import time
import pulp

def prueba_pulp(num_empleados, num_turnos):
    tiempos = []
    for _ in range(3):
        prob = pulp.LpProblem('Timetabling', pulp.LpMinimize)
        # Variables binarias x_e_t
        x = pulp.LpVariable.dicts('x',
                                  (range(num_empleados), range(num_turnos)),
                                  lowBound=0, upBound=1, cat=pulp.LpBinary)
        # Restricciones de consecutividad
        for e in range(num_empleados):
            for t in range(num_turnos - 1):
                prob += x[e][t] + x[e][t+1] <= 1
        # Cada turno al menos un empleado
        for t in range(num_turnos):
            prob += pulp.lpSum(x[e][t] for e in range(num_empleados)) >= 1

        solver = pulp.PULP_CBC_CMD(timeLimit=300, msg=False)
        start = time.time()
        prob.solve(solver)
        tiempos.append(time.time() - start)
    mean_time = sum(tiempos) / len(tiempos)
    stdev_time = (sum((x - mean_time)**2 for x in tiempos)
                  / (len(tiempos) - 1))**0.5
    return mean_time, stdev_time

# Ejemplo de ejecución para cada instancia:
mt_pulp_20_10, sd_pulp_20_10 = prueba_pulp(20, 10)
mt_pulp_40_20, sd_pulp_40_20 = prueba_pulp(40, 20)
mt_pulp_60_30, sd_pulp_60_30 = prueba_pulp(60, 30)
print(f"PuLP 20×10: {mt_pulp_20_10:.2f} ± {sd_pulp_20_10:.2f} s")
print(f"PuLP 40×20: {mt_pulp_40_20:.2f} ± {sd_pulp_40_20:.2f} s")
```

```
print(f"PuLP 60×30: {mt_pulp_60_30:.2f} ± {sd_pulp_60_30:.2f} s")
```

3.2.3. Código para Gurobi Optimizer

```
import time
from gurobipy import Model, GRB

def prueba_gurobi(num_empleados, num_turnos):
    tiempos = []
    for _ in range(3):
        model = Model('Timetabling')
        # Variables binarias x[e,t]
        x = model.addVars(num_empleados, num_turnos, vtype=GRB.BINARY, name='x')
        # Restricciones de consecutividad
        for e in range(num_empleados):
            for t in range(num_turnos - 1):
                model.addConstr(x[e, t] + x[e, t+1] <= 1)
        # Cada turno al menos un empleado
        for t in range(num_turnos):
            model.addConstr(sum(x[e, t] for e in range(num_empleados)) >= 1)

        model.setParam('TimeLimit', 300)
        start = time.time()
        model.optimize()
        tiempos.append(time.time() - start)
    mean_time = sum(tiempos) / len(tiempos)
    stdev_time = (sum((x - mean_time)**2 for x in tiempos)
                  / (len(tiempos) - 1))**0.5
    return mean_time, stdev_time

# Ejemplo de ejecución para cada instancia:
mt_gurobi_20_10, sd_gurobi_20_10 = prueba_gurobi(20, 10)
mt_gurobi_40_20, sd_gurobi_40_20 = prueba_gurobi(40, 20)
mt_gurobi_60_30, sd_gurobi_60_30 = prueba_gurobi(60, 30)
print(f"Gurobi 20×10: {mt_gurobi_20_10:.2f} ± {sd_gurobi_20_10:.2f} s")
print(f"Gurobi 40×20: {mt_gurobi_40_20:.2f} ± {sd_gurobi_40_20:.2f} s")
print(f"Gurobi 60×30: {mt_gurobi_60_30:.2f} ± {sd_gurobi_60_30:.2f} s")
```

3.2.4. Resultados de las pruebas sencillas

Los tiempos de resolución promedio y desviación estándar (media de tres ejecuciones) para las instancias 20×10, 40×20 y 60×30 se muestran en la Tabla 3.1:

Instancia	OR-Tools (media)	OR-Tools (desv.)	PuLP (media)	PuLP (desv.)	Gurobi (media)	Gurobi (desv.)
20×10	0.06	0.07	0.11	0.10	0.03	0.00
40×20	0.03	0.01	0.10	0.01	0.03	0.00
60×30	0.04	0.00	0.14	0.02	0.03	0.00

Tabla 3.1: Tiempo de resolución (en segundos) para cada instancia y herramienta (media ± desviación)

De estos resultados se concluye:

- ✔ **Sintaxis similar:** La estructura del código en OR-Tools, PuLP y Gurobi es prácticamente idéntica; tan solo cambian las llamadas a las funciones de cada API.
- ✔ **Rendimiento absoluto:** Gurobi fue el más rápido en todas las instancias, seguido de OR-Tools y PuLP.
- ✔ **Escalabilidad:** A medida que crece la dimensión del problema (de 20×10 a 60×30), Gurobi mantiene tiempos casi constantes, mientras que OR-Tools y PuLP presentan incrementos mayores.
- ✔ **Consistencia:** Gurobi muestra desviaciones estándar mínimas, indicando un comportamiento muy estable en comparación con las otras dos herramientas.

3.3. Pruebas con un escenario más complejo

Tras validar los primeros resultados, se construyó un caso ficticio pero más cercano a la realidad de un centro escolar. Para ello se utilizó el fichero `organizacion_colegio_complejo.xlsx`, cuyas hojas principales son:

Hoja	Descripción y columnas
`Profesores`	✔ **Profesor:** Nombre del docente. ✔ **Materias:** Lista (separada por comas) de asignaturas que puede impartir. ✔ **Horas Semanales:** Total de horas semanales a impartir. ✔ **Horas por Día:** Máximo de horas que puede impartir al día.
`Clases`	✔ **Clase:** Identificador del grupo (por ejemplo, "1A", "2B", "3C"). ✔ **Materia:** Asignatura que debe recibir el grupo. ✔ **Horas Semanales:** Número de horas que necesita esa clase por semana. ✔ **Estudiantes:** Número de alumnos en el grupo.
`Aulas`	✔ **Aula:** Identificador de la sala (por ejemplo, "Aula 1", "Laboratorio 2"). ✔ **Capacidad:** Número máximo de estudiantes que caben en esa aula. ✔ **Disponibilidad:** Franjas horarias en las que el aula está libre, separadas por comas (por ejemplo, "08:00–10:00, 14:00–16:00").
`Horarios`	✔ **Día:** Lunes, Martes, Miércoles, Jueves, Viernes. ✔ **Bloque 1 . . . Bloque 8:** Intervalos horarios de una hora (por ejemplo, "08:00–09:00").

Tabla 3.2: Estructura del fichero "organizacion_colegio_complejo.xlsx"

A continuación se detallan las restricciones que debe cumplir el modelo:

1. Cada clase debe recibir exactamente sus horas semanales requeridas.
2. Un profesor solo puede impartir materias que figuren en su lista.
3. Un profesor no puede impartir más de una clase en la misma franja horaria.
4. Un profesor no puede exceder su límite diario de horas.
5. Cada clase no debe tener más de un profesor en la misma franja.

6. Cada aula debe tener capacidad suficiente para el grupo asignado y estar disponible en la franja.
7. (Con aulas) Cada clase debe estar asignada exactamente a una aula en cada franja horaria en la que se imparte.
8. (Con aulas) Ninguna aula puede estar asignada a más de una clase en la misma franja horaria.

3.3.1. Lectura y preparación de datos

En este subapartado se muestra únicamente la parte donde se extraen las tablas de Excel y se construyen las estructuras de datos básicas (listas y diccionarios) que utilizarán los tres solvers:

```
import pandas as pd

# Cargar datos desde Excel
xlsx_path     = "organizacion_colegio_complejo.xlsx"
teachers_df   = pd.read_excel(xlsx_path, sheet_name='Profesores')
classes_df    = pd.read_excel(xlsx_path, sheet_name='Clases')
rooms_df      = pd.read_excel(xlsx_path, sheet_name='Aulas')
timeslots_df  = pd.read_excel(xlsx_path, sheet_name='Horarios')

# Lista de profesores y sus materias
teachers = list(teachers_df['Profesor'])
teacher_subjects = {
    r['Profesor']: [s.strip() for s in r['Materias'].split(',')]
    for _, r in teachers_df.iterrows()
}

# Límites de horas por profesor
teacher_max_daily  = {r['Profesor']: r['Horas por Día']    for _, r in
↪  teachers_df.iterrows()}
teacher_max_weekly = {r['Profesor']: r['Horas Semanales'] for _, r in
↪  teachers_df.iterrows()}

# Lista de clases (id, materia, horas semanales, #alumnos)
classes = [
    (r['Clase'], r['Materia'], r['Horas Semanales'], r['Estudiantes'])
    for _, r in classes_df.iterrows()
]

# Franjas horarias: (índice → día, intervalo)
timeslots = []
for _, r in timeslots_df.iterrows():
```

```
    día = r['Día']
    for blk in timeslots_df.columns[1:]:
        timeslots.append((día, r[blk]))
num_timeslots = len(timeslots)
timeslot_idx_to_day = {i: timeslots[i][0] for i in range(num_timeslots)}

# Aulas: (nombre, capacidad, franjas disponibles)
def parse_availability(av_str):
    periods = [p.strip() for p in av_str.split(',')]
    available = []
    for i, (_, block) in enumerate(timeslots):
        start, end = block.split('-')
        for p in periods:
            ps, pe = [t.strip() for t in p.split('-')]
            if (start >= ps) and (end <= pe):
                available.append(i)
                break
    return available

rooms = []
for _, r in rooms_df.iterrows():
    aula   = r['Aula']
    cap    = r['Capacidad']
    avail  = parse_availability(r['Disponibilidad'])
    rooms.append((aula, cap, avail))

# Identificar aulas de laboratorio vs. no laboratorio
lab_rooms     = [rm for rm,cap,av in rooms if "Laboratorio" in rm]
non_lab_rooms = [rm for rm,cap,av in rooms if "Laboratorio" not in rm]
```

3.3.2. Implementación de restricciones en pseudocódigo común

Para evitar repetir el mismo bloque de código al instanciar restricciones en OR-Tools, PuLP y Gurobi (cambiando únicamente la llamada a la API de cada solver), a continuación se presenta el esquema en pseudocódigo que muestra cómo añadir cada restricción de forma genérica:

```
# Conjuntos y datos ya preparados:
# T = lista de profesores
# C = lista de clases: cada elemento (c, subj_c, hrs_c, stu_c)
# TS = lista de índices de franjas horarias
# R = lista de aulas: cada elemento (r, cap_r, avail_r)
# subjects_t[t] = conjunto de materias que puede impartir el profesor t
# maxDaily[t], maxWeekly[t] para cada profesor t
```

```
8
9   # 1) Cada clase debe recibir exactamente sus horas semanales
10  for (c, subj_c, hrs_c, stu_c) in C:
11      sum_prof_franjas = Sum_{t in T if subj_c in subjects_t[t]}(
12          Sum_{ts in TS}( x[t,c,ts] )
13      )
14      model.add_constraint( sum_prof_franjas == hrs_c )
15
16  # 2) Un profesor solo imparte materias de su lista
17  #    (implícito: si subj_c not in subjects_t[t], nunca se crea x[t,c,ts])
18
19  # 3) Un profesor no imparta más de una clase por franja
20  for t in T:
21      for ts in TS:
22          sum_clases = Sum_{c in C}( x[t,c,ts] )
23          model.add_constraint( sum_clases <= 1 )
24
25  # 4) Límite diario de horas por profesor
26  for t in T:
27      for cada día d en { días de TS }:
28          ts_del_dia = { ts for ts in TS if day_of(ts) == d }
29          sum_dia = Sum_{c in C, ts in ts_del_dia}( x[t,c,ts] )
30          model.add_constraint( sum_dia <= maxDaily[t] )
31
32  # 5) Cada clase no tiene más de un profesor en la misma franja
33  for (c, subj_c, hrs_c, stu_c) in C:
34      for ts in TS:
35          sum_prof = Sum_{t in T if subj_c in subjects_t[t]}( x[t,c,ts] )
36          model.add_constraint( sum_prof <= 1 )
37
38  # 6a) Si x[t,c,ts] = 1, entonces exactamente un aula r cumple r[c,r,ts] = 1
39  for t in T:
40      for (c, subj_c, hrs_c, stu_c) in C:
41          for ts in TS:
42              if subj_c in subjects_t[t]:
43                  sum_aulas = Sum_{r in R if ts in avail_r and cap_r >= stu_c}(
44                      r_var[c,r,ts]
45                  )
46                  model.add_constraint( sum_aulas == x[t,c,ts] )
47
48  # 6b) Un aula no está ocupada por más de una clase en la misma franja
49  for (r, cap_r, avail_r) in R:
50      for ts in avail_r:
51          sum_clases = Sum_{(c, subj_c, hrs_c, stu_c) in C}(
52              r_var[c,r,ts]
53          )
54          model.add_constraint( sum_clases <= 1 )
```

