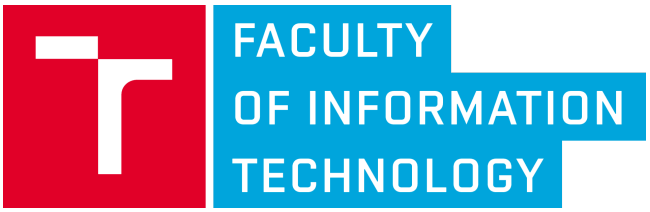


Technická Dokumentace k Software Topicer - Software pro analýzu témat v dokumentech

Michal Hradiš, Martin Dočekal,
Martin Kostelník, Martin Kišš, Martin Fajčík,
Richard Juřica, Marek Sucharda

Vysoké učení technické v Brně Brno
2025

Tento dokument byl vytvořen s finanční podporou MK ČR v rámci programu **NAKI III program na podporu aplikovaného výzkumu v oblasti národní a kulturní identity na léta 2023 až 2030** v projektu semANT - Sémantický průzkumník textového kulturního dědictví.

Číslo a název projektu:

DH23P03OVV060	semANT - Sémantický průzkumník textového kulturního dědictví

Název a popis dílčího výstupu:

Topicer - Software pro analýzu témat v dokumentech
Tento dokument popisuje funkcionalitu a použití software Topicer, který umožňuje analyzovat témata v kolekcích textů a navrhovat tagy pro texty. Software poskytuje python balíček se základní funkcionalitou, REST API i samostatně nasaditelné služby (např. výpočet textových embeddingů). Software poskytuje několik metod využívajících různé přístupy včetně modelů natrénovaných speciálně pro navrhování tagů v češtině.

Jazyk dokumentu

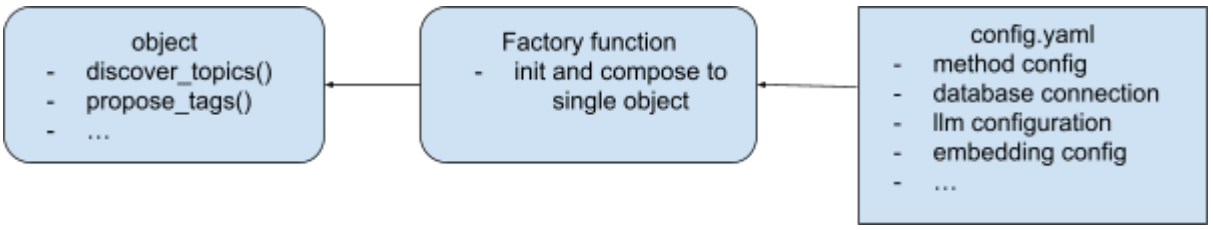
Angličtina

Organizace a řešitel

Vysoké učení technické v Brně	Ing. Michal Hradiš Ph.D.

