

Universidad de La Habana
Facultad de Matemática y Computación



**PLAN-GRAIL: mejorando la correctitud sintáctica y semántica
de agentes basados en *LLMs* para modelación de
problemas de planificación**

Autor:
Ariel González Gómez

Tutores:
Dr. Alejandro Piad Morfiss
Dr. Yudivián Almeida Cruz

Trabajo de Diploma
presentado en opción al título de
Licenciado en Ciencia de la Computación



La Habana
junio de 2025
github.com/arielgg46/Thesis

“It’s the questions we can’t answer that teach us the most.

They teach us how to think.

*If you give a man an answer, all he gains is a little fact.
But give him a question and he’ll look for his own answers.”*

— **Patrick Rothfuss, *The Wise Man’s Fear***

Agradecimientos

Quiero agradecer, en primer y destacado lugar, a toda mi familia, por ser mi apoyo vital, emocional y formador. Solo lo que me han otorgado, con enorme amor y sacrificios, ha hecho posible que llegue tan lejos en la vida.

A mi madre, que fue mi primera tutora, que veló mis primeros pasos en la programación, adentrándome en este mundo tan hermoso, y que me ha acompañado de la mano todo el camino. Ponerla orgullosa de mí motiva mi búsqueda de la perfección, en todo lo que hago.

A mi padre, que me enseñó a ser fuerte, a mantenerme positivo en todo momento y, en las situaciones más difíciles, aguantar y seguir avanzando. De él aprendí que cualquier cosa que me propongo, lo hago bien o no lo hago.

A mi hermana, mi cotutora, mi modelo a seguir. Por siempre desempeñar con excelencia el papel de la hermana mayor, por ser mi bastón ético y moral, y por hacerme reir cuando lo necesitaba.

A mis abuelos, por cuidarme, sostenerme, y mimarme como solo ellos saben hacer.

A mis amigos cercanos, que he tenido la suerte de tener muchos y muy buenos. Si los listo, siempre habrá aquel que se quedó fuera y se pondrá celoso, así que ya me aseguraré de agradecerles en persona. Por estar ahí cuando los necesité, por ser mi fuente inacabable de experiencias, motivación y el equilibrio personal durante esta y otras etapas, y bueno, por las risas.

A mis tutores, por confiar en mi trabajo, por su guía experta, paciencia, atención y apoyo constante a lo largo de este proceso.

A mis profesores de todas las enseñanzas. Ellos construyeron la base académica y docente que me llevó a este punto. Un agradecimiento particular a Lupe, que motivó mi amor por la matemática, y la enseñanza.

Finalmente, a las instituciones y fuentes de financiamiento que hicieron posible el acceso a recursos y herramientas necesarios para llevar adelante esta investigación. A los autores de los trabajos científicos que la anteceden y sobre los que se basa.

Muchísimas gracias a todos.

Opinión del tutor

Como tutor del trabajo de diploma titulado “PLAN-GRAIL: mejorando la correctitud sintáctica y semántica de agentes basados en *LLMs* para modelación de problemas de planificación”, realizado por Ariel González Gómez, expreso mi valoración sobre la importancia, originalidad y calidad de la investigación desarrollada.

El tema abordado en esta tesis se sitúa en el centro de uno de los retos más relevantes de la inteligencia artificial contemporánea: el razonamiento de los Modelos de Lenguaje de Gran Escala (*LLMs*) mediante su integración con herramientas externas. En particular, la generación automática de modelos simbólicos (*PDDL*) a partir de descripciones en lenguaje natural es un problema de alto impacto, pues constituye un cuello de botella para la adopción práctica de planificadores clásicos en numerosos contextos reales. Sin embargo, la integración de *LLMs* con planificadores simbólicos presenta desafíos notables, ya que los modelos tienden a “alucinar” características del entorno, sin comprensión del problema, lo que provoca errores sintácticos y semánticos difíciles de controlar. Por ello, la necesidad de garantizar la correctitud formal de las salidas es crítica para la confiabilidad de los sistemas híbridos simbólico-conectivistas.

La tesis de Ariel representa un avance significativo sobre el estado del arte en este campo. El trabajo propone una arquitectura modular y extensible para agentes modeladores basados en *LLMs*, que resuelve de manera definitiva el problema de la correctitud sintáctica en la generación de modelos *PDDL*. Esto se logra mediante la integración de técnicas de *Grammar-Constrained Decoding (GCD)* y la generación automática de gramáticas especializadas para cada instancia de problema, lo que asegura no solo la validez sintáctica (eliminando completamente los errores de *parsing*), sino también una mejora sustancial en la correctitud semántica de las soluciones generadas—con incrementos de hasta 24.3% respecto a los mejores *baselines* reproducidos y un desempeño de 100% en validez sintáctica, 87.1% en solubilidad y 81.4% en correctitud. Más aún, la integración de mecanismos de reintentos, guiados por *feedback* construido automáticamente y autorreflexión de los modelos, permitió elevar estas métricas a 100%, 98.57% y 84.29%, respectivamente. Finalmente, agregando el componente experiencial, consistente en la recuperación por *RAG* de ejemplos similares de soluciones obtenidas durante el entrenamiento, así como la inclusión en el *prompt* de *insights* de modelación, se consiguió la solubilidad del 100% de los modelos, y la correctitud del 87.14%.

Entre las innovaciones principales de la tesis destacan:

- La generación automática de gramáticas con contenido semántico específico para cada instancia de problema, restringiendo la generación a predicados, aridad y tipos válidos según el dominio y los objetos extraídos.
- La incorporación de un agente experiencial capaz de recolectar y utilizar *insights* y reflexiones a partir de problemas similares, permitiendo un proceso de *Transfer Learning* en un entorno socrático, sin necesidad de *Fine-Tuning* ni modificación de los pesos del modelo.
- El uso combinado de razonamiento estructurado en fases, *feedback* automático, aprendizaje experiencial y *Retrieval-Augmented Generation (RAG)* para enriquecer el contexto y guiar la generación hacia

soluciones más fieles y robustas.

Los resultados obtenidos no solo superan en un orden de magnitud la correctitud de las soluciones generadas respecto a los métodos existentes, sino que establecen un nuevo estándar de referencia en la generación automática de modelos simbólicos asistida por *LLMs*. La arquitectura, los métodos y las métricas de evaluación propuestos son reproducibles y de gran valor para la comunidad, con resultados que son plenamente publicables en el primer nivel científico internacional.

Cabe resaltar que todo el mérito de estos logros corresponde a Ariel González Gómez, quien trabajó con absoluta independencia durante todas las etapas del proceso investigativo: desde la revisión crítica del estado del arte, el diseño conceptual, la implementación técnica y experimental, hasta el análisis y discusión de los resultados. Ariel demostró iniciativa, rigor, creatividad y una capacidad sobresaliente para abordar problemas abiertos y proponer soluciones originales y efectivas.

Por todo lo anterior, considero que esta tesis constituye una contribución destacada al avance del campo y recomiendo su aprobación con la máxima calificación. Felicito a Ariel por su desempeño académico y científico ejemplar.

Dr. Alejandro Piad Morfiss
Facultad de Matemática y Computación
Universidad de la Habana
Junio, 2025

Resumen

El modelado simbólico de tareas de planificación representa uno de los principales cuellos de botella para la adopción de planificadores clásicos en contextos reales. Si bien el paradigma *LLM-as-Planner* ha ganado atención reciente, enfrenta serias limitaciones en verificabilidad y optimalidad. Como alternativa, esta tesis se inscribe en el enfoque *LLM-as-Modeler*, en el que agentes basados en *Large Language Models (LLMs)* generan automáticamente modelos *PDDL (Planning Domain Definition Language)* de los problemas a partir de sus descripciones en lenguaje natural. Se propone un agente modelador modular y extensible, diseñado para mejorar la validez sintáctica, solubilidad y correctitud semántica de los modelos generados. La arquitectura combina múltiples técnicas complementarias: razonamiento estructurado en fases, generación restringida por gramática mediante *Grammar-Constrained Decoding (GCD)*, feedback automático y reflexión sobre fallos, aprendizaje experiencial con extracción y refinamiento de soluciones e *insights*, y *Few-Shot Prompting* enriquecido con *Retrieval-Augmented Generation (RAG)*.

La evaluación empírica se realizó sobre subconjuntos estratificados del *benchmark Planetarium*, comparando *baselines* relevantes del trabajo *LLM+P* y variantes del agente propuesto. Los resultados muestran mejoras significativas en todas las métricas clave, destacando la eliminación completa de errores sintácticos con *GCD* mediante gramáticas especializadas y un aumento de hasta **24.3 %** en correctitud semántica respecto al mejor *baseline* reproducido, cerrando **56.7 %** de su margen de mejora. La inclusión conjunta de razonamiento estructurado, extracción de objetos y *GCD* permitió un desempeño de **100 %** en validez sintáctica, **87.1 %** en solubilidad y **81.4 %** en correctitud. Al incorporar mecanismos de reflexión y retroalimentación automática, se corrigieron errores residuales, elevando la solubilidad a **98.57 %** y la correctitud a **84.29 %**. Finalmente, la combinación sinérgica de reintentos con reflexión, recuperación de ejemplos previos e *insights* construidos manualmente permitió elevar las tasas anteriores a **100 %** y **87.14 %**, respectivamente.

Este trabajo aporta una arquitectura funcional, un marco de evaluación reproducible y evidencia empírica de que los *LLMs*, adecuadamente guiados y entrenados, pueden ser agentes modeladores efectivos. La propuesta abre nuevas perspectivas para el diseño de sistemas híbridos simbólico-conectivistas en planificación automática, con menor dependencia de intervención humana experta.

Abstract

Symbolic modeling of planning tasks remains one of the main bottlenecks for the practical adoption of classical planners in real-world contexts. While the *LLM-as-Planner* paradigm has recently gained attention, it faces serious limitations in verifiability and optimality. As an alternative, this thesis adopts the *LLM-as-Modeler* approach, in which agents based on *Large Language Models (LLMs)* automatically generate *PDDL (Planning Domain Definition Language)* models of planning problems from their natural language descriptions. A modular and extensible modeling agent is proposed, designed to improve the parseability, solvability, and semantic correctness of the generated models. The architecture combines multiple complementary techniques: structured multi-phase reasoning, *Grammar-Constrained Decoding (GCD)*, automatic feedback and failure-driven reflection, experiential learning through solution and *insight* extraction and refinement, and *Few-Shot Prompting* enriched with *Retrieval-Augmented Generation (RAG)*.

Empirical evaluation was conducted on stratified subsets of the *Planetarium* benchmark, comparing relevant *LLM+P* baselines and several variants of the proposed agent. Results show significant improvements across all key metrics, including the complete elimination of syntactic errors via *GCD* with specialized grammars, and an increase of up to **24.3 %** in semantic correctness over the best reproduced *baseline*, closing **56.7 %** of its performance gap. The combined inclusion of structured reasoning, object extraction, and *GCD* enabled a performance of **100 %** parseability, **87.1 %** solvability, and **81.4 %** correctness. By incorporating mechanisms for reflection and automatic feedback, residual errors were corrected, raising solvability to **98.57 %** and correctness to **84.29 %**. Finally, the synergistic combination of retries with reflection, retrieval of prior examples, and manually constructed *insights* further increased these rates to **100 %** and **87.14 %**, respectively.

This work contributes a functional architecture, a reproducible evaluation framework, and empirical evidence that properly guided and trained *LLMs* can serve as effective modeling agents. The proposed approach opens new perspectives for the design of hybrid neuro-symbolic systems in automated planning, with reduced reliance on expert human intervention.

Índice general

Introducción	1
1. Estado del Arte y Marco Teórico-Conceptual	7
1.1. Fundamentos de la Planificación Automática	7
1.1.1. De <i>STRIPS</i> a <i>PDDL</i>	7
1.1.2. Estructura de <i>PDDL</i> y fragmento <i>STRIPS + :typing</i>	8
1.2. Algoritmos y herramientas de planificación	8
1.2.1. El <i>International Planning Competition (IPC)</i>	9
1.2.2. <i>Fast Downward</i>	9
1.2.3. <i>VAL</i>	9
1.3. <i>LLMs</i> y su rol en planificación	10
1.3.1. El paradigma <i>LLM-as-Planner</i> y sus limitaciones	10
1.3.2. El Paradigma <i>LLM-as-Modeler</i>	11
1.4. Aprendizaje Basado en Experiencia para Modelado de problemas de planificación	13
1.4.1. <i>ExpeL: LLM Agents Are Experiential Learners</i>	13
1.4.2. <i>Prompt-based Learning</i>	13
1.4.3. <i>Retrieval-Augmented Generation (RAG)</i>	14
1.4.4. Razonamiento y Auto-mejora	15
1.4.5. <i>In-Context Reinforcement Learning</i>	15
1.5. Modelado gramatical y técnicas de restricción formal	16
1.5.1. <i>Grammar-Constrained Decoding (GCD)</i>	16
1.5.2. <i>Backus-Naur Form (BNF)</i>	18
1.5.3. Extensión moderna: <i>GBNF</i> en <code>llama.cpp</code>	18
1.5.4. Aplicaciones Previas de <i>GCD</i> en planificación automática	19
1.6. El <i>Benchmark Planetarium</i>	19
1.7. Conclusiones	21
2. Propuesta de Solución	22
2.1. <i>Baselines</i>	24
2.1.1. Agentes planificadores	25
2.1.2. Agentes modeladores originales	25
2.2. Agentes modeladores propuestos	26
2.2.1. Razonamiento	27
2.2.2. Extracción de Objetos	28
2.3. <i>Grammar-Constrained Decoding (GCD)</i>	28
2.3.1. <i>GCD</i> para generación de modelos <i>PDDL</i> de problema	28

2.3.2. Componentes específicos según inclusión del módulo <i>DAPS</i>	30
2.3.3. <i>GCD</i> para extracción de objetos en dominios no tipados	33
2.3.4. <i>GCD</i> para extracción de objetos en dominios tipados	34
2.4. Evaluación de modelos <i>PDDL</i> y planes	35
2.5. Agente experiencial	36
2.5.1. Reflexión	37
2.6. Entrenamiento	38
2.6.1. Fase de extracción de <i>insights</i>	41
2.7. Agente de <i>Insights</i>	42
2.8. RAG	44
3. Implementación de la propuesta	45
3.1. Estructura del proyecto	45
3.2. Recursos de dominio	45
3.3. Modelos de Lenguaje	46
3.4. Procesamiento y guardado de las operaciones sobre la base de <i>Insights</i>	48
3.5. RAG	48
3.5.1. <i>Retriever</i>	48
3.6. Manipulación de modelos <i>PDDL</i> de problemas	49
3.7. Correcciones del <i>dataset</i> de <i>Planetarium</i>	49
4. Evaluación	51
4.1. Selección de subconjuntos de <i>Planetarium</i>	52
4.2. Proceso de Evaluación de los Agentes	53
4.3. Fase de entrenamiento del agente experiencial	54
4.4. Comparación con <i>baselines</i> de planificación directa	55
4.5. Resultados generales	56
4.5.1. Impacto del Razonamiento Estructurado	58
4.5.2. Impacto de la Extracción de Objetos	59
4.5.3. Impacto de <i>Grammar-Constrained Decoding (GCD)</i> y su variante <i>Domain-and Problem-Specific (DAPS)</i>	60
4.5.4. Un salto cualitativo sobre los <i>baselines</i> existentes	61
4.6. Entrenamiento del agente modelador experiencial	62
4.7. Aplicación de reintentos, <i>feedback</i> y reflexión a la evaluación	63
4.8. Extracción de <i>insights</i>	64
4.9. Evaluación del agente experiencial con todos los módulos incluidos	64
4.10. Limitaciones de la experimentación y evaluación	65
4.11. Resumen de los resultados	67
4.12. Análisis del cumplimiento de las hipótesis	68
Conclusiones	72
Recomendaciones y Trabajo Futuro	75
Bibliografía	77
Anexos	83

Índice de figuras

1.1. Imagen extraída de Geng et al. (2023): <i>Grammar-Constrained Decoding (GCD)</i> aplicado a la tarea de <i>closed information extraction</i> , donde el objetivo es extraer una lista y de tripletes sujeto–relación–objeto del texto de entrada x . Los sujetos y objetos están restringidos a ser entidades de <i>Wikidata</i> , y las relaciones a ser una relación de <i>Wikidata</i> . Durante la decodificación, solo se consideran continuaciones de <i>tokens</i> válidas que cumplen con la gramática. Por simplicidad, se omiten los símbolos de marcador especiales [s], [r] y [o] en el esquema del proceso de generación.	17
1.2. Dominios de problemas de planificación del <i>benchmark Planetarium</i> , de izquierda a derecha: <i>Blocksworld</i> , <i>Gripper</i> y <i>Floor-Tile</i>	20
1.3. Imagen extraída del <i>paper</i> de <i>Planetarium</i> (Zuo et al., 2024): ejemplo de cómo un único objetivo de planificación puede corresponder a múltiples estados objetivos <i>PDDL</i> correctos. Todos los objetivos <i>PDDL</i> en la fila superior representan la meta mostrada correctamente. La fila inferior ilustra objetivos <i>PDDL</i> con diferentes tipos de errores, mostrando instancias que son solubles (un planificador puede generar un plan, pero para un problema de planificación diferente), sintácticamente válidos (la sintaxis <i>PDDL</i> es correcta pero no producirá ningún plan con un planificador) e inválidos (contiene errores de sintaxis).	21
2.1. Flujo de evaluación de los agentes modeladores propuestos.	24
2.2. Flujo de evaluación de los agentes modeladores experienciales propuestos.	37
2.3. Imagen adaptada del <i>paper</i> de <i>ExpeL</i> (A. Zhao et al., 2024). Izquierda: se opera en tres etapas: (1) Recolección de experiencias de éxito y fracaso en un <i>pool</i> . (2) Extracción/abstracción de conocimiento entre tareas a partir de estas experiencias. (3) Aplicación de los conocimientos adquiridos y recuerdo de éxitos pasados en tareas de evaluación. Derecha: (A) Ilustra el proceso de recolección de experiencias a través del agente reflexionador, permitiendo reintentos de la tarea después de la autorreflexión sobre los fracasos. (B) Ilustra el paso de extracción de conocimiento. Cuando se le presentan pares de éxito/fracaso o una lista de L éxitos, el agente modifica dinámicamente una lista existente de conocimientos \hat{t} utilizando las operaciones ADD , AGREE , REMOVE y EDIT	39
4.1. Porcentaje de planes válidos por agente en tareas con descripción completamente explícita.	56
4.2. Métricas calculadas por agente	58

Ejemplos de código

4.1. Corrección de <code>equal_towers</code> dentro de <code>BlocksworldDatasetGenerator.abstract_description</code>	101
4.2. Mejora de redacción en <code>invert</code> dentro de <code>BlocksworldDatasetGenerator.abstract_description</code>	101
4.3. Corrección lógica en <code>GripperDatasetGenerator.drop_and_pickup</code>	102
4.4. Mejora de redacción en <code>GripperDatasetGenerator.abstract_description</code> para <code>drop_and_pickup</code>	102
4.5. Control de errores en <code>GripperDatasetGenerator.holding</code>	102
4.6. Corrección en descripción abstracta para <code>holding</code>	103
4.7. Mejoras de claridad y validez en <code>GripperDatasetGenerator.abstract_description</code> para <code>juggle</code>	103
4.8. Corrección en generación de predicados para <code>checkerboard</code> en <code>FloorTileDatasetGenerator</code>	103
4.9. Corrección de condiciones y redacción en <code>get_robot_ring_string</code>	104
4.10. Mejora de redacción en <code>abstract_description</code> para <code>paint_x</code>	104

Introducción

En las últimas décadas, los problemas de planificación han adquirido creciente relevancia tanto en el ámbito académico como industrial, al ofrecer soluciones formales para tareas que implican asignación de recursos, secuenciación de actividades o coordinación de agentes autónomos (Ghallab et al., 2004; Haoming Li et al., 2024). Desde la robótica de servicio hasta la gestión de flotas de vehículos, muchas de estas tareas pueden formularse como problemas de planificación (Karpas y Magazzeni, 2020; Haoming Li et al., 2024; Marrella, 2019), resueltos mediante algoritmos clásicos, como planificadores heurísticos sobre lenguajes formales como *Planning Domain Definition Language (PDDL)* (Aeronautiques et al., 1998).

Pese a sus grandes beneficios, el uso de los algoritmos clásicos está limitado por la fase de modelado del problema. Sin embargo, han surgido recientemente enfoques basados en Modelos de Lenguaje de Gran Escala (*Large Language Models - LLMs*), que buscan reducir la carga del modelado simbólico o incluso reemplazar al planificador. El paradigma *LLM-as-Planner* propone generar directamente secuencias de acciones desde lenguaje natural, pero presenta limitaciones importantes en factibilidad, verificabilidad y optimidad (Aghzal, Plaku, Stein y Yao, 2025).

Como alternativa, el enfoque *LLM-as-Modeler* traslada el esfuerzo al modelado automático en *PDDL*, delegando la planificación a algoritmos simbólicos consolidados. Esta estrategia combina capacidades lingüísticas con garantías formales, pero aún enfrenta obstáculos relevantes: errores sintácticos, inconsistencias semánticas y fuerte dependencia de supervisión humana. Para abordar estas limitaciones, se propone utilizar estrategias complementarias como la limitación de la salida del *LLM* a la sintaxis de *PDDL* y su gramática correspondiente con *Grammar-Constrained Decoding (GCD)* (Geng et al., 2023), inclusión de fases previas de razonamiento estructurado y extracción de objetos, técnicas de *Retrieval-Augmented Generation (RAG)* (Y. Gao et al., 2023) y aprendizaje experiencial como en el antecedente *ExpeL* (A. Zhao et al., 2024), y recursos de entrenamiento y evaluación sistemática del *benchmark Planetarium* (Zuo et al., 2024). Esta tesis se inscribe en este panorama, y propone contribuir a la mejora del modelado automático con *LLMs*, sin comprometer su escalabilidad ni requerir intervenciones costosas. El nombre en inglés del sistema propuesto es el acrónimo **PLAN-GRAIL**: *Planning with LLM-based Agents via Neuro-symbolic modeling with Grammar-Constrained Decoding, Reasoning and Active In-Context Learning*.

Motivación y Antecedentes

Desde los años 70, la comunidad de planificación ha desarrollado lenguajes formales como *STRIPS* y *PDDL* (Aeronautiques et al., 1998; Fikes y Nilsson, 1971), junto con planificadores simbólicos clásicos como *Fast Downward* o *LAMA* (Helmert, 2006; Richter et al., 2011), que permiten resolver tareas complejas en entornos estructurados. Sin embargo, su despliegue práctico sigue limitado por el alto costo del modelado formal: construir manualmente dominios y problemas en *PDDL* requiere experticia técnica, atención a la sintaxis y comprensión profunda del dominio, lo cual restringe su adopción en contextos reales.

La aparición de los *LLMs*, como *GPT-3* (Brown et al., 2020), ha motivado nuevas estrategias para aliviar

esta carga. Inicialmente se exploró el paradigma *LLM-as-Planner*, donde el modelo genera directamente un plan desde lenguaje natural (Aghzal, Plaku, Stein y Yao, 2025; H. Wei et al., 2025), pero esta línea enfrenta dificultades severas en factibilidad, optimalidad y verificabilidad de los planes generados (Kambhampati et al., 2024).

Como alternativa, surge el enfoque *LLM-as-Modeler*, en el cual el *LLM* traduce la descripción textual a un modelo en *PDDL* del problema, delegando la búsqueda del plan a los motores clásicos (B. Liu et al., 2023; Tantakoun et al., 2025). Este paradigma permite aprovechar las fortalezas combinadas del procesamiento lingüístico de los *LLMs* y las garantías formales de los planificadores simbólicos, pero sigue enfrentando desafíos importantes: errores sintácticos, inconsistencias semánticas y dependencia de intervención humana.

Múltiples trabajos han propuesto soluciones complementarias. Algunos exploran mecanismos intermedios o ciclos de verificación (Agarwal y Sreepathy, 2024), mientras que otros adoptan *Fine-Tuning* sobre corpus especializados (Zeng et al., 2023; Zuo et al., 2024). No obstante, estas estrategias suelen ser costosas, difíciles de escalar y dependientes de infraestructura técnica avanzada y labor humana experta, lo cual motiva la búsqueda de alternativas más ligeras y generalizables, como el *In-Context Learning* (Dong et al., 2022).

A pesar de los avances recientes, la problemática es evidente: la generación automática de problemas *PDDL* mediante *LLMs* sigue enfrentando retos fundamentales, en especial en lo que respecta a la fidelidad semántica y la validez formal de los modelos producidos (Zuo et al., 2024). Aunque herramientas como *Planetarium* han facilitado su evaluación sistemática, los errores persisten: el uso de predicados no declarados o tipos incorrectos, paréntesis desbalanceados, parámetros duplicados, etc., constituyen errores sintácticos del archivo *PDDL* generado. Semánticamente, el uso de predicados de coexistencia contradictoria, o sin conexión con el estado inicial o los objetivos, impiden la resolución correcta del problema. Estos errores reflejan no solo dificultades con la sintaxis del lenguaje de planificación, sino también una comprensión limitada del dominio y de los requisitos lógicos implícitos en la planificación simbólica. Además, las soluciones a estos mismos problemas propuestas hasta la fecha presentan limitaciones y desventajas propias, fundamentalmente la fuerte dependencia de conocimiento experto e intervención humana laboriosa, y los altos costos computacionales.

La motivación y justificación directa de esta tesis radica en una observación concreta: aunque los algoritmos clásicos de planificación automática están disponibles y son eficaces, su uso en la práctica sigue siendo escaso debido a la complejidad del modelado. Al mismo tiempo, los *LLMs* ya están integrados en diversas herramientas de software, y su potencial para ayudar en tareas estructuradas es alto si se les orienta con estrategias adecuadas, para superar las limitaciones inherentes de estos modelos y de los enfoques actuales. Esta investigación busca precisamente avanzar en esa dirección.

La existencia de ciertos trabajos anteriores inspiran directamente el enfoque que toma esta tesis, y el uso de sus estrategias y técnicas se justifica por las limitaciones identificadas que podrían aliviar. Primero, la técnica de *Grammar-Constrained Decoding (GCD)* permite restringir la salida del modelo a una gramática formal predefinida, asegurando la validez sintáctica¹ del *PDDL* generado sin modificar los pesos del modelo (Geng et al., 2023). Segundo, el enfoque experiencial propuesto en *ExpeL* (A. Zhao et al., 2024) demuestra que los modelos pueden adquirir conocimiento estructurado de planificación a partir de ejemplos resueltos, mediante ciclos de prueba y refinamiento, en un entrenamiento previo del agente. Tercero, el *benchmark Planetarium* (Zuo et al., 2024) provee un *corpus* masivo de pares texto-*PDDL* junto con métricas automáticas de validez sintáctica, solubilidad (posibilidad de encontrar un plan válido en el dominio que llegue del estado inicial al objetivo definidos en el modelo, usualmente por un planificador clásico) y equivalencia semántica (fidelidad del modelo a la descripción en lenguaje natural del problema), lo que permite tanto

¹Parseability, término en inglés que se refiere a la correctitud sintáctica, o la propiedad de ser analizable por un *parser* del lenguaje formal definido, sin errores de sintaxis.

el entrenamiento experiencial como la evaluación sistemática de agentes modeladores.

Estos antecedentes evidencian el potencial de combinar restricciones gramaticales, aprendizaje experiencial e infraestructura de evaluación robusta para avanzar hacia agentes modeladores con *LLMs* más eficaces, generalizables y prácticos. Esta tesis se inscribe en dicha dirección.

Problema científico y Ámbito del problema

Actualmente, los agentes basados en *LLMs* que a partir de descripciones en lenguaje natural generan sus representaciones en *PDDL*, con una dependencia humana reducida, presentan dificultades para asegurar que los modelos generados sean válidos sintácticamente, solubles por un planificador clásico y semánticamente correctos o fieles a la intención expresada en el texto.

El objeto de estudio de esta investigación se enmarca en la modelación automática de problemas de planificación a partir de descripciones en lenguaje natural, utilizando agentes basados en *LLMs*.

El campo de acción se circunscribe al diseño e implementación de agentes basados en *LLMs*, para modelación de problemas de planificación, enfocado en la generación automática de archivos *PDDL* en dominios de planificación bien definidos y deterministas. Se incluye, además, la evaluación empírica de los agentes a través del *benchmark* *Planetarium* mediante métricas de validez sintáctica, solubilidad y fidelidad semántica.

Hipótesis

El presente trabajo plantea una serie de hipótesis acerca del impacto de distintas técnicas y estrategias que buscan mejorar los resultados, en las métricas contempladas, de agentes basados en *LLMs* para modelación de tareas de planificación:

- H1.** La división del proceso de modelado en fases estructuradas —extracción de objetos, razonamiento, especificación del estado inicial y metas, y generación del archivo *PDDL*— mejora la correctitud de los modelos generados, al permitir un razonamiento más controlado, modular y verificable.
- H2.** La aplicación de *GCD* permite una generación más confiable del código *PDDL*, reduciendo significativamente o eliminando por completo la aparición de errores de sintaxis.
- H3.** La introducción de reflexión sobre errores y una mínima retroalimentación humana o automática permite al agente corregir patrones de falla recurrentes, contribuyendo al aumento de la solubilidad y la correctitud del *PDDL* generado.
- H4.** La incorporación de *RAG* para la selección de ejemplos relevantes y la extracción de *insights* a partir de soluciones previas (tanto correctas como erróneas), representa una vía prometedora para mejorar la capacidad de los agentes basados en *LLMs* para modelar tareas de planificación, al permitirles adaptarse a la semántica de nuevas tareas y fortalecer su conocimiento sobre el dominio específico y la modelación de problemas.

Objetivo General

Diseñar, implementar y evaluar agentes basados en *LLMs* con técnicas propuestas para mejorar la correctitud sintáctica y semántica de la generación de códigos *PDDL* en modelación de tareas de planificación, y realizar un análisis comparativo de los agentes implementados para determinar los factores que influyen positivamente en métricas como validez sintáctica, solubilidad y correctitud.

Objetivos Específicos

1. Realizar un estudio del estado del arte de agentes basados en *LLMs* para modelación de problemas de planificación.
2. Presentar un marco teórico-conceptual para establecer las definiciones fundamentales relacionados con la planificación automática, y las técnicas y recursos a utilizar en la propuesta de solución.
3. Diseñar un agente basado en *LLM* y variantes del mismo con técnicas propuestas para mejorar la correctitud sintáctica y semántica de la generación de códigos *PDDL* en modelación de tareas de planificación, así como sus fases de entrenamiento, evaluación y análisis comparativo de los agentes propuestos y *baselines* extraídos de los antecedentes descritos en la literatura.
4. Implementar los *baselines*, el agente diseñado y sus variantes, para el posterior análisis de cómo afectan las distintas técnicas e ideas al rendimiento según métricas de validez sintáctica, solubilidad y correctitud. Además, implementar los *pipelines* de generación de los subconjuntos de problemas para entrenamiento y evaluación de los agentes, así como los algoritmos que describen estos propios procesos.
5. Evaluar las distintas variantes del agente en un *benchmark* que permita analizar el rendimiento de agentes modeladores de problemas de planificación, según las métricas propuestas.
6. Realizar un estudio estadístico y de ablación de los resultados de la evaluación, para identificar concretamente las posibles influencias que tienen las técnicas y métodos usados en las métricas analizadas.

Propuesta de solución

La presente investigación propone el desarrollo de agentes modeladores de problemas de planificación basados en *LLMs*, cuya función principal sea la generación automática de archivos de problema *PDDL* a partir de descripciones en lenguaje natural. El *PDDL* generado se utiliza como entrada a un sistema planificador clásico (*Fast Downward*) para generar un plan eficiente que resuelve el problema. Este agente opera en modalidad *Open-Loop*², asumiendo un entorno determinista y sin retroalimentación del entorno, y es evaluado bajo el marco riguroso del *benchmark Planetarium*, el cual permite medir automáticamente la validez sintáctica, solubilidad y correctitud semántica de los modelos generados. La entrada de los agentes es de base la descripción del dominio del problema, el archivo *PDDL* del dominio, y la descripción en lenguaje natural del problema.

Para asegurar la validez sintáctica del código *PDDL* generado por el agente modelador, se integra *GCD* utilizando como *baseline* una gramática *GBNF* (*Georgi Gerganov's Machine Learning (GGML) Backus-Naur form (BNF)*) ([ggml-org, 2023](#)), derivada de la sintaxis general del subconjunto *STRIPS* y *:typing* de *PDDL*. Se propone también el uso de gramáticas generadas automática y específicamente para el dominio y el problema a partir de una fase previa de extracción de objetos tipados, para limitar la generación de predicados a estos objetos, con aridad y tipado de argumentos correctos. Esta técnica se aplica tanto a la generación final del archivo *PDDL* como a la fase intermedia de extracción de objetos.

²Un agente *Open-Loop* genera un plan completo y estático antes de ejecutar cualquier acción, asumiendo un entorno determinista y sin incorporar retroalimentación durante la ejecución. Este enfoque contrasta con los sistemas *Closed-Loop*, que ajustan el plan dinámicamente ante fallos o cambios en el entorno ([H. Wei et al., 2025](#)).

Con inspiración en el enfoque experiencial del trabajo de *ExpeL* bajo el paradigma *LLM-as-Planner*, se propone su adaptación al paradigma *LLM-as-Modeler*. Durante una fase de entrenamiento sobre un subconjunto estratificado del *dataset Planetarium*, el agente recopila un conjunto de soluciones correctas e incorrectas. A partir de estas, se realiza una extracción sistemática de *insights* (conocimiento condensado), orientada a mejorar el conocimiento general de modelación de problemas de planificación, así como a capturar conocimiento específico por dominio (por ejemplo, sobre el uso de predicados, limitaciones o reglas del dominio, o sus patrones típicos). Esta fase es reforzada en su mayoría con mecanismos de refinamiento mediante *In-Context Reinforcement Learning*, y reflexión sobre fallos con *feedback* (retroalimentación) construido automáticamente o por intervención humana mínima. Estos conocimientos se integran en el comportamiento del agente a través de *In-Context Learning*.

Para mejorar la fidelidad semántica y la coherencia de las salidas, se emplea *Few-Shot Prompting* enriquecido con *RAG*, utilizando *embeddings* de lenguaje natural de las descripciones textuales, para seleccionar ejemplos previos relevantes del *corpus* de soluciones correctas construido durante la fase de entrenamiento. Un conjunto de *insights* evoluciona mediante la actualización automática, por un *LLM*, de los pesos y descripciones de los *insights* mediante operaciones bien definidas. Esto permite una adaptación progresiva del agente basada en su experiencia acumulada durante el entrenamiento, con mínima intervención experta.

Para evaluar el impacto de cada componente del agente propuesto, se lleva a cabo un estudio comparativo incremental (en cuanto a las variantes evaluadas) sobre el *dataset Planetarium*. Se diseñan distintas variantes del agente, activando o desactivando módulos específicos (como *GCD*, Razonamiento estructurado o Extracción de objetos). Además se implementan *baselines* contra los que comparar las variantes propuestas, como los agentes principales del trabajo de *LLM+P* (B. Liu et al., 2023): **LLM-as-P⁻** (agente planificador *Zero-shot*), **LLM-as-P⁺** (agente planificador *One-shot* con ejemplo fijo), **LLM+P⁻** (agente modelador *Zero-shot*) y **LLM+P⁺** (agente modelador *One-shot* con ejemplo fijo). Se miden los resultados de todos los agentes en subconjuntos estratificados del *dataset*. Esto permite analizar la contribución relativa de cada técnica a la mejora de las métricas clave, así como la evolución de la calidad de las salidas del agente conforme se integran mecanismos más sofisticados de comprensión, generación y adaptación.

Estructura de la tesis

La presente tesis se estructura en capítulos organizados de forma progresiva para guiar al lector desde el contexto del problema y los fundamentos conceptuales, hasta el diseño, implementación, evaluación y análisis de la solución propuesta:

- **Introducción**

Se presenta el contexto histórico-social donde se desarrolla la investigación, los antecedentes del problema, la problemática, la motivación y justificación, el diseño teórico, las hipótesis, los objetivos, la propuesta de solución y la estructura del trabajo.

- **Capítulo 1: Estado del Arte y Marco Teórico-Conceptual**

Se realiza una revisión crítica de trabajos previos relevantes, incluyendo estudios sobre generación automática de *PDDL* con *LLMs*, agentes planificadores, agentes modeladores, enfoques recientes, etc. Se identifican las limitaciones existentes y se posiciona la contribución de esta tesis dentro del campo. Se abordan los conceptos fundamentales relacionados con la planificación automática, el lenguaje *PDDL*, los modelos de lenguaje de gran escala (*LLMs*), las técnicas de generación estructurada como *GCD*, y los enfoques de razonamiento, aprendizaje experiencial, autorreflexión, *In-Context Learning* reforzado con *RAG*, etc. También se describe el *benchmark Planetarium* y su sistema de evaluación.

- **Capítulo 2: Propuesta de Solución**

Se describe el diseño del agente planificador propuesto, sus distintas fases (razonamiento, extracción de objetos, generación de *PDDL*) y el uso de *GCD* en diversas etapas. Se detalla el procedimiento de entrenamiento en acumulación de experiencias y extracción de *insights*, el diseño de los *prompts* y las gramáticas.

- **Capítulo 3: Implementación de la propuesta**

Se explican las herramientas utilizadas, la arquitectura del sistema, y las decisiones y detalles de implementación.

- **Capítulo 4: Evaluación**

Se presentan los experimentos realizados sobre el *benchmark Planetarium*, comparando variantes del agente y evaluando su rendimiento según validez sintáctica, solubilidad y correctitud. Se analiza la influencia de cada componente de la mejora propuesta sobre estas métricas. Se justifican los experimentos propuestos respecto al análisis de cumplimiento de las hipótesis, en el contexto con limitaciones a las que las evaluaciones están sujetas.

- **Conclusiones**

Se sintetizan los resultados obtenidos, se evalúa el cumplimiento de los objetivos y la validez de las hipótesis bajo las limitaciones observadas.

- **Recomendaciones y Trabajo Futuro**

Se proponen múltiples líneas de trabajo futuras para mejorar y extender la propuesta, basadas en sus limitaciones e ideas de alto potencial no contempladas en esta investigación.

- **Bibliografía**

Se presenta toda la bibliografía consultada para la realización de esta tesis, así como las referencias bibliográficas a literatura citada en este documento.

Capítulo 1

Estado del Arte y Marco Teórico-Conceptual

1.1. Fundamentos de la Planificación Automática

La *Planificación Automática* (*Automated Planning*, AP) es una subdisciplina de la inteligencia artificial centrada en la síntesis de secuencias de acciones para alcanzar un objetivo a partir de un estado inicial, sujeto a las restricciones impuestas por el entorno o dominio (Tantakoun et al., 2025). Su variante más reconocida es el *problema clásico de planificación* (*Classical Planning Problem*), que se formula bajo un conjunto de supuestos idealizados: el entorno es discreto, determinista, estático, completamente observable y monolítico (de un solo agente) (Peter y Intelligence, 2021). Este tipo de entorno permite diseñar agentes planificadores *Open-Loop*, que ejecutan un plan sin requerir retroalimentación sensorial durante su ejecución.

Formalmente, un problema clásico de planificación puede definirse como una tupla:

$$\mathcal{M} = \langle \mathcal{S}, s^I, \mathcal{G}, \mathcal{A}, \mathcal{T} \rangle$$

donde:

- \mathcal{S} es un conjunto finito y discreto de estados del mundo, definidos por una base de hechos atómicos sobre variables proposicionales.
- $s^I \in \mathcal{S}$ representa el estado inicial del mundo.
- $\mathcal{G} \subseteq \mathcal{S}$ es el conjunto de estados objetivo que satisfacen las condiciones de meta.
- \mathcal{A} es un conjunto finito de acciones simbólicas.
- $\mathcal{T}: \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ es la función de transición que modela los efectos de aplicar una acción sobre un estado.

Una solución al problema es un *plan* $\varphi = \langle a_1, a_2, \dots, a_n \rangle$, tal que las precondiciones de a_1 se satisfacen en s^I , y la aplicación sucesiva de cada a_i mediante \mathcal{T} conduce a un estado que satisface los objetivos en \mathcal{G} (Tantakoun et al., 2025).

1.1.1. De *STRIPS* a *PDDL*

La planificación como área formal emergió a partir de estudios de *state-space search*, resolución de teoremas y teoría de control. Un sistema pionero fue *STRIPS* (*Stanford Research Institute Problem Solver*),

desarrollado por Fikes y Nilsson como planificador para el robot *Shakey* (Peter y Intelligence, 2021). Su estructura general se inspiró en el *General Problem Solver (GPS)* de Newell y Simon, que usaba análisis medios–fines como estrategia de búsqueda.

STRIPS introdujo un lenguaje formal para describir acciones en términos de precondiciones y efectos (adición o eliminación de predicados), y su gramática básica es equivalente a la semántica conjuntista del modelo clásico (Ghallab et al., 2004). A lo largo del tiempo, este lenguaje fue extendido a *ADL* (*Action Description Language*) (Pednault, 1987), incorporando cuantificadores, efectos condicionales y disyunciones. Finalmente, como descendiente natural, surgió *PDDL* (*Planning Domain Definition Language*) como estándar de representación para la *International Planning Competition (IPC)* desde 1998 (Aeronautiques et al., 1998).

PDDL está diseñado para describir tanto dominios clásicos como no clásicos, y permite extensiones que modelan observabilidad parcial, entornos estocásticos, múltiples agentes, concurrencia, planificación continua o a largo plazo. Estas dimensiones definen una taxonomía de variantes del problema clásico, dando lugar a planificadores *Closed-Loop* cuando es necesaria la interacción en tiempo real con un entorno dinámico o parcialmente observable.

