VELD

Versioned Executable Logic and Data

Technical Concept

Stefan Resch

6.8.2024

Austrian Centre for Digital Humanities and Cultural Heritage

Austrian Academy of Sciences

Introduction

VELD is three things: a *design pattern,* a *reference implementation* and a *platform.* Its goals are standardized creations of workflows that are both flexible and stable. It proposes a modular architecture, based solely on git and docker, by defining interoperable components that can be aggregated into reproducible cohesive workflows or be easily rearranged and reused across contexts and machines.

As a design pattern, its ideas can be interpreted more loosely and implemented with any suitable technology, with no constraints other than the definition of the constituting VELD objects and their relations. The second section of this concept proposes a reference implementation detailing stricter requirements regarding choice of tools, metadata syntax and procedures. Finally, the third section elaborates on a possible web platform to browse, find, publish and execute veld objects.

VELD stands for:

- **Versioned**: all components are version controlled
- **Executable**: code and chains of code are easily executable
- **Logic**: code is aggregated into logical processing chains
- **Data**: non-code data is an equally integrated component besides logic

The goals of VELD are:

- Encapsulation of code and data into atomic reusable components
- Flexible aggregation of atomic repositories into cohesive units
- Increased reproducibility of code and its entire project and data context
- Utilization of a minimal set of proven stable technologies
- Avoidance of creating additional software as hard constraint
- Flexibility through a distributed design

The target stack of the design pattern and of the reference implementation are solely:

- Git: Management of components and aggregations of components
- Docker: Management of execution and building of individual code components and aggregations of components

The target audience and use cases of VELD are:

- **Creators:**

    - **Technically skilled users developing veld code for reuse:** By focusing on a code component alone and wrapping it in docker, and by following the standardizing procedure of VELD, it can be more easily adapted to different contexts with similiar data processing needs and across different machines.
    - **Data Scientists manifesting entire data processing pipelines:** By aggregating sets of atomic data and code components into processing chains, the entire context and history of data, parameters, and code versions is persisted and made easily reproducible. This is especially important for Data Scienctists and scholars who have a very high requirement for reproducibility.
- **Consumers**
    - **Less technically skilled users consuming pre-built code packages:** By only requiring docker as the execution platform, VELD products can be made easily executable across machines, operating systems, and time. Users can run ready-made code velds within their own context and with any suitable data, or they take the docker definitions and adapt it to their needs.
    - **Auditors and peer reviewers reproducing processing pipelines:** Through adherence to this design and by following best practices in docker builds and git management, a highly robust and convenient practice of reproduction of entire processing pipelines can be achieved.

VELD as a design pattern

VELD defines exactly three kinds of objects, called velds, their state and execution management and how they relate to each other. These are:

- **data velds**: containing only non-code data
- **code velds:** containing only code
- **chain velds:** containing aggregations of data and code velds

Any of these components can live entirely on their own and are not restricted on VELD-only integration. However, the more components and their aggregations are designed holistically with VELD in mind, synergies will be more pronounced.

State management

Each of the three VELD objects are manifested as version controlled repositories, primarily with git, where data and code velds constitute atomic units, and chain velds the aggregations of the former two. Through this, separation of concern and reusability of components can be increased since they not strongly tied to any other component and are designed with interoperability as their main purpose. Any data set within such design can be processed or produced by any suitable code component, while chain repositories manifest any such processing workflows reliably without forcing constraints down onto its components. The aggregation procedure itself is implemented primarily with git submodules where the states of child repositories are referenced through their commits and aggregated in a commit of the parent repository. Other version control technologies besides git offer similiar functionalities, such as mercurial subrepositories and SVN externals, but they were not tested due to git being the leading version control in the open source space. However, they are expected to be functionally equivalent to git submodules regarding their intention in the VELD design pattern.

Execution management

The execution runtime is primarily based on docker, and the entire execution context (files and parameters) of a chain veld is defined as a docker compose service. VELD is designed with docker as the primary execution tool, but tests with podman also showed promising results. In general however, it can be assumed from the current development of the Open Container Initiative (OCI), that docker manifestations such as dockerfiles and docker compose files may phase away from strict docker-only products to become more generic definitions for any container technology. Podman for example can process dockerfiles natively, and podman-compose is a promising third-party tool capable of processing docker compose files. With this in mind, basing excutable velds such as code velds and chain velds on docker compose files is likely the safest technology choice for long-term reproducibility.

veld.yaml

In the previous section, it was defined that any containerized processing will be executed through docker compose files. As they are integral for the design pattern, these files will be refered to as “veld.yaml” files from now on. In the case of seeing VELD as a design pattern, these veld.yaml files only refer to the docker compose files of code velds and chain velds, while in the reference implementation and platform, they are the source of truth for each distributed

VELD object, and they will also used on data velds as there is a need to coordinate distributed metadata on all veld objects, not only executable ones.

