Engineering Task

1. Use Case:

Text-to-Image generation with spatial controls

Method: `ControlNet` enhances pre-trained text-to-image diffusion models by adding spatial conditioning, such as edges, depth maps, and segmentation masks, allowing for more control over the generated images. It works by training a separate network for spatial conditioning, which reduces hardware requirements and training time compared to retraining the entire model.

`ControlNet` is useful when training data is limited or expensive, enabling efficient prototyping and testing by generating high-quality images from minimal input data. It's applicable in fields like manufacturing, medical imaging, and design.

Potential application:

- Imaging, personalized medicine: simulate imaging methods (e.g., X-ray, MRI) to preview diagnostic images, aiding decision-making and treatment planning. This can be personalized with patient data for tailored simulations.
- Image restoration: enhance or restore low-quality images (e.g., low light, noise), useful for medical imaging and quality control.
- manufacturing: generate detailed 3D visualizations of hardware to improve design precision and detect manufacturing defects.
- Synthetic data ML models: create synthetic datasets for training ML models, improving anomaly detection and production line optimization.

Potential Risks:

- Bias & Errors: simulated data might not reflect real-world complexities, leading to potential bias or errors in decision-making.
- disrupting workflow: introducing ML method to existing production lisne could complicate processes and slow production during initial integration.

Conclusion `ControlNet` offers efficient image generation with fine control, benefiting fields like medical imaging and manufacturing. However, it presents risks such as data bias, workflow disruption, and the potential to overshadow existing solutions in favor of new ML models.

Part 2: Code Assessment and Planning:

Status Quo:

In general, the theoretical foundation of the `ControlNet` algorithm is extensively documented in a research paper and the analysis and architecture of the model

are clear from the repository.

The code base `ControlNet/` and analysis script `awesomedemo.py` are in state of Proof of Concept with a clear application to research:

`ControlNet/`:

- not written as package to be installed but a simple research code base
- requires `CUDA` as hardware requirements so it is not easily executable on any machine
- barely to no code is properly documented
- relies on python 3.8 (no security updates, no bug-fixes)
- `environment.yaml` file available with packages pinned to versions, allowing to run analysis in controlled and reproducible way (of there wasn't the lack of versioning to the repository itself).
- no version control on repository
- latest commit in 2023 (more than two years, no active development to be expected)
- many but mostly unresolved issues raised in the GitHub repository of `ControlNet`
- mix of source code (basic functionality and classes, `ldm/` and `cldm/`, where `cldm/` builds on `ldm`) and its application (highly repetitive code in root of repository)

License `ControlNet/` code base is published with an Apache 2.0 license (allowing it to be used, distributed, and modified in both open-source and commercial projects, as long as proper attribution is provided, contributors cannot make patent claims, and the software is provided without warranties or guarantees).

`awesomedemo.py`:

- simple python script in notebook style that needs to be copy-pasted into `ControlNet/` and relies on two modules in `cldm`, executing investigative analysis.

Implementation/Planning

Python version/ environment: The current code base `ControlNet/` is written with Python 3.8, a deprecated version of python. Dependent on the use case and the required longevity of this project, it might be worth exploring an update to a stable python version, such as Python 3.12 to ensure security/long-term support/compatibility with other tools/better asynchronous programming/…

Additionally, while the package dependencies are currently locked with their respective versions in the `environment.yaml` file, it might make sense to move from `conda` to tools such as poetry that keep track of all dependencies within a `.lock` file and allows for an easy package development (`pip` only, installations from `conda` channels not supported).

Code architecture: I would fork the repo and refactor the project as code is highly repetitive and a large part does not seem to be used. Main idea of the refactoring would be to provide ControlNet as a python package, with a clear focus on avoiding code duplication and removing unused code snippets and clearly separating source code from analysis code.

```
ci_cd/      # pipelines controlling CI/CD, PR validation,
                automated tests, generating hosted documentation
docs/       # generate docs from docstrings, document code base
backend/api # the API and its orchestration
configs/    # hard-coded config values
src/
    controlnet/ # for that defines controlnet model and functionality
    ml_steps/   # orchestration, running training/inference/...
tests/      # test your code!
Dockerfile  # containerize application
.pre-commit-config.yaml  # formatting, code quality, error detection, ...
```

For the simplicity of the use case I'd start developing in a mono-repository, but clearly separate code for different purposes from the beginning, i.e. `src/controlnet/` for packaging the source code of `ControlNet`, `src/analysis/` for code that orchestrates the analysis with code from `src/controlnet`. As a demo back-end shall be provided, the api code shall be separated into `backend/app/` folder. With this structure there is a clear separation of concern of code: model, model training and the API maintenance: `controlnet` can be imported into `analysis` as package and `analysis` can be a dependence for the API. If later the decision is made that the code shall be separated into multiple repositories, this structure allows this to be done without a major refactoring of the code. Motivation for such a decision could be a decoupled development of ML related code to the API code itself.