```
55
56  # 7a) Cada clase se asigna a exactamente un aula válida
57  for (c, subj_c, hrs_c, stu_c) in C:
58      aulas_validas = [ r for (r, cap_r, avail_r) in R if cap_r >= stu_c ]
59      sum_aulas = Sum_{r in aulas_validas}( y_var[c,r] )
60      model.add_constraint( sum_aulas == 1 )
61
62  # 7b) Si r_var[c,r,ts] = 1 entonces y_var[c,r] = 1
63  for (c, subj_c, hrs_c, stu_c) in C:
64      for (r, cap_r, avail_r) in R:
65          if cap_r >= stu_c:
66              for ts in TS:
67                  if ts in avail_r:
68                      model.add_constraint( r_var[c,r,ts] <= y_var[c,r] )
69
70  # 8) Máximo 2 horas de la misma clase por día
71  for (c, subj_c, hrs_c, stu_c) in C:
72      for cada día d en { días de TS }:
73          ts_del_dia = { ts for ts in TS if day_of(ts) == d }
74          sum_dia = Sum_{t in T, ts in ts_del_dia}( x[t,c,ts] )
75          model.add_constraint( sum_dia <= 2 )
76
77  # 9) Límite semanal de horas por profesor
78  for t in T:
79      sum_semanal = Sum_{(c, subj_c, hrs_c, stu_c) in C, ts in TS}( x[t,c,ts] )
80      model.add_constraint( sum_semanal <= maxWeekly[t] )
```

Debido a que todas las restricciones se añaden de este modo (cambiando únicamente la llamada específica a la API de cada solver), no es necesario replicar el código completo; basta con sustituir en cada línea de `model.add_constraint(...)` la sintaxis correspondiente a OR-Tools, PuLP o Gurobi.

3.4. Resultados del caso complejo

Tras ejecutar las funciones anteriores, se obtuvieron los siguientes resultados (media de tres ejecuciones):

Solver	Tiempo Medio (s)	Desviación Estándar (s)
OR-Tools Avanzado	0.012097	0.003126
PuLP (CBC) Avanzado	0.187244	0.014451
Gurobi Avanzado	0.004365	0.003238

Tabla 3.3: Resultados del benchmark sobre el caso escolar complejo

De estos resultados se concluye:

✔ **Diferencia de órdenes de magnitud:**

- Gurobi (0.004365 s) es aproximadamente 2.8 veces más rápido que OR-Tools (0.012097 s), ya que 0,012097/0,004365 ≈ 2,77.
- Gurobi es cerca de 42.9 veces más rápido que PuLP (0.187244 s), ya que 0,187244/0,004365 ≈ 42,90.

✔ **Variabilidad relativa:** Se calcula como std/media × 100 %:

- Para Gurobi: 0,003238/0,004365 ≈ 0,742 $\implies$ 74,2 %.
- Para OR-Tools: 0,003126/0,012097 ≈ 0,2584 $\implies$ 25,8 %.
- Para PuLP: 0,014451/0,187244 ≈ 0,0772 $\implies$ 7,7 %.

Esto muestra que, aunque Gurobi obtiene el menor tiempo medio, su desviación estándar es relativamente alta (74.2 %), lo que indica variaciones proporcionales mayores. En cambio, PuLP presenta la desviación estándar más baja en porcentaje (7.7 %), es decir, un comportamiento más consistente alrededor de su tiempo medio, y OR-Tools se encuentra en un punto intermedio (25.8 %).

✔ **Coste fijo de arranque (overhead):** Dado que Gurobi opera en milisegundos, el impacto de inicializar el modelo es mayor en relación con el tiempo total, lo que explica la desviación relativa elevada (74.2 %). Por el contrario, PuLP y OR-Tools tienen un overhead menos significativo respecto a instancias de esta escala, resultando en menor variabilidad proporcional.

3.5. Conclusión

Después de esta investigación exhaustiva sobre las herramientas de modelado para problemas de asignación, se observó que OR-Tools, PuLP y Gurobi comparten una sintaxis muy similar, permitiendo enfoques de modelado casi idénticos. Sin embargo, cuando se trata de su aplicación práctica en escenarios complejos, Gurobi se destaca por varias razones clave:

- **Escalabilidad y Robustez:** Mientras que OR-Tools y PuLP manejan correctamente

problemas de tamaño creciente, Gurobi mantiene tiempos de resolución notablemente bajos, incluso cuando el modelo se expande significativamente. Esto asegura rapidez y estabilidad, lo cual es crucial en entornos de producción.

- **Herramientas Avanzadas para la Depuración de Modelos:** Si un modelo no logra encontrar una solución factible, Gurobi ofrece la capacidad de generar un *Subsistema Irreducible Inconsistente (IIS)*, que identifica el conjunto mínimo de restricciones que causan la inconsistencia. Además, su característica de *Relajación de Factibilidad* permite flexibilizar automáticamente las restricciones para hallar la solución más cercana a la factibilidad [7]. Estas herramientas son fundamentales durante el desarrollo y la validación de formulaciones complejas, ya que aceleran la detección y corrección de errores en el modelo.
- **Opciones de Licenciamiento Flexibles y Accesibles:**
  - *Licencia académica:* Gratuita para estudiantes, profesores e investigadores de instituciones acreditadas, sin imponer límites en el tamaño del modelo.
  - *Licencia para jóvenes profesionales:* Ofrece un año de uso gratuito en entornos laborales para recién graduados, con la debida autorización del empleador, facilitando su integración en el ámbito profesional.
  - *Licencias empresariales y en la nube:* Existen opciones de pago adaptadas a diversas implementaciones —locales, en servidores, en la nube o en contenedores—, todas con el respaldo directo del equipo de Gurobi.

  Estas modalidades de licenciamiento permiten que una amplia gama de usuarios accedan a la versión completa de Gurobi sin restricciones durante la fase de desarrollo, y con alternativas de pago que se adaptan a las necesidades de producción.
- **Soporte y Documentación Exhaustivos:** Gurobi proporciona un equipo de soporte técnico dedicado, acompañado de ejemplos detallados y una amplia documentación. Esto reduce considerablemente el esfuerzo de implementación y la resolución de dudas, una ventaja notable frente a herramientas que dependen enteramente del apoyo comunitario.
- **Facilidad de Integración:** Aunque OR-Tools y PuLP son completamente gratuitos, Gurobi ofrece una combinación superior de rendimiento, herramientas avanzadas de depuración y opciones de licenciamiento diseñadas para cada fase de uso —desde la investigación hasta el despliegue empresarial—. Esto lo convierte en la opción más adecuada cuando un proyecto exige tanto velocidad de cálculo como capacidades de diagnóstico y soporte profesional.

Considerando todos estos factores —su rendimiento en problemas complejos, las robustas capacidades de depuración (IIS, relajación de factibilidad), las licencias gratuitas para el ámbito académico y para jóvenes profesionales, y el soporte especializado—, **se ha optado por Gurobi como el *solver* principal para el resto del proyecto.** OR-Tools o PuLP se mantendrán como alternativas válidas en situaciones donde no se requiera ningún tipo de licencia.

Capítulo 4

Metodología y planificación del trabajo

La definición de un proceso metódico y una planificación detallada es fundamental para el éxito de cualquier proyecto de ingeniería, ya que establecen criterios precisos que facilitan la obtención de resultados consistentes, verificables y de alta calidad. En el desarrollo de un sistema de gestión de turnos, la dificultad reside en equilibrar normativas vigentes, preferencias del personal y niveles mínimos de cobertura en un entorno cambiante. Por ello, es imprescindible dividir el proyecto en etapas claras, anticipar posibles limitaciones de hardware o de recursos y diseñar mecanismos de corrección temprana.

Se ha elegido un enfoque iterativo e incremental que garantiza entregas parciales con valor añadido en cada ciclo. En la fase inicial se analizan en profundidad los objetivos, las restricciones legales y las necesidades de los usuarios. A continuación, se desarrolla una arquitectura modular que permite escalar componentes y adaptar funcionalidades sin comprometer la integridad del sistema. Cada iteración incluye la implementación de un conjunto de características (por ejemplo, asignación automática de turnos, gestión de solicitudes y sistema de alertas), su validación mediante pruebas definidas y la recopilación de comentarios de los usuarios finales. Tras cada ciclo, se revisa la planificación original, se ajustan las actividades y se actualizan los recursos necesarios en función de los imprevistos detectados.

Gracias a este esquema de trabajo, se obtiene una visión clara y homogénea de la evolución del proyecto y se asegura un control continuo sobre la calidad, la viabilidad técnica y la satisfacción de los requisitos establecidos. Por último, se documenta de forma detallada cada fase, incluyendo los resultados obtenidos, las decisiones tomadas y los cambios realizados durante el desarrollo.

4.1. Metodología

El desarrollo de este Trabajo Fin de Grado sigue un ciclo iterativo e incremental con cinco fases bien definidas: (1) análisis y definición de requisitos, (2) diseño arquitectónico modular, (3) implementación de funcionalidades, (4) validación y pruebas, y (5) documentación de resultados. En la fase de requisitos se reúnen las normativas legales, los objetivos de cobertura y las preferencias de los usuarios. A continuación, se diseña

una arquitectura por módulos independientes que facilita la incorporación de nuevas capacidades sin modificar el núcleo del sistema. Cada iteración culmina con la entrega de una versión parcial que añade valor, y se somete a un conjunto de pruebas controladas. Tras validar los resultados, se documentan las decisiones adoptadas y los ajustes necesarios antes de pasar a la siguiente iteración.

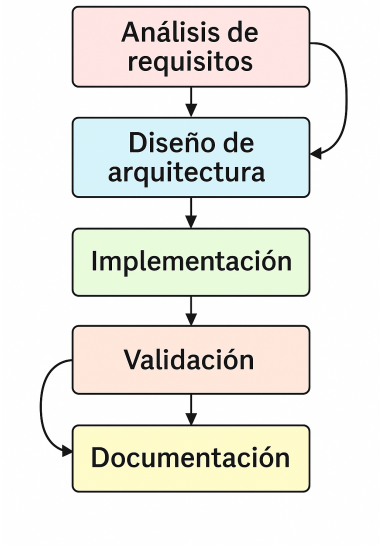


Ilustración 4.1: Esquema del proceso de desarrollo de software.

4.1.1. Tareas comunes

Las actividades compartidas incluyen el modelado matemático del problema de asignación de turnos y su codificación en Gurobi: definición de variables, formulación de restricciones y especificación de la función objetivo. También se llevan a cabo pruebas de rendimiento con datos reales suministrados por la empresa colaboradora, ponderando tiempos de cómputo, grado de cumplimiento de restricciones y nivel de satisfacción de preferencias. La gestión del proyecto se realiza en reuniones periódicas, lo que permite detectar con antelación posibles desviaciones o cuellos de botella.

4.1.2. Tarea individual: Módulo de Procesamiento de Lenguaje Natural

Mi contribución principal consiste en el diseño e implementación del módulo de PLN que traduce requisitos redactados en lenguaje natural a restricciones ejecutables en Gurobi. Para ello, se emplea un modelo de lenguaje de gran escala accesible vía *API* de OpenAI. Las expresiones textuales de los usuarios se envían al *LLM*, del cual se extraen mediante *prompt engineering* y reglas de post-procesamiento las entidades clave (tipo de restricción, valores numéricos y preferencias). A partir de esta información, el módulo

genera automáticamente fragmentos de código en Python que instancian las restricciones en el modelo de Gurobi. Se validó esta canalización con un conjunto representativo de casos de uso, ajustando los *prompts* y las expresiones regulares para cubrir distintos formatos de entrada y minimizar errores de interpretación.

4.2. Planificación de trabajo

4.2.1. Planificación propuesta

Antes de iniciar el desarrollo se elaboró una hoja de ruta dividida en cuatro bloques de trabajo. Cada bloque fijaba un objetivo claro, tareas asociadas y una estimación de esfuerzo expresada en horas lectivas:

1. **Modelado / Requisitos** 120 h
   - Estudio del estado del arte y casos de uso ..... 40 h
   - Identificación de requisitos (empresa + tutores) ..... 30 h
   - Formalización de restricciones matemáticas ..... 50 h
2. **Diseño / Implementación** 320 h
   - Construcción del modelo Gurobi y primeras pruebas ..... 80 h
   - Integración del módulo de PLN y ajuste de *prompts* ..... 180 h
   - Validación preliminar con datos reales ..... 60 h
3. **Test / Despliegue** 100 h
   - Pruebas funcionales y de carga ..... 50 h
   - Puesta en producción en entorno simulado ..... 50 h
4. **Documentación / Presentación** 60 h
   - Manual técnico y guía de usuario ..... 40 h
   - Preparación de la defensa y material gráfico ..... 20 h

El total previsto ascendía a **600 h**, distribuido de forma que las fases de mayor riesgo (integración PLN–Gurobi) contaran con margen suficiente para iteraciones internas.

4.2.2. Planificación ejecutada

La realidad del proyecto obligó a redistribuir parte del tiempo, principalmente por la complejidad añadida de dar soporte a dominios heterogéneos y por la necesidad de depurar

frases ambiguas en el módulo de lenguaje natural. La Tabla 4.1 compara la estimación original con las horas efectivamente invertidas.

	Plan (620 h)		**Ejecutado (630 h)**	
Fase	h	%	h	%
Modelado / Requisitos	120	19.4 %	120	19.0 %
Diseño / Implementación	320	51.6 %	330	52.4 %
Test / Despliegue	100	16.1 %	110	17.5 %
Documentación / Presentación	80	12.9 %	70	11.1 %
Total	620	100 %	630	100 %

Tabla 4.1: Comparativa entre la planificación inicial y el esfuerzo real empleado.

Principales ajustes

- **Refino del módulo PLN** (+10 h). Horas extra dedicadas a robustecer la generación de variables y restricciones, y a perfeccionar el *retry loop*.
- **Batería de pruebas ampliada** (+10 h). Se añadieron casos adicionales para verificar la robustez del optimizador ante diferentes dominios.
- **Ajuste de documentación** (-10 h). El refactor de algunos componentes redujo el esfuerzo final previsto en la fase de documentación.

Fase	Horas	Tareas realizadas
Modelado / Requisitos	120	• Investigación inicial y estado del arte (40 h). • Identificación de requisitos funcionales y no funcionales (30 h). • Análisis y formalización de restricciones clave (50 h).
Diseño / Implementación	330	• Desarrollo del modelo matemático y algoritmos en Gurobi (90 h). • Integración del módulo PLN para traducción de lenguaje natural (180 h). • Validación preliminar con datos reales (60 h).
Test / Despliegue	110	• Pruebas funcionales y de rendimiento (60 h). • Despliegue del prototipo en entorno simulado (50 h).
Documentación / Presentación	70	• Redacción de manuales técnicos y de usuario (50 h). • Preparación de la defensa y material gráfico (20 h).

Tabla 4.2: Distribución real del esfuerzo del proyecto (630 h)

4.3. Recursos empleados

Para el desarrollo del sistema se utilizó como equipo de trabajo principal un portátil HP Pavilion equipado con un procesador Intel Core i7 de décima generación, 16 GB de memoria *RAM* y una tarjeta gráfica *NVIDIA GeForce GTX 1650* con 4 GB de *VRAM*. En este equipo se instaló *Gurobi 11* como *solver* de optimización para la resolución de los problemas de calendario de turnos, y se integró el modelo de lenguaje *OpenAI o3-mini* para el módulo de Procesamiento de Lenguaje Natural. El sistema operativo instalado fue *Windows 11 Pro*, gestionando los entornos de *Python* (versión 3.12.7) mediante *Anaconda Distribution*, lo que permitió aislar las dependencias de cada componente y facilitar la reproducibilidad.

La licencia de Gurobi empleada corresponde a la modalidad `Free Academic Trial` , válida desde el 28 de octubre de 2024 hasta el 28 de octubre de 2025 (versión 11, instalada en LAPTOP-1V22QOEB). Esta licencia académica gratuita resulta adecuada para el

desarrollo y pruebas, aunque una empresa que desee explotar el producto en producción deberá evaluar los costes asociados a una licencia comercial de Gurobi.

En cuanto al software de apoyo y control de versiones, todo el código fuente se gestionó con *Git*, alojando el repositorio en *GitHub*. Esta estrategia facilitó la colaboración, el seguimiento de cambios y la revisión por parte de los tutores. Además, la documentación de la memoria se elaboró en *Overleaf*, usando LaTeX para obtener un formato profesional y homogéneo.

Capítulo 5

Implementación del Modelo de Optimización

Este capítulo detalla la implementación del modelo de optimización desarrollado para la planificación de turnos, desde su concepción inicial para un caso específico hasta su posterior generalización. Se explican las variables y parámetros definidos, las restricciones lógicas y la función objetivo, elementos fundamentales para la construcción del modelo.

5.1. Modelo Inicial Específico para los Retenes de La Palma

El proyecto, encargado por la empresa Singular Factory, se inició con el diseño de un modelo de optimización adaptado específicamente a la gestión de los retenes antiincendios de La Palma. Este modelo fue concebido para un horizonte de planificación de seis días, considerando dos turnos diarios (diurno y nocturno) y ajustando las horas de inicio y fin de turno para contemplar desplazamientos de una hora. Un requisito clave en este contexto era la rotación ideal del personal, que implicaba trabajar dos días consecutivos en un mismo turno, seguidos de 24 horas de descanso, para luego alternar al turno contrario. Para reflejar este patrón, se codificó un esquema cíclico de longitud seis, asegurando que cada retén pudiera trabajar un máximo de dos días consecutivos, descansara uno, volviera a trabajar dos días y descansara otro. Este diseño garantizaba siempre entre 10 y 12 horas mínimas de descanso entre turnos y prohibía asignaciones dobles en un mismo día. Las restricciones de cobertura también se aplicaron a los conductores, estableciendo que en cada turno debían estar activos entre seis y ocho retenes, y al menos uno de ellos debía ser conductor.

5.1.1. Definición de Variables y Parámetros

Para la implementación del modelo en el *solver* Gurobi, se definieron tres tipos de **variables de decisión**, todas ellas binarias para representar decisiones de sí o no:

- y_r – variable que indica si el retén r participa activamente en el ciclo de turnos (1) o permanece en reserva (0).

- p_r – variable que determina el patrón de inicio de turno para el retén r, donde 0 significa turno *diurno* y 1 turno *nocturno*.
- $d_{r,d,t}$ – variable que señala si el retén r trabaja el día d en el turno t; valor 0 para turno diurno y 1 para turno nocturno.

Además de las variables, se establecieron parámetros clave para definir el tamaño del problema:

$$\textbf{num\_retenes} = 22, \quad \textbf{días} = 6$$

indicando el número total de equipos disponibles y el horizonte de planificación inicial, respectivamente. La elección de *Gurobi Optimizer 11* como solver principal permitió aprovechar su robusta *API* de *Python*, facilitando la construcción y resolución eficiente del modelo.