VELD objects (velds), high-level description

This section describes the three VELD objects. It assumes git and docker as tools for implementation, but as described above, similiar tools should be functionally equivalent.

data veld

a data veld

This is simply a repo containing only data files, without any code or execution logic, designed for integration into a chain veld, either as input or output.

code veld

a code veld

A repo that contains source code and its docker build definition. It can be executed by running its docker compose services expressed in veld.yaml files. If a code veld offers to process certain kinds of input and output data, it should declare so in its docker compose file and also describe which sections of this file are needed to be adapted, such as volume mounts which define input and output folders and environment variables which define key-value parameter definitions, such as file names or other kinds of code-specific configurations. Ideally such a docker compose file is adequately commented to support other users or developers in its proper usage.

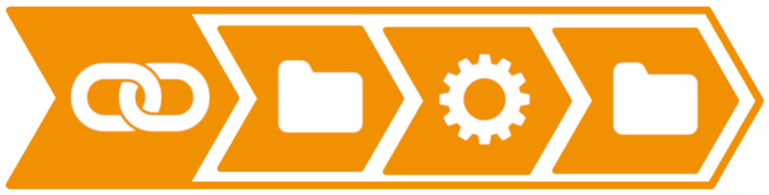
A code veld must be designed to fulfill at least one of two functionalities:

- **Without chain veld integration:** while the code veld itself is a VELD object, it does not require a chain veld context or anything regarding VELD other than itself. This way, the code can be downloaded and used with arbitrary data ad hoc, simply by providing template input and output folders directly into the repo, as well as possibilities to tweak parameters directly in the veld.yaml file.
- **With chain veld integration:** the code veld and its veld.yaml can be designed so that a user with sufficient knowledge in the technical stack and the VELD design can integrate it into custom chain velds. This would be a more ideal use case of a code veld, but it requires more understanding than the prior more direct usage.

chain veld

a chain veld

A chain veld is an aggregation of other veld objects. While it is the more complex veld, it also has the highest flexibility and least constraints. It’s supposed to contains other velds (integrated with git submodules) of linear processing where input data velds are processed by some code veld and their results are persisted into an output data veld.



a chain veld aggregating other velds

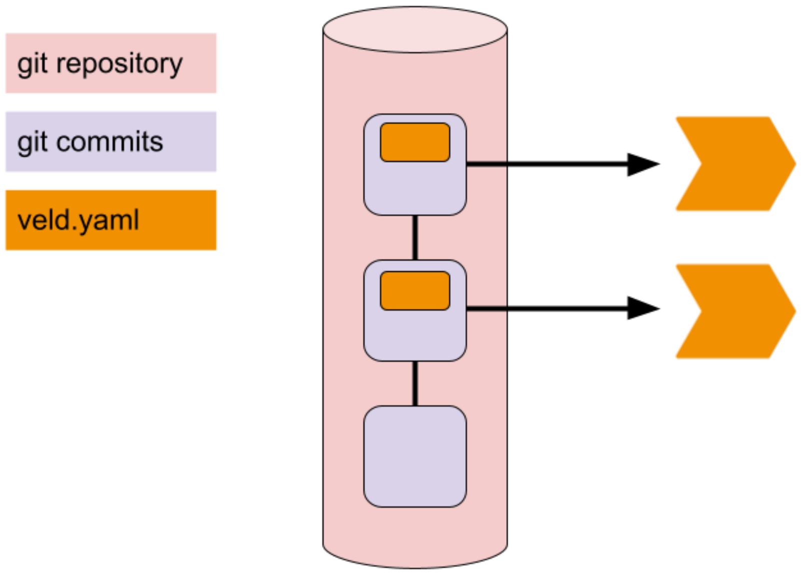
The chain veld itself provides execution points through its own docker compose files, identically to that of code velds. But the chain veld’s docker services within such a veld.yaml file actually inherit from the veld.yaml file of the respective code veld. Through this, the chain veld reuses all the pre-built docker execution of the code veld while still allowing to override or extend any specifics such as data volumes and environment variables (key-value pairs) to its own need. By this design, a clean separation is achieved between reusable logic defined in the code veld and its concrete usages within chain velds. Thus, a chain veld is the full context of an entire processing logic, where data and code is wired tightly together and made easily reproducible. Note, that besides code and data velds, a chain veld could theoretically contain other chain velds as well and reuse their docker compose files. While this is technically possible, it is not advised since it breaks with the mental model of VELD which aims at a clear distinction between a tool (the code veld) and its usage (the chain veld).

It’s supposed that a chain veld would call a code veld for execution on data velds. However, to keep the design flexible and pragmatic, this is not mandatory. As stated above, a chain veld is actually the least constrained of all the veld objects, and thus it allows execution of any docker code on any data, where none of these must be git submodules or velds. Theoretically, the least VELD conforming chain veld would be a repo with input, output data and code integrated directly into the chain repo itself, and none of its data or code components being velds. Of course, it makes little sense to define such a chain veld, but it would still be conforming to the design pattern. The overall rationale for such high flexibility are use cases where hybrid approaches would make more sense or are simply a requirement due to non-adaptability of other components.