Additionally, rewrite the code to remove hard-coded parameters. Define schemas for the Python interface using `pydantic` or a similar data validation solutions. Separate hard-coded values into dedicated configuration files (i.e. separate `configs/` folder, use configuration management tool such as `hydra` for managing multiple experiments and hierarchical configurations.

The code base should be clearly documented: provide clear docstrings and documentation on the setup and functionally of the project (setup instructions, model details, dependencies) to ensure reproducibility of the demo and understand the underlying processes. Use tools such as `sphinx` and enforce standards to commits (i.e. docstrings for every function) to document code and automatically host documentation for other developers and users.

Apply rigurous git workflow from the beginning

- use tools such as `pre-commit` for a time-saving approach to ensure code quality, consistent formatting, catch common errors, enforce best practices,

etc.

- set up a branch protection for `main` branch and, if needed a `develop` branch
- develop on separate feature branches with a clear naming convention for branches, i.e. `feature/*`, `fix/*`, `docs/*`, and merge feature branches with clear and concise commit messages (use squash commit to reduce commits and keep clear overview over features released to `main` branch)
- enforce peer-reviewing to catch errors and ensure code quality of repository
- have a pre-defined set of tests that are automatically run before a PR can be merged onto the `main` branch (continuous integration)
- for continuous deployment, use a versioning approach, i.e. semantic versioning on the repository (i.e. `commitizen`) to ensure reproducibility
- dependent on company and demo requirements, consider using `blackduck` and `sonarcube` or other software component analysis tools

Containerize solution Containerize solution for consistency, portability, scalability, and security, i.e. into `docker` or `podman`.

Infrastructure The current implementation of the model relies on `CUDA` for GPU acceleration, but with recent updates, PyTorch has expanded support to also include CPU-based execution. The API to provide inference could be adjusted to run on CPU, providing greater flexibility for local use, particularly on consumer-grade hardware. `ControlNet`, which is basically fine-tuning a large diffusion model, was originally trained on a single consumer graphics card within five days, the infrastructure for use-case specific training allows for a development of many specialized models at extremely lower costs than when training large diffusion models.

Depending on the Business Unit's specific requirements and the technical expertise of its users, hosting the application with a user interface via a web app might be beneficial. The decision will depend on factors such as the target user base and security needs. For example, a simple web app hosted on Azure with multi-user access management could meet basic use case requirements. However, if security or data isolation is a priority, a more sophisticated setup with isolated instances for each user may be necessary.

To ensure consistency, faster deployments, and scalability, the infrastructure should be managed using Infrastructure as Code (IaC) tools like Terraform. This approach will also improve reproducibility and facilitate rapid recovery in the event of an issue.

Life-Cycle For an improvement of the model, if requested, one could consider fine-tuning the model for a specific use case, such as medical imagery. Apply fine-tuning on a periodic basis (if needed, after all this is meant to be a simple demo) to fore come concept-drift. But as the promised delivery to the Business Unit is just a simple use-case demo, not a maintained service with requirements

on quality and reliability, this should be kept as simple as possible. The provided pre-trained models on `huggingface` are not versioned. To ensure some accountability of the current demo to work, one should provide a ZEISS-internal instance of the model with a model versioning tool (i.e. `mlflow` or `dvc`), so each version of the model is clearly tracked with relevant metadata (e.g., training datasets, hype rparameters, performance metrics).

While this demo will not have the requirements of a full service, implementing basic error logging and model performance tracking would ensure that we can detect any significant issues with the demo.

Use data validation concept for input data If required, one can consider to apply a data validation step for input data, to control data formats, input size, and others to ensure data correctness and prevent system failure.

Part 3. Implementation

The usage of the application is documented in the repository, from building the docker image and spinning a container.

For the implementation, `fastAPI` was used as it provides a simple setup with minimal code that yet is scalable to larger applications. Additionally it provides an automated documentation, data validation and asynchronous support.