1.1.2. Estructura de *PDDL* y fragmento *STRIPS* + :typing

PDDL organiza una tarea de planificación en dos archivos: el archivo de dominio *DF* y el archivo de problema *PF* (Tantakoun et al., 2025). El dominio define las constantes lógicas de la tarea, como predicados y acciones. Cada acción $a \in \mathcal{A}$ está definida por sus parámetros $\text{Par}(a)$ (y sus tipos), sus precondiciones $\text{Pre}(a)$ y sus efectos $\text{Eff}(a)$, que determinan cómo se modifica el estado mediante la función \mathcal{T} .

El archivo de problema incluye los objetos concretos de la instancia, el estado inicial s^I y las metas \mathcal{G} . Gracias a su sintaxis clara y declarativa, *PDDL* es fácilmente interpretable por modelos de lenguaje, que son capaces de mapear descripciones textuales a código *PDDL* debido a su frecuente aparición en *corpus* de entrenamiento modernos (Tantakoun et al., 2025).

Este trabajo se enfoca en un subconjunto del lenguaje, correspondiente al fragmento *STRIPS*, junto con la extensión :typing. El fragmento *STRIPS* restringe las acciones a precondiciones positivas y efectos que consisten únicamente en la adición o eliminación de predicados. La extensión :typing permite declarar jerarquías de tipos para los objetos, así como restricciones de tipado en los parámetros de acciones, facilitando la generalización y validación de modelos. Esta decisión técnica responde a las restricciones del benchmark *Planetarium*, basado exclusivamente en tareas de planificación clásica formuladas bajo el fragmento *STRIPS* (Aeronautiques et al., 1998; Fikes y Nilsson, 1971).

El uso de este fragmento también reduce la complejidad del razonamiento sintáctico y semántico, facilitando la implementación de métodos de decodificación restringida por gramática que se emplean en esta tesis.

1.2. Algoritmos y herramientas de planificación

En la práctica de la planificación automática, el desarrollo de algoritmos y herramientas ha sido central para la resolución eficiente de tareas expresadas en lenguajes formales como *PDDL*. Este apartado introduce brevemente los principales sistemas y recursos que han marcado el avance del campo, tanto en términos de evaluación como de implementación, con especial énfasis en aquellos utilizados en el desarrollo de esta tesis. Se destacan en particular el *International Planning Competition (IPC)*, el planificador *Fast Downward* y el validador *VAL*.

1.2.1. El *International Planning Competition (IPC)*

El *IPC* es una competencia internacional que, desde su primera edición en 1998, ha desempeñado un rol crucial en el establecimiento de estándares, el impulso de la investigación y la evaluación comparativa de sistemas de planificación. Organizado inicialmente por Drew McDermott, el *IPC* promovió el uso de *PDDL* como lenguaje común, lo que facilitó la creación de colecciones de referencia para evaluación (*benchmarks*) y la comparación sistemática entre planificadores (Coles et al., 2012).

Además de establecer el estándar lingüístico, el *IPC* ha generado conjuntos de problemas diversos que abarcan dimensiones como entornos deterministas, de aprendizaje y de incertidumbre. Sus ediciones sucesivas han contribuido no solo a determinar qué algoritmos lideran el estado del arte, sino también a proporcionar un repositorio confiable de datos y tareas que impulsan la experimentación. Los resultados del *IPC* se presentan habitualmente en sesiones del congreso *ICAPS*, lo que ha consolidado su influencia en la comunidad científica.

Entre las herramientas destacadas que surgieron o se consolidaron en el marco del *IPC* se encuentran el planificador *Fast Downward* y el validador *VAL*, ambos ampliamente utilizados como base o referencia en la mayoría de las evaluaciones modernas.

1.2.2. *Fast Downward*

Fast Downward es un sistema de planificación clásica orientado a problemas deterministas expresados en el fragmento proposicional de *PDDL 2.2*. Utiliza técnicas de búsqueda heurística y se caracteriza por su enfoque en la planificación por progresión (búsqueda hacia adelante), como otros sistemas predecesores tales como *FF* y *HSP* (Helmert, 2006).

Una de sus principales innovaciones es la traducción de las tareas de planificación a una representación alternativa llamada *multi-valued planning tasks* (tareas de planificación multivaluadas), que permite hacer explícitas muchas de las restricciones implícitas en la representación proposicional. Basándose en esta estructura, emplea una heurística basada en descomposición jerárquica conocida como *causal graph heuristic* (heurística de grafo causal), la cual ha demostrado ser efectiva en múltiples dominios. Su rendimiento fue validado empíricamente al ganar la categoría clásica del *IPC* en su cuarta edición (2004), lo que consolidó su uso como sistema de referencia en *benchmarks* posteriores.

En el marco de esta tesis, *Fast Downward* es el planificador utilizado para evaluar la calidad de los modelos generados automáticamente a partir de descripciones en lenguaje natural. Su compatibilidad con el subconjunto *STRIPS + :typing* de *PDDL* y su confiabilidad lo convierten en una herramienta ideal para validaciones controladas.

1.2.3. *VAL*

VAL es una herramienta diseñada para validar automáticamente planes expresados en *PDDL*. Dado un dominio y problema definidos formalmente, *VAL* permite verificar que una secuencia de acciones constituye un plan válido, es decir, que cada acción es aplicable en el estado en el que se ejecuta, y que la aplicación secuencial de dichas acciones conduce a un estado que satisface las condiciones objetivo (Howey et al., 2004).

A lo largo de múltiples ediciones del *IPC*, *VAL* se ha consolidado como el estándar de facto para la validación de planes generados, garantizando conformidad sintáctica y semántica con la especificación formal del dominio. Su soporte para predicados derivados, efectos continuos y acciones durativas le otorgan gran versatilidad.

1.3. LLMs y su rol en planificación

En los últimos años, la aparición de los Modelos de Lenguaje de Gran Escala (*Large Language Models -LLMs*) ha marcado un cambio de paradigma en el campo de la inteligencia artificial, abriendo nuevas oportunidades en tareas tradicionalmente complejas como la planificación automática. Los *LLMs* son modelos estadísticos del lenguaje natural, comúnmente redes neuronales de tipo autorregresivo, entrenadas mediante aprendizaje auto-supervisado sobre grandes *corpus* de texto. Dados una secuencia de *tokens* $x = \{x_1, x_2, \dots, x_{l-1}\}$, estos modelos estiman la probabilidad del siguiente *token* como $p(x_l | x_{<l})$. Entre los ejemplos más conocidos de *LLMs* se encuentran la familia *GPT* de OpenAI (Brown et al., 2020), *PaLM* y *LaMDA* de Google (Chowdhery et al., 2023; Thoppilan et al., 2022), y modelos ajustados mediante *Fine-Tuning* para seguir instrucciones, como *InstructGPT*, *Flan-T5* y *LLaMA-2 Chat* (Taori et al., 2023; J. Wei, Bosma et al., 2021). Al ser entrenados adicionalmente sobre tareas en formato instrucción-entrada-respuesta, estos modelos presentan una capacidad destacada para seguir instrucciones expresadas en lenguaje natural, reduciendo así la necesidad de diseñar *prompts* complejos (Chung et al., 2024; J. Wei, Bosma et al., 2021).

Aunque estos modelos fueron concebidos originalmente para completar texto, su comportamiento emergente ha demostrado capacidades notables en tareas como razonamiento, uso de herramientas, seguimiento de instrucciones y planificación (Xie et al., 2023; A. Zhao et al., 2024). Mediante el uso de ejemplos de tarea en el *prompt* (conocido como *Few-Shot Prompting*), los *LLMs* pueden adaptarse rápidamente a tareas específicas sin necesidad de reentrenamiento (Devlin et al., 2019; Raffel et al., 2020). Esta flexibilidad ha impulsado el diseño de múltiples metodologías para aprovechar su potencial como núcleo cognitivo de agentes autónomos, abriendo la posibilidad de mejorar las capacidades de planificación automática al integrar razonamiento abstracto en el espacio del lenguaje (Aghzal, Plaku, Stein y Yao, 2025).

1.3.1. El paradigma *LLM-as-Planner* y sus limitaciones

Cuando se emplea un *LLM* para proponer directamente planes, el estado s se representa mediante el contexto provisto en el *prompt* de entrada, que codifica entorno, objetivos y restricciones. El modelo genera texto o símbolos condicionados a dicha entrada, modelando la probabilidad de seleccionar un plan π dado s . El estado inicial s_{init} y el objetivo se describen en lenguaje natural (Aghzal, Plaku, Stein y Yao, 2025). Como el razonamiento ocurre en el espacio lingüístico, se restringe el espacio de acciones a opciones ejecutables usualmente listando las acciones posibles y brindando ejemplos correctos.

Una estrategia común es la **descomposición jerárquica**, que divide el problema $P = (S, A, T, s_{\text{init}}, G)$ en una abstracción de alto nivel $P^H = (S^H, A^H, T^H, s_{\text{init}}^H, G^H)$, generando un plan abstracto π^H refinado en subplanes concretos hasta obtener un plan completo π . Enfoques como *Chain-of-Thought (CoT)* (Kojima et al., 2022; J. Wei, X. Wang et al., 2022), *Least-to-Most* (Zhou et al., 2022) y *Plan and Solve* (Lei Wang et al., 2023) aprovechan esta idea bajo la suposición de que los *LLMs* son más eficaces en tareas de razonamiento a corto plazo. Sin embargo, estos enfoques dependen de *prompts* específicos más que de comprensión algorítmica real (Stechly et al., 2024), fallan en tareas que superan la complejidad de los ejemplos y sufren degradación por límites de contexto (T. Li et al., 2024). Además, requieren entornos completamente observables y sus planes rara vez son óptimos (Aghzal, Plaku, Stein y Yao, 2025).

Otra línea es el **refinamiento de planes**, donde el modelo mejora iterativamente sus conocimientos y planes usando retroalimentación externa o auto-reflexión sobre el resultado de sus acciones en el entorno. Métodos como (I. Singh et al., 2023; Sun et al., 2023; S. Yao, J. Zhao et al., 2023) desacoplan razonamiento y acción, permitiendo ajustar el plan tras cada ejecución, mientras que otros (Kim et al., 2023; Madaan et al., 2023; Shinn et al., 2023) confían en la autoevaluación para generar mejoras. Aunque flexibles, estos enfoques dependen de que la retroalimentación (interna o externa) sea útil y correctamente interpretada,

lo cual tiene limitaciones demostradas (J. Huang et al., 2023; Stechly et al., 2024; Verma et al., 2024); además, su naturaleza iterativa conlleva altos costos computacionales y resultados inefficientes en tareas que requieren muchas iteraciones.

Los **métodos basados en búsqueda** aprovechan la capacidad de los *LLMs* para evaluar soluciones parciales. En *CoT-SC* (X. Wang et al., 2022), el modelo genera múltiples cadenas de razonamiento y agrega los resultados. Métodos posteriores como *Tree of Thoughts* (S. Yao, Yu et al., 2023) y *Reasoning as Planning* (Hao et al., 2023) estructuran el razonamiento como un árbol donde los nodos representan pasos de planificación, y las ramas sub-planes potenciales. Generalmente, un modelo genera acciones y otro las evalúa (Chen et al., 2024). Extensiones como *Graph of Thoughts* (Besta et al., 2024) usan grafos para agregar y refinar ideas. Estos métodos, sin embargo, asumen que el *LLM* puede evaluar fiablemente acciones, lo cual es dudoso en tareas que requieren predicción a largo plazo, y el crecimiento exponencial de ramas los hace costosos y poco prácticos, especialmente en entornos parcialmente observables.

Otra vía explorada es el ***Fine-Tuning* sobre grandes volúmenes de datos**, que ha demostrado mejorar la calidad de los planes generados (Aghzal, Plaku y Z. Yao, 2023; Bohnet et al., 2024; W. Li et al., 2024). Entrenando el modelo con instancias bien estructuradas, puede aprender patrones útiles para tareas similares. Esto se formaliza como la optimización de los parámetros θ para minimizar la pérdida esperada respecto a las soluciones de referencia π^* del conjunto de entrenamiento \mathcal{D} :

$$\min_{\theta} \mathbb{E}_{(s_{\text{init}}, \pi^*) \sim \mathcal{D}} [\ell(\pi(s_{\text{init}}, \theta), \pi^*)]$$

Sin embargo, estos modelos son sensibles a desviaciones del dominio de entrenamiento, lo cual los vuelve poco confiables. En la práctica es imposible incluir todas las configuraciones válidas en el conjunto de entrenamiento, sobretodo en entornos dinámicos o estocásticos. Además, requieren enormes cantidades de datos y poder de cómputo y recursos para el reentrenamiento, lo que los hace imprácticos.

1.3.2. El Paradigma *LLM-as-Modeler*

Una línea de investigación emergente en la intersección entre los *LLMs* y la planificación automática es el paradigma denominado *LLM-as-Modeler*, cuyo objetivo es utilizar modelos de lenguaje para construir especificaciones simbólicas —principalmente en *PDDL*— a partir de entradas expresadas en lenguaje natural. Este enfoque contrasta con el de *LLM-as-Planner*, donde el modelo generaba directamente una secuencia de acciones. En este caso, el modelo actúa como generador de conocimiento declarativo que puede ser luego utilizado por planificadores clásicos.

De acuerdo con la taxonomía propuesta por Tantakoun et al. (2025), las técnicas de generación de modelos (*Model Generation*) pueden clasificarse en tres ramas: *Task Modeling* (modelado de tareas), *Domain Modeling* (modelado del dominio) y *Hybrid Modeling* (modelado híbrido). En ***Domain Modeling***, el objetivo es inducir los componentes estructurales del dominio, como tipos, acciones y relaciones. Ejemplos relevantes incluyen el trabajo de Fine-Morris et al. (2024), que traduce objetivos en estructuras simbólicas intermedias y luego en métodos de descomposición en *HDDL*, y el sistema *PDDLEGO* de L. Zhang et al. (2024), que realiza una refinación recursiva de subproblemas mediante observaciones progresivas del entorno. Por otro lado, ***Hybrid Modeling*** busca generar tanto el dominio como el problema desde una misma fuente textual; por ejemplo, Smirnov et al. (2024) utilizan pasos de preprocesamiento como la generación de *JSON*, comprobaciones de consistencia y bucles de corrección de errores. Su marco de trabajo también incluye un proceso de análisis de alcanzabilidad (*reachability analysis*) para extraer retroalimentación de dominios defectuosos o estados inalcanzables, junto con un análisis de dependencia para verificar el uso de predicados en ambos archivos (*DF* y *PF*).

El presente trabajo se enmarca dentro del ***Task Modeling***, en el cual se busca traducir entradas en lenguaje natural en estados iniciales y metas de planificación. Esta rama puede dividirse en tres subcategorías: *Goal-only Specification* (especificación solo de metas), *Multi-agent Goal Collaboration* (colaboración multiactor para metas), y *Complete PDDL Tasks* (tareas completas en *PDDL*), siendo esta última la de interés específico en esta investigación.

La subcategoría de ***Goal-only Specification*** se enfoca exclusivamente en derivar condiciones de meta, sin especificar el estado inicial. Por ejemplo, [Lyu et al. \(2023\)](#) proponen un enfoque que combina razonamiento paso a paso (*Chain-of-Thought*) con técnicas de validación semántica, y [Collins et al. \(2022\)](#) demuestran que el *Few-Shot Prompting* puede ser suficiente para inducir metas coherentes a partir de descripciones breves.

En este mismo contexto, [Xie et al. \(2023\)](#) realizan una evaluación sistemática del desempeño de diversos *LLMs* al traducir tareas expresadas en lenguaje natural con distintos niveles de ambigüedad. Aunque su sistema no produce tareas completas de planificación, su análisis evidencia los desafíos lingüísticos implicados en la interpretación de objetivos expresados en lenguaje natural y su traducción a formatos simbólicos estructurados.

Por su parte, la línea de ***Multi-agent Goal Collaboration*** aborda la coordinación de submetas entre múltiples agentes. [S. Singh et al. \(2024\)](#) emplean *LLMs* para anticipar desviaciones en los planes previstos de humanos, activando mecanismos de replanificación, mientras que [Izquierdo-Badiola et al. \(2024\)](#) utilizan *LLMs* para asignar metas parciales a distintos agentes y ajustar dinámicamente los costos de acción.

La subcategoría de ***Complete PDDL Tasks*** tiene como propósito traducir descripciones naturales completas de una situación en especificaciones válidas de planificación, incluyendo estado inicial y condición de meta. Esta línea se bifurca en enfoques *Closed Loop*, donde existe retroalimentación del entorno, y *Open Loop*, donde la generación del modelo se realiza de forma estática sin interacción posterior con el entorno o el usuario. Esta investigación se enfoca específicamente en el modelado de tareas completas en *PDDL* bajo modalidad *Open Loop*.

Trabajos como ***LLM+P*** ([B. Liu et al., 2023](#)) han mostrado que es factible traducir descripciones textuales a especificaciones completas en *PDDL*, que luego pueden ser resueltas por planificadores clásicos. Este sistema aplica *Few-Shot Prompting* en dominios conocidos, utilizando ejemplos manualmente construidos como contexto. Si bien supera en rendimiento a métodos que utilizan los *LLMs* como planificadores directos, presenta importantes limitaciones: las evaluaciones dependen de juicios humanos, lo cual limita la escalabilidad por falta de métricas automáticas de equivalencia semántica; además, el método mantiene una alta dependencia de conocimiento experto e intervención manual para garantizar calidad en la salida ([B. Liu et al., 2023; Tantakoun et al., 2025; Xie et al., 2023](#)).

El enfoque ***TIC (Translate-Infer-Compile)*** ([Agarwal y Sreepathy, 2024](#)) plantea una arquitectura modular dividida en tres etapas: traducción del texto a una representación intermedia, inferencia lógica de hechos relevantes, y compilación final a *PDDL*. Esta segmentación permite obtener resultados con muy alta fidelidad, incluso cercanos al 100% en varios dominios del benchmark *LLM+P*. No obstante, el sistema requiere múltiples llamadas al modelo, una considerable inversión en *Prompt Engineering*, y el desarrollo manual de reglas lógicas del dominio que alimentan razonadores externos. Esta dependencia dificulta la escalabilidad y generalización del enfoque, además de elevar considerablemente los costos computacionales.

Otra vía explorada para mejorar el desempeño de los *LLMs* en tareas de modelado simbólico consiste en realizar ***Fine-Tuning*** ([Zeng et al., 2023; Zuo et al., 2024](#)) de modelos pequeños, sobre un *corpus* especializado de ejemplos de problemas de planificación. Esta estrategia ha demostrado que, al ajustar los parámetros del modelo sobre datos relevantes, se pueden obtener mejoras notables en la calidad sintáctica y semántica de las salidas. Sin embargo, también presenta limitaciones importantes. Al igual que su análogo en el paradigma *LLM-as-Planner*, en primer lugar, requiere construir, depurar y anotar un *corpus*

adecuado de entrenamiento, lo cual puede ser costoso y laborioso. Además, el proceso de *Fine-Tuning* es computacionalmente intensivo, requiere ajustes delicados de hiperparámetros y conocimientos técnicos específicos, y no es trivial de reproducir ni de generalizar. Más aún, por razones de escala y costo, esta técnica solo es viable sobre modelos relativamente pequeños (del orden de unos pocos millones a unos cientos de millones de parámetros), ya que los *LLMs* más potentes —con decenas de miles de millones a billones de parámetros— no pueden ser ajustados fácilmente sin acceso a infraestructuras de cómputo masivas y capacidades de almacenamiento distribuido. Estas restricciones reducen la aplicabilidad práctica del *Fine-Tuning* en contextos con recursos limitados, y refuerzan la motivación por explorar enfoques más ligeros y generalizables, como el uso de *In-Context Learning* (Dong et al., 2022; Liang Wang et al., 2023).

1.4. Aprendizaje Basado en Experiencia para Modelado de problemas de planificación

El enfoque propuesto en esta tesis se inspira directamente en la arquitectura del agente *ExpeL* (A. Zhao et al., 2024), diseñado originalmente para resolver tareas de decisión secuencial en entornos embebidos. Si bien nuestro problema está centrado en la generación completa de modelos *PDDL* de planificación, muchas de las ideas subyacentes en *ExpeL* —como el aprendizaje a partir de experiencia, razonamiento reflexivo, inclusión de ejemplos en contexto y mecanismos de memoria— son adoptadas y adaptadas para este nuevo dominio. A continuación se revisan los componentes principales de dicho enfoque, estructurados en cuatro pilares conceptuales: *Prompt-based Learning*, *Retrieval-Augmented Generation*, *Reasoning and Self-improvement*, y *Reinforcement Learning*.

1.4.1. *ExpeL: LLM Agents Are Experiential Learners*

El agente *ExpeL* (A. Zhao et al., 2024) introduce un marco de aprendizaje experiencial para agentes basados en modelos de lenguaje, permitiéndoles mejorar su desempeño sin necesidad de ajustes paramétricos. Inspirado en el aprendizaje humano, *ExpeL* se basa en tres componentes fundamentales: recopilación de experiencias, extracción de ideas y aplicación de conocimientos adquiridos.

Durante la fase de *training*, el agente interactúa con diversas tareas, recopilando trayectorias exitosas y fallidas que almacena en un *pool* (conjunto o repositorio) de experiencias. A partir de estas, extrae *insights* en lenguaje natural que capturan patrones útiles para futuras tareas. En la fase de evaluación o inferencia, el agente utiliza estos *insights* y ejemplos contextuales relevantes para abordar nuevas tareas de manera informada.

Este enfoque permite al agente aprender de manera autónoma, adaptarse a nuevas situaciones y transferir conocimientos entre tareas, todo sin modificar los parámetros del modelo base. Además, *ExpeL* demuestra mejoras consistentes en el rendimiento a medida que acumula experiencias, destacando su eficacia en entornos de toma de decisiones secuenciales.

1.4.2. *Prompt-based Learning*

El aprendizaje basado en *prompts* es un paradigma donde un modelo de lenguaje preentrenado, originalmente diseñado para predecir una salida \mathcal{Y} dado un contexto \mathcal{C} , mejora su rendimiento modificando dicho contexto (P. Liu et al., 2023). Este marco permite aprovechar *LLMs* entrenados sobre grandes volúmenes de texto, sin necesidad de modificar sus parámetros. En particular, el uso de *Few-Shot* o *Zero-Shot Prompting* permite adaptar el modelo a tareas específicas con pocos o ningún ejemplo anotado. Esta modalidad se conoce como *Tuning-free Prompting*, en la que el modelo genera directamente una respuesta dada una instrucción cuidadosamente formulada.

Una variante poderosa del *Prompt-based Learning* es el *In-context Learning*, donde los *prompts* incluyen ejemplos ya respondidos, como en *GPT-3* (Brown et al., 2020), *LAMA* (Petroni et al., 2019), y *CoT* (J. Wei, X. Wang et al., 2022). Los beneficios de este enfoque incluyen eficiencia computacional, ausencia de actualización de parámetros, resistencia al *catastrophic forgetting*¹ y aplicabilidad en entornos con escasez de datos etiquetados. No obstante, su precisión depende altamente de la calidad del *prompt*, lo que conlleva una considerable carga de diseño manual y necesidad de conocimiento experto.

Para aliviar estas exigencias, trabajos como *AutoPrompt* y *Zero-shot-CoT* (Kojima et al., 2022; Z. Zhang et al., 2022) automatizan la generación de cadenas de razonamiento para tareas de procesamiento de lenguaje natural (*Natural Language Processing - NLP*). En esta misma línea, el agente *ExpeL* propone un mecanismo automático para recolectar experiencias en tareas secuenciales, extraer ideas clave de ellas, y utilizarlas junto con ejemplos exitosos como contexto, eliminando así la dependencia de expertos humanos. Esta misma filosofía es adoptada en esta tesis, trasladando la estrategia de construcción de *prompts* efectivos al problema de traducción completa de modelos de problemas de planificación.

1.4.3. *Retrieval-Augmented Generation (RAG)*

La técnica de *RAG* consiste en enriquecer la entrada de un modelo de lenguaje con información relevante recuperada desde una base de datos externa, con el objetivo de mejorar la calidad, fidelidad y precisión de las respuestas generadas. A diferencia del enfoque tradicional que se basa únicamente en el conocimiento interno del modelo preentrenado, *RAG* permite complementar la generación textual con datos contextuales obtenidos dinámicamente (Huayang Li et al., 2022). Esta recuperación se realiza típicamente mediante una función de similitud semántica sobre representaciones vectoriales del texto, y los elementos recuperados se integran en el *prompt* del modelo generador, formando así una instancia de inferencia contextualizada.

La técnica de *RAG* ha ganado popularidad como método para mitigar la alucinación y ampliar el acceso de los *LLMs* a bases de conocimiento externas (Huayang Li et al., 2022). En el ámbito del procesamiento del lenguaje natural, múltiples estudios han demostrado la efectividad de recuperar e incorporar ejemplos al contexto desde bases de datos de demostraciones etiquetadas (Rubin et al., 2021; Liang Wang et al., 2023).

En contraste, tanto *ExpeL* como esta investigación se enfocan en recuperar ejemplos generados por el propio agente a partir de sus experiencias pasadas, lo cual reduce aún más la carga ingenieril por parte del usuario y la necesidad de conocimientos expertos. En nuestro caso, estos ejemplos corresponden a modelos correctos de problemas de planificación (y de los pasos intermedios para su generación, de existir), cuya información es reutilizada para guiar la generación de nuevos modelos *PDDL*.

En esta investigación, se adopta una formulación específica de *RAG* orientada a problemas de planificación automática, donde el agente recupera ejemplos relevantes a partir de una base de datos construida con problemas previamente resueltos durante el entrenamiento.

Formalmente, sea $\mathcal{P}_{\text{train}} = \{p_1, p_2, \dots, p_N\}$ el conjunto de problemas de planificación resueltos previamente, y p_{new} un nuevo problema en lenguaje natural. Cada descripción $p_i \in \mathcal{P}_{\text{train}} \cup \{p_{\text{new}}\}$ se representa mediante un vector de *embedding* $\mathbf{e}_i \in \mathbb{R}^d$, generado por un codificador semántico $f_\theta: \mathcal{T} \rightarrow \mathbb{R}^d$, donde \mathcal{T} denota el espacio de textos en lenguaje natural y f_θ corresponde a un modelo del tipo *Sentence Transformer* preentrenado (Reimers y Gurevych, 2019).

La recuperación se realiza mediante búsqueda por similitud de coseno (*cosine similarity*) (Singhal et al., 2001). Definimos la similitud entre p_{new} y cada problema p_i como:

¹ *Catastrophic forgetting*, también conocida como *Catastrophic interference*, es la tendencia de una red neuronal artificial a olvidar información previamente aprendida, de forma abrupta y drástica cuando aprende una información nueva.

$$\text{sim}_{\cos}(\mathbf{e}_{\text{new}}, \mathbf{e}_i) = \frac{\langle \mathbf{e}_{\text{new}}, \mathbf{e}_i \rangle}{\|\mathbf{e}_{\text{new}}\| \cdot \|\mathbf{e}_i\|}$$

Luego, se seleccionan los k ejemplos más similares según dicha métrica:

$$\mathcal{C}(p_{\text{new}}) = \arg \max_k \{\text{sim}_{\cos}(\mathbf{e}_{\text{new}}, \mathbf{e}_i) | p_i \in \mathcal{P}_{\text{train}}\}$$

Estos k problemas recuperados conforman el *contexto semántico* de entrada, y se concatenan con el *prompt* dado al modelo generador, guiando así la producción del modelo *PDDL* correspondiente al nuevo problema.

1.4.4. Razonamiento y Auto-mejora

Los *LLMs* han demostrado capacidad de planificación en entornos embebidos incluso en modalidad *Zero-Shot* (W. Huang et al., 2022). Sin embargo, múltiples trabajos han señalado que estas capacidades pueden ser potenciadas mediante la incorporación de **mecanismos de razonamiento explícito** (J. Wei, X. Wang et al., 2022; S. Yao, Yu et al., 2023). Un ejemplo notable es el agente *ReAct* (S. Yao, J. Zhao et al., 2023), que combina razonamiento textual y ejecución, logrando resultados superiores a los agentes que únicamente generan acciones, además de ofrecer interpretabilidad sobre su proceso decisional.

Otra línea complementaria es la de los **métodos de auto-mejora**, que aprovechan la capacidad de los *LLMs* para reflexionar verbalmente sobre sus propios errores (Z. Liu et al., 2023; Shinn et al., 2023). *Reflexion* (Shinn et al., 2023) es un agente que genera hipótesis sobre las causas del fallo y se beneficia de un segundo intento. Sin embargo, estos métodos asumen que las tareas son repetibles y que la retroalimentación del entorno está disponible durante la ejecución.

ExpeL propone un enfoque más robusto: en lugar de confiar en retroalimentación del entorno en tiempo real, acumula trayectorias exitosas y fallidas, y extrae de ellas ideas generalizables (*insights*) que luego son utilizadas como guía en nuevas instancias. Esta tesis adopta un mecanismo análogo para mejorar la calidad de los modelos *PDDL* generados, permitiendo que el agente aprenda de sus fracasos previos mediante autorreflexión. Además se incluye retroalimentación sobre el fallo, construida automáticamente.

Agentes con **mecanismos de memoria persistente a largo plazo** han demostrado resultados prometedores en entornos multiagente (Maas et al., 2023; Park et al., 2023; Qian et al., 2023). Estos trabajos generalmente emplean múltiples instancias de agentes generativos que interactúan entre sí y simulan sociedades humanas o escenarios ficticios. Los agentes generativos de Park et al. (2023), por ejemplo, cuentan con un sistema de memoria capaz de recuperar información basada en criterios como la antigüedad, relevancia e importancia, de manera análoga a cómo las personas evocan distintos recuerdos a lo largo del día. Aunque estas líneas de investigación suelen orientarse a tareas abiertas (*open-ended*), los agentes *ExpeL* se enfocan en la resolución concreta de tareas. Al igual que los agentes generativos, tanto el anterior como el presente trabajo incorporan memoria: ejemplos exitosos e *insights* extraídos son almacenados como una forma condensada de memoria, generada íntegramente a partir de la experiencia acumulada por el propio agente.

1.4.5. In-Context Reinforcement Learning

El agente *ExpeL* también toma inspiración de técnicas de *Reinforcement Learning* (*RL*) en su capacidad de recolectar experiencia de manera autónoma (R. S. Sutton, Barto et al., 1998). En particular, emplea una estrategia de *off-policy learning* (Watkins y Dayan, 1992), donde se recurre a *Reflexion* durante el entrenamiento y luego se aplica una etapa de mejora de política (*policy*) mediante recuperación de ejemplos y extracción de *insights*.

Este proceso es análogo a la técnica de *experience replay* (Lin, 1992), donde se seleccionan ejemplos relevantes del pasado para mejorar el aprendizaje del agente (Schaal et al., 2015; Yue et al., 2023). No obstante, a diferencia de estos métodos clásicos, ni *ExpeL* ni el agente propuesto en esta tesis requieren acceso a los parámetros del modelo, ni diseño explícito de funciones de recompensa o pérdida, ni mucha interacción con el entorno.

1.5. Modelado gramatical y técnicas de restricción formal

La generación automática de descripciones en *PDDL* requiere que las salidas sean tanto semánticamente correctas como sintácticamente válidas. Para garantizar esto, se emplean técnicas de restricción formal basadas en gramáticas, que limitan el espacio de generación de los modelos de lenguaje a secuencias válidas. Entre estas, destaca la *Decodificación Restringida por Gramática* (*Grammar-Constrained Decoding - GCD*), que restringe cada paso de decodificación según una gramática formal. Este enfoque se apoya en notaciones como *BNF* y su extensión moderna *GBNF*, diseñadas para estructurar la salida de los *LLMs* en tareas de modelado estructurado como las de esta tesis.

1.5.1. *Grammar-Constrained Decoding (GCD)*

Los modelos de lenguaje autorregresivos generan secuencias de *tokens* de manera iterativa, lo que permite intervenir en el proceso de decodificación para garantizar que la salida se adhiera a una estructura predefinida. Este principio es la base de *GCD*, una técnica que impone una gramática formal durante la generación de texto por parte del modelo, asegurando que las secuencias producidas pertenezcan a un lenguaje formal válido (Geng et al., 2023).

En tareas de *Natural Language Processing (NLP)*, si bien las entradas $x = \langle x_0, \dots, x_{n-1} \rangle$ pueden ser secuencias de *tokens* arbitrarias, las salidas $y = \langle y_0, \dots, y_{m-1} \rangle$ frecuentemente deben seguir estructuras rígidas. Dado que los lenguajes formales brindan un marco completo y riguroso para describir la estructura de cualquier conjunto computable de objetos (de acuerdo a la tesis de Church-Turing (Church, 1936; Turing et al., 1936)), ofrecen una forma prometedora de definir el espacio de salida de tareas estructuradas de *NLP*. Para modelar estas restricciones de salida, *GCD* introduce el uso de gramáticas formales como representación del espacio de salidas posibles. Una gramática formal libre de contexto se define como una tupla $\mathcal{G} = (\mathcal{V}, \Sigma, \mathcal{P}, S)$, donde:

- \mathcal{V} es el conjunto finito de símbolos no terminales,
- Σ es el conjunto finito de símbolos terminales (el vocabulario),
- \mathcal{P} es el conjunto de reglas de producción,
- $S \in \mathcal{V}$ es el símbolo inicial.

Para aplicar esta gramática en la generación de *tokens* por un modelo de lenguaje, es necesario adaptar la gramática a nivel de *tokens*. Esto se logra mediante una transformación de la gramática \mathcal{G} a una versión *tokenizada* $\mathcal{G}_{tok} = (\mathcal{V}, \Sigma_{tok}, \mathcal{P}_{tok}, S)$, donde Σ_{tok} contiene los *tokens* generados por el *tokenizador* del modelo, y las reglas \mathcal{P}_{tok} se obtienen aplicando el *tokenizador* a las cadenas terminales en \mathcal{P} . De este modo, \mathcal{G}_{tok} genera el mismo lenguaje que \mathcal{G} pero en términos del espacio de salida real del modelo.

Durante la decodificación, el modelo genera en cada paso una distribución de probabilidad sobre el vocabulario completo. *GCD* modifica esta distribución restringiendo las opciones posibles a aquellas permitidas por la gramática en el contexto de la secuencia parcial generada hasta el momento. Esta restricción

se impone utilizando un *parser incremental*, que actúa como un motor de completado: dado un prefijo $y_{0:k}$ y la gramática \mathcal{G}_{tok} , devuelve el subconjunto de *tokens* válidos que pueden continuar la secuencia sin violar las reglas gramaticales, y su distribución de probabilidad.

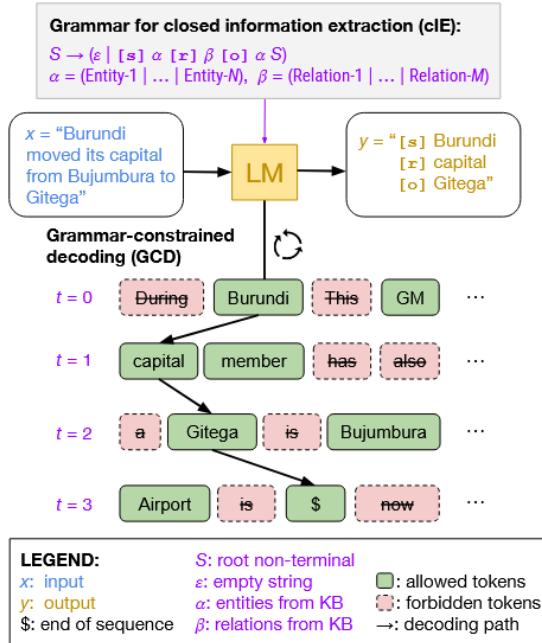


Figura 1.1: Imagen extraída de Geng et al. (2023): *Grammar-Constrained Decoding (GCD)* aplicado a la tarea de *closed information extraction*, donde el objetivo es extraer una lista y de tripletes sujeto–relación–objeto del texto de entrada x . Los sujetos y objetos están restringidos a ser entidades de *Wikidata*, y las relaciones a ser una relación de *Wikidata*. Durante la decodificación, solo se consideran continuaciones de *tokens* válidas que cumplen con la gramática. Por simplicidad, se omiten los símbolos de marcador especiales $[s]$, $[r]$ y $[o]$ en el esquema del proceso de generación.

Este mecanismo es compatible con cualquier algoritmo de decodificación (como *greedy decoding*, *beam search*, *top-k sampling*, entre otros) y puede aplicarse a cualquier modelo de lenguaje autorregresivo que ofrezca acceso a su distribución de salida *token* a *token*. Sin embargo, es importante notar que servicios como los ofrecidos por OpenAI mediante *API* no permiten acceso a dicha distribución, lo que imposibilita el uso directo de *GCD* en estos contextos.

La técnica también puede combinarse eficazmente con *Few-Shot Prompting*. En lugar de permitir que el modelo genere texto libre condicionado a los ejemplos del *prompt*, *GCD* impone una gramática que define la estructura válida de la salida esperada, restringiendo así el espacio de búsqueda y guiando al modelo hacia soluciones sintácticamente correctas. Esto permite aprovechar el conocimiento previo del modelo junto con una especificación formal del dominio de salida, lo que resulta particularmente útil en tareas estructuradas donde se requiere una alta precisión en el formato.

El enfoque de *GCD* representa así una herramienta crucial para la generación segura y verificable de texto estructurado, habilitando aplicaciones en modelado de código, extracción de información, y como se explorará más adelante, en la generación de descripciones válidas en *PDDL*.

1.5.2. *Backus-Naur Form (BNF)*

La Notación de *Backus-Naur* (*Backus-Naur Form - BNF*) surgió a finales de la década de 1950 como una herramienta formal para describir la sintaxis de lenguajes de programación. En 1959, John Backus propuso un conjunto de “fórmulas metalingüísticas” para definir la estructura del lenguaje *ALGOL 58*, basándose en ideas de gramáticas libres de contexto (Backus, 1959). Posteriormente, Peter Naur refinó esta notación en el informe de *ALGOL 60* de 1963, denominándola “Forma Normal de Backus” (*Backus Normal Form*) (Backus et al., 1963). Sin embargo, Donald Knuth sugirió renombrarla como *Backus-Naur Form* para reconocer la contribución de Naur y evitar la confusión con otras formas normales existentes en teoría de lenguajes formales (Knuth, 1964).

BNF es un metalenguaje que permite expresar gramáticas libres de contexto mediante reglas de producción. Cada regla tiene la forma:

```
<no-terminal> ::= <expresión>
```

Donde *<no-terminal>* representa una categoría sintáctica y *<expresión>* define cómo se puede construir esa categoría utilizando otros símbolos terminales o no terminales. Por ejemplo, una regla podría ser:

```
<expresión> ::= <término> | <expresión> "+" <término>
```

Esta notación facilita la definición precisa y estructurada de la sintaxis de lenguajes de programación, siendo fundamental en el desarrollo de compiladores y analizadores sintácticos.

1.5.3. Extensión moderna: *GBNF* en `llama.cpp`

La notación *Backus-Naur* (*BNF*) ha sido fundamental en la definición formal de lenguajes de programación. Sin embargo, con el auge de los *LLMs*, ha surgido la necesidad de formatos más expresivos y adaptados a estos nuevos contextos. En este sentido, fue introducida *Georgi Gerganov's Machine Learning (GGML) Backus-Naur Form (GBNF)*, una extensión de *BNF* diseñada específicamente para restringir y estructurar las salidas de los *LLMs* en aplicaciones prácticas (ggml-org, 2023).

GBNF se implementa en el proyecto `llama.cpp`, una versión en `C++` de los modelos *LLAMA* desarrollada por Gerganov. Esta notación permite definir gramáticas que los modelos deben seguir al generar texto, asegurando que las salidas cumplan con estructuras sintácticas específicas. A diferencia de *BNF*, *GBNF* incorpora características modernas similares a las expresiones regulares, lo que permite una mayor flexibilidad y control sobre las salidas generadas por los modelos (ggml-org, 2023).

Aunque *GBNF* no cuenta con una publicación académica formal, su documentación principal se encuentra en el repositorio oficial de `llama.cpp`, específicamente en el archivo `grammars/README.md`. Esta fuente proporciona ejemplos y directrices sobre cómo implementar y utilizar *GBNF* en proyectos que requieren salidas estructuradas de modelos de lenguaje (ggml-org, 2023).

En el contexto de esta tesis, *GBNF* se utiliza como el mecanismo formal que habilita la aplicación práctica de *GCD* para el modelado automático de problemas en *PDDL*. Al restringir la salida de un *LLM* mediante una gramática *GBNF*, se garantiza que las descripciones generadas sean sintácticamente válidas conforme a la especificación del lenguaje objetivo, facilitando tanto su validación como su posterior análisis por planificadores clásicos.

1.5.4. Aplicaciones Previas de *GCD* en planificación automática

Grammar Prompting for Domain-Specific Language Generation with Large Language Models (B. Wang et al., 2023) propone una técnica denominada *grammar prompting*, que permite a los *LLMs* generar la gramática, expresada en *BNF*, que seguirán sus salidas posteriores mediante *GCD*. En el contexto de planificación, el modelo primero predice una gramática especializada para una tarea dada y luego genera un plan *PDDL* que cumple con dicha gramática. Aunque este enfoque logra mejoras en la generación de planes sintácticamente válidos, se centra únicamente en la producción de planes, no en la generación completa de modelos de problemas de planificación.

Por otro lado, *Syntactic and Semantic Control of Large Language Models via Sequential Monte Carlo* (Loula et al., 2025) introduce un marco basado en *Sequential Monte Carlo* para controlar la generación de texto por *LLMs* bajo restricciones sintácticas y semánticas. Aplicado a la generación de modelos *PDDL*, este enfoque utiliza una gramática *STRIPS* general y se limita al dominio de *Blocksworld* del *benchmark Planetarium* con hasta 10 objetos, asumiendo que el estado inicial (`:init`) es proporcionado. Sus resultados experimentales brindaron indicios positivos de la utilidad de *GCD* para el modelado de tareas de planificación. Sin embargo, aunque el método propuesto en el artículo permite una generación más controlada y coherente, su dependencia de información previa sobre el estado inicial restringe su capacidad para generar modelos completos a partir de descripciones en lenguaje natural. Además, no se hace una exploración a profundidad de la técnica al limitarse a un subconjunto muy reducido y solo parcialmente representativo de *Planetarium*.