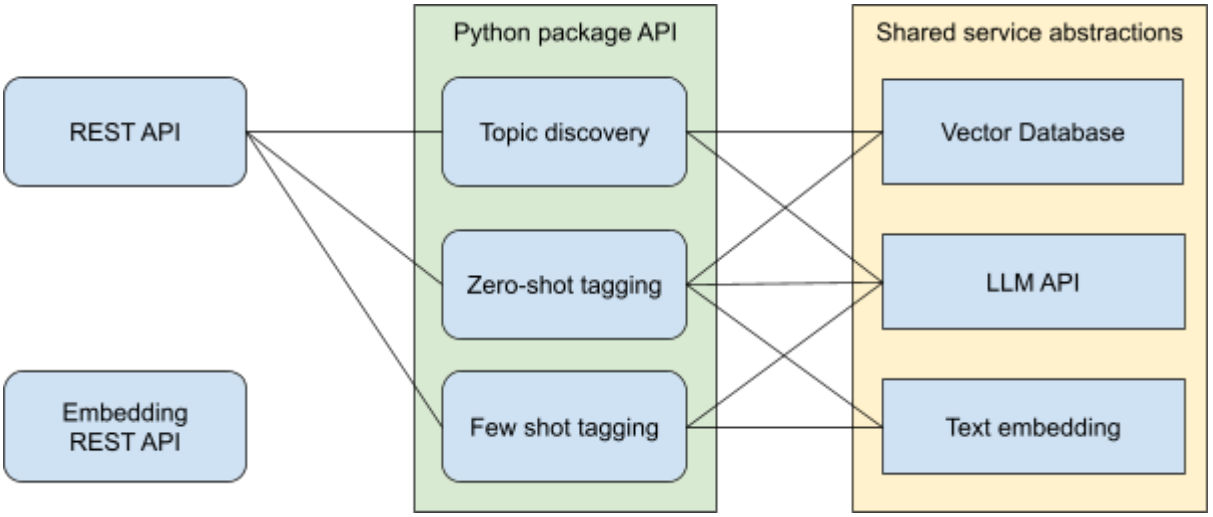
Usage overview

Topicer includes a python package for topic discovery and general tagging, pre-trained models and REST API for simple deployment and integration. The package is optimized for ease of use - it provides a single factory function which builds an object based on a config file. These objects have a unified API and are self-contained.



Alternatively a REST API can be directly deployed. The REST API can host a number of initialized methods.

The package includes shared abstractions for vector databases, LLM APIs (Ollama, OpenAI, …) and text embedding (both local inference and APIs). It also includes a simple Embedding REST API which can be used by the shared text embedding service.



The package provides these types of top level functionalities:

- Topic discovery - Given a number of text fragments, the system proposes topics, names them, describes them and provides the estimated assignment between the topics and text fragments.
- Zero-shot tag proposal - Given a tag **name** and an optional **description**, the system finds probable instances of a tag in a large collection of text fragments or proposes probable tagging of a single text with multiple tags.
- Few-shot tag proposal - This is similar to Zero-shot tag proposal, but the system can use examples of previously tagged texts.

Availability

The software is available from https://github.com/DCGM/topicer including trained models. The git repository includes:

- Python package topicer
- REST API topicer server
- REST API text embedding server

The latest version of python package can be installed directly from the git repository by running:

```
pip install git+https://github.com/DCGM/topicer
```

You can install specific version by running:

```
pip install git+https://github.com/DCGM/topicer@TAG
```

List of released versions is available at: `https://github.com/DCGM/topicer/tags`

Links to all available models are available in the README.md file.

License

BSD 3-Clause License

Software and hardware requirements overview

In general a python 3.12 is recommended.

API deployment using Docker requires standard Docker Engine and Docker Compose installation and Nvidia container toolkit.

Some of the methods use OpenAI or other commercial models and may require a valid API key (e.g. for OpenAI https://platform.openai.com/). Alternatively local inference using a Ollama inference server is supported as an alternative.

Some methods require running models locally. Those generally require a GPU or other accelerator compatible with PyTorch.

Top level package API and structure

The API uses pydantic objects which are defined in `topicer/schemas.py` and the API interface is defined in `topicer/base.py` including a factory function and configurable abstract classes.

In general, any method implemented in the package can be initialized and used by:

```
from topicer import factory
method_instance = factory("method_config.yaml")
# Example of use
method_instance.propose_tags(text_chunk, tags_to_propose)
```

The method instances support the following API:

```
async def discover_topics_sparse(self, texts: Sequence[TextChunk], n: int | None = None) -> DiscoveredTopicsSparse:

async def discover_topics_dense(self, texts: Sequence[TextChunk], n: int | None = None) -> DiscoveredTopics:

async def discover_topics_in_db_sparse(self, db_request: DBRequest, n: int | None = None) -> DiscoveredTopicsSparse:

async def discover_topics_in_db_dense(self, db_request: DBRequest, n: int | None = None) -> DiscoveredTopics:

async def propose_tags(self, text_chunk: TextChunk, tags: list[Tag]) -> TextChunkWithTagSpanProposals:

async def propose_tags_in_db(self, tag: Tag, db_request: DBRequest) -> list[TextChunkWithTagSpanProposals]:
```

Each API function has two variants:

- Without `in_db` suffix - processes text chunks passes as a parameter
- With `in_db` suffix - processes text chunks retrieved from a database. db_request object must be provided as an argument to limit the scope.

`TextChunk` objects represent identifiable text fragments. These can represent any part of a document, but for most use cases where vector-based semantic retrieval is used, should contain 500 to 4000 characters.