Implementation with git

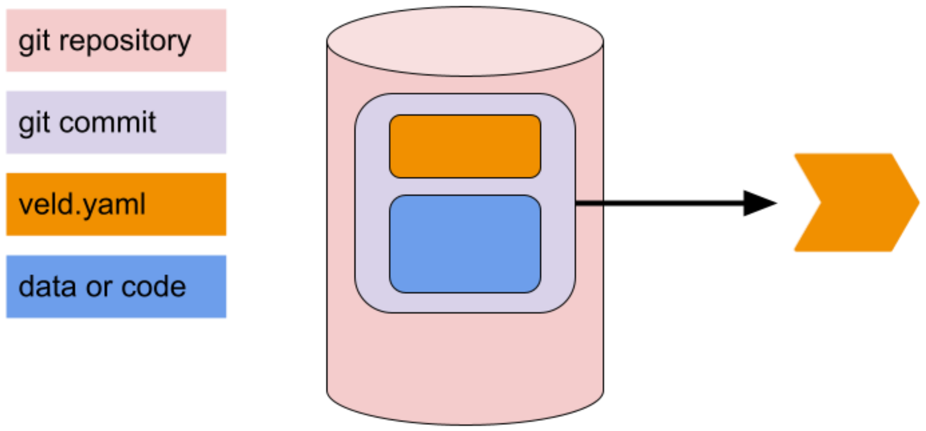
git structure

A veld is represented by a veld.yaml file persisted in a given commit in a git repository. This means that several commits represent several velds.



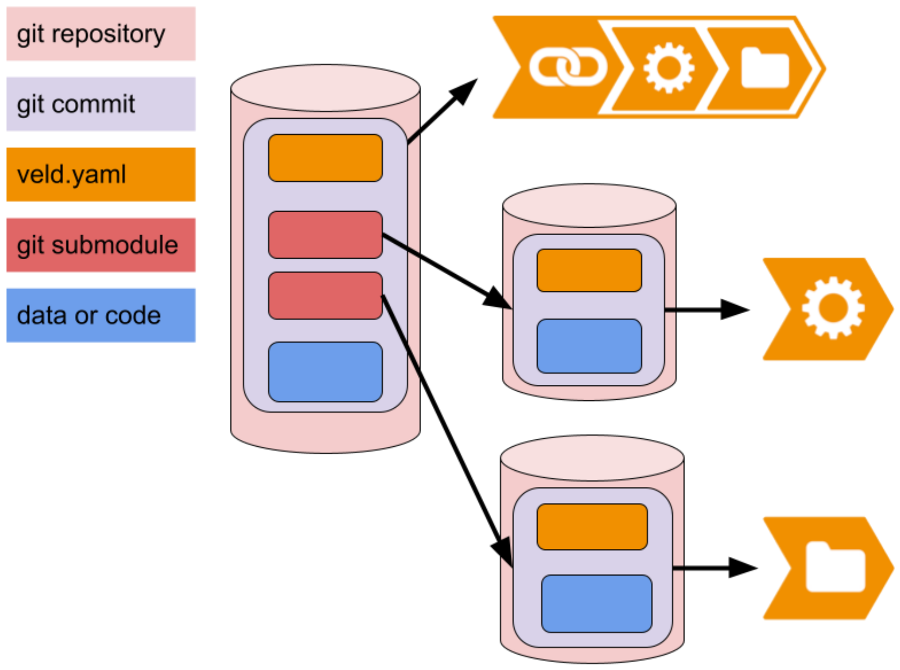
Two of three commits of a repo containing valid veld.yaml files, representing two velds

The commit of a data veld contains just data, while a code veld contains source code and docker build definitions.