PLATFORM requirements

To run this application, `CUDA` is required on your platform. First of all, as the current code implementation requires `CUDA` and my MacBook does not offer any NVIDIA-GPU. For the developemnt I used a Linux VM on Azure (Ubuntu 20.04) with Standard NC4as T4 v3 (4 vcpus, 28 GiB memory). It should easily be possible to adapt the code to run on CPU only as only inference is required (there are also discussions in the git repo about it), but for time reasons, I will not focus on this point.

For simplicity and limitations of disc space on the VM, I did not use a `CUDA` base image, i.e. `nvidia/cuda:11.4.2-cudnn8-runtime-ubuntu20.04` and installing miniconda explicitly but used the miniconda base image directly, assuming `CUDA` to be available on the platform where the docker is executed. This is generally bad practice and should not be done in any customer facing project. A multi-stage build here is not necessarily required and I refrained from for time reasons (usually done to reduce image size, have a separation of concerns, and improve security and efficiency).

Container setup

To *version* the implementation of `ControlNet/` and the pre-trained model with the most minimal effort, I download the files from git (`.zip`, not `.git` as git history is irrelevant), versioned by the latest commit hash. For simplicity, the

original model used in `awesomedemo.py` (`control_sd15_canny.pth`) is directly downloaded into the docker from `huggingface` (takes some time for an initial `docker build`). To control the version of this model that is used for inference, I specify the version by replacing the reference to the `main` branch in the link with the associated git hash. `ControlNet/` is referenced in the code as a python module. The project's python environment is built based on the `ControlNet/evironment.yaml`. No additional packages needed to be installed. Due to time restrictions `ControlNet` is not packaged and imported, as `environment.yaml` from `ControlNet/` was sufficient, no other explicit environment file for the API was created.

Code Structure

For the development of the API, I did not use any repository templates to provide a clear and minimal code example without polluting the repository with unused code snippets.

I focused on separating the code clearly based on purpose such that different folders can be packaged later and only transitive dependencies are allowed.

`awesomedemo.py` was refactored into smaller functions in a separated `src/` folder to clearly separate the API from the ML orchestration code:

```
ControlNet/         # REQUIRED (not in repo) - docker copies folder here,
                      #   current version of code assumes this repo here
README.md           # general description of this repo and pre-requisites
controlnet_api
   ci_cd/           # CI/CD pipelines, dependent on remote also
                      #   called `.github` or `.azure_devops` (not done)
    Dockerfile
    README.md       # how to generate the docker and run it
    backend
        api         # API with all orchestration code
            README.md       # description how to run API locally
            app.py          # main app that orchestration API
            model.py        # router for model
            utils   # utilities used in app.py and model.py
                controlnet_orchestration.py # awesomedemo.py - refactored
                logging_utils.py
                model.py
        schemas     # control the input with pydantic contracts
            base.py
     .pre-commit-config.yaml
     .gitignore      # ignore files for git
     .dockerignore   # ignore files for docker
    tests
```

Functionality

Access API

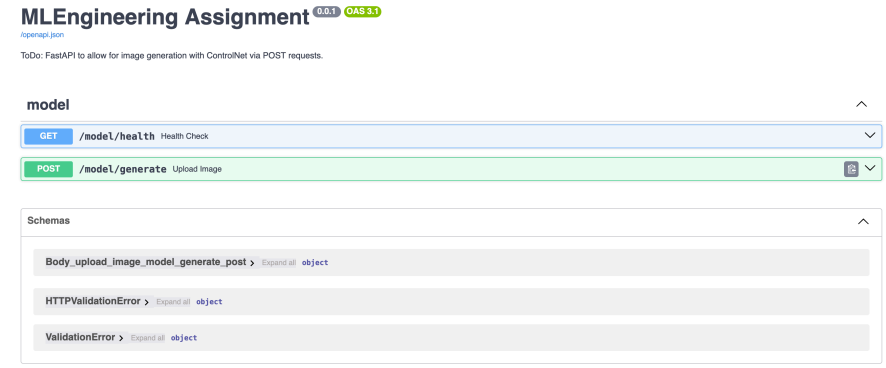
To access the API (if hosted in VM), make sure to use the public API of your virtual machine, not just 0.0.0.0, i.e. `hostname -I`. Then navigate in the browser to `http://<MY-PUBLIC-IP>:8080` (ensure firewall traffic to allow port 8080 (`sudo ufw allow 8080`, `sudo ufw enable`). Additionally, make sure that inbound traffic is open on `8080`. It might be, due to company policies, that such an access is not allowed.

