# Documentation auto-kg-lit

Dorus Keijzer

June 1st 2025 - September 23, 2025

## 1 Introduction

This file documents the state of the auto-kg-lit project as of the state of writing this, with the intention that the next person working on this project knows what to expect of the state that it was left in, how the program is structured so far and how to set it up.

Should anything by left unclear to the point that you cannot figure it out yourself, you can contact me (the author of the program as well as this document) at `doruskeijzer@gmail.com`

## 2 Overview

This project is part of a larger research project that aims to consolidate scientific data about sustainable building design in one knowledge graph in order to create a data foundation for an interactive decision-support system tailored for sustainable building design.

The point of this particular project was to create a program that takes as input a research paper, or a set of research papers which relate sustainable building design strategies to ecosystem services and to output a knowledge (sub)graph that displays the links between these strategies, services, the specific example buildings they are employed in, any commercial products that employ a specific strategy, etc.

To this end, we have created a program that is able to take in a (set of) research papers (in pdf form), and produce a knowledge graph from these based on specific prompts and other configurations. Two problems remain: The first problem is that with the current instructions, the output does not closely match our intent (i.e. the knowledge graph produced is of poor quality), hence: what is left to be done is to find the exact prompts and configurations that cause this system to produce a knowledge graph that is closest to our expectations. The second problem is getting the knowledge graphs visualized to Katherina Hecht, an ecologist who will revise the extracted data for their validity and further categorize and recognize them. To allow for ease of visualization we use GraphPolaris: a graph database visualization program. Setting this up should be done in cooporation with the people at GraphPolaris.

## 3 Getting started

When I was on this project I was granted access to the UU server on `http://hpvig.science.uu.nl/`. If the code is still there, it should be in `BETAL109242 ~/AUTO-KG-LIT`. If you have access to this server and the files are still there, you should be able to continue from there.

If you cannot continue on this server for whatever reason or wish to develop locally, you are first of all required to install docker (including docker compose) and you ccan then pull all the different microservices from GitLab. They can be found at `https://git.science.uu.nl/auto-kg-lit`. Each microservice defines its dependencies using poetry, so make sure you are familiar with that. The README of the orchestration service contains important information on how to set up docker such that it can access the GPU.

The program is written with the following folder structure in mind. Make sure to structure your directories the same way:

```
> orchestration
> microservices
| > your microservices
```

Most of the microservices require a `.env` file where you specify certain environment variables. There should be `.env.template` files in there that provide a template as to how these should be constructed.

The commands to run the knowledge extraction service is documented in 4.1.1.

## 3.1 List of microservices

The full program consists of a few different microservices which are listed below. Some of the microservices in this list should be considered "vestigial", as in: no longer useful to the knowledge extraction project. They were initially created before the scope of the project reduced from "automated literature search + knowledge graph extraction" to just "knowledge graph extraction". These vestigial microservices are marked with a star in the list below. If you are not concerned with automated literature search, you might as well disregard them (as in: please don't bother to set them up if you don't know for sure that you will need them. Most likely you don't). I kept them in and briefly documented them here to avoid confusion as to why they exist at all and perhaps they might be useful again in the future, should the scope of the project expand back again.

1. Knowledge Exctaction
2. Orchestration
3. Frontend *
4. Backend *
5. API *
6. Document Database *
7. Paper Crawler *

# 4 microservices

## 4.1 Orchestration

This repo orchestrates the other microservices, including the neo4j instance, a postgres instance and the ollama service. From here you can run the knowledge extraction service in 2 ways: either as a standalone docker compose setup or by running the script `run_experiment.py` (both will be explained below). This repo is where you are expected to call the program from. The README inside the repo provides more information and useful commands than discussed here.

### 4.1.1 docker compose files

This microservice contains 2 main docker compose files: `docker-compose.yml` and `docker-compose.gpu` The former contains the main services and should be able to be run on a machine without access to a (cuda)-gpu (so that you can run and develop the program locally even if you don't have a gpu), the second one is meant to be only used in conjunction with the former and allows for those services that benefit from being run on the GPU to be run on the GPU (very much recommended if you have GPU access, unless you want to waste hours waiting).