En contraste, esta tesis propone un enfoque que utiliza *GCD* para la generación completa de modelos *PDDL* (incluyendo `:init`, `:goal`, `:objects`, etc.) a partir de descripciones en lenguaje natural y del dominio *PDDL* correspondiente. Más aún, se evalúa en todas las dimensiones (dominio, cantidad de objetos, nivel de abstracción, etc.) del *benchmark Planetarium*. De forma adicional a la gramática general, se introducen nuevos niveles de restricción especializada que aseguran el uso exclusivo de predicados disponibles, objetos declarados y la correcta aridad y tipado de argumentos, permitiendo así una generación más precisa y aplicable a una variedad de dominios y problemas específicos.

1.6. El *Benchmark Planetarium*

La evaluación moderna de modelos generados automáticamente en *PDDL* requiere herramientas específicas que permitan analizar no solo la validez sintáctica del código, sino también su estructura semántica y capacidad de resolución. En esta línea, el *benchmark Planetarium* (Zuo et al., 2024) constituye una de las contribuciones más destacadas, al ofrecer un conjunto masivo de ejemplos texto–*PDDL* junto con un protocolo de evaluación riguroso basado en estructuras de grafo intermedias. Este recurso permite evaluar la generación automática de problemas de planificación desde descripciones en lenguaje natural con criterios que trascienden la simple exactitud textual o el *string matching*.

La representación central empleada por *Planetarium* es el *scene graph* (grafo de escena), una estructura de datos ampliamente utilizada en visión por computadora y gráficos computacionales para representar objetos, sus atributos y relaciones. Formalmente, un *scene graph* es un grafo dirigido $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, donde $\mathcal{V} = \mathcal{O} \cup \mathcal{P}$ incluye nodos de tipo objeto (\mathcal{O}) y proposición (\mathcal{P}), y cada arista $e \in \mathcal{E}$ tiene atributos que codifican el predicado al que pertenece, la posición del argumento en la proposición, y si la proposición corresponde al estado inicial o al estado meta. Para cada archivo de problema en *PDDL*, se construyen dos grafos: uno que representa el estado inicial y otro que representa las proposiciones objetivo. Estos se combinan en un *problem graph*, definido como la unión etiquetada de ambos:

$$\text{ProblemGraph} = (\mathcal{O} \cup \mathcal{P}_{\text{init}} \cup \mathcal{P}_{\text{goal}}, \mathcal{E}_{\text{init}} \cup \mathcal{E}_{\text{goal}})$$

La noción de equivalencia entre problemas se formaliza como un isomorfismo de grafos sobre los *problem graphs*, es decir, una biyección entre nodos que preserve tanto la conectividad como los tipos y atributos. Esta definición permite comparar problemas generados automáticamente contra su referencia canónica sin depender del soporte textual de los archivos.

El algoritmo propuesto para verificar esta equivalencia comienza transformando cada conjunto de proposiciones en grafos de escena, completando los grafos objetivo con aristas triviales (como proposiciones que se pueden inferir verdaderas), y finalmente construyendo un *problem graph* a partir de la unión. Si existe un isomorfismo entre el grafo generado y el de referencia, se considera que el problema es estructuralmente equivalente y, por tanto, correcto.

El conjunto de datos de *Planetarium* incluye más de 100 000 pares texto–*PDDL*, generados proceduralmente a partir de plantillas que combinan configuraciones (subtipos de estados o subtareas específicas) de los estados iniciales y objetivos. Los dominios utilizados son *Blocksworld*, *Gripper* y *Floor-Tile*, elegidos por su uso extensivo en la literatura y su dificultad inherente. Cada problema contiene tanto una descripción en lenguaje natural como una instancia en *PDDL* construida a partir de la combinación de las plantillas predefinidas. La generación automática varía sistemáticamente la abstracción de las descripciones textuales (desde descripciones explícitas hasta resúmenes abstractos de estados) y el tamaño de los problemas medido por la cantidad total de proposiciones u objetos.

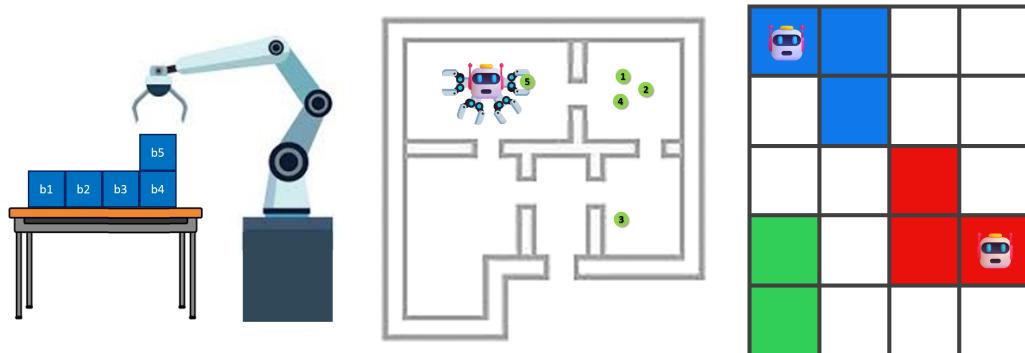


Figura 1.2: Dominios de problemas de planificación del *benchmark Planetarium*, de izquierda a derecha: *Blocksworld*, *Gripper* y *Floor-Tile*

El protocolo de evaluación del dataset considera tres métricas: el número de modelos sintácticamente válidos, el número de modelos solubles y el número de modelos correctos. Un modelo es considerado sintácticamente válido si es analizable por un *parser* de *PDDL*, y es posible extraer desde su salida un problema en *PDDL* válido y convertirlo en un grafo estructuralmente bien formado. La solubilidad se computa determinando la existencia de un plan válido que lleva desde el estado inicial al estado objetivo, utilizando para ello planificadores específicos para los dominios *Blocksworld* y *Gripper*, y *Fast Downward* para problemas del dominio *Floor-Tile*. Finalmente, un problema se considera correcto si, siendo sintácticamente válido y soluble, es también estructuralmente equivalente al problema original según el algoritmo de isomorfismo de grafos. Para asegurar la validez de los planes generados, todos los resultados se verifican con la herramienta *VAL*.

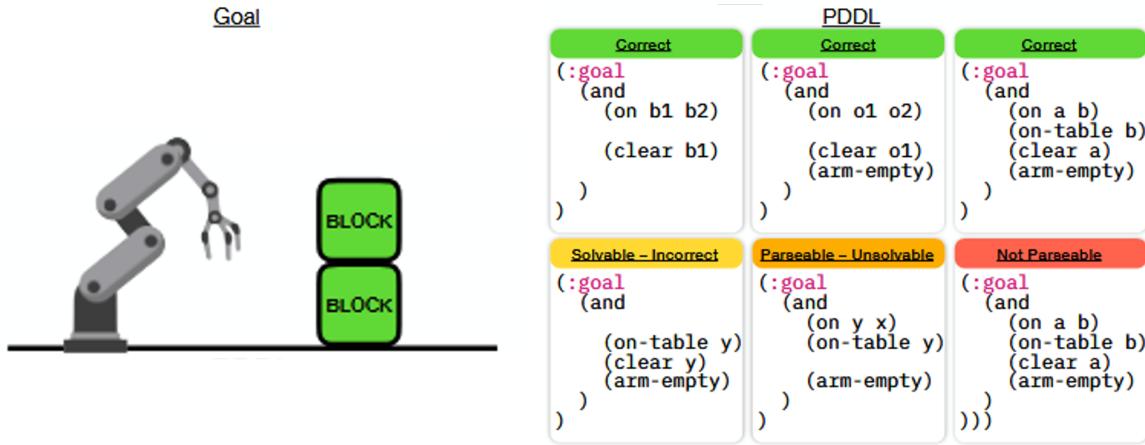


Figura 1.3: Imagen extraída del *paper* de *Planetarium* (Zuo et al., 2024): ejemplo de cómo un único objetivo de planificación puede corresponder a múltiples estados objetivos *PDDL* correctos. Todos los objetivos *PDDL* en la fila superior representan la meta mostrada correctamente. La fila inferior ilustra objetivos *PDDL* con diferentes tipos de errores, mostrando instancias que son solubles (un planificador puede generar un plan, pero para un problema de planificación diferente), sintácticamente válidos (la sintaxis *PDDL* es correcta pero no producirá ningún plan con un planificador) e inválidos (contiene errores de sintaxis).

Este protocolo permite una evaluación robusta de modelos basados en lenguaje natural y ofrece un *baseline* reproducible para medir avances en la generación automática de modelos de planificación.

1.7. Conclusiones

Este capítulo ha presentado un recorrido exhaustivo por los fundamentos, avances y desafíos actuales en el campo de la *planificación automática*, así como por las oportunidades emergentes que surgen del uso de *LLMs* para la generación estructurada de modelos en *PDDL*.

En conjunto, se evidencia que, si bien los *LLMs* poseen un enorme potencial para tareas de planificación estructurada, su aplicación directa a la generación de modelos *PDDL* enfrenta barreras significativas de robustez y validación. Las técnicas revisadas —aprendizaje experiencial, decodificación gramatical, evaluación estructural con *Planetarium*— ofrecen elementos clave para superar estas barreras. Sobre esta base se plantea en esta tesis el diseño de un agente modelador que:

- Divide el proceso de modelado en fases de razonamiento estructurado, extracción de objetos y generación de *PDDL*;
- Aprende de sus errores mediante ciclos de experiencia y reflexión;
- Está asistido por restricciones gramaticales explícitas mediante *GCD* y *GBNF*;
- Es evaluado rigurosamente con el protocolo estructural de *Planetarium*.

Este enfoque integral busca cerrar la brecha entre el poder expresivo de los *LLMs* y la precisión requerida por los sistemas simbólicos de planificación, contribuyendo así a un nuevo paradigma de generación automática de modelos válidos, verificables y reutilizables en el contexto de la *planificación automática*.

Capítulo 2

Propuesta de Solución

La propuesta desarrollada en esta investigación consiste, principalmente, en un sistema compuesto por múltiples agentes especializados, cuyo objetivo es la generación automática de modelos y planes *PDDL* de problemas de planificación a partir de descripciones en lenguaje natural, con foco en dominios definidos en el lenguaje *STRIPS* extendido con *:typing*. El diseño sigue una arquitectura jerárquica y modular, basada en el uso de *LLMs* para la comprensión semántica de los problemas, junto con mecanismos de entrenamiento experiencial, razonamiento estructurado, decodificación restringida por gramáticas, retroalimentación y reflexión sobre fallos. La validez de esta propuesta se evalúa de forma rigurosa sobre el *benchmark Planetarium*, que proporciona un conjunto amplio de tareas con evaluación automática de validez sintáctica, solubilidad y correctitud semántica de los modelos.

Inicialmente, se definen dos conjuntos de *baselines*: por un lado, los *agentes planificadores*, encargados de emitir planes directamente en formato *PDDL* y, por otro, los *agentes modeladores originales del trabajo LLM+P*, sin razonamiento explícito ni restricción gramatical, representando el paradigma más simple de modelado asistido por *LLMs*. Ambos incluyen modalidad de *Zero-Shot* (sin ejemplo resuelto añadido al *prompt*) y *One-Shot Prompting* (con un ejemplo incluido). Estos agentes son reproducidos fielmente del trabajo *LLM+P*, y adaptados a su evaluación en *Planetarium*.

En contraste, los *agentes modeladores propuestos* están compuestos por múltiples submódulos especializados opcionales, que permiten descomponer y controlar la tarea de modelado. Primero, se permite ejecutar una **fase de razonamiento estructurado** para identificar los objetos relevantes, el estado inicial y objetivos del problema. Luego, se puede llevar a cabo una **fase de extracción de objetos**, en la que se listan explícitamente los objetos que formarán parte del modelo de problema. Esta descomposición semántica permite controlar la generación mediante *GCD*.

La producción final del modelo *PDDL* se realiza mediante *GCD*, que garantiza la validez sintáctica del modelo generado. Como punto de partida se utiliza una gramática *GBNF*, diseñada específicamente para codificar la sintaxis del subconjunto de *PDDL* utilizado. Adicionalmente, se extiende este enfoque mediante la construcción dinámica de gramáticas especializadas, que se denominarán ***Domain-and Problem-Specific (DAPS)***, a partir de los objetos (tipados o no) previamente identificados, limitando la generación a predicados con aridad y argumentos válidos en el contexto del dominio y problema. Esto se consigue especializando las reglas de producción de la gramática, utilizando los objetos extraídos en la fase previa correspondiente. La técnica de *GCD* se emplea tanto en la fase final de generación como durante la etapa previa de extracción de objetos, y se distinguen mecanismos diferenciados para dominios tipados y no tipados.

Un componente principal de esta propuesta es el *agente experiencial*, con capacidad de reflexionar sobre los intentos fallidos de modelado, y realizar un entrenamiento que le permite acumular experiencias e

insights. A través de un modelo *LLM* especializado en reflexión estructurada, este agente analiza tanto los errores de *parsing* como las fallas semánticas que impiden la solubilidad o equivalencia del modelo con la descripción del problema, proponiendo hipótesis correctivas que se integran en las siguientes iteraciones del agente modelador. El *feedback* que alimenta esta fase se construye automáticamente mediante el cálculo de las métricas concebidas en el *benchmark Planetarium*, además de variantes parciales (validez o correctitud individual de :*init* y :*goal*) mediante la manipulación del modelo *PDDL* generado.

La propuesta incluye además una **fase de entrenamiento** dedicada primero a la **acumulación de soluciones correctas**, y luego a la **extracción de insights**. En esta se recopilan tanto salidas exitosas como fallidas del agente durante la resolución de múltiples tareas, generando una base de conocimientos estructurados. Los *insights* abarcan buenas prácticas de planificación y conocimiento de los dominios. La extracción se realiza mediante un agente especializado basado en *LLM*, al cual se le presentan conjuntos de soluciones correctas e incorrectas para su análisis. El agente modifica su base de conocimientos mediante operaciones definidas para añadir, editar y ponderar los *insights* en función de su relevancia y utilidad.

Este conocimiento se incorpora al comportamiento del agente a través de técnicas de **In-Context Learning**. Los ejemplos de soluciones correctas presentados (**Few-Shot Prompting**) son provistos manualmente por el experto, o se seleccionan mediante mecanismos de **Retrieval-Augmented Generation (RAG)**. En el caso de inclusión de este módulo, las descripciones de los problemas se transforman en vectores de *embeddings* de lenguaje natural que permiten recuperar los ejemplos más cercanos en el espacio semántico, del *pool* de experiencias acumuladas, mediante el cálculo de *cosine similarity*.

Se consideran múltiples variantes del agente con diferentes combinaciones de módulos activos o inactivos (por ejemplo, con o sin *GCD*, reflexión, o *RAG*). La modularidad y flexibilidad de la propuesta facilita el análisis de la influencia de estos componentes en la calidad de los resultados.

En resumen, la solución propuesta no se limita a la generación directa de modelos *PDDL*, sino que incorpora mecanismos robustos de restricción sintáctica, razonamiento estructurado, aprendizaje progresivo y reflexión sobre errores. Esta integración de múltiples niveles de inteligencia simbólica permite alcanzar mejoras sustanciales en la calidad de los modelos generados, acercando el uso de *LLMs* en planificación a una solución confiable y escalable. A continuación, se presenta a detalle todo el diseño de la propuesta.

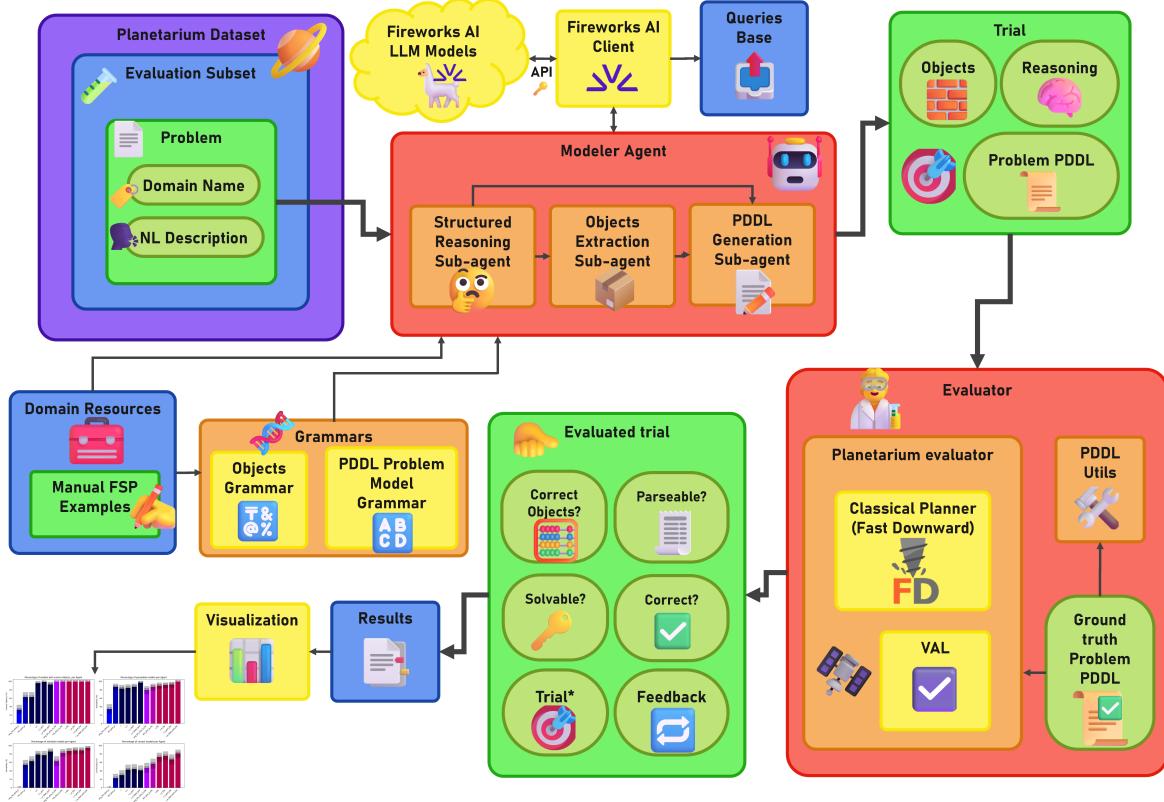


Figura 2.1: Flujo de evaluación de los agentes modeladores propuestos.

2.1. *Baselines*

Con el fin de evaluar el impacto de los métodos propuestos para mejorar la generación de modelos *PDDL*, se realiza una comparación con los agentes presentados en el trabajo *LLM+P* (B. Liu et al., 2023). Este trabajo incluye tanto un agente planificador como un agente modelador, ambos en modalidad *Zero-Shot* (sin ejemplos previos), así como sus respectivas variantes *One-Shot* (con un ejemplo fijo).

Se concibe una reproducción fiel de los agentes modeladores originales, realizando únicamente adaptaciones necesarias en sus *prompts* para adecuarlos a los dominios del *benchmark Planetarium*. Si bien el dominio *Blocksworld* coincide exactamente con el utilizado en *LLM+P*, los dominios *Gripper* y *Floor-Tile* presentan pequeñas diferencias estructurales. Por esta razón, se adaptaron los ejemplos de *FSP* manteniendo la intención de los problemas originales, pero reformulados según las variantes correspondientes de *Planetarium*.

Se observó que los *prompts* utilizados en el trabajo original eran considerablemente subóptimos para una evaluación automatizada en el entorno de *Planetarium*, particularmente en el caso *Zero-Shot*. En esta modalidad, el único contexto proporcionado al agente modelador es la descripción en lenguaje natural de las acciones disponibles. A partir de esta descripción, el agente debe inferir la totalidad de los predicados requeridos por el modelo *PDDL*, incluyendo su nombre y sus argumentos. Esta condición genera una desventaja significativa, ya que la evaluación automática de *Planetarium* exige que los predicados utilizados coincidan exactamente con aquellos definidos en el dominio.

En el trabajo original, este tipo de desalineaciones no representaba una limitación crítica debido a que la evaluación de los modelos *PDDL* generados se realizaba manualmente. Por el contrario, en el presente trabajo se requiere una validación automática que depende estrictamente de la precisión sintáctica y semántica de los modelos generados.

En la modalidad *One-Shot*, el agente recibe como entrada un ejemplo completo, pero sin la descripción explícita de todos los predicados disponibles. Aunque este enfoque ofrece al agente más información que la modalidad *Zero-Shot*, todavía presenta deficiencias importantes en cobertura y comprensión semántica.

Por estas razones, se propone una reimplementación de los agentes modeladores originales, incorporando el modelo *PDDL* del dominio como parte del contexto, además de una estructura de *prompt* mejorada. A partir de estas versiones se pretende derivar los agentes propuestos en esta tesis, a los cuales se integran los distintos módulos de mejora.

De igual forma, los agentes planificadores son reimplementados, realizando ligeros ajustes en los *prompts* con el objetivo de presentar explícitamente el formato de las acciones disponibles y adecuarlos a las particularidades de las variantes de *Planetarium*.

2.1.1. Agentes planificadores

Los agentes planificadores propuestos utilizan *LLMs* para resolver tareas de planificación automática. A partir de la entrada compuesta por el modelo del dominio en *PDDL* y la descripción en lenguaje natural del problema, generan como salida un plan válido en el formato requerido por el validador externo *VAL*, así como el conteo de *tokens* consumidos durante la generación.

Basados en el enfoque propuesto en *LLM+P*, se proponen dos variantes de agentes cuya única diferencia es la inclusión opcional de un ejemplo de tipo *FSP* en el *prompt*. Esta inclusión es modular y depende de los objetivos del experimento. Ambos agentes comparten una misma plantilla general que estructura la información presentada al modelo.

El *prompt* comienza situando al modelo en el rol de un agente de planificación avanzado. Luego se introduce el dominio: su nombre, una breve descripción en lenguaje natural y el listado de acciones disponibles. Esta información proviene de una base de recursos estructurada por dominio, construida a partir de conocimiento experto. A continuación, se describe la tarea a realizar y se especifica el formato de salida esperado, también obtenido automáticamente desde la misma base.

Cuando se activa el módulo *FSP*, se incorpora un ejemplo representativo del dominio, compuesto por una descripción del problema y su plan correspondiente. Este ejemplo sirve como guía contextual y es tomado directamente del conjunto de recursos por dominio.

Finalmente, se presenta el nuevo problema a resolver, tal como aparece en el *dataset* de *Planetarium*, y se espera que el modelo complete la sección final con el plan correspondiente.

2.1.2. Agentes modeladores originales

Los agentes modeladores originales, inspirados en la propuesta de *LLM+P* (B. Liu et al., 2023), generan como salida un archivo *PDDL* que representa formalmente el problema de planificación descrito en lenguaje natural. Además, se incorpora como métrica auxiliar el número total de *tokens* utilizados durante la generación, para facilitar el análisis de eficiencia.

La definición y *prompt* de estos agentes varía en función de si se encuentra o no activado el módulo *FSP*, que determina la presencia de ejemplos explícitos en el *prompt*. Cuando dicho módulo no está activo, el modelo recibe como contexto una descripción del dominio —incluyendo sus acciones disponibles— seguida directamente por el problema en lenguaje natural a resolver. Toda esta información es obtenida desde

la base de recursos por dominio, construida a partir de conocimiento experto. El modelo debe entonces generar el archivo *PDDL* correspondiente, sin explicaciones intermedias.

En la modalidad con *FSP*, el agente recibe primero un ejemplo completo, que incluye la descripción en lenguaje natural de un problema representativo y su respectivo archivo *PDDL*. A continuación, se presenta un nuevo problema cuya descripción debe ser transformada, nuevamente, en su representación formal. Los ejemplos utilizados en esta variante fueron elaborados manualmente y adaptados desde la propuesta original de *LLM+P*, ajustándolos a las especificidades de los dominios del *benchmark Planetarium*.

Este diseño modular del *prompt* permite estudiar el impacto de la demostración explícita de pares problema-solución sobre la calidad del modelado automático.

2.2. Agentes modeladores propuestos

La solución propuesta consiste en un agente generador de modelos *PDDL*, diseñado para ser modular y extensible. A partir de una versión mejorada de los agentes modeladores originales —descritos en la sección anterior— se construye una base común utilizada tanto en los *baselines* reforzados como en las variantes experimentales con módulos avanzados. Esta arquitectura permite incorporar, de manera controlada, múltiples mecanismos de mejora.

El agente base recibe como entrada el identificador numérico del problema, el dominio correspondiente y su descripción en lenguaje natural. Como salida, produce el archivo *PDDL* que modela el problema y el conteo total de *tokens* consumidos en la generación.

Con el objetivo de facilitar la comparación entre variantes, se adopta una estructura de *prompt* estandarizada, sobre la cual se añaden condicionalmente diferentes bloques. El *prompt* comienza estableciendo que el modelo actúa como un agente especializado en generación de *PDDL*, y presenta al LLM el nombre y una breve descripción del dominio, junto con su definición formal en *PDDL*¹. A continuación, se enuncia la tarea: generar un modelo de problema en *PDDL* que se ajuste a los requerimientos del subconjunto correspondiente, específicamente *:strips* o *:strips + :typing*, según la presencia o no de tipos. Estos componentes se extraen de la base de recursos de dominio instanciada por el experto.

De forma opcional, se puede incluir una indicación sobre el uso de comentarios en la sección *:init* o *:goal*, cuando el módulo *Comments* está activado. También puede incorporarse una o más demostraciones mediante ejemplos completos (*Few-Shot Prompting*, módulo *FSP*), seleccionados de forma manual desde la base de recursos del dominio, o recuperados dinámicamente por un componente *Retriever* en caso de la activación del módulo *RAG*.

Cada ejemplo presenta la descripción en lenguaje natural del problema, razonamientos previos y objetos relevantes si están activos los módulos *Reasoning* y *Objects Extraction*, respectivamente, seguidos del modelo *PDDL* correspondiente.

Cuando el módulo *Insights* está habilitado, el *prompt* incluye un conjunto estructurado de conocimientos adquiridos previamente, organizados en tres categorías: conocimiento del mundo del dominio, y reglas o buenas prácticas de modelado de planificación, generales y específicas del dominio. Este conocimiento busca guiar al modelo, basado en experiencia acumulada.

En la parte final del *prompt* se presenta el nuevo problema a resolver, expresado en lenguaje natural. Si el agente pertenece al subconjunto experiencial, y no es el primer intento sobre ese problema, se añade un bloque de reflexión que incluye la salida previa generada, el *feedback* de evaluación automática, y

¹En el caso del dominio *Floor-Tile*, con inclusión de ejemplo *FSP*, se opta por omitir las acciones definidas en el *PDDL* del dominio provisto, pues se observó mejor rendimiento que al agregarlos tanto en el *prompt* de generación de *PDDL* como en el de extracción de objetos. Esto se puede deber a que estas acciones no proveen de conocimiento relevante para esas fases, sino que representan ruido innecesario. En la fase de razonamiento, sin embargo, sí contribuye positivamente al rendimiento.

una reflexión explícita generada por un sub-agente. Esta capacidad permite reintentos con autocrítica fundamentada, constituyendo un ciclo de mejora iterativa.

Finalmente, en presencia de los módulos de razonamiento y extracción de objetos, el *prompt* puede incorporar razonamientos sobre el problema, así como la lista de objetos que deberían ser utilizados, ambos generados por sub-agentes especializados en una fase previa. Estos componentes se explican a detalle en la sección siguiente. Esta integración progresiva y jerárquica de información permite una evaluación sistemática del impacto de cada técnica sobre la calidad del modelado, respetando siempre la estructura común del *prompt* para garantizar la comparabilidad entre configuraciones.

Los algoritmos de instanciación del agente modelador, su asignación de un nuevo problema a modelar, y su proceso de resolución se presentan a modo de pseudocódigos en los Anexos.

2.2.1. Razonamiento

Diversos trabajos previos han demostrado que permitir a los *LLM* realizar razonamientos previos a la ejecución de una tarea puede mejorar significativamente la calidad de sus resultados. Siguiendo esta línea de investigación, se propone una fase de razonamiento, previa y separada de la generación del modelo *PDDL*. Esta separación se realiza con el objetivo de modularizar el proceso, facilitar la inspección individual del razonamiento generado y permitir su reutilización en diferentes fases de modelado del problema. Además, el razonamiento explícito constituye una fuente valiosa de explicabilidad del proceso de generación automática, alineándose con principios de transparencia y trazabilidad.

El razonamiento se lleva a cabo por medio de un *prompt* distinto, diseñado para inducir al *LLM* a adoptar el rol de un agente especializado en análisis y razonamiento sobre la modelación de tareas de planificación. El *prompt* guía de forma estructurada al modelo para razonar en tres fases bien delimitadas: **objetos, estado inicial y estado objetivo**. Estas fases reflejan directamente la estructura del modelo *PDDL* de problema que se generará posteriormente, permitiendo que el razonamiento funcione como un soporte semántico directo para la construcción del modelo.

- En la sección de **objetos**, se requiere enumerar de forma explícita todos los objetos mencionados o inferidos a partir de la descripción del problema. Estos objetos se corresponden con los elementos declarados en la sección `:objects` del modelo *PDDL*.
- En la sección de **estado inicial**, el modelo debe describir detalladamente la situación inicial del problema. Se permite que el agente haga suposiciones razonables en caso de que la descripción del problema provea información ambigua o incompleta. Este contenido se traduce posteriormente en los predicados de la sección `:init`.
- Finalmente, en la sección de **estado objetivo**, se indica el conjunto de condiciones que deben cumplirse para considerar que el problema ha sido resuelto. Nuevamente, se induce razonamiento para completar posibles lagunas semánticas. Esta parte se traduce en la sección `:goal` del modelo *PDDL*.

El *prompt* utilizado establece el rol del modelo como un agente experto en razonamiento aplicado a planificación, y presenta el dominio con su descripción en lenguaje natural, su definición formal en *PDDL* y una descripción textual del nuevo problema. Se solicita razonar paso a paso para resolver ambigüedades y anticipar la construcción del modelo *PDDL*, mediante tres párrafos concisos con la intención anteriormente descrita. Se permite la inferencia explícita en caso de información incompleta, y se enfatiza que no debe razonarse sobre la planificación en sí, sino únicamente sobre la representación del problema. De manera análoga al *prompt* principal, pueden añadirse ejemplos previos (*FSP*), *insights*, o reflexiones de intentos anteriores, dependiendo de los módulos activos.

2.2.2. Extracción de Objetos

Con el objetivo de anclar de forma más precisa la generación del modelo *PDDL*, se propone una fase adicional de extracción estructurada de los objetos que participan en el problema de planificación. Esta fase no solo permite listar de forma clara y detallada los objetos que forman parte de los predicados en las secciones `:init` y `:goal`, sino que también facilita la construcción de una gramática más restrictiva para el *GCD*. La salida del modelo en esta fase también es restringida mediante una gramática especializada. Ambas se explican a detalle en la siguiente sección.

La extracción de objetos se realiza como una etapa independiente y posterior a la fase de razonamiento (cuando esta se encuentra activa), permitiendo aprovechar directamente la información inferida en dicha fase y garantizar la consistencia entre ambos procesos. Los objetos extraídos, así como sus posibles tipos, son representados en formato *JSON*, de forma estructurada y con el nivel de detalle necesario para apoyar tanto la generación posterior del *PDDL* como el uso de técnicas que dependen del conocimiento explícito de la estructura del problema.

El *prompt* empleado define al modelo como un agente especializado en extracción de objetos, y presenta el dominio con su descripción en lenguaje natural y su definición formal en *PDDL*. Luego, se introduce el problema a resolver, y se solicita generar una lista completa de objetos relevantes en formato *JSON*. Si el dominio es tipado, se indica que los objetos deben agruparse por tipo. Además, dependiendo de los módulos activos, el *prompt* puede incluir ejemplos previos (*FSP*), razonamiento generado previamente, o información proveniente de intentos fallidos anteriores. En particular, la reflexión sobre fallos se incluye solo si no está activa la fase de razonamiento, ya que agregar el resultado de ambas sería redundante.

2.3. Grammar-Constrained Decoding (*GCD*)

Con el objetivo de asegurar la correctitud sintáctica de los modelos *PDDL* generados, se propone el uso de la técnica de *GCD*. Esta técnica permite imponer una gramática libre de contexto a la salida generada por un modelo de lenguaje, garantizando que la estructura del texto producido respete determinadas reglas sintácticas.

Los modelos de *Fireworks AI*, utilizados en este trabajo, soportan de forma nativa la inclusión de gramáticas *GBNF* a través de su *API*. En el presente trabajo, la técnica de *GCD* es empleada para dos propósitos fundamentales: la generación de modelos *PDDL* de problema y la extracción estructurada de objetos, descrita previamente.

2.3.1. *GCD* para generación de modelos *PDDL* de problema

El primer paso consiste en construir una gramática general en formato *GBNF* que cubra el subconjunto de `:strips` + `:typing` del lenguaje *PDDL* para modelos de problema. Para ello, se parte de la definición original en *BNF* de *PDDL 3.1* (Kovacs, 2011), la cual es filtrada y adaptada cuidadosamente para ajustarse tanto al subconjunto requerido como al formato *GBNF* exigido por el sistema.

Durante este proceso se definen varios niveles de especificidad en la aplicación de la gramática:

- **Base:** se define una gramática general válida para todo dominio que utilice el subconjunto `:strips` + `:typing`.
- **Restricción por dominio:** se limita el conjunto de predicados permitidos en las secciones `:init` y `:goal` a los definidos por el dominio correspondiente.

- **Restricción por aridad:** se asegura que la aridad de los predicados utilizados coincide exactamente con la declarada en el dominio.
- **Restricción por objetos:** se restringen los objetos utilizados a los declarados explícitamente en la sección :objects y, en dominios tipados, se verifica además que los tipos asignados a los argumentos de los predicados coincidan con los tipos de los objetos correspondientes.

El nivel de especificidad de la gramática generada depende de los módulos activos en el agente modelador. Se definen dos módulos relevantes: *GCD* y *DAPS* (acrónimo de *Domain-and Problem-Specific*, es decir, “específico del dominio y del problema”). Cuando el módulo *GCD* está activo, la generación se realiza utilizando una gramática restringida. Si además está activo el módulo *DAPS*, se incluyen las restricciones más específicas relacionadas con los predicados y/u objetos particulares del problema dado.

La estructura general de la gramática *GBNF* es la siguiente:

Gramática *GBNF* general

```

root ::= define
define ::= "(" ws "define" ws problemDecl domainDecl requireDef
          objectDecl init goal ")"

problemDecl ::= "(" ws "problem" ws "<Nombre del problema>" ws ")"
ws
domainDecl ::= "(" ws ":domain" ws "<Nombre del dominio>" ws ")"
ws
objectDecl ::= "(" ws ":objects" ws "<Objetos>" ws ")"
ws
requireDef ::= "(" ws ":requirements" ws "<Requerimientos>" ws ")"
ws
init ::= "(" ws ":init" ws initEl* ")"
ws
initEl ::= literal
goal ::= "(" ws ":goal" ws preGD ")" ws

<Producción de literal>
<Producción de fórmula atómica>

preGD ::= "(and" ws GD+ ")"
<Producción de GD>

<Producciones de los objetos>
ws ::= [ \t\n]*
```

Los elementos entre signos <...> representan fragmentos variables que dependen del problema de planificación, del dominio y de los módulos activos:

- <Nombre del problema> y <Nombre del dominio> se determinan a partir del problema específico que se desea modelar.
- <Requerimientos> toma el valor :strips :typing si el dominio está tipado, y :strips en caso contrario.
- <Producción de literal> y <Producción de GD> varían según la inclusión del módulo *Comments*, que permite añadir comentarios en el modelo *PDDL*.

Si el módulo *Comments* no está activo, las producciones son:

Producciones sin módulo *Comments*

```
literal ::= atomicFormula ws
GD ::= atomicFormula ws
```

En caso contrario, se añade la posibilidad de insertar comentarios al inicio de cada predicado:

Producciones con módulo *Comments*

```
literal ::= comment atomicFormula ws | atomicFormula ws
GD ::= comment atomicFormula ws | atomicFormula ws
comment ::= ";" [^\n]* "\n"
```

2.3.2. Componentes específicos según inclusión del módulo *DAPS*

Los componentes variables de la gramática *GBNF* —específicamente <Objetos>, <Producciones de los objetos> y <Producción de fórmula atómica>— dependen directamente de la activación o no del módulo *DAPS*. A continuación se describen detalladamente ambos escenarios.

Sin inclusión del módulo *DAPS*

Cuando el módulo *DAPS* no está activo, la gramática permite una mayor libertad en la generación del modelo *PDDL*, admitiendo cualquier nombre para objetos y tipos. Los fragmentos relevantes son los siguientes:

Fragmento <Objetos> sin *DAPS*

```
typedList
```

Producciones de los objetos sin *DAPS*

```
typedList ::= <typedList RHS>
type ::= "(either" primitiveType+ ")" | primitiveType
primitiveType ::= name | "object"

name ::= letter anyChar*
letter ::= [a-zA-Z]
anyChar ::= letter | digit | "-" | "_"

digit ::= [0-9]
```

El componente <typedList RHS> (*Right-Hand Side - RHS*) depende de si el dominio es tipado:

- En dominios tipados: (name ws)+ "ws type ws)+
- En dominios no tipados: (name ws)*

Finalmente, la producción de fórmulas atómicas también es completamente general:

Producción de fórmula atómica sin *DAPS*

```
atomicFormula ::= "(" name (ws object)* ")"
```

Esta formulación no impone restricciones sobre el conjunto de predicados permitidos, ni sobre su aridad o los tipos de argumentos, por lo que es más propensa a errores semánticos, aunque garantiza la sintaxis básica.

Con inclusión del módulo *DAPS*

La activación del módulo *DAPS* permite restringir fuertemente la generación, ajustándola al dominio y problema específicos. Para ello, se utilizan tanto los predicados definidos en el dominio, incluyendo su aridad (o tipos de argumentos en caso de dominios tipados), como los objetos obtenidos durante la fase de extracción estructurada.

Dominios no tipados En estos casos, <Objetos> se representa como una lista de nombres separados por espacio, entre comillas dobles. Por ejemplo, si los objetos extraídos de un problema de *Blocksworld* son [b1, b2, b3] el fragmento queda:

Ejemplo de fragmento <Objetos> no tipado con *DAPS*

```
"b1 b2 b3"
```

<Producciones de los objetos> se sustituye por:

Fragmento <Objetos> no tipado con *DAPS*

```
object ::= "<obj1>" | "<obj2>" | ... | "<objn>"
```

Donde <obj1>, <obj2>, ..., <objn> son los nombres de los objetos determinados en la fase de extracción, cada uno entre comillas dobles y todos unidos mediante el operador de disyunción |. La producción correspondiente al ejemplo anterior restringe de forma explícita los objetos disponibles:

Ejemplo de producciones de los objetos no tipados con *DAPS*

```
object ::= "b1" | "b2" | "b3"
```

<Producción de fórmula atómica> considera cada predicado posible y su aridad exacta. En este caso empieza con el prefijo *atomicFormula* ::=, y por cada predicado proporcionado del dominio se construyen las opciones posibles de fórmulas atómicas, unidas por el operador de disyunción |. Cada opción es de la forma:

Producción de fórmula atómica en dominios no tipados con *DAPS*

```
"(<Nombre del predicado>
  (ws object){<Aridad del predicado>} ")"
```

Donde <Nombre del predicado> y <Aridad del predicado> se sustituyen por sus valores correspondientes. Por ejemplo, para el predicado (on ?x ?y) del dominio *Blocksworld*, se construye el siguiente fragmento:

Ejemplo de producción de fórmula atómica con *DAPS* (no tipado)

```
atomicFormula ::= "(on" ws (object){2} ")"
```

Dominios tipados En dominios que incluyen tipado, <Objetos> se organiza por tipo, de la forma:

Objetos en dominios tipados con *DAPS*

```
"<Objetos de tipo 1> <Objetos de tipo 2> ... <Objetos de tipo m>"
```

Donde cada fragmento <Objetos de tipo i> es una lista con los nombres de los objetos de dicho tipo, y terminada con el sufijo - <Nombre del tipo>. Por ejemplo, en *Floor-Tile*, sean los objetos extraídos:

Ejemplo de <Objetos> en dominios tipados con *DAPS*

```
{
  "tile": ["tile1", "tile2", "tile3", "tile4"],
  "robot": ["robot1", "robot2"],
  "color": ["color1"]
}
```

El fragmento queda:

Fragmento <Objetos> en dominios tipados con *DAPS*

```
"tile1 tile2 tile3 tile4 - tile robot1 robot2 - robot color1 - color"
```

Las producciones de objetos y de fórmulas atómicas se restringen, además, en función de los tipos esperados por los predicados.

Esto permite al modelo generar únicamente estructuras válidas desde el punto de vista sintáctico y semántico, respetando tanto los tipos como las aridades esperadas.

<Producciones de los objetos> es un conjunto de producciones (una por cada tipo del dominio) separadas por líneas de la forma:

Producciones de los objetos por tipo (tipado con *DAPS*)

```
obj-<Nombre del tipo> ::= "<obj1>" | "<obj2>" | ... | "<objn>"
```

Donde <obj1>, <obj2>, ..., <objn> son los nombres de los objetos extraídos cuya clasificación corresponde a ese tipo o a alguno de sus subtipos. Siguiendo el ejemplo anterior, el fragmento resultante sería:

Ejemplo de producciones por tipo

```
obj-tile ::= "tile1" | "tile2" | "tile3" | "tile4"
obj-robot ::= "robot1" | "robot2"
obj-color ::= "color1"
```

Estas producciones asumen que cada objeto pertenece a exactamente un tipo. Asimismo, se asume una jerarquía de tipos sin multiherencia, lo cual es una simplificación válida dado que, aunque el lenguaje

PDDL permite multiherencia, la gran mayoría de dominios utilizados en *benchmarks* estándar como *IPC* no la emplean. Esta suposición permite construir un árbol de herencia de tipos con `object` como raíz.

