CLARIN 2022, Prague

A Lightweight NLP Workflow Engine for CLARIN-BE

Adriaan **Lemmens**

PhD researcher @ KU Leuven, CLARIAH-VL

Vincent **Vandeghinste**

Coordinator CLARIN-BE @ KU Leuven & ivdnt.org

CONTEXT

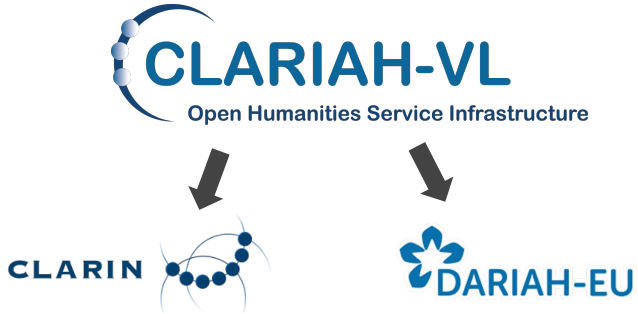
Text Analytics for the Humanities

Points of Friction

CONTEXT

Users: researchers in humanities w/ non-tech profile

- Overwhelming choice of tools
  - Differences subtle
- Need to be combined to be actually useful
  - Manual effort -> replicability issues



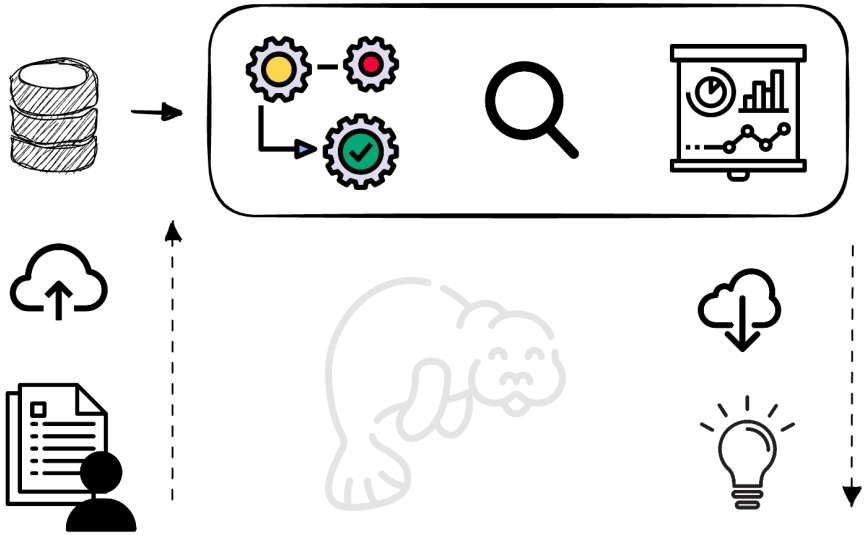
Our Aims

CONTEXT

- Cohesive environment 👩‍💻 for executing text analysis workflows
- UX design: radical simplicity
  - Users confronted with choice only if necessary
- Cover 80% of use cases
- Export to *familiar* formats
  - XML ☹️ ↔ CSV 😊
- Re-use existing CLARIN tools
- But also collab with other unis/groups on new tools & workflows (🇧🇪/🌎)