Running the knowledge extraction service without the GPU is done by the following command:

```
docker compose --profile extract up
```

Running *with* the GPU is done by:

```
docker compose --profile extract -f docker-compose.yml -f docker-compose.gpu.yml up
```

`docker-compose.yml` contains the services: neo4j, ollama, postgres as well as each of the microservices (including those that are not relevant to knowledge extraction). The ones relevant to knowledge extraction are neo4j (for hosting the resultant knowledge graph), ollama (for hosting the LLM that produces said knowledge graph) and the knowledge extraction service (for prompting the LLM and writing the results to neo4j). As seen in the example above, you can call different subsets of these services through their *profiles*, the profile that is relevant for knowledge extraction is called `extract`.

### 4.1.2 Config file

The config files for knowledge extraction are stored in the orchestration repo. A config file is a json file that allows you to specify 1. the LLM that you are using 2. The embedding model that you are using 3. the schema that this LLM uses 4. a system prompt for this LLM. Config files should be stored in `orchestration/experiments`. In here, you can currently find two examples: `orchestration/experiments/test_config.json` and `orchestration/experiments/test_config_2.json`. Those file currently use environment variables to define the LLM and embedding model, because of a script that automatically pulls the LLM based on the context of your .env file, but you could also define them in the config themselves, as long as you make sure the model is pulled from Ollama.

The **LLMs** and **embedder models** are is hosted locally via the Ollama service and you should therefore be able to choose any LLM pulled from `https://ollama.com/search`, as long as it fits on your GPU.

The **schema** defines the types of nodes and relationships that can occur within your resulting knowledge graph, and "patterns", i.e. how these nodes and relationships relate to eachother: which nodes are allowed to be linked to each other and through which relationships.

Nodes are specified under `node_types` and can take a label (a string), description (a string) and a list of properties (each with a name, a type and whether this property is requierd). An example of a node is:

```
"label": "Ecosystem service",
"description": "A benefit humans derive from the ecosystem.
From the list: 'Regulation of temperature', 'Regulation of water quality',
'provision of habitat', etc.",
     "properties": [
       {"name": "name", "type": "STRING", "required": true}
  ]
```

The LLM will attempt to find anything in a document that matches this description and if it finds something, it will create a node in neo4j with the label "Ecosystem service" and the property "name" reflecting the name of the ecosystem service. In the example, the property "name" is required, meaning that it forces this node to have the property "name" and will not create a node if it cannot find a name. A node/relationship can have multiple properties and properties can also *not* be required, in that case it will only fill in the property if it finds something matching the property.

Relationships are defined under `relationship_types`. Relationships also take a label, description and a list of properties identically to how nodes are specified.

Patterns are specified under `patterns`: a list of triples of the form `[node, RELATIONSHIP, node]`. E.g.: `["Design strategy", "EVIDENCE", "Ecosystem service"]`. This pattern will ensure that relationships of the type "EVIDENCE" will only occur between nodes of the type "Design strategy" and "Ecosystem service".

The **system prompt** is defined in the `prompt_template` as a string. Here you give instructions to the LLM on what it is tasked with. Cursory testing proved to me that providing examples output really helps with ensuring correct output (see the example config files for how this is done). Here you need to be really really really careful that your prompt does not mess up the json formatting of the config file, ultimately the config file needs to remain a valid json and you can easily mess this up if you don't escape quotation marks properly. Pro tip: you can use the `jq` utility on linux to see if you formatted your file correctly.

### 4.1.3 run_experiments.py

This script orchestrates the running of multiple experiments in succession (here, an experiment entails: running the pipeline for all config files in `experiments` over all papers in `test_papers` and storing the resulting graph). It does so by creating the temporary folder structure for a given experiment, setting up the environment variables and then calling the docker-compose file from these temporary directories as a subproces. For each config, it ultimately creates a dump of the state of neo4j written to a `.cypher` file. This

is stored in `/results`. This dump can be opened in Neo4j again if you want to access the resulting knowledge graph.

You can run this file by calling

```
poetry run run_experiments.py
```

from the root of the orchestration repo.