Formalmente, si se tiene un conjunto de tipos T y una relación de herencia $\prec \subseteq T \times T$, donde $t_1 \prec t_2$ indica que t_1 es subtipo de t_2 , entonces el conjunto de objetos incluidos en la producción `obj-t` es:

$$\mathcal{O}_t = \{o \in O \mid \text{type}(o) = t' \text{ y } t' \sqsubseteq t\}$$

donde \sqsubseteq es la clausura reflexiva y transitiva de \prec , y `type(o)` es el tipo de o .

La <Producción de fórmula atómica> también se adapta para aprovechar esta estructura. Se inicia con el prefijo:

Estructura general de `atomicFormula`

```
atomicFormula ::=
```

y por cada predicado definido en el dominio, se añade una opción con su nombre y las producciones de objetos correspondientes a los tipos de sus argumentos. Cada opción toma la forma:

Plantilla por predicado (tipado)

```
"(<Nombre del predicado>" ws obj-<Tipo1> ws ... ws obj-<Tipok> ws ")"
```

Por ejemplo, para el predicado (`robot-at ?r - robot ?x - tile`), la producción correspondiente sería:

Ejemplo `atomicFormula` para predicado tipado

```
"(robot-at" ws obj-robot ws obj-tile ws ")"
```

Esto permite una validación estricta no solo sintáctica, sino también semántica, al restringir los objetos posibles según su tipo esperado.

2.3.3. *GCD* para extracción de objetos en dominios no tipados

Para asegurar la validez sintáctica en la fase de extracción estructurada de objetos cuando el dominio no es tipado, se construye también una gramática *GBNF* dedicada a esta tarea. La gramática es diseñada para aceptar exclusivamente cadenas que representaran una estructura tipo *JSON* con una sola clave llamada `objects`, cuyo valor es una lista de cadenas que representaran los nombres de los objetos:

Gramática para extracción de objetos (no tipado)

```
root ::= "{" dq "objects" dq ":" ["] obj (", " obj)* "] " "}""
obj ::= dq letter anyChar* dq

letter ::= [a-zA-Z]
anyChar ::= letter | digit | "-" | "_"
digit ::= [0-9]
dq ::= "\\\\""
```

Este esquema asegura una estructura válida en la respuesta del modelo, acorde con el formato esperado por los módulos posteriores. Por ejemplo, en un problema del dominio *Blocksworld* que involucra los bloques b1, b2, b3 y b4, la salida generada por el agente tiene la forma:

Ejemplo de salida válida (no tipado)

```
{"objects": ["b1", "b2", "b3", "b4"]}
```

De esta manera, la aplicación de *GCD* a esta fase también garantiza una representación formalmente consistente de los objetos extraídos, lo que resulta clave para los pasos posteriores en la construcción del problema en *PDDL*.

2.3.4. *GCD* para extracción de objetos en dominios tipados

Cuando el dominio de planificación es tipado, se utiliza una gramática específica para la fase de extracción de objetos. Esta gramática permite representar una estructura tipo *JSON* en la que cada tipo del dominio se asocia con una lista de objetos del problema actual. La gramática general propuesta es la siguiente:

Gramática para extracción de objetos (dominios tipados)

```
root ::= <Descripción del diccionario de objetos>
obj-list ::= "[" obj (", " obj)* "]"
obj ::= dq letter anyChar* dq

letter ::= [a-zA-Z]
anyChar ::= letter | digit | "-" | "_"
digit ::= [0-9]

tab ::= "\t"
dq ::= "\\\""
el ::= "\n"
```

El símbolo <Descripción del diccionario de objetos> se define de acuerdo con los tipos presentes en el dominio específico, y representa una cadena *JSON* formateada en múltiples líneas. Cada línea corresponde a una clave del diccionario (el nombre de un tipo) y su valor asociado (una lista de objetos). La construcción completa sigue el siguiente patrón:

Estructura de diccionario por tipo

```
"{\n"
  tab dq "<Nombre del tipo 1>" dq ":" obj-list ","
  ...
  tab dq "<Nombre del tipo m-1>" dq ":" obj-list ","
  tab dq "<Nombre del tipo m>" dq ":" obj-list
"\n}"
```

Por ejemplo, si se considera un problema del dominio *Floor-Tile* con una matriz de 2×2 (con las celdas tile1, tile2, tile3, tile4), dos robots (robot1, robot2) y un color disponible (color1), la salida generada por el modelo para esta fase es:

Ejemplo de salida de objetos extraídos (tipado)

```
{
  "tile": ["tile1", "tile2", "tile3", "tile4"],
  "robot": ["robot1", "robot2"],
  "color": ["color1"]
}
```

Esta salida no solo facilita el uso posterior de los objetos en la generación de modelo *PDDL*, sino que también posibilita la construcción de una gramática adaptada al problema específico, como se detalló en la sección anterior.

2.4. Evaluación de modelos *PDDL* y planes

Una vez generado el modelo *PDDL*, se procede a su evaluación mediante las métricas proporcionadas por *Planetarium* y otras métricas adicionales diseñadas específicamente para este trabajo, de carácter parcial.

La función `planetarium.evaluate` para un problema recibe como parámetros el *PDDL ground truth* (referencia correcta), el *PDDL* a evaluar, un valor booleano que indica si se desea evaluar la *solubilidad* (*solvability*), y otro booleano para el parámetro `is_placeholder`, que define cómo se compara el problema evaluado con la referencia. Cuando `is_placeholder` es verdadero, la función ignora la identidad específica de los objetos, permitiendo que cualquier permutación de ellos que cumpla los estados iniciales y finales sea considerada correcta. Esto es útil para problemas abstractos o donde los objetos son intercambiables, como en el caso de construir una torre de una altura específica sin importar qué bloques se utilizan. En cambio, cuando `is_placeholder` es falso, se evalúa que los objetos individuales en los estados inicial y final del problema evaluado correspondan exactamente a los de la referencia. Este caso considera la identidad y correspondencia precisa de los objetos involucrados en el problema. La salida de esta función consta de tres valores booleanos que indican los valores de validez sintáctica, solubilidad y correctitud del *PDDL*.

Recordando brevemente el funcionamiento de esta evaluación, primero se verifica que el *PDDL* sea sintácticamente válido, luego que sea posible resolverlo con un planificador automático, y finalmente que el plan generado sea correcto según la referencia establecida.

Además de estas métricas estándar, se proponen otras evaluaciones parciales que permitieron un análisis más detallado del desempeño del agente modelador:

Se evalúa que la cantidad de objetos extraídos, agrupados por tipo en el caso de dominios tipados, coincida con la referencia. Para esta tarea se diseñó un método sencillo que permite extraer automáticamente los objetos desde la sección correspondiente de un modelo *PDDL* de problema. El método consiste en localizar la sección `:objects` y recorrerla hasta el cierre (detección de un paréntesis cerrado) almacenando los objetos definidos, incluyendo sus tipos si estos se encuentran declarados (tras un guión).

En los casos en que el modelo *PDDL* resulta sintácticamente válido pero no soluble, se analiza la posible presencia de predicados conflictivos o contradictorios en las secciones `:init` o `:goal` o en ambas. Para ello, se generan versiones del problema en las que los predicados de una de estas secciones se replican tanto en `:init` como en `:goal`, y se evalúa la solubilidad de esta instancia con predicados idénticos en ambas secciones. De esta manera, se puede determinar si las contradicciones internas en alguno de los estados inicial o objetivo provocan la insatisfacibilidad del problema. Para la evaluación se compara el *PDDL* modificado consigo mismo, tomándolo como el *ground truth* y la predicción simultáneamente. Cabe destacar que no todas las contradicciones se detectan debido a limitaciones impuestas por las restricciones

del dominio o el funcionamiento de los planificadores automáticos, y no se profundizó en el estudio de estas causas, pues escapan del enfoque central de esta tesis. Sin embargo, esta evaluación adicional resulta útil para identificar con mayor precisión el origen del fallo, y permite construir un *feedback* (retroalimentación) más específico para guiar la reflexión y el reintento de los agentes modeladores.

Cuando el modelo *PDDL* es sintácticamente válido y soluble, pero incorrecto, se aplica un procedimiento análogo al anterior para detectar cuál o cuáles estados (`:init` y/o `:goal`) son responsables del error. Este análisis consiste en duplicar los predicados de cada estado dentro de ambos estados (inicial y objetivo) en el *PDDL* generado y en el *ground truth*, y luego evaluar la correctitud en base a esta modificación. La idea fundamental es que la correctitud se determina analizando el isomorfismo entre los grafos de escena generados a partir de los estados inicial y objetivo. Al replicar los predicados, se aisla el problema y se logra detectar cuando uno de los estados generados incumple independientemente la estructura o semántica esperada. No obstante, este método no es infalible, pues existen casos en los que ambos estados resultan correctos individualmente, pero su combinación genera un problema incorrecto. Este fenómeno se observó en configuraciones específicas, denominadas en *Planetarium* como `strictly_both: swap` e `invert` en *Blocksworld*, `swap` y `juggle` en *Gripper*, y en la configuración `disconnected_rows` en *Floor-Tile*. A pesar de esta limitación, la evaluación parcial adicional proporciona información valiosa para detectar fallos más precisos y construir *feedback* automático detallado que facilita la mejora iterativa de los agentes.

Dependiendo de los resultados de estas evaluaciones, se genera un texto en lenguaje natural que describe la naturaleza de los errores o confirma la correctitud del *PDDL* evaluado, proporcionando así un *feedback* comprensible y orientado a la mejora.

2.5. Agente experiencial

En esta tesis se diseñaron estrategias inspiradas en el trabajo presentado por *ExpeL* (A. Zhao et al., 2024), que propone un enfoque para el aprendizaje autónomo en agentes basados en *LLMs*. En particular, *ExpeL* destaca la relevancia de una fase de entrenamiento en la que el agente recopila experiencias consistentes en soluciones correctas a problemas de planificación, así como conocimientos útiles o *insights* relacionados con reglas, buenas prácticas de modelación y características específicas del dominio. Además, enfatiza la importancia de que el agente pueda reflexionar sobre los errores cometidos durante intentos fallidos para corregirlos posteriormente y potenciar así la calidad de su aprendizaje.

Siguiendo esta línea, el presente trabajo adopta esta idea para diseñar un agente capaz de acumular experiencias y extraer *insights* de modelación de planificación, de manera autónoma durante una fase de entrenamiento, y luego emplearlos para mejorar la toma de decisiones en tareas no vistas.

Durante la fase de entrenamiento, el agente interactúa con el entorno mediante un proceso iterativo de prueba y error, almacenando las experiencias en un *pool* o repositorio de experiencias, siguiendo la noción clásica propuesta por (Lin, 1992). A partir de este *pool*, el agente extrae *insights* de forma similar al *off-policy learning* (Watkins y Dayan, 1992), donde es posible aprender de comportamientos pasados para mejorar la política *policy* actual. Posteriormente, durante la evaluación, el agente utiliza estas experiencias e *insights* para afrontar nuevas tareas, mejorando su desempeño sin requerir ajustes adicionales en el modelo.

De esta forma, la presente investigación reutiliza y adapta el marco conceptual y metodológico de *ExpeL*, demostrando la viabilidad de incorporar un agente experiencial de modelación que aprende autónomamente de sus propias interacciones, y que puede transferir ese conocimiento a nuevas tareas, aportando a la mejora continua de la generación automática de modelos *PDDL*.

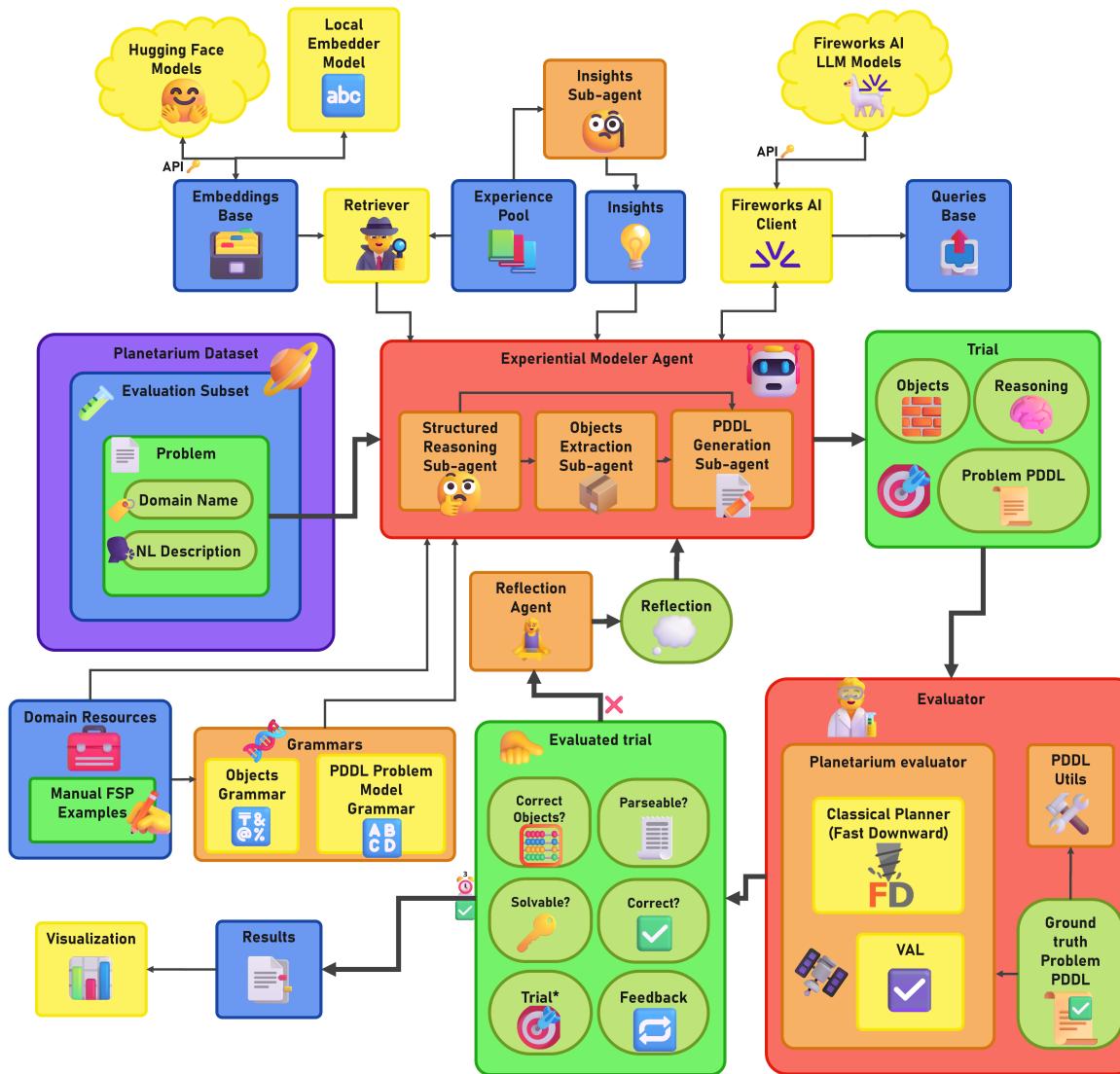


Figura 2.2: Flujo de evaluación de los agentes modeladores experienciales propuestos.

2.5.1. Reflexión

Se propone un agente especializado en reflexionar sobre intentos fallidos de generación de modelos *PDDL* de problemas de planificación. Su propósito es analizar los errores cometidos por un agente modelador al generar un modelo de problema, y proponer posibles correcciones a partir del análisis de retroalimentación proporcionado por la evaluación formal descrita en secciones anteriores.

Este agente de reflexión opera a partir de un *prompt* cuidadosamente diseñado que sintetiza toda la información relevante del intento fallido, incluyendo la descripción del dominio, el modelo *PDDL* del dominio, la descripción natural del problema, el razonamiento previo del agente modelador, los objetos utilizados, el modelo *PDDL* generado en el intento fallido, y el *feedback* obtenido tras la evaluación. Con base en esta información, el agente debe inferir la posible causa del fallo y sugerir modificaciones, de manera razonada y específica.

El *prompt* instruye al modelo para que actúe como un agente experto en modelado que reflexiona sobre errores cometidos en la generación de problemas de planificación. Se le presenta un intento fallido de modelado junto con una descripción completa del contexto: dominio (nombre, explicación breve y modelo *PDDL*), descripción natural del problema, razonamiento generado y objetos utilizados (si aplican), modelo generado y retroalimentación sobre el error. El agente debe considerar una jerarquía de posibles errores —errores sintácticos, problemas de solubilidad y errores semánticos— y entender que errores de menor nivel invalidan automáticamente la posibilidad de evaluar los de nivel superior (por ejemplo, un modelo con errores sintácticos no puede ser evaluado semánticamente)².

El modelo debe generar un único párrafo en el que explique de forma razonada cuál pudo haber sido la causa del fallo, e indique explícitamente qué elementos deben ser corregidos: el estado inicial, el estado objetivo, o ambos. Esta reflexión se integra como una fase posterior al proceso de evaluación y puede ser utilizada por los agentes modeladores para fundamentar nuevos intentos. La capacidad de razonar sobre errores, explicarlos y proponer soluciones convierte a esta etapa en un componente crucial del ciclo iterativo de mejora basado en retroalimentación estructurada.

2.6. Entrenamiento

Con el objetivo de acumular conocimiento útil para abordar tareas futuras, se diseña un proceso de entrenamiento para el agente modelador experiencial. Este proceso consiste en dos fases bien diferenciadas: una fase de acumulación de experiencias y una fase de extracción de *insights*.

La inspiración conceptual y metodológica para esta etapa proviene del trabajo presentado en *ExpeL* (A. Zhao et al., 2024), el cual introdujo una estrategia sistemática de entrenamiento mediante intentos múltiples, autorreflexión y reutilización de trayectorias pasadas, tanto como ejemplos de *FSP* como para la extracción de *insights*.

En esta tesis se reutiliza y adapta dicho enfoque para el entrenamiento del agente modelador experiencial propuesto. Concretamente, se selecciona un subconjunto del *dataset Planetarium* que tenga poca intersección con el subconjunto reservado para evaluación. Sobre este subconjunto se ejecuta un proceso iterativo, en el cual el agente intenta resolver cada una de las tareas de planificación como máximo tres veces, o hasta lograr una solución correcta.

Durante cada intento, el agente recibe como entrada la descripción natural del problema, el dominio *PDDL* correspondiente, y un conjunto de ejemplos *few-shot* fijos. En el primer intento, no se proporciona ninguna reflexión adicional. Sin embargo, si el intento falla, se activa el agente reflexionador descrito previamente, el cual genera una reflexión en lenguaje natural sobre las causas del fallo. Esta reflexión se incorpora al *prompt* del siguiente intento, permitiendo al agente modelador considerar sus errores pasados y adaptar su estrategia en consecuencia. Este mecanismo de retroalimentación iterativa resulta clave para mejorar la tasa de éxito en intentos subsiguientes.

Todas las trayectorias generadas —ya sean fallidas o exitosas— son almacenadas en un *Experience Pool* (repositorio de experiencias), que constituye una memoria a largo plazo del agente. Esta base de datos de experiencias tiene una doble función en el sistema. Por una parte, se utiliza como fuente para la recuperación de ejemplos exitosos mediante *RAG* durante la fase de evaluación. Por otra, sirve como insumo para la fase de extracción de *insights*, donde se analizan varias experiencias previas para identificar patrones útiles y generar recomendaciones generalizables. Esta segunda fase será detallada en la subsección siguiente.

²Al menos bajo el alcance de esta tesis no es posible hacerlo de forma automática y determinista, pero sí sería posible que en un modelo con errores sintácticos un humano experto identifique el significado semántico subyacente y sus posibles errores.

Adicionalmente, se propone la incorporación de *insights* agregados manualmente por expertos humanos, lo cual permite contrastar el conocimiento extraído automáticamente con el criterio experto, e incluso enriquecer el *corpus* de recomendaciones disponibles durante la evaluación.

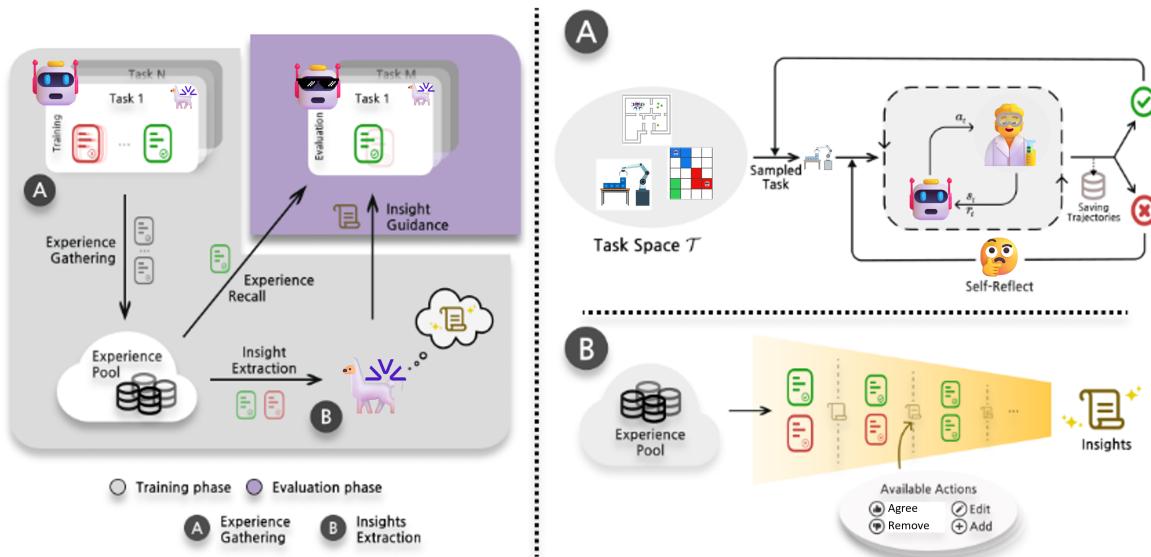


Figura 2.3: Imagen adaptada del *paper* de *ExpeL* (A. Zhao et al., 2024). Izquierda: se opera en tres etapas: (1) Recolección de experiencias de éxito y fracaso en un *pool*. (2) Extracción/abstracción de conocimiento entre tareas a partir de estas experiencias. (3) Aplicación de los conocimientos adquiridos y recuerdo de éxitos pasados en tareas de evaluación. Derecha: (A) Ilustra el proceso de recolección de experiencias a través del agente reflexionador, permitiendo reintentos de la tarea después de la autorreflexión sobre los fracasos. (B) Ilustra el paso de extracción de conocimiento. Cuando se le presentan pares de éxito/fracaso o una lista de L éxitos, el agente modifica dinámicamente una lista existente de conocimientos \hat{t} utilizando las operaciones ADD, AGREE, REMOVE y EDIT.

A continuación se presenta el pseudocódigo que describe el proceso de entrenamiento:

Algoritmo 2.1: Entrenamiento del agente experiencial

Input :

Agente modelador \mathcal{A}_{mod}
 Conjunto de tareas de entrenamiento $\mathcal{T}_{\text{train}}$
 Número máximo de intentos por tarea Z
 Indicador de continuación de entrenamiento previo: `resume`
 Indicador de retroalimentación humana: `human_feedback`

Output:

Conjunto acumulado de experiencias B

```

1 Inicializar ruta de progreso y cargar o reiniciar estado;
2 Inicializar conjunto de experiencias:  $B \leftarrow \emptyset$ ;
3 si resume entonces
4   | Cargar índice de tarea  $i$  desde archivo de progreso;
5 en otro caso
6   |  $i \leftarrow 0$ ;
7   | Guardar progreso inicial en disco;
8 fin
9 para  $i = i$  a  $|\mathcal{T}_{\text{train}}| - 1$  hacer
10  |  $t_i \leftarrow \mathcal{T}_{\text{train}}[i]$ ;
11  | Inicializar reflexiones previas:  $\lambda \leftarrow \emptyset$ ;
12  | Llamar  $\mathcal{A}_{\text{mod}}.\text{asignar\_tarea}(t_i)$ ;
13  | para  $z = 0$  a  $Z - 1$  hacer
14    |   Resolver tarea:  $\mathcal{P}_{\pi}^{(z)} \leftarrow \mathcal{A}_{\text{mod}}.\text{resolver}()$ ;
15    |   Evaluar resultado:  $e_{i,z} \leftarrow \text{evaluar}(t_i, \mathcal{P}_{\pi}^{(z)})$ ;
16    |   Guardar experiencia:  $B \leftarrow B \cup \text{store\_exp}(t_i, z, \lambda[z-1], \mathcal{P}_{\pi}^{(z)}, e_{i,z})$ 
17    |   si  $e_{i,z}.\text{correct} = \text{True}$  entonces
18      |     | break;
19    |   en otro caso
20      |     | si  $z = Z - 1$  y human_feedback entonces
21        |       | Mostrar evaluación fallida y solicitar entrada humana;
22        |       |  $\nu_z \leftarrow \text{input}()$ ;
23    |   en otro caso
24      |     | Generar reflexión automática:  $\nu_z \leftarrow \text{reflect}(\mathcal{A}_{\text{mod}}.P_{\text{desc}}, \mathcal{A}_{\text{mod}}.D, \lambda, \mathcal{P}_{\pi}^{(z)}, e_{i,z})$ 
25    |   fin
26    |   Agregar reflexión:  $\lambda \leftarrow \lambda \cup \{\nu_z\}$ ;
27    |   Actualizar  $\mathcal{A}_{\text{mod}}.\tau \leftarrow \mathcal{A}_{\text{mod}}.\tau \cup (\mathcal{P}_{\pi}^{(z)}, e_{i,z}, \nu_z)$ ;
28  | fin
29 fin
30 Insertar  $B$  en base persistente;
31 Actualizar archivo de progreso con índice  $i$ ;
32 fin
33 Marcar entrenamiento como completado en archivo de progreso;
34 devolver  $B$ ;

```

2.6.1. Fase de extracción de *insights*

Una vez concluida la fase de acumulación de experiencias, se procede a una segunda fase centrada en la generación y depuración de *insights*, con el propósito de sintetizar lecciones aprendidas a partir de las trayectorias recopiladas. Esta fase busca abstraer conocimientos útiles tanto específicos del dominio como generales, que puedan ser aprovechados por el agente modelador durante la evaluación para mejorar su desempeño.

En esta tesis se retoma la propuesta metodológica presentada en *ExpeL* (A. Zhao et al., 2024), donde se argumenta que un agente puede analizar sus experiencias de manera sistemática para extraer conocimiento valioso. En particular, se identificaron dos mecanismos complementarios de análisis. El primero consiste en comparar una trayectoria fallida con una trayectoria exitosa sobre la misma tarea, lo cual permite identificar de forma concreta las acciones incorrectas en contraste con las correctas. El segundo mecanismo se basa en el análisis de patrones comunes entre múltiples trayectorias exitosas, provenientes de distintas tareas dentro de un mismo dominio. Esta segunda forma de análisis permite detectar buenas prácticas recurrentes, que pueden ser generalizadas como reglas efectivas de modelación.

Tomando como base dicha propuesta, se estructura la fase de extracción de *insights* en dos modos operativos principales:

- **Comparación entre intentos fallidos y exitosos del mismo problema.** Para cada problema del conjunto de entrenamiento resuelto con éxito, se identifican todos los intentos fallidos previos al exitoso realizados por el agente. Con estos datos se construyen pares de la forma (*incorrecto_i*, *correcto*), donde cada *incorrecto_i* representa una trayectoria fallida del mismo problema, y *correcto* corresponde a la solución exitosa. Estos pares permiten al sistema inferir causas de error específicas, reconocer desviaciones de comportamientos correctos y extraer recomendaciones de mejora concretas.
- **Comparación de múltiples soluciones correctas dentro de un mismo dominio.** Para cada dominio de planificación, se coleccionan todas las trayectorias exitosas generadas durante la fase anterior. Estas soluciones se agrupan en lotes (*batches*) de tamaño configurable; en este trabajo se emplean *batches* de tamaño 2. Estos lotes son utilizados para identificar patrones comunes entre problemas diferentes pero enmarcados en un mismo contexto semántico. De esta forma, se buscan regularidades y estructuras repetidas que reflejen buenas prácticas aplicables a otros problemas del dominio.

El sistema responsable de esta fase —denominado Agente de *Insights*— recibe como entrada ambos conjuntos: los pares (*incorrecto_i*, *correcto*) y los *batches* de trayectorias exitosas. A partir de ellos, es capaz de actualizar progresivamente su base interna de *insights*, acumulando información que luego podría emplearse para guiar al agente modelador durante la evaluación.

Los *insights* extraídos se clasifican en tres categorías distintas:

- **Conocimiento del mundo:** Incluye información semántica específica sobre el dominio, como el significado y uso típico de predicados, restricciones físicas o abstractas implícitas, y relaciones relevantes entre objetos. Este tipo de conocimiento permite al agente comprender mejor la estructura lógica subyacente al dominio de planificación.
- **Reglas para la planificación en el dominio:** Contiene estrategias específicas de modelación aprendidas por observación. Por ejemplo, el uso correcto de ciertos predicados en la formulación de problemas, la inclusión necesaria de ciertos objetos o acciones, o la estructuración típica de los objetivos en ese dominio.

- **Reglas generales de modelación:** Se refiere a principios que resultan útiles en múltiples dominios, y que no dependen de un contexto semántico específico. Estos incluyen, por ejemplo, recomendaciones sobre la forma de declarar objetos, cómo seleccionar hechos iniciales relevantes, o patrones efectivos en la formulación de metas.

La base de *insights* se inicializa vacía, y es construida de forma incremental a medida que se procesan los conjuntos de experiencias mencionados. El agente de extracción evalúa la utilidad y relevancia de cada observación, descartando aquellas poco informativas o irrelevantes, y consolidando las más consistentes o frecuentes. Esta base se diseña como un componente reutilizable, que puede enriquecer progresivamente su contenido con nuevas experiencias de entrenamiento o intervención humana experta.

Este mecanismo de análisis diferenciado de errores y éxitos, junto con la categorización estructurada del conocimiento, permite dotar al sistema de una memoria inferencial rica, orientada a facilitar la generalización y la transferencia de aprendizajes durante la fase de evaluación.

2.7. Agente de *Insights*

Como se explicó en la sección anterior, los *insights* representan unidades de conocimiento estructurado, útiles para la modelación de problemas de planificación automática. Cada *insight*, independientemente de su tipo (conocimiento del mundo, reglas del dominio o reglas generales), se representa mediante una breve descripción textual y un valor numérico que indica su relevancia acumulada, interpretado como una medida de confianza, utilidad o frecuencia.

Con el objetivo de mantener y actualizar este conjunto de *insights*, se propone un sistema especializado: el *Agente de Insights*. Este agente opera sobre la base de un conjunto de acciones bien definidas que le permiten construir, refinar y filtrar progresivamente el conocimiento extraído de las experiencias almacenadas.

Este enfoque toma como referencia directa el mecanismo descrito por *ExpeL* (A. Zhao et al., 2024), donde se propone iniciar con un conjunto vacío de *insights*, y actualizarlo de forma iterativa a partir del análisis de pares (*fallo, éxito*) o listas de trayectorias exitosas, previamente recolectadas en la fase de acumulación de experiencias. En cada iteración, el modelo de lenguaje recibe alguno de estos conjuntos como contexto, y debe decidir qué operaciones aplicar sobre los *insights* actuales.

Las operaciones permitidas por el agente son las siguientes:

- **ADD** (agregar): introduce un nuevo *insight* que no esté representado aún en la base. Este nuevo *insight* es añadido con el texto sugerido por el agente, y un valor inicial de relevancia igual a 2. Esta operación permite extender el conocimiento acumulado.
- **EDIT** (editar): modifica el contenido textual de un *insight* ya existente, con el objetivo de mejorar su claridad, generalidad o aplicabilidad. Esta operación también incrementa su valor numérico en 1, como señal de reafirmación de su utilidad tras la edición.
- **AGREE** (confirmar): reafirma la validez de un *insight* existente, confirmando que sigue siendo relevante para los ejemplos analizados. Esta operación incrementa su valor numérico en 1.
- **REMOVE** (eliminar o devaluar): indica desacuerdo con un *insight* actual, por considerarlo erróneo, irrelevante, redundante, demasiado específico, o *superseded* (superado por otro más general o actualizado). Esta operación disminuye su valor numérico. Para controlar la sobreacumulación de *insights*, se define un umbral *M* (establecido en 10). Si el número de *insights* de un tipo dado es mayor o igual a *M*, esta operación disminuye el valor numérico del *insight* seleccionado en 3 unidades; en caso

contrario, la disminución es de 1. Si el valor resultante no es positivo, el *insight* es eliminado de la base.

Esta política de actualización se diseña para ser robusta frente a sesgos o errores puntuales en las trayectorias, permitiendo reforzar progresivamente aquellos *insights* que son validados de forma reiterada en contextos distintos, y depurar los que no resultan útiles o se contradicen con ejemplos posteriores. Este diseño también considera que incluso trayectorias exitosas pueden ser subóptimas, y por tanto sus recomendaciones deben ser sometidas a verificación iterativa.

Además, para mantener la estabilidad y eficiencia del sistema, se establece un número máximo de operaciones por tipo de *insight* durante cada sesión de procesamiento. En este trabajo, se fija un límite de 4 operaciones por tipo, lo cual obliga al agente a priorizar sus modificaciones más relevantes en cada interacción.

Mediante este mecanismo, se pretende que el *Agente de Insights* construya de manera incremental una base de conocimientos estructurados y validados, capaz de asistir al agente modelador durante la etapa de evaluación, tanto en la selección de hechos iniciales como en la formulación de objetivos o en la verificación de consistencia semántica.

Para permitir que el agente opere sobre conjuntos complejos y estructurados de conocimiento, se concibe un *prompt* parametrizable y flexible que guía el comportamiento del modelo de lenguaje durante la fase de generación o depuración de *insights*. Este *prompt* adopta una estructura instruccional clara, definiendo el contexto del dominio, las reglas operacionales, las limitaciones por tipo de *insight*, y la información sobre los intentos previos que deben ser tenidos en cuenta.

Al igual que en agentes anteriores, algunos componentes del *prompt* son completados dinámicamente en función del dominio en cuestión, del estado actual de la base de *insights* y de las experiencias a analizar. El nombre del dominio, su descripción breve en lenguaje natural y su correspondiente modelo en *PDDL* se extraen directamente de la base de datos del dominio, y constituyen el contexto sobre el que se espera que el agente razonne. Junto a ello, se incluye el conjunto actual de *insights* organizados en las tres categorías establecidas (conocimiento del mundo, reglas específicas del dominio y reglas generales), presentados de forma numerada para su referencia.

El *prompt* también informa al agente sobre el límite máximo de operaciones por tipo de *insight*, y define de forma explícita los elementos que debe considerar al tomar decisiones, como reflexiones, retroalimentaciones de evaluación, o trayectorias exitosas y/o fallidas previas. Según el procedimiento que haya generado ese conjunto de entrada, se incorpora también una descripción del modo de extracción, y si se detecta una cantidad excesiva de *insights* de cierto tipo (es decir, si se ha superado un umbral establecido), se añade una instrucción adicional que obliga al agente a priorizar operaciones de depuración o eliminación, en lugar de seguir incrementando la base de conocimiento.

Este *prompt* es utilizado durante la fase de construcción del conocimiento derivado, permitiendo al agente analizar críticamente los datos acumulados y refinar su base de *insights* de manera progresiva, controlada y fundamentada.

Comparación de un par de intentos de un mismo problema, uno fallido y uno exitoso

Uno de los mecanismos de extracción de *insights* es la comparación directa entre un intento fallido y uno exitoso sobre el mismo problema. Este enfoque permite al agente identificar cambios clave que condujeron a una solución correcta, extrayendo lecciones reutilizables que pueden informar futuros intentos. Esta estrategia se inspira directamente en el enfoque de *ExpeL*, donde se formaliza la reflexión comparativa como una técnica eficaz para detectar errores frecuentes, sistematizar buenas prácticas y fortalecer el desempeño del agente a lo largo del tiempo.

En este modo de operación, el *prompt* proporcionado al agente incluye información detallada sobre ambos intentos. Para definir el contexto, se especifica que se trata de dos intentos —uno exitoso y otro fallido— de una misma tarea de planificación, en los cuales el agente modelador recibió siempre la misma descripción del dominio, el archivo *PDDL* del dominio, y una descripción en lenguaje natural del problema. Esta información contextual se incluye con una redacción explícita que enmarca la tarea de contraste.

Luego, se le indica al modelo que debe enfocar su análisis en patrones, errores recurrentes o lecciones que puedan deducirse al comparar ambos intentos. El bloque posterior incluye una representación completa del intento fallido —con razonamiento y objetos seleccionados (si aplican), el modelo *PDDL* generado y la retroalimentación obtenida— así como la reflexión generada tras la evaluación si esta se encuentra disponible. A continuación, se presenta también el intento exitoso con su salida generada, lo que permite una visión paralela de las decisiones tomadas antes y después de corregir los errores. Esta estructura está pensada para proporcionar un marco claro al agente, en el que puede detectar, comparar y abstraer conocimiento útil a partir de los contrastes observados.

Comparación de varias soluciones correctas de distintos problemas del mismo dominio

Además del anterior, se incorpora un modo de extracción que opera sobre múltiples intentos exitosos correspondientes a distintos problemas del mismo dominio. Este enfoque busca generalizar buenas prácticas a partir de trayectorias que han demostrado ser válidas, capturando regularidades que puedan aplicarse a nuevas instancias similares. De nuevo, este procedimiento sigue las recomendaciones de *ExpeL*, donde se argumenta que las soluciones correctas también son fuentes valiosas de conocimiento estructurado.

En este caso, el *prompt* indica explícitamente que todas las entradas corresponden a intentos exitosos en un mismo dominio, manteniendo el contexto —la descripción del dominio y su modelo en *PDDL*— solo variando la formulación del problema en lenguaje natural, para cada tarea analizada. Esta información se incorpora consecutivamente, y prepara al agente para analizar los datos bajo un criterio común.

El conjunto de datos presentado al agente incluye varios intentos exitosos, cada uno estructurado de forma consistente, con información sobre el problema, el razonamiento realizado y los objetos determinados (si aplican), y el modelo generado. Una instrucción dada orienta al modelo a identificar patrones o lecciones presentes en estas soluciones válidas, permitiéndole extraer *insights* que refuerzen estrategias útiles y recurrentes dentro del mismo dominio de planificación.

2.8. RAG

Durante la fase de acumulación de experiencias, se generan múltiples soluciones correctas de modelación de problemas de planificación. Estas soluciones exitosas se estiman útiles como ejemplos de *FSP* para resolver nuevos problemas, mediante *RAG*.

Cuando se presenta una nueva tarea a resolver, y se requieren k ejemplos de *FSP*, el sistema selecciona las k soluciones almacenadas en el *Experience Pool* cuya descripción en lenguaje natural sea más similar a la del nuevo problema. Esta similitud se determina mediante el cálculo de *embeddings* de lenguaje natural de las descripciones de los problemas, y el uso de la medida de *cosine similarity* (similitud del coseno) para obtener las más cercanas.

En el capítulo de Implementación se analiza a detalle el proceso que se lleva a cabo.

Capítulo 3

Implementación de la propuesta

3.1. Estructura del proyecto

El sistema propuesto fue implementado en **Python 3.12**, haciendo uso extensivo de bibliotecas modernas para procesamiento de lenguaje natural, planificación automática y manejo de datos. La arquitectura del código siguió un diseño modular y extensible, con componentes claramente separados por responsabilidad: agentes modeladores y planificadores, interacción con *LLMs*, evaluación de modelos, entrenamiento experiencial, y recuperación semántica con *RAG*. Cada módulo contó con su propia estructura de datos y utilidades específicas, lo que facilita tanto la reproducibilidad como la incorporación de mejoras o nuevas estrategias de modelado.

Se integraron herramientas como `sentence-transformers` para *embeddings* de lenguaje natural, `Fast Downward` para planificación clásica, y `VAL` para validación de planes. Las consultas a modelos de lenguaje se gestionaron mediante la *API* de *Fireworks AI*, y se realizó un seguimiento detallado del consumo de *tokens* para evaluar la eficiencia del sistema. La base de datos del *benchmark Planetarium* se almacenó en formato *SQLite*, permitiendo un acceso eficiente y estructurado.

El conjunto del sistema puede ejecutarse desde un único punto de entrada (`main.py`), facilitando la ejecución de experimentos completos de principio a fin. La estructura del proyecto implementado se presenta en el Anexo correspondiente. A continuación se describen a detalle algunos de los componentes relevantes implementados.

3.2. Recursos de dominio

Para cada uno de los dominios considerados en este trabajo, se construyó un conjunto de recursos que permitió automatizar y estandarizar los procesos de razonamiento, extracción y generación. Estos recursos fueron esenciales tanto para la etapa de entrenamiento y ajuste de los agentes modeladores, como para su evaluación. Los elementos incluidos en esta base de recursos fueron:

- Un problema de ejemplo del dominio, utilizado para *FSP*. Este consiste en:
 - Una descripción en lenguaje natural del problema (`fsp_ex_nl.txt`).
 - Un razonamiento estructurado sobre la modelación del problema (objetos, estado inicial, estado objetivo) (`fsp_ex_reasoning.txt`).
 - El conjunto estructurado de objetos relevantes extraídos del problema, tipados si el dominio lo requiere (`fsp_ex_objects.json`).