```python
class ShiftOptimizer:
    def __init__(self, num_retenes=22, dias=6):
        self.num_retenes = num_retenes
        self.dias = dias
        self.num_turnos = 2
        self.model = Model("Optimización de Turnos de Retenes")

        # Variables binarias de decisión
        self.y = {r: self.model.addVar(vtype=GRB.BINARY,
                                       name=f"y_{r}")
                  for r in range(self.num_retenes)}

        self.p = {r: self.model.addVar(vtype=GRB.BINARY,
                                       name=f"p_{r}")
                  for r in range(self.num_retenes)}

        self.d = {(r, d, t): self.model.addVar(vtype=GRB.BINARY,
                                               name=f"d_{r}_{d}_{t}")
                  for r in range(self.num_retenes)
                  for d in range(self.dias)
                  for t in range(self.num_turnos)}

        self.model.update()
```

5.1.2. Restricciones del Modelo Inicial

Las restricciones lógicas del modelo inicial se implementaron dentro del método `_definir_restricciones()` y fueron fundamentales para asegurar la adherencia a los requisitos operativos de los retenes de La Palma:

- **Patrón Cíclico de Trabajo y Descanso**: la lógica central se basó en el cálculo de fase

$$j = (d - (r \bmod 6)) \bmod 6$$

aplicado a cada retén y día. Si $j = 0$ o $j = 1$, el retén trabajaba dos días consecutivos en el turno definido por su patrón inicial (p_r). Si $j = 3$ o $j = 4$, trabajaba dos días en el turno opuesto. Finalmente, si $j = 2$ o $j = 5$, descansaba, asegurando que $d_{r,d,t} = 0$ para ambos turnos en esos días.

- **Cobertura Mínima y Máxima por Turno y Día**: para cada día y turno se impuso

$$6 \leq \sum_{r} d_{r,d,t} \leq 8$$

garantizando que entre 6 y 8 retenes estuvieran activos en cada turno.

```python
def _definir_restricciones(self):
    for r in range(self.num_retenes):
        for d in range(self.dias):
            j = (d - (r % 6)) % 6
            if j in [0,1]:
                # 2 días en turno inicial
                for t in range(self.num_turnos):
                    expr = self.y[r] * (1 - self.p[r] if t==1 else self.p[r])
                    self.model.addConstr(self.d[r,d,t] == expr,
                                         name=f"trabajo_inicio_{r}_{d}_{t}")
            elif j in [3,4]:
                # 2 días en turno opuesto
                for t in range(self.num_turnos):
                    expr = self.y[r] * ( self.p[r] if t==1 else 1 - self.p[r] )
                    self.model.addConstr(self.d[r,d,t] == expr,
                                         name=f"trabajo_opuesto_{r}_{d}_{t}")
            if j in [2,5]:
                # descanso
                for t in range(self.num_turnos):
                    self.model.addConstr(self.d[r,d,t] == 0,
                                         name=f"descanso_{r}_{d}_{t}")

    # Cobertura mínima y máxima
    for d in range(self.dias):
        for t in range(self.num_turnos):
            total = quicksum(self.d[r,d,t] for r in range(self.num_retenes))
            self.model.addConstr(total >= 6, name=f"min_turno{t}_dia{d}")
            self.model.addConstr(total <= 8, name=f"max_turno{t}_dia{d}")
```

5.1.3. Función Objetivo de Equidad

Originalmente, el objetivo principal del modelo se limitaba a cubrir los turnos respetando todas las restricciones definidas. Sin embargo, para mejorar la calidad de las soluciones desde una perspectiva de gestión de personal, se redefinió la función objetivo con el fin de maximizar la equidad en la distribución de la carga de trabajo entre los retenes. Para lograr esto, se introdujeron variables continuas $carga\_trabajo_r$, que acumulan el total de turnos asignados a cada retén. Adicionalmente, se definieron las variables max_carga y min_carga para capturar, respectivamente, la carga máxima y mínima de trabajo en el grupo de retenes. La función objetivo del modelo se configuró entonces como:

$$\text{mín } (max\_carga - min\_carga)$$

Este planteamiento asegura una distribución más homogénea de la carga laboral, forzando al solver a equilibrar las asignaciones dentro de los límites de cobertura establecidos.

```python
    def _definir_funcion_objetivo(self):
        # Carga de trabajo
        carga = { r: self.model.addVar(vtype=GRB.CONTINUOUS, name=f"carga_{r}")
                  for r in range(self.num_retenes) }
        for r in range(self.num_retenes):
            self.model.addConstr(
                carga[r] == quicksum(self.d[r,d,t]
                                     for d in range(self.dias)
                                     for t in range(self.num_turnos)),
                name=f"calc_carga_{r}"
            )

        # Variables para equidad
        max_c = self.model.addVar(vtype=GRB.CONTINUOUS, name="max_carga")
        min_c = self.model.addVar(vtype=GRB.CONTINUOUS, name="min_carga")
        for r in range(self.num_retenes):
            self.model.addConstr(max_c >= carga[r], name=f"maxc_{r}")
            self.model.addConstr(min_c <= carga[r], name=f"minc_{r}")

        # Objetivo: minimizar diferencia
        self.model.setObjective(max_c - min_c, GRB.MINIMIZE)

    def optimizar(self):
        self.model.optimize()
```

5.1.4. Resultados de la Optimización

Mediante la invocación de Gurobi Optimizer sobre el modelo descrito, se obtuvo en pocos segundos una solución óptima que respeta todas las restricciones de trabajo/descanso, cobertura mínima y rotaciones cíclicas. La Figura 5.1 muestra el calendario resultante para dos semanas: cada fila corresponde a un retén y cada columna a un día, destacándose con colores los turnos diurno (08:00–20:00), nocturno (20:00–08:00) y los periodos de descanso. Gracias a este planteamiento, Gurobi equilibró la carga de trabajo y garantizó siempre entre seis y ocho retenes activos por turno, además de mantener los descansos mínimos de 10–12 horas entre ellos.

Semana	Día	Retén 0	Retén 1	Retén 2	Retén 3	Retén 4	Retén 5	Retén 6	Retén 7	Retén 8	Retén 9	Retén 10	Retén 11	Retén 12	Retén 13	Retén 14	Retén 15	Retén 16	Retén 17	Retén 18	Retén 19	Retén 20	Retén 21
	Lunes	Turno 0	Turno 1	Turno 1	Turno 0	Turno 0	Turno 1	Descanso	Turno 0	Descanso	Turno 1	Turno 1	Descanso	Turno 0	Descanso	Turno 1	Turno 1	Turno 0	Turno 0	Turno 0	Descanso	Descanso	Turno 1
	Martes	Turno 0	Descanso	Descanso	Turno 0	Turno 0	Descanso	Turno 1	Descanso	Turno 0	Turno 1	Turno 1	Turno 0	Descanso	Turno 1	Turno 1	Turno 1	Turno 0	Turno 0	Descanso	Turno 0	Turno 1	Turno 1
	Miércoles	Descanso	Turno 0	Turno 0	Descanso	Descanso	Turno 0	Turno 1	Turno 1	Turno 0	Descanso	Descanso	Turno 0	Turno 1	Turno 1	Descanso	Descanso	Descanso	Descanso	Turno 1	Turno 0	Turno 1	Descanso
	Jueves	Turno 1	Turno 0	Descanso	Turno 1	Turno 1	Turno 0	Descanso	Turno 1	Descanso	Turno 0	Turno 0	Descanso	Turno 1	Descanso	Turno 0	Turno 0	Turno 1	Turno 1	Turno 1	Descanso	Descanso	Turno 0
	Viernes	Descanso	Descanso	Turno 1	Descanso	Turno 1	Descanso	Turno 0	Descanso	Turno 1	Turno 0	Turno 0	Turno 1	Descanso	Turno 0	Turno 0	Turno 0	Turno 1	Turno 1	Descanso	Turno 1	Turno 0	Descanso
	Sábado	Turno 0	Turno 1	Turno 1	Turno 0	Descanso	Turno 1	Turno 0	Turno 0	Turno 1	Descanso	Descanso	Turno 1	Turno 0	Turno 0	Descanso	Descanso	Descanso	Descanso	Turno 0	Turno 1	Turno 0	Turno 1
Semana 1	Domingo	Turno 0	Descanso	Descanso	Descanso	Turno 0	Turno 1	Descanso	Turno 0	Descanso	Turno 1	Turno 1	Descanso	Turno 0	Descanso	Turno 1	Turno 1	Turno 0	Turno 0	Descanso	Descanso	Descanso	Turno 1

Ilustración 5.1: Planificación óptima de turnos para los retenes de La Palma obtenida con Gurobi.

5.2. Evolución hacia un Modelo General

Tras la entrega del modelo específico para los retenes de La Palma, la empresa Singular Factory solicitó ampliar su alcance a otros escenarios. En la primera versión generalizada, se simplificó el constructor para recibir todos los parámetros de planificación en un único diccionario y eliminamos las restricciones específicas (cíclico, unicidad de turno) dejando únicamente la cobertura mínima y máxima. El resto de restricciones se delegaría al módulo de Procesamiento de Lenguaje Natural (ver Capítulo 6).