The bulk of a commit will most likely be the content, i.e. data or code

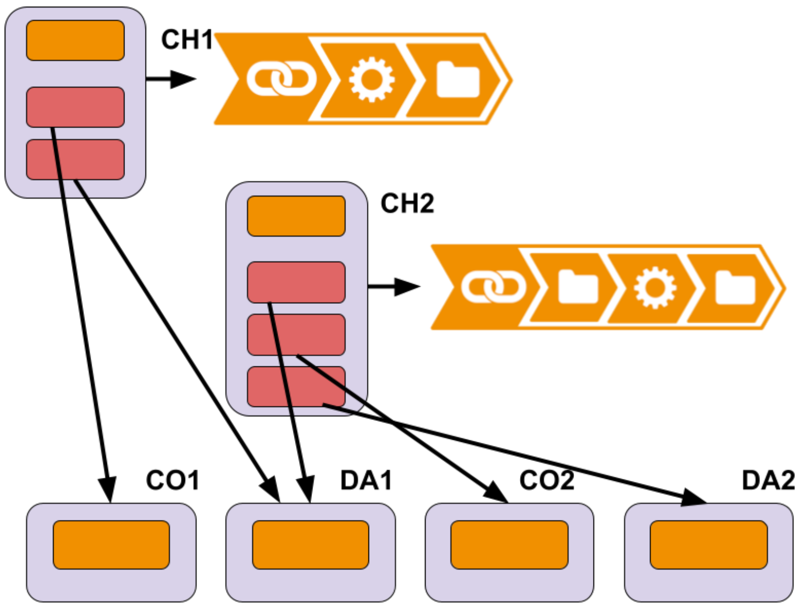
In contrast to data velds and code velds, which exist on their own in an atomic way, a chain veld aggregates one ore more other velds by utilizing git submodules. With this, the commit of a chain veld contains references to commits of other veld repos and a chain veld can achieve exact versioning of the entire tree of its constituting components throughout the whole history of the repositories.



git submodules are used for aggregating velds

Example: reusability

The VELD design enables high flexibility since data velds and code velds are atomic units that can be recomposed across contexts. The more atomic and the less context-sensitive they are, the more reusable they become. For example, here, a chain veld **CH1** uses code veld **CO1** to produce a data veld **DA1**. Another chain veld **CH2** reuses that data veld **DA1** as input for another code veld **CO2** which produces as output data veld **DA2**.



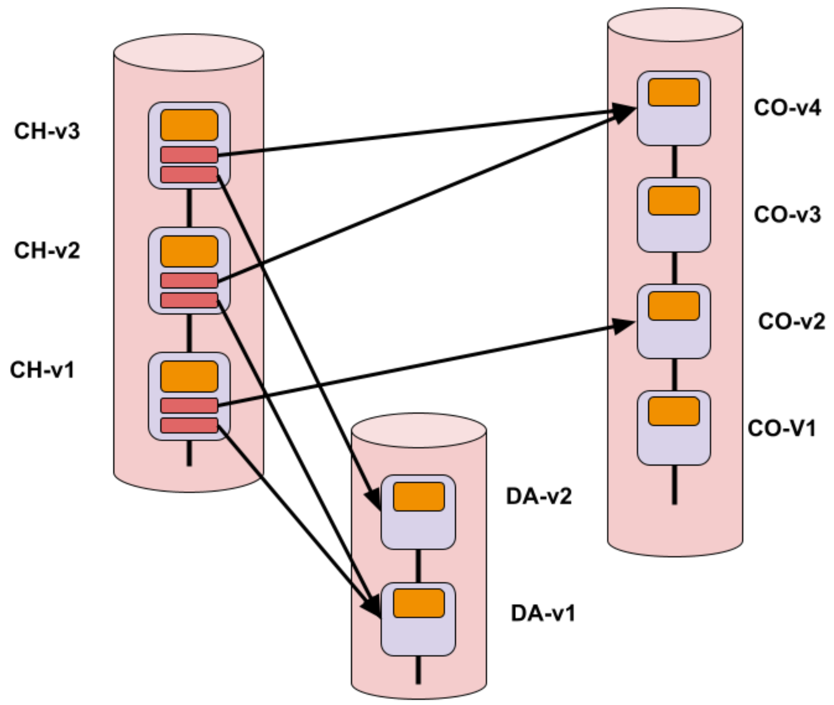
showcasing reusability and history of a data set

Example: history tracing

Due to git's functionalities, the full history of a veld can be traced back, checked out, experimented with, or rerun. As an example assuming three VELD repositories containing data velds, code velds, and chain velds. All of these have their own development histories, where the data veld repo has two commits, the code veld repo has four, and the chain veld has three.

The history / commits of the chain velds and its components are:

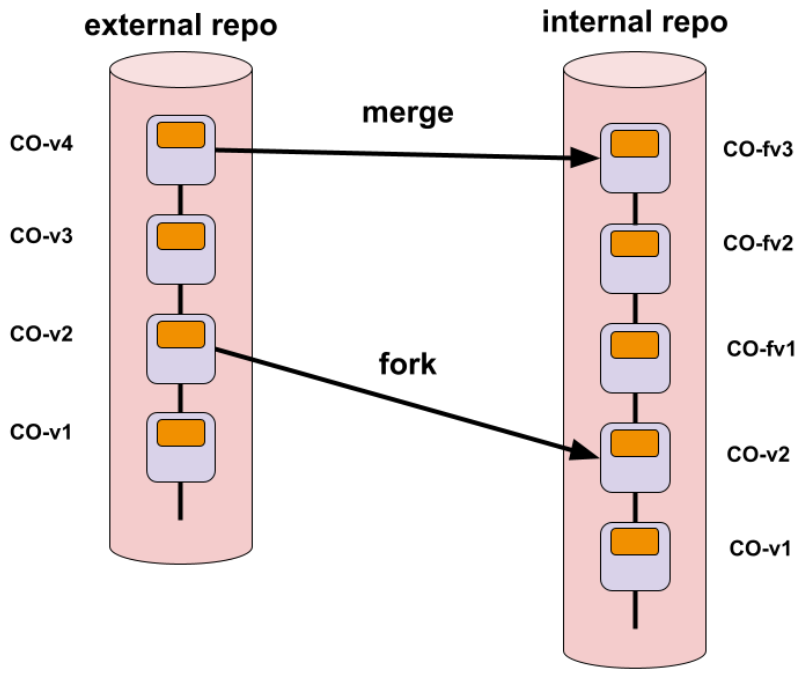
- chain veld **CH-v1** points to the data veld **DA-v1** and code veld **CO-v2** (maybe because at the time of creation of **CH-v1,** the code veld **CO-v2** was the most up-to-date version or **CH-v3** did already exist but there was some reason to not use it)
- chain veld **CH-v2** still points to the data veld **DA-v1** but has moved to code veld **CO-v4** (maybe because **DA-v2** did not yet exist and **CO-v4** provided some improved functionality**)**
- chain veld **CH-v3** updates the pointer to the data veld **DA-v2** and keeps code veld **CO-v4** (maybe because the updated **DA-v2** carries better data, while **CO-v4** is the most current version)



An interlocked history of data, code, and its aggregations

Example: forking and branching

VELD can also benefit from git's branching and forking functionality. For example, there might be an external repo, providing code velds. This repo is of use for some internal process, but the code needs some minor tweaking, so the external repo is forked to an internal one, where the necessary modifications are done. The fork was done when the external repo was at the commit **CO-v2**. After this initial fork, Internal development on the forked repo produces some new commits / code velds: **CO-fv1** and **CO-fv2.** Then after some time, improved functionality was added to the original external repo's code that is of use for the interal one. So the latest external git history is merged into a new internal commit **CO-fv3**. The same functionality applies to git branches too.



History of an upstream repo and its forked downstream repo with modifications

Implementation with docker

Motivation and Usage

Docker compose was chosen to carry out the execution of VELD logic, due to its capabilities in combining the definitions of a docker build and its local context (files, folders, parameters). As outlined above, the veld.yaml files of code velds and chain velds are fully conforming docker compose files. To run any such, only the command “docker compose -f veld.yaml up” is required.

Note that code that is run by docker can be run either by copying the source code into a docker image directly, or alternatively the docker image can serve only to build the runtime context and the dependencies, while the source code is merely mounted as volume. The latter option, mounting code over integrating into a docker image, is recommended for development since it doesn’t require building images each time something in the code has changed. Once some codebase has reached maturity and is meant to be shared on docker, then building a full image might be better. It’s ultimately up to the developers, but in the context of VELD and its prototyping / data science target audience, mounting code via volumes is recommended over copying it into the docker image.

Docker inheritance from code veld to a chain veld

Both code velds and chain velds are docker compose services, and both can be executed. The important difference is that the code veld may be executed alone with custom data ad hoc, but it should also offer itself as an integrateable component to a chain veld for more economic code reuse across contexts. Such reuse of a code veld by a chain veld is done with docker compose's extend command, by which a docker compose file can refer to another locally available compose file and its contained service like so:

```
extends:
  file: ./veld_code/veld.yaml
  service: veld
```

And a matching data veld is integrated by using docker volumes to mount it into the code veld's container, like so:

```
volumes:
  - ./veld_data:/veld/input/data.json
```

This means that such a docker compose service of a chain veld is effectively identical to the service of the code veld except for:

- the addition of necessary context as outlined by the code veld
- the possibility of overriding attributes of the code veld by the chain veld

VELD as a reference implementation

The previous section on the design pattern has explained the main VELD objects and their interrelations on a higher level, as to decouple the architectural ideas from the concrete technical implementations. This section now proposes a more detailed reference implementation, based on specific tooling and usage; and it also introduces a metadata schema that facilitates the description and interoperability of velds.

veld.yaml

At the core of the reference implementation lies the veld.yaml file, its schema, and how it's used to manifest velds and their interrelations. While the previous section on the design pattern introduced the veld.yaml as the docker compose file only for code and chain velds, the reference implementation will extend this file with two important aspects:

- The veld.yaml file will be enriched with non-docker specific metadata only relating to the VELD design. This metadata will be identified by a root yaml element "x-veld" and its child elements representing the veld description. The "x-" prefix is necessary as it's conforming with docker compose specification in such that it will be ignored by docker compose.
- The veld.yaml file now also describes data velds. In contrast to the veld.yaml files of code and chain velds where the files contain both functional and non-functional description with regards to docker, the veld.yaml files for a data veld will only contain docker-non-functional description, i.e. the "x-veld" element.