### 4.1.4 Evaluation

In order to compare the quality of a resulting graph, the plan was to compare them to a hand annotated gold-standard graph created for the following papers: "Green roof systems for Rainwater and Sewage Treatment" by Yan at al (2024), "Temperature reduction effects of rooftop garden arrangements" by Kim et al (2020), "Management strategies for maximizing the ecohydrological benefits of multilayer blue-green roofs in mediterranean urban areas" by Cristiano et al (2023) and "Hydrologic and thermal performance of a full-scale farmed blue-green roof" by Alamaaitah et al. (2022), where Katharina highlighted the relevant sections. The code to perform this evaluation, as well as the decision on which graph similarity metrics to focus on is left for you to do.

## 4.2 Knowledge Extraction Service

This microservice is the part that ultimately reads papers it is given and produces the resulting knowledge graph. Keep in mind that you are not expected to call this microservice from this repo directly, but rahter from the orchestration service.

The knowledge extraction portion of this program closely follows the instructions found on *thisuserguideprovidedbyneo4j* [1], so that is a good source to consult if you notice discrepancies between the information on the currently live neo4j site and the program.

The code lives in `./src/main.py`. This code requires the `NEO4J_URI`, `NEO4J_USER`, `NEO4J_PASSWORD` and `APP_CONFIG_PATH_IN_CONTAINER` environment variables to be set to respectively the URI where Neo4j is being run, the username that is set for the Neo4j database it should write to, the password that is set for Neo4j and where the config file is stored in the container that runs it. If you are running it using the docker compose files from the orchestration repo, this will be done for you. If these are not set, the program will exit with status code 1.

### 4.2.1 Pipeline

The program will start by creating a `PipelineRunner` object from the Neo4j Experimental library from the config file that is specified. See section 4.1.2 for what can be specified. The pipelineRunner object is responsible for ingesting papers and writing to neo4j.

Between ingesting papers and writing to neo4j, it splits the paper up in several smaller chunks ( 30 for one research paper) to make sure they are able to fit in the context length of the LLM you are using. Then, for each contiguous triple of chunks (i.e. chunks 1,2,3 together, chunks 2,3,4 together, chunks 3,4,5 together etc.) it follows the specifications of the schema and system prompt stored in the config file (from section 4.1.2). If it succeeds

[1] Perhaps this archived version: `https://web.archive.org/web/20250522160128/https://neo4j.com/docs/neo4j-graphrag-python/current/user_guide_kg_builder.html#user-guide-kg-builder` is even better, should they make changes to that page after I have written this

in finding a part of the text that matches the description of a node or relationship, it fills in the relevant schema and creates a json object. This json is then parsed and passed to Neo4j to create a new note/relationship. It will also create a node for each chunk in the paper, which has a relationship to every node created from that chunk. This way you can see which part of a document a node came from.

It does this for every paper one by one and if it finishes succesfully, it exits with status code 0.

### 4.3 API, frontend, backend

These are somewhat relevant to knowledge extraction, but I doubt their importance to whatever you might be tasked with. At some point in the project when the GPU integration was not done succesfully, the knowledge extraction service ran really slowly (because it was not utilizing the GPU correctly). Because this took so long, I saw value in being able to schedule "experiments" (i.e. testing out the results of a given configuration file) in a queue. The frontend/backend/API repos were meant to give the user a graphical interface to check in on the process of this queue. These were largely vibe coded and abandoned after we fixed the faulty GPU integration because they lose their usefulness when experiments did not take nearly as long.

### 4.4 Paper Crawler

Not relevant to knowledge graph extraction, so I will be brief: at some point the scope of the project included automated literature search. This microservice was intended to do that by scraping the Semantic Scholar API using relevant keywords. This repo was abandoned when we reduced the scope.

### 4.5 Document database

Again: not relevant to knowledge graph extraction, this mostly just holds a docker file to spin up the document database that the paper crawler would have written to.

## 5 Passwords and usernames

The ones used on the hpvig.science.uu.nl server should be recoverable from the .env files in the repos, but if for some reason you lose them: usernames are typically the name of the service themselves, i.e. "neo4j" is the username for the Neo4J service. Passwords are either `476ADRome` for neo4j and `1453ADConstantinople` for the postgres service.