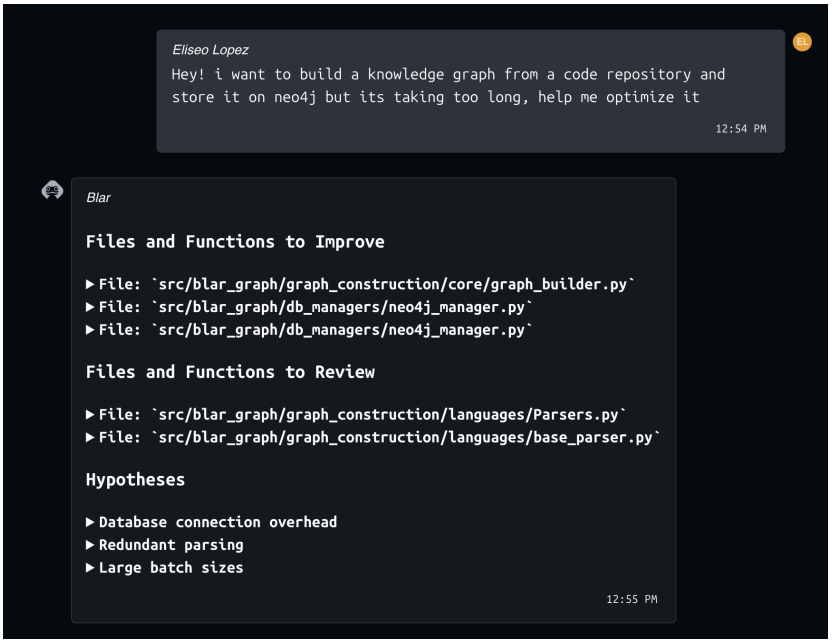
Prueba Técnica - Eliseo López

Primer acercamiento al problema, leer el código, hacer la cuenta e instancia de Neo4j y correr el programa. El único cambio fue en la función skip_files que omita archivos que no sean de código (.pdf, .jpg, etc)

Luego leer y entender el archivo graph_builder.py para construir el archivo y neo4j_manager.py para el envío de información, y visualizar el grafo.

Luego, subí código al repositorio privado para conectarlo a Blar y aprovechar el agente optimizador



En base a esta conversación queda claro que hay dos puntos relevantes, el uso de multithreading y el uso de caches, principalmente en el archivo graph_builder.py

<u>Multithreading:</u>
En primera instancia, esta solución me complica debido a que el uso de varios threads hay que tener cuidado con un buen uso de locks para no perder información.

```
    # Add entries to the queue
    for entry in os.scandir(path):
        entry_queue.put(entry)

    # Use a thread pool to process entries
    num_threads = 4
    with ThreadPoolExecutor(max_workers=num_threads) as executor:
        for _ in range(num_threads):
            executor.submit(process_entry)

    entry_queue.join()
```

Agregue estas líneas para crear un multithread en el método *_scan_directory*, y process_entry hace el escaneo con un lock para crear los nodos y relaciones correspondientes, sin embargo, probando de distintas maneras y cantidad de threads no logre reducir el tiempo original.

Caches:

El uso de caches puede ahorrar tiempo pero aumenta la memoria del programa, el uso del decorador @lru_cache en métodos como _skip_file, puede ahorrar tiempo en archivos con nombres repetidos como config.py o __init__.py que se pueden ver varias veces en un proyecto, pero esta alternativa también agregaba tiempo debido a que los métodos en los que se empleó eran relativamente baratos computacionalmente o no lo suficientemente repetitivos. Sin embargo, agregarlo al método _is_package logro levemente reducir los tiempos.

```
@lru_cache(maxsize=None)
def _skip_file(self, path: str) -> bool:
    skipped_extensions = {".png", ".bmp", ".jpeg", ".sqlite3", ".gif", ".pkl", ".msg", ".jpg", ".pdf"}
    skipped_filenames = {"package-lock.json", "yarn.lock"}

    if path.startswith(".") or path.endswith("lock") or any(path.endswith(ext) for ext in skipped_extension
        return True
    if path in skipped_filenames or (self.skip_tests and ("test" in path or "legacy" in path)):
        return True
    return False
```

Ajustes generales:

En el método _scan_directory es el que maneja el grueso del programa, por el que intente optimizarlo analizando sus subpartes, y un punto de mejora es el uso de generadores, la linea:

"entries = (entry for entry in os.scandir(path) if not self._skip_file(entry.name) and not self._skip_directory(entry.name))"

Evita que entren al loop principal entradas que deberían ser saltadas, y evita malgastar recursos. Ademas, de reescribir ciertas partes del código y analizando cómo las estructuras de datos (listas, sets, diccionarios) afectan a la velocidad del programa.

Conclusiones:
El uso de threads para escanear los archivos en teoría parece ser una opción muy viable que se debe investigar en profundidad para llegar a la opción más óptima del programa, y también para el envió de información a Neo4j. Además, el uso de cache fue probado pero muchos métodos retornan objetos diferentes (como nodos) por lo que difícilmente va a crear un cambio radical. Además, es necesario probar con más de un proyecto para ver como afecta el rendimiento con varios casos de prueba.

Resultados:

En los resultados, probando con el repositorio de llama-index la diferencia es mínima, pero ahondar en los puntos anteriores se puede obtener mejores resultados.

Tiempo de ejecución original:

Tiempo de ejecución mi programa: