

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“AGENTE ASTRONÓMICO CON HERRAMIENTAS Y GENERACIÓN AUMENTADA POR RECUPERACIÓN”

VICENTE ALVAREZ AGUAYO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Mauricio Solar
Profesor Correferente: Mauricio Araya

Enero - 2025

DEDICATORIA

Le dedico este trabajo de memoria a mi familia, quienes siempre estuvieron ahí para apoyarme, incluso cuando tropecé y volví a tropezar. Nunca me recriminaron o me hicieron sufrir por los traspiés que pude sufrir, y siempre me motivaron a seguir adelante, a “hacerlo nomás”, y siempre me dijeron lo orgullosos que están de mí.

AGRADECIMIENTOS

Agradezco nuevamente a mi familia; mi papá y mi mamá por siempre apoyarme, por preocuparse por mí, por motivarme a seguir y por aconsejarme en el cómo; y a mi hermano, por ser un amigo permanente a mi lado, con quien comparto más y quien me conoce mejor. Agradecer a mis amigos, de la universidad, de colegio, o los que conozco por otros medios, gracias por darme tantas risas y consejos, ya sea académicos o de la vida. A mi pareja, por hacer los días más lindos con su sola presencia, y por retarme cuando no trabajo.

RESUMEN

Resumen— En este trabajo se desarrolló la primera iteración de un asistente virtual de tipo “chatbot” para el Observatorio Virtual. Este asistente corresponde a un “agente”; un Modelo de Lenguaje Grande con acceso a herramientas que le permiten recuperar, transformar y entregar información al usuario. Se desarrolló desde su grafo de procesamiento y sus herramientas individualmente, hasta la interfaz gráfica con la que interactúa el usuario. Luego, se realizaron pruebas con usuarios de distintos niveles de conocimiento astronómico y se recolectaron datos respecto al correcto funcionamiento, cumplimiento de metas y sugerencias para mejoría del asistente. Se realizó un análisis de los datos recolectados y se concluye finalmente que el asistente prueba ser útil y despertar el interés en los usuarios, quienes creen en el potencial del asistente, pero necesita más iteraciones para realmente cumplir con el objetivo de acercar a la población al trabajo detrás de la astronomía y facilitar este trabajo a los profesionales.

Palabras Clave— Observatorio Virtual, Agente, IA, Asistente, Astronomía

ABSTRACT

Abstract— In this work, the first iteration of a virtual assistant of the “chatbot” type for the Virtual Observatory was developed. This assistant is an “agent”; a Large Language Model with access to tools that allow it to retrieve, transform and return information to the user. It was developed from its processing graph and its individual tools, to the graphic interface which the user interacts with. Then, tests were run with users of different levels of astronomical knowledge and data was collected regarding the correct functioning, goal achievement and suggestions for improvement of the assistant. An analysis of the collected data was made and it is finally concluded that the assistant proves to be useful and lights an interest in the users, who believe in the assistant’s potential, but needs more iterations to really achieve the goal of bringing the population closer to the work behind astronomy and making this work easier for professionals.

Keywords— Observatorio Virtual, Agent, AI, Assistant, Astronomy

GLOSARIO

IVOA: *International Virtual Observatory Alliance*
VO: *Virtual Observatory*
EduIG: *Education Interest Group*
EuroVO: *European Virtual Observatory*
ChiVO: *Chilean Virtual Observatory*
VLT: *Very Large Telescope*
ELT: *Extremely Large Telescope*
ALMA: *Atacama Large Millimeter/submillimeter Array*
XML: *eXtensible Markup Language*
SQL: *Structured Query Language*
ADQL: *Astronomical Data Query Language*
PQL: *Parameterized Query Language*
FITS: *Flexible Image Transport System*
TAP: *Table Access Protocol*
SIAP: *Simple Image Access Protocol*
SLAP: *Simple Line Access Protocol*
SCS: *Simple Cone Search*
SSA: *Simple Spectral Access*
SAMP: *Simple Application Messaging Protocol*
UWS: *Universal Worker Service*
TOPCAT: *Tool for Operations on Catalogues And Tables*
FONDEF: *Fondo de Fomento al Desarrollo Científico y Tecnológico*
RNN: *Recurrent Neural Network*
GPT: *Generative Pre-trained Transformer*
AWS: *Amazon Web Services*
LLM: *Large Language Model*
JSON: *JavaScript Object Notation*
API: *Application Programming Interface*
JWT: *Jason Web Token*

ÍNDICE DE CONTENIDOS

RESUMEN	IV
ABSTRACT	IV
GLOSARIO	V
ÍNDICE DE FIGURAS	VIII
ÍNDICE DE TABLAS	VIII
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	3
1.1 Justificación del Proyecto	3
1.2 Objetivos	4
1.2.1 Objetivo General:	5
1.2.2 Objetivos específicos:	5
CAPÍTULO 2: MARCO CONCEPTUAL	6
2.1 Observatorio Virtual e IVOA	6
2.1.1 Estructura de IVOA y estándares para el Observatorio Virtual	7
2.1.2 Métodos para conectarse al Observatorio Virtual	10
2.1.3 Observatorio Virtual Chileno (ChiVO)	13
2.2 IA, Búsqueda Inteligente y <i>Machine Learning</i>	13
2.2.1 <i>Machine Learning</i> y Redes Neuronales	14
2.2.2 Transformers y Procesamiento de Lenguaje Natural	16
2.3 Competencias y Otras herramientas Educativas	19
2.3.1 Stellarium	19
2.3.2 Aladin	21
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	23
3.1 Investigación de posibles soluciones	23
3.2 Definición de arquitectura de la solución	27
3.3 Desarrollo del asistente virtual	28
3.3.1 Bases del agente	28
3.3.2 Comunicación con el front-end	30
3.3.3 Definición de herramientas (<i>tools</i>), estructura de comunicación e implementación	31
Registry	32
Simple Image Access	34
Simple Spectral Access	37
Simple Cone Search y Simple Line Access	39
3.0.4 Generación asistida de recuperación de documentos (RAG)	41

3.0.5	Implementación completa del flujo de trabajo del agente	43
3.0.6	Definición de interfaz gráfica y presentación de información	46
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN		50
4.1	Metodología de validación y definición de medio	50
4.1.1	Usuarios y resultados	51
4.2	Análisis de datos recolectados	51
CAPÍTULO 5: CONCLUSIONES		58
5.1	Cumplimiento de los objetivos	58
5.2	Trabajo futuro	58
ANEXOS		60
REFERENCIAS BIBLIOGRÁFICAS		61

ÍNDICE DE FIGURAS

1	Arquitectura nivel 1 IVOA.	7
2	Arquitectura nivel 2 IVOA.	8
3	Pantalla inicial de TOPCAT.	11
4	Pantalla de TAP quey al VO.	12
5	Arquitectura de un modelo Transformer.	17
6	Interfaz web de Stellarium.	19
7	Interfaz web de Stellarium.	20
8	Interfaz web de Stellarium.	21
9	Interfaz principa de Aladin. De Izquierda a derecha, arriba a abajo son: Árbol de colecciones, Vista, Stack, Medidas	22
10	Representación de alto nivel de una Red Neuronal Recurrente.	23
11	StateGraph de chatbot inicial.	30
12	StateGraph del Agente complejo con herramientas y RAG.	45
13	Cantidad de usuarios por nivel de conocimiento (rango nivel: 1 a 10)	51
14	Nivel de conocimiento promedio de los usuarios que buscaron previamente datos duros astronómicos.	52
15	Porcentaje de usuarios que tuvieron problemas al usar la aplicación	53
16	Distribución de calificación de facilidad de uso (rango calificación: 1 a 7).	53
17	Calificación respecto a cumplimiento del objetivo de facilitar el acceso a datos astronómicos (rango calificación: 1 a 7).	55
18	Probabilidades de utilización de la aplicación a futuro (rango prob.: 1 a 10)	55

ÍNDICE DE TABLAS

INTRODUCCIÓN

La astronomía es el estudio científico de los objetos celestes y fenómenos que se originan fuera de la atmósfera terrestre. Estos fenómenos van desde las características físicas y químicas de los objetos celestes y su comportamientos hasta la formación y origen del universo. Esta ciencia interdisciplinaria se divide en 3 grandes áreas: Cultura y Sociedad, Ciencia e Investigación, y Tecnología y Habilidades. Es en esta última área que se encuentran los computadores, el crear hardware más rápido y software más complejo, metas cruciales en la era de la tecnología y de los grandes flujos de datos. Debido a esto, ciencias como la astronomía están optando por el “descubrimiento científico orientado a los datos”[Borne, 2010], que es de donde nace la sub-disciplina de la **astroinformática**.

La astroinformática incluye áreas de la informática como *Computer Science*, lo que incluye redes neuronales y *machine learning*, el Desarrollo de Software para apoyar a los astrónomos y astrofísicos en sus labores, el desarrollo de Tecnologías de la Información para sustentar toda la comunicación entre los centros de investigación, el almacenamiento y procesamiento de los datos y el mantenimiento y despliegue de aplicaciones. Por último, los Sistemas de Información son necesarios para integrar hardware y software, permitiendo que los grupos de investigación y organizaciones realicen sus funciones principales.

Cada una de estas áreas es crucial dentro del proceso de investigación científica de la astronomía y la astrofísica, y es en las áreas de Desarrollo de Software y *Computer Science* que surge el problema de la inmensa cantidad de datos que es generada por los instrumentos astronómicos actualmente.

Las últimas 2 décadas se ha hablado de la importancia que tendrán las disciplinas orientadas al manejo de grandes volúmenes de datos, ya que estimaba el crecimiento desmesurado de la cantidad de datos, donde no todos son útiles, por lo que el proceso de minería y análisis de estos datos sería crucial para el correcto crecimiento y funcionamiento de las ciencias en el futuro. Actualmente existen sistemas y herramientas que asisten a los investigadores en la búsqueda, procesamiento y análisis de estos datos, como lo es el Observatorio Virtual junto con la Alianza Internacional de Observatorios Virtuales (IVOA). El Observatorio Virtual (VO, por sus siglas en inglés) es una colección de *data centers* y herramientas que buscan masificar el acceso a datos astronómicos y facilitar la utilización y procesamiento de estos. La *International Virtual Observatory Alliance* (IVOA) tiene como rol dictar y hacer cumplir estándares para la estructura y el acceso a los datos, de forma que todo quien desee acceder a estos pueda hacerlo sin dificultad. Esto es cierto solamente para individuos con un considerable conocimiento técnico y astronómico, ya que la forma de acceso a estos datos es mediante herramientas especializadas, las cuales por lo mínimo requieren el suficiente conocimiento astronómico para poder entender qué dato se desea buscar, donde hacerlo y como poder leerlo, debido a que los datos astronómicos no están estructurados de forma tal que una persona común, sin conocimientos especializados, pueda leerlos. Esto ha resultado en una baja popularidad de los Observatorios Virtuales entre astrónomos.

En este trabajo se busca solventar esta problemática mediante una interfaz de comunicación con el Observatorio Virtual. La naturaleza de la interfaz será una aplicación web que emula el comportamiento de chatos, pero haciendo uso de las capacidades de un “agente” para poder razonar según la interacción con el usuario y tomar acción para poder buscar, formular y entregar la información solicitada. Durante el trabajo se explicará el marco conceptual del Observatorio Virtual, IVOA y las herramientas ya existentes para acceder a los datos que proporcionan. Después, se indagarán posibles implementaciones para la solución propuesta, para llegar a la conclusión de un agente con herramientas y capacidad de recuperación de documentos. Se implementará un *backend* con un *Large Language Model* como nodo de raciocinio del Agente y se otorgará acceso al VO con herramientas. También se desarrollará una interfaz en un *frontend* que permitirá comunicarse con el agente de forma sencilla y cómoda. Luego de la implementación de la solución, se realizarán encuestas a usuarios de niveles de conocimiento distintos para poder obtener conclusiones respecto a si se alcanzaron los objetivos y como seguir desarrollando a futuro.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

Tomando en cuenta la relevancia de la astronomía en el quehacer humano, y las dificultades de alcance que está sufriendo debido a su complejidad técnica y académica, es que se propone una aplicación de búsqueda basada en lenguaje natural que funcione directamente con las bases de datos astronómicas del Observatorio Virtual, la cual permitirá la interacción directa sin necesidad de tener un mayor conocimiento astronómico ni técnico, entregando los datos e información requeridos de una forma simple.

Serán considerados los siguientes actores en el contexto de este trabajo:

1. IVOA y el Observatorio Virtual:

- **Relevancia:** IVOA es la organización que se preocupa de los estándares con los que debe cumplir el VO, así como su divulgación y alcance. El observatorio Virtual es de donde provienen todos los datos y herramientas a utilizar.
- **Problema:** La complejidad de las herramientas para acceder y procesar los datos del VO ha significado una baja popularidad entre los astrónomos y un difícil acercamiento a personas sin conocimiento especializado.

2. Usuario de bajo conocimiento astronómico:

- **Relevancia:** El usuario de la aplicación a quien se intenta integrar más en la astronomía real”.
- **Problema:** La complejidad de los lenguajes de consulta para acceder a datos del VO, junto con la poca difusión de fuentes de información y otras herramientas, vuelven casi imposible el acceder a información real”.

3. Usuario de alto conocimiento astronómico:

- **Relevancia:** Quien puede hacer uso de la información del VO para hacer ciencia y apoyarse en sus investigaciones.
- **Problema:** Utilizar las herramientas del VO puede llegar a ser engorroso, incluso para investigadores y astrónomos, lo que desmotiva la utilización de estos datos.

1.1. Justificación del Proyecto

La situación de popularidad de los Observatorios Virtuales cobra mayor importancia cuando se toma en cuenta que uno de los principales objetivos de la astronomía es su divulgación y alcance; el acercar la astronomía, sus maravillas y descubrimientos a la gente, sobre todo a

escuelas, alumnos, profesores y personas que quieran aprender por su cuenta. La ciencia de la astronomía, en las últimas décadas, a cambiado su enfoque a uno de "*big science*", haciéndose más compleja y haciendo más difícil el despertar la curiosidad e interés en el público, más allá de el asombro visual que generan las imágenes captadas por los instrumentos. Es por esto que la misma IVOA creó en 2012[lafrate, 2019] un grupo dedicado exclusivamente a la recolección de recursos para el aprendizaje de estudiantes y profesores, sus experiencias, y formas en las que el Observatorio Virtual puede avanzar y mejorar sus capacidades educativas. Este grupo es denominado el "*educational interest group*"(EduIG¹). El EduIG busca ayudar a IVOA a jugar un rol importante en la sociedad; el acercar la astronomía a la gente.

Dentro de las actividades que le conciernen al EduIG se encuentran:

- Identificar científicos y/o educadores interesados en trabajar dentro de EduIG.
- Recolectar información de actividades educativas que podrían verse beneficiadas por recursos del VO o que ya los usen.
- Compilar listas de recursos educativos generales en astronomía que podrían resultar útiles en el medio ambiente del VO.
- Abrir canales de comunicación con proyectos activos en astronomía y educación a nivel mundial.
- Evaluar y discutir requerimientos educacionales en herramientas, datos y guías que pudiesen ser enviadas a IVOA.
- Entregar apoyo desde la IVOA a actividades y acciones provenientes de los VO nacionales en búsqueda de obtener reconocimiento y financiamiento por parte de estructuras nacionales educacionales de alto nivel y/o de organizaciones de profesores a escala nacional.
- Crear y mantener un repositorio online donde se guarde documentación, listas de recursos disponibles y proyectos para educadores así como desarrolladores del VO y astrónomos.

Es gracias a agrupaciones como estas que han logrado surgir proyectos como el VO *for education*, del Observatorio Virtual Europeo (EuroVO, por sus siglas en inglés). El EuroVO tiene por objetivo el llevar la ciencia a los estudiantes, hacer que se involucren y darles una visión de lo que es la astronomía profesional.

1.2. Objetivos

A continuación, se describen los objetivos que guiarán el desarrollo del proyecto:

¹<https://wiki.ivoa.net/twiki/bin/view/IVOA/IvoaEducation>

1.2.1. Objetivo General:

Crear una aplicación web que permita realizar búsquedas en lenguaje natural sobre bases de datos de los observatorios virtuales y, además, permita depurar la búsqueda a través del diálogo con el buscador para enriquecer los resultados.

1.2.2. Objetivos específicos:

- Crear asistente virtual como modelo de lenguaje natural con conocimiento astronómico específico y actualizado que pueda interactuar con el usuario
- Dar acceso al asistente al Observatorio Virtual, para que pueda recuperar datos actualizados de estudios y observaciones
- Crear una interfaz fácil de usar para usuarios independiente de su nivel de conocimiento, de forma que puedan acceder a la información sin tener que estudiar el como
- Validar la solución y analizar formas en la que pueda seguir creciendo y mejorando para acercar a más gente a la astronomía

CAPÍTULO 2

MARCO CONCEPTUAL

2.1. Observatorio Virtual e IVOA

El Observatorio Virtual es una iniciativa internacional nacida en 2001 [Quinn *et al.*, 2002] y dirigida por la Asociación Internacional de Observatorios Virtuales (IVOA, por sus siglas en inglés) con el objetivo de otorgar acceso internacional a datos astronómicos de primera fuente a todo quien necesite de estos, sin necesidad de pertenecer a asociaciones de investigación científica ni instituciones académicas, ni de tener que ir a visitar instalaciones donde se ubiquen grandes agrupaciones de servidores.

La necesidad del Observatorio Virtual (en adelante VO, por sus siglas en inglés) viene de dos problemáticas: La explosión en el tamaño de los conjuntos de datos astronómicos que viene incrementando desde inicios del milenio, gracias a los avances en tecnologías y herramientas para observar los cielos y más allá. Los datos que componen las grandes bases de datos del VO provienen de distintos instrumentos alrededor del mundo, como lo son el *Very Large Telescope* (VLT), el *Extremely Large Telescope* (ELT) y el *Atacama Large Millimeter/submillimeter Array* (ALMA), entre otros.

Todos estos instrumentos realizan observaciones los 365 días del año, generando gigabytes de datos en 1 día. Como ejemplo, se estimó en su momento que el observatorio ALMA, en su máxima capacidad, debería ser capaz de generar 80 Gb de datos diarios, cuando en realidad genera aproximadamente 1TB diario (1000 GB)². Esta cantidad de datos no es manejable ni procesable por los astrónomos individualmente, ni el tiempo ni la capacidad de memoria y computo de los equipos de los investigadores y científicos daría abasto para ello. Y, además del tamaño de los datos a ser guardados y estudiados.

La otra situación que amerita la existencia del VO es que, a inicios de los años 2000, los conjuntos de datos astronómicos estaban completamente desconectados entre sí, desaprovechando la increíble oportunidad que se ofrecía al lograr conectarlos en un “universo digital”[Quinn *et al.*, 2002].