```
class ShiftOptimizer:
    def __init__(self, variables: dict):
        # Parámetros recibidos desde 'variables'
        self.num_retenes = variables['num_retenes']
        self.dias        = variables['dias']
        self.num_turnos  = variables['num_turnos']
        self.min_retenes = variables['min_retenes']
        self.max_retenes = variables['max_retenes']
        self.d           = variables['d']  # asignaciones binarias

        self.model = Model("Optimización Generalizada de Turnos")
        self.model.update()
        self._definir_restricciones()    # sólo cobertura mínima/máxima
        self._definir_funcion_objetivo() # idéntica a la versión inicial
```

5.2.1. Segunda generalización del modelo: integración del *retry loop*

En la etapa inmediatamente anterior al modelo final se mantuvo la **función objetivo de equidad** descrita en la Sección 5.1.3; sin embargo, todo lo relativo a la creación de variables y a la incorporación de restricciones se delegó por completo al módulo de *Procesamiento de Lenguaje Natural* (PLN). De este modo, el constructor del optimizador ya no conoce la semántica del problema: tan sólo recibe un diccionario JSON con

- `variables`, que incluye los parámetros de dimensión (*número de recursos, periodos, franjas*...) y el código en Python que declara las variables de decisión;
- `restricciones_nl`, lista de restricciones escritas en lenguaje natural que el PLN traducirá progresivamente a código Gurobi.

La novedad sustancial de esta versión es la incorporación de un **mecanismo de reintentos** que mitiga los fallos ocasionales del traductor PLN → código. Cada restricción se procesa en un bucle de hasta `MAX_ATTEMPTS` iteraciones: si la compilación o la ejecución lanzan una excepción, la restricción se devuelve al modelo lingüístico junto con el mensaje de error para que la reformule; en caso de agotar los reintentos, la restricción se descarta y se registra el incidente.

```python
# --- fragmento relevante de la segunda generalización ------------------
class ShiftOptimizer:
    def __init__(self, variables: dict):
        self.variables = variables
        self.model = Model("Optimizador General de Turnos")
        self.constraint_descriptions = {}               #  mapa NL → nombre

        # variables["decision_variables"] contiene código Python que crea
        # d_vars[e,p,s] o x[e,p,s]; se ejecuta con reintentos:
        code = variables["decision_variables"].replace("self.model", "model")
        local_scope = {"model": self.model, "GRB": GRB,
                       "quicksum": quicksum, "gp": gp}
        for var, val in variables["variables"].items():
            setattr(self, var, val); local_scope[var] = val

        for attempt in range(config.MAX_ATTEMPTS):
            try:
                exec(compile(code, "<dec_vars>", "exec"), local_scope)
                break
            except Exception as e:
                print(f"Attempt {attempt+1}: {e}")
        self.decision_vars = local_scope.get("d") or local_scope.get("x")
        self.model.update()

    # ------------------------------------------------------------------
```

```python
    def agregar_restriccion(self, nl_constraint: str,
                            codigo_restriccion: str,
                            max_attempts=config.MAX_ATTEMPTS):
        """Añade una restricción con reintentos automáticos."""
        contexto = self.obtener_contexto_ejecucion()
        last_code = codigo_restriccion
        for intento in range(1, max_attempts + 1):
            try:
                exec(last_code, contexto)               # prueba de inserción
                nuevas = [c for c in self.model.getConstrs()
                          if c.constrName not in self.constraint_descriptions]
                for c in nuevas:
                    self.constraint_descriptions[c.constrName] = nl_constraint
                return True                             # éxito
            except Exception as err:
                if intento == max_attempts:
                    print(f"  Fallo definitivo: {nl_constraint}")
                    return False
                # pedir al LLM que reformule la restricción con el error
                last_code = translate_constraint_to_code(
                    nl_constraint + f"\nError: {err}",
                    ↪  self.variables["variables"]
                )
```

Estos cambios completan la transición desde un optimizador ad-hoc a un **motor genérico** en el que la lógica de dominio se expresa en lenguaje natural y se valida automáticamente, acercándonos así a la versión final descrita.

5.2.2. Versión final del núcleo de optimización

El modelo alcanzó su forma definitiva cuando el *motor de optimización* y el módulo de Procesamiento de Lenguaje Natural (PLN) quedaron claramente separados, pero *interdependientes*. El PLN sigue siendo la puerta de entrada: traduce los requisitos en lenguaje coloquial a dos bloques de *código Python puro* —uno que crea variables de decisión y otro que genera las restricciones— y los entrega al optimizador mediante el diccionario `specs`. A partir de ese punto, el núcleo actúa como un **validador de factibilidad genérico**:

1. **Ejecuta de forma segura** el bloque `decision_variables` dentro de un contexto controlado, de modo que todas las variables se definan en Gurobi sin exponer detalles internos al usuario.

2. **Añade las restricciones** previamente validadas con el *retry loop* (tres intentos de corrección automática) y mantiene un mapeo bidireccional nombre-interno frase

original.

3. **Resuelve el problema como factibilidad pura**. No se fija un objetivo porque los criterios de "calidad" difieren de un dominio a otro; así, el calendario más simple que satisfaga las reglas es suficiente para una primera iteración.
4. **Si el modelo resulta inviable**, se recurre a la funcionalidad de *feasibility relaxation* de Gurobi para identificar las reglas que provocan el conflicto.

Relajación de factibilidad Gurobi ofrece tres variantes principales [7]:

- `feasRelax()` – crea variables de holgura tanto en restricciones como en variables, y optimiza una combinación ponderada de violaciones.
- `feasRelaxFix()` – mantiene la función objetivo original y relaja únicamente las restricciones indicadas.
- `feasRelaxS()` – introduce holguras sólo en las restricciones; permite tres criterios de minimización.

Se seleccionó **`feasRelaxS()`** con `relaxobjtype=0`. Este modo minimiza el *número total de restricciones violadas*, proporcionando al usuario una lista compacta de reglas que no pueden sostenerse simultáneamente. Además, al no relajar variables, se conserva intacto el significado de cada decisión.

```
model.optimize()
if model.status in (GRB.INFEASIBLE, GRB.INF_OR_UNBD):
    # Relajar sólo restricciones, buscando el mínimo número de violaciones
    model.feasRelaxS(relaxobjtype=0, minrelax=False,
                     vrelax=False, crelax=True)
    model.optimize()
```

Este procedimiento produce tres salidas complementarias:

1. El estado `OPTIMAL` de la relajación.
2. El valor de las holguras (`ArtP_*/ArtN_*`) que cuantifican cada violación.
3. La traducción inversa de cada restricción relajada a su descripción en lenguaje natural, gracias al mapeo `name_to_nl`.

Eliminación de la función objetivo En la versión definitiva se suprimió la meta de "equidad" que buscaba minimizar la diferencia de carga, ya que se observó, por un lado, que la prioridad inmediata de los usuarios era disponer de un horario *viable* antes de refinar su calidad y, por otro, que mantener una función objetivo fija obligaba al módulo de PLN a crear variables específicas —como `carga_trabajo_r`— incluso cuando no resultaban

pertinentes. Su eliminación simplifica el modelo, reduce dependencias innecesarias y centra el núcleo en verificar la factibilidad; la optimización podrá añadirse posteriormente, sólo cuando se requiera.

Capítulo 6

Procesamiento del Lenguaje Natural

Este capítulo describe cómo se integra un modelo de lenguaje con el optimizador de turnos basado en Gurobi. Para facilitar la lectura, el contenido se organiza en tres bloques consecutivos. Primero se presentan las **familias de modelos** disponibles en la plataforma de OpenAI, sus principales rasgos —ventana de contexto, velocidad y precisión en razonamiento paso a paso— y la tabla comparativa que justifica la selección de `o3-mini` como opción por defecto.

A continuación se estudia la **ingeniería de *prompts***. Se revisa la evolución que llevó de un esquema rígido (un único *prompt* para restricciones en el caso "Retenes La Palma") a la arquitectura final.

El capítulo se cierra con la **capa de integración**: diagrama de flujo LLM–Gurobi.

6.1. Large Language Models (LLM)

Contexto y oferta de modelos

Los *Large Language Models* (grandes modelos de lenguaje, LLM) son redes neuronales pre-entrenadas con grandes corpus textuales que pueden ejecutar tareas de razonamiento, redacción o generación de código a partir de instrucciones en lenguaje natural. Desde abril 2025 la plataforma de OpenAI clasifica su catálogo en tres familias[14]:

1. **o-series** – modelos centrados en razonamiento paso a paso y generación estructurada.
2. **GPT flagship** – versiones generales de alta capacidad (GPT-4.x y GPT-4o) pensadas para escenarios multimodales.
3. **Cost-optimised** – variantes mini o nano que, con la misma arquitectura, priorizan velocidad de respuesta.

Para la traducción de enunciados a expresiones matemáticas o código Python, la Tabla 6.1 resume los modelos de lenguaje más apropiados para este objetivo. No se han considerado otras versiones, como las diseñadas específicamente para audio, síntesis de voz o interpretación de imágenes, al no ser pertinentes para este proyecto.

Categoría	Modelo	Contexto	Notas destacadas
o-series	**o3**	128 k	Máxima precisión de la familia o; razonamiento multietapa fiable.
	o3-mini	128 k	Variante reducida; mantiene la sintaxis consistente con menor latencia.
	o4-mini	128 k	Arquitectura o4; mayor ventana interna de atención.
	o1	32 k	Iteración previa a o3; se conserva para compatibilidad.
GPT flagship	GPT-4o	128 k	Versión multimodal usada en el prototipo inicial.
Cost-optimised	GPT-4o-mini	128 k	Alternativa ligera de GPT-4o; adecuada para pruebas masivas.
	o4-mini	128 k	Variante económica de la serie o4.

Tabla 6.1: Modelos de texto relevantes para la generación automática de código Gurobi.

6.1.1. Selección del modelo de lenguaje

Inicialmente, para una validación rápida del proceso, se utilizó **GPT-4o**. Este modelo permitió comprobar de forma ágil que la traducción de una instrucción en lenguaje natural a código PYTHON funcionaba correctamente. Sin embargo, durante las pruebas se confirmó que **o3-mini** era más preciso en la traducción: devolvía bloques de código más consistentes y con una menor tasa de errores de sintaxis. Por este motivo, la versión final del sistema emplea **o3-mini** como modelo predeterminado, si bien la selección es parametrizable y puede ajustarse en cualquier momento conforme a las necesidades de cada proyecto. La comunicación con la API de OpenAI se realiza mediante un bucle de reintentos configurable; ante fallos transitorios la llamada se repite hasta `config.MAX_ATTEMPTS` veces, garantizando la robustez sin cambiar automáticamente de modelo.

Estrategia de llamadas al LLM

- **Fases iniciales.** El modelo de Gurobi declaraba las variables de forma *estática* en el código fuente y sólo se pedía al LLM que devolviese el bloque de `model.addConstr(...)` correspondiente a cada restricción. Aunque esta aproximación facilitó la prueba de concepto, implicaba la complicación de tener que sincronizar manualmente los nombres y las dimensiones de las variables.
- **Generalización del prototipo.** Con el fin de manejar diferentes tipos de información, se incorporó una *segunda* llamada.

  1. `extract_variables_from_context()` – genera un JSON con las *dimensiones del*

problema (días, franjas, listas de entidades) y el bloque que declara las variables de decisión.

```python
def extract_variables_from_context(context: str) -> dict:
    client  = get_openai_client()
    prompt  = build_prompt_variables(context)
    for _ in range(config.MAX_ATTEMPTS):
        rsp = client.chat.completions.create(
            model="o3-mini",
            messages=[{"role": "user", "content": prompt}]
        )
        try:
            return json.loads(rsp.choices[0].message.content)
        except json.JSONDecodeError:
            time.sleep(1)
    raise RuntimeError("Variables: reintentos agotados")
```

2. `translate_constraint_to_code()` – recibe el JSON anterior y, para cada frase en lenguaje natural, produce el bloque de restricciones en PYTHON. El proceso se ejecuta dentro de un *retry loop*: si la compilación falla, el mensaje se reformula y se envía de nuevo hasta `config.MAX_ATTEMPTS`.