- El modelo *PDDL* del problema (`fsp_ex_pddl.pddl`).
- Un plan que resuelve el problema, en formato *PDDL plan*, una secuencia de acciones concretas del dominio (`fsp_ex_plan.pddl`).
- Una descripción corta del dominio en lenguaje natural (`domain_description.txt`).
- Una descripción en lenguaje natural de las acciones disponibles en el dominio, incluyendo su nombre, parámetros, precondiciones y efectos (`actions_description.txt`).
- La descripción de la sintaxis esperada de la salida del planificador, es decir, el formato del plan esperado (`planner_output_syntax.txt`).
- El modelo *PDDL* del dominio completo (`domain.pddl`).
- La lista de predicados definidos en el dominio, junto con su aridad. En el caso de dominios tipados, también se indicaron los tipos de cada uno de los argumentos.
- La jerarquía de tipos del dominio, en caso de que este cuente con tipado.

Para facilitar el acceso y reutilización de esta información en diferentes componentes del sistema, se implementaron métodos utilitarios simples en *Python* que permitieron importar y consultar dinámicamente estos recursos a partir del nombre del dominio. Esto permitió no solo una mayor modularidad en el diseño del agente modelador, sino también una integración directa con los módulos de razonamiento, extracción y validación semántica.

3.3. Modelos de Lenguaje

Todos los agentes modeladores desarrollados en este trabajo utilizan *LLMs* a través de la interfaz de programación de aplicaciones (*API*) provista por la plataforma *Fireworks AI*. Esta plataforma permite acceder de manera flexible y configurable a modelos de última generación, incluyendo diversas funcionalidades clave que se detallan a continuación:

- **Selección de modelos:** La *API* ofrece acceso a distintos modelos del estado del arte. Para este trabajo se utilizó el modelo **Llama 4 Maverick Instruct (Basic)** ([FireworksAI, 2025](#)). Esta decisión se debe a la buena relación entre su alta capacidad y su costo relativamente bajo en comparación con otras alternativas de similar potencia.
- **Construcción del *prompt*:** Se emplea el esquema de diálogo estructurado típico en los *LLMs* modernos, donde se definen dos roles: *system* y *user*. El *system prompt* se utiliza para establecer el comportamiento general del modelo, mientras que el *user prompt* contiene la tarea específica a ejecutar. Esta separación permitió modularidad y claridad en las interacciones, facilitando la adaptación del agente a diferentes subtareas de modelado. En el Anexo, sin embargo, se presentan los *prompts* sin separación de roles para facilitar la lectura.
- **Parámetro de temperatura:** La temperatura es un parámetro que controla la aleatoriedad de la salida del modelo. Valores cercanos a 0 hacen que el modelo tienda a elegir las salidas más probables (mayor determinismo), mientras que valores mayores introducen más variabilidad y creatividad. En todos los experimentos de este trabajo se fijó la temperatura en 0, con el objetivo de aumentar la

garantía de resultados deterministas y reproducibles¹. Esta decisión fue especialmente importante dado que las tareas implicaban la generación de código o estructuras altamente sensibles a errores sintácticos o semánticos, donde la consistencia era prioritaria.

- **Activación del modo Grammar:** La *API* permite activar un modo de salida restringido por gramática, denominado *Grammar Mode*, basado en la integración con *llama.cpp*. Esta fue la funcionalidad determinante para el uso de esta plataforma, al facilitar la ejecución del enfoque de *GCD*, permitiendo de forma nativa restringir la salida del modelo a una gramática *GBNF*.

Con el objetivo de documentar rigurosamente los experimentos realizados y facilitar su análisis posterior (incluyendo auditoría, depuración y replicación), cada consulta a los *LLMs* fue registrada en un archivo individual en formato *JSON*. Cada uno de estos archivos contiene:

- El contenido completo del *system prompt* y el *user prompt*.
- La identificación del *LLM* utilizado (mediante la *URL* provista por *Fireworks AI*).
- La gramática utilizada, en caso de haber estado activo el módulo de *GCD*.
- La fecha y hora del registro.
- El tiempo total de respuesta del modelo.
- El contenido generado como respuesta por el modelo.
- La cantidad total de *tokens* consumidos, discriminando entre los usados en el *prompt* y los usados en el *completado*.

Dado que todas las interacciones con los modelos se realizaron a través de una *API* remota, se consideraron distintos mecanismos de recuperación ante posibles fallos, como desconexiones, errores de red, o restricciones por límite de *tokens*. Para ello se mantuvo un conjunto de claves de autenticación (*API Keys*) válidas de *Fireworks AI*, y ante cualquier fallo se definió una política de reintentos: se reintenta la consulta hasta un máximo de tres veces por cada clave antes de pasar a la siguiente, repitiendo el proceso hasta que la petición sea exitosa o se agoten todas las claves disponibles.

Este sistema robusto de consulta y registro fue fundamental para asegurar la trazabilidad completa de las decisiones tomadas por los agentes modeladores en cada instancia de generación.

¹Aunque en teoría $temperatura = 0$ corresponde a una decodificación puramente *greedy* (seleccionando siempre el *token* de mayor probabilidad), en la práctica los modelos de *LLM* utilizados vía *APIs* (como *Fireworks AI*) pueden seguir produciendo salidas distintas en ejecuciones repetidas. Esto se debe a varios factores: (1) cálculos de punto flotante en *hardware paralelo (GPUs)* no asociativos, cuyos errores de redondeo pueden alterar la elección del *token* más probable; (2) arquitecturas de mezcla de expertos (*Mixture of Experts - MoE*), donde la asignación de *expertos* puede variar según el estado del sistema y el *batching*; (3) implementaciones reales de *softmax*, *top-k* y *top-p* que incluyen operaciones no deterministas o empantan *logits* y eligen arbitrariamente entre ellos ([Toman, 2023, 25 de agosto](#); [Weinmeister, 2024, 14 de marzo](#)). Por ello, fijar la temperatura a cero reduce la aleatoriedad, pero *no asegura* salidas idénticas. Para una *reproducibilidad* real, es necesario considerar entornos controlados localmente (mismo *hardware*, ejecución secuencial, semillas fijas) o emplear múltiples ejecuciones para estimar la variabilidad.

3.4. Procesamiento y guardado de las operaciones sobre la base de *Insights*

La respuesta del agente de *insights* consiste en una lista de operaciones en el formato estructurado definido por el *prompt*. Para su procesamiento, se implementó una función de análisis léxico y sintáctico que realiza un *parsing* de la salida textual, para extraer las operaciones individuales y sus componentes (tipo, índice, contenido).

Dicha función valida cada operación en cuanto a su estructura y semántica, descartando aquellas malformadas o inválidas según las reglas impuestas. Posteriormente, se aplican las operaciones válidas al conjunto correspondiente de *insights*, actualizando su contenido y relevancia de acuerdo con las instrucciones proporcionadas por el agente.

Además, en cada paso del proceso se almacena información detallada que incluye: (1) la respuesta textual del agente, (2) las operaciones extraídas tras el *parsing*, y (3) las operaciones válidas efectivamente aplicadas. Esta información es registrada para permitir trazabilidad, análisis posterior, revisión manual y depuración durante la experimentación.

3.5. RAG

Los *embeddings* fueron generados con el modelo *all-mnlp-base-v2*, una red de *transformers* disponible en la biblioteca *sentence-transformers* de *Hugging Face* ([HuggingFace, 2025](#)). Aunque este modelo puede ejecutarse localmente, hacerlo de forma reiterada por cada nueva consulta implicaría un elevado costo computacional, tanto en tiempo como en recursos. Asimismo, utilizar la *API* oficial de *Hugging Face* implicaría incurrir en costos económicos asociados al uso de *tokens*.

Como alternativa eficiente, se optó por precalcular los *embeddings* correspondientes a los subconjuntos seleccionados del *dataset Planetarium* —tanto de entrenamiento como de evaluación— utilizando un entorno de ejecución en *Google Colab*, aprovechando su capacidad de cómputo. Los *embeddings* generados fueron almacenados en un archivo con formato *.npz* (archivo comprimido de múltiples arreglos *NumPy*), lo cual permitió descargarlos y reutilizarlos localmente sin necesidad de recálculo o gasto adicional de *tokens*.

Adicionalmente, se desarrolló una implementación para cálculo local de *embeddings* mediante la *API* oficial, orientada a pruebas puntuales o procesamiento de nuevos problemas no contenidos en el conjunto precalculado.

3.5.1. Retriever

Se implementó una clase denominada *Retriever*, encargada de realizar la recuperación eficiente de ejemplos relevantes de *FSP* desde el conjunto de soluciones exitosas. Esta clase carga los *embeddings* pre-calculados de los problemas almacenados en el *Experience Pool*, así como el *embedding* del nuevo problema planteado.

Formalmente, sea $E = \{e_1, e_2, \dots, e_n\}$ el conjunto de *embeddings* vectoriales correspondientes a las descripciones de los n problemas resueltos exitosamente, y sea e_{query} el *embedding* del nuevo problema. La clase *Retriever* calcula la similitud del coseno entre e_{query} y cada e_i del conjunto E , definida como:

$$\text{sim}(e_{\text{query}}, e_i) = \frac{e_{\text{query}} \cdot e_i}{\|e_{\text{query}}\| \|e_i\|}$$

A continuación, selecciona los k vectores e_i con mayor valor de similitud, y retorna las correspondientes soluciones correctas asociadas como ejemplos de *FSP*. Esta operación garantiza que el contexto

proporcionado al agente modelador en la etapa de evaluación esté compuesto por ejemplos relevantes, semánticamente cercanos al problema actual.

3.6. Manipulación de modelos *PDDL* de problemas

Para implementar las evaluaciones parciales propuestas en el capítulo anterior se desarrollaron diversas funciones utilitarias, con base en la función `planetarium.evaluate` que brinda la biblioteca de *Python* del *benchmark*:

- `get_pddl_substr`: localiza el *substring* correspondiente al modelo *PDDL* dentro de un modelo. Para ello, se busca el primer paréntesis abierto desde el inicio del texto, y a partir de ese punto se recorre la cadena hasta encontrar una secuencia válida de paréntesis balanceados, la cual es retornada inmediatamente. Este algoritmo, aunque sencillo, demostró ser eficaz para delimitar la sección relevante de modelo en la práctica, dado que los modelos *PDDL* son estructurados con paréntesis anidados correctamente.
- `extract_typed_objects`: extrae los objetos y sus tipos desde la sección `:objects` de un modelo de problema *PDDL*. En caso de que algún objeto no tenga declarado explícitamente un tipo, se asume que pertenece al tipo primordial `object`. La función retorna una lista agrupada adecuadamente, incluso si los objetos de un mismo tipo están declarados de forma no contigua.
- `split_pddl_problem_sections`: divide un modelo *PDDL* sintácticamente válido en cinco secciones: un prefijo, los predicados del estado inicial, una sección intermedia, los predicados del estado objetivo, y un sufijo. El prefijo comprende todo hasta la primera aparición de la subcadena `:init`, que marca el inicio de los predicados del estado inicial. La sección intermedia abarca desde el cierre de `:init` hasta el comienzo de la sección del estado objetivo, incluyendo la cadena `:goal` y, en algunos casos, `and()`. El sufijo comienza justo después de terminar los predicados del estado objetivo. Esta división facilitó el análisis independiente de cada sección del problema para la evaluación parcial.

3.7. Correcciones del *dataset* de *Planetarium*

Durante la etapa de pruebas previas a la ejecución de los experimentos, se identificaron diversos errores en las descripciones generadas de los problemas incluidos en el *dataset Planetarium*. Algunos de estos errores se debieron al hecho de que la versión publicada en la plataforma *Hugging Face*, enlazada desde el repositorio oficial del proyecto, no se encontraba actualizada. Dicha versión aún no incluía los cambios realizados en el último *commit* del repositorio, en el cual los autores corregían ciertos errores en la generación de las instancias.

Para subsanar esto, se procedió a clonar el repositorio oficial, instalar sus dependencias, y ejecutar los pasos descritos en la documentación técnica con el objetivo de reproducir localmente el *dataset* actualizado. A pesar de esto, se observó que algunos errores persistían incluso en la versión más reciente disponible, lo que indicaba que dichos problemas no habían sido identificados ni resueltos en los cambios previos.

Tras un análisis detallado del código fuente encargado de la generación de las descripciones, se determinaron las causas exactas de varios errores. A continuación, se presentan los errores detectados, junto con las correcciones propuestas. Se recomienda a los autores del artículo *Planetarium* y de su implementación oficial considerar la inclusión de estos ajustes en futuras versiones del proyecto.

En el dominio *Blocksworld*, se corrigieron errores relacionados con la función `abstract_description` de la clase `BlocksworldDatasetGenerator`. Entre estos, la función `equal_towers` reportaba incorrectamente

la cantidad de torres, debido a un mal uso de la función `len()`. Además, la descripción para la tarea `invert` contenía una redacción confusa que dificultaba la comprensión de la meta.

En *Gripper*, los problemas se encontraron principalmente en las funciones `drop_and_pickup`, `holding` y `abstract_description`. El generador no garantizaba la existencia de una sala vacía diferente a la inicial ni que al menos un *gripper* sostuviera una pelota, además de no restringir la ubicación de las pelotas a la primera sala. También se mejoró la descripción de la tarea `juggle`, aclarando la dirección del movimiento de las pelotas y la asignación inicial a los *grippers*.

Finalmente, en *Floor-Tile*, se hicieron correcciones a la generación del tablero y la descripción de las tareas. La función encargada del predicado `checkerboard` fue modificada para evitar desalineaciones en los colores del tablero. En la función `abstract_description.get_robot_ring_string`, se corrigieron errores de lógica y redacción relacionados con la ubicación inicial del robot en anillos concéntricos. Además, la descripción abstracta de la tarea `paint_x` fue mejorada para especificar con mayor precisión los objetivos visuales, especialmente en relación al número de colores requeridos.

En el Anexo se comparte la implementación y descripción detallada de los cambios realizados.

Capítulo 4

Evaluación

Este capítulo describe el diseño experimental y los resultados de la evaluación del sistema propuesto, centrada en la validación empírica de cuatro hipótesis relacionadas con la efectividad de distintas técnicas aplicadas al modelado automático de problemas de planificación. Cada hipótesis busca demostrar la contribución individual de uno o varios componentes específicos al desempeño global del agente modelador basado en *LLMs*, ya sea en términos de correctitud sintáctica, solubilidad o fidelidad semántica del modelo generado:

- H1.** La división del proceso de modelado en fases estructuradas —extracción de objetos, razonamiento, especificación del estado inicial y metas, y generación del archivo *PDDL*— mejora la correctitud de los modelos generados, al permitir un razonamiento más controlado, modular y verificable.
- H2.** La aplicación de *GCD* permite una generación más confiable del código *PDDL*, reduciendo significativamente o eliminando por completo la aparición de errores de sintaxis.
- H3.** La introducción de reflexión sobre errores y una mínima retroalimentación humana o automática permite al agente corregir patrones de falla recurrentes, contribuyendo al aumento de la solubilidad y la correctitud del *PDDL* generado.
- H4.** La incorporación de *RAG* para la selección de ejemplos relevantes y la extracción de *insights* a partir de soluciones previas (tanto correctas como erróneas), representa una vía prometedora para mejorar la capacidad de los agentes basados en *LLMs* para modelar tareas de planificación, al permitirles adaptarse a la semántica de nuevas tareas y fortalecer su conocimiento sobre el dominio específico y la modelación de problemas.

La evaluación se desarrolla en dos fases principales. La primera corresponde a una evaluación comparativa estática, sobre un subconjunto común del *benchmark Planetarium*, en la cual se analiza el comportamiento de múltiples variantes del agente modelador. En esta fase se incluyen, en primer lugar, los agentes planificadores básicos originales introducidos por *LLM+P*, a saber, *LLM-as-P* (modelo *Zero-Shot*) y *LLM-as-P⁺* (modelo *One-Shot* con un ejemplo fijo), que no generan código *PDDL* sino directamente planes a partir de descripciones en lenguaje natural. Estos agentes actúan como *baselines* para evaluar la dificultad del *benchmark* y las ventajas del modelado explícito. En segundo lugar, se evalúan también los agentes modeladores de planificación originales de *LLM+P*, esto es, *LLM+P* y *LLM+P⁺*, que sí generan archivos de problema en *PDDL* y constituyen un punto de partida relevante para contrastar la efectividad de los módulos propuestos en esta tesis. Los agentes modeladores originales de *LLM+P* se reimplementan para

mejorar su *prompt*, adaptarlo mejor a la evaluación en *Planetarium*, y estructurarlo, para que sirva como *baseline* y como el estándar o base sobre la cual se añaden las mejoras propuestas.

A continuación, se consideran variantes del agente propuesto en esta investigación, construidas de manera incremental mediante la incorporación progresiva de módulos diseñados para mejorar la generación de modelos. Estas variantes permiten analizar empíricamente la influencia directa de cada técnica propuesta. Se parte de una reimplementación básica del agente de *LLM+P*, sobre la cual se introducen primero mecanismos de razonamiento estructurado y extracción de objetos —ambos dirigidos a modular el proceso de modelado, conforme a la hipótesis **H1**—. Luego se incorpora *Grammar-Constrained Decoding (GCD)* para garantizar la validez sintáctica de las salidas, con énfasis especial en su variante condicionada por conocimiento del dominio (*DAPS GCD*), en línea con la hipótesis **H2**. Además, se aplica *Few-Shot Prompting*, con un ejemplo fijo por dominio, construido manualmente. Todas estas variantes son evaluadas con un *LLM* de gran tamaño y alta capacidad: *Llama 4 Maverick Instruct (Basic)* (FireworksAI, 2025).

Complementando esta evaluación comparativa, la segunda fase del experimento aborda el proceso de entrenamiento del agente experiencial, es decir, aquel capaz de realizar reintentos informados y con reflexión autocrítica, y aprovechar información derivada de su experiencia acumulada durante el proceso de entrenamiento previo. Esta fase incluye la acumulación de experiencias exitosas y fallidas (**H4**), con capacidad de reintentos guiados por *feedback* y autorreflexión (**H3**), y la extracción automática de *insights* mediante un proceso iterativo controlado por un *LLM* especializado (**H4**). Esta dinámica se orienta no solo a mejorar el rendimiento del agente en tareas futuras, sino también a evaluar la calidad y utilidad de los conocimientos derivados de la experiencia.

Finalmente, se propone la evaluación sobre el mismo conjunto de evaluación anterior, los agentes modeladores experienciales. Estos agentes integran mecanismos de retroalimentación automática, y reflexión autocrítica sobre errores pasados, asistidos por *LLMs*, o una forma mínima de retroalimentación humana, en el marco de la hipótesis **H3**. Se realiza la evaluación de los mecanismos de reintento guiado sobre los problemas modelados incorrectamente en los intentos únicos del mejor agente de la fase anterior.

Se exploran también los mecanismos de *FSP* potenciados con *RAG* para seleccionar ejemplos semánticamente similares del *corpus* de soluciones válidas, como se sugiere en **H4**. En relación con esta misma hipótesis, se prueba la integración de un conjunto de *insights* previamente extraídos del historial de errores y aciertos.

La estructura de este capítulo refleja esta metodología. Primero se describe en detalle el diseño de los experimentos, estableciendo la relación de cada fase con las hipótesis planteadas. Luego se presentan los resultados obtenidos, acompañados de métricas cuantitativas y visualizaciones comparativas. A continuación, se discute críticamente la influencia de cada componente en los resultados, y se identifican observaciones relevantes sobre las interacciones entre técnicas. Finalmente, se exponen las limitaciones de la evaluación, se resumen los resultados obtenidos y se analiza el grado de cumplimiento de cada hipótesis dentro del marco experimental planteado.

4.1. Selección de subconjuntos de *Planetarium*

Para garantizar evaluaciones rigurosas y comparables, el proceso de selección de datos se organizó en tres fases complementarias: muestreo de pares (`init`, `goal`), filtrado estratificado sobre la base SQLite, y generación de subconjuntos finales de evaluación y entrenamiento.

En la primera fase, se definió estáticamente un conjunto amplio de todas las parejas posibles de configuraciones o subtipos de tareas de los estados inicial (`init`) y objetivo (`goal`) para cada dominio (*Blocksworld*, *Gripper* y *Floor-Tile*). A partir de estas parejas, se seleccionaron aleatoriamente subconjuntos de casos de entrenamiento y prueba, empleando una semilla fija para asegurar reproducibilidad. Este muestreo inicial

permite controlar la cobertura de distintos tipos de tareas.

En la segunda fase, cada pareja elegida se usó para filtrar directamente la tabla `problems` de la base SQLite, construyendo máscaras que combinan rangos de número de objetos, cantidad de proposiciones en `:init` y `:goal`, y niveles de abstracción de los estados (abstracto o explícito). Para cada celda de esta cuadrícula multivariada, se extrajeron hasta N ejemplos al azar, posibilitando un muestreo estratificado sobre toda la diversidad del dataset.

La tercera fase concluyó la construcción de los subconjuntos:

- Con los casos de prueba iniciales filtrados, se depuraron duplicados y se agruparon por dominio. Se extrajeron finalmente 16 instancias de *Blocksworld*¹, 27 de *Gripper* y 27 de *Floor-Tile*, produciendo el conjunto de evaluación con un total de 70 instancias. Durante este muestreo, se llevaron estadísticas de distribución en función de los *layout pairs* (combinaciones de `init` y `goal`), los distintos patrones de abstracción, el tamaño total de proposiciones y el número de objetos, asegurando cobertura equilibrada.
- Para el conjunto de entrenamiento se fusionaron los subconjuntos originales de entrenamiento y prueba (excluyendo los casos reservados para evaluación y un tercio aleatorio de *layout pairs*). A partir de los ejemplos restantes, se seleccionaron aleatoriamente 50 casos de *Blocksworld* y 50 de *Gripper*, para un total de 100 instancias, manteniendo igualmente la estratificación por dominio, abstracción, tamaño y número de objetos.

Esta metodología asegura que ambos conjuntos cubren de manera representativa la variabilidad de escenarios del *benchmark Planetarium*, facilitando una evaluación precisa de la capacidad de generalización y robustez de los agentes modeladores propuestos.

4.2. Proceso de Evaluación de los Agentes

Con el objetivo de obtener resultados empíricos fiables y trazables que permitan evaluar cuantitativamente las hipótesis planteadas en esta tesis, se diseñó una rutina de evaluación automatizada y estructurada, capaz de ejecutar de forma sistemática todos los agentes definidos sobre un conjunto común de tareas. Esta rutina permite medir, de manera uniforme y controlada, el impacto individual de cada componente propuesto, así como las interacciones entre ellos. La estandarización del proceso garantiza la comparabilidad entre variantes y asegura que cualquier diferencia observada en el rendimiento pueda atribuirse directamente a la activación o desactivación de un módulo específico, aspecto crucial para la validación de todas las hipótesis, de **H1** a **H4**.

La evaluación de los agentes se organiza mediante una rutina central que recorre sistemáticamente cada combinación de agente y problema seleccionados, registra el progreso y almacena los resultados de forma estructurada. El procedimiento se inicia invocando la función `run_evaluations(False, eval_cases)`, que solicita confirmación si se va a reiniciar la evaluación o, en su defecto, retoma desde el último punto guardado en un archivo de progreso.

En primer lugar, se define la lista de agentes activos, que incluye planificadores puros (`l1m_planner`, con o sin ejemplo *FSP*) y los agentes modeladores originales y mejorados (`orig_l1m_plus_p` y variantes), ambos de *LLM+P*, y los agentes modeladores con extensiones propuestas (`r`, `r_o`, `r_o_gcd`, etc.). A continuación se inicializan dos estructuras vacías, una para recopilar resultados por agente y otra para acumular estadísticas por dominio. Si se reanuda una ejecución previa, se carga el estado (índice de agente y de ejemplo) desde

¹Se obtuvieron menos instancias en este dominio que en el resto debido a que el *split* de evaluación (*test*) de *Planetarium* solo contiene dos parejas de configuraciones de estados para *Blocksworld*: (`swap`, `swap`) e (`invert`, `invert`).

un *JSON* de progreso; de lo contrario, se crea este fichero y se arranca desde el primer agente y el primer caso de evaluación.

Para cada agente en la lista, se instancia dinámicamente el objeto correspondiente: si su nombre coincide con uno de los planificadores, se utiliza un constructor específico de planificador; si pertenece al grupo original *LLM+P*, se invoca el agente base; en caso contrario, se crea un agente modelador avanzado que incluye, según su descripción, módulos como razonamiento, extracción de objetos, *GCD*, etc. Se crea luego un directorio de resultado para almacenar los datos de cada problema.

Dentro de cada agente, el bucle recorre todos los casos o tareas de evaluación. Para cada problema, se guarda la descripción en lenguaje natural y el modelo *PDDL* de referencia en ficheros bajo el directorio asignado al caso. Luego se asigna el problema a resolver al agente mediante el método `agent.set_task(idx, domain, n1)`, y se mide el tiempo de generación de la predicción invocando el método `agent.solve_task()`. El contador de *tokens* utilizados (entrada y salida) se extrae de la respuesta y se imprime para diagnóstico.

Si el agente es un planificador (`l1m_planner` con o sin *FSP*), la salida es un plan *PDDL*, que se guarda y valida con la herramienta externa *VAL*. En caso de agentes modeladores, la salida es un modelo *PDDL* del problema de planificación, al que se aplica un evaluador interno (`eval_trial`) que determina tres métricas: *parseability* (validez sintáctica), *solvability* (capacidad de un planificador simbólico para resolverlo) y *correctness* (equivalencia semántica con la referencia). El resultado de esta validación se incorpora al registro de resultados.

Si el modelo *PDDL* generado es soluble, se invoca adicionalmente un planificador clásico (*Fast Downward*) sobre el problema predicho para comparar el plan obtenido con el plan de referencia; este paso permite evaluar no solo la generación sino también la funcionalidad práctica del modelo. Asimismo, para todos los casos se genera un plan de referencia a partir del *ground truth*, de modo que cada ejecución deja constancia del plan esperado y del plan predicho.

Tras procesar todos los problemas para un agente, se almacenan las estructuras acumuladas en ficheros *JSON* con *timestamp*, uno organizado por agente y otro por dominio. Finalmente, se marca el progreso como completado y se informa al usuario de la finalización de la evaluación.

Este diseño de evaluación proporcionó una base empírica sólida para el análisis de las hipótesis planteadas, al aplicar sistemáticamente un mismo *pipeline* a todos los agentes y tareas, lo que garantizó comparabilidad y permitió evaluar el efecto de cada componente. La ejecución controlada, con gestión de progreso y semillas fijas, asegura la reproducibilidad de los resultados, mientras que la validación exhaustiva, tanto sintáctica como semántica, preserva la integridad del proceso. La organización estructurada de agentes por variantes modulares, el uso de métricas objetivas estandarizadas y la cobertura de un conjunto común de tareas permiten trazar con cierto grado de precisión el impacto individual y combinado de cada técnica propuesta sobre el rendimiento del agente.

4.3. Fase de entrenamiento del agente experiencial

La fase de entrenamiento, subdividida en acumulación de experiencias y extracción de *insights*, fue abordada en detalle en el capítulo de Propuesta de solución. Por ello, en esta sección no se reiteran los aspectos técnicos ni los algoritmos específicos, sino que se enfatiza su rol y contribución en la evaluación empírica de las hipótesis planteadas.

El proceso de acumulación de experiencias consistió en la ejecución iterativa de tareas, donde el agente experiencial almacenó tanto soluciones correctas como incorrectas, junto con reflexiones automáticas y posibilitando retroalimentación humana mínima (aunque esta no se utilizó en los experimentos realizados).

La extracción automática de *insights* a partir del historial de experiencias acumuladas constituyó el componente exploratorio para la hipótesis **H4**, la cual sostiene que la utilización de conocimiento derivado

de errores y aciertos previos tiene el potencial de contribuir a una mejora semántica y robustez en la generación de modelos *PDDL*.

Finalmente, la integración de mecanismos de reflexión asistida por *LLMs*, durante el entrenamiento y los reintentos de evaluación, proporcionó una base para analizar la hipótesis **H3**, que postula que este proceso facilita la corrección progresiva y la mejora continua del agente experiencial.

En conjunto, la fase de entrenamiento estableció un entorno controlado para la generación y acumulación de conocimiento, cuya calidad y utilidad fueron evaluadas posteriormente, contribuyendo de manera directa a validar las propuestas en esta investigación.

4.4. Comparación con *baselines* de planificación directa

Los agentes planificadores básicos, reproducidos a partir del enfoque *LLM+P*, generan directamente planes a partir de descripciones en lenguaje natural. Las implementaciones utilizadas se denominaron `llm_planner` y `llm_planner_fsp`, y sirven como *baselines* para evaluar tanto la dificultad del *benchmark* como las ventajas del modelado explícito. Sin embargo, la evaluación automática de estos agentes dentro del *benchmark* Planetarium presenta limitaciones inherentes a la naturaleza de sus salidas y de las herramientas disponibles.

La herramienta *VAL* permite verificar si un plan constituye una solución válida para un problema dado. Para ello, recibe como entrada los archivos *PDDL* del dominio, del problema y del plan generado, y determina si el plan resuelve correctamente el problema descrito en el modelo *PDDL*. No obstante, todos los modelos *PDDL* de problemas proporcionados como *ground truth* en el conjunto de datos Planetarium son definiciones explícitas de los estados iniciales y metas, incluso cuando la descripción en lenguaje natural asociada es abstracta o ambigua. Esto significa que, aunque una instancia permita múltiples interpretaciones válidas a partir de su descripción, el *ground truth* representa solo una de ellas.

Dado este escenario, un plan generado por un agente planificador podría ser correcto bajo una interpretación razonable del problema, pero no coincidir con la interpretación específica codificada en el *ground truth*. Como consecuencia, este plan sería evaluado como incorrecto por *VAL*, a pesar de su validez potencial en otro contexto. Esto limita la evaluación automática de estos agentes únicamente a aquellos problemas cuya descripción en lenguaje natural define de forma explícita tanto el estado inicial como la meta.

De los 70 problemas seleccionados para el subconjunto de evaluación, solo 28 cumplen con este criterio de explicitud: 8 pertenecientes al dominio *Blocksworld*, 12 al dominio *Gripper* y 8 al dominio *Floor-Tile*. Los agentes planificadores fueron evaluados exclusivamente en este subconjunto, y la única métrica considerada fue la validez de los planes generados. Por su parte, los agentes modeladores, que también incluyen estas 28 tareas en su evaluación, deben generar modelos *PDDL* correctos para cada problema. Luego, los planes son obtenidos mediante el planificador *Fast Downward*, y su validez depende directamente de la corrección de dichos modelos.

La figura 4.1 resume los resultados obtenidos:

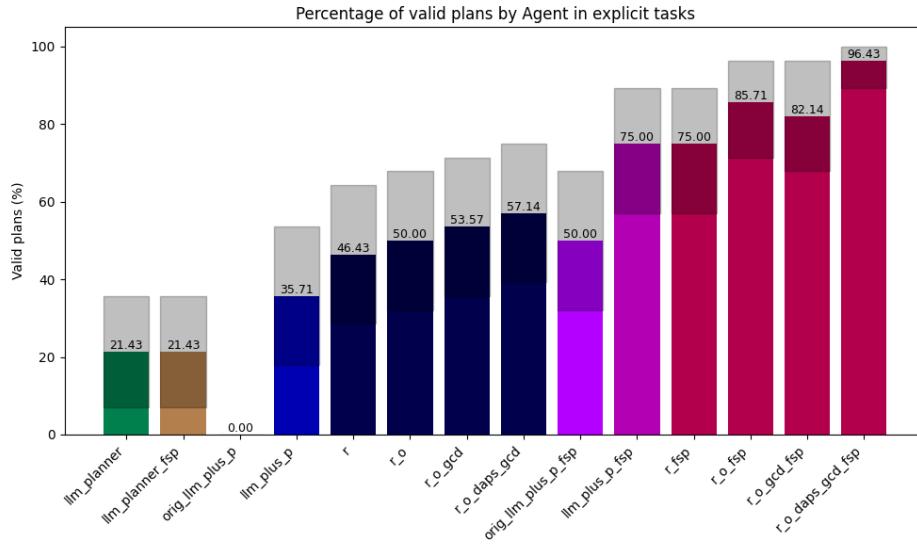


Figura 4.1: Porcentaje de planes válidos por agente en tareas con descripción completamente explícita.

A la izquierda se muestran los dos agentes planificadores reproducidos. A la derecha se presentan los agentes modeladores, organizados en dos grupos: los seis primeros no utilizan *FSP*, mientras que los seis últimos sí lo hacen (indicados mediante barras en tonos más rojizos). Dentro de cada grupo, el orden corresponde a: agente original *LLM+P*, reimplementación estructurada, y variantes con módulos adicionales. En todos los casos se sombra el intervalo de confianza (IC) del 95 %.

Como se aprecia en la figura, ambos agentes planificadores generan planes válidos únicamente en el 21.43 % (IC: [7.14 %, 35.71 %]) de los problemas evaluados, es decir, 6 soluciones correctas. En cambio, todos los agentes modeladores —con excepción de *orig_llm_plus_p*— superan ampliamente este resultado. Además, se observa una mejora progresiva en la métrica con la incorporación de los módulos propuestos. El agente que incluye todos los componentes alcanza el mejor desempeño, con una tasa de validez del 96.43 % (IC: [89.29 %, 100.00 %]), fallando solo en uno de los 28 problemas.

Estos resultados evidencian la complejidad del *benchmark* Planetarium, que representa un reto considerable para agentes planificadores directos, y refuerzan la superioridad del enfoque basado en modelado explícito. Asimismo, proporcionan evidencia empírica del impacto positivo de los módulos de mejora propuestos, como antesala a los resultados globales que se presentan a continuación.

4.5. Resultados generales

En esta sección se presentan los resultados obtenidos por los distintos agentes modeladores evaluados sobre los 70 problemas del conjunto de prueba estratificado del *benchmark* Planetarium, distribuidos en tres dominios: *Blocksworld* (16 problemas), *Gripper* (27 problemas) y *Floor-Tile* (27 problemas).

Se comparan 12 variantes de agentes: los agentes originales de *LLM+P*, sus reimplementaciones estructuradas, y múltiples versiones incrementales mejoradas con los módulos propuestos en esta tesis. Las métricas consideradas incluyen: tasa de modelos con objetos correctamente identificados (*Correct Objects count*), tasa de sintaxis válida (*Parseable*), tasa de solubilidad (*Solvable*) y tasa de problemas completamente correctos (*Correct*). Cada métrica se expresa como porcentaje del total de problemas evaluados. Todos los intervalos de confianza (IC) presentados son del 95 %.

Tabla comparativa de métricas

Tabla 4.1: Resultados comparativos por agente modelador

Agente	COC (%)	Parseable (%)	Solvable (%)	Correct (%)
orig_llm_plus_p	32.86	34.29	0.00	0.00
llm_plus_p	62.86	<u>87.14</u>	54.29	22.86
r	62.86	82.86	62.86	30.00
r_o	<u>95.71</u>	84.29	<u>78.57</u>	<u>42.86</u>
r_o_gcd	98.57	<u>87.14</u>	77.14	44.29
r_o_daps_gcd	92.86	97.14	85.71	41.43
orig_llm_plus_p_fsp	100.00	80.00	62.86	47.14
llm_plus_p_fsp	100.00	87.14	82.86	57.14
r_fsp	100.00	90.00	87.14	72.86
r_o_fsp	100.00	91.43	<u>88.57</u>	<u>75.71</u>
r_o_gcd_fsp	100.00	<u>92.86</u>	<u>88.57</u>	67.14
r_o_daps_gcd_fsp	100.00	100.00	94.29	81.43

En la tabla anterior, los valores en verde indican el mejor resultado por métrica dentro de cada grupo (*Zero-Shot* arriba y *One-Shot* abajo), los subrayados corresponden al segundo mejor valor, y los rojos representan el peor desempeño observado. En la tabla siguiente, de estas evaluaciones se destacan de igual manera los intervalos de confianza al 95%.

Tabla 4.2: Intervalos de confianza al 95% para cada métrica por agente.

Agente	COC IC (%)	Parseable IC (%)	Solvable IC (%)	Correct IC (%)
orig_llm_plus_p	[22.86, 44.29]	[22.86, 45.71]	[0.00, 0.00]	[0.00, 0.00]
llm_plus_p	[51.43, 74.29]	[78.57, 94.29]	[42.86, 65.71]	[12.86, 32.86]
r	[51.43, 74.29]	[74.29, 91.43]	[51.43, 74.29]	[20.00, 41.43]
r_o	[90.00, 100.00]	[75.71, 92.86]	[68.57, 87.14]	[31.43, 54.29]
r_o_gcd	[95.71, 100.00]	[78.57, 94.29]	[67.14, 87.14]	[32.86, 55.71]
r_o_daps_gcd	[85.71, 98.57]	[92.86, 100.00]	[77.14, 92.86]	[30.00, 52.86]
orig_llm_plus_p_fsp	[100.00, 100.00]	[70.00, 88.57]	[51.43, 74.29]	[35.71, 58.57]
llm_plus_p_fsp	[100.00, 100.00]	[78.57, 94.29]	[74.29, 91.43]	[45.71, 68.57]
r_fsp	[100.00, 100.00]	[82.86, 97.14]	[78.57, 94.29]	[61.43, 82.86]
r_o_fsp	[100.00, 100.00]	[84.29, 97.14]	[80.00, 95.71]	[65.71, 85.71]
r_o_gcd_fsp	[100.00, 100.00]	[85.71, 98.57]	[80.00, 95.71]	[55.71, 78.57]
r_o_daps_gcd_fsp	[100.00, 100.00]	[100.00, 100.00]	[88.57, 98.57]	[71.43, 90.00]

Gráficas comparativas

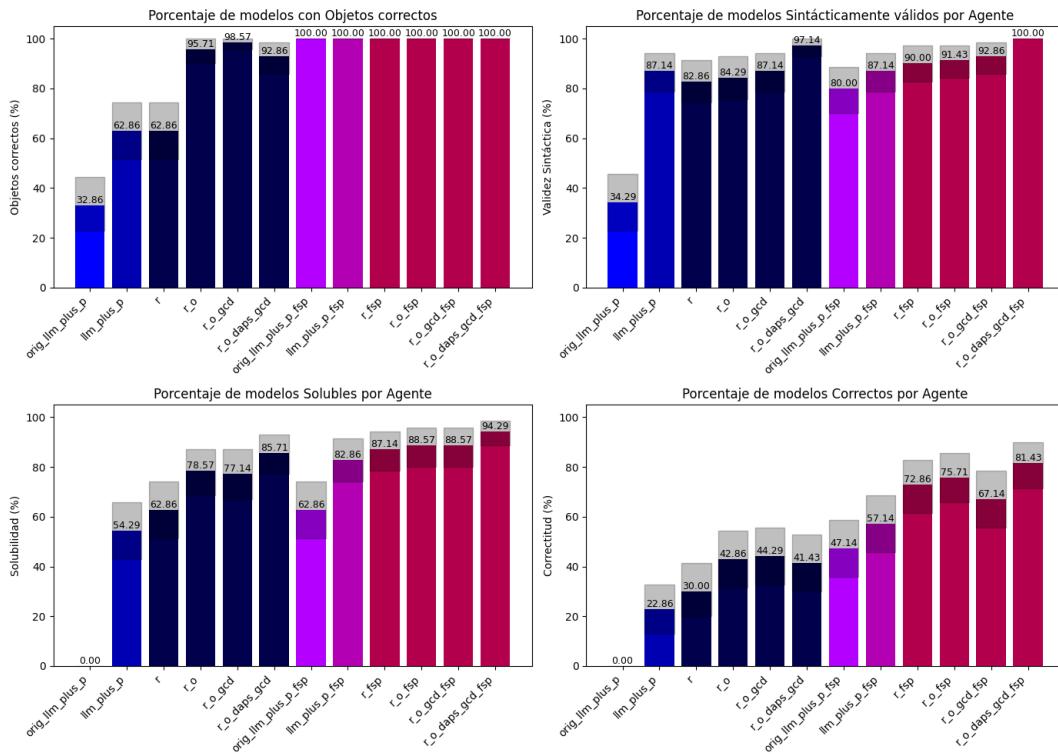


Figura 4.2: Métricas calculadas por agente

Estas gráficas de barras ilustran visualmente la evolución del desempeño de los agentes en cada una de las métricas clave. Los agentes se agrupan en dos bloques: los seis primeros no utilizan *Few-Shot Prompting* (*FSP*), mientras que los seis últimos sí lo hacen (indicados con barras en tonos más rojizos). En cada grupo, el orden de aparición sigue el patrón: agente original de *LLM+P*, reimplementación estructurada, y variantes con módulos adicionales.

4.5.1. Impacto del Razonamiento Estructurado

El razonamiento estructurado fue incorporado como una fase previa y separada a la generación del modelo *PDDL*, con el objetivo de descomponer semánticamente el problema y mejorar la calidad del modelado automático. Esta etapa adicional fue diseñada para inducir al *LLM* a procesar explícitamente los objetos, el estado inicial y el estado objetivo antes de generar la solución final.

El impacto de este componente puede analizarse con claridad a partir de comparaciones directas entre agentes que difieren únicamente en la presencia del razonamiento. Por ejemplo, al comparar *llm_plus_p* con *r*, se observa una mejora en la proporción de modelos solubles, que aumenta de 54.29 % (IC: [42.86 %, 65.71 %]) a 62.86 % (IC: [51.43 %, 74.29 %]), así como en la proporción de problemas *correctos*, que pasa de 22.86 % (IC: [12.86 %, 32.86 %]) a 30.00 % (IC: [20.00 %, 41.43 %]). Aunque los intervalos de confianza se superponen, la diferencia observada indica un efecto positivo consistente del componente, atribuible al cambio en la estrategia cognitiva del agente.