Integrada por 24 miembros alrededor del mundo y fundada en Junio de 2002[ESO, 2002], la Asociación Internacional de Observatorios Virtuales (IVOA) tiene como misión “facilitar la coordinación internacional y colaboración necesaria para el desarrollo y despliegue de herramientas, sistemas y estructuras organizacionales necesarias para permitir la utilización internacional de archivos astronómicos como un integrado e interoperable observatorio virtual”[IVOA, 2023]. Para esto, IVOA define estándares que se deben cumplir para poder formar parte de esta, estándares respecto métodos de acceso a los datos y la información, así como la estructura de estos datos. Es por esto que se crearon distintos protocolos de

²<https://www.almaobservatory.org/en/factsheet/>

acceso a los datos, protocolos que serán descritos más adelante, así como la estructura de estos datos. Es responsabilidad de IVOA exigir y hacer cumplir estos estándares para que organizaciones e instituciones pueda integrarse a la increíble cantidad de recursos que ofrece el VO.

Gracias a IVOA y el VO es que organizaciones de todo el mundo han podido colaborar entre ellas y proyectos científicos se han logrado financiar. Solo durante 2002 el financiamiento de 3 proyectos internacionales fue discutido, financiamiento que llegaba a los 20 millones de dolares[Quinn *et al.*, 2002].

2.1.1. Estructura de IVOA y estándares para el Observatorio Virtual

EL Observatorio Virtual no es solo un conjunto de *data centers* a los que se puede acceder, son toda la estructura de los datos y servicios que permiten acceder a los mismos, así como las herramientas y programas que permiten interactuar con estos datos.

La arquitectura definida por IVOA del VO se puede separar en 3 capas:

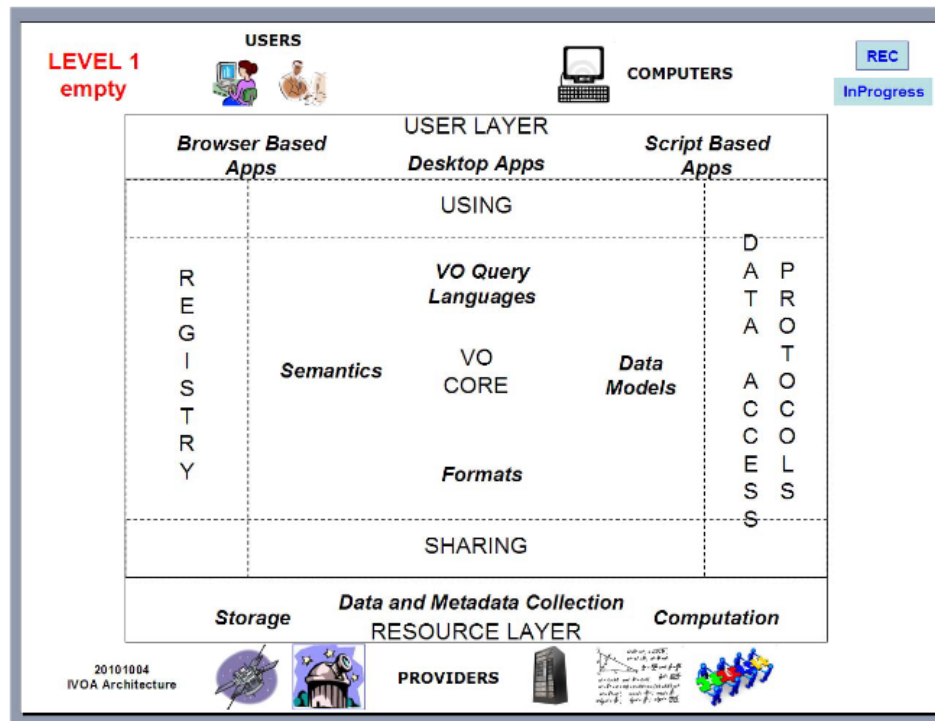


Figura 1: Arquitectura nivel 1 IVOA.

Fuente: IVOA Architecture.

- **Capa de Recursos:** Son los proveedores de los datos y/o servicios computacionales a través del internet. Aquí es donde se ubican los servidores que contienen los datos,

los servicios de almacenamiento, procesamiento y computación.

- **Capa Usuaría:** Estos son los sistemas computacionales, grupos de investigadores o astrónomos, que utilizan los servicios y consumen los datos. Aquí se ubican las aplicaciones web, de escritorio u otras.
- **Capa Media:** Aquí es donde se centra el Observatorio Virtual. Esta es la capa que permite que los usuarios se conecten con los recursos. Permite que los usuarios encuentren, compartan, obtengan y usen los servicios y datos que ofrece la capa de recursos. También define los estándares de los recursos.

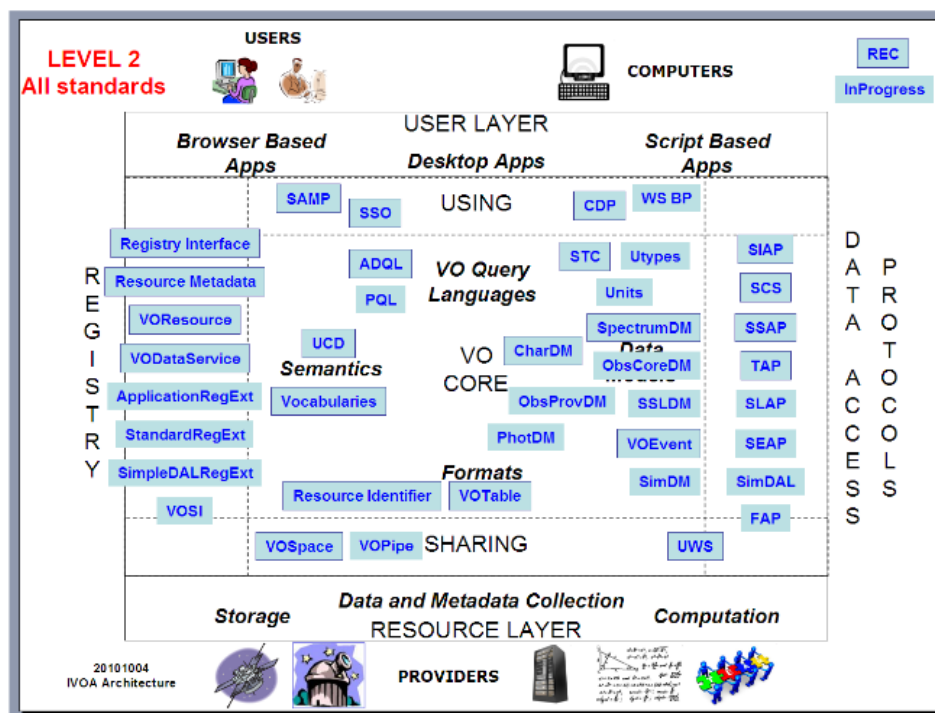


Figura 2: Arquitectura nivel 2 IVOA.

Fuente: IVOA Architecture.

La capa intermedia tiene un nivel de complejidad superior al que se muestra en la figura 1. En este, se detallan los protocolos de acceso a datos y estándares de calidad y estructura que deben cumplir. Debido a que es necesario comprender como están compuestos y como acceder a los datos dentro del VO es que es necesario describir esta información.

1. **La subsección Registry:** El Registry (o Registro) de IVOA permite a los usuarios y aplicaciones de la capa Usuaría encontrar datos y colecciones de metadata, así como servicios, aplicaciones y demases, todo en la capa de Recursos. Los recursos son cualquier cosa perteneciente al VO que posea un identificador único y sea mantenible, ejemplos

son instrumentos, ideas, organizaciones, datos, aplicaciones, servicios de accesos, etc. Todos estos recursos están definidos en base a los estándares que impone IVOA:

- *Resource Metadata* o Metadata de los Recursos es el estándar más básico que deben cumplir los recursos. Los usuarios necesitan saber que recursos están disponibles y donde, de donde provienen, como usarlos, su composición, etc. Este estándar describe como asegurar esto.
- *VOResource* o RecursoVO describe un estándar de codificación XML que finalmente son los metadata de los recursos. Este estándar es necesario para que los publicadores puedan describir sus recursos en el registro y para que usuarios y aplicaciones puedan encontrarlos.
- *Registry Interface* o Interfaz del Registro especifica como los registros intercambian la metadata definida en el estándar anterior o acceden a esta, así como aplicaciones también lo hacen. Está compuesto por dos interfaces: Registros de búsqueda, que permite a la capa usuaria hallar registros, y Registros de publicación.

2. **La subsección VO Core:** Esta subsección define estándares para desarrolladores y proveedores de servicios, como los descritos previamente. Dado que el enfoque de este proyecto no es el de proveer un nuevo servicio para el VO, si no una aplicación que acceda a los mismos, se describirá superficialmente al VO Core.

El VO Core está compuesto de 4 grupos de estándares:

- **Estándares de Modelos de Datos:** Debido a que los proveedores de datos guardan estos dependiendo de sus necesidades, siendo estas diferentes, se volvería difícil integrarlos todos en el VO. Para esto se definen modelos de datos que describen las relaciones entre las colecciones de datos astronómicos y sus metadata, así se vuelven interoperables.
- **Estándares de Lenguajes de Consulta:** Existen variantes del *Structured Query Language* (SQL) dependiendo del área. Para bases de datos astronómicas existe el *Astronomical Data Query Language* (ADQL).
- **Semánticas:** Se especifican formatos estandarizados para vocabularios que las aplicaciones VO puedan aprovechar. Estos son vocabularios para describir posiciones, iluminación, tiempo, calificaciones, nombres, etc.
- **Estándares de Formato:** Hay 2 formatos: *VOTable* y *Resource Finder*. El primero es el identificador único y global de un recurso. El segundo es un formato de almacenamiento e intercambio de datos tabulares, enfocado en tablas astronómicas. Está hecho para asemejarse al estándar astronómico FITS [Hanisch et al., 2001].

3. **La subsección Data Access Protocols:** Aquí se definen los servicios de consulta y acceso a los datos presentes en la capa de recursos. Se define un estándar para cada tipo de datos. Estos servicios sirven para acceder a los datos, no solo leer sus descripciones:

- **Table Access Protocol (TAP):** Define una interfaz simple para consultar datos tabulares en general, lo que incluye catálogos astronómicos y otras tablas genéricas. El protocolo otorga soporte para distintos lenguajes de consulta, como lo son el *Astronomical Data Query Language* (ADQL) y el *Parameterized Query Language* (PQL). Además, estas consultas pueden ser síncronas o asíncronas.
 - **Simple Image Access Protocol (SIAP):** Define una interfaz simple para poder obtener datos en formato de imagen de varios repositorios de imágenes astronómicas.
 - **Simple Line Access Protocol (SLAP):**
 - **Simple Cone Search (SCS):** Define un protocolo de consulta que, dado una posición en el cielo y distancia angular, define un cono en el cielo, retornando una lista de fuentes astronómicas que calcen dentro del cono, todo en formato *VO-Table*.
 - **Simple Spectral Access (SSA):** Describe una interfaz uniforme para descubrir y acceder al espectro unidimensional que provenga de colecciones de datos y metadatos. Puede ser utilizado desde aplicaciones VO para acceder a los recursos de espectro asociados.
4. **La subsección Using:** Aquí se describen tres estándares, dos de los cuales aplican para casos donde es necesario cierto nivel de autorización, ya que existen datos que pueden estar protegidos o no sean de acceso público. Por ejemplo, cuando los datos aún estén en tiempo propietario del astrónomo o investigador que los descubrió. El otro estándar es el SAMP.
- El *Simple Application Messaging Protocol* (SAMP) es un estándar que permite a herramientas de software el intercambiar información de datos y control para facilitar la interoperabilidad de herramientas y, así, permitir que los usuarios solucionen sus distintos problemas. Este protocolo no es exclusivo de herramientas astronómicas, sino para todo trabajo que involucre la astronomía.
5. **La subsección Sharing:** Esta subsección habla de estándares y protocolos que están más cercanos a la capa de Recursos, en la cual este proyecto no ahondará. De todas formas, se mencionarán los estándares por razones de completitud:
- **VOSpace:** Punto de acceso para una red de almacenamiento distribuida. permite manipular datos.
 - **UWS:** Permite el desarrollo de aplicaciones que trabajen con llamadas asíncronas.

2.1.2. Métodos para conectarse al Observatorio Virtual

Los métodos para conectarse al observatorio virtual son varios, y algunos ya fueron descritos o mencionados. La forma más básica de conectarse al VO es ADQL [naki Ortiz, 2008],

generando una consulta o *query* en base a criterios recuperados durante el dialogo con el interlocutor.

Otra opción es usar aplicaciones desarrolladas para las consultas al VO. Algunas de estas ya fueron descritas y exploradas, como Aladin³, el atlas celestial. Otros ejemplos incluyen TOPCAT⁴.

TOPCAT es un visualizador gráfico interactivo y editor para datos tabulares. Puede ser utilizado para datos astronómicos como no astronómicos, aunque su enfoque principal es hacia el primer tipo. En su mismo sitio web dice que “porque utiliza estándares del Observatorio Virtual, puede cooperar tranquilamente con otras herramientas, servicios y datasets en el mundo del VO”[Group, 2022].

Dentro de las funcionalidades que ofrece TOPCAT están el visualizar datos en formato tablas e inspeccionar sus descripciones. Estos datos pueden provenir localmente o desde el VO, del cual se pueden extraer datos bajo los protocolos mencionados previamente en la sección 2.1.1. Además, TOPCAT ofrece herramientas para graficar los datos de distintas formas: en plano, 3D con coordenadas cartesianas o polares, en series de tiempo y en el cielo.

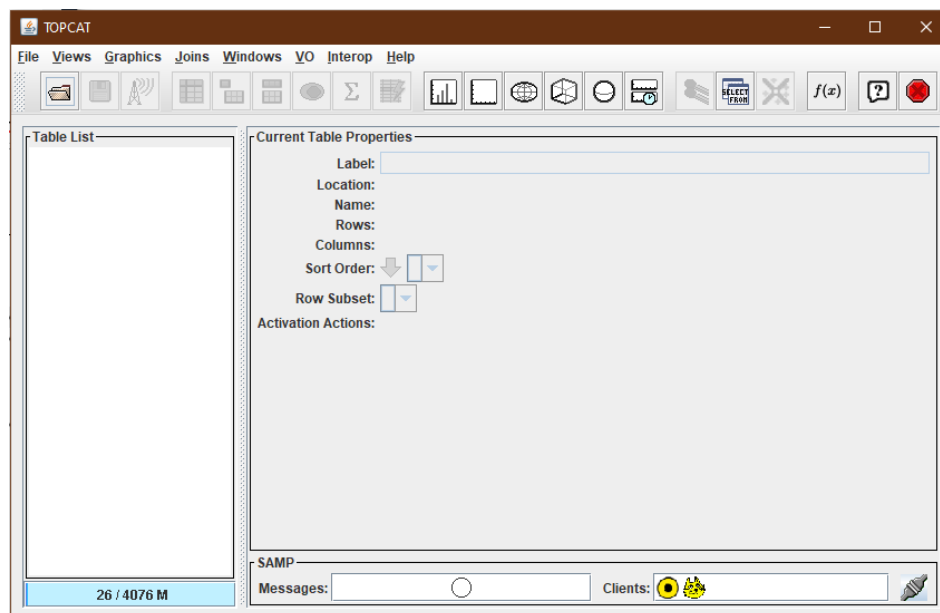


Figura 3: Pantalla inicial de TOPCAT.

Fuente: TOPCAT.

En la figura 4 se puede observar la pantalla para realizar una consulta a los servicios TAP del Observatorio Virtual. En esta pestaña se puede buscar por palabras clave en los registros del VO. Una vez seleccionado el servicio se puede leer la información del mismo, así como una descripción de sus contenidos, sus columnas y se pueden filtrar sus contenidos con una

³<http://aladin.cds.unistra.fr/aladin.gml>

⁴<https://www.star.bris.ac.uk/mbt/topcat/>

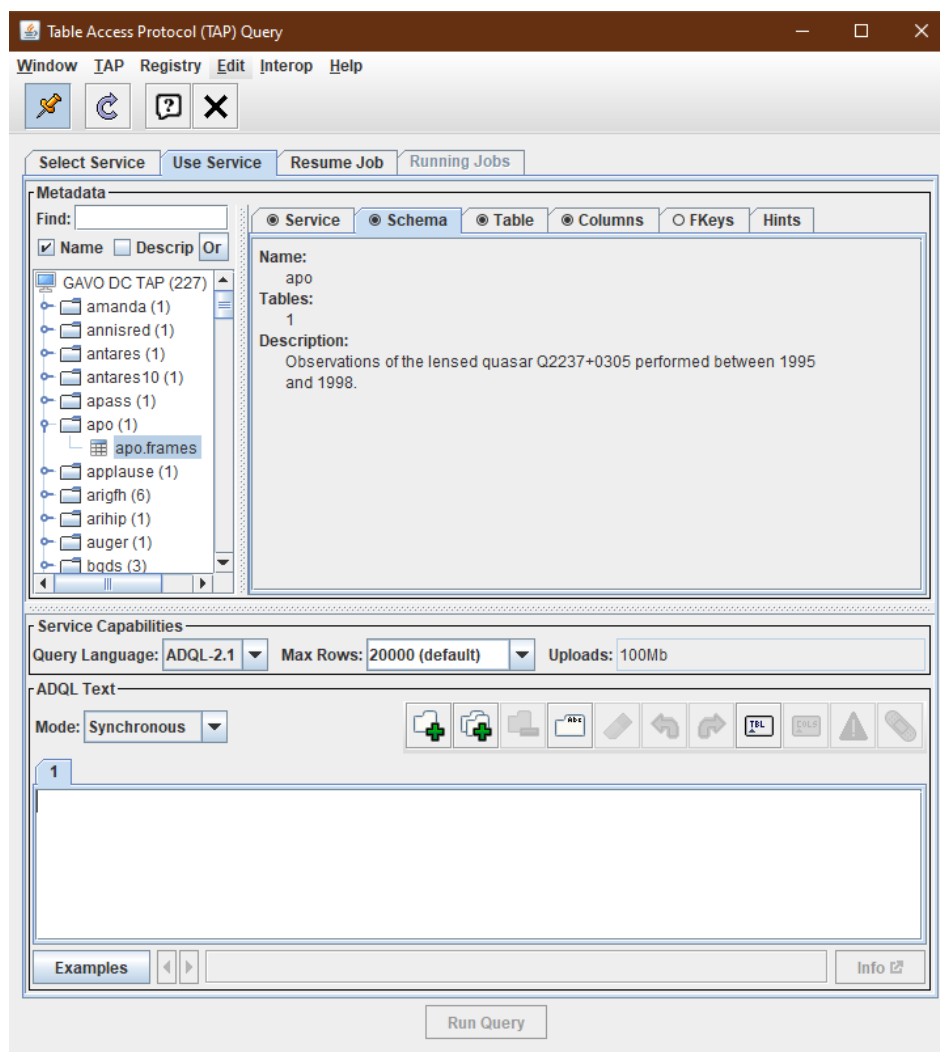


Figura 4: Pantalla de TAP query al VO.
Fuente: TOPCAT.

consulta en formato ADQL.

Además de los dos casos anteriores de herramientas/aplicaciones que permiten el acceso al VO, también hay librerías en lenguajes de programación que podrían ayudar a la recuperación de datos del VO. Una de estas, y la más conocida y confiable es **Astropy**.