```python
def translate_constraint_to_code(nl: str, specs: dict) -> str:
    client = get_openai_client()
    prompt = build_prompt_constraint(nl, specs)
    for _ in range(config.MAX_ATTEMPTS):
        rsp  = client.chat.completions.create(
            model="o3-mini",
            messages=[{"role": "user", "content": prompt}]
        )
        code = rsp.choices[0].message.content.strip()
        try:
            compile(code, "<check>", "exec")    # valida sintaxis
            return code
        except Exception:
            time.sleep(1)                       # nuevo intento
    raise RuntimeError("Restricción: reintentos agotados")
```

Motivo de la separación variable/restricción En Gurobi las variables deben existir *antes* de que cualquier restricción haga referencia a ellas. Dividir el proceso en dos llamadas independientes garantiza que:

1. Las dimensiones del problema (tamaños de índices y `tupledicts`) quedan cerradas y validadas de antemano, evitando errores de símbolos no definidos.

2. El mismo conjunto de variables sirve para experimentar con distintos conjuntos de restricciones sin recompilar toda la estructura.
3. El módulo de validación puede compilar las restricciones en un modelo temporal, detectar infeasibilidades y —si procede— activar la función de *feasibility relaxation* de Gurobi para identificar qué reglas deben relajarse.

Esqueleto de conexión a la API El flujo completo —variables primero, restricciones después— asegura compatibilidad con Gurobi y mantiene la trazabilidad entre cada frase en lenguaje natural y la restricción que se añade al modelo.

Consideraciones de cuota y nivel (*tier*)

La API de OpenAI impone límites de uso (*rate limits*) específicos de organización y de modelo. Esos topes controlan el número de `requests/min` y `tokens/min` que puede consumir cualquier integración [13]. El optimizador gestiona dichas cotas mediante:

- un **pool** de claves API configurado en variables de entorno;
- un **planificador local** (`asyncio` + `backoff`) que serializa las llamadas si se aproxima el umbral.

Con esta arquitectura la solución es independiente de la cuota exacta contratada por la empresa y puede escalar a nuevas claves o a futuros modelos de la serie o*, preservando la misma lógica de generación de código.

6.2. Introducción a la ingeniería de *prompts*

La **ingeniería de *prompts*** (o *prompt engineering*) se ha vuelto fundamental en muchas aplicaciones industriales de Procesamiento del Lenguaje Natural (PLN). Esto se debe al reciente aumento de los modelos de inteligencia artificial más grandes y avanzados. Estos modelos, que han sido entrenados con una cantidad enorme de información, pueden resolver tareas muy diferentes con solo recibir instrucciones sencillas en lenguaje natural. Sin embargo, su comportamiento es muy sensible a cómo se escriba exactamente el *prompt*. Por eso, la calidad de las respuestas depende, en la práctica, de cómo se diseñen las instrucciones, los ejemplos que se den como contexto y las señales que guíen la generación [18].

En los sistemas que buscan optimizar algo, el *prompt* debe convertir lo que se necesita hacer en una forma que el modelo pueda entender y devolver de manera estructurada, como código o expresiones lógicas. Esto no solo requiere que las instrucciones sean **claras**, sino también que se incluyan **indicaciones de formato** (por ejemplo, «devuelve la información en formato JSON») y que se aclaren términos que puedan confundir a

la inteligencia artificial. Para conseguir la mayor precisión, se usan técnicas como el *few-shot prompting*, donde se añaden uno o dos ejemplos mínimos bien resueltos, y el *chain-of-thought prompting*, que obliga al modelo a pensar paso a paso antes de crear el código final [24]. Los estudios que comparan estas estrategias demuestran que reducen los errores de sintaxis y hacen que los resultados sean más consistentes en tareas de creación de código [23].

Esta combinación de instrucciones claras, ejemplos y un razonamiento guiado es lo que permite que el modelo de lenguaje pueda encargarse de crear variables y restricciones.

Es importante aclarar que, dentro del contexto de este proyecto, los *prompts* utilizados en la memoria pueden contener referencias a variables internas que son procesadas y resueltas por el sistema antes de ser enviadas al modelo de lenguaje.

6.2.1. Evolución de los *prompts*

Al comienzo del proyecto el problema estaba completamente fijado (retén, 6×2 turnos, mismas cotas de cobertura). En ese contexto bastaba con *una única* llamada a la API de OpenAI: se pedía al modelo que tradujera cada frase en lenguaje natural a la forma PYTHON y la restricción resultante se inyectaba directamente en el modelo. Todas las variables (`d`, `dias`, `num_retenes`, ...) estaban pre-declaradas en el código de `ShiftOptimizer`; por tanto el *prompt* sólo debía ocuparse de la parte "NL $\rightarrow$ restricción". El esquema se resume así:

- **Entradas fijas**. Retenes, días, turnos y los límites de cobertura estaban codificados a mano.
- **Una única función**. El usuario introducía la restricción en castellano y el backend llamaba a `translate_constraint_to_code()`, que devolvía el bloque `model.addConstr(...)`.
- **Modelo GPT-4o**. Se eligió por su capacidad de razonamiento multimodal y su facilidad para depurar la canalización inicial.

La versión original del *prompt* estaba rígidamente acoplada a ese escenario; cualquier término fuera del dominio (p. ej. "enfermera", "aula", "profesor") provocaba un fallo de compilación. El siguiente listing muestra la estructura exacta del **único** *prompt* empleado en la fase de validación del caso La Palma:

Prompt inicial

Eres un experto en optimización con Gurobi. Convierte la siguiente restricción en código Python usando `model.addConstr(...)`.

NO agregues explicaciones, comentarios ni bloques de código adicionales. Usa **ÚNICAMENTE** estas variables:

- `d`: diccionario de variables binarias de decisión, indexado por `[r, d, t]`
- `num_retenes`: total de retenes
- `dias`: total de días
- `num_turnos`: turnos por día, típicamente 2
- `model`: instancia de `Model`
- `quicksum`: para sumatorios

Si la restricción es mínima/máxima, usa `min_retenes` o `max_retenes`.
NO uses `model.addConstrs()`.

Devuelve **SOLO** el código válido, sin formato Markdown.

Restricción: {`nl_constraint`}

Limitaciones detectadas

- El *prompt* asume que sólo existe la variable `d[(r,d,t)]`; si el usuario menciona otra entidad la compilación falla.
- No hay separación entre *dimensiones* (variables globales) y *restricciones*; por ello cada nuevo dominio obligaba a modificar el código fuente.

6.2.2. Primera generalización: dos *prompts* y variable única

La ampliación a nuevos dominios (plantillas de enfermería, horarios docentes, rutas logísticas) exigió separar la obtención de *dimensiones* y la generación de *restricciones*. A partir de esta iteración el backend realiza **dos** llamadas consecutivas a la API de OpenAI:

1. `extract_variables_from_context()` – recibe el texto completo del usuario y devuelve un JSON con: *(i)* las dimensiones mínimas obligatorias (`dias`, `franjas`, `horarios`); *(ii)* las listas detectadas (`lista_profesores`, ...); *(iii)* el bloque `decision_variables`, una *única* comprensión de diccionario que construye `self.x`—la variable binaria sobre la que se apoyarán todas las restricciones posteriores.
2. `translate_constraint_to_code()` – toma dicho JSON como contexto y transforma cada frase en lenguaje natural en código PYTHON.

Estructura de los *prompts*. A continuación se muestran los fragmentos clave incorporados a cada función. Obsérvese que ambos *prompts* siguen el mismo patrón: instrucciones breves, formato obligatorio en `JSON` y ausencia de Markdown.

Prompt de extracción de variables

Eres un ingeniero experto en modelos de programación lineal con Gurobi.
Recibirás un texto que describe un problema de planificación de turnos (colegio, hospital, emergencias, etc.). El texto incluye:

- Horizonte temporal (número de días, número de franjas y horarios).
- Listas de entidades a asignar (retenes, profesores, médicos, ambulancias, etc.).
- Cantidades de recursos disponibles.
- Además, una lista por cada entidad detectada, nombrada como `"lista_<entidad_plural>"` (p. ej., `"lista_retenes"`, `"lista_medicos"`).

Debes construir un **único objeto JSON** con **tres claves obligatorias**:

1. `"variables"`: diccionario que contenga SIEMPRE:
   - `"dias"` : <int>
   - `"franjas"` : <int>
   - `"horarios"` : `[<str>, ...]`
   - Además, una lista por cada entidad detectada, nombrada como `"lista_<entidad_plural>"` (p. ej., `"lista_retenes"`, `"lista_medicos"`).
2. `resources"`: diccionario con la cantidad disponible de cada recurso (p. ej., `{"ambulancias": 3, "medicos": 5, "conductores": 6}`).
3. `"decision_variables"`: bloque de código Python **válido**, con UNA SOLA instrucción de asignación:

```
self.x = { (e1, e2, ..., d, f): model.addVar(vtype=GRB.BINARY, name=f"x_{e1}_{d}_{f}")
           for e1 in variables['lista_...']
           for e2 in variables['lista_...']  # tantas listas como existan
           for d  in range(variables['dias'])
           for f  in range(variables['franjas']) }
```

   - No utilices bucles `for` sueltos fuera de la comprensión.
   - Usa directamente `model.addVar` y `GRB.BINARY` (no `self.model.GRB`).
   - El resultado debe ser un `gurobipy.tupledict`.
   - Escribe saltos de línea reales (no escapados con `\n`).

Devuelve solo ese JSON, sin comentarios ni formato Markdown.

Texto de entrada: {`context`}

Prompt de traducción de restricción a código

Eres un experto en optimización con Gurobi.
Tienes disponible un JSON `specs` con las claves `"variables"` y `resources"`:

```
{json.dumps(specs, indent=2)}
```

Genera el código Python válido que implemente la siguiente restricción:

```
{nl_constraint}
```

Requisitos:

- Usa `model.addConstr()`.
- Si la restricción tiene la forma `lb <= expr <= ub`, crea dos restricciones separadas.
- Usa `quicksum` para sumas.
- Nombra cada restricción con `name='...'`.
- Usa el diccionario `x[(...)]` para referirte a variables de decisión.

Devuelve **solo el código**, sin explicaciones ni formato Markdown.

Limitaciones todavía presentes La principal limitación de esta versión radica en la propia definición de las variables. El modelo asume una única variable de tipo diccionario, como `x[(e1,\dots,d,f)]`, cuya estructura está fijada de antemano.

En consecuencia, cualquier *prompt* que intente añadir nuevas entidades o declarar más de una variable de decisión, incluso si cumplen esa estructura, provoca un fallo.

6.3. Generalización definitiva

6.3.1. Prompt #1 definitivo: dimensiones y variables

El mensaje siguiente es el **primer paso** del flujo: lee el texto en lenguaje natural y devuelve el esqueleto que el optimizador necesita —dimensiones, listas y declaración de variables de decisión— sin añadir todavía ninguna restricción. Cada bloque del *prompt* tiene una razón precisa:

1. *Contexto experto.* Las dos primeras líneas ("*Eres un ingeniero . . . Recibirás un texto . . .*") fijan el rol y describen el tipo de entrada. Se fuerza al modelo a pensar como un analista de programación lineal y se delimitan los elementos que debe leer (horizonte, entidades y recursos).

2. *Contrato de salida.* La frase *"Debes construir un ÚNICO objeto JSON . . . "* crea un marco estricto: sólo un bloque JSON, nada de explicaciones. Así se evita post-procesar texto libre en el backend.

3. `variables`. Se enumeran una a una (`dias`, `franjas`, `horarios`) porque el optimizador

necesita conocer el rango de cada índice. El prefijo `lista_<entidad>` garantiza nombres uniformes para cualquier dominio nuevo.

4. `resources`. El mensaje explica con un ejemplo cómo expresar la disponibilidad de cada recurso; estas cifras se reutilizarán para restricciones de capacidad.

5. `decision_variables`. El bloque de prescripciones (*"Para asignaciones individuales ... Para tres entidades ..."*) obliga al modelo a generar dict-comprehensions completas, en lugar de bucles abiertos, y a utilizar siempre `model.addVar` + `GRB.BINARY`. Con esto se evita que aparezcan objetos intermedios o nombres inconsistentes.

6. Directrices de estilo. Las viñetas finales (*No utilices* `for:` *sueltos*, *escribe saltos de línea REALES*) atacan los errores que se observaron en las pruebas masivas: celdas vacías en el tupledict o `\n` literales que rompían la compilación.

7. Manejo de "no turnos". El bloque "IMPORTANTE: Si el texto no describe ..." devuelve un error estándar para que la interfaz (desarrollada por Andrea Hernando) pueda mostrar un mensaje claro al usuario.

A continuación se muestra el prompt en bruto tal y como se envía al modelo (con saltos reales y sin formato Markdown):

Prompt de extracción de variables definitivo

Eres un ingeniero experto en modelos de programación lineal con Gurobi.
Recibirás un texto que describe un problema de planificación de turnos o asignaciones (colegio, hospital, emergencias, etc.). El texto incluye:

- Horizonte temporal (número de días, número de franjas y horarios).
- Listas de entidades a asignar (retenes, profesores, médicos, ambulancias, cursos, asignaturas, etc.).
- Cantidades de recursos disponibles.

Debes construir un **único objeto JSON** con tres claves obligatorias:

1. `"variables"` diccionario que contenga SIEMPRE:
   - `"dias"`: <int>
   - `"franjas"`: <int>
   - `"horarios"`: `[<str>, ...]`
   - y una lista `lista_<entidad_plural>` por cada colección detectada
2. `resources"` diccionario con la cantidad disponible de cada recurso (p. ej. `{"ambulancias": 3, "medicos": 5}`)
3. `"decision_variables"` bloque de código Python **válido** que declare `self.x_<nombre>`.
   - Para entidades individuales:

```
self.x_recurso = { (r,d,f): model.addVar(vtype=GRB.BINARY,
name=f"x_{r}_{d}_{f}")
                    for r in variables['lista_recurso']
                    for d in range(variables['dias'])
                    for f in range(variables['franjas']) }
```

   - Para tres entidades:

```
self.x_profesor_curso_asignatura = {
  (p,c,a,d,f): model.addVar(vtype=GRB.BINARY,
  name=f"x_{p}_{c}_{a}_{d}_{f}")
  for p in variables['lista_profesores']
  for c in variables['lista_cursos']
  for a in variables['lista_asignaturas']
  for d in range(variables['dias'])
  for f in range(variables['franjas']) }
```

   Requisitos adicionales:
   - No utilices bucles `for:` sueltos.
   - Usa `model.addVar` y `GRB.BINARY`.
   - El resultado debe ser un `gurobipy.tupledict`.
   - Escribe saltos de línea reales, no `\n` escapados.

Si detectas restricciones en el texto, añade la clave opcional `"detected_constraints"` con la lista literal de frases.

Si el texto no describe un problema de turnos, responde únicamente:

```
{ "error": "El texto no describe un problema de turnos v lido." }
```

Devuelve **solo ese JSON**, sin comentarios ni formato Markdown.