By extending the central veld.yaml file with those two characteristics, a consistent, pragmatic and distributed practice can be achieved to describe veld object metadata whlie retaining their functional definition for docker execution.

Constraints on veld.yaml files

For all velds:

- **Hard constraints:**
  - there **must** be either **one** or **several** veld.yaml files at the root of a repository:
    - If, for a repository, one veld.yaml file describes one atomic unit of data or code functionality, its name **must** be **veld.yaml .**
    - If, for a repository, several veld.yaml files describe subsets of data or code functionalities, these file names must start with **veld_ .**
  - Each veld.yaml file **must** contain a metadata section named **x-veld.**
- **Soft constraints:**

  - Each veld.yaml file should be as informative as possible and utilize the possible VELD description to its fullest.

For data velds:

- **Hard constraints:**
  - The veld.yaml files **must** contain **only** a section **x-veld: data** adhering to its respective VELD metadata schema.
  - **Nothing must** be relating to docker in these veld.yaml files.
- **Soft constraints:**
  - The purpose of the veld.yaml files for data velds is to describe its containing data by its formats, contents, et cetera. The data and its description **should** be of highest possible consistency.

For code velds:

- **Hard constraints:**
  - The veld.yaml files **must** contain a section **x-veld: code** adhering to its respective VELD metadata schema.
  - Each veld.yaml **must** also contain **exactly one** section defining a **docker compose service**, adhering to the docker compose specification.
  - Each veld.yaml **must** act as the authoritative entry point and persisted execution definition of its function.
  - Each veld.yaml **must** provide **one** or **both** of these functionalities:
    - The ability to run its code without any other VELD context (i.e. chain or data), potentially just on some ad-hoc data, with prepared folders for data input and output embedded directly in the code veld’s repository.
    - Ability of integrating this code veld into a chain veld.
- **Soft constraints:**
  - Each veld.yaml should describe the code veld’s functionality both in potential relation to other VELD objects as well as regarding its usage alone with ad hoc data. Code reuse and ease of building and execution **should** be of highest priority for a code veld.

For chain velds:

- **Hard constraints:**
    - Each veld.yaml **must** contain a section **x-veld: chain** adhering to its respective VELD metadata schema.
    - Each veld.yaml **must** contain **only one docker compose service**, that **must** by defined by **one of two** possibilities:
        - The chain's service is inherited from a code veld. If this is the case, conformitty to the code veld's definitions is required. This approach **should** be prefered over the other.
        - define its own docker service, with custom docker build contexts or references to pre-built images. Through this, a chain veld's execution is not defined in an external code veld, but within the chain's own repository. This approach **should** be avoided in favor of reuse of code velds, but might still make sense in some scenarios.
- **Soft constraints:**
    - Each veld.yaml of a chain veld is the culminative result of some data and code velds. It represents a persisted workflow with all its necessary technical builds, data scopes, and parameters. Thus a workflow manifested as a chain **should** be tested thouroughly, ideally across machines and OSes, to increase abstraction and stability.

VELD objects (velds), implementation description

data veld

The most simple veld object, as it refers to non-code data only, it should describe the format of its files (e.g. "json") and the overall content they contain (e.g. "Named entity recognition gold data") so that data sources are discoverable and connectable to code velds with matching input / output requirements.

It is advised to create data veld repositories for focused and atomic data sets, if possible, rather than aggregating data sets of different formats and consistencies into bigger collection repositories. By having smaller focused repositories over aggregating ones, any VELD processing chain with the data as input or output is more focused as well. Also irrelevant data for some processing is filtered out more cleanly too.

Example data veld.yaml:

```
# x-veld section mandatory for all veld objects
x-veld:

 # next section describes this as a data veld
  data:

   # general information about this veld object / repo
    about:

     # Freeform textual description
      description: "NER gold data created by in-house experts."

     # Possibility to provide the topics this veld concerns, in a tag-like fashion, multiple entries
     # possible. Will relate to items of a controlled vocabulary.
      topics:
        - "NLP"
        - "Machine learning"
        - "Named entity recognition"

   # Description of the file format used, ideally it’s a common MIME type; but custom ones must
   # be allowed as well (e.g. “spaCy models”).
    file_formats: "json"

   # Description of what is inside the files
    contents:
      - "gold data"
      - "NLP gold data"
      - "NER gold data"
```

code veld

In a repository there can be several code velds, each manifested as respective and uniqule name veld_*.yaml files and each expression a certain functionality of a shared code base. It is advised to keep the functionalities of a code veld's repo rather focused and as atomic as possible. But should there be several functionalities offered by a code base and dependencies that have a bigger overlap, than it makes sense to aggregate those into one repository to avoid redundancy. It is up to the code veld's creators to draw the line between focused repositories with a lower number of functionalities and multi-feature repositories with a wider selection of functionalities.