Astropy [Team, 2023] es un *core package* para astronomía que utiliza el lenguaje de programación **Python**⁵. Que sea un *core package* o “paquete núcleo” quiere decir que reúne varios paquetes de librerías que proporcionan funcionalidades distintas para trabajar con datos astronómicos, lo que incluye su acceso, manipulación, visualización y análisis. Estos paquetes se dividen entre el paquete núcleo y los paquetes afiliados. En el paquete núcleo se encuen-

⁵<https://www.python.org/>

tran las clases, utilidades y marcos de trabajo o *frameworks* que entregan las herramientas más utilizadas en astronomía. Por otro lado, los paquetes afiliados son otros paquetes que están relacionados con la astronomía, pero no son nucleares a Astropy. Dentro de estos paquetes existe uno llamado **PyVO**.

PyVO⁶ permite la conexión con el VO y el acceso a sus recursos desde código en Python. Estos recursos son a los que se puede acceder utilizando los estándares de la IVOA descritos en los *Data Access Protocols* de la subsección 2.1.1: TAP, SIA, SSA, SCS y SLAP. Además, PyVO proporciona acceso al Registry o Registro, permitiendo encontrar datos respecto a sus descripciones y/o protocolos de acceso.

2.1.3. Observatorio Virtual Chileno (ChiVO)

Siendo Chile uno de los países con mayor actividad astronómica en el mundo, es lógico que genere una cantidad considerable de datos. El problema surge cuando esos datos no tienen donde ser almacenados, o más aún, se desaprovechan. Por este motivo fue necesario que Chile tuviera su propio Observatorio Virtual, para esto existe ChiVO.

La iniciativa comenzó el 2013 luego de postular a financiación a través del Fondo de Fomento al Desarrollo Científico y Tecnológico (FONDEF) y se estableció como integrante oficial de la IVOA el mismo año. Es un esfuerzo por parte de la Universidad Técnica Federico Santa María y la Universidad de Chile, junto a Red Nacional de Investigación Chilena⁷ y ALMA. ChiVO obtiene sus datos desde el *Atacama Large Millimeter/submillimeter Array* (ALMA), el cual genera al rededor de 30TB de datos mensualmente. Si se toma en consideración que el proyecto empezó a funcionar desde el 2011, a día de hoy se han generado 144TB de información, los cuales deben ser almacenados y estar disponibles para su manipulación y análisis.

2.2. IA, Búsqueda Inteligente y *Machine Learning*

La Inteligencia Artificial o IA es un área de las ciencias de la computación que busca emular el razonamiento y comportamiento humano junto con la capacidad y velocidad de cómputo de las máquinas para facilitar procesos y tareas que, de otra forma, ocuparían mucho tiempo valioso de la gente o que el delegarlos a una máquina o programa mejoraría la calidad de vida de la persona. Cuando hablamos de IA, usualmente nos referimos a la Inteligencia Artificial Débil (*Weak AI*), estos son los programas o máquinas que tienen 1 objetivo específico, el cual logran de manera excepcional*. El otro tipo de IA es la Inteligencia Artificial General, la cual es una IA hipotética que podría emular o superar el pensamiento humano. El estudio de la IA General no está dentro de los objetivos de este trabajo, por lo que "IA" en este trabajo hará referencia a la IA débil.

⁶<https://pyvo.readthedocs.io/en/latest/index.html>

⁷<https://www.reuna.cl/somos/>

La IA tiene distintas aplicaciones y la podemos ver en el día a día en cada momento: Cuando nuestro telefono nos sugiere una alarma a cierta hora para despertar, cuando las puertas del metro se abren al nosotros acercarnos, cuando utilizamos alguna aplicación de mapas para llegar a nuestro destino al hablarle a nuestro asistente personal en nuestro *smart phone*, etc. Estos y otros ejemplos son logrados gracias a distintas áreas de la IA: Machine Learning, Redes Neuronales, Procesamiento de Lenguaje Natural, entre otras. Varias de estas áreas están implicadas en el desarrollo de una aplicación de búsqueda inteligente.

2.2.1. *Machine Learning* y Redes Neuronales

El **Machine Learning** (de ahora en adelante ML) o Aprendizaje de Máquina es un subconjunto de la IA que apunta a la optimización de procesos, predicción de comportamiento, identificación de imágenes, procesamientos de texto, entre otras cosas.

Está el ML clásico, el cual depende de la información humana que se le entregue. Estos modelos de ML tienden a ser entrenados en *datasets* etiquetados, lo que significa que cada dato que se le pase tiene definidas características que lo distinguen de otros datos o asemejan a otros datos. Es con estos datos que el modelo de ML puede aprender a distinguir imágenes, texto, clasificar ambos tipos, además de llegar a recomendaciones o resolver preguntas cuyas respuestas pueden ser clasificadas.

Otro tipo de ML es el *Deep Learning* o aprendizaje profundo. Este subconjunto de ML ha surgido en popularidad en la última y la década pasada, evidenciado no solo por los avances tecnológicos que se encuentran catalogados en esta área, si no la cantidad de trabajos y artículos científicos publicados en esta categoría, como en el archivo de artículos académicos ArXiv*, en donde la cantidad de artículos sobre *deep learning* pasó de 19,864 en la década del 2010 a 58,242 en lo que se lleva de la década de 2020 hasta la fecha*[2024]*[en los 2000 fueron 31].

El *deep learning* es utilizado para tareas de mayor complejidad que el ML clásico, como detectar fraudes, asistentes virtuales o hasta generar texto e imágenes. Esta capacidad se da gracias al **aprendizaje no supervisado**, lo que significa que, si bien puede hacer uso de *datasets* etiquetados, no los necesita. Esto abre una nivel de entrenamiento muy superior, ya que se podrá hacer uso de grandes repositorios de datos sin necesidad de tener una persona etiquetándolos (algo muy útil en la era del *Big Data*), además de poder utilizar datos no estructurados (esto es, datos como imágenes, documentos de texto, audio, etc.). La capacidad de aprender de enormes *datasets* es gracias al nivel de complejidad de estos modelos de ML, de ahí viene su nombre, ya que utiliza muchas capas de *neuronas* para poder extraer las características de los datos, clasificarlos y poder hacer predicciones, o inclusive generar datos. Estas neuronas son lo que se conoce como Redes Neuronales.

Las **Redes Neuronales** (RRNN) (o NN por sus siglas en inglés)* son el núcleo del funcionamiento detrás de los modelos de ML. Se llaman así porque emulan el funcionamiento de las

neuronas, pasando datos de una a otra, modificándola, para llegar a una conclusión o resultado. Cada neurona o “nodo” tiene un peso, que es la influencia que tiene esa neurona en el resultado final, y un valor de *threshold* o umbral, el cual de ser superado, envía datos a la neurona siguiente. Se entrenan utilizando 2 sets de información: un set de entrenamiento y otro de validación. El set de entrenamiento es bastante más grande que el de validación, el cuál guarda los resultados esperados de los datos a ingresar.

Todas las neuronas se agrupan en capas, y las RRNN están compuestas por 3:

- **Input Layer** o Capa de Entrada: Es la capa donde se le alimenta de datos a la red para que los procesos (los distintos datasets, etiquetados o no).
- **Hidden Layer** o Capa Oculta: Esta capa se llama así ya que su complejidad varia según el tipo de red que se esté usando. La complejidad de la red depende de la complejidad de la tarea para la que se entrena (e.g. la capa oculta de una red de clasificación de alimentos será más simple que la red que decifra el lenguaje humano).
- **Output Layer** o Capa de Salida: La capa de salida proporciona el resultado final de todo el procesamiento de datos que realiza la red neuronal artificial. Puede tener uno o varios nodos. Por ejemplo, si tenemos un problema de clasificación binaria (sí/no), la capa de salida tendrá un nodo de salida que dará como resultado 1 o 0. Sin embargo, si tenemos un problema de clasificación multiclase, la capa de salida puede estar formada por más de un nodo de salida.

Hay muchos tipos de redes neuronales. Algunas de estas son:

- **Red Convolucional (CNN):** Las Redes Convolucionales trabajan principalmente con imágenes, reconociendo patrones y clasificando, esto mediante un proceso de filtro y *pooling*. El filtro es cuando una matriz de números identifica sectores con características deseadas, y el *pooling* es cuando los datos se reducen en tamaño, para acelerar el procesamiento.. Esto se hace varias veces hasta que se obtenga el resultado esperado.
- **Red Recurrente (RNN):** Las Redes Recurrentes trabajan con datos secuenciales, lo que son vídeos o textos. Son capaces de procesar estos datos gracias a las “células recurrentes” que los componen. Estas pasan su output a otras células recurrentes, donde corrigen sus valores para aumentar la precisión de la red. Estas redes eran las principales encargadas de el procesamiento de texto y traducción de texto.

Las RNN han sido la arquitectura de red típica para la traducción durante bastante tiempo, ya que procesan el lenguaje secuencialmente de izquierda a derecha o de derecha a izquierda. Al leer una palabra cada vez, esto obliga a las RNN a realizar múltiples pasos para tomar decisiones que dependen de palabras muy alejadas entre sí. Investigaciones anteriores han demostrado que, a grandes rasgos, cuantos más pasos de este tipo requieran las decisiones,

más difícil le resultará a una red recurrente aprender a tomarlas [Uszkoreit, 2017]. Es por eso que las RNN han sido paulatinamente reemplazadas por otros modelos de ML con arquitecturas de redes neuronales distintas, como lo son los **modelos generativos**. Algunos ejemplos incluyen:

- Red Generativa Adversativa (GAN): Estas redes, al igual que los modelos de difusión, se enfocan en generar datos. Se componen de 2 redes, una genera resultados y la otra discrimina si estos datos son reales o no. La idea es que ambas redes se empujen la una a la otra para poder crear mejores datos o ser mejor al discernirlos.
- Modelos de Difusión (DDPM): Estos modelos se encargan de agregar ruido a imágenes y luego reconstruyen la imagen quitándolo, así, se entrenan para poder inclusive crear imágenes.
- Autoencoder Variacional (VAE): Similar al modelo DDPM, solo que en vez de agregar ruido, comprime y descomprime los datos. Son más rápidos, pero sacrifican calidad.

Otro tipo de red neuronal y modelo de ML que es ideal para analizar, comprender y generar texto son los Transformers. Estos modelos han crecido en popularidad exponencialmente en los últimos años, y prometen ser el siguiente gran paso de la IA.

2.2.2. Transformers y Procesamiento de Lenguaje Natural

Existen sistemas de inteligencia artificial que son capaces de mantener una conversación contigo, de responderte de forma natural y de realizar tareas que le pidas, desde darte información sobre un tema hasta escribirte una redacción o un poema sobre él, además de otras tareas. Estos sistemas están creados en base al modelo de “Deep Learning Transformers”.

Las **Transformer Network** son arquitecturas de redes neuronales similares a las RNN. Procesan datos secuenciales (como el texto o un vídeo) de manera no secuencial, gracias a 2 características propias: *self-attention* (atención) y *positional encodings* (codificación posicional).

La “atención” le da un peso a cada parte del input, los que cambian y se determinan a medida que la red se va entrenando. Esto se hace comparando valores que se le asignan a cada parte de input ya tokenizado (osea, transformado en algo más fácil de leer para la red). Estos valores son Query, Key y Value. Query es un valor que se compara con otras palabras, y así determinar su importancia. Luego, Query es comparado con Key para calcular su puntaje de atención. Una vez se determina la atención de cada dato, Value, que es el valor actual del dato, es modificado en base al valor de atención. Este proceso se repite múltiples veces por capa, hasta que se computa el valor final

Para poder procesar grandes volúmenes de datos, los transformers procesan datos (por ejemplo, palabras de una oración) en paralelo (de forma no-secuecnial). Eso significa que no entienden el orden de las palabras así como las CNN y las RNN. Para eso, antes de pasar los datos a la red, se les asigna una *codificación posicional*, lo que indica a la red la posición global o relativa de una palabra en una oración. Gracias a estos 2 componentes, los transformers pueden entender como las palabras se relacionan en un texto.

Las redes Transformer tienen una estructura Encoder-Decoder, lo que es, un bloque Encoder que baja el nivel de complejidad del input, tomando solo las partes relevantes, y un Decoder, que reconstruye el input “encriptado”, haciendo uso de los valores de atención calculados en las capas de atención.

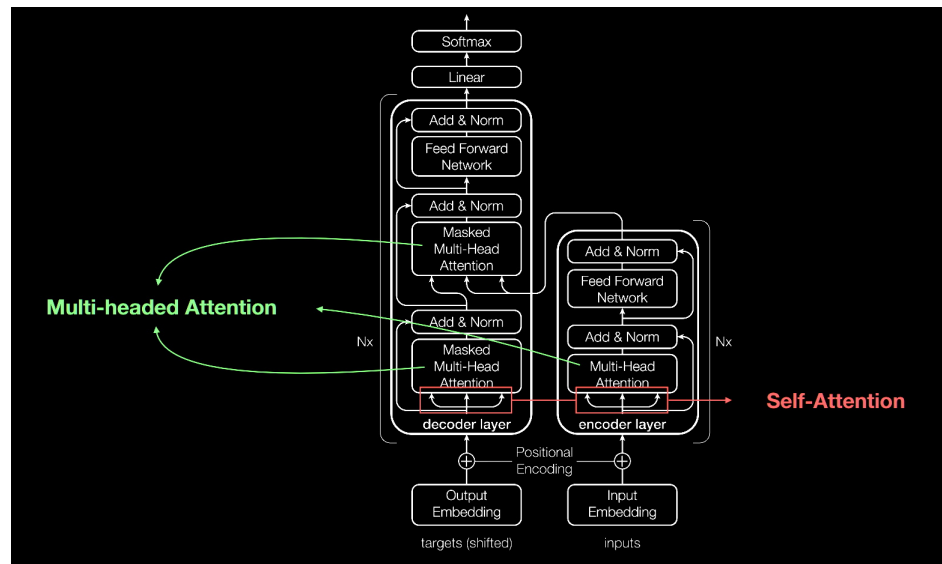


Figura 5: Arquitectura de un modelo Transformer.

Fuente: Nvidia.

En la figura 5 podemos ver, de abajo hacia arriba, los input y output “embeddings”, que son la representación de los datos para las redes, la codificación posicional, de izquierda a derecha la capa decodificadora y la codificadora, la atención de cada capa y la atención entre las capas. En la capa decodificadora, al final, se encuentran las funciones linear y softmax, que son las últimas capas en computar el output de la red.

Gracias al *positional encoding*, las capas de atención y su arquitectura Encoder-Decoder, los modelos Transformer son denominados “*foundation models*” o modelos fundacionales. Esto ya que permiten ser la base de muchas tareas de procesamiento de lenguaje natural (NLP), debido a su versatilidad y escalabilidad al poder adoptar diferentes “misiones” en distintas áreas de la ciencia y tecnología y de distintos niveles de complejidad, y su capacidad de ser pre-entrenados con grandes alijos de texto, lo que permite “afinarlos” a los requerimientos de la tare para los que se quiera usar. Uno de los mejores exponentes de los *foundational models* *Large Language Models*.

Large Language Models (LLM) o Modelos de Lenguaje Grandes son un tipo de modelo de *deep learning* con arquitectura basada en transformers. Estos modelos son entrenados con datasets de escalas del internet, con billones de parametros, mediante aprendizaje no-supervisado. Son capaces de generar textos, código, dibujar e inclusive mantener un dialogo con una persona. Por ejemplo, un LLM es capaz de aprender el lenguaje de las secuencias de proteínas que componen las vacunas, para luego ayudar a los científicos a encontrar nuevas y mejores curas y tratamientos. También pueden ser entrenados para servir a los consumidores de grandes marcas en forma de *chatbots*.

Ejemplos de LLM son el Megatron-Turing Natural Language Generation de Nvidia*, entrenado con más de 530 billones de parametros, y el famoso* modelo GPT *Generative Pre-trained Transformer* (3, 3.5 y 4) de OpenAI. Este último es un sistema que ha sido entrenado para mantener conversaciones con cualquier persona, y en cualquier idioma en el que le escribas. Sus algoritmos analizan las palabras que le escribes, entienden su orden, su sentido e interpretan lo que quieres decir, y luego generan una respuesta basada en la información con la que ha sido entrenada.

Una parte muy importante de lo que define a un *foundational model* como lo son los LLM es la capacidad de *Zero-shot learning*. El *Zero-shot* o *many-shot learning* es la capacidad que tienen los LLM de generar texto sin mayor entrenamientos. Basta con darle unas simples instrucciones en lenguaje natural y, debido a su entrenamiento previo, es capaz de entender estas instrucciones y corregir su funcionamiento acorde a las mismas. Si se quiere mejorar aún más la especificidad del modelo, así como su precisión, se utiliza *fine-tuning* o, más recientemente, *prompt-tuning*.

Mientras el fine-tuning implica entregar una gran cantidad de ejemplos etiquetados al modelo de ML para entrenarlo brevemente, el *prompt-tuning* y el *prompt-engineering* implican darle contexto al modelo para que pueda hacer mejor una tarea específica. El *prompt-engineering* funciona de la misma forma que el *zero/many-shot learning* descrito previamente, entregando directamente al modelo uno o más “*prompts*” como ejemplos y contexto, estos *prompts* directos son llamados “*hard-prompts*”. Para tareas más complejas se utilizan “*soft-prompts*”, *prompts* diseñadas por IA. Estos *soft-prompts* nos son comprensibles por personas, ya que están en forma de *embeddings* (*strings* o “cadenas” de números).

Gracias a esta forma de *prompt-tuning*, los modelos son capaces de ser entrenados para distintas tareas y en el momento (lo que se llama *multi-task learning*). El aspecto en el que el *prompt-tuning* es inferior al *prompt-engineering* es que el LLM no puede explicar su comportamiento al usuario, además que el *prompt-engineering*, por ser humanamente legible, da mucha más flexibilidad al momento de “afinar” un LLM.

Tomando en cuenta la arquitectura encoder-decoder de los transformers, hay tres casos de LLM:

- *Encoder Only*: Son modelos que se limitan a la clasificación y el análisis de datos.
- *Decoder Only*: Estos están orientados específicamente a la generación de datos.

- *Encoder-decoder*: Los LLM con ambos tipos de bloque son capaces de comprender texto y traducirlo, además de su capacidad de síntesis y dialogo.

2.3. Competencias y Otras herramientas Educativas

El EuroVO, para cumplir con sus objetivos, ofrece 2 software los cuales serán considerados como competencia de la aplicación a diseñar; Stellarium y Aladin.

2.3.1. Stellarium

Stellarium⁸ es un software astronómico que simula el cielo , todas sus estrellas, constelaciones, planetas y otros cuerpos celestes y estructuras cósmicas. El software es *open source* y funciona como aplicación de escritorio, aplicación web y aplicación móvil, para iOS y Android.

La interfaz de la aplicación de escritorio puede verse en la siguientes figuras:



Figura 6: Interfaz web de Stellarium.

Fuente: Stellarium desktop.

En la figura 6, se puede ver la pantalla principal de Stellarium. Se presenta un domo celestial en el cual están visibles los distintos objetos celestes. Arrastrando el ratón por la pantalla se puede rotar la bóveda celeste, la cual cambia con el tiempo, el cual se puede acelerar

⁸<https://stellarium.org/>

o pausar. Se pueden activar o desactivar características como las líneas azimutales, los objetos celestes profundos, planetas, la atmósfera, se puede modificar la cámara, entre otras características.