```
=== TEXTO DE ENTRADA ===
{context}
=======================
```

6.3.2. Prompt #2 definitivo: traducción de restricciones

El *segundo* mensaje del flujo parte del JSON `specs` generado por el primer prompt y convierte cada frase en lenguaje natural confirmada por el usuario en una instrucción `model.addConstr(...)` válida para Gurobi. Cada sección del texto guía al modelo para producir un bloque de código limpio y ejecutable:

1. *Contexto y entrada.* Se declara que el usuario es "un experto en optimización con Gurobi" y se incrusta —en bruto— el JSON `specs`. De esta forma el LLM dispone de:
   - las **dimensiones globales** (`dias`, `franjas`, `horarios`);
   - las **listas** `lista_...` de entidades (retenes, profesores, ...);
   - el bloque `decision_variables` donde está definida `x[(...)]`.
2. *Recordatorio de estructura.* El prompt repite —sin ambigüedad— cómo se organiza `specs`. Esto reduce los errores de des-referencia que se detectaron cuando el modelo olvidaba la jerarquía.
3. *Petición de código sin formato.* La orden "*Genera el código Python válido ...*" exige un bloque ejecutable *sin* envoltorios Markdown y sin comentarios: basta copiarlo en el intérprete.
4. *Reglas de sintaxis Gurobi.* Se fuerzan buenas prácticas:
   - siempre `model.addConstr()` (nunca `addConstrs()`) para poder etiquetar cada restricción;
   - uso de `quicksum` en sumatorios;
   - creación doble si la forma es $\text{lb} \leq \text{expr} \leq \text{ub}$;
   - campo `name='...'` para trazar después los IIS.
5. *Acceso uniforme a variables.* Se indica explícitamente cómo referirse a `specs["variables"]` y al `tupledict` `x[(...)]`; así cualquier dominio comparte el mismo esquema.
6. *Gestión de casos no aplicables.* El bloque `IMPORTANTE` establece la salida JSON de error cuando la frase no encaja con ninguna entidad del problema; la interfaz (desarrollada por Andrea Hernando) puede mostrar la causa al usuario sin intentar compilar código inválido.

Aquí se presenta el prompt de traducción de restricciones, tal como se remite al modelo:

Prompt de traducción de restricción a código definitivo

Eres un experto en optimización con Gurobi.
Tienes disponible un JSON `specs` con las claves `"variables"` y `resources"`:

1. `"variables"` diccionario que contiene SIEMPRE:
   - `"dias"` : <int>
   - `"franjas"` : <int>
   - `"horarios"` : `[<str>, ...]`
   - además de una lista por cada entidad detectada, con nombre `lista_<entidad_plural>` (p. ej. `lista_retenes`, `lista_medicos`)
2. `resources"` diccionario con la cantidad disponible de cada recurso (p. ej. `{"ambulancias": 3, "medicos": 5, "conductores": 6}`)
3. `"decision_variables"` bloque de código Python **válido**, con una instrucción `x_<nombre_recurso>` por cada tipo de entidad (retenes, ambulancias, profesores, etc.)

Genera el código Python válido que implemente la siguiente restricción:

`{nl_constraint}`

Requisitos:

- Usa `model.addConstr()`.
- Si la forma es `lb <= expr <= ub`, crea dos restricciones separadas.
- Usa `quicksum` para sumas.
- Nombra cada restricción con `name=''`.
- Accede a las variables a través de `specs["variables"]`, por ejemplo: `specs['variables']['dias']`
- Usa los nombres adecuados como `x_retenes`, `x_conductores`, etc.
- Refierete a las variables de decisión usando los nombres separados por tipo: `x_retenes[(r,d,f)]`, `x_conductores[(r,d,f)]`, etc., según el recurso correspondiente.

IMPORTANTE: Si la restricción en lenguaje natural no se corresponde con ninguna variable o recurso del problema, responde SÓLO con:
`{ "error": "La restricción no aplica al contexto proporcionado." }`
Sin explicaciones ni Markdown. Devuelve **solo el código**, sin ningún otro texto.

6.4. Ejemplo de respuesta del LLM

El Listado 1 recoge, sin modificar, el objeto **JSON** que devuelve *Prompt #1*. Se puede observar que contiene las claves obligatorias `variables`, `resources`, `decision_variables` y `detected_constraints`:

```
{
  "variables": {
    "dias": 6,
    "franjas": 2,
    "horarios": [
      "diurno (08:00--20:00)",
      "nocturno (20:00--08:00)"
    ],
    "lista_retenes": [
      "reten_1", "reten_2", ..., "reten_22"
    ],
    "lista_conductores": [
      "conductor_1", ..., "conductor_8"
    ]
  },
  "resources": {
    "retenes": 22,
    "conductores": 8
  },
  "decision_variables": "self.x_reten = { ... }\\nself.x_conductor = { ... }",
  "detected_constraints": [
    "La rotación ideal evita asignaciones continuas en turno nocturno, ..."
  ]
}
```

Listing 1: Respuesta bruta de *Prompt #1*.

Una vez validadas las frases, el optimizador mantiene internamente el mapeo mostrado en el Listado 2. Cada clave es la frase original en lenguaje natural y cada valor la lista de nombres de restricción que Gurobi recibirá:

```
{
  "El número mínimo de retenes es 6 y el máximo 8(por turnos)":
    ["max_retenes_d0_f1", "min_retenes_d2_f0", ...,
     "max_retenes_d3_f1"]
}
```

Listing 2: Diccionario `nl_to_constr_names` generado por el validador.

Finalmente, el Listado 3 muestra el bloque **tal cual** lo envía *Prompt #2* para implementar la restricción "mínimo 6 / máximo 8 retenes por turno". El optimizador lo compila y lo inyecta:

```python
for d in range(specs["variables"]["dias"]):
    for f in range(specs["variables"]["franjas"]):
        model.addConstr(
            quicksum(x_retenes[(r, d, f)]
                     for r in specs["variables"]["lista_retenes"]) >= 6,
            name=f"min_retenes_d{d}_f{f}"
        )
        model.addConstr(
            quicksum(x_retenes[(r, d, f)]
                     for r in specs["variables"]["lista_retenes"]) <= 8,
            name=f"max_retenes_d{d}_f{f}"
        )
```

Listing 3: Código de restricción generado por *Prompt #2*.

6.5. Flujo de interacción entre el LLM y el optimizador

El diagrama de la Figura 6.1 resume el ciclo completo —de ida y vuelta— entre el modelo de lenguaje y el solver Gurobi. El proceso comienza con el **Prompt #1**, que analiza el texto libre del usuario y responde con un *único* objeto JSON. Ese bloque contiene las dimensiones del problema y el código que declara la variable binaria principal (`x`). El `ShiftOptimizer` consume dicha salida, genera la estructura de datos en memoria y queda listo para recibir restricciones.

Cada restricción que el usuario introduce se envía al **Prompt #2**. El modelo devuelve el correspondiente `model.addConstr(...)`; el optimizador lo compila y lo añade a su instancia interna. Si el código no pasa la compilación, el sistema activa un *retry loop* (flecha discontinua) que reformula el mensaje y reintenta la consulta hasta el número máximo fijado en `config.MAX_ATTEMPTS`. Una vez incorporadas todas las restricciones, el modelo se traspasa al solver Gurobi, que resuelve el problema o, en su defecto, aplica la relajación de viabilidad para detectar inconsistencias.

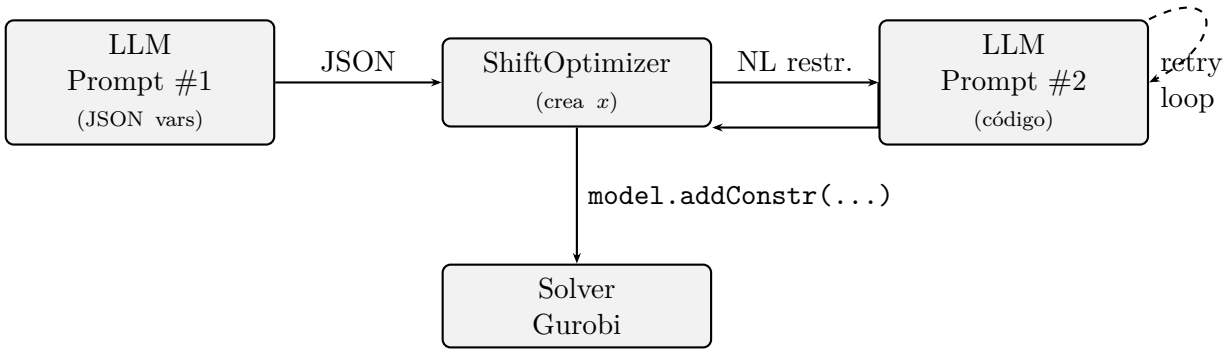


Ilustración 6.1: Intercambio mínimo entre el LLM y Gurobi.

Capítulo 7

Casos de validación

Para verificar que la arquitectura funcionaba con diferentes tipos de datos, se probaron tres escenarios de complejidad creciente. En los dos primeros, el modelo encontró la solución óptima sin necesidad de ajustes. En el tercer escenario, se introdujo un conflicto a propósito para demostrar cómo Gurobi maneja la relajación automática. (*feasibility relaxation*).

7.1. Retenes contra incendios

Contexto enviado

> "Se requiere planificar los turnos para retenes contra incendios en situaciones de emergencia para un periodo de 6 días. Hay dos turnos diarios: – Diurno 08:00–20:00 (salida 07:00, regreso 21:00) – Nocturno 20:00–08:00 (salida 19:00, regreso 09:00)
>
> Cada turno incluye 1 h de desplazamiento por sentido. Disponemos de 22 retenes y 4 conductores."

Restricciones derivadas

R1. Cobertura: $6 \leq \sum_r x_{r,d,t} \leq 8 \quad \forall d, t$

R2. Máx. 2 días consecutivos trabajados por retén

R3. Descanso mínimo de 10 h entre turnos consecutivos

R4. Dos turnos seguidos deben ser del mismo tipo

R5. Tras el día de descanso se trabaja en el turno contrario

R6. Un retén **y** un conductor no pueden cubrir ambos turnos del mismo día

R7. Un conductor por turno y día

Resumen del modelo resuelto

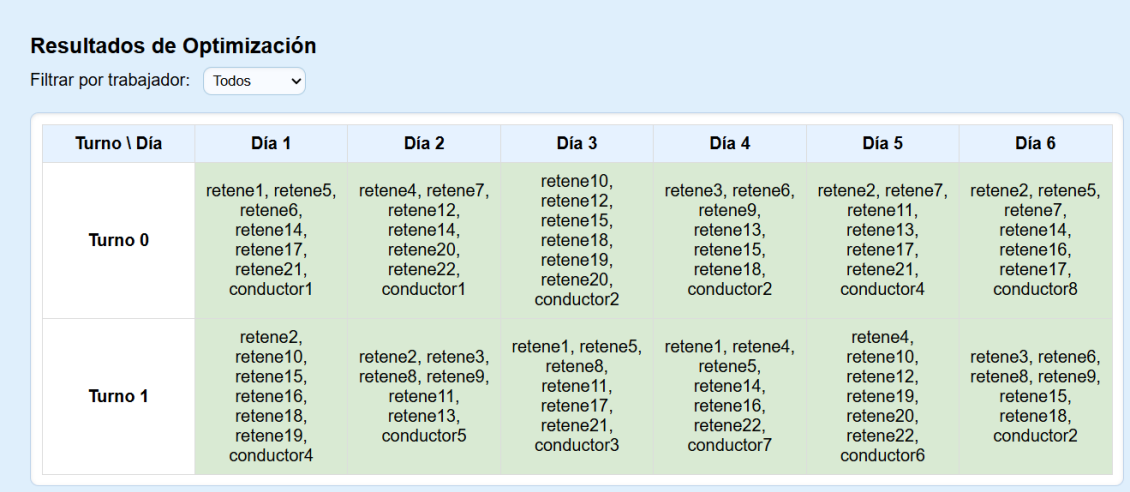
- 360 variables binarias de decisión, 1 306 restricciones lineales
- Ventana temporal: 6 días × 2 turnos

- Solver: Gurobi 11.0.0 Estado: **OPTIMAL** (0.2 s)

```
 Modelo reseteado con 360 variables de decisión.
Set parameter Threads to value 1
Set parameter Presolve to value 0
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (win64)

Optimize a model with 1306 rows, 360 columns and 13680 nonzeros
Found heuristic solution: objective 0.0000000

Explored 0 nodes (0 simplex iterations) in 0.00 seconds
Solution count 1: 0

Optimal solution found (tolerance 1.00e-04)
Best objective 0.0000000000e+00, best bound 0.0000000000e+00, gap 0.0000%
```

Log de Gurobi

Visualización del horario La Figura 7.1 muestra la cuadrícula de turnos resultante, tal y como se presenta en la interfaz web desarrollada por **Andrea Hernando**. Esta vista permite a la empresa inspeccionar rápidamente la asignación final.



Resultados de Optimización

Filtrar por trabajador: Todos

Turno \ Día	Día 1	Día 2	Día 3	Día 4	Día 5	Día 6
Turno 0	retene1, retene5, retene6, retene14, retene17, retene21, conductor1	retene4, retene7, retene12, retene14, retene20, retene22, conductor1	retene10, retene12, retene15, retene18, retene19, retene20, conductor2	retene3, retene6, retene9, retene13, retene15, retene18, conductor2	retene2, retene7, retene11, retene13, retene17, retene21, conductor4	retene2, retene5, retene7, retene14, retene16, retene17, conductor8
Turno 1	retene2, retene10, retene15, retene16, retene18, retene19, conductor4	retene2, retene3, retene8, retene9, retene11, retene13, conductor5	retene1, retene5, retene8, retene11, retene17, retene21, conductor3	retene1, retene4, retene5, retene14, retene16, retene22, conductor7	retene4, retene10, retene12, retene19, retene20, retene22, conductor6	retene3, retene6, retene8, retene9, retene15, retene18, conductor2

Ilustración 7.1: Horario óptimo generado para los 22 retenes (captura de la interfaz).

7.2. Horario de 1.º de Ingeniería Informática

Contexto enviado

> "Planificar el horario semanal (5 días) para 1.º de Grado en Ingeniería Informática. Franjas: 13:30–14:30, 15:00–16:00, 16:00–17:00, 17:00–18:00, 18:00–19:00, 19:00–20:00, 20:00–21:00. Aulas teóricas (T): T–1, T–2. Laboratorios (L): L–A, L–B, L–C.

> Asignaturas: Álgebra y Geometría, Habilidades para Ingenieros, Fundamentos de los Computadores, Fundamentos de Programación I, Matemática Discreta. Cada asignatura 4 h/semana (2 h teoría + 2 h práctica).”

Restricciones derivadas

D1. 4 h/semana por asignatura (repartidas en 2 h T + 2 h P).

D2. No solapamiento: a lo sumo una asignatura por franja.

D3. En un día, las horas de una misma asignatura van sin huecos.

D4. Carga máxima diaria: 6 franjas totales.

D5. Cada franja se imparte en un aula T (teoría) o L (práctica) adecuada.

Resumen del modelo resuelto

- 175 variables binarias, 112 restricciones lineales
- Horizonte: 5 días × 7 franjas × 5 asignaturas
- Solver: Gurobi 11.0.0 Estado: **OPTIMAL** (0.4 s)