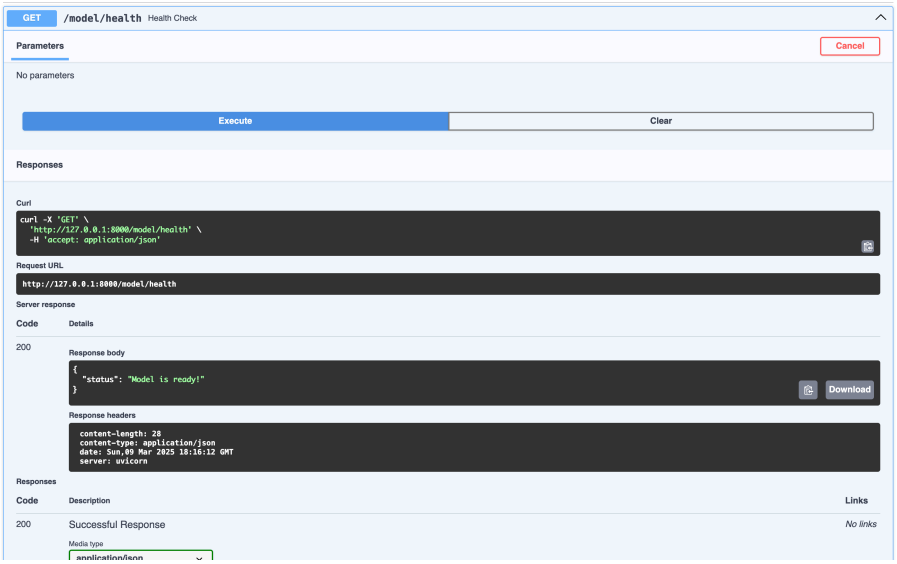
Once the images are generated they can be downloaded with a zip file.

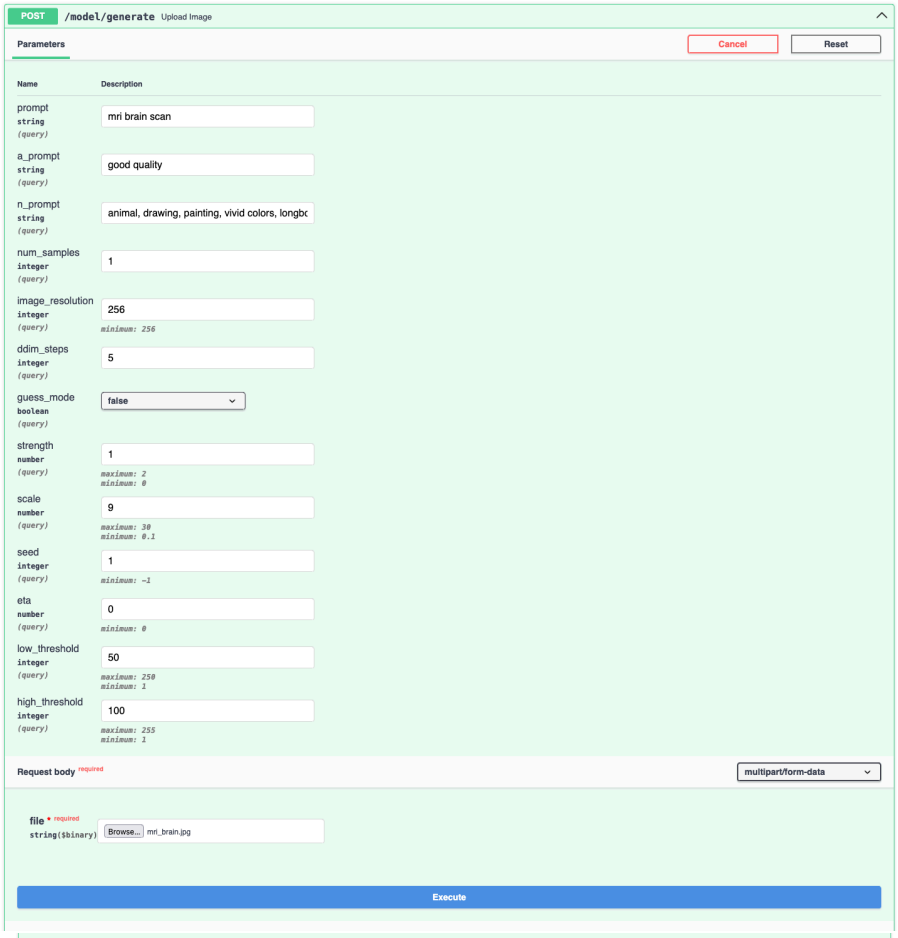
Open/ToDos: Code base:

- `tests/` for time reasons I did not write tests
- serialization: this topic is barely touched, but fixed versions of `ControlNet/` and model `control_sd15_canny.pth` are downloaded
- analysis: I did not touch the analysis content-wise, only refactored and modularized content from `aweseomedemo.py` or what it specifically returns but kept it as is for time reasons. This has per consequence that the `num_samples` parameter counts in a pythonic way. So setting this parameter to 1 in the API will generate two images, which is fine as the first returned image is the Canny image from the image that was uploaded.
- documentation: no `docs/` folder was created to render documentation in html-format
- configuration: no separated `configs/` folder with hard-coded configuration values, pre-sets for API are written into `pydantic` contract, other hard-coded values are to be found on top of files.
- no multi-container spin-up with `docker-compose.yaml` for development with separated storage space
- lacking functionality:
  - upload one image only
  - model configurations are not stored in the *.zip file
  - for starting this application, the model needs to be loaded. Could have been solved by adding `app.on_event("startup")` call with the python module `threading`.
- data:
  - no persistent storage or accessible database - generated images are currently written into docker file system (limited storage space, limited access to files, no data persistence (no backup), container I/o overhead, security vulnerability, …)

Final Proof


MLEngineering Assignment
0.0.1
OAS 3.1
/openapi.json
ToDo: FastAPI to allow for image generation with ControlNet via POST requests.
model
GET /model/health Health Check
POST /model/generate Upload Image
Schemas
Body_upload_image_model_generate_post
HTTPValidationError
ValidationError
Expand all
object


GET /model/health Health Check
Parameters
Cancel
No parameters
Execute
Clear
Responses
Curl
curl -X 'GET' \
'http://127.0.0.1:8000/model/health' \
-H 'accept: application/json'
Request URL
http://127.0.0.1:8000/model/health
Server response
Code
Details
200
Response body
{
"status": "Model is ready!"
}
Download
Response headers
content-length: 28
content-type: application/json
date: Sun,09 Mar 2025 18:16:12 GMT
server: uvicorn
Responses
Code
Description
Links
200
Successful Response
No links
Media type
application/json


POST
/model/generate
Upload Image
Parameters
Cancel
Reset
Name
Description
prompt
string
(query)
mri brain scan
a_prompt
string
(query)
good quality
n_prompt
string
(query)
animal, drawing, painting, vivid colors, longbc
num_samples
integer
(query)
1
image_resolution
integer
(query)
256
minimum: 256
ddim_steps
integer
(query)
5
guess_mode
boolean
(query)
false
strength
number
(query)
1
maximum: 2
minimum: 0
scale
number
(query)
9
maximum: 30
minimum: 0.1
seed
integer
(query)
1
minimum: -1
eta
number
(query)
0
minimum: 0
low_threshold
integer
(query)
50
maximum: 250
minimum: 1
high_threshold
integer
(query)
100
maximum: 255
minimum: 1
Request body
required
multipart/form-data
file
required
string($binary)
Browse...
mri_brain.jpg
Execute
Request body
required
multipart/form-data
file
required
string($binary)
Browse...
mri_brain.jpg
Execute
Clear
Responses
Curl
curl -X 'POST' \
'http://127.0.0.1:8000/model/generate?prompt=mri%20brain%20scan&a_prompt=good%20quality&n_prompt=animal%2C%20drawing%2C%20painting%2C%20vivid%20colors%2C%20longbody%2Clowres%2C%20bad%20anat
-H 'accept: */*' \
-H 'Content-Type: multipart/form-data' \
-F 'file=@mri_brain.jpg;type=image/jpeg'
Request URL
http://127.0.0.1:8000/model/generate?
prompt=mri%20brain%20scan&a_prompt=good%20quality&n_prompt=animal%2C%20drawing%2C%20painting%2C%20vivid%20colors%2C%20longbody%2Clowres%2C%20bad%20anatomy%2C%20bad%20hands%2C%20missing%20f
ingers%2C%20extra%20digit%2C%20fewer%20digits%2C%20cropped%2C%20worst%20quality%2C%20low%20quality&num_samples=1&image_resolution=256&ddim_steps=5&guess_mode=false&strength=1&scale=9&seed=
1&eta=0&low_threshold=50&high_threshold=100
Server response
Code
Details
200
Response body
Download file
Response headers
content-disposition: attachment; filename=generated_images.zip
content-type: application/zip
date: Sun,09 Mar 2025 18:21:27 GMT
server: uvicorn
transfer-encoding: chunked