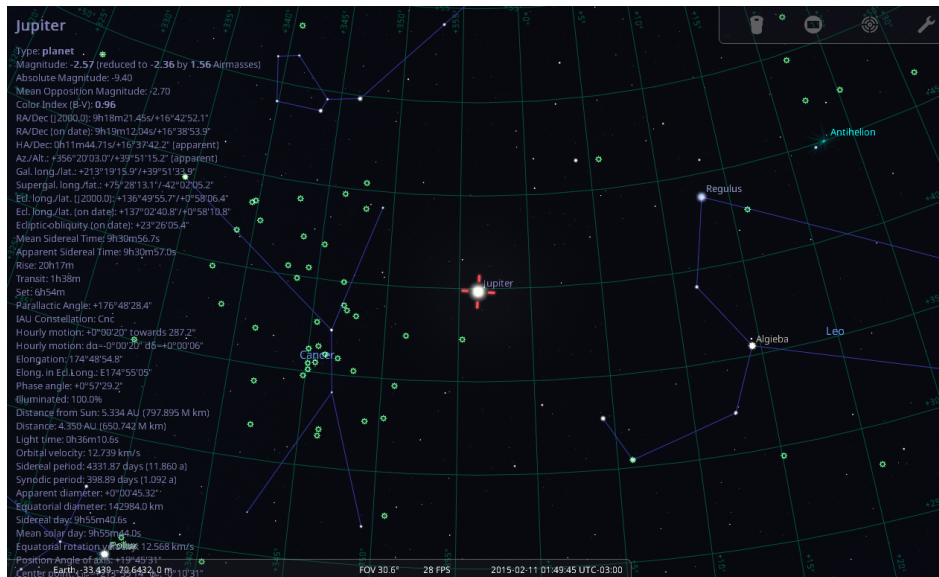


Figura 7: Interfaz web de Stellarium.

Fuente: Stellarium desktop.

En la figura 7 se puede ver que, al seleccionar cualquier objeto celeste, se presenta toda su información a la derecha, como: título, coordenadas, magnitud, índice de color, distancia del sol, velocidad orbital, etc.

Por último, en la figura 8 se ven varias otras de las características de Stellarium, como lo son la localización, la cual cambia la posición de los objetos en la bóveda, opciones del cielo, el sistema solar, objetos de catálogos estelares, censos astronómicos e inclusive cambiar las etiquetas según la cultura espacial, la que incluye distintas civilizaciones.



Figura 8: Interfaz web de Stellarium.
Fuente: Stellarium desktop.

2.3.2. Aladin

Aladin es un atlas celestial interactivo que permite al usuario visualizar y manipular imágenes astronómicas digitalizadas y acceder de forma interactiva a datos relacionados entre sí.

La visualización de datos en Aladin está compuesta por 4 partes:

- El Árbol de colecciones. Donde se muestran las más de 20,000 colecciones de datos
- El Stack o “pila”. Aquí aparece información respecto lo que se está trabajando.
- La Vista. Despliega la porción de la imagen de acuerdo al zoom y superimpone símbolos gráficos correspondientes a los objetos astronómicos en las tablas y catálogos seleccionados
- Las Medidas. Muestra los parámetros de los objetos astronómicos seleccionados en la vista.

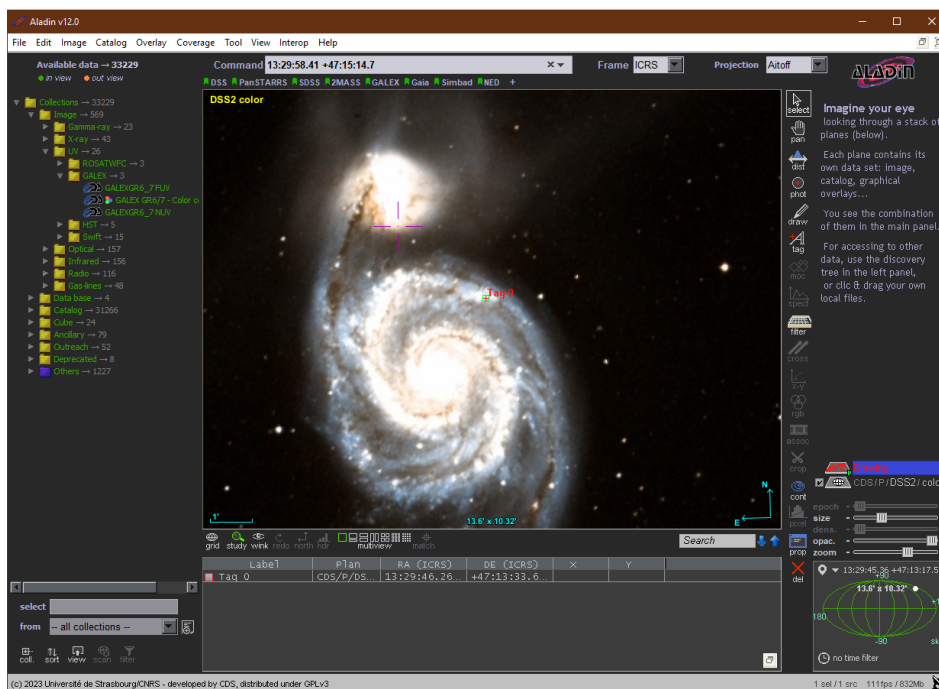


Figura 9: Interfaz principa de Aladin. De Izquierda a derecha, arriba a abajo son: Árbol de colecciones, Vista, Stack, Medidas

Fuente: Aladin desktop.

Aladin tiene soporte para 3 tipos de datos principales: Imagen, Catalogo y *Graphical Overlays*

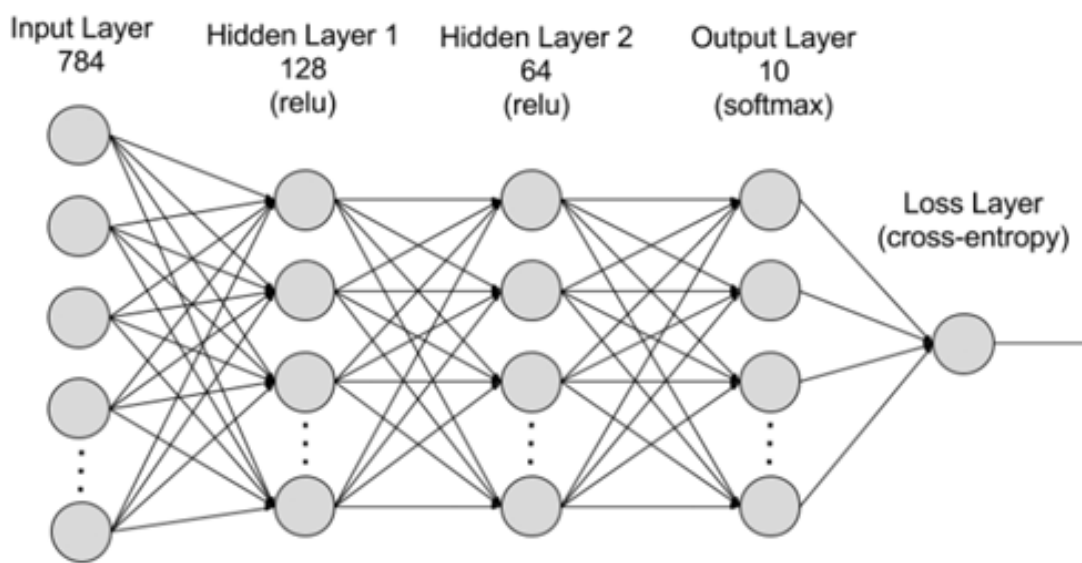


Figura 10: Representación de alto nivel de una Red Neuronal Recurrente.
Fuente: Fuente: AWS.

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Para remediar la situación de ignorancia respecto al verdadero funcionamiento y la realidad de los datos que se manejan en astronomía y sus otras ramas por parte de el público general, sumado a un desinterés por parte de los profesionales del área en la utilización del Observatorio Virtual, se propone como solución una interfaz de búsqueda asistida para el VO, la que pueda dialogar con el usuario en respecto a materia de astronomía, esté versada en esta área y pueda ejecutar consultas al VO sin necesidad de conocer lenguajes específicos de consulta o **query** (i.e. ADQL[naki Ortiz, 2008]).

3.1. Investigación de posibles soluciones

En primera instancia, es necesario investigar posibles soluciones y acercamientos en las que se puede lograr el objetivo final, con tal de escoger la más adecuada.

Indagando respecto formas de crear un “asistente de búsqueda” o una “aplicación de búsqueda inteligente” se obtuvieron las siguientes opciones:

- Aplicar *fine-tuning* a un modelo pre-entrenado (*foundational models*[Murphy, 2023])

y adecuarlo al comportamiento de un asistente para el VO.

- Utilizar **IBM WatsonX**[IBM, 2024] para crear un *AI Assistant* con acceso a toda la documentación de IVOA y PyVO.
- Utilizar **LangChain**[LangChain, a] para crear un *chatbot* con **Retrieval Augmented Generation** (RAG)[Merritt, 2024]

La primera opción fue descartada debido a la dificultad de adquisición de datos de entrenamiento. Como se indica en la introducción, el VO no es utilizado ampliamente por astrónomos, mucho menos por legos, por lo mismo el reunir ejemplos de uso del observatorio virtual es difícil, así como lograr una conexión con el VO sin ayuda de *frameworks* ni librerías pre-existentes, ya que la dificultad de esa tarea sería propia de un trabajo de título aparte.

IBM Watson.x promete una “plataforma de desarrollo para IA”[IBM, 2024], la cual está dotada de múltiples modelos fundacionales[Murphy, 2023] y herramientas para crear asistentes corporativos. Pero ahí es donde está la principal razón para no utilizarlo. IBM Watson.x está orientado a crear asistentes de documentación para optimizar el trabajo de empleados de una organización, así como chatbots para interactuar con los clientes. No está alineado con objetivos de descubrimiento de data e interacción con sistemas de query[naki Ortiz, 2008].

Finalmente queda **LangChain**. Langchain es “un framework para el desarrollo de aplicaciones, motorizado por LLMs”[LangChain, a]. Este framework entrega las facilidades para poder estructurar una aplicación que incluya un LLM, pero no se limite a ser un chatbot, gracias a su sistema de *tools* o herramientas, las cuales permiten al modelo (LLM) convertirse en un “Agente”[Yao et al., 2022] capaz de ejecutar código según estime sea necesario para poder responder los requerimientos del usuario. Para crear agentes complejos, que puedan asistirse de múltiples herramientas e interactúen con el usuario de formas específicas existe otro framework llamado **Langgraph**[LangChain, b]. Langgraph es similar a LangChain en el hecho que entrega facilidades para crear cadenas de LLM, pero es más un orquestador de flujos de trabajo de agentes. Estos frameworks se complementan, LangChain entregando las herramientas de alto nivel para manejar LLMs, y LangGraph las de bajo nivel, para coordinar el comportamiento de un agente.

Langchain ofrece un sistema de “bloques” y componentes, los cuales se estructuran en una “cadena”, lo cual permite integrar distintas capacidades a la aplicación, según sea necesario. Adicionalmente, posee un sistema de *tracing* o seguimiento llamado **LangSmith**[LangChain, c], en el cuál se puede revisar el comportamiento del modelo y la “cadena” construida, mostrando todas sus llamadas y los argumentos con cual ejecuta cada componente de la cadena.

Dentro de los componentes y “bloques” que ofrece LangChain, están:

- **LLMs**: Los ya discutidos *Large Language Models*. Es el componente principal de la aplicación, que interactúa con el usuario directamente y orquesta al funcionamiento del

Agente.

- **Prompts:** Instrucciones previas que se le entregan al modelo (LLM) para definir su interacción con el usuario, condicionar su comportamiento y incluso restringir de cierta forma las respuestas que pueda dar.
- **Parsers:** Componente que directamente modifica y reestructura la respuesta que el modelo le entrega al usuario o envía a otros componentes.
- **Streaming:** Es en recibir *output* por parte del LLM de forma progresiva. En vez de recibir solamente la última respuesta del LLM o estado del Agente, entrega cada paso de la ejecución del programa y su *metadata* asociada: mensajes del usuario, mensajes de la IA (LLM o Agente) y mensajes de herramientas
- **Memoria:** En la creación de agentes hay 2 tipos de memoria. La memoria a corto plazo y a largo plazo. La memoria a corto plazo se maneja con el estado del agente, el que puede estar compuesto como uno estime conveniente, y asignado a un *thread* o hilo. Por otro lado, la memoria a largo plazo corresponde más a información que el modelo deba recordar independiente del hilo de conversación. Los mensajes o información se vecotiza y pasa a una *vector store*, conceptos que se discutirán más adelante
- **Document loaders:** Cargan documentos a la memoria del agente. Ya sea para responder preguntas directamente o para obtener una respuesta compuesta de varios documentos.
- **Retrievers:** Métodos que “recuperan” documentos relevantes para la solicitud actual. La solicitud proviene del LLM, el cual analiza la solicitud del usuario y entrega los requisitos bajo los cuales el *retriever* debe extraer documentos. Estos documentos no son específicamente archivos completos, sino subdivisiones de estos, creados a partir de *text splitters* o separadores de texto.
- **Text splitters:** Se preocupan de subdividir textos (mensajes o documentos). El como lo hacen depende del algoritmo que se decida utilizar.
- **Embedding models:** Estos modelos transforman documentos en *embeddings*, lo que son representaciones compactas de los documentos ilegibles para el usuario pero optimizadas para los modelos de lenguaje. Los embeddings utilizados actualmente son representados como vectores, cuyo valor es comparable con el de una solicitud o contexto entregado por el usuario, y así los retrievers saben que documento recuperar
- **Vector stores:** Estas son bases de datos optimizadas para el almacenamiento de embeddings y documentos.

Y el componente o utilidad más importante para el desarrollo de un asistente (agente) para el VO, las **herramientas**. El sistema de creación de herramientas permite utilizar algunas ya incluidas dentro de las librerías de LangChain o crear una personalizada. El sistema de

personalización permite crear herramientas a base de funciones, las cuáles han de tener un tipado fuerte en los argumentos que acepta la función, así como una descripción que permita al LLM entender en que situaciones debe llamar a esta herramienta. Esto se puede lograr de distintas formas, incluidas el documentar cada función detalladamente o crear una clase con la librería Pydantic[Pydantic,] la cual contenga las variables, funciones, descripciones y métodos de la herramienta.

Una desventaja de este sistema es que es relativamente limitante, debido a que cualquier argumento que se le entregue a la herramienta debe ser un tipo numérico (int), texto (string), o un diccionario⁹, cuyos valores deben ser de los mismos tipos anteriores u otro diccionario. Además, todo output que entregue la herramienta debe ser serializable como un JSON. Ambas limitantes se deben a que el LLM solo recibe y entrega texto, ya que un LLM a grandes rasgos es una función que “recibe un token y predice el siguiente”, no posee un entorno de ejecución ni interprete.

Para poder crear un asistente para el VO que responda preguntas en conversación mientras puede acceder a información de los VO de forma condicional, no basta solo con las herramientas de LangChain. Es necesario poder controlar el flujo del programa de forma condicional, y poder manejar cualquier error que surja. Para esta situación es que utilizaremos LangGraph.

Langgraph ofrece un framework para poder controlar el flujo de la aplicación basada en LLM: ciclos de trabajo, persistencia de memoria y “estado”, manejo de errores, soporte para *streaming*. Al igual que LangChain, LangGraph ofrece herramientas, flujos y agentes prefabricados, y al igual que LangChain, ofrece un alto nivel de personalización de estos componentes. Uno de los más importantes es el **ReAct Agent**.

El **ReAct Agent**[Yao et al., 2022] es un framework y un paradigma para la construcción de aplicaciones asistidas por LLM. La sigla proviene de las palabras *Reasoning*, que significa “razonar” y *Acting*, que significa “actuar”. Este nombre es una buena descripción del objetivo del paradigma, el cual es el combinar el raciocinio de un LLM con la capacidad de actuar del código. De esta forma se utiliza el LLM como núcleo de raciocinio para la toma de decisiones, las cuales son ejecutadas por funciones diseñadas para los distintos requerimientos.

Tomando todo lo que ofrece LangChain y LangGraph, se decidió que la solución será un “Agente ReAct con capacidad de *Retrieval Augmented Generation*”. Esta solución ofrece la posibilidad de versar al LLM en materias más recientes de ciencias y, más específicamente, astronomía y astrofísica, además de entregarle la capacidad al agente de ejecutar funciones que permitan comunicarse con los distintos Observatorios Virtuales.

⁹Son objetos que representan pares de llave y valor. Se accede a uno de los valores del diccionario pasándole la llave respectiva

3.2. Definición de arquitectura de la solución

Ya decidida la solución que se implementara, se procedió a investigar y definir el stack de servicios, frameworks y librerías a utilizar para la creación de la aplicación.

El agente necesitara un LLM capaz de llamar herramientas tanto como mantener una conversación. Langchain ofrece múltiples conexiones con APIs de distintos proveedores. Haciendo una investigación sucinta se optó por utilizar la API de OpenAI y su modelo GPT-4. En caso del LLM se optó por OpenAI por familiaridad, más adelante se podría hacer un estudio respecto al desempeño de la aplicación dependiendo del modelo que se use como núcleo. Pero también es necesario escoger un modelo para la creación de los embeddings. Al igual que para el LLM, se optó por utilizar la API de OpenAI. Comparado con otras opciones como Cohere o Anthropic, OpenAI posee un desempeño similar, inclusive inferior en ciertos idiomas[Wiik, 2024], pero como ya se está utilizando OpenAI para el LLM, por consistencia, también lo utilizaremos para embeddings.

Para la conexión con el VO se seguirá utilizando **PyVO**[Developers, a] por estar basada en Python, su excelente documentación y constante mantenimiento, además de pertenecer a la colección de paquetes **Astropy**[Team, 2023] que tiene múltiples utilidades para tratar con unidades astronómicas, tipado y manejo de archivos como FITS[Hanisch et al., 2001].

Siguiente, es necesario un almacenamiento en la nube para los documentos que utilizará el agente para responder preguntas específicas, así como las embeddings que correspondan a esos documentos y los historiales y estados de las distintas conversaciones que distintos usuarios puedan sostener. Para este requisito se decidió utilizar **Qdrant Vector Database**[Qdrant,]. El servicio *cloud* que ofrece Qdrant tiene una interfaz intuitiva y fácil de usar. Uno de sus puntos fuertes es su capacidad de aislamiento de datos, lo que facilita el tener múltiples *threads* (o hilos) para distintos usuarios y sus historiales. Otra ventaja es su alta escalabilidad y su método de **cuantización binaria**[Kasliwal, 2023], lo que reduce requerimientos de almacenamiento y acelera operaciones de búsqueda.