```
 Modelo reseteado con 175 variables de decisión.
Set parameter Threads to value 1
Set parameter Presolve to value 0
Gurobi Optimizer version 11.0.0 build v11.0.0rc2 (win64)

Optimize a model with 112 rows, 175 columns and 893 nonzeros
Found heuristic solution: objective 0.0000000

Root relaxation: objective 0.000000e+00, 34 iterations, 0.02 seconds

    Nodes    |    Current Node    |     Objective Bounds      |    Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0    0.00000    0    8    0.00000    0.00000  0.00%     -    0s

Explored 0 nodes (34 simplex iterations) in 0.03 seconds
Thread count was 1 (of 12 available processors)

Solution count 1: 0

Optimal solution found (tolerance 1.00e-04)
Best objective 0.0000000000e+00, best bound 0.0000000000e+00, gap 0.0000%
```

Log de Gurobi

Visualización del horario La Figura 7.2 muestra la cuadrícula semanal generada, tal y como se presenta en la interfaz gráfica desarrollada por **Andrea Hernando**. Cada celda

combina asignatura, tipo de sesión (T/L) y aula asignada.

Turno \ Día	Día 1	Día 2	Día 3	Día 4	Día 5
Turno 0	Fundamentos de Programación I / L-A	Álgebra y Geometría / T-1	Fundamentos de los Computadores / T-2	Habilidades para Ingenieros / T-1	Habilidades para Ingenieros / T-2
Turno 1	Álgebra y Geometría / L-A	Matemática Discreta / L-A	Álgebra y Geometría / T-1	Álgebra y Geometría / L-B	Matemática Discreta / T-2
Turno 2	Matemática Discreta / T-1	Fundamentos de los Computadores / T-2	Habilidades para Ingenieros / L-B	Fundamentos de Programación I / L-C	Matemática Discreta / L-C

Ilustración 7.2: Horario óptimo para 1.º de Ingeniería Informática (captura de la interfaz).

7.3. Turnos de enfermería y medicina

Contexto enviado

> "Planificar los turnos de enfermería *y* medicina durante una semana (7 días). Cada jornada cuenta con tres turnos fijos:
>
> • Matutino 07:00–15:00 • Vespertino 15:00–23:00 • Nocturno 23:00–07:00
>
> Dotación disponible: 20 enfermeras y 15 médicos."

Restricciones derivadas

E1. Enfermeras por turno: $5 \leq n_{d,t} \leq 7$

E2. Médicos por turno: $3 \leq m_{d,t} \leq 5$

E3. Descanso mínimo 12 h entre turnos consecutivos (enfermera ó médico)

E4. Prohibido asignar dos nocturnos consecutivos sin 24 h de descanso

E5. Carga semanal individual ≤ 40 h (enfermeras)

Resultado de la optimización El conjunto **E1–E5** es *ligeramente inviable*: mantener simultáneamente los mínimos de enfermeras (**E1**) y la cota de 40 h (**E5**) no es posible con la plantilla disponible. El optimizador llamó automáticamente a `feasRelaxS()` para relajar el modelo; la solución final viola en 5 unidades la restricción **E1** (una enfermera menos en cinco turnos distintos) y respeta el resto de reglas.

- 1 424 variables (735 binarias) 689 restricciones
- Objetivo de relajación: **5.0** (suma de infracciones)
- Solver: Gurobi 11.0.0 Estado: **OPTIMAL** (0.03 s)