The code veld's highest responsibility is to offer a pre-built and encupsalted processing tool, making it executable as easily as possible. As described above in the constraints, it must aim for one or both functionalities, where one is the execution of a code veld on its own with data ad hoc for less technical users and for quick experimentation, and the other is the potential to integrate a code veld into a chain veld for manifestation of an entire processing workflow where data provenance is of higher priority.

Example code veld.yaml:

```
# x-veld section mandatory for all veld objects
x-veld:

 # indication of this being a code veld
  code:

   # general description
    about:
      description: "A NER trainig setup, utilizing spaCy 3's config system."
      topics:
        - "NLP"
        - "Machine learning"
        - "Named entity recognition"
   # Description of this code veld's possible input formats and where it expects the folder
   # inside the container
    inputs:
      volume: /veld/input/
      file_formats: "json"
      contents:
        - "gold data"
        - "NLP gold data"
        - "NER gold data"

   # Description of its output
    outputs:
      volume: /veld/output/
      file_formats: "spacy model"
      contents: "NLP model"

   # environment variables, here used for names of files, and one basic NLP hyperparameter
    environment:
      in_train_json_file:
        type: file
        volume: /veld/input/
     epochs:
       type: int
      out_model_folder:
        description: "path to the spacy model"
        type: folder
        volume: /veld/output/

# this is the docker compose section
services:
  veld:
    build: .

   # These are template volumes and folders provided already in the code veld’s repo, so that
   # it can be used on its own. Note that a conforming chain veld would overwrite these.
    volumes:
      - ./src/train/:/veld/code/
      - ./data/docbin/:/veld/input/
      - ./data/model/:/veld/output/


   # environment variables; not initiated or set with some default values. Can be adjusted in a
   # copy of a code veld or overwritten by a chain veld.
    environment:
      in_train_json_file: null
     epochs: 10
     out_model_folder: null
```

chain veld

A chain veld is the manifested combination of data velds with code velds. It is also the one with the least constraints as it should be possible for a chain to process veld objects and non-veld objects, since not all project relevant data sets and tools can be assumed to be VELD conforming. Also contrasting to code and data velds, where it is recommened, if possible, to split individual code and data velds across dedicated atomic repositories, this same recommendation does not necessarily apply to chain velds. A chain veld represents a manifested workflow of a given project, and since most projects require different workflows with different code stacks and data sets, it would be too constraining to advise to spread each workflow across dedicated repositores in line with code and data velds. Also, besides hard constraints on a project, it would also create data redundancy, in the case where some data set A was the result of a workflow 1 and the input of workflow 2. If these two workflows were to be spread across dedicated repositories, the data set A would exist twice on a local machine, once as output and once as input.

Overall, it is the arbitraty project itself which defines what should be aggregated into its repository. Any integrated chain velds and their constituing code and data velds are just additive parts, which can coexist next to project-specifics. By putting the least constraints on chain velds, the integration cost of VELD will be always limited to its objects, and never spill over to other aspects of a project at hand.
However, if a chain veld is integrating code and data velds into cohesive reproducible workflows, it lies in the responsibility of the chain veld to conform to the code and data velds as closely as possible.

Example chain veld.yaml:

```
# x-veld section mandatory for all veld objects
```

```
x-veld:

 # indication of this being a chain veld; which is not to be reused in different context, hence
 # there is actually much less metadata other than a description of its intention.
  chain:
    about:
      description: "A NER trainig setup, utilizing spaCy 3's config system."
      topics:
        - "NLP"
        - "Machine learning"
        - "Named entity recognition"

# The docker compose service definition
services:
  veld:
    extends:
     # here, the veld.yaml of a local code veld is referenced as the definition for this docker
     # compose service.
      file: ./veld_code_train_spacy_ner/veld.yaml
      service: veld

   # in the volumes section, data velds are refenced as input and output
    volumes:
      - ./veld_data_ner_gold/:/veld/input/
      - ./veld_data_ner_spacy_models/:/veld/output/

   # the environment variables are overwritten as necessary
    environment:
      in_train_json_file:: "train.json"
      out_model_folder: "model_1”
```

VELD as a platform: veldhub

VELD wants to increase reusability and reproducibility of arbitrary workflows. Anyone adopting the design pattern or following the reference implementation can already achieve these goals with only git and docker. It is clear however, that since VELD's design is inherentely distributed, this necessitates some centralized coordination to keep track of the scattered VELD objects. An adopting user or organization could be pragmatic and may just use a spreadsheet for this (or similiar information management tools) and track their VELD objects. This approach is extremely cheap and reliable, and serves the purpose sufficiently.