La aplicación deberá estar integrada con una vista que pueda utilizar el usuario para poder sostener una conversación con el asistente, así como visualizar los datos que entregue, posiblemente en tablas o como imágenes. Comparando los frameworks más populares hasta la fecha[Mhatre, 2023] se llegó a 3 opciones: React, Vite y Vue. Se optó por **Vite**[Contributors,] por temas de velocidad de compilación. Además, debido al enfoque a legos de la aplicación y futura implementación para estudiantes, una visualización atractiva y dinámica se estimó necesaria. Librerías de estilización que se tomaron en cuenta fueron Material UI, Bootstrap y Tailwind CSS. El objetivo de este trabajo no es medir efectividad de distintas interfaces gráficas ni crear un estilo propio, así que el estilo pre-configurado de Bootstrap y MUI es preferible. Entre las dos opciones, MUI es una librería más ligera que bootstrap debido a su modularidad, así que se optó últimamente por **Material UI**[SAS,].

Finalmente, es necesaria una forma de conectar el procesamiento y actuar del LLM con las

visualización del front-end. Se tomó en consideración utilizar Chainlit para esta tarea, pero durante el desarrollo se encontraron problemas para mantener de forma simultanea personalización e integración con un back-end de basado en Python. Debido a estas necesidades se optó por utilizar una librería más simple, pero robusta, como lo es **FastAPI**[Tiangolo,]. FastAPI otorga facilidad de uso, es completamente basado en Python y está altamente integrado con Pydantic, librería que ofrece clases, y métodos para la validación de tipos, funcionalidad muy necesaria para que el agente pueda utilizar correctamente las herramientas a crear. Además, ofrece seguridad y autenticación mediante JWT¹⁰.

3.3. Desarrollo del asistente virtual

Para el desarrollo del asistente virtual y futuro “Agente con Herramientas” fue necesario el estudio del framework, factible gracias a cursos en DeepLearningAI[Harrison Chase,], impartidos por Harrison Chase, co-fundador de LangChain. Luego de los cursos introductorios y con mayor familiaridad, se continuó con los tutoriales proporcionados en la página de LangChain y LangGraph, los cuáles implicaron la creación de un LLM prototípico con funcionalidad de RAG y la creación de herramientas personalizadas y un agente capaz de ejecutarlas. Con esto, es posible iniciar el desarrollo y configuración del agente.

3.3.1. Bases del agente

Para la creación del agente se inició con la creación de un chatbot básico para poder interactuar con el usuario.

Langgraph se basa en el concepto de un grafo para la creación de agentes (el chatbot inicial, aunque simple, es un agente). Estos grafos se componen de *nodes*, *edges* y *state* o nodos, bordes y un estado.

El primer paso es el *StateGraph*, *state* o estado del grafo. Este *State* es lo que convierte a la aplicación en una “máquina de estado”. El State de la aplicación es una estructura de datos que representa el estado actual de la aplicación. La información que se guarda en el estado se puede definir según necesidad, pero para el correcto funcionamiento de un agente, es necesario guardar una lista con los mensajes de la sesión actual, lo que vendría traducándose en una **memoria a corto plazo**.

Para la implementación del State del Agente, se utilizará una clase basada en el tipo TypedDict de la librería typing_extensions. La lista de mensajes se definirá con Annotated de la librería typing. Esto nos permite darle definición y, en este caso, funciones para añadir los mensajes, como lo es `add_messages` de `langgraph.graph.message`

¹⁰Jason Web Tokens: Son claves encriptadas que otorgan acceso temporalmente a sitios web

Ya con el State definido se puede crear el StateGraph, que es efectivamente la aplicación que será utilizada por el usuario, aunque aún está incompleta.

Para continuar construyendo el StateGraph es necesario agregar nodos. El primero de estos es el LLM que tomará las decisiones de el Agente. Como fue discutido previamente, se utilizará el modelo de OpenAI, particularmente el **gpt-4o-mini**. Se optó por este modelo porque, si bien su poder de procesamiento es inferior al **gpt-4o**, es más rápido y menos costoso. Dado que durante el desarrollo de este asistente se hicieron múltiples pruebas y la tarea principal del LLM es de interlocutor y toma de decisiones, no es necesario un gran poder de computo.

Los nodos en LangGraph están hechos en base a funciones, por lo que se definirá la función `call_model`, la que recibe como argumento el State. Dentro de la función crearemos un *prompt* a partir de la clase `langchain_core.messages.SystemMessage`. El siguiente prompt será utilizado:

```
You are an assistant to astronomers and people interested in
astronomy. Your speciality is aswering questions related to the
virtual Observatory and return data from it using the tools you
will be provided with. When a tool of this is executed, it will
respond to you with a success message describing the state of
success or an error message that states why it failed. When
successful, the tool will create data and store it in the state
of this application, which is you do not have access to. DO NOT
CREATE YOUR OWN ANSWER if you called a tool, just limit yourself
to inform the user of the success of the query and any desription
the tool gave you.
```

La elección de un prompt en inglés es para que el agente sea consistente con la nomenclatura utilizada en el observatorio virtual. Más adelante se podría implementar en el LLM un verificador de nombres, cosa que si se da una instrucción similar a “Quiero imágenes de Saturno” el Agente utilice el argumento “Saturn”.

Una vez definido el prompt, se invoca al LLM con el método `.invoke(input, config)`. El `input` es compuesto por el prompt inicial más los mensajes que ha recibido el Agente hasta el momento, incluido el actual mensaje con el que se invoca. El argumento `config` es un objeto con la siguiente estructura:

```
config = {
    "configurable":
        {
            "thread_id": "abc123"
        }
}
```

El valor `thread_id` es el identificador de el *thread* o hilo que guarda la memoria del chat.

Esto nos permite aislar las conversaciones para que el usuario pueda sostener múltiples y las conversaciones de distintos usuarios no se entrelacen.

El valor de retorno de la función del nodo, y de todas las funciones que sean nodos, debe ser una actualización del State en forma de un diccionario que contenga las propiedades a ser actualizadas con el valor respectivo. En este caso se pasa un diccionario con la propiedad de mensajes y el valor de la respuesta en una lista. Al pasar como retorno la respuesta en una lista, hacemos uso de la función `add_messages`, para agregar la respuesta a la ya existente lista de mensajes del State.

Conectar el nodo al StateGraph se hace mediante el metodo `add_node(node_name, node_function)`. El nombre del nodo será `"agent"` y la función es `call_model`.

Ya con un nodo presente en el StateGraph, es necesario un *entry point* o punto inicial, el cual será `"agent"` y se *setea* mediante el método `.set_entry_point(node_name)`. A su vez, es necesario un *finish point* o punto final, que se setea con el método y argumentos `.add_edge("agent", END)`, siendo `END` una constante importada desde `langgraph.graph`.

Por último, se compila el StateGraph con el método `.compile()`, pasando como argumento un *checkpointer*, el cual podemos obtener de a clase `MemorySaver` de `LangGraph`. Sin esto, la conversación no se asignaría al *thread*.

Con la configuración descrita, la representación visual del StateGraph es la siguiente¹¹

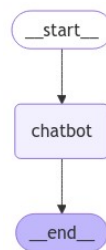


Figura 11: StateGraph de chatbot inicial.

Fuente: Elaboración propia.

3.3.2. Comunicación con el front-end

Para poder establecer la interfaz de comunicación con el chatbot rudimentario, posterior Agente, es necesario crear el *endpoint* [Goodwin, 2024] para invocar al StateGraph. Como se discutió previamente, se utilizará FastAPI para ello y se hará en un archivo distinto al del Agente.

Primero, el servicio se crea utilizando la clase `FastAPI(title, version, description)`,

donde los valores quedan a juicio de quién implemente el servicio.

Se declaran los orígenes permitidos mediante un *middleware*¹¹ y el mecanismo CORS¹² en una lista. Es imperante colocar la URL donde se levantará el front-end.

Para agregar los orígenes y establecer la configuración, se recomienda los siguientes valores:

```
app.add_middleware(      # Se agrega un middleware
    CORSMiddleware,      # El middleware es CORS*
    allow_origins=origins, # Lista de orígenes permitidos
    allow_credentials=True, # Permitir credenciales para identificación
    allow_methods=["*"],   # Metodos permitidos: todos
    allow_headers=["*"],   # headers permitidos: todos
)
```

Los métodos son las formas de comunicación HTTP que tienen las aplicaciones, estos definen que tipo de solicitud es y que tipo de respuesta se espera. Los *headers* o encabezados son metadatos que definen ciertas características del mensaje y de quien hace la petición.

Se crea un endpoint para poder comunicarse con el agente y se utiliza método POST. Este método se utiliza para generar cambios en una base de datos o si es necesario enviar información sensible en el cuerpo y cabeceras de la petición. Como no solo vamos a pedir información, si no que vamos a enviar un mensaje que alterará el historial e conversación y espera una respuesta, el método POST es adecuado.

También es necesaria una función que pase los mensajes del usuario y devuelva la respuesta del Agente, para pasarla como respuesta a la petición HTTP. Esta función se declarará en el archivo del Agente, y debe recibir el mensaje del usuario como argumento y entregar la respuesta del modelo como valor de retorno (es preferible utilizar una invocación asíncrona, debido a la naturaleza de las solicitudes a el observatorio virtual. Una implementación asíncrona también implicaría que las funciones de los endpoints deben ser asíncronos.

En esta etapa, el chatbot resultante y el servicio para consultarlo pueden ser probados realizando consultas HTTP con programas similares a Postman[Postman,].

3.3.3. Definición de herramientas (*tools*), estructura de comunicación e implementación

Para agregar funcionalidad e interacción con el VO es necesario crear y otorgar acceso al Agente a herramientas especializadas. Langchain dispone de herramientas pre-diseñadas para tareas simples como consultar el clima o buscar por la web, pero debida a la especificidad de la tarea, es necesario crear herramienta personalizadas.

Una herramienta o *tool* se compone de:

¹¹Software con el que las diferentes aplicaciones se comunican entre sí

¹²CORS es una funcionalidad de seguridad implementada por navegadores web, la cual restringe que aplicaciones web que se ejecuten bajo cierto dominio puedan hacer peticiones hacia otros dominios.

- **Función:** esta función es, en resumen, la herramienta. Es el código que ejecutará el Agente cuando el modelo estime conveniente. Para un mejor comportamiento del Agente, el tipo de retorno debe ser especificado, agregando al final de la función y antes del doble punto “->tipo”.
- **Decorador de función:** El decorador `@tool()` de la función sirve para darle las propiedades de herramienta a la función. Además, se puede especificar el nombre de la herramienta, si no se quiere usar el de la función, el schema y el tipo de descripción, que se explicarán en los siguientes puntos.
- **Descripción de la función:** Esta descripción es lo que le permite saber al Agente la utilidad y especificaciones de la herramienta. Es de crítica importancia ser muy explícito al momento de escribirla, sin especificar muchos detalles técnicos ya que, para efectos del Agente, las herramientas son “cajas negras”¹³ con instrucciones.
- **Schema de los argumentos:** Junto con la descripción es el componente más importante de una herramienta, ya que permite al Agente saber que tipo de datos debe entregar a la función, el como debe “traducir” la información entregada por el usuario a argumentos útiles. El *schema* se puede definir de varias formas. En este caso, se utilizará *docstring* para las descripciones u schemas más simples, y una *InputClass* será añadida en caso de argumentos más complejos.

Para poder otorgar al Agente el mayor acceso posible a los datos e información del VO que proporciona el framework PyVO, serán definidas 5 herramientas, cada una con acceso a: *Registry*, *Simple Image Access*, *Simple Spectral Access*, *Simple Cone Search* y *Simple Line Access*.

Registry

Para crear la herramienta que permite comunicarse con el Registry, primero serán definidos los argumentos y el tipo de retorno.

Según la documentación de PyVO[Developers, b], la forma principal de comunicarse con el Registry es mediante `pyvo.registry.search(args)`. Este método puede recibir 10 tipos de argumentos, de los que serán utilizados 6:

- **keywords:** Palabras clave que se usan para filtrar resultados. Debido a la interfaz del Registry por parte de IVOA, deben estar en inglés. Ejemplos son “Sun”, “white dwarf”, “black hole”, “magnetar”, etc. El tipo de este argumento será una lista de strings indefinidos.

¹³El concepto de “caja negra” quiere decir que el usuario (o en este caso, el Agente) desconoce el funcionamiento interno de la función o herramienta, y solo sabe que dato entregar y que esperar a cambio.

- **servicetype:** El tipo de servicio que proporciona el recurso para acceder a sus datos. Sus valores fijos son (agrupados por equivalencia): ('sia', 'sia1'), 'sia2', ('ssa', 'ssap'), ('scs', 'conesearch'), 'line' y ('tap', 'table'). El tipo de este argumento será una lista de strings de valores predefinidos.
- **waveband:** La ubicación espectral de los datos que proporciona el recurso. De acuerdo con IVOA, los posibles valores son: 'EUV', 'Gamma-ray', 'Infrared', 'Millimeter', 'Neutrino', 'Optical', 'Photon', 'Radio', 'UV', 'X-ray'. El tipo de este argumento será una lista de strings de valores predefinidos.
- **author:** Un creador cuyo trabajo este incluido en el recurso. El tipo de este argumento será un string indefinido.
- **ivoid:** Id del recurso. El tipo de este argumento será una string de valor indefinido.
- **temporal:** Rango de fechas en el que se ubican los datos del recurso. El tipo de este argumento será una string cuyo valor debe calzar con una expresión regular¹⁴ que se adecúe al formato de fecha ISO-8601

De estos argumentos, solamente las keywords **no son opcionales**, por definición de PyVO. Los argumentos y sus tipos deben ser especificados en la declaración de la función, así como dentro del decorador `@tool()`. Esto se puede hacer mediante una clase de python que herede del `BaseModel` de Pydantic.

El retorno de la función `.search()` es un arreglo de recursos; entradas de una tabla, donde cada una corresponde a un recurso del Registry, el cual es descrito con columnas como: título, descripción, ancho de banda, fecha de creación, links de referencia, etc.

Para mejor estructuración de la respuesta, se crea una clase que hereda de `BaseModel`; `RegistryResource`, la cual tiene dos atributos: una respuesta en texto que describe la cantidad de resultados obtenidos y los datos obtenidos de la consulta al servicio (en este caso, el Registry). El primer atributo será directamente alimentado al LLM del Agente, para que pueda generar una respuesta que acompañe a los resultados. El segundo atributo será definido mediante una clase que hereda del tipo `TypeDict` de nombre `VoResource`, cuyos atributos corresponden a toda la información que se quiera devolver por cada recurso obtenido del Registry.

La implementación de la función empieza creando un diccionario de Python cuyas claves corresponden a los nombres de los argumentos para la función `pyvo.registry.search()` y sus valores los entregados en los argumentos de la función. Luego, mediante desempaqueado de diccionario, los argumentos son pasados a la función `.search()`. De esta forma, si uno de los argumentos no es especificado, la función de búsqueda no lo tomará en cuenta. Los resultados de la búsqueda son pasados formato tabla con el método `to_table()`, especificando las columnas deseadas. Se itera sobre los resultados para extraer los modos de

¹⁴Las expresiones regulares son combinaciones de caracteres especiales que sirven como un molde para verificar si una string calza con la especificación.

acceso y las URLs de acceso con sus respectivos métodos, para luego agregar los resultados a una lista en forma de VoResource.

Por último, se genera una respuesta semántica que describa brevemente los recursos encontrados y se crea la respuesta de tipo RegistryResponse con la respuesta semántica y la lista de VoResources, para finalmente retornar este objeto.

Para la implementación de esta herramienta, la descripción fue la siguiente:

```

"""This function takes in constraints that are passed to a query
    against the virtual Observatory's Registry, a collection of
    metadata records of the resources in the VO.
The query returns a set of VO registry resources whose files and
    content match the given constraints.
If you are asked to search the registry, to provide "all data related
    to" specific objects or type of objects,
or if the request of the user doesn't specify from which service
    should you pull this information, use this tool.

return:
    A JSON serializable object that contains a semantic response for
    the Agent to read and pure data that is given straight to the
    user.
"""

```

Simple Image Access

Como interfaz para acceder al Simple Image Access se optó por 2 funciones:

La primera función, query_sia(), recibe cuatro argumentos necesarios para consultar el Registry por fuentes de datos gráficos, ya sea imágenes en distintos formatos o archivos FITS. Luego busca en cada una de estas fuentes hasta encontrar resultados, que finalmente retorna en un objeto VoToolResponse, el cual está compuesto por dos atributos: una respuesta semántica para alimentar al LLM (cosa que genere un repuesta para el usuario) y los datos tabulares de estos recursos obtenidos como resultado de la query al servicio.

Los argumentos son:

- Posición [str]: El nombre de un objeto astronómico del cuál se quiere obtener datos gráficos o en formato FITS. Este nombre será buscado en base de datos astronómicas para intentar obtener sus coordenadas celestiales.
- Área [float]: El tamaño del área alrededor del objeto en la cual deben ubicarse los datos que devuelva la consulta.

- Unidad [str]: la unidad del área de búsqueda.
- URL [str]: El URL del servicio en el que se quiere buscar datos. Es un valor opcional y de no ser proporcionado se buscará en el Registry los servicios disponibles

El proceso para obtener los datos empieza obteniendo los URL de los servicios a consultar. Si uno fue proporcionado en los argumentos, se mete en una lista unitaria y se prosigue. De otra forma, se realiza una búsqueda en el Registry de todos los resultados para el `servicetype = 'sia'`. Estos resultados se pasan a formato tabla, se extraen los valores de la columna `access_urls` y se guardan en una lista que luego se randomiza.

En un bloque `try-except` se intenta obtener la posición del objeto astronómico pasado como argumento, haciendo uso del método `SkyCoord.from_name()` de la librería `Astropy`, esto debido a que podría no encontrarse coordenadas astronómicas para este objeto, ya sea porque no existe, hubo un error de tipeo o la base de datos no está actualizada¹⁵. De no encontrarse las coordenadas, se retorna un objeto `VoToolResponse` con un mensaje de error en el atributo correspondiente a la respuesta semántica.

Con los datos necesarios obtenidos, se itera sobre la lista de URLs y en un bloque `try-except` se intenta acceder a los recursos: Primero, se obtiene el servicio de la URL, se realiza una búsqueda con argumentos la posición del objeto y el tamaño del área de búsqueda. Si la cantidad de resultados es nula, es continua con la siguiente URL. Si se encuentra uno o más resultados, se agrega una respuesta semántica al objeto `VoToolResponse` que describa la cantidad de datos encontrados y donde. Luego, se itera sobre los resultados, los cuales son una instancia de `SIARRecord` de `PyVO`, lo que facilita la extracción de metadatos gracias a su interfaz. Los metadatos a guardar serán:

- Título: El título del conjunto de datos/imagen (recurso). Atributo: `title`
- Formato: El formato en el que está el archivo (imagen o FITS). Atributo: `format`
- Tamaño: Tamaño del archivo, en bytes. Atributo: `size`
- URL: URL de acceso al recurso. Atributo: `acref`
- Instrumento: El instrumento utilizado para recolectar los datos del recurso. Atributo: `instr`
- Posición: Las coordenadas celestiales de la observación. Atributo: `pos`
- Bandpass: El nombre o identificador de la *bandpass*¹⁶ usada para la observación. Atributo: `bandpass_id`
- Bandpass de referencia: La longitud de onda, frecuencia o energía característica de la *bandpass*. Composición de los atributos `bandpass_refvalue` y `bandpass_unit`.