Tag proposals. Functions `propose_tags()` and `propose_tags_in_db()` return objects `TextChunkWithTagSpanProposals` which contain list of `TagSpanProposal` objects. `TagSpanProposal` objects contain an identified `Tag`, its position in the text represented as `span_start` and `span_end` which represent character positions in the text. Some tag proposal methods provide `confidence` value or **`reason`**

(a natural language description why the tag was assigned to the text).

Topic discovery. The topic functions return either `DiscoveredTopicsSparse` or `DiscoveredTopics` objects. They both contain a list of distinct topics identified in the processed texts and an assignment between the identified topics and texts. The difference is in how the assignment is represented. The dense representation (`DiscoveredTopics`) contains a single floating point value between 0 and 1 for each topic-text pair. The sparse representation contains a list for each identified topic which contains pairs of text enumerator and score only for strongly associated text-topic pairs (the cutoff could be based on score threshold or other mechanism defined by the specific method).

Errors. Topicer methods typically do not implement all six functions. Typically, only `discover_topics…` methods or `propose_tags…` functions are supported by a specific method. If a function not implemented by a topicer method is called, an exception `NotImplementedError` is raised.

Repository structure

- `deploy/ - docker deployment files`
- `docs/` - markdown documentation for the individual topicer methods
- `examples/` - example usage and configuration
- `tests/` - pytest unit tests
- `embedding_service/` - Text embedding service with REST API.
- `topicer_api/` - REST API implementation providing access to topicer methods.
- `topicer/`
    - `database/` - Implementations of database service abstractions.
    - `embedding/` - Implementations of text embedding service abstractions.
    - `llm/` - Implementations of LLM API service abstractions.
    - `tagging/` - Implementations of tag proposal methods (generally implementing functions `propose_tags/ and propose_tags_in_db)`
    - `topic_dicscovery/ –` Implementations of topic discovery methods.
    - `utils/` - Shared functions and
    - `base.py` - Abstract classes and factory function. This is the definition of topicer public API.
    - `schemas.py` - Pydantic classes for the top-level API.
    - __init__.py - Has to import

Configuration mechanism

The topicer package uses package `classconfig`[1] for configuration. With this package classes have class attributes which can be directly initialized from yaml configuration. The package `classconfig` provides robust config parsing, error handling and automated config documentation generation. Each configuration file is loaded and processed using `topicer.factory()` function which attempts to initialize an object TopicerFactory. TopicerFactory includes a single topicer method and optionally one of each of the shared service types. The shared services are then passed to the topicer method instance which checks if the required service abstractions are available.

General structure of a configuration file is:

```
topicer:
  cls: ClassNameImplementingATopicerMethod
  config:
    # Any method-specific parameters
llm_service:
  cls: ClassNameImplementingALLMService
  config:
    # Any service-specific parameters
db_connection:
  cls: ClassNameImplementingADatabaseService
  config:
    # Any service-specific parameters
embedding_service:
  cls: ClassNameImplementingAnEmbeddingService
  config:
    # Any service-specific parameters
```

Contributions

Topicer is designed to be easily extendable. You can implement new topicer methods and shared service abstractions. Topicer methods should be placed in separate files in `tagging/` or `topic_discovery/` directories - a single file per method should be placed in these directories with a name matching the method class name. If the code can not be contained in a single .py file, create a subdirectory with the same name (except the extension) and place further code there. The new methods should follow this template:

```
from classconfig import ConfigurableMixin, ConfigurableValue
from topicer.schemas import TagSpanProposal, TextChunk, DBRequest,
Tag, TextChunkWithTagSpanProposals

class NewTopicerMethod(BaseTopicer, ConfigurableMixin):
     config_parameter_1       =       ConfigurableValue(desc="First
```

[1] classconfig python package - Pypi: https://pypi.org/project/classconfig/, github repository: https://github.com/mdocekal/classconfig

```
parameter")
      …

      def __post_init__(self):
            # any initialization code

      def check_init(self):
            # Check if required "shared services" are available
            # in self.llm_service, self.db_connection,
            # or self.embedding_service.
            # Raise MissingServiceError if some are missing.

      # Implement any of:
      # discover_topics_sparse, discover_topics_dense
      # discover_topics_in_db_sparse, discover_topics_in_db_dense
      # propose_tags, propose_tags_in_db
```

Shared services should be implemented similarly, placed in one of the corresponding directories (database/, embedding/ or llm/) and should inherit from BaseLLMService, BaseDBConnection or BaseEmbeddingService.