Seaku – Text Analytics Dashboard

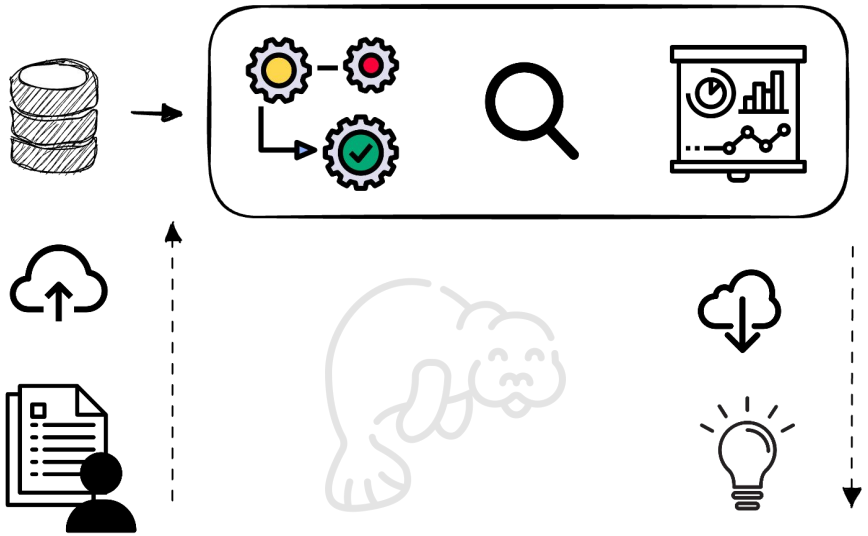
CONTEXT



Seaku – Text Analytics Dashboard

CONTEXT

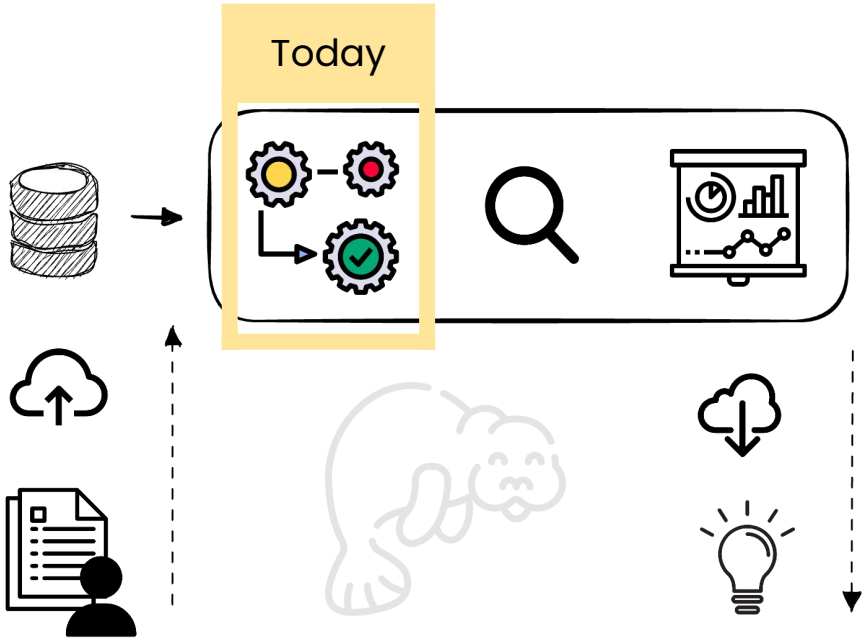
- Web application
  - Easier for everyone
- Core functionality ➡️
  - Manage corpora
  - ⚡ **predefined** workflows
  - Inspect results
  - Export
    - Annotations
    - Analysis summary



Seaku – Text Analytics Dashboard

CONTEXT

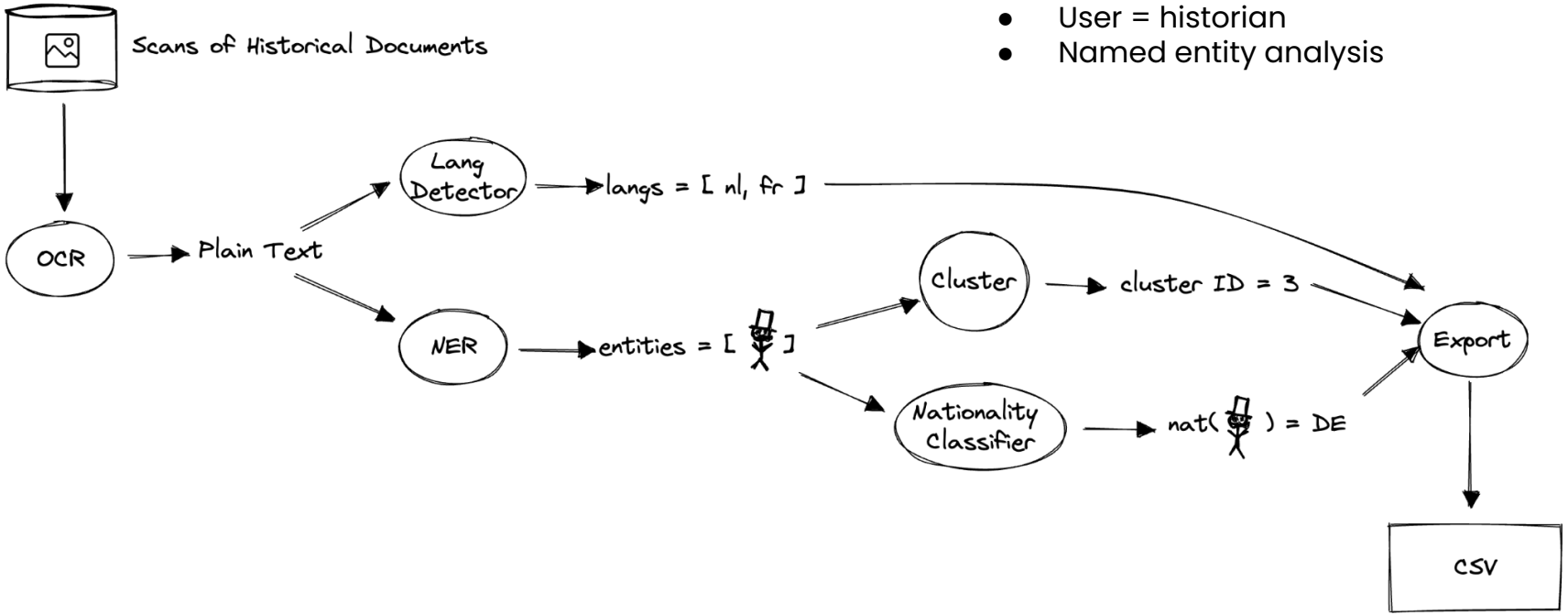
- Web application
  - Easier for everyone
- Core functionality ➡️
  - Manage corpora
  - ⚡ **predefined** workflows
  - Inspect results
  - Export
    - Annotations
    - Analysis summary