¹⁵`SkyCoord` utiliza `Sesame[Centre, ]` para traducir nombres de objetos a coordenadas, y puede suceder que no encuentra estas coordenadas, como pasa con el planeta Neptuno (Neptune)

¹⁶Un filtro a que permite el paso a las frecuencias dentro de su rango, y atenúa a otras

- Rango de bandpass: Los límites superior e inferior de la bandpass. Composición de los atributos `bandpass_lolimit` y `bandpass_hilimit`.

Estos metadatos se guardan en un objeto tipo diccionario y se agrega a una lista, la cual al finalizar la iteración por los recursos, se pasa como respuesta de datos del objeto `VoToolResponse`, y se retorna este último como respuesta exitosa de la función.

Si cualquier error ocurre en el proceso de acceso a servicios o extracción de metadatos, se atrapa en la parte `except`, se muestra por consola y se continúa iterando sobre la lista de URLs.

Por último, la descripción de esta herramienta es la siguiente:

```

"""This function is for retrieving image resources. Execute it when
the user requests images or visual or graphic data.
More specifically, this function queries a specified resource with
the Simple Image Access protocol to retrieve an image or FITS
file whose data was obtained by a telescope of a region or an
object in space. It takes in the position and size of an area in
the skydome (what we see when we look into space) and the url of
a resource that may contain records and images of this area.

Args:
    position: The particular name of an astronomical object. A
        function will parse it into sky coordinates. It is crucial that
        the name is a 'particular' name and not a generic 'type of
        object'.
    area: Size of the area in which records are to be searched
    unit: Unit of the amount described by the area arg
    url: (optional) url of the service to query

return:
    A JSON serializable object that contains a semantic response for
    the Agent to read and a list of resources to get data from. The
    later is only ever shown to the user.
"""

```

Notar como en esta descripción de la función también describimos que y para que son los argumentos, a diferencia de `get_registry`. Esto es porque no definimos en una clase aparte la descripción ni los tipos de los argumentos.

La segunda función, `get_img()`, es una forma más simple de `query_sia`. Recibe un solo argumento; el nombre de un objeto astronómico, y retorna una imagen aleatoria encontrada en la base de datos de IVOA, junto con algunos metadatos de la imagen (Título, Formato, tamaño y URL).

El procedimiento de la función es similar al de `query_sia`, solo que en lugar de iterar sobre

los recursos obtenidos de la query a uno de los servicios, se escoge un al azar y se retorna en un objeto VoToolRepsonse, donde en vez de tener una lista de objetos tipo diccionario, se guarda 1 solo diccionario.

La descripción de la herramienta es la siguiente:

```

"""Use this function when the user asks to be shown an (any) image
    of a specific astronomical object. This function returns a random
    image that is realted to said astronomical object.
Do not confuse this function with the 'query_sia' function, which is
    more complete and returns a set of resources of graphical and
    FITS data.

Args:
    astro_object: The name of the astronomical object or event the
        user requests an image of.

return:
    A JSON serializable object that contains a semantic response for
        the Agent to read and an object that contains the properties of
        the image, as well as the link.
"""

```

Simple Spectral Access

Debido a la naturaleza tabular de los datos del Registry, la función para acceder al SSA (nombre: `query_ssa()`) y el resto de herramientas siguen un flujo similar a las funciones para acceder al Registry y SIA.

`query_ssa()` recibe 5 argumentos para poder consultar servicios que permitan acceder a datos espectrales, estos servicios se buscan en el Registry (al igual que se hizo anteriormente), se consultan y se extraen en un objeto de tipo VoToolResponse, el mismo que se usó anteriormente.

Los argumentos necesarios son:

- Posición [str]: El nombre de un objeto astronómico del cuál se quiere obtener data espectral. Este nombre será buscado en base de datos astronómicas para intentar obtener sus coordenadas celestiales.
- Diámetro [float]: El **diámetro** de la región circular alrededor de la posición en la cual buscar. Se asumen grados decimales icrs¹⁷
- Banda [tuple]: El ancho de banda que los datos deben calzar. Se asumen metros y es un argumento opcional.

¹⁷<https://docs.astropy.org/en/stable/coordinates/index.html>

- Tiempo [tuple]: El rango de tiempo que los datos deben calzar. Es un argumento opcional.
- URL [str]: El URL del servicio en el que se quiere buscar datos. Es un valor opcional y de no ser proporcionado se buscará en el Registry los servicios disponibles

El proceso inicia intentando obtener la posición del objeto de la misma forma que el `query_sia` y construyendo un objeto diccionario con las llaves 'pos', 'diameter', 'band' y 'time', que corresponden respectivamente a la posición (en SkyCoord), el diámetro, la banda y el tiempo.

Luego, se obtiene las URLs de los servicios que entregan al interfaz SSA, si es que no se proporcionó una URL. Este paso se realiza de la misma forma que el `query_sia()`.

Para consultar los servicios se itera sobre las URLs encontradas y se consulta cada una, al igual que se hizo en las herramientas anteriores. Los argumentos se pasan al método `search()` del servicio a través de desempaqueado de diccionario (en caso que alguno de los argumentos sea nulo) y se verifica que la cantidad de resultados obtenidos sea mayor que cero. Si es así, se pasan a formato tabla (`to_table()`) y se extraen los nombres de las columnas con el atributo `colnames` y los valores de las filas pasando los resultados a un `numpy` array, a diferencia de las herramientas previas. El motivo es una de las mismas razones que justifican la necesidad de esta aplicación; Las distintas fuentes de datos y los distintos servicios para acceder a ellas no poseen una interfaz regulada. Es decir, los nombres de las columnas varían según la fuente de información. Por lo tanto se opta por recuperar las tablas de datos sin mayor transformación ni traducción.

Una vez extraídas las filas, se arma una lista con objetos tipo diccionario, donde las llaves son las columnas y los valores los datos de cada fila extraída. Esta lista se agrega como respuesta de data a un objeto `VoToolResponse` y se añade una respuesta semántica que describa la cantidad de datos recuperados, para finalmente retornar el objeto.

La consulta al servicio en cada URL es envuelta en un bloque `try-except`, por si sucede un error inesperado en las consultas. El valor de retorno en caso de no encontrar resultados es una respuesta semántica informando la situación en el objeto `VoToolResponse`, sin respuesta de data.

Finalmente, la descripción de la herramienta es:

```
"""This tool queries a Simple Spectral Access service to retrieve
spectra-related data from the VO. Use it when the user asks
specifically for 'spectral data of...' and if they ask for a
particular object, and not a type of object (this is very
important). If a url is not provided it will query the registry
to find a service that matches the constraints passed in as
arguments. If a url is provided or a matching service was found,
it will query it and return a dictionary with relevant data of
the resource.
```

Args:

```

position: The particular name of an astronomical object. A
          function will parse it into sky coordinates. It is crucial that
          the name is a 'particular' name and not a generic 'type of
          object'.
diameter: The diameter of the circular region around position in
          which to search, assuming icrs decimal degrees
band: (optional) The bandwidth range the data needs to match,
          assuming meters
time: (optional) The datetime range the data needs to match
url: (optional) The url of the resource to be queried. If none
          provided, the position will be used as keywords to search the
          registry
"""

```

Simple Cone Search y Simple Line Access

Las funciones que describen las herramientas para acceder al Simple Cone Search y Simple Line Access son similares a la para acceder a Simple Spectral Access, a excepción de 3 partes: los argumentos de la función, la clase de la *Data Access Layer* que se utiliza para acceder a los servicios y la descripción de la función.

Partiendo por Simple Cone Search. El objetivo de esta función es obtener toda la data disponible dentro de un cono descrito por las coordenadas de un objeto astronómico y un radio. Su función toma 3 argumentos:

- Posición [str]: El nombre de un objeto astronómico cuyas coordenadas están en el centro del cono en el que se deben encontrar los datos. Este nombre será buscado en base de datos astronómicas para intentar obtener sus coordenadas celestiales.
- Radio [float]: Radio del cono en el que deben ubicarse los datos a buscar.
- URL [str]: El URL del servicio en el que se quiere buscar datos. Es un valor opcional y de no ser proporcionado se buscará en el Registry los servicios disponibles.

Luego de obtener las coordenadas del objeto, se pasan a un diccionario con la llave 'pos' y el radio se pasa en la llave 'radius'.

La clase *service* que se utiliza es *SSAService()* y los argumentos del método *search()* se pasan desempaquetando el diccionario de argumentos.

La descripción de la función es la siguiente:

```

"""

```

This tool is similar to the tool 'get_registry', the difference is that this tool queries services recovered from the registry via Simple Cone Search. This means that any resource that has data whose position lies within a cone described by the position and radius given to this tool as arguments is returned in a data table that is structured as a python dictionary.

Args:

position: The 'particular' name of the astronomical object that represents the center of the circular search region where the resource data must lie within to match the constraint. This value must not be a 'type' of astronomical object, but the specific name. The area of this region is described by the radius.

radius: Radius of the circular search region.

url: (optional) The url of the resource to be queried. If none provided, the position will be used as keywords to search the registry.

Returns:

A JSON serializable object that contains a semantic response for the Agent to read and a list of resources to get data from. The later is only ever shown to the user.

"""

Para Simple Line Access el objetivo de la función es recuperar data de líneas espectrales referente a observaciones que caigan dentro de un rango de longitud de onda especificado. Su función toma 3 argumentos:

- Longitud de onda [tuple[float]]: Un par de numeros float que describen el rango de longitud de onda en el que se deben hayar los datos de las observaciones.
- Unidad [str]: La unidad de los valores proporcionados en la longitud de onda. El valor por defecto es 'meters'
- URL [str]: El URL del servicio en el que se quiere buscar datos. Es un valor opcional y de no ser proporcionado se buscará en el Registry los servicios disponibles.

El rango de longitud de onda se debe componer usando la clase Quantity de PyVO y los argumentos para longitud de onda y unidad. La clase de pyvo.dal que se usa para acceder a los servicios es SLAServive y los argumentos del método search() son solamente el rango de longitud de onda obtenido con Quantity.

la descripción de la función es la siguiente:

```

"""
This tool takes in a wavelength range and queries the VO using the
Simple Line Access protocol to recover data of
observations of elements in celestial bodies and space.

Args:
    wavelength: Pair of floats that describe a wavelength range. The
        results of the query to the service lie within this range.
    url: (optional) The url of the resource to be queried. If none
        provided, the position will be used as keywords to search the
        registry.

Returns:
    A JSON serializable object that contains a semantic response for
    the Agent to read and a list of resources to get data from. The
    later is only ever shown to the user.
"""

```

3.0.4. Generación asistida de recuperación de documentos (RAG)

Para dar al Agente información actualizada y específica a los tópicos de astronomía y observatorios virtuales es necesario otorgarle funcionalidad de recuperación de documentos. Estos documentos pueden ser artículos científicos, páginas web de divulgación o documentación de programas, entre otros. Para dar esa funcionalidad al agente es necesario que pueda: recuperar (cargar o descargar) documentos (previamente especificados), integrarlos como contexto a la pregunta o solicitud del usuario, evaluar la relevancia de los documentos y actuar acorde a esta. Para cada paso mencionado, se crea una función.

Para **recuperar documentos**, creamos un retriever. Este retriever toma URLs especificadas y descarga, divide y transforma los documentos, para guardarlos en una *vector store*, la cual puede ser consultada cada vez que sea necesario, sin necesidad de volver a realizar el proceso descrito. Para implementar el retriever, primero se cargan los documentos (en esta implementación se decidió utilizar páginas de las definiciones de estándares de IVOA). Estas URLs se encuentran en un documento en formato json el cual se compone de una lista de objetos con la URL de la página y un valor bool que describe si ya fue leída la página. Se extraen todas las URLs que no se hayan leído y se actualiza su estado en el documento. Luego, utilizando el cargador de documentos *WebBaseLoader* de *LangChain*, se descargan los documentos en una lista, para luego extraer la información en otra. Se crea el *"text splitter"* o separador de documentos. En este caso se utiliza el *RecursiveCharacterTextSplitter* de *LangChain* y su método *from_tiktoken_encoder* con un tamaño de "chunk" (o unidad) de 100 y un *overlap* de 50. Luego, se utiliza el método

`split_documents()` para separar los documentos.

Se crean los *embeddings* utilizando `OpenAIEmbeddings` y el modelo “text-embedding-3-small” y un cliente de la vector store (`QdrantClient`). Como fue decidido en la sección de arquitectura del agente, se utilizará **Qdrant Vector Store** mediante la clase `QdrantClient` y se hosteará en **Qdrant Cloud**.

El nombre de la colección a usar está definido en las variables de entorno. Si no existe, la vector store se crea con el método `from_documents()` de la clase `QdrantVectorStore`, pasando como argumentos los “*splits*” generados por el *text splitter*, los *embeddings*, el nombre de la colección y la URL y *ApiKey* del *cluster* en Qdrant Cloud. Por otro lado, si ya existe una colección con ese nombre, se crea con el método `from_existing_collection()`, pasando los mismos argumentos previos, a excepción de los “*splits*”. Estos se agregan manualmente con el método `add_documents()`.

Finalmente se crea el *retriever* a partir de la vector store usando el método `as_retriever()`

Para **integrar los documentos** es necesario agregar el *retriever* como nodo independiente al flujo de trabajo del Agente. Para eso usamos la función `create_retriever_tool()`, que toma por argumentos el *retriever*, el nombre que queramos darle a la herramienta de recuperación y una descripción. Usamos la clase `ToolNode` de `LangChain` para crear el nodo y lo agregamos al `StateGraph` igual que los nodos previos.

Luego de integrar el *retriever* al flujo de trabajo, el Agente necesita decidir si generar la respuesta con los documentos que haya recuperado o reformular la pregunta del usuario para obtener mejores documentos. Para esto se requiere una función que permita evaluar la relevancia de los documentos recuperados, así como funciones que actúen según el resultado de la evaluación. Primero se define la función que reescribe la pregunta inicial del usuario en caso que los documentos recuperados no sean relevantes, esto para que el *retriever* entregue otros resultados. La función recibe como argumento el `AgentState`, del cual extrae la pregunta del usuario. Luego, crea un `HumanMessage` que solicita que reformule la pregunta del usuario, la que se agrega dentro del mensaje. Este “mensaje humano” se entrega a un LLM pequeño que se crea dentro de la función, y su respuesta se entrega como resultado final de la función de reescritura.

La función que genera la respuesta con los documentos, en caso de ser relevantes, también recibe el `AgentState` y extrae la pregunta del usuario y los documentos entregados por el *retriever*. Utilizando un *prompt* obtenido desde `langchain.hub`, crea una cadena con este, un LLM pequeño creado dentro de la función y a la clase `StrOutputParser()` de `LangChain`, la cual es invocada y se le pasa como argumento un diccionario con los documentos en la llave “context” y la pregunta del usuario en “question”. El retorno de la función es la actualización de los mensajes de `AgentState` con la respuesta de la cadena.

La función que tomará la decisión respecto a la relevancia de los documentos recibe como argumento el `AgentState` y su retorno es el nombre de uno de los nodos basados en las 2 funciones descritas. Empieza creando una *class-object* con un atributo *str* descrito por la función `pydantic.Field` con su descripción definiéndola como un “yes” o un “no”. Esta clase se pasa a un LLM con `with_structured_output()` y luego se crea un *prompt* con `PromptTemplate()` con los siguientes argumentos:

```

template="""You are a grader assessing relevance of a retrieved
    document to a user question. \n
    Here is the retrieved document: \n\n {context} \n\n
    Here is the user question: {question} \n
    If the document contains keyword(s) or semantic meaning related to
    the user question, grade it as relevant. \n
    Give a binary score 'yes' or 'no' score to indicate whether the
    document is relevant to the question.""",
input_variables=["context", "question"]
    
```

se crea una cadena con el prompt y el LLM, la que se invoca pasando con argumentos la pregunta del usuario en “question” y los documentos en “context”.

Se extrae el atributo `binary_score` de la respuesta de la cadena; si el valor es “yes”, se devuelve el nombre del nodo para generar la respuesta, en otro caso, se retorna el nombre de nodo para reescribir la pregunta. “Con estas tres funciones se puede añadir el flujo de RAG al Agente. Primeramente se agregan los 3 nodos correspondientes a las funciones recientemente descritas, luego, se agregan 2 “bordes” condicionales. Los bordes condicionales se añaden al `StateGraph` con `add_conditional_edges()` pasando como argumentos: el nombre del nodo desde donde se origina el borde, la función que determina el siguiente nodo y, de ser necesario, un diccionario de pares palabra-nombre de nodo.

El primer borde parte desde el nodo “agent” (LLM) y usa una función simple que: recibe el `AgentState`, si el valor de la llave “name” del atributo “`tool_calls`” del último mensaje no es nulo, retorna “retrieve”, si es nulo entonces retorna “__end__”, lo que indica el final de el flujo del Agente. Como retorna nombres distintos a las funciones, el diccionario de argumento para el borde condicional tiene los pares llave valor “tools”: “tools”, “retrieve”: “retrieve” y “END”: “END”¹⁸

El segundo borde condicional parte desde “retrieve” y utiliza a función de evaluación de documentos.

Finalmente agregamos dos bordes finales: “rewrite”->“agent” para volver a buscar documentos y “generate”->END para entregar la respuesta final.

3.0.5. Implementación completa del flujo de trabajo del agente

Ya con cada herramienta implementada y sus interfaces definidas, se procede a definir el flujo de trabajo complejo para el Agente, el que le permitirá razonar cuando usar cada herramienta y definirá como entrega los datos al front-end para utilización del usuario.

Empezando por el `AgentState` o estado del agente. Inicialmente, el estado estaba compuesto solo de la lista de mensajes definida con `typing.Annotated`. Como el `class-object` retornado por las herramientas posee una respuesta semántica y una de data, es necesario guardar estos por separado, así que se añaden 2 atributos más a `AgentState`:

¹⁸Constante de `LangGraph` equivalente a “__end__”

- Última herramienta [str]: El nombre de la última herramienta que se llamó. Esta información servirá al front-end para renderizar correctamente los datos. En caso de que el LLM responda directamente, puede ser nulo.
- Datos de la herramienta [any]: El diccionario o lista de diccionarios que contiene(n) la información obtenida de los datos del Observatorio Virtual. Su valor puede ser nulo y en el mismo caso que la última herramienta.

Se actualiza el nodo `call_model` con un prompt más completo, dado que el Agente ahora tiene capacidades de RAG y utilización de herramientas. El prompt se crea con `SystemMessage` y se inserta dentro del prompt la fecha del momento:

```
system_prompt = SystemMessage(
    """The current date is {0}. You are an assistant to astronomers
       and people interested in astronomy. Your speciality is
       aswering questions related to the Virtual Observatory, by
       either answering questions about it (using the retriever tool)
       or returning data and information from it's services using
       the VO tools you will be provided with. When a VO tool
       is executed, it will respond to you with a success message
       describing the state of success or an error message that
       states why it failed. When successful, the VO tool will
       create
       data and store it in the state of this application, which you do
       not have access to. DO NOT CREATE YOUR OWN ANSWER if you
       called a VO tool, just limit yourself to inform the user
       of the success of the query and any desription the VO tool gave
       you.
       If the user's query is ambiguous or doesn't specify the name of
       the specific object to look for, use the VO tool for the
       Registry. If it's more specific and you have enough arguments
       ,
       query the other services using the other VO tools.
       You may only call 1 (one) tool per user's request.
       """.format(current_date)
)
```

Para otorgar acceso al Agente a las herramientas inicializamos el modelo con el método `bind_tools()`, pasando como argumento una lista con referencias a las funciones de las herramientas y especificando el argumento `tool_choice` como "auto", cosa que el LLM escoja que herramienta usar autónomamente. Además, debido a la naturaleza de las herramientas, es necesario implementar manualmente el nodo `tool_node` que ejecutará las herramientas que el LLM decida llamar.