IMPORTANT: All classes have to be imported in topicer/__init__.py. That registers the implementation.

For **proposals of changes** which affect the package API or any behavior or logic not contained in a single method or service, create an issue on GitHub. Pull requests of new functionalities following the existing package logic may be accepted even without previous discussion and approval - changes to the API will not.

Topic Discovery

We implemented a subpackage, **topic_discovery**, that performs unsupervised topic discovery. In addition to identifying topics, the subpackage supports automatic topic naming in natural language, provides an explanation for the generated topic name, and produces a detailed topic description with a focus on temporal events.

Topic names are generated using a large language model (LLM) based on word-level and document-level representatives of each topic. Topic descriptions are created from the generated topic name, document representatives, and a randomly selected set of example documents containing temporal expressions.

For topic discovery, we use the FASTopic[2] library. Document embeddings are obtained using the SentenceTransformers[3] Python package with a model based on Gemma2[4]. Because the processed data is in a highly inflective language (Czech), we apply lemmatization using

[2] Wu, X., Nguyen, T., Zhang, D., Wang, W., & Luu, A. (2024). FASTopic: Pretrained Transformer is a Fast, Adaptive, Stable, and Transferable Topic Model. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

[3] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics.

[4] https://huggingface.co/BAAI/bge-multilingual-gemma2

Morphodita[5]. Lemmatization is applied only to inputs for FASTopic and is not used for inputs to the LLM.

Topic naming and description generation are performed using the gpt-5-mini model.

The topic discovery process can operate directly on provided text chunks/documents, or it can process data retrieved from a database. The database may also store precomputed vector embeddings, enabling significant computational savings by reusing existing representations.

The entire pipeline is fully configurable via YAML configuration files. This configuration system provides substantial flexibility; for example, it allows the use of different LLMs and prompts for topic name and description generation. Right now, we allow models hosted on Ollama, which allows us also to use local infrastructure, and the OpenAI API.

The system follows a modular design with clearly defined component interfaces, enabling straightforward integration of new modules. This architecture allows seamless addition of new LLM services, embedding services, or database backends.

Tag proposals

We implemented several methods that can find probable instances of a given tag in a large collection of text fragments.

LLM Tag proposal

This module localizes tags within the provided text by leveraging an external LLM service, such as the OpenAI API, Ollama, or similar providers. Communication with the specific backend is handled through an abstract `llm_service` class. A fixed system instruction prompt is used to constrain and stabilize the LLM's behavior.

The LLM is responsible for extracting text passages from a continuous input text that correspond to predefined tags. Requesting character-level start and end indices directly from the LLM has proven unreliable. Instead, the LLM returns each extracted passage along with a short contextual prefix—typically 5–10 words immediately preceding the passage—to facilitate accurate localization.

The extracted passages are mapped back to the original text using a multi-stage matching strategy. First, an exact string match is attempted. If that fails, a fuzzy[6] match is performed using a maximum Levenshtein distance of 10% of the passage length. If both approaches fail, the passage is discarded. Such failures can occur when the LLM introduces additional formatting, punctuation changes, or unintended corrections to the text.

---

[5] Straková, J., Straka, M., & Hajič, J. (2014). Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (pp. 13–18). Association for Computational Linguistics.

[6] https://pypi.org/project/fuzzysearch/

The functionality is compatible with the API of the package. It implements the following methods:

- `async def propose_tags(self, text_chunk: TextChunk, tags: list[Tag]) -> TextChunkWithTagSpanProposals:`
- `async def propose_tags_in_db(self, tag: Tag, db_request: DBRequest) -> list[TextChunkWithTagSpanProposals]:`

To instantiate this module, the user should pass “LLMTopicer” as the cls field in the config. The specific configuration is then:

- span_granularity: str - an indication of how long the tag passages should be, this is used in the instruction prompt of the LLM (default=phrase)

Gliner tag proposal

The Gliner module uses a pretrained multilingual GLiNER[7] model, which is based on the DeBERTa[8] model. It generates candidate spans from the input text with lengths ranging from one token up to a maximum of twelve tokens. For each span–tag pair, the model computes a probability score and retains only those pairs whose score exceeds a predefined threshold.