However, in order to facilitate the adoption of VELD, an official centralised platform will be developed. This platform is under development and currently called veldhub.

veldhub's goals are:

- Provide documentation, tutorials, and metadata schemas, as the authoritative source for everything regarding VELD
- Offer the availability to search, browse, use, and publish VELD objects. Similiar to github or dockerhub, it will provide a catalogue of open source VELD objects and their links with each other. Note that veldhub will not actually store the git repositories themselves, but only metadata and links to sources. In order to publish to veldhub, procedures have yet to be designed, but will most likely involve github actions, where users would push to their git repositories and initiate an upload of metadata to veldhub with a github action.
- Help with the creation of chain velds, by offering to aggregate compatible code and data velds into generated git-setup scripts and veld.yaml files for easy local creation. Also it is aimed to provide the online execution of whitelisted chain velds for registered users.

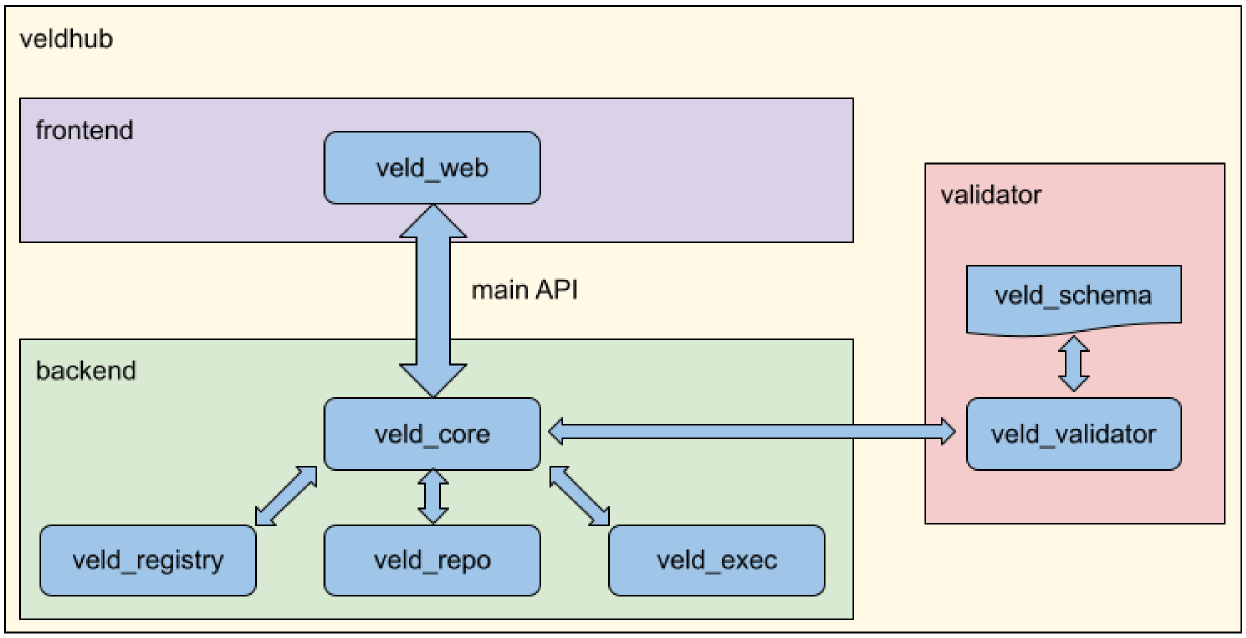
veldhub architecture

The architecture is split into three modules:

- **Frontend**: The public facing website of veldhub. It is designed to be a separate module from the backend to allow flexible decoupling.
- **Backend**: The main processing and persistence component. It offers a central API for all its functionalities, so that any component, be it a web client or some other tool, can utilize it. It is responsible for all CRUD operations, execution of whitelisted chain velds, and data persistence.

- **Validator**: a stand-alone module with its most important part being the VELD metadata schema, expressed as a yaml schema. Besides the schema, there will be some minimal tool integrated into this module that validates veld.yaml files against the VELD metadata schema.

A diagram of the architecture looks as this and its modules are explained as follow:



veld_core

The focal point of the backend. It connects all modules and serves the calls incoming from via main API, to be consumed primariliy by the frontend.

veld_registry

The metadata persistence layer. It registers and processes metadata, controls for their schemas, and provides an overview of available velds as well as their persisted and potential interactions (e.g. what data velds are compatible with what code velds).

veld_repo

The repository manager. Since veldhub offers to execute chain velds, it must be able to clone veld git repositories, commit results and push them to their remotes.

veld_exec

The docker execution manager. It builds and execute docker images of chain velds.

veld_schema

An authoritative yaml schema. It will be the single source of truth for all veld.yaml files and defines a strict syntax on the metadata section.

veld_validator

Small tool checking yaml files for VELD validity. It will use the authoritative veld_schema for this, and also will be a standalone application to enable quick prototyping of the schema and make it independent from veldhub, since it serves very different functionality conceptually.

veld_web

The frontend for discoverability. It presents aggregated and registered velds, and makes them available to clone / download, or be executed on the platform itself if they got whitelisted manually.