La comparación en presencia de *Few-Shot Prompting* (*FSP*) ofrece aún mayor contraste. Al comparar *llm_plus_p_fsp* con *r_fsp*, se aprecia un aumento en la métrica de solubilidad de 82.86 % (IC: [74.29 %,

91.43 %]) a 87.14 % (IC: [78.57 %, 94.29 %]), y en la de *correctitud completa* de 57.14 % (IC: [45.71 %, 68.57 %]) a 72.86 % (IC: [61.43 %, 82.86 %]). Esta diferencia es significativa tanto en magnitud como en intervalo de confianza, y puede explicarse por el hecho de que los ejemplos de *FSP* fueron cuidadosamente diseñados para mostrar razonamientos y desambiguaciones correctas, lo que actúa como una guía concreta para que el *LLM* aprenda cómo estructurar sus propias inferencias sobre tareas de planificación.

Los beneficios del razonamiento no se extienden a las métricas de *objetos correctos y modelo con sintaxis válida*, pero no tiene un efecto negativo palpable. Por ejemplo, entre *1lm_plus_p* y *r*, la proporción de objetos correctamente identificados se mantiene constante en 62.86 % (IC: [51.43 %, 74.29 %]), lo cual sugiere que este componente no deteriora la precisión de este aspecto, mientras que la validez sintáctica se mantiene elevada (de 87.14 % (IC: [78.57 %, 94.29 %]) a 82.86 % (IC: [74.29 %, 91.43 %])), aunque con una ligera caída en la media. Estas fluctuaciones pueden deberse a alucinaciones, que en ausencia de ejemplos de *FSP* y razonamiento ineficiente pueden acentuarse. Más aún, se observa que la validez sintáctica aumenta ligeramente en el par de agentes *One-Shot*, de 87.14 % (IC: [78.57 %, 94.29 %]) a 90.00 % (IC: [82.86 %, 97.14 %]). Sin embargo, estos resultados no son concluyentes y no comprometen la interpretación general.

Cabe destacar que, debido al diseño incremental del sistema, los resultados de los agentes más avanzados (como *r_o_gcd* o *r_o_daps_gcd*) reflejan interacciones acumulativas entre múltiples componentes. Por tanto, el impacto aislado del razonamiento estructurado debe analizarse principalmente en los pares donde es el único cambio introducido. Aun así, su presencia constante en los agentes de mayor rendimiento (tanto con como sin *FSP*) sugiere que su efecto es no solo positivo, sino posiblemente sinérgico con otros módulos.

Estos resultados empíricos se alinean con hallazgos previos en la literatura, que indican que la inclusión de pasos intermedios de razonamiento mejora las capacidades de los *LLMs* en tareas complejas (J. Wei, X. Wang et al., 2022; S. Yao, Yu et al., 2023). En particular, trabajos como *ReAct* (S. Yao, J. Zhao et al., 2023) han demostrado que combinar razonamiento textual con ejecución no solo mejora la calidad de las salidas, sino que también aumenta la interpretabilidad del proceso. Los resultados obtenidos en esta tesis refuerzan dicha hipótesis, mostrando que una etapa explícita de razonamiento puede traducirse en mejoras cuantificables en desempeño y robustez del agente modelador.

4.5.2. Impacto de la Extracción de Objetos

La Extracción de Objetos se incorporó como una etapa estructurada y especializada con el objetivo de mejorar la precisión semántica y sintáctica del modelo *PDDL* generado. Al separar explícitamente la identificación de objetos relevantes del resto del proceso, se busca garantizar la integridad del conjunto de elementos que participan en el problema de planificación. Esta fase se implementó posterior al razonamiento (cuando está activo), permitiendo heredar y organizar las inferencias previas, y su salida se genera en un formato *JSON* que facilita su uso posterior en componentes como la generación asistida por gramática (*GCD*).

Desde el punto de vista empírico, el impacto de este componente puede observarse de forma clara al comparar agentes que difieren únicamente en la presencia de esta fase. El caso más representativo es el contraste entre *r* y *r_o*, donde se mantiene el razonamiento estructurado y se añade únicamente la extracción de objetos. En esta comparación, la proporción de objetos correctamente identificados aumenta de 62.86 % (IC: [51.43 %, 74.29 %]) a 95.71 % (IC: [90.00 %, 100.00 %]), lo cual representa una mejora sustancial en el componente más directamente afectado por esta fase.

Además, esta mejora en la identificación de objetos se traduce en ganancias indirectas en otras métricas. La validez sintáctica se mantiene estable, con un ligero aumento de 82.86 % (IC: [74.29 %, 91.43 %]) a 84.29 % (IC: [75.71 %, 92.86 %]). La solubilidad mejora sustancialmente, de 62.86 % (IC: [51.43 %, 74.29 %]) a 78.57 % (IC: [68.57 %, 87.14 %]). En correctitud total, *r* alcanza 30.00 % (IC: [20.00 %, 41.43 %]), mientras que *r_o* sube a 42.86 % (IC: [31.43 %, 54.29 %]). Esto sugiere que la correcta ex-

tracción y estructuración de objetos permite al modelo generar instancias más completas y coherentes, impactando positivamente tanto en la validez sintáctica como semántica.

Un patrón similar se repite en los agentes que incorporan *FSP*. Comparando *r_fsp* y *r_o_fsp*, se observa una mejora en solubilidad de 87.14 % (IC: [78.57 %, 94.29 %]) a 88.57 % (IC: [80.00 %, 95.71 %]), así como una ganancia en correctitud total de 72.86 % (IC: [61.43 %, 82.86 %]) a 75.71 % (IC: [65.71 %, 85.71 %]). Aunque estas diferencias son más sutiles, reflejan una consolidación de los beneficios observados en el grupo sin *FSP*.

Estas observaciones refuerzan la utilidad de una fase de extracción dedicada, especialmente en contextos donde la ambigüedad o la omisión de objetos puede deteriorar gravemente la calidad del modelo generado. Al permitir una representación explícita, detallada y consistente de los objetos del dominio, este componente actúa como un estabilizador semántico clave que fortalece las fases posteriores del *pipeline* de modelado.

Finalmente, es importante señalar que este módulo facilita no solo la calidad de los modelos, sino también su inspecciónabilidad y adaptabilidad. Su diseño como fase independiente y estructurada habilita un nivel adicional de trazabilidad que puede ser aprovechado tanto para depuración como para transferencia de conocimiento entre tareas similares, especialmente cuando se combina con componentes como *GCD* o reflexión estructurada. Permite, más aún, la aplicación de la modalidad *DAPS* concebida de *GCD*, al restringir la salida del modelo a una gramática específica al problema a modelar, limitando los argumentos utilizados en las proposiciones a aquellos objetos extraídos en esta fase, con correcto tipado si lo presentan.

4.5.3. Impacto de *Grammar-Constrained Decoding (GCD)* y su variante *Domain-and Problem-Specific (DAPS)*

El módulo de *GCD* fue introducido con el objetivo de garantizar la validez sintáctica de los modelos *PDDL* generados, mediante la incorporación de una gramática explícita sobre la estructura del lenguaje. Como punto de partida, se utilizó una gramática en formato *GBNF* que captura la sintaxis del subconjunto de *PDDL* empleado, incluyendo soporte para dominios tipados y la estructura general de *STRIPS*. Esta gramática permite restringir la generación a secuencias sintácticamente válidas, pero no impone restricciones semánticas específicas al dominio o problema en cuestión.

Para superar estas limitaciones, se introdujo una variante más estricta denominada *Domain-and Problem-Specific (DAPS)*, la cual incorpora de forma dinámica los predicados, tipos y objetos específicos del problema en curso. En particular, se especializa la producción de fórmulas atómicas dentro de las secciones `:init` y `:goal`, de modo que solo se aceptan predicados válidos del dominio, con aridad exacta y argumentos seleccionados exclusivamente del conjunto de objetos declarados. Esto impide que el modelo produzca contenido fuera del esquema válido, como razonamientos embebidos o correcciones autorreferenciales dentro de estructuras que requieren literalidad.

Un ejemplo de este comportamiento problemático puede verse en la producción básica de fórmulas atómicas sin restricciones específicas:

```
atomicFormula ::= "(" name (ws object)* ")"
```

Esta definición permite que casi cualquier secuencia de texto sea interpretada como el nombre de un predicado o un argumento, lo que en la práctica genera errores de sintaxis, especialmente cuando el *LLM* introduce razonamientos o comentarios no deseados dentro de las proposiciones. En contraste, en la modalidad *DAPS* se sustituye esta producción por una versión especializada que considera únicamente combinaciones válidas y completas de predicado y argumentos.

En términos empíricos, el impacto de *GCD* puede observarse en el paso de *r_o* a *r_o_gcd*, donde se mantiene el razonamiento y la extracción de objetos, pero se añade el uso de la gramática general. La

validez sintáctica mejora de 84.29 % (IC: [75.71 %, 92.86 %]) a 87.14 % (IC: [78.57 %, 94.29 %]), lo cual muestra una ganancia modesta, posiblemente limitada por las desviaciones mencionadas. Estas también afectan la solubilidad, donde se observa un ligero descenso, de 78.57 % (IC: [68.57 %, 87.14 %]) a 77.14 % (IC: [67.14 %, 87.14 %]), aunque no es estadísticamente significativo considerando la intersección casi total de los intervalos de confianza. La correctitud total sube levemente de 42.86 % (IC: [31.43 %, 54.29 %]) a 44.29 % (IC: [32.86 %, 55.71 %]).

Sin embargo, la modalidad *DAPS* muestra una mejora mucho más marcada. En *r_o_daps_gcd*, se observa que la validez sintáctica alcanza el 97.14 % (IC: [92.86 %, 100.00 %]). También se reporta un aumento significativo en solubilidad, a 85.71 % (IC: [77.14 %, 92.86 %]), aunque esta mejora no se observa en la correctitud, posiblemente por la ausencia en el contexto de buenos ejemplos de razonamiento estructurado. Estos resultados respaldan empíricamente la hipótesis de que una gramática altamente especializada aporta grandes beneficios en la validez sintáctica de los modelos.

Este efecto se ve reforzado cuando se analiza la interacción con *Few-Shot Prompting (FSP)*. En el caso del agente *r_o_fsp*, que ya alcanza valores altos en todas las métricas, se obtiene una validez sintáctica de 91.43 % (IC: [84.29 %, 97.14 %]), una *solubilidad* de 88.57 % (IC: [80.00 %, 95.71 %]) y una *correctitud total* de 75.71 % (IC: [65.71 %, 85.71 %]). Al incorporar la gramática general mediante *GCD* en *r_o_gcd_fsp*, la validez sintáctica mejora levemente a 92.86 % (IC: [85.71 %, 98.57 %]), y la solubilidad se mantiene constante, aunque la correctitud total disminuye a 67.14 % (IC: [55.71 %, 78.57 %]), probablemente debido a las desviaciones observadas por control gramatical y semántico incompleto. Finalmente, al incorporar la modalidad *DAPS* en *r_o_daps_gcd_fsp*, se logra validez sintáctica perfecta (100 %, IC: [100 %, 100 %]), una solubilidad de 94.29 % (IC: [88.57 %, 98.57 %]) y una correctitud total de 81.43 % (IC: [71.43 %, 90.00 %]), lo cual representa el mejor desempeño alcanzado por cualquier agente evaluado en todas las métricas, con una diferencia marcada.

Cabe señalar que, pese a la formalización gramatical, la modalidad *GCD* no garantiza validez sintáctica perfecta por sí sola. Esto se debe a que las gramáticas generales no impiden por completo que el *LLM* inserte contenido erróneo o se desvíe del formato literal requerido, como ocurre al generar proposiciones en `:init` y `:goal`. Solo mediante la especialización del conjunto de producciones a través de *DAPS* se evita completamente este comportamiento.

Estos hallazgos empíricos respaldan las observaciones de trabajos previos como (Loula et al., 2025), que demostraban el potencial de *GCD* para mejorar la coherencia sintáctica en dominios acotados. Sin embargo, la evaluación presentada en esta tesis extiende significativamente dicho análisis al considerar la generación completa de modelos *PDDL* a partir de descripciones en lenguaje natural y una evaluación exhaustiva sobre un subconjunto estratificado de problemas más representativo del *benchmark Planetarium*. Además, se introduce y valida empíricamente un refinamiento fundamental: la especialización semántica de la gramática con base en el dominio y problema, lo cual constituye un avance práctico y conceptual en el uso de *GCD* para modelado automático en planificación.

4.5.4. Un salto cualitativo sobre los *baselines* existentes

Para ilustrar la magnitud del avance logrado por la variante *r_o_daps_gcd_fsp*, es fundamental contrastarla con los *baselines* *orig_llm_plus_p_fsp* y *llm_plus_p_fsp*. En validez sintáctica, los valores de 80.00 % y 87.14 % alcanzados por los respectivos *baselines* son completamente superados: *r_o_daps_gcd_fsp* logra una cobertura total del 100 %, cerrando el 100 % del margen restante en ambos casos.

En solubilidad, la mejora es igualmente destacable. *orig_llm_plus_p_fsp* alcanzaba 62.86 %, quedando un 37.14 % por recorrer; *r_o_daps_gcd_fsp* logra 94.29 %, salvando 31.43 puntos, lo que equivale a un 84.63 % de ese margen. Similarmente, frente al 82.86 % de *llm_plus_p_fsp*, que dejaba 17.14 % por cubrir, *r_o_daps_gcd_fsp* salva 11.43 puntos, un 66.67 % del margen de mejora.

La diferencia es aún más relevante en *correctitud total*, la métrica más exigente e importante. En ella `orig_llm_plus_p_fsp` obtenía 47.14 % (quedando un 52.86 % restante), y `r_o_daps_gcd_fsp` alcanza 81.43 %, cubriendo 34.29 puntos, un 64.86 % del espacio disponible. Frente a `llm_plus_p_fsp` (57.14 %), que dejaba 42.86 % por salvar, la misma variante logra cubrir 24.29 puntos, es decir un 56.67 % de dicho faltante. Dicho de otro modo, más de la mitad de lo que los agentes anteriores no lograban modelar correctamente, es resuelto por la variante que aplica todas las mejoras propuestas.

Estos cálculos muestran que, más allá de la diferencia absoluta en puntos porcentuales, la combinación de *GCD* especializada (*DAPS*) con *FSP* manual, mecanismos de *extracción de objetos* consistentes, y el uso explícito de *razonamiento estructurado*, proporciona una mejora **proporcionalmente muy significativa** sobre lo que faltaba en los *baselines*. Estos resultados demuestran empíricamente el potencial de los componentes propuestos actuando en conjunto para cerrar brechas que los enfoques anteriores apenas lograban mitigar.

4.6. Entrenamiento del agente modelador experiencial

Se llevó a cabo el entrenamiento propuesto del agente modelador experiencial sobre el subconjunto seleccionado de 100 problemas, 50 del dominio *Blocksworld* y 50 del dominio *Gripper*. El agente implementó la configuración `r_o_daps_gcd_fsp`, incorporando los módulos de Razonamiento Estructurado, Extracción de Objetos, *GCD DAPS* y un ejemplo *FSP* definido manualmente, representando así la mejor versión evaluada en iteraciones anteriores. Además, se integró el mecanismo de reintentos, que permitía al agente iterar hasta tres veces sobre cada problema, guiado por *feedback* automático y autorreflexión para mejorar su rendimiento. El agente resultante será denominado `exp` por simplicidad.

La tabla que se presenta a continuación resume los resultados de este proceso de entrenamiento. Cada fila corresponde a una configuración particular de resultados a través de los tres intentos permitidos por problema. Para cada intento, se registran tres las métricas definidas anteriormente: *Parseable* (\mathcal{P}), *Solvable* (\mathcal{S}) y *Correct* (\mathcal{C}). Una **V** representa un resultado positivo, mientras que una **F** indica un resultado negativo para esa métrica en ese intento. El índice numérico en cada métrica indica el intento (ej., \mathcal{P}_1 corresponde a la validez sintáctica en el primer intento). La última columna, “Número de Problemas”, indica la cantidad de problemas que determinaron esa secuencia específica de resultados a lo largo de los intentos del agente.

Tabla 4.3: Resultados del Agente en Problemas de Planificación

Intento 1			Intento 2			Intento 3			Número de Problemas
\mathcal{P}_1	\mathcal{S}_1	\mathcal{C}_1	\mathcal{P}_2	\mathcal{S}_2	\mathcal{C}_2	\mathcal{P}_3	\mathcal{S}_3	\mathcal{C}_3	
V	V	V	-	-	-	-	-	-	87
V	V	F	V	V	V	-	-	-	2
V	V	F	V	V	F	V	V	V	1
V	V	F	V	V	F	V	V	F	2
V	F	F	V	V	V	-	-	-	4
V	F	F	V	F	F	V	V	V	1
V	F	F	V	F	F	V	F	F	1
F	F	F	V	V	V	-	-	-	1
F	F	F	V	F	F	V	V	V	1

Los resultados revelan una alta tasa de éxito en el primer intento, con 87 problemas siendo resueltos correctamente de inmediato. Sin embargo, la inclusión del mecanismo de reintentos demostró ser crucial

para mejorar el rendimiento general. Si bien algunos problemas presentaron dificultades persistentes a lo largo de los tres intentos, la autorreflexión y el *feedback* automático permitieron al agente corregir errores y alcanzar soluciones completamente correctas en intentos posteriores para 10 de los 13 problemas inicialmente fallidos. De los 3 problemas cuyo modelado no fue correcto en ninguno de los intentos, 2 resultaron solubles. Contando únicamente los primeros intentos por cada problema, se alcanzaban tasas de validez sintáctica de 98% (debido a truncamientos de la salida del *LLM*), 92% de solubilidad y 87% de correctitud. Finalmente, con la integración de los reintentos, el agente `exp` alcanzó en el entrenamiento una validez sintáctica del 100%, solubilidad del 99%, y 97% de correctitud. Este análisis sugiere que la combinación de capacidades de modelado robustas con mecanismos adaptativos de auto-mejora contribuye significativamente a la capacidad del agente para abordar una amplia gama de desafíos de planificación.

4.7. Aplicación de reintentos, *feedback* y reflexión a la evaluación

El agente `exp` aplicó la política de reintentos definida anteriormente a los problemas modelados incorrectamente por el agente `r_o_daps_gcd_fsp`, sobre el subconjunto de evaluación. Las salidas (razonamiento, objetos extraídos y *PDDL* del problema) generados en su primer y único intento por tarea, y el *feedback* construido automáticamente sobre este fallo, contribuyeron a un segundo intento que incluyó reflexión sobre la retroalimentación dada. De igual forma, se permitió un tercer intento apoyado en los resultados del segundo, si este era incorrecto.

La evaluación de `exp` se resume en la tabla siguiente:

Tabla 4.4: Resultados del Agente en el conjunto de Evaluación

Intento 1			Intento 2			Intento 3			Número de Problemas
\mathcal{P}_1	\mathcal{S}_1	\mathcal{C}_1	\mathcal{P}_2	\mathcal{S}_2	\mathcal{C}_2	\mathcal{P}_3	\mathcal{S}_3	\mathcal{C}_3	
V	V	V	-	-	-	-	-	-	57
V	V	F	V	V	F	V	V	F	5
V	V	F	V	V	F	F	F	F	2
V	V	F	V	F	F	V	V	F	1
V	V	F	F	F	F	V	V	F	1
V	F	F	V	V	F	V	V	V	2
V	F	F	V	V	F	V	F	F	1
V	F	F	V	F	F	V	F	F	1

Los resultados obtenidos en la evaluación son menos alentadores que los del entrenamiento. En general, el agente con reflexión (`exp`) mostró dificultades para mejorar su desempeño, e incluso en algunos casos obtuvo resultados inferiores en reintentos. Esto podría atribuirse a la mayor complejidad inherente a los problemas no resueltos, que frecuentemente involucraban modelos *PDDL* con un centenar de proposiciones. Los reintentos se caracterizaron por dos factores que afectaron negativamente al agente: un contexto mucho más extenso, influenciado por las salidas previas, y una reflexión propensa a alucinaciones. La degradación del rendimiento de los *LLMs* con contextos extensos es un fenómeno bien documentado (T. Li et al., 2024). Esta problemática se exacerbó en los problemas del dominio *Floor-Tile*, ausente en el conjunto de entrenamiento anteriormente analizado. Este dominio requiere múltiples proposiciones para definir explícitamente la adyacencia entre celdas, utilizando predicados como (*up tileX tileY*) y (*right tileX tileY*). Con frecuencia, el agente alucinaba la existencia y necesidad de predicados similares como *left* y *down*. Aun-

que estos predicados no son permitidos por la restricción gramatical implementada con *DAPS GCD*, su alucinación en la reflexión o razonamiento constituye ruido que dificulta el modelado correcto.

A pesar de estos resultados iniciales, el potencial de los mecanismos propuestos no debe descartarse. De los 13 problemas que el agente no logró modelar correctamente en el intento inicial de la etapa de evaluación, se corrigieron 2 gracias a los reintentos. La evaluación inicial ya mostraba resultados elevados, con un 100 % de validez sintáctica, un 94.29 % de solubilidad y un 81.43 % de correctitud, con dificultad de mejoría. Sin embargo, al considerar el mejor resultado obtenido en los reintentos para cada problema, estas métricas se incrementan a 100 %, 98.57 % y 84.29 %, respectivamente.

4.8. Extracción de *insights*

La fase de extracción de *insights*, posterior a la acumulación de experiencias durante el entrenamiento, se llevó a cabo de forma exploratoria. Sin embargo, los resultados iniciales no fueron del todo satisfactorios. Los *insights* generados por el *LLM*, actuando como agente extractor, tendían a ser vagos, genéricos o poco relevantes. En muchos casos, no lograban captar conocimiento concreto o útil del dominio que pudiera contribuir de forma significativa al modelado de nuevos problemas.

Además, las respuestas del modelo presentaban divagaciones extensas, con escasa alineación a las operaciones válidas del entorno, y sin un control efectivo sobre las restricciones sintácticas y semánticas de las acciones. Esto generó dificultades importantes para el análisis automático de las salidas, afectando la robustez del procesamiento posterior y provocando errores de ejecución en algunos casos. Estos comportamientos apuntan a posibles deficiencias en la ingeniería de *prompts*, particularmente en su capacidad para guiar al modelo hacia la identificación de patrones estructurados y reutilizables de conocimiento relevante.

4.9. Evaluación del agente experiencial con todos los módulos incluidos

En la fase final de experimentación se evaluó el agente modelador completo, que integra todos los componentes diseñados en esta tesis para mejorar progresivamente los resultados. Este agente experiencial, reforzado con ejemplos derivados de experiencias anteriores, incorpora módulos de recuperación de ejemplos vía *RAG* y conocimiento experto curado manualmente.

En concreto, se reutilizaron los ejemplos de *FSP* aprendidos durante la fase de entrenamiento para los dominios *Blocksworld* y *Gripper*, recuperados automáticamente mediante *RAG*. Para el dominio *Floor-Tile*, sin instancias incluidas en el entrenamiento, se empleó el ejemplo construido manualmente.

Dado el rendimiento limitado observado en la extracción automática de *insights*, se diseñó y empleó una base de *Human Insights* (*HI*) creada manualmente. Esta decisión permite evaluar el impacto de integrar conocimiento estructurado y explícito, extraído por un experto humano, en lugar de depender exclusivamente de la inferencia del modelo, cuya mejora se delega a trabajo futuro.

El agente evaluado, denominado `exp_rag_hi`, combina así dos mecanismos de conocimiento experiencial: *Retrieval-Augmented Generation* (*RAG*) y *Human Insights* (*HI*). Los resultados obtenidos se presentan a continuación.

Tabla 4.5: Resultados en el conjunto completo de evaluación

Intento 1			Intento 2			Intento 3			Número de Problemas
P_1	S_1	C_1	P_2	S_2	C_2	P_3	S_3	C_3	
V	V	V	-	-	-	-	-	-	57
V	V	F	V	V	V	-	-	-	1
V	V	F	V	V	F	V	V	V	1
V	V	F	V	V	F	V	V	F	8
V	V	F	V	V	F	F	F	F	1
V	F	F	V	V	V	-	-	-	2

Tabla 4.6: Resultados restringidos a los dominios *Blocksworld* y *Gripper*

Intento 1			Intento 2			Intento 3			Número de Problemas
P_1	S_1	C_1	P_2	S_2	C_2	P_3	S_3	C_3	
V	V	V	-	-	-	-	-	-	38
V	V	F	V	V	V	-	-	-	1
V	V	F	V	V	F	V	V	V	1
V	V	F	V	V	F	V	V	F	1
V	F	F	V	V	V	-	-	-	2

Considerando únicamente el primer intento del agente en cada problema, `exp_rag_hi` iguala los niveles de validez sintáctica (100 %) y correctitud (81.43 %) logrados por `r_o_daps_gcd_fsp`, pero mejora de forma significativa la tasa de solubilidad, que asciende a 97.14 %. Esta diferencia se vuelve aún más relevante al considerar el mejor resultado entre los reintentos permitidos por el sistema: `exp_rag_hi` alcanza una solubilidad del 100 % y eleva la correctitud hasta 87.14 %.

El análisis centrado en los dominios *Blocksworld* y *Gripper*, para los que el agente disponía de ejemplos de entrenamiento exitosos, revela mejoras aún más marcadas. Mientras `r_o_daps_gcd_fsp` alcanzaba 100 % de validez sintáctica, 90.70 % de solubilidad y 88.37 % de correctitud, el agente `exp_rag_hi` eleva estos valores a 100 %, 100 % y 97.67 %, respectivamente, estableciendo un nuevo umbral de desempeño dentro del conjunto experimental.

4.10. Limitaciones de la experimentación y evaluación

Esta experimentación presenta varias limitaciones. Principalmente: no fue posible evaluar el resultado de los agentes resultantes de todas las combinaciones de los módulos propuestos, ni evaluar a los agentes en todo el *dataset* de *Planetarium*. Ambas limitantes tienen la misma causa: el costo computacional, económico y temporal que implicaría semejante evaluación a gran escala. Después de las correcciones realizadas a la generación del *dataset* de *Planetarium*, la cantidad de tareas de este era de 134485, separadas en 15957 del *split test* y 118528 del *split train*, frente a los 70 problemas utilizados para la evaluación y los 100 utilizados para el entrenamiento.

Se hizo una evaluación de los 12 agentes no experienciales descritos en los 70 problemas seleccionados aleatoriamente sobre una preselección balanceada por todas las dimensiones del *dataset*. Aunque estos no

representaban la distribución real del *dataset*, para balancear todas las dimensiones del mismo, no están demasiado alejados. Esta evaluación, aunque modesta, requirió 2378725 *tokens* de *prompt* y 649094 de *completado* en las consultas a la *API* de *Fireworks AI*. El *LLM* utilizado, **Llama 4 Maverick Instruct (Basic)**, tenía un costo de \$0.22 por millón de *tokens* de *prompt*, y \$0.88 por millón de *tokens* de *completado* ([FireworksAI, 2025](#)). Esto daba un costo de $\$0.22 \times 2378725 / 1000000 = \0.52 en *prompt* y $\$0.88 \times 649094 / 1000000 = \0.57 en *completado*, para un costo total de \$1.09 (sin contar este costo quintuplicado durante la experimentación previa a los resultados finales). Extrapolando estos costos a la totalidad del *dataset*, resulta en aproximadamente \$2100, lo cual supera con creces el presupuesto de esta modesta tesis de pregrado.

Más aún, el tiempo total de generación de los *LLMs* consultados durante el proceso de evaluación de los 12 agentes (que no incluían mecanismos de reintentos y conocimiento experiencial) en los 70 problemas fue de 11979 segundos, es decir, casi 200 minutos, o 3 horas y un tercio. Esto no tiene en cuenta el tiempo de evaluación de las soluciones, la carga de recursos y otras complejidades algorítmicas, aunque se puede decir con seguridad que la evaluación completa tomó más de 4 horas. Extrapolando este costo temporal al *dataset* entero, se hubieran requerido aproximadamente 23014226 segundos, es decir, 383570 minutos, o 6392 horas, o 266 días...

Si se tienen en cuenta todas las combinaciones de módulos propuestos, el resultado crece exponencialmente. Debido a esto, se hizo el análisis por componentes de forma incremental, comparando directamente la aplicación de los componentes en un orden lógico, lo cual no garantiza el análisis de la influencia aislada de estos módulos. Sin embargo, esta limitación no es tan grave, debido a la sinergia que comparten estos componentes, y que un objetivo es demostrar su utilidad conjunta.

Para aliviar la limitación sobre el subconjunto reducido de problemas a evaluar del *dataset*, se optó por hacer la selección de forma que se cubrieran y se balancearan todas las dimensiones del *dataset*, dígase los dominios, configuraciones o subtipos de estado inicial y estado objetivo, su abstracción, cantidad de proposiciones en el *PDDL* de referencia y cantidad de objetos participantes. La selección aleatoria permitió seguir limitadamente dicha distribución. Además, los resultados se presentan con sus intervalos de confianza calculados, para lo cual se utilizó la técnica de *Bootstrapping* ([Efron y Tibshirani, 1994](#)) con 100000 remuestreos. Esta técnica consiste en generar muchas muestras con reemplazo a partir de la muestra original, y calcular los estadísticos de interés (en este caso, la media) sobre cada una de estas muestras generadas. El intervalo de confianza se obtiene a partir de los percentiles correspondientes a los niveles deseados: 2.5 % y 97.5 % para un intervalo de confianza del 95 %.

Otras limitaciones relacionadas tienen que ver con el modelo *LLM* base y la configuración de los parámetros de comportamiento de los agentes. Aumentar el número de modelos a evaluar o explorar más configuraciones de parámetros multiplica todos los costos anteriores. Las selecciones de los parámetros utilizados constituyen decisiones de diseño, fundamentadas también en los valores observados en la literatura, en particular en el trabajo precedente de *ExpeL*.

Como en la mayoría de trabajos con *LLMs*, no se condujo un análisis de la variación de los *prompts* utilizados, y resulta infactible probar siquiera una pequeña fracción de las variaciones lógicas posibles a dichas instrucciones y contextos. Los *prompts* finales deben su uso a la búsqueda de una representación muy estructurada del contexto de los problemas, y de la instrucción, de forma concisa y directa, al tiempo que dieron los mejores resultados experimentales previos.

Otra limitación es que los resultados, y por tanto las hipótesis, se validan sobre los tres dominios del *benchmark* *Planetarium*, de acotada complejidad, frente a la vasta y diversa cantidad de dominios y *benchmarks* actuales en el campo de investigación, que tampoco pueden reflejar la complejidad de los entornos prácticos de la vida cotidiana o la industria. Estos dominios, más bien, corresponden a lo que se conoce como *toy problems*. En contextos científicos, un *toy problem* es un problema que no tiene un

interés inmediato en sí mismo, pero que se utiliza como recurso expositivo para ilustrar una característica que puede estar presente en instancias más complejas, o para explicar una técnica general de resolución de problemas. Son útiles para probar y demostrar metodologías, y comparar el rendimiento de diferentes algoritmos. En sistemas complejos, se suelen descomponer los problemas grandes en muchos *toy problems* más pequeños que han sido bien entendidos. Frecuentemente, estos problemas destilan algunos aspectos importantes de problemas más complicados para que puedan estudiarse de forma aislada. Por ello, los *toy problems* resultan útiles para generar intuiciones sobre fenómenos que se manifiestan en situaciones más complejas, algo alineado con los objetivos de esta tesis.

4.11. Resumen de los resultados

Con la inclusión incremental de los módulos de mejora propuestos se observa, por lo general, un crecimiento sostenido de las métricas evaluadas.

- **Objetos correctos:**

- En modalidad *Zero-Shot*: `orig_llm_plus_p` (32.86%) → `llm_plus_p` (62.86%) → `r` (62.86%) → `r_o` (95.71%) → `r_o_daps_gcd` (92.86%). El salto cuantitativo relevante lo ofrece la inclusión del módulo de **Extracción de objetos (o)** en `r_o`, que mejora en 32.85 puntos con respecto al *baseline* `llm_plus_p`, lo cual representa un 88.44% del margen de mejora.
- En modalidad *One-Shot* todos los agentes alcanzan el 100% en esta métrica, lo cual sugiere que incluso configuraciones simples pueden beneficiarse significativamente de la exposición a un único ejemplo resuelto correctamente.

- **Validez sintáctica:**

- En modalidad *Zero-Shot*: `orig_llm_plus_p` (34.29%) → `llm_plus_p` (87.14%) → `r` (82.86%) → `r_o` (84.29%) → `r_o_daps_gcd` (97.14%). El incremento más significativo ocurre con la inclusión del módulo **DAPS GCD (daps_gcd)**, que eleva la tasa de precisión en 10 puntos con respecto a `llm_plus_p`, lo cual cubre un 77.76% de su margen de mejora.
- En modalidad *One-Shot*: `orig_llm_plus_p_fsp` (80.00%) → `llm_plus_p_fsp` (87.14%) → `r_fsp` (90.00%) → `r_o_fsp` (91.43%) → `r_o_daps_gcd_fsp` (100.00%). El efecto acumulativo de los módulos adicionales permite cerrar completamente el margen restante, con el módulo `daps_gcd` como último y más relevante salto cuantitativo.

- **Solubilidad:**

- En modalidad *Zero-Shot*: `orig_llm_plus_p` (0.00%) → `llm_plus_p` (54.29%) → `r` (62.86%) → `r_o` (78.57%) → `r_o_daps_gcd` (85.71%). La inclusión de cada componente beneficia en gran medida a los resultados en esta métrica, donde el resultado final del mejor agente `r_o_daps_gcd` supera en 31.42 puntos al *baseline* `llm_plus_p`, lo que consiste en un cierre de 68.74% del margen al resultado perfecto.
- En modalidad *One-Shot*: `orig_llm_plus_p_fsp` (62.86%) → `llm_plus_p_fsp` (82.86%) → `r_fsp` (87.14%) → `r_o_fsp` (88.57%) → `r_o_daps_gcd_fsp` (94.29%). Se observa igualmente la mejora acumulativa que brinda cada módulo, en especial el **Razonamiento estructurado (r)** y el **DAPS GCD (daps_gcd)**, consiguiendo una mejora en `r_o_daps_gcd_fsp` de 11.43 puntos de porcentaje, salvando efectivamente el 66.68% del error del *baseline* `llm_plus_p_fsp`.

- Sobre estos resultados ya obtenidos, el agente `exp` aplica reintentos únicamente sobre los problemas fallados por `r_o_daps_gcd_fsp`, usando además *feedback* automático y reflexión para mejorar la salida. Esta política logra elevar la solubilidad a **98.57%**. En contraste, `exp_rag_hi` se evalúa desde cero sobre todo el subconjunto de evaluación, permitiendo hasta tres intentos por problema y seleccionando el mejor. En este esquema, alcanza una solubilidad perfecta de **100 %**, validando la efectividad de integrar *RAG* y *Human Insights*.

■ **Correctitud:**

- En modalidad *Zero-Shot*: `orig_llm_plus_p` (0.00 %) → `llm_plus_p` (22.86 %) → `r` (30.00 %) → `r_o` (**42.86 %**) → `r_o_daps_gcd` (**41.43 %**). Las mejoras más relevantes se dan al incorporar los módulos de **Razonamiento estructurado (r)** y de **Extracción de objetos (o)**, que aportan en conjunto 20 puntos respecto al *baseline* `llm_plus_p_fsp`, abarcando el 25.92% del margen de mejora posible.
- En modalidad *One-Shot*: `orig_llm_plus_p_fsp` (**47.14 %**) → `llm_plus_p_fsp` (**57.14 %**) → `r_fsp` (**72.86 %**) → `r_o_fsp` (**75.71 %**) → `r_o_daps_gcd_fsp` (**81.43 %**). Se obtiene consistentemente una mejora incremental con cada módulo, en especial del **Razonamiento estructurado (r)**, que le valen a `r_o_daps_gcd_fsp` una mejora total de **24.29** puntos porcentuales, consiguiendo el **56.67 %** de la mejora posible del *baseline* `llm_plus_p_fsp`.
- Sobre esta base, `exp` logra una mejora moderada al corregir dos fallos mediante reintentos informados, alcanzando una correctitud de **84.29 %**. Por su parte, `exp_rag_hi` consigue una mejora más significativa en el contexto de reintentos completos, alcanzando **87.14 %** de correctitud al considerar el mejor intento por problema.

4.12. Análisis del cumplimiento de las hipótesis

A continuación, se analiza el cumplimiento de las hipótesis **H1**, **H2**, **H3** y **H4** a la luz de los resultados obtenidos, restringiendo la discusión exclusivamente al marco de la evaluación realizada y reconociendo explícitamente sus limitaciones metodológicas y empíricas.

H1. División del proceso de modelado en fases estructuradas

Hipótesis H1. La división del proceso de modelado en fases estructuradas —extracción de objetos, razonamiento, especificación del estado inicial y metas, y generación del archivo *PDDL*— mejora la correctitud de los modelos generados, al permitir un razonamiento más controlado, modular y verificable.

Los resultados observados ofrecen apoyo empírico para esta hipótesis. Si bien el diseño experimental no permite aislar completamente los efectos individuales de cada fase en todos los casos (dado que algunos agentes combinan múltiples componentes), se cuenta con comparaciones controladas entre variantes que difieren únicamente en la presencia de razonamiento estructurado o extracción de objetos. Estas comparaciones permiten inferencias válidas sobre el impacto de dichas fases.

En particular, la incorporación de razonamiento estructurado en la transición de `llm_plus_p` a `r`, y luego de extracción de objetos en `r_o`, muestra un incremento en la *correctitud total* desde 22.86 % → 30.00 % → 42.86 %, respectivamente. Este patrón de mejora se mantiene cuando se introduce *Few-Shot Prompting (FSP)*, lo que permite observar una secuencia más completa: `orig_llm_plus_p_fsp` (**47.14 %**) → `llm_plus_p_fsp` (**57.14 %**) → `r_fsp` (**72.86 %**) → `r_o_fsp` (**75.71 %**) → `r_o_daps_gcd_fsp` (**81.43 %**). Esta progresión refuerza la hipótesis de que una estructuración explícita del proceso cognitivo del agente mejora significativamente la calidad semántica de los modelos generados.

No obstante, deben tenerse presentes varias limitaciones del estudio. La evaluación se restringe a un subconjunto representativo, pero reducido, del *benchmark Planetarium*, con problemas de dominio público relativamente simples en complejidad. Además, aunque se emplearon intervalos de confianza para estimar la incertidumbre, no se aplicaron pruebas de significancia estadística formal entre pares de agentes. Por tanto, si bien los patrones observados son consistentes y plausibles, no pueden considerarse como evidencia definitiva de causalidad en entornos más generales o exigentes.

En conclusión, la hipótesis **H1** se considera respaldada dentro del marco experimental considerado. La evidencia sugiere que la división en fases estructuradas aporta claridad, control y modularidad al proceso de modelado, con impactos positivos cuantificables sobre la *correctitud total* del modelo *PDDL* generado.

H2. Aplicación de Grammar-Constrained Decoding (GCD)

Hipótesis H2. La aplicación de *GCD* permite una generación más confiable del código *PDDL*, reduciendo significativamente o eliminando por completo la aparición de errores de sintaxis.

Los resultados empíricos obtenidos permiten validar con solidez esta hipótesis, dentro del alcance de la evaluación realizada. En particular, la métrica de validez sintáctica muestra mejoras consistentes al incorporar el componente *GCD*, y aún más notorias al utilizar su versión especializada *DAPS*.

Por ejemplo, en el entorno sin *FSP*, el paso de *r_o* a *r_o_gcd* mejora la validez sintáctica de 84.29 % a 87.14 %. Al aplicar *DAPS*, *r_o_daps_gcd* alcanza 97.14 %. En el entorno con *FSP*, donde ya se observan altos niveles de validez sintáctica, la secuencia culmina en *r_o_daps_gcd_fsp* con un 100 %, eliminando por completo los errores de sintaxis.

Estos resultados confirman la utilidad de *GCD* como mecanismo efectivo para prevenir errores de generación que violen las reglas gramaticales del lenguaje objetivo. Sin embargo, también se identifican limitaciones importantes. La modalidad general de *GCD* no evita por completo errores de sintaxis, especialmente cuando el *LLM* inserta razonamientos o frases fuera de lugar dentro de secciones que requieren literalidad. Sólo mediante la especialización del conjunto de producciones gramaticales —como lo hace *DAPS GCD*— se logra un control completo y fiable sobre la estructura de la salida.

Es importante señalar que estas conclusiones están limitadas por el uso de una gramática formal derivada de un subconjunto de *PDDL*, centrado en las funcionalidades de *STRIPS* con posible tipado simple sin multiherencia. No se evaluaron gramáticas para extensiones más complejas como *numeric fluents*, *temporal actions* o *derived predicates*, por lo que los resultados no pueden generalizarse directamente a todos los dialectos de *PDDL*. Esta restricción en sí se debe a que esta investigación se limita a modelado de problemas clásicos de planificación, que se formulan bajo un conjunto de supuestos idealizados: el entorno es discreto, determinista, estático, completamente observable y monológico (de un solo agente), por lo que la modalidad de planificación es *Open-Loop*. Precisamente el subconjunto de *PDDL* tomado cubre específicamente dichas restricciones.

En síntesis, dentro del alcance y limitaciones de esta tesis, la hipótesis **H2** queda empíricamente respaldada. La aplicación de *GCD* —y especialmente su modalidad especializada *DAPS*— permite una generación completamente libre de errores sintácticos en todos los casos evaluados, demostrando su eficacia como componente esencial en el *pipeline* de modelado automático.

H3. Reflexión sobre errores y retroalimentación automática

Hipótesis H3. La introducción de reflexión sobre errores y una mínima retroalimentación humana o automática permite al agente corregir patrones de falla recurrentes, contribuyendo al aumento de la solubilidad y la correctitud del *PDDL* generado.

La validación de esta hipótesis se llevó a cabo mediante la implementación de un agente modelador experiencial (`exp`) que, sobre una base sólida de modelado automático (`r_o_daps_gcd_fsp`), incorporó un mecanismo de reintentos guiado por *feedback* automático y autorreflexión. El sistema permitía hasta dos reintentos nuevos por problema, utilizando las salidas y errores previos como insumo para generar correcciones guiadas.