The module supports two operating modes: single-label and multi-label. In single-label mode, each span can be assigned at most one tag. The desired mode can be specified in the configuration. Since the model was originally trained for named entity recognition, the predicted spans typically correspond to short textual indicators of the presence of a given tag.

The functionality is compatible with the API of the package. It implements the following methods:

- `async def propose_tags(self, text_chunk: TextChunk, tags: list[Tag]) -> TextChunkWithTagSpanProposals:`
- `async def propose_tags_in_db(self, tag: Tag, db_request: DBRequest) -> list[TextChunkWithTagSpanProposals]:`

To instantiate this module, the user should pass “GlinerTopicer” as the cls field in the config. The specific configuration is then:

- model: str - a pretrained model from GLiNER Huggingface or local path to HF compatible GLiNER model (default=urchade/gliner_multi-v2.1)
- threshold: float - threshold for filtering out predictions. Higher the threshold, lower number of predicted spans (default=0.5)
- multi_label: bool - allow assigning multiple tags to a single span (default=False)

---

[7] Zaratiana, U., Tomeh, N., Holat, P., & Charnois, T. (2024, June). Gliner: Generalist model for named entity recognition using bidirectional transformer. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)* (pp. 5364-5376).

[8] He, P., Liu, X., Gao, J., & Chen, W. DEBERTA: DECODING-ENHANCED BERT WITH DISENTANGLED ATTENTION. In *International Conference on Learning Representations*.

Cross-Encoder BERT tag proposal

The cross_bert module implements a custom fine-tuned variant of a BERT model, which serves as a cross-encoder between a given tag and text. The training data for training these models were created by clustering a large collection of texts and their gemma-2 embeddings. Then, the gpt-5-mini model was used to generate tags present in each cluster. Lastly, a few texts closest to the centroid of each cluster were sampled and gpt-5-mini was again used to locate the tags within the texts.

The input is a single tag name together with its description and the text. The model then calculates a score for each token, indicating the presence of the given tag at this location. Then, spans are extracted from the predicted scores, with a configurable parameter gap_tolerance, which fills a N token gap between two prediction spans, allowing for lower fragmentation.

The functionality is compatible with the API of the package. It implements the following methods:

- `async def propose_tags(self, text_chunk: TextChunk, tags: list[Tag]) -> TextChunkWithTagSpanProposals:`
- `async def propose_tags_in_db(self, tag: Tag, db_request: DBRequest) -> list[TextChunkWithTagSpanProposals]:`

To instantiate this module, the user should pass “CrossBertTopicer” as the cls field in the config. Available models can be found in the package README. Each model contains a model_config.yaml file with the recommended configuration for the model. The specific configuration is then:

- model: str - path to either a HuggingFace BERT-like model or a path to local directory containing model files downloaded from the package
- threshold: float - threshold for determining whether a token belongs to the provided tag, user should set the value based on the model recommended value (default=0.5)
- device: str - either “cuda” or “cpu” (default=”cuda”)
- max_length: int : maximum token length for the HuggingFace tokenizer, user should set the value based on the model recommended value (default=512)
- gap_tolerance: int - if two predicted spans are separated by less than “gap_tolerance” tokens, they get merged into one, which leads to lower fragmentation (default=0)
- normalize_score: bool: whether to normalize scores before calculating token assignment probabilities, user should set the value based on the model recommended value (default=True)
- soft_max_score: bool: whether to use soft max score instead of hard max, user should set the value based on the model recommended value (default=True)

Database abstractions

The database abstractions provide two basic functionalities:

1. `get_text_chunks` - Load texts from a database based on a simple metadata filtering. This is often used by discover_topics_dense and `discover_topics_in_db_sparse(...)`.
2. `find_similar_text_chunks` - Retrieve relevant texts from a database with metadata filtering. This is used often in `propose_tags_in_db(...)` to find texts which likely should be tagged without explicitly processing all texts which satisfy the metadata filtering criteria. This generally requires the underlying database to prove vector retrieval capabilities.

Additionally if embedding vectors are stored in a database, they can be retrieved using `get_embeddings(...)`.