El `tool_node` se define como una función cuyo argumento único es el `AgentState`. Primero

se definen una lista de outputs y una variable donde guardar la respuesta de datos de una herramienta. Luego, se itera sobre el atributo `tool_calls` del último mensaje en la lista del `AgentState`, esto en caso de que el LLM decida llamar a más de una herramienta (por error). Dentro del ciclo, se invoca a la herramienta llamada, obteniéndola del diccionario previamente definido y pasando como llave el valor de la llave “name” de la `tool_call`, y haciendo uso del método `invoke()`, pasando como argumentos los valores en la llave “args” del diccionario de la `tool_call`. Una vez extraído el resultado, se agrega un objeto `ToolMessage` de la librería `LangChain` a la lista de outputs, pasando la respuesta semántica del resultado de la herramienta en el argumento “content”, el nombre de la función en “name” y el id de la llamada en “tool_call_id”, obtenido de la llave “id” de la `tool_call`.

Si el resultado es de tipo `VoToolRepsonse`, se guarda la respuesta de datos del resultado de la herramienta, para finalmente retornar un objeto que actualice: lista de mensajes, la última herramienta llamada y los datos de la herramienta, con los valores generados.

Se agrega el nodo de herramientas al `Stategraph` y se actualizan el borde condicional que redirige a “retrieve” o END como la función que toma la decisión. Para la función se agrega que, si el atributo `tool_calls` no es nulo y la última herramienta llamada es el `retriever`, se retorna “retrieve”, si no, se retorna “tools”, y el valor de retorno default es “__end__”.

El flujo complejo del Agente ha sido definido en su totalidad y la representación gráfica es la siguiente¹².

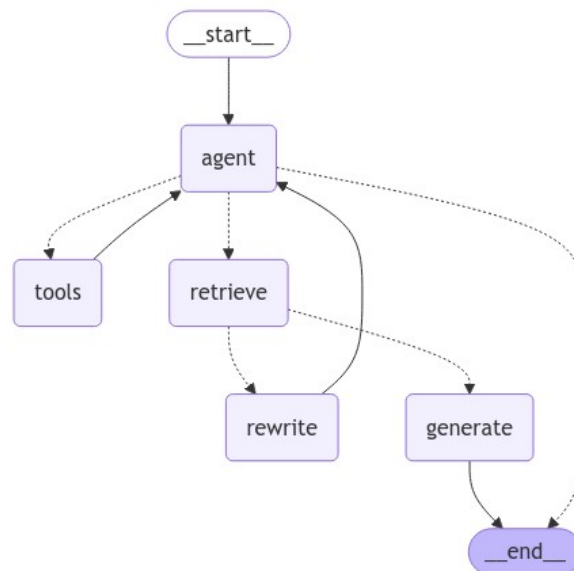


Figura 12: StateGraph del Agente complejo con herramientas y RAG.

Fuente: Elaboración propia.

El flujo comienza con un mensaje del usuario, el cual es alimentado al nodo “agent” que toma la decisión de utilizar una herramienta en “tools” o recuperar documentos relevantes en “retrieve”. En el primer caso, el resultado es devuelto directamente a “agent” para luego

dirigirse al fin del flujo. En el segundo caso, los documentos recuperados son evaluados y, de ser relevantes, se genera una respuesta con ellos como contexto y se dirige al fin del flujo, de otra forma, se reformula la pregunta del usuario y se vuelven a recuperar documentos.

Ya completado el flujo, es necesario redefinir la función con la que el endpoint recibe solicitudes y envía respuestas al front-end.

La función debe ser asíncrona debido a la naturaleza de las consultas a los servicios del Observatorio Virtual. Toma como argumentos el mensaje del usuario y el objeto config definido anteriormente. Se invoca al StateGraph compilado con el objeto config y un diccionario que será el AgentState inicial: los mensajes son una lista con 1 mensaje que es el del usuario y la última herramienta llamada se inicia con una string "None". Si el AgentState retornado por el Agente invocó una herramienta, se retorna un objeto con: el nombre de la última herramienta llamada, el último mensaje del LLM y los datos entregados por la herramienta llamada. Si no se llamó ninguna herramienta, entonces el objeto solo tendrá el último mensaje del LLM.

3.0.6. Definición de interfaz gráfica y presentación de información

Con el Agente completo, falta crear el front-end mediante el cual el usuario podrá comunicarse con el Agente. Como se discutió antes, se creará una app utilizando **Vite**. Para crear la app se necesita instalar:

- Node.js
- npm
- Vite

Luego, se ejecuta el comando `npm create vite@latest`, seleccionando "React" como el *framework* y "JavaScript" como lenguaje.

Para posibilitar la comunicación con el Agente, es necesario conectarse al endpoint definido previamente. Esto se puede lograr de múltiples formas, pero por simplicidad se utiliza la función `fetch()`, nativa de javascript. Pero antes es necesario crear una interfaz para poder escribir el mensaje para el Agente, enviarlo, recibirlo y mostrarlo. Para esto se crean:

- Un cuadro de texto para enviar los mensajes al back-end.
- Una función que maneje el envío, la recepción y actualización de los mensajes.
- Un grupo de componentes que rendericen estos mensajes, dependiendo si es texto, datos y el tipo de datos.
- Un contenedor que despliegue los mensajes rederizados.

- Otro contenedor que integre ambos, el contenedor de mensajes y el cuadro de texto.
- Por último, un componente que integre todos los anteriores y maneje el estado de la aplicación.

Debido a que el objetivo de este trabajo no es el de crear una interfaz atractiva o evaluar la repsonsividad e innovación de la interfaz, solo se discutirán las partes cruciales para la interacción con la interfaz y la de esta con el Agente. De todas formas, en los anexos se dejara evidencia de los estilos específicos utilizados para la creación de esta app, los que se referenciarán al lado de la mención del componente.

Primero se crea **el componente que une todos los demás**, debido a la necesidad de crear los *hooks*¹⁹ de tipo `useState`. Estos hooks nos permiten conservar y modificar información de la aplicación mientras está en ejecución:

- Mensajes: El valor del hook corresponde a una lista que contendrá todos los mensajes a enseñar en la conversación
- Mensaje input: Corresponde al valor del mensaje del usuario que será enviado al back-end
- Cargando: Hook que permite mostrar un símbolo de carga mientras el back-end responde
- Error: Hook que permite desplegar un mensaje de error sin parar la aplicación.

Luego, la función que maneja la interacción con mensajes inicia verificando si el mensaje input no es nulo. De ser así, se setea el hook de carga y se construye un objeto mensaje, con una id, el contenido del mensaje y quien lo envía (en este caso, el usuario). Se modifica la lista de mensajes, agregando el recientemente creado y se elimina el mensaje de input. Luego, utilizando al función `fetch()`, se crea una promesa²⁰ para comunicarnos con el endpoint del agente, definiendo el método http como “POST”, el tipo de contenido como “application/json” en los cabecales y el mensaje en el cuerpo, asegurando que el nombre del valor sea el mismo que se espera en el endpoint.

Usando el método `then` se pasa la respuesta de la solicitud a formato “json” y se revisa el contenido. Si contiene un mensaje de error, se actualiza el hook de error y se arroja con las sentencia “throw”, para evitar el colapso de la aplicación. Si no hay error, se crea un nuevo mensaje, donde el contenido es el recibido por parte del Agente y quien lo envía es el mismo. Se agrega el mensaje al resto y se desactiva el hook de carga.

¹⁹Un hook es una función que permite alterar el estado de la aplicación, cambiando el valor de variables que afectan el como se presenta o comporta la aplicación. Por ejemplo, un hook `useState` podría permitir cambiar la página en una tabla de datos.

²⁰Una promesa es una operación que retornará un valor a futuro, de manera asíncrona.

El componente del **cuadro de texto** consta de un `mui-TextField`, un **botón** y un **contenedor** para unir ambos. Para el `TextField`, la propiedad `“value”` es el valor del mensaje input, la propiedad `“onChange”` es una función que toma el `“event”`²¹ y ejecuta el hook para setear este mensaje y la propiedad `“onKeyDown”` es una función corta que ejecuta la función para manejar la interacción con mensajes si la tecla presionada fue `“enter”`.

El componente del botón tiene la función para manejar los mensajes en su propiedad `“onClick”` y el contenedor encierra a ambos el cuadro de texto y el botón.

Son cinco los **componentes para renderizar los mensajes**: el contenedor de la conversación, el contenedor de cada mensaje, el que renderiza los mensajes de solo texto, el que renderiza los mensajes con datos y el que tabulariza los datos.

El **contenedor de la conversación** es un componente `mui-Paper` estilizado. Dentro del contenedor hay otro componente `mui-List`, y dentro de esta lista renderizamos 1 o 2 `mui-ListItem` por mensaje. El factor que decide la cantidad es el tipo de mensaje: si es solo un mensaje directo del LLM, se renderiza 1 componente de mensaje de texto dentro del `ListItem`; si es un mensaje que contiene una respuesta semántica y de datos, se renderizan ambos, un componente de mensaje de texto y uno de mensaje con datos, ambos rodeados por su respectivo componente `ListItem`.

El **contenedor de cada mensaje** es un componente `mui-Box` estilizado según el remitente del mensaje. El **componente que renderiza solo texto** toma como propiedad quien envía el mensaje y el contenido del mensaje y entrega como retorno el contenedor a quien le pasa el remitente con un `mui-ListItemText` con el contenido.

El **componente que renderiza los datos** revisa cual fue la última herramienta llamada por el Agente (porque la respuesta del Agente es su `AgentState`²² simplificado). Si la última herramienta es la que entrega una imagen de un objeto astronómico inmediatamente, retorna un contenedor de mensaje con un `mui-Box` con los valores `“img”` y el link de la imagen en sus propiedades `“component”` y `“src”` respectivamente. El link se obtiene de la propiedad `link` del `class-object` que retorna la herramienta. Si es otra de las herramientas que acceden a los servicios del Observatorio Virtual, entonces se renderiza el componente que tabulariza los datos.

El **componente que tabulariza los datos** se define como una función que recibe un diccionario como atributo, el que corresponde a los datos entregados en la respuesta del Agente, y utiliza el componente `DataGrid` para crear una tabla con esos datos. Primero, a los datos se les agrega un atributo `“id”` que es un número único. Luego, de uno de los datos del diccionario se extraen los campos y se transforman en columnas del `mui-DataGrid`, cada una compuesta por compuesta de 2 o 3 campos: `“field”` y `“headerName”`, que son el nombre de la columna y el nombre que se muestra en tabla, y `renderCell` para columnas que contengan un link. Una vez extraídas las columnas, se retorna un componente `mui-Paper`, que contiene el `mui-DataGrid`, al que se pasan los datos y las columnas en sus propiedades, además de todas las modificaciones que se quieran hacer a su comportamiento.

²¹Los `“event”` o eventos son las interacciones que tiene el usuario con la interfaz.

²²el mensaje es parseado a JSON en la recepción, lo que permite acceder al `AgentState` como si fuera el back-end

Descritos todos los componentes, se unen en el retorno del componente que maneja el estado, compuesto por el contenedor que integra ambos: cuadro de texto y contenedor de la conversación. Dentro del contenedor de la conversación está la lista que genera un `ListItem` por cada mensaje, y los contenedores apropiados para cada caso.

Este componente final es la Interfaz Usuari (UI) de la aplicación, con lo que finaliza el desarrollo de la app y da comienzo a la validación.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

El objetivo de la aplicación creada es el de facilitar el acceso a información astronómica mantenida en el Observatorio Virtual, tanto para profesionales del área y subáreas de astronomía, como para gente común sin conocimiento especializado, acercando el conocimiento y el trabajo real de la astronomía a la gente.

4.1. Metodología de validación y definición de medio

Para medir la efectividad de la aplicación en cumplir los objetivos planteados, se considera pertinente una evaluación de tipo **cualitativa** donde usuarios de distintos niveles de conocimiento astronómico usen la aplicación y estimen si se cumplen ciertos criterios, los cuales serán sinónimo del nivel de efectividad de la herramienta. Para esto se realizara una prueba de la aplicación. Luego, cada usuario deberá contestar una encuesta en **Google Forms**, que recoge automáticamente los datos y permite visualizarlos con sus herramientas.

Las preguntas a realizar en el formulario serán las siguientes:

1. ¿Cuál es tu nivel de conocimiento astronómico? Responde en la escala del 1 al 10, donde 1 es ignorancia total y 10 es experto en el área.
2. ¿Haz intentado obtener datos astronómicos reales previamente? Si no, ¿por qué?
3. ¿Conocías el Observatorio Virtual?
4. ¿Qué tan fácil te fue usar la aplicación? Responde en la escala del 1 al 7, donde 1 es muy difícil y 7 es muy fácil.
5. ¿Encontraste problemas para obtener información? ¿Cuáles fueron?
6. ¿Consideras que la aplicación facilita el acceso a datos astronómicos? Responde en la escala del 1 al 7, donde 1 es que no facilita nada y 7 es que lo vuelve una tarea sencilla.
7. ¿Cuál es la probabilidad que vuelvas a usar la aplicación más adelante? Responde en la escala del 1 al 10, donde 1 es que nunca la usaras de nuevo y 10 es que estás seguro de usarla con regularidad.
8. Por último, ¿Qué funcionalidades agregarías a la aplicación?

Con estas preguntas se estima poder caracterizar a los usuarios anónimos en su nivel de conocimiento astronómico y familiaridad con el Observatorio Virtual, para luego de utilizar

la aplicación, rescatar su perspectiva respecto a la facilidad de uso y potencial para obtener datos astronómicos.

4.1.1. Usuarios y resultados

Los usuarios contactados para la validación de la aplicación constan de ocho: cinco personas con interés en la astronomía, pero que no se desempeñan en el área, y tres personas que poseen conocimiento astronómico considerable y se desempeñan (o lo harán) en el área. Los resultados fueron recolectado mediante un formulario de Google Forms, y las estadísticas de las respuestas, así como el análisis de los datos se presentan a continuación.

4.2. Análisis de datos recolectados

De los resultados entregados por la encuesta pre y post *testing* de la aplicación, podemos rescatar las siguientes relaciones, visualizadas mediante grafos para una mejor apreciación.

¿Cuál es tu nivel de conocimiento astronómico? Responde en la escala del 1 al 10, donde 1 es ignorancia total y 10 es experto en el área.
8 respuestas

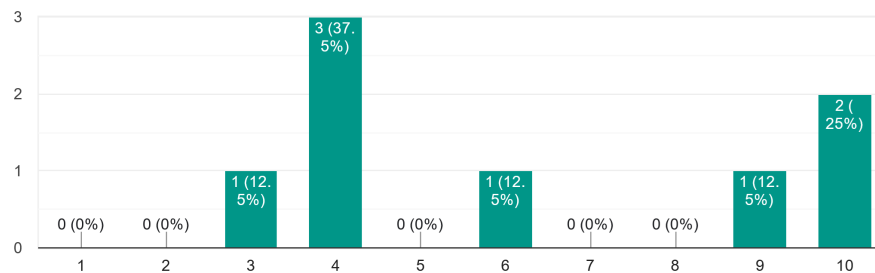


Figura 13: Cantidad de usuarios por nivel de conocimiento (rango nivel: 1 a 10)

Fuente: Elaboración propia.

En la figura 13 se puede apreciar el nivel de conocimiento que poseen los usuarios encuestados. De estos usuarios se obtuvo el promedio de nivel de conocimiento según habían buscado previamente datos astronómicos tabulares o duros, tipo de datos que se encuentra en el Observatorio Virtual, en su mayoría. Los resultados se observan en la figura 14.

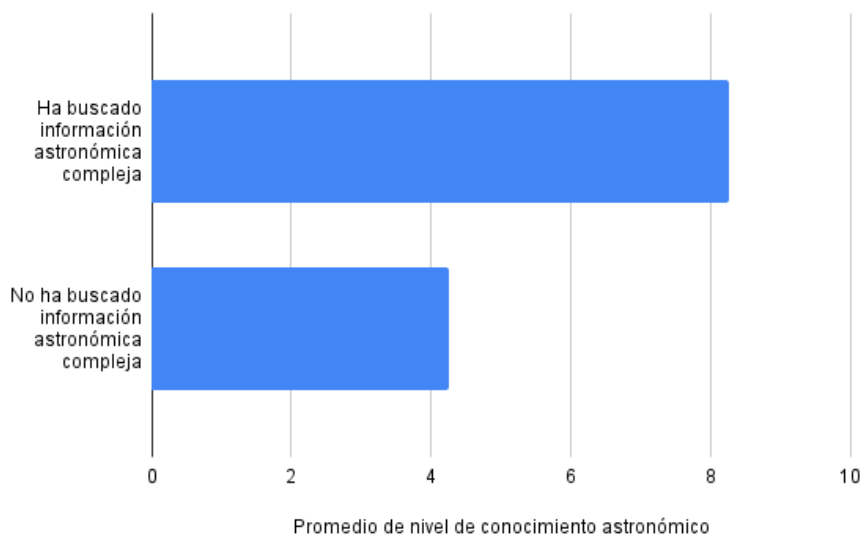


Figura 14: Nivel de conocimiento promedio de los usuarios que buscaron previamente datos duros astronómicos.

Fuente: Elaboración propia.

Se evidencia que efectivamente, quienes han buscado previamente información dura son quienes a poseían un conocimiento avanzado en la materia.

Esta misma relación se intentó hacer respecto a previo conocimiento del Observatorio Virtual. Los resultados de la comparativa fueron no concluyentes; es decir, gente con alto conocimiento desconocía el observatorio tanto como gente de bajo conocimiento. Esto puede deberse a difusión por parte de las universidades responsables del ChiVO (ya que el estudio se realizó en Chile). De todas formas, los datos se encuentran en el anexo 1.

Ya realizadas las pruebas prácticas, donde los usuarios utilizaron el agente con total libertad, los resultados y percepciones de los usuarios se pueden ver en las figuras 15, 16, 17 y 18.

¿Encontraste problemas para obtener información?