Durante la fase de entrenamiento, aplicada a un subconjunto de 100 problemas conocidos, los resultados fueron notablemente positivos: el agente logró corregir 10 de los 13 errores cometidos inicialmente, alcanzando una validez sintáctica del 100 %, una solubilidad del 99 % y una correctitud del 97 %. Este desempeño indica que los mecanismos de reintento, cuando son aplicados sobre dominios conocidos y con un contexto relativamente controlado, pueden corregir con éxito patrones de falla y mejorar de manera significativa el rendimiento del agente.

Sin embargo, los resultados obtenidos durante la fase de evaluación revelan ciertas limitaciones. El agente `exp` aplicó su política de reintentos únicamente sobre los problemas modelados incorrectamente por `r_o_daps_gcd_fsp`. Aunque el desempeño inicial ya era elevado (100 % de validez sintáctica, 94.29 % de solubilidad y 81.43 % de correctitud), los mecanismos de reintento permitieron corregir 2 de los 13 errores restantes, elevando las métricas finales a 100 %, 98.57 % y 84.29 %, respectivamente. Si bien esto representa una mejora modesta en el número absoluto de problemas corregidos, sí valida parcialmente la hipótesis: el uso de reflexión y *feedback* automático puede mejorar el desempeño incluso cuando se parte de una base sólida. El análisis de los fallos persistentes sugiere que la dificultad de algunos problemas se debía tanto a la complejidad del nuevo dominio (*Floor-Tile*) como a limitaciones del modelo en contextos extensos.

No obstante, la efectividad del enfoque se ve reforzada por los resultados del agente `exp_rag_hi`, que combinó mecanismos de recuperación de ejemplos y conocimientos curados manualmente (*Human Insights*) con la política de reintentos. En este caso, la evaluación completa del agente sobre todo el conjunto experimental (no solo los errores de otro agente) arrojó mejoras más claras: alcanzó una solubilidad del 100 % y una correctitud del 87.14 %, partiendo de una base de 97.14 % y 81.43 % en su primer intento. Este resultado sugiere que los mecanismos reflexivos y adaptativos tienen un mayor impacto cuando se combinan con conocimiento previo estructurado y ejemplos adecuados.

En resumen, los resultados permiten considerar empíricamente respaldada la hipótesis **H3**, aunque con matices. La reflexión sobre errores y el *feedback* automático son herramientas útiles para mejorar el desempeño de un agente modelador, pero su efectividad puede depender del dominio, del contenido y claridad del contexto previo, y de la disponibilidad de conocimiento estructurado que apoye la corrección. En ausencia de estos elementos, los beneficios pueden verse limitados.

H4. Recuperación de ejemplos e incorporación de *insights*

Hipótesis H4. La incorporación de *RAG* para la selección de ejemplos relevantes y la extracción de *insights* a partir de soluciones previas (tanto correctas como erróneas), representa una vía prometedora para mejorar la capacidad de los agentes basados en *LLMs* para modelar tareas de planificación, al permitirles adaptarse a la semántica de nuevas tareas y fortalecer su conocimiento sobre el dominio específico y la modelación de problemas.

La validación de esta hipótesis se abordó a través de dos mecanismos complementarios: la extracción de *insights* desde experiencias pasadas, y la recuperación de ejemplos relevantes mediante *Retrieval-Augmented Generation (RAG)*. En principio, se esperaba que ambos mecanismos, basados en el aprendizaje experiencial del agente, sirvieran como fuentes de conocimiento acumulado que pudieran ser explotadas en fases futuras de modelado.

En la práctica, la extracción automática de *insights* a partir de soluciones anteriores (tanto correctas como erróneas) no arrojó los resultados esperados. Las salidas generadas por el modelo, actuando como

agente extractor, tendían a ser demasiado genéricas, carentes de contenido estructurado, o incluso irrelevantes para el dominio y la tarea. Con frecuencia, el modelo producía divagaciones extensas con pobre alineación a las restricciones sintácticas de las operaciones permitidas para la interacción con la base de *insights*, lo que dificultaba su aprovechamiento. Estas deficiencias limitan el valor práctico de esta estrategia en su forma actual, y apuntan a causas de no optimalidad del *prompt*, así como a la necesidad de mayor control estructural sobre el contenido extraído.

Ante estas limitaciones, se construyó una base manual de *Human Insights* (HI), que actúa como *proxy* de una extracción efectiva. Esta base consistía en observaciones expertas estructuradas, condensando buenas prácticas de modelado, estructuras comunes, y conocimiento físico o abstracto clave de los dominios. Su incorporación tenía el objetivo de evaluar el impacto potencial de contar con *insights* útiles, independientemente del mecanismo con que estos se obtuvieran, permitiendo así explorar de forma indirecta el beneficio de una futura extracción experiencial exitosa.

Sobre esta base, se integró también un mecanismo de *RAG* para recuperar automáticamente ejemplos previamente exitosos del entrenamiento, utilizando medidas de similitud textual entre tareas. Este mecanismo fue especialmente útil para los dominios *Blocksworld* y *Gripper*, sobre los cuales el agente había acumulado experiencia previa.

El agente evaluado con estas capacidades fue `exp_rag_hi`, que integra *RAG* con ejemplos útiles y la base de *Human Insights*. Considerando únicamente el primer intento de modelado por problema —el más representativo del efecto directo de los módulos, sin influencia de reintentos o reflexión—, `exp_rag_hi` igualó los niveles de validez sintáctica (100 %) y correctitud (81.43 %) de `r_o_daps_gcd_fsp`, pero mejoró significativamente la tasa de solubilidad, que aumentó de 94.29 % a 97.14 %. Al considerar el mejor resultado entre los reintentos permitidos, estas métricas se elevan aún más: 100 % en validez sintáctica y solubilidad, y 87.14 % en correctitud. Este efecto es especialmente marcado en los dominios *Blocksworld* y *Gripper*, donde el agente disponía de ejemplos aprendidos durante el entrenamiento: en este subconjunto, `exp_rag_hi` alcanzó 100 % de validez sintáctica y solubilidad, y 97.67 % de correctitud, superando a `r_o_daps_gcd_fsp`, que lograba 100 %, 90.70 % y 88.37 % en las mismas métricas, respectivamente. Estos resultados cuantifican con claridad el impacto positivo de integrar conocimiento reutilizable —en forma de ejemplos previos e *insights* estructurados— en el proceso de modelado.

Cabe destacar que esta mejora se obtiene en un entorno ya altamente optimizado por otros mecanismos adicionales presentes en el agente, tales como razonamiento estructurado y *DAPS GCD*. Por tanto, la validación de **H4** se realiza en un contexto donde los beneficios de *RAG* e *insights* se expresan en sinergia con una arquitectura compleja. A pesar de esta interacción, el salto observado en solubilidad y correctitud refuerza la utilidad práctica de incorporar mecanismos de conocimiento experiencial en agentes de modelado automático.

En resumen, los resultados respaldan de forma parcial y condicionada la hipótesis **H4**. La extracción automática de *insights* desde la experiencia del modelo aún presenta desafíos abiertos y se configura más como una línea futura de desarrollo que como una técnica consolidada en esta tesis. No obstante, el uso de conocimiento experto preestructurado y su combinación con ejemplos recuperados mediante *RAG* muestra un efecto positivo y cuantificable sobre el rendimiento del agente. Esto valida la dirección planteada por la hipótesis, de carácter más exploratorio, y abre un camino prometedor para la integración de aprendizaje experiencial más robusto en sistemas de modelado automático.

Conclusiones

La presente tesis tuvo como objetivo general diseñar, implementar y evaluar agentes basados en *Large Language Models (LLMs)* con técnicas propuestas para mejorar la correctitud sintáctica y semántica de la generación de códigos *PDDL* en modelación de tareas de planificación, además de realizar un análisis comparativo de los agentes implementados para determinar los factores que influyen positivamente en métricas clave como validez sintáctica, solubilidad y correctitud. Este objetivo se basó en las limitaciones y oportunidades observadas en la revisión de la literatura, específicamente: la potencial futilidad del paradigma *LLM-as-Planner*; el cuello de botella que representa la fase de modelado de los problemas de planificación de los algoritmos clásicos; las limitaciones de los trabajos actuales situados en el paradigma *LLM-as-Modeler*, que son la presencia de errores sintácticos, inconsistencias semánticas y fuerte dependencia de conocimiento, supervisión y labor experta; el potencial de los métodos que inducen razonamiento intermedio en los *LLMs*, la técnica de *Grammar-Constrained Decoding (GCD)* y los mecanismos de aprendizaje experiencial introducidos por el trabajo de *ExpeL*. A lo largo del trabajo se cumplieron todos los objetivos específicos planteados, a través de una propuesta sólida, una implementación exhaustiva y una evaluación empírica rigurosa, aunque limitada por restricciones computacionales, económicas y temporales.

En primer lugar, se realizó un estudio profundo del estado del arte, que permitió identificar las limitaciones actuales de los enfoques basados en *LLMs* para la planificación automática, en particular la falta de robustez sintáctica y semántica de los modelos generados, así como la ausencia de mecanismos que guíen o estructuren el razonamiento del agente. A partir de este análisis, se propuso un agente modelador de problemas de planificación capaz de generar automáticamente modelos *PDDL* válidos a partir de descripciones en lenguaje natural, valiéndose de técnicas como razonamiento estructurado, extracción de objetos, *Few-Shot Prompting (FSP)*, *Retrieval-Augmented Generation (RAG)*, aprendizaje experiencial con extracción de *insights* y reflexión, y generación controlada por gramática con *GCD* y una variante especializada introducida denominada *DAPS*.

La implementación del agente fue acompañada de una estructura modular y extensible, incluyendo las variantes experimentales de cada uno de sus componentes, así como los *baselines* de la literatura con los que se comparó el desempeño. Se construyó además un *pipeline* completo de entrenamiento, evaluación y análisis de los agentes, junto a un conjunto de herramientas auxiliares para la generación y validación automática de archivos *PDDL*. Todo este desarrollo fue publicado en un repositorio de *GitHub* (github.com/arielgg46/Thesis), documentado y reproducible, lo cual constituye uno de los principales aportes prácticos de esta tesis para la comunidad investigadora. Además, se identificaron errores en la generación del *dataset Planetarium*, y se propusieron e implementaron correcciones, ayudando a garantizar la correctitud y utilidad de tan importante *benchmark* para la evaluación de agentes modeladores de problemas de planificación.

Durante la etapa de evaluación, se diseñó un experimento basado en subconjuntos estratificados del *benchmark Planetarium*, seleccionados para balancear las distintas dimensiones del espacio de problemas (como dominios, estructuras del estado inicial y metas, cantidad de objetos y predicados, etc.). Se evaluaron 13 agentes en un total de 70 tareas de prueba, midiendo su desempeño en cuanto a validez sintáctica, solu-

bilidad y correctitud, y se realizó un análisis incremental para determinar el impacto de cada componente individual y de sus combinaciones.

A pesar de las limitaciones experimentales derivadas del alto costo computacional, económico y temporal de la evaluación a gran escala, el estudio ofreció evidencia empírica clara sobre la efectividad de los módulos propuestos. En particular:

- La división del proceso de modelado en fases estructuradas permitió mejoras sustanciales en la correctitud semántica, facilitando el razonamiento modular y verificable del agente. Específicamente, en métricas de validez sintáctica, solubilidad y correctitud, se pasó respectivamente de **87.14%**, **54.29%** y **22.86%** en el *baseline Zero-Shot* reimplementado (`11m_plus_p`) a **84.29%**, **78.57%** y **42.86%** con la inclusión de estos módulos en `r_o`; y de **87.14%**, **82.86%** y **57.14%** en el *baseline One-Shot* reimplementado (`11m_plus_p_fsp`) a **91.43%**, **88.57%** y **75.71%** en `r_o_fsp`. Esta conclusión respalda la hipótesis **H1** en el contexto limitado evaluado.
- La aplicación de *GCD*, en concreto en su forma especializada *DAPS*, eliminó por completo los errores sintácticos en los experimentos realizados, validando empíricamente la hipótesis **H2** dentro del alcance de esta tesis. Se partió de una validez sintáctica de **87.14%** en el *baseline* reimplementado a una de **100%** en modalidad *One-Shot*.
- El agente que integró todos los componentes propuestos alcanzó los mayores valores en las métricas clave: **100 %** de validez sintáctica, **87.14 %** de solubilidad y **81.43 %** de correctitud, dentro del conjunto de evaluación construido, superando ampliamente tanto a sus versiones parciales como a los *baselines* extraídos de la literatura. Se cerraron efectivamente el **100 %**, **66.68 %** y **56.67 %** de los márgenes de mejora del *baseline* `11m_plus_p_fsp` en las métricas respectivas.
- Al incorporar mecanismos de reflexión y retroalimentación automática (`exp`), se corrigieron errores residuales, elevando la solubilidad a **98.57 %** y la correctitud a **84.29 %**, lo que valida parcialmente la hipótesis **H3**. Finalmente, la combinación sinérgica de reflexión, reintentos, recuperación de ejemplos previos e *insights* construidos manualmente (`exp_rag_hi`) permitió alcanzar los valores más altos de todo el estudio: **100 %** de validez sintáctica, **100 %** de solubilidad y **87.14 %** de correctitud, respaldando empíricamente también la hipótesis **H4**. Este agente logró superar a `r_o_daps_gcd_fsp`, incluso en dominios complejos, como evidencia del potencial de integrar mecanismos de conocimiento reutilizable en el modelado automático.

No obstante, los resultados deben interpretarse dentro de los márgenes de validez impuestos por la evaluación. Las tareas utilizadas pertenecen a tres dominios clásicos del *benchmark Planetarium*, lo que limita la generalización a contextos más complejos o realistas. La combinación de módulos fue evaluada de forma incremental y no exhaustiva, lo cual no permite aislar por completo el efecto de cada técnica. Además, las decisiones de diseño de *prompts*, *LLMs* base, límites de operaciones e interacciones con los *LLMs* estuvieron sujetas a restricciones prácticas, sin exploración sistemática de hiperparámetros. Adicionalmente, la fase de extracción de *insights* no obtuvo el desempeño esperado, lo que limitó la validación de **H4** a la evaluación usando una base de *Human Insights*.

Pese a estas limitaciones, los resultados de la tesis constituyen un aporte significativo al campo del modelado automático con *LLMs* en planificación. La propuesta demuestra que es posible integrar múltiples técnicas complementarias para construir agentes robustos y adaptativos, capaces de generar modelos válidos y funcionales a partir de descripciones en lenguaje natural, con una tasa de éxito comparable a enfoques manuales o guiados por expertos humanos.

Este trabajo deja como legado no sólo una arquitectura funcional de agente modelador, sino también una base experimental, un marco de evaluación y un conjunto de herramientas replicables que sientan las bases para investigaciones futuras.

En conclusión, esta tesis cumple satisfactoriamente sus objetivos propuestos, aportando evidencia, herramientas y desarrollos que consolidan el uso de *LLMs* como una tecnología prometedora para la generación automática de modelos de planificación, y abre un camino viable hacia agentes más inteligentes, precisos y generalizables en tareas de modelado cognitivo.

Recomendaciones y Trabajo Futuro

Esta investigación deja abiertas diversas oportunidades para profundizar y extender los resultados obtenidos. En primer lugar, sería deseable abordar las limitaciones experimentales identificadas: la evaluación no abarcó todas las combinaciones posibles de módulos del agente, ni fue exhaustiva sobre el *dataset* de *Planetarium*, y tampoco incluyó un análisis sistemático de la sensibilidad a los *prompts*, modelos base o parámetros internos. Superar estas restricciones permitiría obtener conclusiones más robustas y generalizables sobre el impacto de cada componente del sistema.

Otra dirección prometedora consiste en ampliar el alcance del modelado más allá de los supuestos clásicos de la planificación automática, como la naturaleza determinista, estática, completamente observable y monoagente del entorno. La arquitectura propuesta podría extenderse a escenarios más realistas y desafiantes, como entornos estocásticos, parcialmente observables, multiagente, con restricciones temporales, dinámicos o con objetivos preferenciales. Además, se puede explorar la integración en entornos con entrada visual. De este modo, el sistema podría aplicarse a problemas más complejos y cercanos a los de la vida cotidiana o la industria.

Una línea prioritaria de investigación futura consiste en mejorar la calidad de la extracción automática de *insights* desde la experiencia del agente. Aunque esta fase fue parcialmente implementada en esta tesis, sus resultados fueron subóptimos debido a la baja estructuración y especificidad del contenido extraído. Se recomienda explorar técnicas que impongan restricciones gramaticales o esquemáticas más estrictas sobre las salidas del modelo extractor, como *GCD*. También se propone la optimización de los *prompts* que recibe el modelo extractor de *insights*, con énfasis en la derivación de conocimiento concreto y relevante.

Se contemplan adicionalmente otras hipótesis que la arquitectura confeccionada permite explorar:

- H5.** Los *insights* y ejemplos de *Few-Shot Prompting* específicos de un dominio pueden ser utilizados para mejorar los resultados del agente en otro dominio, como especie de *Transfer learning* ([Zhuang et al., 2020](#)) sin modificación de pesos, mediante *In-Context Learning*.
- H6.** Los resultados del entrenamiento (*insights* y ejemplos de soluciones correctas) del agente experiencial basado en un *LLM* más complejo y costoso pueden contribuir significativamente a los resultados del mismo agente basado en un *LLM* más sencillo (de varios cientos de millones menos de parámetros), abaratando costos mientras se mantiene un alto grado de rendimiento.

Se conciben algunas propuestas para su validación: se analiza una forma de transferencia de conocimiento entre dominios, donde se entrena al agente experiencial en dos dominios (*Blocksworld* y *Gripper*) y luego se aplica, sin reentrenamiento, en un dominio nuevo no visto durante el entrenamiento (*Floor-Tile*), mediante técnicas de *In-Context Learning*, conforme a la hipótesis **H5**. En todos los casos, los agentes se evalúan usando dos *LLMs*, uno básico y uno avanzado, y se considera el escenario en el cual el modelo más sencillo reutiliza los *insights* y ejemplos *FSP* generados por el modelo más complejo, como forma de reducir el costo computacional manteniendo el rendimiento, en línea con **H6**.

También se recomienda avanzar hacia una representación más abstracta de los modelos generados, incorporando extensiones de *PDDL* con cuantificadores lógicos. Esto permitiría preservar la generalidad de descripciones expresadas en lenguaje natural, evitando fijar arbitrariamente objetos específicos al traducir estados iniciales o metas. De este modo se facilitaría la generación de planes más eficientes y se reduciría la necesidad de desambiguación.

Una idea con gran potencial identificada durante el desarrollo de la tesis consiste en dotar al agente modelador de la capacidad de construir y refinar funciones o *skills* que generen proposiciones *PDDL* para resolver subtareas comunes. Estas funciones, codificadas por ejemplo en Python, podrían seleccionarse de forma contextual mediante *RAG* sobre un repositorio en evolución, permitiendo modularidad, reutilización y aprendizaje acumulativo. Esta estrategia se inspira en trabajos como *Voyager* (G. Wang et al., 2023), y representa un paso hacia agentes con conocimientos estructurados y transferibles.

Otras líneas complementarias incluyen la incorporación de fuentes de conocimiento externo (por ejemplo, bases de sentido común o conocimiento científico), el uso de herramientas auxiliares como módulos aritméticos, la especialización funcional de los agentes mediante diferentes *LLMs* por rol reentrenados con *Fine-Tuning*, la integración con memorias externas estructuradas, y la exploración de métodos de combinación o ensamblado de modelos (*Ensembles*). Estas estrategias pueden fortalecer la capacidad del sistema para razonar, generalizar y adaptarse a nuevos dominios, consolidando el agente modelador como una herramienta flexible y potente para tareas de planificación automática en entornos complejos.

Bibliografía

- Aeronautiques, C., Howe, A., Knoblock, C., McDermott, I. D., Ram, A., Veloso, M., Weld, D., Sri, D. W., Barrett, A., Christianson, D., et al. (1998). Pddl| the planning domain definition language. *Technical Report, Tech. Rep.* (vid. págs. 1, 8).
- Agarwal, S., & Sreepathy, A. (2024). TIC: Translate-Infer-Compile for accurate “text to plan” using LLMs and Logical Representations. *International Conference on Neural-Symbolic Learning and Reasoning*, 222-244 (vid. págs. 2, 12).
- Aghzal, M., Plaku, E., Stein, G. J., & Yao, Z. (2025). A survey on large language models for automated planning. *arXiv preprint arXiv:2502.12435* (vid. págs. 1, 2, 10).
- Aghzal, M., Plaku, E., & Yao, Z. (2023). Can large language models be good path planners? a benchmark and investigation on spatial-temporal reasoning. *arXiv preprint arXiv:2310.03249* (vid. pág. 11).
- Backus, J. W. (1959). The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. *ICIP Proceedings*, 125-132 (vid. pág. 18).
- Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., et al. (1963). Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1), 1-17 (vid. pág. 18).
- Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Podstawska, M., Gianinazzi, L., Gajda, J., Lehmann, T., Niewiadomski, H., Nyczyk, P., et al. (2024). Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16), 17682-17690 (vid. pág. 11).
- Bohnet, B., Nova, A., Parisi, A. T., Swersky, K., Goshvadi, K., Dai, H., Schuurmans, D., Fiedel, N., & Sedghi, H. (2024). Exploring and Benchmarking the Planning Capabilities of Large Language Models. *arXiv preprint arXiv:2406.13094* (vid. pág. 11).
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877-1901 (vid. págs. 1, 10, 14).
- Chen, Z., White, M., Mooney, R., Payani, A., Su, Y., & Sun, H. (2024). When is tree search useful for llm planning? it depends on the discriminator. *arXiv preprint arXiv:2402.10890* (vid. pág. 11).
- Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. (2023). Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240), 1-113 (vid. pág. 10).
- Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., et al. (2024). Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70), 1-53 (vid. pág. 10).
- Church, A. (1936). An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2), 345-363 (vid. pág. 16).
- Coles, A., Coles, A., Olaya, A. G., Jiménez, S., López, C. L., Sanner, S., & Yoon, S. (2012). A survey of the seventh international planning competition. *Ai Magazine*, 33(1), 83-88 (vid. pág. 9).

- Collins, K. M., Wong, C., Feng, J., Wei, M., & Tenenbaum, J. B. (2022). Structured, flexible, and robust: benchmarking and improving large language models towards more human-like behavior in out-of-distribution reasoning tasks. *arXiv preprint arXiv:2205.05718* (vid. pág. 12).
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 4171-4186 (vid. pág. 10).
- Dong, Q., Li, L., Dai, D., Zheng, C., Ma, J., Li, R., Xia, H., Xu, J., Wu, Z., Liu, T., et al. (2022). A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (vid. págs. 2, 13).
- Efron, B., & Tibshirani, R. J. (1994). *An introduction to the bootstrap*. Chapman; Hall/CRC. (Vid. pág. 66).
- Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4), 189-208 (vid. págs. 1, 8).
- Fine-Morris, M., Hsiao, V., Smith, L. N., Hiatt, L. M., & Roberts, M. (2024). Leveraging LLMs for Generating Document-Informed Hierarchical Planning Models: A Proposal. *AAAI 2025 Workshop LM4Plan* (vid. pág. 11).
- FireworksAI. (2025). Llama 4 Maverick Instruct (Basic) Model [Accedido el 12 de junio de 2025]. (Vid. págs. 46, 52, 66).
- Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, H., & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2, 1 (vid. pág. 1).
- Geng, S., Josifoski, M., Peyrard, M., & West, R. (2023). Grammar-constrained decoding for structured NLP tasks without finetuning. *arXiv preprint arXiv:2305.13971* (vid. págs. 1, 2, 16, 17).
- ggml-org. (2023). GBNF: Format for defining formal grammars to constrain model outputs in llama.cpp [Accedido el 15 de mayo de 2025]. (Vid. págs. 4, 18).
- Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: theory and practice*. Elsevier. (Vid. págs. 1, 8).
- Hao, S., Gu, Y., Ma, H., Hong, J. J., Wang, Z., Wang, D. Z., & Hu, Z. (2023). Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992* (vid. pág. 11).
- Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26, 191-246 (vid. págs. 1, 9).
- Howey, R., Long, D., & Fox, M. (2004). VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. *16th IEEE International Conference on Tools with Artificial Intelligence*, 294-301 (vid. pág. 9).
- Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., & Zhou, D. (2023). Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798* (vid. pág. 11).
- Huang, W., Abbeel, P., Pathak, D., & Mordatch, I. (2022). Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *International conference on machine learning*, 9118-9147 (vid. pág. 15).
- HuggingFace. (2025). sentence-transformers/all-mpnet-base-v2 [Accedido el 12 de junio de 2025]. <https://huggingface.co/sentence-transformers/all-mpnet-base-v2> (vid. pág. 48).
- Izquierdo-Badiola, S., Canal, G., Rizzo, C., & Alenyà, G. (2024). Plancollabnl: Leveraging large language models for adaptive plan generation in human-robot collaboration. *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 17344-17350 (vid. pág. 12).
- Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L. P., & Murthy, A. B. (2024). Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. *Forty-first International Conference on Machine Learning* (vid. pág. 2).

- Karpas, E., & Magazzeni, D. (2020). Automated planning for robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 3(1), 417-439 (vid. pág. 1).
- Kim, G., Baldi, P., & McAleer, S. (2023). Language models can solve computer tasks. *Advances in Neural Information Processing Systems*, 36, 39648-39677 (vid. pág. 10).
- Knuth, D. E. (1964). Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12), 735-736 (vid. pág. 18).
- Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35, 22199-22213 (vid. págs. 10, 14).
- Kovacs, D. L. (2011). BNF definition of PDDL 3.1. *Unpublished manuscript from the IPC-2011 website*, 15 (vid. pág. 28).
- Li, H. [Haoming], Chen, Z., Zhang, J., & Liu, F. (2024). LASP: Surveying the State-of-the-Art in Large Language Model-Assisted AI Planning. *arXiv preprint arXiv:2409.01806* (vid. pág. 1).
- Li, H. [Huayang], Su, Y., Cai, D., Wang, Y., & Liu, L. (2022). A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110* (vid. pág. 14).
- Li, T., Zhang, G., Do, Q. D., Yue, X., & Chen, W. (2024). Long-context llms struggle with long in-context learning. *arXiv preprint arXiv:2404.02060* (vid. págs. 10, 63).
- Li, W., Chen, C., & Varakantham, P. (2024). Unlocking Large Language Model's Planning Capabilities with Maximum Diversity Fine-tuning. *arXiv preprint arXiv:2406.10479* (vid. pág. 11).
- Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8, 293-321 (vid. págs. 16, 36).
- Liu, B., Jiang, Y., Zhang, X., Liu, Q., Zhang, S., Biswas, J., & Stone, P. (2023). Llm+ p: Empowering large language models with optimal planning proficiency. *arXiv preprint arXiv:2304.11477* (vid. págs. 2, 5, 12, 24, 25).
- Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, 55(9), 1-35 (vid. pág. 13).
- Liu, Z., Bahety, A., & Song, S. (2023). Reflect: Summarizing robot experiences for failure explanation and correction. *arXiv preprint arXiv:2306.15724* (vid. pág. 15).
- Loula, J., LeBrun, B., Du, L., Lipkin, B., Pasti, C., Grand, G., Liu, T., Emara, Y., Freedman, M., Eisner, J., et al. (2025). Syntactic and semantic control of large language models via sequential monte carlo. *arXiv preprint arXiv:2504.13139* (vid. págs. 19, 61).
- Lyu, Q., Havaldar, S., Stein, A., Zhang, L., Rao, D., Wong, E., Apidianaki, M., & Callison-Burch, C. (2023). Faithful chain-of-thought reasoning. *The 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (IJCNLP-AACL 2023)* (vid. pág. 12).
- Maas, C., Wheeler, S., Billington, S., et al. (2023). To infinity and beyond: Show-1 and showrunner agents in multi-agent simulations. *To infinity and beyond: Show-1 and showrunner agents in multi-agent simulations* (vid. pág. 15).
- Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. (2023). Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 46534-46594 (vid. pág. 10).
- Marrella, A. (2019). Automated planning for business process management. *Journal on data semantics*, 8(2), 79-98 (vid. pág. 1).
- Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *Proceedings of the 36th annual acm symposium on user interface software and technology*, 1-22 (vid. pág. 15).

- Pednault, E. (1987). Formulating multiagent, dynamic-world problems in the classical planning framework. *Reasoning about actions and plans*, 47-82 (vid. pág. 8).
- Peter, N., & Intelligence, R. S. A. (2021). *A Modern Approach*. Pearson Education, USA. (Vid. págs. 7, 8).
- Petroni, F., Rocktäschel, T., Lewis, P., Bakhtin, A., Wu, Y., Miller, A. H., & Riedel, S. (2019). Language models as knowledge bases? *arXiv preprint arXiv:1909.01066* (vid. pág. 14).
- Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., Liu, Z., & Sun, M. (2023). Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 6(3) (vid. pág. 15).
- Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140), 1-67 (vid. pág. 10).
- Reimers, N., & Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (vid. pág. 14).
- Richter, S., Westphal, M., & Helmert, M. (2011). LAMA 2008 and 2011. *International Planning Competition*, 117-124 (vid. pág. 1).
- Rubin, O., Herzig, J., & Berant, J. (2021). Learning to retrieve prompts for in-context learning. *arXiv preprint arXiv:2112.08633* (vid. pág. 14).
- Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (vid. pág. 16).
- Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 8634-8652 (vid. págs. 10, 15).
- Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., & Garg, A. (2023). Progprompt: Generating situated robot task plans using large language models. *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 11523-11530 (vid. pág. 10).
- Singh, S., Swaminathan, K., Arora, R., Singh, R., Datta, A., Das, D., Banerjee, S., Sridharan, M., & Krishna, M. (2024). Anticipate & Collab: Data-driven Task Anticipation and Knowledge-driven Planning for Human-robot Collaboration. *arXiv preprint arXiv:2404.03587* (vid. pág. 12).
- Singhal, A., et al. (2001). Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4), 35-43 (vid. pág. 14).
- Smirnov, P., Joublin, F., Ceravola, A., & Gienger, M. (2024). Generating consistent PDDL domains with Large Language Models. *arXiv preprint arXiv:2404.07751* (vid. pág. 11).
- Stechly, K., Valmeeckam, K., & Kambhampati, S. (2024a). Chain of thoughtlessness? an analysis of cot in planning. *The Thirty-eighth Annual Conference on Neural Information Processing Systems* (vid. pág. 10).
- Stechly, K., Valmeeckam, K., & Kambhampati, S. (2024b). On the self-verification limitations of large language models on reasoning and planning tasks. *arXiv preprint arXiv:2402.08115* (vid. pág. 11).
- Sun, H., Zhuang, Y., Kong, L., Dai, B., & Zhang, C. (2023). Adapanner: Adaptive planning from feedback with language models. *Advances in neural information processing systems*, 36, 58202-58245 (vid. pág. 10).
- Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction* (Vol. 1). MIT press Cambridge. (Vid. pág. 15).
- Tantakoun, M., Zhu, X., & Muise, C. (2025). LLMs as Planning Modelers: A Survey for Leveraging Large Language Models to Construct Automated Planning Models. *arXiv preprint arXiv:2503.18971* (vid. págs. 2, 7, 8, 11, 12).
- Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., & Hashimoto, T. B. (2023). Stanford alpaca: An instruction-following llama model. (Vid. pág. 10).

- Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. (2022). Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239* (vid. pág. 10).
- Toman, P. (2023, 25 de agosto). *Why LLM Outputs Aren't Always Reproducible: Sources of Non-Determinism* [Blog personal, accedido el 12 de junio de 2025]. <https://www.pamelatoman.net/blog/2023/08/nondeterminism-in-llms/> (vid. pág. 47).
- Turing, A. M., et al. (1936). On computable numbers, with an application to the Entscheidungsproblem. *J. of Math.*, 58(345-363), 5 (vid. pág. 16).
- Verma, M., Bhambri, S., & Kambhampati, S. (2024). On the brittle foundations of react prompting for agentic large language models. *arXiv preprint arXiv:2405.13966* (vid. pág. 11).
- Wang, B., Wang, Z., Wang, X., Cao, Y., A Saurous, R., & Kim, Y. (2023). Grammar prompting for domain-specific language generation with large language models. *Advances in Neural Information Processing Systems*, 36, 65030-65055 (vid. pág. 19).
- Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., & Anandkumar, A. (2023). Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291* (vid. pág. 76).
- Wang, L. [Lei], Xu, W., Lan, Y., Hu, Z., Lan, Y., Lee, R. K.-W., & Lim, E.-P. (2023). Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091* (vid. pág. 10).
- Wang, L. [Liang], Yang, N., & Wei, F. (2023). Learning to retrieve in-context examples for large language models. *arXiv preprint arXiv:2307.07164* (vid. págs. 13, 14).
- Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., & Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (vid. pág. 11).
- Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, 8, 279-292 (vid. págs. 15, 36).
- Wei, H., Zhang, Z., He, S., Xia, T., Pan, S., & Liu, F. (2025). PlanGenLLMs: A Modern Survey of LLM Planning Capabilities. *arXiv preprint arXiv:2502.11221* (vid. págs. 2, 4).
- Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2021). Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (vid. pág. 10).
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35, 24824-24837 (vid. págs. 10, 14, 15, 59).
- Weinmeister, R. (2024, 14 de marzo). *Is a Zero Temperature Deterministic?* [Blog en Medium, accedido el 12 de junio de 2025]. <https://medium.com/google-cloud/is-a-zero-temperature-deterministic-c4a7faef4d20> (vid. pág. 47).
- Xie, Y., Yu, C., Zhu, T., Bai, J., Gong, Z., & Soh, H. (2023). Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128* (vid. págs. 10, 12).
- Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., & Narasimhan, K. (2023). Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36, 11809-11822 (vid. págs. 11, 15, 59).
- Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). React: Synergizing reasoning and acting in language models. *International Conference on Learning Representations (ICLR)* (vid. págs. 10, 15, 59).
- Yue, Y., Kang, B., Ma, X., Huang, G., Song, S., & Yan, S. (2023). Offline prioritized experience replay. *arXiv preprint arXiv:2306.05412*, 6 (vid. pág. 16).

- Zeng, A., Liu, M., Lu, R., Wang, B., Liu, X., Dong, Y., & Tang, J. (2023). Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823* (vid. págs. 2, 12).
- Zhang, L., Jansen, P., Zhang, T., Clark, P., Callison-Burch, C., & Tandon, N. (2024). Pddlego: Iterative planning in textual environments. *arXiv preprint arXiv:2405.19793* (vid. pág. 11).
- Zhang, Z., Zhang, A., Li, M., & Smola, A. (2022). Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (vid. pág. 14).
- Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y.-J., & Huang, G. (2024). Expel: Llm agents are experiential learners. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17), 19632-19642 (vid. págs. 1, 2, 10, 13, 36, 38, 39, 41, 42).
- Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., et al. (2022). Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625* (vid. pág. 10).
- Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., & He, Q. (2020). A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1), 43-76 (vid. pág. 75).
- Zuo, M., Velez, F. P., Li, X., Littman, M. L., & Bach, S. H. (2024). Planetarium: A rigorous benchmark for translating text to structured planning languages. *arXiv preprint arXiv:2407.03321* (vid. págs. 1, 2, 12, 19, 21).

Anexos

Agentes planificadores

El esquema general del *prompt* empleado para estos agentes fue el siguiente:

Prompt genérico del agente planificador

You are an advanced Planning AI Agent. You are given the description of a planning domain and the available actions, and the natural language descriptions of problems in this domain, and for each you provide an optimal plan to solve the problem.

Domain: <Nombre del dominio>

<Descripción corta del dominio en lenguaje natural>

<Descripción de las acciones disponibles en el dominio>

Task:

You will be given natural language descriptions of planning problems in this domain. Provide an optimal plan, in the way of a sequence of actions, to solve the problem.

<Descripción y sintaxis de la salida>

<Ejemplos de FSP>

New problem:

<Descripción del problema en lenguaje natural>

Plan:

A continuación se describen los componentes dinámicos del *prompt* utilizado por los agentes planificadores, cuya inclusión depende del dominio, del problema específico y de la activación del módulo *FSP*.

- <Nombre del dominio>, <Descripción corta del dominio en lenguaje natural>, <Descripción de las acciones disponibles en el dominio> y <Descripción y sintaxis de la salida> son extraídos automáticamente de una base estructurada de recursos por dominio, construida a partir del conocimiento experto proporcionado por un humano. Esta base se encuentra documentada en el Anexo correspondiente.
- <Descripción del problema en lenguaje natural> corresponde a la entrada específica del problema a

resolver, tal como aparece en el conjunto de datos del *benchmark Planetarium*.

- La sección *<Ejemplos de FSP>* se incluye condicionalmente, en función de si el módulo *FSP* se encuentra activado. En caso afirmativo, se incorpora un ejemplo representativo del dominio, seleccionado desde la misma base de recursos y estructurado bajo el siguiente formato:

Ejemplo incluido con *FSP*

Example #1:

Problem:

<Descripción del problema de ejemplo en lenguaje natural>

Plan:

<Plan que resuelve el problema de ejemplo>

Los componentes *<Descripción del problema de ejemplo en lenguaje natural>* y *<Plan que resuelve el problema de ejemplo>* provienen directamente del recurso *FSP* proporcionado por el experto humano para el dominio correspondiente.

Agentes modeladores originales

El *prompt* empleado por estos agentes depende de la activación o no del módulo *FSP*. A continuación se detallan ambas variantes.

Sin *FSP*

Cuando el módulo *FSP* no está activado, el agente recibe como contexto una descripción en lenguaje natural del dominio y de las acciones disponibles, seguida por la descripción del problema a resolver. El *prompt* adoptado es el siguiente:

Prompt del agente modelador sin *FSP*

<Descripción del dominio y las acciones>

Now consider a planning problem. The problem description is:

<Descripción del problema en lenguaje natural>

Provide me with the problem PDDL file that describes the planning problem directly without further explanations?

La sección *<Descripción del dominio y las acciones>* es provista por el experto humano y se extrae de la base estructurada de recursos por dominio. La sección *<Descripción del problema en lenguaje natural>* corresponde a la entrada específica del problema, obtenida directamente del dataset *Planetarium*.

Con *FSP*

En el caso en que el módulo *FSP* esté activado, el agente recibe como contexto un ejemplo completo, consistente en una descripción de un problema representativo y un modelo *PDDL* correspondiente. A

continuación, se presenta el nuevo problema a resolver. El *prompt* en esta modalidad tiene la siguiente estructura:

Prompt* del agente modelador con *FSP

I want you to solve planning problems. An example planning problem is:
<Descripción del problema de ejemplo en lenguaje natural>

The problem PDDL file to this problem is:
<Modelo PDDL del problema de ejemplo>

Now I have a new planning problem and its description is:
<Descripción del problema en lenguaje natural>

Provide me with the problem PDDL file that describes the planning problem directly without further explanations?

Tanto la *<Descripción del problema de ejemplo en lenguaje natural>* como el *<Modelo PDDL del problema de ejemplo>* provienen del recurso *FSP* elaborado por el experto humano para el dominio correspondiente. Este ejemplo fue adaptado a partir de las versiones originales presentadas en *LLM+P*, reformuladas para ajustarse a las variantes de dominio del *benchmark Planetarium*.