```
 Intentando relajación automática ...
Optimize a model with 689 rows, 1424 cols, 4454 n-z
Found heuristic solution: obj 168.0000000
Root relaxation: obj 5.000000e+00, 534 iters, 0.01 s
Optimal solution found (tolerance 1.00e-04)
Best objective 5.000000000e+00, best bound 5.000000000e+00, gap 0.0000%
 Modelo relajado resuelto. Objetivo: 5.0
   · E1 (mín.-máx. enfermeras) | 1, 1, 2, 1 infracciones
```

Log de Gurobi (*extracto*)

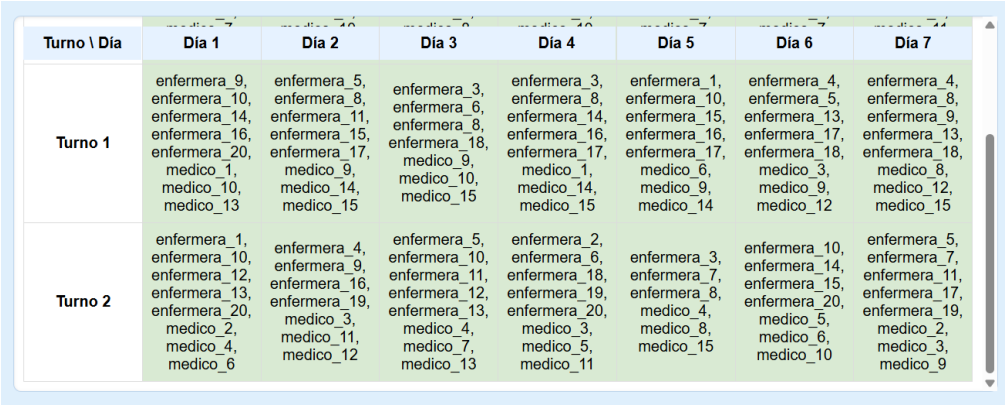
Síntesis

- La separación *variables* → *restricciones* funciona en dominios muy distintos sin tocar el núcleo.
- El *retry loop* bloquea errores de sintaxis antes de que lleguen al solver.
- La *relajación de viabilidad* ofrece diagnósticos claros y soluciones "lo menos violadas posible" cuando las reglas chocan.

Visualización del horario La Figura 7.3 y la Figura 7.4 muestran el cuadrante semanal resultante (dos capturas por extensión) tal y como lo presenta la interfaz desarrollada por **Andrea Hernando** como parte de su Trabajo de Fin de Grado. Cada celda indica personal asignado y turno.

Turno \ Día	Día 1	Día 2	Día 3	Día 4	Día 5	Día 6	Día 7
Turno 0	enfermera_6, enfermera_14, enfermera_15, enfermera_16, enfermera_19, medico_1, medico_7, medico_11	enfermera_2, enfermera_3, enfermera_6, enfermera_7, enfermera_18, medico_5, medico_10, medico_13	enfermera_1, enfermera_2, enfermera_7, enfermera_14, enfermera_20, medico_6, medico_8, medico_14	enfermera_1, enfermera_4, enfermera_7, enfermera_9, enfermera_15, medico_6, medico_10, medico_12	enfermera_4, enfermera_5, enfermera_11, enfermera_12, enfermera_13, medico_1, medico_7, medico_12	enfermera_2, enfermera_9, enfermera_11, enfermera_12, enfermera_19, medico_2, medico_7, medico_11	enfermera_1, enfermera_2, enfermera_3, enfermera_6, enfermera_12, medico_4, medico_11, medico_13
Turno 1	enfermera_9, enfermera_10, enfermera_14, enfermera_16, enfermera_20, medico_1, medico_10, medico_13	enfermera_5, enfermera_8, enfermera_11, enfermera_15, enfermera_17, medico_9, medico_14, medico_15	enfermera_3, enfermera_6, enfermera_8, enfermera_18, medico_9, medico_10, medico_15	enfermera_3, enfermera_8, enfermera_14, enfermera_16, enfermera_17, medico_1, medico_14, medico_15	enfermera_1, enfermera_10, enfermera_15, enfermera_16, enfermera_17, medico_6, medico_9, medico_14	enfermera_4, enfermera_5, enfermera_13, enfermera_17, enfermera_18, medico_3, medico_9, medico_12	enfermera_4, enfermera_8, enfermera_9, enfermera_13, enfermera_18, medico_8, medico_12, medico_15

Ilustración 7.3: Horario Enfermeria: Turno 0 y Turno 1.



Turno \ Día	Día 1	Día 2	Día 3	Día 4	Día 5	Día 6	Día 7
Turno 1	enfermera_9, enfermera_10, enfermera_14, enfermera_16, enfermera_20, medico_1, medico_10, medico_13	enfermera_5, enfermera_8, enfermera_11, enfermera_15, enfermera_17, medico_9, medico_14, medico_15	enfermera_3, enfermera_6, enfermera_8, enfermera_18, medico_9, medico_10, medico_15	enfermera_3, enfermera_8, enfermera_14, enfermera_16, enfermera_17, medico_1, medico_14, medico_15	enfermera_1, enfermera_10, enfermera_15, enfermera_16, enfermera_17, medico_6, medico_9, medico_14	enfermera_4, enfermera_5, enfermera_13, enfermera_17, enfermera_18, medico_3, medico_9, medico_12	enfermera_4, enfermera_8, enfermera_9, enfermera_13, enfermera_18, medico_8, medico_12, medico_15
Turno 2	enfermera_1, enfermera_10, enfermera_12, enfermera_13, enfermera_20, medico_2, medico_4, medico_6	enfermera_4, enfermera_9, enfermera_16, enfermera_19, medico_3, medico_11, medico_12	enfermera_5, enfermera_10, enfermera_11, enfermera_12, enfermera_13, medico_4, medico_7, medico_13	enfermera_2, enfermera_6, enfermera_18, enfermera_19, enfermera_20, medico_3, medico_5, medico_11	enfermera_3, enfermera_7, enfermera_8, medico_4, medico_8, medico_15	enfermera_10, enfermera_14, enfermera_15, enfermera_20, medico_5, medico_6, medico_10	enfermera_5, enfermera_7, enfermera_11, enfermera_17, enfermera_19, medico_2, medico_3, medico_9

Ilustración 7.4: Horario Enfermeria: Turno 1 y Turno 2.

7.4. Discusión de Resultados

En este apartado se repasan los resultados obtenidos en los tres casos de validación de este capítulo, se evalúan sus fortalezas y limitaciones, y se analiza la viabilidad del sistema implementado para su posible explotación práctica. Además, se incluyen comentarios sobre los alcances del modelo, las restricciones no contempladas y una estimación aproximada de los recursos necesarios para su despliegue.

7.4.1. Análisis de los tres escenarios

Caso 6.1: Retenes contra incendios. La resolución de este modelo demostró que la arquitectura puede manejar el horizonte de 6 días y 2 turnos diarios, con 22 retenes y 4 conductores, respetando todas las restricciones (R1–R7). La estructura modular del modelo permite recalcular horarios de forma eficiente ante cambios, aunque el caso asume disponibilidad total de retenes y conductores sin contemplar ausencias de última hora, lo que en la práctica podría requerir reoptimizaciones adicionales.

Caso 6.2: Horario de 1.º de Ingeniería Informática. En este escenario, el modelo cubrió 5 días y 7 franjas diarias para 5 asignaturas, garantizando que cada materia recibiera sus 4 h semanales sin huecos. La rapidez con la que se obtuvo el horario valida el enfoque para planes de estudio con un número moderado de asignaturas y aulas. Sin embargo, no se incluyeron preferencias de profesores ni restricciones de aforo de las aulas (por ejemplo, asignar aulas con suficiente capacidad para el número de estudiantes o disponibilidad de laboratorios especializados), lo que limitaría su uso en centros con requisitos de espacio o equipamiento muy específicos. La implementación, sin embargo, queda preparada para incorporar estas condiciones en iteraciones posteriores.

Caso 6.3: Turnos de enfermería y medicina. Para 7 días y 3 turnos fijos, el modelo integró criterios de carga semanal y mínimos de personal. Al detectar conflicto en mínimos de enfermeras y límite de horas semanales, se recurrió a la relajación de factibilidad, obteniendo una solución con infracciones en algunos turnos. Esto demuestra la utilidad de `FeasibilityRelaxation` para identificar cuáles restricciones generan inviabilidad y permitir una propuesta de horario ajustada. No obstante, al relajar mínimos de personal, el resultado deja huecos que en entornos reales implicarían costes adicionales (subcontratación, horas extra). En producción, sería deseable incorporar penalizaciones económicas o criterios de redundancia para prever estos casos.

7.4.2. Valoración global y limitaciones

En los tres escenarios se confirma que separar claramente las variables y restricciones facilita adaptar el modelo a dominios distintos sin modificar su núcleo. La capacidad para detectar inconsistencias mediante relajación de factibilidad aporta flexibilidad, pero también revela que ciertas restricciones (mínimos de personal, carga máxima) pueden entrar en conflicto según la plantilla disponible.

Entre las limitaciones y alcances destacan:

- **Cobertura de restricciones no incluidas:** No se han modelado preferencias de personal (guardias voluntarias, peticiones de descanso) ni disponibilidad parcial de aulas (equipamiento específico, mantenimiento).
- **Dependencia de solver comercial:** Para mantener tiempos muy bajos en horizontes semanales, la elección de Gurobi es clave. Otros solvers funcionan para casos más reducidos, pero podrían no cumplir requisitos de eficiencia en entornos de mayor escala.
- **Reoptimización ante imprevistos:** Si un recurso (profesor, aula, retén) causa baja imprevista, el modelo actual no ofrece ajuste incremental. Sería conveniente explorar *warm starts* o heurísticas locales que modifiquen únicamente la parte afectada, en lugar de recalcular todo desde cero.
- **Validación de datos de entrada:** Se asume que la información inicial (disponibilidades, capacidades) está libre de errores. En un sistema real, deberían implementarse rutinas de comprobación y saneamiento de datos antes de lanzar la optimización.

7.4.3. Viabilidad del producto y análisis de costes

Viabilidad técnica. La estructura modular del código permite incorporar o relajar restricciones según convenga, por lo que el sistema puede adaptarse con relativa facilidad a distintos entornos (hospitales, instituciones educativas, servicios de emergencia). La integración con una interfaz web o con sistemas de gestión existentes resulta factible sin reescribir la lógica central.

Viabilidad económica. Para obtener una licencia profesional de Gurobi, sería necesario contactar directamente con el equipo de ventas de Gurobi y solicitar un presupuesto personalizado según el número de núcleos de CPU y el tipo de implementación (local, en la nube, contenedores, etc.). A este coste de licencia se sumarían las horas de integración (carga/descarga de datos, validación de formatos), cuyo esfuerzo dependerá de la complejidad del sistema existente.

- *Sustitución del proceso manual:* Si un planificador dedica varias horas semanales a generar horarios manualmente, el ahorro en tiempo de personal podría amortizar rápidamente la inversión en herramientas de optimización.
- *Mantenimiento y actualizaciones:* Los costes para actualizar restricciones (por ejemplo, nuevos tipos de turnos o aulas) se limitan principalmente a horas de programación, sin requerir licencias adicionales mientras se mantenga el mismo solver.

7.4.4. Conclusión de la discusión

En síntesis, los casos de validación demuestran que el modelo es capaz de generar horarios factibles y de calidad para dominios variados, con un tiempo de ejecución adecuado para entornos en tiempo cercano al real. La principal fortaleza radica en la flexibilidad para añadir o relajar restricciones y en las funciones avanzadas de depuración de Gurobi. Entre las limitaciones, destaca la falta de modelado de preferencias internas y la dependencia de un solver comercial para grandes escalas, así como la ausencia de mecanismos de reoptimización incremental.

7.5. Integración con la interfaz web externa

La eficacia de un motor de optimización no depende únicamente de la sofisticación de sus métodos internos. Un aspecto igualmente crucial es su capacidad para intercambiar información de forma fluida con sistemas externos.

En este contexto, `ResQPlan\AI` se ha diseñado como un servicio ligero que utiliza el formato JSON para el envío y la recepción de datos estructurados. Gracias a esta elección, cualquier aplicación capaz de generar texto en lenguaje natural y procesar respuestas puede beneficiarse plenamente de las capacidades de traducción y resolución proporcionadas por el modelo.

Durante el proyecto se desarrolló, de manera coordinada, una interfaz web independiente —obra de **Andrea Hernando** como parte de su TFG— que demuestra la conexión práctica con el optimizador: la página envía el enunciado del problema, recibe las dimensiones y la declaración automática de variables, reenvía las restricciones validadas y finalmente muestra un resumen con las asignaciones y el registro de Gurobi. Esa misma mecánica de intercambio podría ser reutilizada por un planificador de producción, un panel de control interno o un script de automatización sin necesidad de modificar el núcleo de

`ResQPlan\AI`; basta con generar peticiones equivalentes en JSON y procesar la respuesta.

En resumen, el sistema resultante no está ligado a una interfaz concreta: la separación entre la lógica de optimización y la capa de presentación garantiza su reutilización en futuros proyectos o en despliegues donde se requiera orquestar distintas herramientas.

Capítulo 8

Conclusiones y trabajo futuro

8.1. Líneas de trabajo futuro

La versión actual del sistema se centra en garantizar la *viabilidad* de la solución. En la práctica, una vez instanciadas todas las condiciones, el `ShiftOptimizer` invoca a Gurobi sin definir un objetivo de optimización; si el solver declara el modelo `INFEASIBLE`, el módulo activa de forma automática la *feasibility–relaxation* para aflojar las restricciones mínimas imprescindibles y, aun así, entregar siempre un horario propuesto.

Esta estrategia es deliberadamente conservadora: asegura que el usuario obtenga *alguna* planificación en todos los casos, pero a costa de renunciar a la búsqueda de soluciones realmente óptimas.

A continuación se enumeran las líneas de mejora más prometedoras para futuras iteraciones del proyecto.

1. Objetivos definidos por el usuario

Objetivos definidos por el usuario

- **Entrada en lenguaje natural.** Permitir que el usuario exprese metas del tipo "*minimizar la diferencia de horas entre enfermeras*" o "*maximizar el uso del Aula T-1*". *Cómo*: añadir un tercer *prompt* cuyo único cometido sea traducir la frase en una llamada `model.setObjective(...)`. El LLM recibiría, además del JSON `specs`, una tabla con los nombres `VarName` de Gurobi para evitar errores de referencia.
- **Validación estructural automática.** Antes de ejecutar el objetivo convertido, `ShiftOptimizer` debería compilarlo en un *modelo sombra* y verificar:
  - la existencia de todas las variables referenciadas;
  - que el sentido de la función objetivo (minimizar o maximizar) está correctamente definido.

2. Mejoras adicionales

- **Cache LLM a nivel de frase.** Memorizar las parejas $\langle$restricción NL, código$\rangle$ para acelerar sesiones repetitivas y reducir consumo de tokens.
- **Detección temprana de contradicciones.** Integrar el algoritmo *Irreducible Inconsistent Subsystem* de Gurobi en la fase de validación, de modo que las restricciones incompatibles se marquen antes de la optimización final.
- **Re-optimización incremental.** Implementar un modo *hot start* que preserve la base factible actual y sólo re-optimice las partes afectadas por nuevas restricciones, útil cuando se añaden reglas en tiempo real.
- **Empaquetado y explotación comercial**. Evaluar la puesta en producción conjunta del motor `ShiftOptimizer` y la GUI web desarrollada por **Andrea Hernando**. El objetivo es que **Singular Factory** pueda ofrecer el servicio de planificación a terceros bajo demanda.

  a) *Despliegue interno.* Integrar el contenedor del optimizador en la infraestructura de la empresa para cubrir los proyectos de planificación que ya gestiona.

  b) *Servicio para clientes externos.* Publicar una API HTTP/JSON que permita enviar texto en lenguaje natural y recibir el horario optimizado junto con el informe de viabilidad. Sería preciso incorporar:

  - control de cuotas y seguimiento de uso;
  - almacenamiento seguro de los escenarios de cada cliente;
  - un panel de métricas (latencia, tasa de éxito, coste).

  Requisito de licencia. Para explotar la solución con fines comerciales se deberá contratar la licencia correspondiente del solver Gurobi (*Commercial Edition*), ya que la versión académica no autoriza prestaciones a terceras partes.

8.2. Conclusiones

El proyecto ha demostrado que es posible acoplar un *Large Language Model* con un solver de optimización de forma segura y reutilizable. La lógica desarrollada en este trabajo se encarga de:

- descomponer el problema en **dos** pasos claros (*dimensiones/variables* y *restricciones*);
- compilar y validar cada bloque de código antes de unirlo al modelo principal;
- gestionar los errores mediante un *retry loop* y, en última instancia, aplicar la relajación de viabilidad de Gurobi para ofrecer siempre una solución.

Sobre esta base, la interfaz gráfica implementada por **Andrea Hernando González** como parte de su Trabajo de Fin de Grado proporciona al usuario final un flujo completo de edición, validación y descarga de resultados sin necesidad de conocer la sintaxis de Gurobi. El sistema satisface las especificaciones de Singular Factory y ha resuelto con éxito los tres escenarios de validación — retenes contra incendios, horario universitario y plantilla de enfermería—, mostrando tanto la capacidad de obtener soluciones óptimas como de explicar las soluciones inviables de forma comprensible.

En conjunto, la arquitectura resultante ofrece:

- **Flexibilidad** para nuevos dominios de planificación con un simple cambio de texto de entrada.
- **Trazabilidad** entre cada sentencia en lenguaje natural y la restricción concreta del modelo matemático.
- **Escalabilidad** gracias al desacople entre LLM, validación y solver, listo para evolucionar hacia los objetivos personalizables descritos en la Sección 8.1.

Así, la colaboración entre el sistema de optimización y la interfaz ha dado como resultado una solución completa que no solo supera las expectativas iniciales, sino que también sienta las bases para futuras mejoras.

8.2.1. Aprendizaje personal alcanzado.

El proyecto ha supuesto un avance sustancial en la comprensión y la práctica de la optimización combinatoria y de la integración de modelos de lenguaje de gran tamaño (LLM).

Optimización y modelado. El trabajo diario con Gurobi ha permitido afianzar la formulación de modelos mixtos entero–binarios y la aplicación de técnicas de *feasibility relaxation*. Se ha fortalecido la capacidad de diseñar soluciones óptimas para problemas complejos, incluso bajo restricciones difíciles.

Arquitectura de software e ingeniería. El diseño de una arquitectura basada en servicios ha reforzado las competencias de ingeniería de software orientada a la interoperabilidad. Se ha mejorado la habilidad para crear sistemas que se comunican y trabajan de forma cohesionada.

Interacción con LLM. La definición iterativa de *prompts* ha proporcionado un conocimiento profundo sobre cómo guiar a los LLM para producir código fiable. Se ha logrado minimizar el consumo de tokens y los fallos de sintaxis, resultando en un código generado más eficiente y útil.

Colaboración y aplicabilidad comercial. El desarrollo en paralelo de la GUI web, a cargo de **Andrea Hernando González**, ha consolidado la experiencia en colaboración

interdisciplinar. Este esfuerzo ha evidenciado la capacidad del módulo para integrarse con productos externos y, en última instancia, servir como base tecnológica que Singular Factory puede ofrecer a su cartera de clientes.

En resumen, el proyecto ha contribuido al avance de las capacidades técnicas y ha fortalecido la habilidad para colaborar y crear soluciones con valor comercial.

Bibliografía

[1] Burke, E. K., De Causmaecker, P., Vanden Berghe, G., and Van Landeghem, H. (2004). The state of the art of nurse rostering. *Journal of Scheduling*.

[COIN-OR] COIN-OR. Cbc (coin-or branch and cut). https://github.com/coin-or/Cbc. Último acceso: Junio 2025.

[Domene and Planelles] Domene, D. and Planelles, C. Optimización con python: Pyomo vs gams vs ampl. https://es.slideshare.net/slideshow/optimizacion-con-python-pyomo-vs-gams-vs-ampl/55836243. Último acceso: Junio 2025.

[4] Ernst, A. T., Jiang, H., Krishnamoorthy, M., and Sier, D. (2004). Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*.

[GNU Project] GNU Project. Gnu linear programming kit (glpk). https://www.gnu.org/software/glpk/. Último acceso: Junio 2025.

[6] Gu, L., Zhang, Y., and Lin, K. (2025). From integer programming to machine learning: A technical review on solving university timetabling problems. *Journal of Operational Research Advances*, 38(2).

[7] Gurobi Optimization LLC (2024). *Feasibility Relaxation*. Documentación de Gurobi.

[8] Gurobi Optimization, LLC (n.d.). *Gurobi Optimizer Reference Manual*. Disponible en: https://www.gurobi.com.

[9] Kluger, D. M., Aizenbud, Y., Jaffe, A., Parisi, F., Aizenbud, L., Minsky-Fenick, E., Kluger, J. M., Farhadian, S., Kluger, H. M., and Kluger, Y. (2020). Impact of healthcare worker shift scheduling on workforce preservation during the covid-19 pandemic. *Infection Control & Hospital Epidemiology*.

[10] Laborie, P., Rogerie, J., Shaw, P., and Vilím, P. (2018). Ibm ilog cp optimizer for scheduling. *Constraints*.

[11] Mitchell, S. (2017). Pulp: A linear programming toolkit for python. Disponible en: https://github.com/coin-or/pulp.

[12] Motl, J. and Kordík, P. (2021). Fast and exact audit scheduling optimization. In *Proceedings of the ITAT*.

[13] OpenAI (2024). Rate limits and tiers. https://platform.openai.com/docs/guides/rate-limits. Acceso: 27 mayo 2025.

[14] OpenAI (2025). Openai models overview. https://platform.openai.com/docs/models. Consultado 28 mayo 2025.

[15] Peng, M. and Li, M. Z. (2025). Recovair: Model-driven airline scheduling tool for disruption recovery. *Frontiers in Built Environment*, 11.

[16] Perron, L. (2011). *OR-Tools*. Google. Disponible en: https://developers.google.com/optimization.

[17] Pillay, N. and Banzhaf, W. (2010). A study of heuristic combinations for hyper-heuristics for the exam timetabling problem. *European Journal of Operational Research*.

[18] Sahoo, P., Singh, A. K., Saha, S., and Jain, V. (2024). A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927*.

[19] Tang, P. M., Leong, K. K. H., Shaik, N., and Lau, H. C. (2024). Automated conversion of static to dynamic scheduler via natural language. Preprint. School of Computing and Information Systems, Singapore Management University.

[20] Topaloglu, S. and Ozkarahan, I. (2011). An integrated nurse scheduling and assignment system for hospitals. *Computers & Operations Research*.

[21] Van den Bergh, J., Beliën, J., De Bruecker, P., Demeulemeester, E., and De Boeck, L. (2013). Personnel scheduling: A literature review. *European Journal of Operational Research*.

[22] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, volume 30.

[23] Wei, J. and et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. In *Proc. of the 36th Conference on Neural Information Processing Systems*.

[24] Zhou, S. and et al. (2023). Large language models are human-level prompt engineers. *arXiv preprint arXiv:2305.01625*.

[Zuse Institute Berlin] Zuse Institute Berlin. Scip (solving constraint integer programs). https://www.scipopt.org/. Último acceso: Junio 2025.