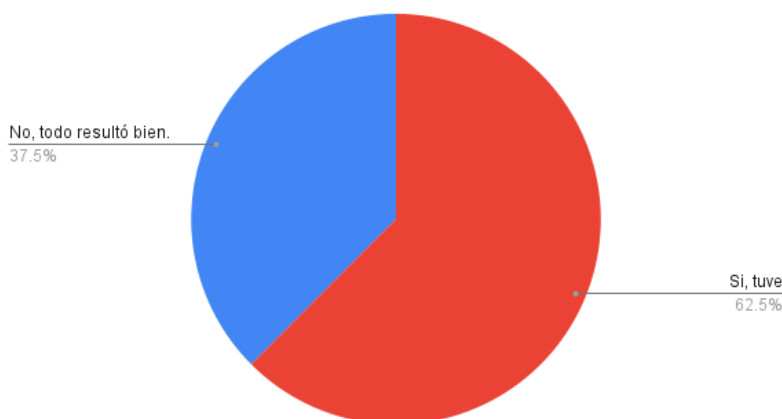


Figura 15: Porcentaje de usuarios que tuvieron problemas al usar la aplicación
Fuente: Elaboración propia.

¿Qué tan fácil te fue usar la aplicación? Responde en la escala del 1 al 7, donde 1 es muy difícil y 7 es muy fácil.

8 respuestas

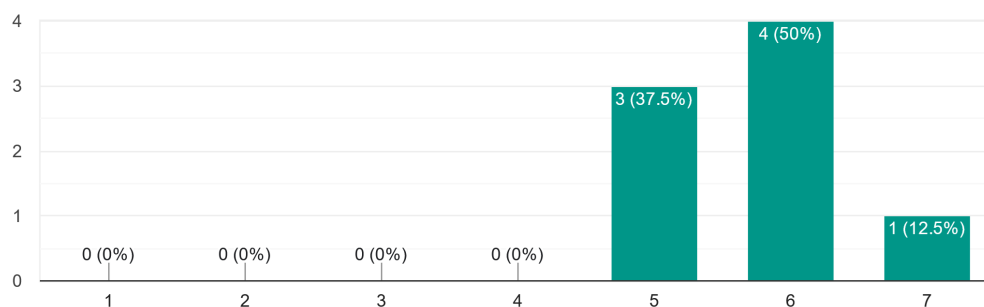


Figura 16: Distribución de calificación de facilidad de uso (rango calificación: 1 a 7).
Fuente: Elaboración propia.

Contrastando las calificaciones a la facilidad de uso de la aplicación con la cantidad de usuarios que encontraron problemas, se puede conjeturar que los problemas no vinieron del incorrecto funcionamiento de la aplicación, sino de la incapacidad de esta de poder encontrar datos solicitados. Esto se hace aún más evidente tomando en cuenta los motivos descritos por los usuarios que encontraron problemas:

- *“Hubo tipos de datos que no encontré como SED del sol o de enanas blancas, en otras respuestas cuando quise acceder al link este no funcionaba.”*

- *“No logré acceder a muchos datos reales (de objetos astronómicos) del observatorio virtual. Aunque tal vez se deba a que no sabía bien qué datos tiene el VO y, por lo tanto, no supe preguntar bien.”*
- *“Las imágenes que me retornaba el sistema no fueron muy útiles. Por otro, lado si ayudó a encontrar estrellas dada una breve descripción”*
- *“Al consultar información específica, para algunos casos se entrega una tabla con la información, sin una manera de saber qué hacer con ella, personas que se acerquen por primera vez van a necesitar esa información para entender como hacerlo. Además no se tiene una funcionalidad que permita hacer un resumen de la información para el caso en que se requiera evitar ver los datos de cada fila de la tabla que entrega.”*
- *“El chatbot entrega una tabla con ‘recursos’ de los cuales a veces los links están rotos (recomiendo verificar de parte de quien están rotos los links). Si bien entrega información de objetos conocidos como Eta carinae o Vega, no entrega información de una query general si no se entrega el nombre del objeto en específico. Tampoco provee una lista de objetos a los cuales se puede consultar. Se logro acceder a algunos archivos fits de objetos conocidos, pero no se logro obtener ningún gráfico de valor como curva de luz, o espectros de longitud de onda vs flujo.”*

Efectivamente, el problema radica en la in-eficiencia o incapacidad del sistema de recuperar información especificada, a pesar de estar equipado con las herramientas para lograrlo, y tomando en cuenta la existencia de ejemplos funcionales. Se pueden conjeturar 2 motivos los cuales son los causantes de este problema: Primero, la toma de decisiones del agente sucede en el nodo que contiene al LLM, el cual es consciente de las herramientas a su disposición y cuando debería usarlas gracias a el prompt inicial que se le entrega y las descripciones de la funciones que componen estas herramientas, las cuales fueron elaboradas por el autor de este trabajo. El “prompt tuning”[Martineau, 2023] es una disciplina iterativa en la que, a base de instrucciones semánticas, se intenta regular el comportamiento de un programa. En este caso, es posible que las instrucciones no fueran suficientemente claras para normar el comportamiento del agente, ya sea por ser muy simplistas o, al contrario, ser demasiado enrevesadas. La segunda razón está relacionada con el origen de los datos y la forma de acceder a ellos. Se estipulo en la introducción y justificación del problema que uno de los motivos de la baja popularidad del observatorio virtual es que, a pesar de esfuerzos, no hay una nomenclatura estandarizada entre observatorios virtuales (recursos). Esto vuelve difícil el buscar información referenciando ciertas palabras clave, ya que pueden estar escritas de distintas formas. Respecto al método de acceso, en el desarrollo de la aplicación se explicó que para acceder a datos, en su mayoría es necesario obtener las coordenadas celestiales de los objetos astronómicos a buscar, esto mediante la clase SkyCoord de la librería Astropy, la cuál usa el motor de traducción nombre-coordenada Sesame[Centre,]. Este método prueba ser ineficiente a momento de resolver ciertos objetos astronómicos, particularmente dentro del sistema solar, donde radica la mayoría del interés de la gente no astrónoma.

A pesar de los inconvenientes encontrados por los usuarios al momento de obtener información, la mayoría de los usuarios consideran que la aplicación cumple en cierta medida con el objetivo de facilitar el acceso a datos astronómicos, como se puede observar en la figura 17. De hecho, más del setenta por ciento de los usuarios estiman que es probable que vuelvan a usar la aplicación (como se puede observar en la figura 18), y de ese porcentaje, un sesenta por ciento está seguro de volver a utilizarlo.

Del 1 al 7, ¿En que medida consideras que la aplicación facilita el acceso a datos astronómicos?

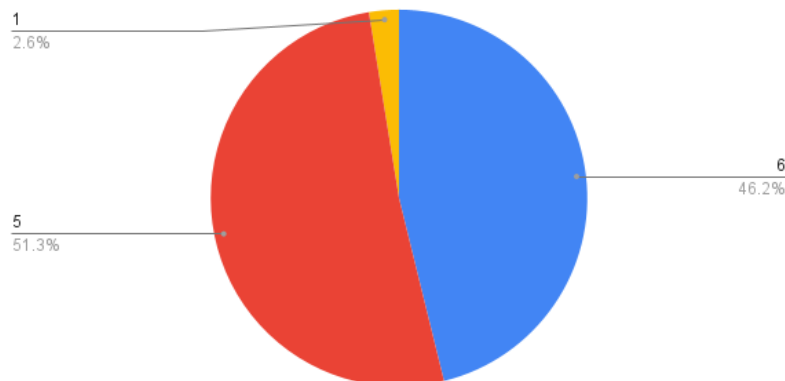


Figura 17: Calificación respecto a cumplimiento del objetivo de facilitar el acceso a datos astronómicos (rango calificación: 1 a 7).

Fuente: Elaboración propia.

¿Cuál es la probabilidad que vuelvas a usar la aplicación más adelante? Responde en la escala del 1 al 10, donde 1 es que nunca la usaras de nuevo y 10 es que estás seguro de usarla con regularidad.
8 respuestas

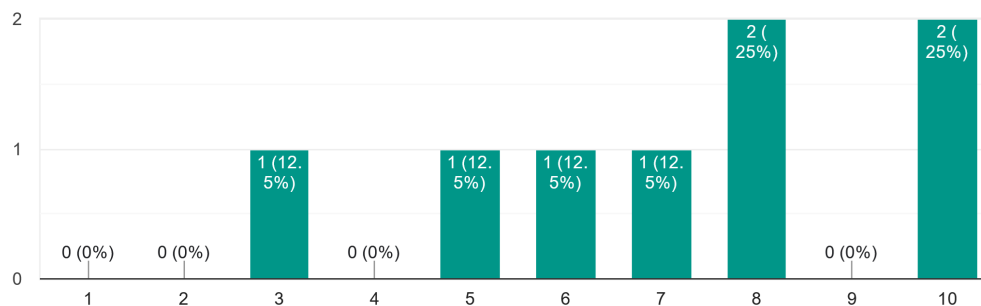


Figura 18: Probabilidades de utilización de la aplicación a futuro (rango prob.: 1 a 10)

Fuente: Elaboración propia.

Finalmente, se les solicitó a los usuarios que proporcionaran sugerencias de funcionalidades que podrían ayudar a la aplicación a cumplir de mejor manera el objetivo de proporcionar datos astronómicos reales del observatorio virtual y acercar la astronomía a la población:

- *Agregaría un historial de chats. Y también una mejor introducción sobre los tipos de datos que se pueden acceder.*
- *Que las imágenes se mostraran en el chat en vez de tener que ir a un link externo para visualizarlas..*
- *Me parece que es difícil identificar la utilidad de las respuestas otorgadas en tablas, creo que sería mejor tener un resumen de cada objeto o conjunto de datos encontrado junto al link de acceso, el cual en caso de ser de interés contendrá la metadata que los astrónomos buscan.*
Me parece que también sería bueno que se entregara el tipo de dato contenido en cada recurso para ahorrar tiempo.
De ser posible, sería bueno poder tener un “preview” de los datos, gráficos de espectros, magnitudes en filtros, mediciones que sean interesantes según el contexto de la pregunta.
- *Que la aplicación explique mejor qué clase de datos tiene el VO. Por ejemplo, que indique primero un listado de los tipos de objetos a cuyos datos se puede acceder (supernovas, estrellas, galaxias, etc.). Y luego de seleccionar un tipo de objeto (como estrellas por ejemplo), te diga qué clase de datos tiene (luminosidad, distancia, curvas de luz, metalicidad, etc.).*
- *Recibir la imagen al instante, en vez de un link.*
- *Funcionalidad de visualizar las imágenes desde un enlace de la tabla, o la posibilidad de poder entregar instrucciones para acceder a datos de tablas. O también poder hacer un tutorial guiado para poder saber el uso de la aplicación para personas que no conocen astronomía.*
- *Iniciar una sección limpia de preguntas.*
- - *Poder consultar lista de objetos por tipo en la base de datos*
 - *Que muestre gráficos directamente en el chat*
 - *Que muestre los espectros de los objetos que se le piden (si esta disponible, el procesado, si no el fits).*
 - *Poder ver la curva de luz del objeto consultado.*
 - *Hacer consultas por coordenadas centradas con un radio definido.*
 - *Que se pueda descargar un archivo .dat con la información consultada*
 - *Poder consultar por las bandas de ciertos objetos, las fechas en que se observo.*
 - *Obtener todas las referencias de un objeto*

- *Agregaría ejemplos mas complejos y una guía de como interactuar con las tablas de "recursos"*

Dentro de estas sugerencias podemos encontrar funcionalidades que mejorarían la facilidad de uso y experiencia usuario (historial, muestra de imágenes, nuevo chat), otras que permitirían no solo un mejor uso de la herramienta para el usuario, sino una mejor comprensión de los datos visualizados (resumen de objeto, tutorial guiado, tipos de objeto y datos que se pueden obtener de este), pero también se vuelve a evidenciar un problema con el comportamiento del asistente, ya que en la sección de propuesta de solución se describe una herramienta cuyo propósito es efectivamente el de mostrar una imagen al usuario, pero distintos usuarios sugieren esta funcionalidad.

CAPÍTULO 5

CONCLUSIONES

Una vez analizados los datos y el “feedback” obtenido mediante las pruebas de usuarios con la aplicación, se procede a describir las conclusiones que se pueden obtener de los datos obtenidos.

5.1. Cumplimiento de los objetivos

La primera conclusión y más importante que se debe obtener del trabajo es si los objetivos planteados al principio fueron llevados a cabo de forma exitosa, y los motivos por los cuales se logró o no.

Empezando por el objetivo general (1.2.1) de crear una aplicación web con acceso a los observatorios virtuales. Tomando en cuenta el desarrollo exitoso de la solución y que el consenso entre los usuarios es que, efectivamente, la aplicación facilita el acceso a estos datos del observatorio virtual, es correcto afirmar que el objetivo fue logrado. Específicamente se logró crear un asistente virtual en su completitud, no así el darle acceso completo sobre el Observatorio Virtual; las herramientas creadas y el Agente que toma la decisión de utilizarlas pueden mejorarse, demostrado por las sugerencias de los usuarios y los problemas al momento de obtener cierto tipo de información.

Considerando los resultados obtenidos respecto a la facilidad de uso de la aplicación, es prudente concluir que se logró crear una interfaz fácil de usar para usuarios, no así el que puedan acceder a información sin tener que investigar como. Si tomamos en cuenta las sugerencias entregadas como “explicación de que clase de datos posee el VO o cada tipo de objeto” y también el “poder hacer un tutorial guiado”, es evidente que hace falta una explicación sucinta o una guía para acostumbrar a los usuarios a usar la aplicación de forma que sepan obtener resultados del VO.

5.2. Trabajo futuro

Los resultados de este trabajo, aunque no cien por ciento positivos, demuestran que todos los objetivos son alcanzables y hay lugar para una mejora sustancial en la aplicación. Los usuarios concluyen que este asistente al observatorio virtual si tiene futuro como herramienta de divulgación científica y como apoyo en la investigación, siempre y cuando se sigan las sugerencias y se pueda asegurar un funcionamiento sin interrupciones y exitoso.

Formas en los que se puede mejorar la herramienta es, por ejemplo, creando un sistema de

tutorial guiado para demostrar el correcto funcionamiento y las utilidades de la aplicación. También funcionalidades de experiencia usuaria como lo son un historial de chats o la capacidad de crear una nueva conversación, sin memoria de los mensajes previos. Otro punto importante es el de experimentar con distintas prompts iniciales, más y menos específicas, para poder obtener un raciocinio correcto por parte del Agente.

Finalmente, el asistente agente con herramientas y capacidad de recuperación de documentos es un trabajo prometedor. Los usuarios no astrónomos mostraron interés en la capacidad que tiene el asistente de contestar sus preguntas científicas con certeza, además de poder acceder directamente a bases de datos astronómicas. Pero, al mismo tiempo, sugieren una representación más visual, y asistencia para poder utilizar la herramienta al cien por ciento. Por otro lado, los usuarios profesionales de la astronomía mostraron interés en el potencial que tiene el asistente para poder facilitar el trabajo que, de otra forma, tomaría semanas. De todas formas, también sugieren modificaciones, como una representación más compuesta de los elementos, y una forma más “asistida” de utilizar el asistente.

El Asistente Virtual para el VO es una propuesta prometedora que, de ser pulida y trabajada, podría ser ampliamente utilizada, ampliando el alcance de la astronomía, facilitando la investigación y acercando a la gente al trabajo detrás de todas las imágenes que nos dejan maravillados con la belleza del universo y fascinados con el misterio del cosmos.

ANEXOS

REFERENCIAS BIBLIOGRÁFICAS

- [Borne, 2010] Borne, K. D. (2010). Astroinformatics: Data-oriented astronomy research and education. *Earth Science Informatics*, 3(1-2):5-17.
- [Centre,] Centre, S. A. D. Accessed on March 2nd, 2024.
- [Contributors,] Contributors, V. I. . V. Next generation frontend tooling. Accessed on June 28th, 2024.
- [Developers, a] Developers, P. Pyvo. Accessed on Feb 4th, 2024.
- [Developers, b] Developers, P. Pyvo. Accessed on Feb 4th, 2024.
- [ESO, 2002] ESO (2002). Toward an international virtual observatory. Accessed on July 6th, 2023.
- [Goodwin, 2024] Goodwin, M. (2024). What is an api?
- [Group, 2022] Group, T. B. A. (2022). Topcat. Accessed on July 7th, 2023.
- [Hanisch et al., 2001] Hanisch, R. J., Farris, A., Greisen, E. W., Pence, W. D., Schlesinger, B. M., Teuben, P. J., Thompson, R. W., y Warnock, A. (2001). Definition of the flexible image transport system (fits). *Astronomy and Astrophysics*, 376(1):359-380.
- [Harrison Chase,] Harrison Chase, A. N. Langchain for llm application development. Accessed on Jan 25th, 2024.
- [Iafrate, 2019] Iafrate, G.; Ranella, M. (2019). Vo for education: Virtual and remote observing. En *Astronomical Data Analysis Software and Systems XXVII*, volumen 522 de *ASP Conference Series*.
- [IBM, 2024] IBM (2024). Ibm watsonx, una plataforma de ia y datos creada para las empresas.
- [IVOA, 2023] IVOA (2023). What is the ivoa. Accessed on July 6th, 2023.
- [Kasliwal, 2023] Kasliwal, N. (2023). Binary quantization - vector search, 40x faster.
- [LangChain, a] LangChain. Introduction. Accessed on Jan 8th, 2024.
- [LangChain, b] LangChain. Langgraph. Accessed on Jan 8th, 2024.
- [LangChain, c] LangChain. Langsmith. Accessed on Jan 10th, 2024.
- [Martineau, 2023] Martineau, K. (2023). What is prompt-tuning?
- [Merritt, 2024] Merritt, R. (2024). What is retrieval-augmented generation aka rag?

- [Mhatre, 2023] Mhatre, S. (2023). Vite vs create react app: Choosing the right tool for your react.js project.
- [Murphy, 2023] Murphy, M. (2023). What are foundation models?
- [naki Ortiz, 2008] naki Ortiz, Jeff Lusted, P. D. A. S. Y. S. M. A. N. S. M. O. W. O. P. O. (2008). Ivoa astronomical data query language.
- [Postman,] Postman, I. What is postman? Accessed on July 6th, 2024.
- [Pydantic,] Pydantic. Pydantic. Accessed on Jan 11th, 2024.
- [Qdrant,] Qdrant. High-performance vector search at scale. Accessed on Apr 20th, 2024.
- [Quinn *et al.*, 2002] Quinn, P., Benvenuti, P., Diamond, P., Genova, F., Lawrence, A., y Mellier, Y. (2002). Astrophysical virtual observatory (avo): a progress report. *Proceedings of SPIE - The International Society for Optical Engineering*.
- [SAS,] SAS, M. U. Move faster with intuitive react ui tools. Accessed on July 2nd, 2024.
- [Team, 2023] Team, A. (2023). Astropy | about. Accessed on July 7th, 2023.
- [Tiangolo,] Tiangolo. Fastapi. Accessed on Apr 2nd, 2024.
- [Uszkoreit, 2017] Uszkoreit, J. (2017). Transformer: A novel neural network architecture for language understanding. Accessed on July 8th, 2023.
- [Wiik, 2024] Wiik, L. (2024). Best embedding mode openai / cohere / google / e5 / bge.
- [Yao *et al.*, 2022] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., y Cao, Y. (2022). React: Synergizing reasoning and acting in language models.

Tiempos SCT [horas]	Planificación	Búsqueda de Información	Análisis	Desarrollo	Edición	Total
	3	10	13	22	3	51 horas