LLM service abstractions

The LLM service abstractions provide two basic functionalities:

1. `process_text_chunks` - Process a set of texts given an instruction and model name. This function returns raw LLM responses.
2. `process_text_chunks_structured` - Process a set of texts given an instruction, model name and a Pydantic Basemodel. This function parses the raw LLM response into the provided Basemodel.

Embedding abstractions

The embedding abstractions provide two basic functionalities:

1. `embed` - Process a set of texts given a prompt (optional) and a normalizaton setting. This function returns a Numpy array containing text embeddings. If the normalization parameter is set, the embeddings are L2 normalized.
2. `embed_queries` - Embed a set of queries with a fixed prompt and a normalization setting. If the normalization parameter is set, the embeddings are L2 normalized. If the specific embedding service does not allow this, the two methods work the same.

REST API

The REST API provides simple encapsulation of the python package. It provides seven endpoints: one for listing all available configurations (`/v1/configs`) and six for the methods of the topicers:

- `/v1/topics/discover/texts/sparse`: Discover topics in provided texts using a sparse approach (`discover_topics_sparse`).
- `/v1/topics/discover/texts/dense`: Discover topics in provided texts using a dense approach (`discover_topics_dense`).
- `/v1/topics/discover/db/sparse`: Discover topics in texts stored in a database using a sparse approach (`discover_topics_in_db_sparse`).
- `/v1/topics/discover/db/dense`: Discover topics in texts stored in a database using a dense approach (`discover_topics_in_db_dense`).
- `/v1/tags/propose/texts`: Propose tags on provided text chunk (`propose_tags`).

- `/v1/tags/propose/db`: Propose tags on texts stored in a database (`propose_tags_in_db`).

If a user specifies a topicer that does not exist in the REST API, a response with status code 404 (Not Found) is returned. In case a user specifies an invalid topicer-method combination, the response has status code `409` (`Conflict`). Otherwise, the REST API returns `200` (`OK`) when the request succeeds or `422` (`Unprocessable Content`) if the passed parameters don't have correct structure. The REST API is implemented using `fastapi` and `uvicorn` packages and the source codes are in the `topicer_api` directory inside the git repository. The implementation also contains a simple command-line client which connects to the REST API and runs the specified topicer's method.

Run API server directly

1. Create python virtual environment with python >3.10
2. Install dependencies by `pip install -r ./topicer_api/requirements.txt`
3. Set environment variables (optional):
    - `APP_HOST` - Network interface to bind the server to. Use `127.0.0.1` for local-only access or `0.0.0.0` for public access (default `127.0.0.1`)
    - `APP_PORT` - Port on which the REST API will run (default `8000`)
    - `TOPICER_API_CONFIGS_DIR` - Path to a directory with configurations of topicers (default `./configs`)
    - `TOPICER_API_CONFIGS_EXTENSION` - Specification of the configuration files extension (default `.yaml`)
4. Run the server from `./topicer_api`: `python run.py`
5. Verify functionality by checking auto-generated REST API documentation at: `http://localhost:8000/docs`
6. Verify functionality by running `client.py`

API server docker deployment

Docker provides the most convenient way of deployment. All relevant files are located in the directory `deploy/`. Run the docker image by:

1. Install Docker Engine and Docker Compose
2. If you intend to use topicer methods which use GPUs, install Nvidia container toolkit.
3. Checkout the git repository and change directory to deploy/
4. Add configuration files of your choice into deploy/data/configs
5. Add any models into deploy/data/configs/models
6. Change paths in your configuration files to /app/data/configs/.. - this will be the deploy/data/configs host directory.
7. Build the image: `./update.sh build`
8. Run the server: `./update.sh up` (or in background: `./update.sh up topicer -d`)

9. The API should be now available at `http://localhost:8080`. Check functionality as in Run API server directly (e.g. visiting `http://localhost:8080`/docs)

`Configurations can be added, removed or changed in deploy/data/configs` on the host. The service has to be restarted for the changes to take effect. E.g. by `./update.sh down && ./update.sh up -d`

NO NVIDIA GPU AVAILABLE. If you don't have a nvidia gpu available, you don't need to install the Nvidia container toolkit. Afterward comment the "`deploy`" section in `compose.yaml`. This, however, may disable some topicer methods or it may make them slow.