Algoritmos de los agentes modeladores propuestos

Se presentan los algoritmos que participan en el uso de los agentes modeladores propuestos:

Algoritmo 4.1: \mathcal{A}_{mod} – Instanciación del agente modelador con módulos activos

Input : Conjunto de módulos activos $\mathcal{A}_{\text{mod}}.M \subseteq \{\text{raz, obj, fsp, rag, com, ins, ref, gcd, daps}\}$

Output: Agente instanciado \mathcal{A}_{mod}

- 1 Inicializar modelo de generación de problemas en PDDL: LLM_{genPDDL} ;
- 2 **si** $\text{obj} \in \mathcal{A}_{\text{mod}}.M$ **entonces**
- 3 | Cargar gramática no tipada: $\mathcal{A}_{\text{mod}}.\mathcal{G}_{\text{extObj}}^{\text{no-tipada}}$;
- 4 **fin**
- 5 **si** $\text{com} \in \mathcal{A}_{\text{mod}}.M$ **entonces**
- 6 | Definir directriz de comentarios: $\mathcal{A}_{\text{mod}}.\kappa \leftarrow \text{"comentarios breves en :init y :goal ...";}$
- 7 **fin**
- 8 **en otro caso**
- 9 | $\mathcal{A}_{\text{mod}}.\kappa \leftarrow \text{"";}$
- 10 **fin**
- 11 **si** $\text{ins} \in \mathcal{A}_{\text{mod}}.M$ **entonces**
- 12 | Cargar conjunto de insights: $\mathcal{A}_{\text{mod}}.\iota \leftarrow \text{load_insights}();$
- 13 **fin**
- 14 **en otro caso**
- 15 | $\mathcal{A}_{\text{mod}}.\iota \leftarrow \emptyset;$
- 16 **fin**
- 17 Inicializar conjunto de ejemplos FSP: $\mathcal{A}_{\text{mod}}.\phi \leftarrow \emptyset;$
- 18 Inicializar lista de intentos anteriores con *feedback* y reflexión: $\mathcal{A}_{\text{mod}}.\tau \leftarrow \emptyset;$
- 19 Inicializar recuperador de ejemplos: $\mathcal{A}_{\text{mod}}.\mathcal{R};$
- 20 **devolver** $\mathcal{A}_{\text{mod}};$

Algoritmo 4.2: \mathcal{A}_{mod} – Asignación de tarea

Input : Identificador de problema P_{id} , dominio D_{nomb} , descripción en lenguaje natural P_{desc}

Output: Agente \mathcal{A}_{mod} actualizado con contexto de tarea

```
1 si  $\mathcal{A}_{\text{mod}}.D_{\text{nomb}} \neq D_{\text{nomb}}$  entonces
2   Cargar descripción del dominio:  $\mathcal{A}_{\text{mod}}.D_{\text{desc}} \leftarrow \text{get\_domain\_description}(D_{\text{nomb}});$ 
3   Cargar PDDL completo del dominio:  $\mathcal{A}_{\text{mod}}.D_{\text{PDDL}} \leftarrow \text{get\_domain_pddl}(D_{\text{nomb}});$ 
4   si  $D_{\text{nomb}} = "floor-tile"$  entonces
5     | Cargar PDDL sin acciones:  $\mathcal{A}_{\text{mod}}.D_{\text{PDDL}}^- \leftarrow \text{get\_domain_pddl\_without\_actions}(D_{\text{nomb}});$ 
6   fin
7   Cargar predicados del dominio:  $\mathcal{A}_{\text{mod}}.D_{\text{predicados}} \leftarrow \text{get\_domain\_predicates}(D_{\text{nomb}});$ 
8   Cargar requisitos del dominio:  $\mathcal{A}_{\text{mod}}.D_{\text{req}} \leftarrow \text{get\_domain\_requirements}(D_{\text{nomb}});$ 
9   Cargar tipos del dominio:  $\mathcal{A}_{\text{mod}}.D_{\text{tipos}} \leftarrow \text{get\_domain\_types}(D_{\text{nomb}});$ 
10  si  $\text{obj} \in \mathcal{A}_{\text{mod}}.M$  entonces
11    | si  $\mathcal{A}_{\text{mod}}.D_{\text{tipos}} \neq \emptyset$  entonces
12      |   | Cargar gramática tipada:  $\mathcal{A}_{\text{mod}}.\mathcal{G}_{\text{extObj}} \leftarrow \mathcal{G}_{\text{extObj}}^{\text{tipada}}(D_{\text{tipos}});$ 
13    fin
14    | en otro caso
15    |   | Usar gramática no tipada:  $\mathcal{A}_{\text{mod}}.\mathcal{G}_{\text{extObj}} \leftarrow \mathcal{A}_{\text{mod}}.\mathcal{G}_{\text{extObj}}^{\text{no-tipada}};$ 
16    fin
17  fin
18  si  $\text{fsp} \in \mathcal{A}_{\text{mod}}.M$  y  $\text{rag} \notin \mathcal{A}_{\text{mod}}.M$  entonces
19    | Cargar ejemplo manual FSP:  $\mathcal{A}_{\text{mod}}.\phi \leftarrow \text{get\_fsp\_example}(D_{\text{nomb}});$ 
20  fin
21 fin
22 Asignar identificador del problema:  $\mathcal{A}_{\text{mod}}.P_{\text{id}} \leftarrow P_{\text{id}};$ 
23 Asignar descripción del problema:  $\mathcal{A}_{\text{mod}}.P_{\text{desc}} \leftarrow P_{\text{desc}};$ 
24 si  $\text{rag} \in \mathcal{A}_{\text{mod}}.M$  entonces
25   | Recuperar ejemplos FSP relevantes:  $\mathcal{A}_{\text{mod}}.\phi \leftarrow \mathcal{A}_{\text{mod}}.\mathcal{R}.\text{get\_top\_similar}(P_{\text{id}}, k);$ 
26 fin
```

Estos algoritmos, concebidos como métodos del agente, permiten su instanciación con el enfoque modular propuesto, y la asignación del problema de planificación a modelar. La instanciación requiere la selección de las mejoras modulares a aplicar, y carga los recursos que podría necesitar el agente, de forma preventiva. Un proceso similar hace la asignación del problema, que completa toda la información requerida, al incluir la descripción en lenguaje natural del problema y el dominio de planificación en el que se desarrolla. Finalmente, con la información cargada, se puede proceder a la modelación del problema:

Algoritmo 4.3: \mathcal{A}_{mod} – Modelación del problema

Input : Tarea ya asignada al agente \mathcal{A}_{mod}

Output: Modelo *PDDL* del problema P_π

```

1 si raz ∈  $\mathcal{A}_{\text{mod}}.M$  entonces
2   | Construir prompts para Razonamiento:  $\Pi_{\text{raz\_sist}}, \Pi_{\text{raz\_usr}}$ ;
3   | Obtener razonamiento:  $\mathcal{A}_{\text{mod}}.\rho \leftarrow LLM_{\text{raz}}(\Pi_{\text{raz\_sist}}, \Pi_{\text{raz\_usr}})$ ;
4 fin
5 si obj ∈  $\mathcal{A}_{\text{mod}}.M$  entonces
6   | Construir prompts para Extracción de objetos:  $\Pi_{\text{extObj\_sist}}, \Pi_{\text{extObj\_usr}}$ ;
7   | Obtener objetos:  $\mathcal{A}_{\text{mod}}.\omega \leftarrow LLM_{\text{extObj}}(\Pi_{\text{extObj\_sist}}, \Pi_{\text{extObj\_usr}}, \mathcal{A}_{\text{mod}}.\mathcal{G}_{\text{extObj}})$ ;
8 fin
9 si ref ∈  $\mathcal{A}_{\text{mod}}.M$  y  $\mathcal{A}_{\text{mod}}.\tau \neq \emptyset$  entonces
10  | Recuperar intento previo con feedback y reflexión:  $\tau \leftarrow \mathcal{A}_{\text{mod}}.\tau[-1]$ ;
11 fin
12 en otro caso
13  |  $\tau \leftarrow \emptyset$ ;
14 fin
15 si gcd ∈  $\mathcal{A}_{\text{mod}}.M$  entonces
16   | si daps ∈  $\mathcal{A}_{\text{mod}}.M$  entonces
17     |   | Construir gramática
18     |   |   |  $\mathcal{G}_{\text{genPDDL}} \leftarrow \mathcal{G}_{\text{genPDDL\_DAPS}}(\mathcal{A}_{\text{mod}}.\omega, \mathcal{A}_{\text{mod}}.D_{\text{predicados}}, \mathcal{A}_{\text{mod}}.D_{\text{tipos}}, \mathcal{A}_{\text{mod}}.\kappa)$ ;
19   |   fin
20   | en otro caso
21   |   | Construir gramática  $\mathcal{G}_{\text{genPDDL}} \leftarrow \mathcal{G}_{\text{genPDDL\_general}}(\mathcal{A}_{\text{mod}}.\kappa)$ ;
22   |   fin
23 fin
24 en otro caso
25  |  $\mathcal{G}_{\text{genPDDL}} \leftarrow \emptyset$ ;
26 Construir prompt del sistema para Generación de PDDL;;
27  $\Pi_{\text{genPDDL\_sist}}^{\text{req}} \leftarrow \{\mathcal{A}_{\text{mod}}.D_{\text{nombre}}, \mathcal{A}_{\text{mod}}.D_{\text{desc}}, \mathcal{A}_{\text{mod}}.D_{\text{PDDL}}, \mathcal{A}_{\text{mod}}.D_{\text{req}}\}$ ;
28  $\Pi_{\text{genPDDL\_sist}}^{\text{opt}} \leftarrow \{\mathcal{A}_{\text{mod}}.\kappa, \mathcal{A}_{\text{mod}}.\phi, \mathcal{A}_{\text{mod}}.\iota\}$ ;
29  $\Pi_{\text{genPDDL\_sist}} \leftarrow \Pi_{\text{genPDDL\_sist}}^{\text{req}} \cup \Pi_{\text{genPDDL\_sist}}^{\text{opt}}$ ;
30 Construir prompt del usuario para Generación de PDDL;;
31  $\Pi_{\text{genPDDL\_usr}}^{\text{req}} \leftarrow \{\mathcal{A}_{\text{mod}}.P_{\text{desc}}\}$ ;
32  $\Pi_{\text{genPDDL\_usr}}^{\text{opt}} \leftarrow \{\mathcal{A}_{\text{mod}}.\tau, \mathcal{A}_{\text{mod}}.\rho, \mathcal{A}_{\text{mod}}.\omega\}$ ;
33  $\Pi_{\text{genPDDL\_usr}} \leftarrow \Pi_{\text{genPDDL\_usr}}^{\text{req}} \cup \Pi_{\text{genPDDL\_usr}}^{\text{opt}}$ ;
34 Llamar al modelo:  $\mathcal{P}_\pi \leftarrow LLM_{\text{genPDDL}}(\Pi_{\text{genPDDL\_sist}}, \Pi_{\text{genPDDL\_usr}}, \mathcal{A}_{\text{mod}}.\mathcal{G}_{\text{genPDDL}})$ ;
35 devolver  $\mathcal{P}_\pi$ ;

```

Generación de *PDDL*

Con el objetivo de garantizar la modularidad de los métodos implementados y permitir una comparación estandarizada entre las diferentes configuraciones del agente modelador, se adoptó una estructura común de

prompt. A esta base se le añaden de forma condicional, diferentes fragmentos que corresponden a módulos específicos activados en cada variante del agente.

El *prompt* del agente modelador propuesto se estructuró de la siguiente manera:

Prompt base del agente modelador propuesto

You are an advanced Planning Modeler AI Agent specialized in PDDL generation.

You are given the description and PDDL code of a planning domain and the natural language descriptions of problems in this domain, and for each you provide the PDDL code of the problem.

Domain: <Nombre del dominio>

<Descripción corta del dominio en lenguaje natural>

Domain PDDL:

<Modelo PDDL del dominio>

Task:

You will be given natural language descriptions of planning problems in this domain.

Provide the PDDL code (that conforms to the grammar of the <Requerimientos del subconjunto de PDDL> subset of PDDL) of this problem, without further explanation. <Sugerencia sobre uso de comentarios>

<Ejemplos de FSP>

<Insights>

New problem:

<Descripción del problema en lenguaje natural>

<Información y reflexión sobre intento anterior>

<Razonamiento previo sobre el problema>

<Objetos determinados a utilizar>

Problem PDDL:

Cada sección entre corchetes angulares es determinada dinámicamente en función de las características del dominio, del problema específico y de los módulos activados en el agente. Esta construcción jerárquica del *prompt* permitió evaluar de forma aislada y combinada el impacto de técnicas como el uso de *FSP*, razonamiento estructurado, utilización de *insights* extraídos previamente, la determinación anticipada de objetos relevantes para el modelado del problema, etc.

A continuación, se describen los componentes que pueden ser incorporados dinámicamente:

- <Nombre del dominio>, <Descripción corta del dominio en lenguaje natural>, y <Modelo PDDL del dominio>; estos elementos son obtenidos de la base de recursos del dominio, la cual es proporcionada por el experto humano. Dicha base se incluye como material de referencia en el Anexo correspondiente.

- <Requerimientos del subconjunto de PDDL>: esta sección del *prompt* especifica los requerimientos del subconjunto de *PDDL* a utilizar. En los casos en que el dominio incluía objetos tipados, se indica que la generación debe ajustarse al subconjunto *STRIPS + :typing*; de lo contrario, se indica únicamente *STRIPS*.
- <Sugerencia sobre uso de comentarios>: este fragmento es incluido solo si se activa el módulo *Comments*. El mensaje añadido es el siguiente:

"Before each predicate in the :init and :goal sections you can use at most 1 short one-line comment to describe the start of a relevant section of definitions."

- <Ejemplos de FSP>: este bloque se incorpora únicamente cuando el módulo *FSP* está activado. La composición y origen de los ejemplos depende de los módulos activos. Si el módulo *RAG* no está habilitado, el ejemplo se selecciona directamente de la base de recursos del dominio proporcionada por el experto humano. En cambio, si *RAG* está activo, se utiliza un componente *Retriever* para seleccionar los k ejemplos más similares al problema a resolver, a partir del *Experience Pool* (conjunto de soluciones correctas generadas durante la fase de entrenamiento, incluyendo los ejemplos originales).

Para cada ejemplo e_i ($1 \leq i \leq k$), se añade al *prompt* el siguiente bloque:

***Prompt* base del agente modelador propuesto**

Example #i:

Problem:

<Descripción del problema de ejemplo en lenguaje natural>

<Razonamiento sobre el problema de ejemplo>

<Objetos determinados a utilizar>

Problem PDDL:

<Modelo PDDL del problema de ejemplo>

Los campos <Razonamiento sobre el problema de ejemplo> y <Objetos determinados a utilizar> se incluyen únicamente si los módulos *Reasoning* y *Objects Extraction* están activados, respectivamente. Estos mantienen el mismo formato que en el *prompt* base del agente.

- <Insights>: este segmento se añade si se activa el módulo *Insights*. Su objetivo es proporcionar conocimiento previamente extraído y estructurado a partir de la experiencia del agente durante la fase de entrenamiento. El fragmento incorporado al *prompt* tiene la siguiente forma:

Bloque de *insights*

This are some insights you have gathered through experience, to guide your response:

- DOMAIN_KNOWLEDGE (insights related to world knowledge of the given planning domain):

...

- DOMAIN_RULES (domain-specific modeling rules, tips, or best practices):

...
- **GENERAL** (general modeling principles applicable across domains):
...

Make sure to FOLLOW CLOSELY this insights, specially the DOMAIN_RULES.

Si no existen *insights* asociados a alguno de los tres tipos (*DOMAIN_KNOWLEDGE*, *DOMAIN_RULES*, o *GENERAL*), dicho apartado se omite del *prompt*.

El bloque final del *prompt* está destinado al problema de planificación que se desea resolver. La construcción de esta sección también es modular y depende de los componentes activos en el agente modelador.

- <*Descripción del problema en lenguaje natural*>: este fragmento representa la formulación textual del problema de planificación a resolver, tal como fue proporcionada en el conjunto de datos de entrada. Siempre se incluye, y constituye el núcleo del ejemplo objetivo.
- <*Información y reflexión sobre intento anterior*>: este bloque se incorpora únicamente en los casos en que el agente pertenece al subconjunto denominado *experiencial*, donde se permite la re-ejecución de múltiples intentos sobre un mismo problema. Este tipo de agentes se caracteriza por incorporar mecanismos de reflexión sobre fallos anteriores, y su prompt particular se presenta más adelante. En este caso, cuando el módulo *Reflection* está activo y no se trata del primer intento sobre un problema, se incluye la siguiente estructura en el *prompt*:

Bloque de reflexión sobre intento anterior

You have tried the problem before, unsuccessfully. This was your answer:

<*Razonamiento sobre el problema de ejemplo*>

<*Objetos determinados a utilizar*>

Generated Problem PDDL:

<*Modelo PDDL del problema generado en el intento anterior*>

Evaluation feedback:

<*Feedback sobre el fallo en la evaluación*>

Reflection on the previous trial:

<*Reflexión sobre el fallo*>

Try again, taking into account the previous trial.

New trial:

En este bloque, los fragmentos <*Razonamiento sobre el problema de ejemplo*> y <*Objetos determinados a utilizar*> son añadidos solo si se encuentran activos los módulos *Reasoning* y *Objects Extraction*, respectivamente, siguiendo el formato descrito previamente.

El componente *<Feedback sobre el fallo en la evaluación>* es generado automáticamente a partir de las métricas de evaluación y otros resultados parciales del intento anterior. Este se expresa en lenguaje natural, y su proceso de construcción se detalla en secciones posteriores de este capítulo.

Finalmente, la sección *<Reflexión sobre el fallo>* es generada por un sub-agente especializado en reflexión sobre errores en modelado, el cual se presenta también en detalle en secciones subsiguientes.

- *<Razonamiento previo sobre el problema>* y *<Objetos determinados a utilizar>*: estos bloques se incluyen en el *prompt* principal únicamente si están activados los módulos *Reasoning* y *Objects Extraction*, respectivamente.

- El componente *<Razonamiento previo sobre el problema>* es generado por un sub-agente especializado en análisis y razonamiento sobre la modelación de problemas de planificación, que busca desambiguar y predecir las posibles estructuras del problema en base al conocimiento del dominio, la descripción del problema y los principios generales de modelado. Este agente se describe en detalle más adelante.
- El componente *<Objetos determinados a utilizar>* corresponde a una lista explícita de objetos relevantes que deben incluirse en el modelo *PDDL* generado. Esta información es proporcionada por un sub-agente especializado en extracción de objetos, cuyo funcionamiento también se analiza en secciones posteriores.

Razonamiento

El *prompt* utilizado para esta tarea se muestra a continuación:

Prompt del agente de razonamiento

You are an advanced Planning Modeler AI Agent specialized in reasoning. You are given the description and PDDL code of a planning domain and the natural language descriptions of problems in this domain, and for each you provide a structured reasoning about the problem for its translation to PDDL.

Domain: *<Nombre del dominio>*

<Descripción corta del dominio en lenguaje natural>

Domain PDDL:

<Modelo PDDL del dominio>

Task:

You will be given natural language descriptions of planning problems in this domain.

Reason step by step to resolve any ambiguities or missing details, to help to improve the semantic correctness of a posterior problem PDDL generation by another Planning Modeler Agent.

Write your reasoning as exactly 3 paragraphs of no more than 100 words each. In them you should cover, in order:

1. **Objects:** list every object by name.

2. **Initial state:** describe the initial state, and any assumptions you make to fill in missing info.

3. Goal state: describe the goal state, and any assumptions you make to fill in missing info.

Fully specify the subgoals and initial state as detailed as you can. Don't reason about the planning itself, just the PDDL. Don't extend beyond 3 short paragraphs.

<Ejemplos de FSP>

<Insights>

New problem:

<Descripción del problema en lenguaje natural>

<Información y reflexión sobre intento anterior>

Reasoning:

Al igual que en el *prompt* del agente generador de *PDDL*, los fragmentos representados entre signos <...> son determinados dinámicamente en función del problema a resolver y de los módulos activos en la configuración del agente. El contenido de estos elementos se obtiene y estructura del mismo modo descrito previamente en las secciones correspondientes.

Extracción de Objetos

El siguiente *prompt* fue utilizado para inducir al *LLM* a realizar la tarea de extracción de objetos:

Prompt del agente de extracción de objetos

You are an advanced Planning Modeler AI Agent specialized in objects extraction.

You are given the description and PDDL code of a planning domain and the natural language descriptions of problems in this domain, and for each you provide a JSON of all the objects in the PDDL problem instance.

Domain: *<Nombre del dominio>*

<Descripción corta del dominio en lenguaje natural>

Domain PDDL:

<Modelo PDDL del dominio>

Task:

You will be given natural language descriptions of planning problems in this domain.

Provide a JSON of all the objects in the PDDL problem instance *<Indicación de tipado>*.

<Ejemplos de FSP>

New problem:

<Descripción del problema en lenguaje natural>

<Información y reflexión sobre intento anterior>

<Razonamiento previo sobre el problema>

Objects to use:

Al igual que en los *prompts* anteriores, los elementos representados entre signos <...> son agregados o modificados en función del problema específico y de los módulos activados en el agente. A continuación se describe el significado y la lógica de inclusión de los elementos particulares de este *prompt*:

- <Información y reflexión sobre intento anterior> se incluye únicamente si el módulo *Reflection* está activo y el agente ha realizado al menos un intento anterior fallido sobre el mismo problema. Sin embargo, este fragmento no se incluye si también se encuentra activo el módulo de razonamiento, ya que la reflexión se utiliza exclusivamente en los *prompts* para generación o reintentos.
- <Razonamiento previo sobre el problema> se añade si está habilitado el módulo *Reasoning*, y su contenido se genera a partir del subagente especializado en razonamiento descrito en la sección anterior.
- <Indicación de tipado> es una indicación opcional que se añade al final de la instrucción principal si el dominio modelado utiliza objetos tipados. En este caso, se inserta la frase “, separated by type”, con el fin de que el modelo agrupe los objetos extraídos por tipo en la estructura *JSON*. Si el dominio no está tipado, este fragmento se omite.

Reflexión

A continuación se muestra el *prompt* empleado:

Prompt del agente de reflexión

You are an advanced Planning Modeler AI Agent, capable of reflecting on failed attempts of planning problems within a given domain. In each case, a modeler agent was provided with the domain description, its corresponding PDDL file, and a natural language description of the problem, then attempted to generate the Problem PDDL file.

Domain: <Nombre del dominio>

<Descripción corta del dominio en lenguaje natural>

Domain PDDL:

<Modelo PDDL del dominio>

Task:

You will be presented with a natural language description of a planning problem, along with a failed attempt to generate its corresponding PDDL representation. You will also receive a description of

the errors in the attempted PDDL.

The generated problem PDDL may suffer from one or more of the following issues:

- **Incorrect Object Count:** The number of declared objects is inaccurate. If typed, the number of objects of specific types may be wrong.
- **Syntactical Errors:** The PDDL is not parseable due to violations of the *<Requerimientos>* subset of PDDL syntax, use of undefined objects, or other syntactical mistakes.
- **Unsolvable Problem:** No plan can achieve the goal state from the given initial state due to one of the following reasons:
 - Impossible Initial/Goal State: The initial and/or goal state contain contradictory predicates.
 - Path Infeasibility: Even if both initial and goal states are individually consistent, there is no sequence of actions that can transform the initial state into the goal state.
- **Semantically Incorrect:** The PDDL representation is not faithful to the natural language description. This is caused by inaccuracies in the initial state, the goal state, or both. While deemed incorrect, elements of the initial and/or goal states may still be partially correct but lack necessary predicates for a complete specification.

Unparseable PDDL cannot be solvable or semantically correct. Similarly, unsolvable PDDL cannot be semantically correct. In these cases, no specific feedback will be provided on later issues, but you may be able to identify/prevent them.

Provide a single paragraph reflecting on the potential causes of the failed attempt and propose corrections. Carefully consider whether modifications are necessary for the initial state, goal state, or both.

Problem:

<Descripción del problema en lenguaje natural>

<Razonamiento sobre el problema realizado en el intento anterior>

<Objetos determinados a utilizar en el intento anterior>

Generated problem PDDL:

<Modelo PDDL del problema generado en el intento anterior>

Evaluation feedback:

<Feedback de evaluación sobre el fallo del intento anterior>

Your reflection:

Al igual que en las descripciones anteriores de agentes, los elementos encerrados entre símbolos <> fueron construidos o incluidos en dependencia del dominio y problema particular, así como del resultado del intento anterior y la retroalimentación generada. Particularmente:

- <Razonamiento sobre el problema realizado en el intento anterior> y <Objetos determinados a utilizar en el intento anterior> son incluidos solamente si el agente modelador responsable del intento previo realizó las fases de razonamiento y extracción de objetos, respectivamente.
- <Modelo PDDL del problema generado en el intento anterior> corresponde al resultado concreto del intento previo del agente modelador, modelado en *PDDL*.
- <Feedback de evaluación sobre el fallo del intento anterior> es la descripción en lenguaje natural del fallo generada en la evaluación, donde se indican explícitamente las causas del fallo, ya fueran sintácticas o semánticas.

Agente de *Insights*

Para permitir que el agente de *insights* opere sobre conjuntos complejos y estructurados de conocimiento, se diseñó un *prompt* parametrizable y flexible que guía el comportamiento del modelo de lenguaje durante la fase de generación o depuración de *insights*. Este *prompt* adopta una estructura instruccional clara, definiendo el contexto del dominio, las reglas operacionales, las limitaciones por tipo de *insight*, y la información sobre los intentos previos que deben ser tenidos en cuenta:

Prompt del agente de *insights*

You are an advanced Planning Modeler AI Agent that can revise a structured set of insights by adding, editing, removing, or agreeing with existing insights. Your operations will be based on <Información sobre el modo de extracción>.

Domain: <Nombre del dominio>
<Descripción corta del dominio en lenguaje natural>

Domain PDDL:
<Modelo PDDL del dominio>

<Insights existentes>

Your task is to update these insight lists based on <Elementos a considerar>. Each operation must be applied to the list corresponding to the selected insight type.

The available operations are:

- **AGREE:** affirm an existing insight that is still relevant and valuable.
- **REMOVE:** discard an insight that is incorrect, redundant, too narrow, or superseded.
- **EDIT:** rewrite an existing insight to improve generality, clarity, or usefulness.
- **ADD:** introduce a new insight that is not already represented in the list.

You can apply up to <Límite de operaciones por tipo de insight> operations per insight type. Each insight can only receive one operation, and any insight not explicitly marked with AGREE,

REMOVE, or EDIT will remain unchanged. All generated insights must be general, reusable, and concise, focusing on helping the agent produce PDDL that respects the domain constraints and problem description.

<Información de los intentos>

<Instrucción si hay demasiados insights>

Below are the operations you do to the above list of EXISTING INSIGHTS:

De forma análoga a otros agentes del sistema, los elementos entre símbolos <...> son completados dinámicamente en función del dominio, el problema en cuestión, los intentos previos realizados, y los *insights* existentes en la base actual. A continuación se describe detalladamente el significado de cada uno de los componentes del *prompt*:

- *<Nombre del dominio>, <Descripción corta del dominio en lenguaje natural> y <Modelo PDDL del dominio>*: extraídos directamente de la base de datos de dominios, representan el contexto principal sobre el cual se realizan las recomendaciones.
- *<Límite de operaciones por tipo de insight>*: representa la cantidad máxima de operaciones que el agente puede aplicar a cada tipo de *insight* en una sesión. En este trabajo, se fijó este parámetro en 4.
- *<Insights existentes>*: sección que contiene la descripción de los tres tipos de *insights* definidos (conocimiento del dominio, reglas específicas del dominio, y reglas generales), junto con las listas actuales en cada categoría, numeradas para referencia.
- *<Información sobre el modo de extracción>*: depende del procedimiento utilizado para generar el conjunto de entrada. Se describe a detalle próximamente.
- *<Elementos a considerar>*: especifica el contenido relevante a considerar por el modelo en la generación o depuración de *insights*.
- *<Información de los intentos>*: provee los detalles sobre los intentos del agente modelador correspondientes al conjunto de entrada, ya sea trayectorias exitosas, trayectorias fallidas, reflexiones generadas, o retroalimentaciones de evaluación.
- *<Instrucción si hay demasiados insights>*: bloque opcional, que se añade cuando la cantidad de *insights* de un tipo sobrepasa un umbral definido ($M = 10$), e instruye al agente a priorizar la eliminación o depuración en lugar de seguir agregando nuevos elementos.

Este *prompt* se utilizó de forma iterativa en la fase de generación de *insights*, permitiendo al agente refining de manera controlada su base de conocimiento a partir del análisis crítico de los datos acumulados.

Comparación de un par de intentos de un mismo problema, uno fallido y uno exitoso

Para este modo de extracción, el *prompt* proporcionado al agente de *insights* incluye información detallada sobre ambos intentos —el fallido y el exitoso— del mismo problema. A continuación se presenta la forma en que se instancian los fragmentos variables del *prompt*:

- <Información sobre el modo de extracción>: se describe el tipo de comparación a realizar, incluyendo la fuente de información y el contexto de los intentos:

Modo de extracción de *insights*

two attempts (one successful and one unsuccessful) of the same task. In each attempt, the modeler agent received the domain description, the Domain PDDL file, and a natural language description of the problem, and produced a Problem PDDL file as output

- <Elementos a considerar>: se indica al agente qué patrones o señales debe analizar para proponer nuevas modificaciones al conjunto de *insights*:

Elementos a considerar

patterns, lessons or recurring mistakes observed by contrasting the failed trial to the successful trial

- <Información de los intentos>: se provee una representación detallada de ambos intentos, así como la reflexión realizada tras el intento fallido (si aplica):

Información de los intentos

Problem:

<Descripción del problema en lenguaje natural>

Failed attempt:

<Razonamiento realizado sobre el problema>

<Objetos determinados a utilizar>

<Modelo PDDL generado del problema>

<Feedback de evaluación sobre el fallo>

Reflection:

<Reflexión sobre el intento fallido>

Successful attempt:

<Razonamiento realizado sobre el problema>

<Objetos determinados a utilizar>

<Modelo PDDL generado del problema>

Comparación de varias soluciones correctas de distintos problemas del mismo dominio

En este caso, el agente de *insights* recibe un lote de problemas correctamente resueltos por el agente modelador dentro del mismo dominio, y debe derivar *insights* observando regularidades útiles en la representación del problema. A continuación se presenta la forma en que se instancian los fragmentos variables del *prompt*:

- <Información sobre el modo de extracción>: se describe que todos los intentos presentados fueron exitosos, y se mantiene el contexto común de dominio:

Modo de extracción de *insights*

successful task attempts. In each task, the same planning domain was used, and the modeler agent received the domain description, the Domain PDDL file, and a natural language description of the problem, and produced a Problem PDDL file as output

- <Elementos a considerar>: el objetivo se centra en la identificación de patrones positivos:

Elementos a considerar

patterns or lessons observed from the successful trials

- <Información de los intentos>: se construye una entrada por cada problema exitoso, separadas con una línea divisoria. Por cada intento *i*, se provee:

Información de los intentos exitosos

Problem #i:

<Descripción del problema en lenguaje natural>

<Razonamiento realizado sobre el problema>

<Objetos determinados a utilizar>

<Modelo PDDL generado del problema>

Estructura del proyecto implementado

Esta es la estructura que siguió la implementación, tal cual se presenta en el repositorio de *GitHub* (github.com/arielgg46/Thesis):

```
src (carpeta raíz o source)
| -- agents (implementación de agentes modeladores y planificadores)
| | -- modeler_agents.py (agentes modeladores con mejoras propuestas)
| | -- orig_llm_plus_p_agents.py (agentes modeladores baselines originales del trabajo LLM+P)
| | -- planner_agents.py (planificadores clásicos usados como baselines)
| | -- reflection_agent.py (agente reflexionador)
| -- classical_planner (wrapper para uso de planificadores externos)
```

```

| |-- planner.py (interfaz con Fast Downward y procesamiento de planes)
| |-- client (interfaz con APIs de LLMs)
| | |-- queries (registro de consultas a los modelos LLM en formato JSON)
| | |-- token_consumption.json (registro del consumo de tokens por consulta)
| | |-- client.py (código para realizar consultas a LLMs mediante la API de Fireworks AI)
| |-- dataset (carga, manejo y documentación del dataset)
| | |-- subsets (subconjuntos estratificados del benchmark Planetarium)
| | |-- dataset-v4.db (base de datos SQLite del dataset)
| | |-- dataset.py (funciones para acceder y modificar el dataset y sus subconjuntos)
| | |-- report.md (documentación técnica del dataset)
| |-- domains (recursos de los dominios)
| | |-- blocksworld (recursos de Blocksworld)
| | | |-- actions_description.txt (descripciones textuales de las acciones)
| | | |-- domain.pddl (modelo PDDL del dominio)
| | | |-- domain_description.txt (descripción textual breve del dominio)
| | | |-- fsp_ex_nl.txt (descripción en lenguaje natural del problema de ejemplo FSP)
| | | |-- fsp_ex_objects.json (objetos extraídos del ejemplo FSP)
| | | |-- fsp_ex_pddl.pddl (modelo PDDL del problema del ejemplo FSP)
| | | |-- fsp_ex_plan.pddl (plan generado para el ejemplo FSP)
| | | |-- fsp_ex_reasoning.txt (razonamiento estructurado asociado al ejemplo FSP)
| | | |-- planner_output_syntax.txt (estructura de salida esperada del planificador)
| | |-- floor-tile (archivos análogos para el dominio Floor-Tile)
| | |-- gripper (archivos análogos para el dominio Gripper)
| | |-- utils.py (utilidades para manejo de los recursos de dominio)
| |-- exp (componentes del módulo de aprendizaje experiencial)
| | |-- exps (pool de experiencias acumuladas durante entrenamiento)
| | |-- operations (registro de operaciones de la extracción de insights)
| | |-- experience_pool.py (gestión del pool de experiencias acumuladas)
| | |-- insights.json (archivo con insights extraídos)
| | |-- insights_extraction.py (lógica de extracción de insights)
| | |-- insights_extraction_progress.json (progreso de extracción de insights)
| | |-- training.py (entrenamiento del agente experiencial)
| | |-- training_progress.json (progreso del entrenamiento experiencial)
| |-- grammar (construcción de gramáticas para GCD)
| | |-- grammar.py (métodos para construcción de gramáticas)
| | |-- kovacs-pddl-3.1-2011.pdf (especificación oficial BNF de PDDL 3.1)
| | |-- pddl_bnf.py (definición de la gramática PDDL como BNF programática)
| |-- rag (módulo de Retrieval-Augmented Generation (RAG))
| | |-- api_embedder.py (generación de embeddings usando API de HuggingFace)
| | |-- embeddings_with_ids_test5.npz (embeddings precalculados del subconjunto de prueba)
| | |-- embeddings_with_ids_train5.npz (embeddings precalculados del subconjunto de entrenamiento)
| | |-- local_embedder.py (generación local de embeddings con sentence-transformers)
| | |-- retriever.py (clase Retriever para recuperación de experiencias similares via RAG)
| |-- results (registro de los resultados de la evaluación)
| |-- utils (utilidades varias para todo el sistema)

```

```

| |-- evaluation_utils.py (métodos para evaluación de los modelos generados)
| |-- io_utils.py (utilidades de entrada/salida)
| |-- pddl_utils.py (utilidades para manipulación de archivos PDDL)
| |-- planning_utils.py (funciones auxiliares para uso de planificadores clásicos)
| |-- result_utils.py (formateo y procesamiento de resultados experimentales)
| |-- tokens_utils.py (análisis y manejo del consumo de tokens)
|-- validator (módulo para validación planes)
| |-- validator.py (integración con VAL)
|-- visualizer (visualización de los resultados)
| |-- visualizer.py (generación de diagramas de las métricas evaluadas)
|-- check_planetarium_generation.py (verificación de generación del dataset Planetarium)
|-- config.py (configuración global de hiperparámetros y rutas)
|-- evaluations_progress.json (registro del progreso en evaluaciones básicas)
|-- exp_evaluation_progress.json (registro del progreso en evaluaciones del agente experiencial)
|-- main.py (punto de entrada principal del sistema)

```

Correcciones del *dataset* de *Planetarium*

Blocksworld

- **`equal_towers`**: la descripción indicaba incorrectamente la cantidad de torres generadas. Este error se encontraba en la función `abstract_description` de la clase `BlocksworldDatasetGenerator`, y se debía a una utilización incorrecta de la función `len()` para contar las torres.

```

1 # ISSUE: Incorrect tower count due to using len(blocks_list) instead of len
#          (num_blocks).
2 # FIX: Replaced all occurrences of len(blocks_list) with len(num_blocks).
3
4 case ("equal_towers", True):
5     num_blocks = self._equal_towers(blocks_list)
6     return f"You have {sum(num_blocks)} blocks, b1 through b{sum(num_blocks)}
7         ), stacked into {len(num_blocks)} towers of equal heights, and your arm
8         is empty."
9
10 case ("equal_towers", False):
11     num_blocks = self._equal_towers(blocks_list)
12     return f"Your goal is to stack the blocks into {len(num_blocks)} towers
13         of equal heights."

```

Ejemplo de código 4.1: Corrección de `equal_towers` dentro de `BlocksworldDatasetGenerator.abstract_description`

- **`invert`**: la descripción original contenía una redacción confusa y poco natural en inglés, dificultando la comprensión de la meta propuesta. Este error también se localizó en la función `abstract_description` de la clase `BlocksworldDatasetGenerator`.

```

1 # MINOR ISSUE: Awkward phrasing in goal description made it hard to
#               understand.
2 # FIX: Rephrased for clarity and naturalness.
3

```

```

4 case ("invert", True):
5     return f"You have {num_blocks} blocks, stacked into {len(blocks_list)}
towers of heights {', '.join(str(h) for h in blocks_list)}, and your arm
is empty."
6
7 case ("invert", False):
8     return f"Your goal is to invert each individual stack of blocks, such
that in each tower the block that was originally on the bottom will be
on the top."

```

Ejemplo de código 4.2: Mejora de redacción en `invert` dentro de `BlocksworldDatasetGenerator.abstract_description`

Gripper

- `drop_and_pickup`: se detectaron dos problemas principales. Primero, el generador no aseguraba que existiera una sala vacía distinta de la sala inicial, lo cual era un requisito lógico del escenario. Segundo, la descripción de la meta era ambigua y difícil de interpretar.

```

1 # POTENTIAL ISSUE: It requires at least one empty room other than the
starting one.
2 # FIX: Added exception handling for this case.
3
4 if not any([b == 0 for b in balls_in_rooms[1:]]):
5     raise ValueError("No empty room other than the starting one")

```

Ejemplo de código 4.3: Corrección lógica en `GripperDatasetGenerator.drop_and_pickup`

```

1 # POTENTIAL ISSUE: Unclear phrasing regarding balls not held by robbie.
2 # FIX: Clarified phrasing and ensured exception handling exists for layout
constraints.
3
4 case ("drop_and_pickup", False):
5     return "Your goal is to drop all the balls held by grippers in an empty
room that the robbie didn't start in, and pick up the balls that were
not initially held by the robbie."

```

Ejemplo de código 4.4: Mejora de redacción en `GripperDatasetGenerator.abstract_description` para `drop_and_pickup`

- `holding`: el generador no garantizaba que al menos un *gripper* sostuviera una pelota. Además, no se aseguraba que las pelotas estuvieran únicamente en la primera sala, como se requería. Estas verificaciones fueron agregadas a la función `holding`.

```

1 # ISSUE: Not ensured that at least one gripper is holding a ball.
2 # FIX: Added validation to ensure a ball is being held.
3
4 if sum(balls_in_grippers) == 0:
5     raise ValueError("Holding requires at least one gripper holding a ball"
)

```

Ejemplo de código 4.5: Control de errores en `GripperDatasetGenerator.holding`

```

1 # FIX: Ensured all balls not in grippers are in room 1 (init layout
2   constraint assumed).
3
4 case ("holding", True):
5   return f"{objects}. {max(sum(balls_in_grippers), 1)} balls are
6   distributed across the same number of grippers, and the rest are in the
7   first room. The robbie is in the first room."
8
9 case ("holding", False):
10   return f"Your goal is to make sure robbie is holding {max(sum(
11     balls_in_grippers), 1)} balls."

```

Ejemplo de código 4.6: Corrección en descripción abstracta para `holding`

- **juggle**: las descripciones abstractas no especificaban con claridad la dirección del desplazamiento de las pelotas, ni cómo se asignaban originalmente a los *grippers*. También era posible que se generaran configuraciones inválidas si no había suficientes *grippers* con pelotas. Se realizaron las aclaraciones necesarias en la función `abstract_description`.

```

1 # ISSUE: Ambiguity in description of juggling operation and initial state
2   mapping.
3 # FIX: Specified shift direction and explicit ball-gripper mapping in
4   description.
5
6 case ("juggle", True):
7   return f"{objects}. {sum(balls_in_grippers)} balls are distributed
8   across the same number of grippers (ball1 in gripper1, ball2 in gripper2
9   ... ball{sum(balls_in_grippers)} in gripper{sum(balls_in_grippers)}),
10  and the rest are in the first room. The robbie is in the first room."
11
12 case ("juggle", False):
13   return 'Your goal is to "juggle" the balls between the grippers that
14   started with balls, such that the balls are in the same grippers as
15   before, but shifted by one, to the left. The remaining balls should
16   remain untouched.'

```

Ejemplo de código 4.7: Mejoras de claridad y validez en
`GripperDatasetGenerator.abstract_description` para `juggle`

Floor-Tile

- **checkerboard**: el generador de predicados construía incorrectamente la matriz que representa el tablero, produciendo desalineaciones en la disposición de los colores. Además, los bucles anidados que asignaban los colores a las casillas no utilizaban correctamente los índices del tablero.

```

1 # POTENTIAL ISSUE: Incorrect tile matrix construction
2 # FIX: Corrected the loop ranges to match grid dimensions.
3
4 grid = [
5   [next(tiles_iter) for _ in range(grid_size_y)] for _ in range(
6     grid_size_x)

```

```

6 ]
7
8 predicates = []
9
10 # POTENTIAL ISSUE: Index loop variables were incorrect
11 # FIX: Adjusted to proper grid indexing.
12
13 for i in range(grid_size_x):
14     for j in range(grid_size_y):
15         if (i + j) % 2 == 0:
16             color = colors[0]
17         else:
18             color = colors[1]
19         predicates.append(Predicate("painted", grid[i][j], color))
20
21 return predicates

```

Ejemplo de código 4.8: Corrección en generación de predicados para `checkerboard` en `FloorTileDatasetGenerator`

- **rings**: la descripción de la posición inicial del robot en términos de anillos concéntricos era incorrecta, particularmente en las condiciones que determinaban su localización dentro del tablero. Se identificaron errores de lógica y redacción en la función `abstract_description.get_robot_ring_string`:

```

1 # ISSUE: Incorrect corner logic and ordinal computation
2 # FIX: Adjusted corner conditions and ring numbering.
3
4 if pos_x == pos_y and pos_x < (grid_size_x + 1) // 2:
5     pos_str = f"the top-left corner of the {int_to_ordinal(pos_x + 1)} ring
6         from the outside"
7
8 elif pos_x == pos_y and pos_x >= (grid_size_x + 1) // 2:
9     pos_str = f"the bottom-right corner of the {int_to_ordinal(grid_size_x
10 - pos_x)} ring from the outside"
11
12 elif pos_x == grid_size_x - pos_y - 1 and pos_x < (grid_size_x + 1) // 2:
13     pos_str = f"the top-right corner of the {int_to_ordinal(pos_x + 1)} ring
14         from the outside"
15
16 elif pos_y == grid_size_y - pos_x - 1 and pos_x >= (grid_size_x + 1) // 2:
17     pos_str = f"the bottom-left corner of the {int_to_ordinal(pos_y + 1)} ring
18         from the outside"

```

Ejemplo de código 4.9: Corrección de condiciones y redacción en `get_robot_ring_string`

- **paint_x**: la descripción abstracta generada para la tarea de pintar una forma en X era poco clara. Dependiendo del número de colores requeridos, se mejoraron las formulaciones para especificar con precisión los objetivos visuales de la tarea.

```

1 # POTENTIAL ISSUE: Vague or awkward phrasing of goal description
2 # FIX: Clarified intent and use of one or two colors for painting an "X".
3
4 case ("paint_x", False):

```

```
5     if n_colors == 1:  
6         return "Your goal is to paint, with a single color, an 'X' shape  
across the grid."  
7     elif n_colors == 2:  
8         return "Your goal is to paint, with a single color, an 'X' shape  
across the grid, and every other tile should be painted with a different  
color."
```

Ejemplo de código 4.10: Mejora de redacción en `abstract_description` para `paint_x`