Analysis Workflow Example

CONTEXT

- User = historian
- Named entity analysis



What We Need

CONTEXT

- Way to define & execute multi-step data processing logic, i.e. workflows
- Easy for non-core contributors to add workflows & components
- Easy to on-board junior-level devs
- Cheap to run
  - No on-demand cloud compute -> finite server capacity
- ‘Clustering’ algo in prev. ex. -> **batch** processing

ENGINEERING

Building a **Custom** NLP Workflow Engine

Hold on...


Other CLARIN devs
This presentation
DIDN'T WE DO THIS YESTERDAY?
#GroundhogDay

BACKGROUND

WFEs Developed within CLARIN (NLP-focused)

WebLicht

- = App + **WFE**
- Tried and tested
- Rich tool metadata descriptions
- Single docs, not batches* #con
  - *Last time I checked
- Tools always online/running #con
- Java code base #con
- Unclear how to deploy #con

BACKGROUND

WFEs Developed within CLARIN (NLP-focused)

WebLicht

- = App + **WFE**
- Tried and tested
- Rich tool metadata descriptions
- Single docs, not batches* #con
  - *Last time I checked
- Tools always online/running #con
- Java code base #con
- Unclear how to deploy #con

CLARIN-PL WFE

- Powers CLARIN-PL's vast tool inventory
- Batch processing ✅
- Tools run on-demand ✅
- Tools are containerized ✅
- Python API (+ Java, C++)
- Architectural similarities w our system

BACKGROUND

WFEs Developed within CLARIN (NLP-focused)

WebLicht

- = App + **WFE**
- Tried and tested
- Rich tool metadata descriptions
- Single docs, not batches* #con
  - *Last time I checked
- Tools always online/running #con
- Java code base #con
- Unclear how to deploy #con

CLARIN-PL WFE

- Powers CLARIN-PL's vast tool inventory
- Batch processing ✅
- Tools run on-demand ✅
- Tools are containerized ✅
- Python API (+ Java, C++)
- Architectural similarities w our system
- **Overlooked in initial survey** 😐

Design Aims & Assumptions

ENGINEERING

Aims:

- Support batch processing
- Minimal idle footprint
- Easy to maintain & setup
  - Use ‘boring’ 3rd-party dependencies
  - Limit moving parts
- Dev-friendly API for authoring NLP workflows & components
- Introspectable by client apps, e.g. *Seaku*

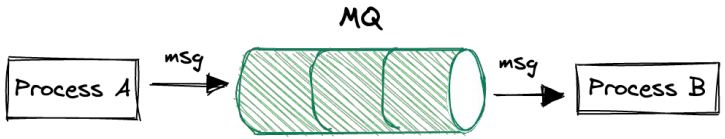
Assumptions:

- Modest traffic expected. Infinite scaling unnecessary.

orcaNLP : WF Engine + Python Library

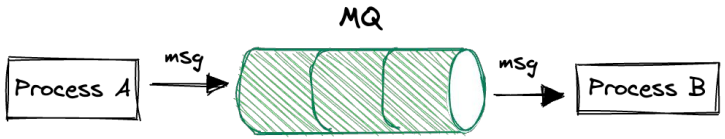
ENGINEERING

- As an engine…
  - Based on Message Queue (**MQ**) architecture
    - (Just like CLARIN-PL's WFE)
  - Distributable (No K8s needed)



orcaNLP : WF Engine + Python Library

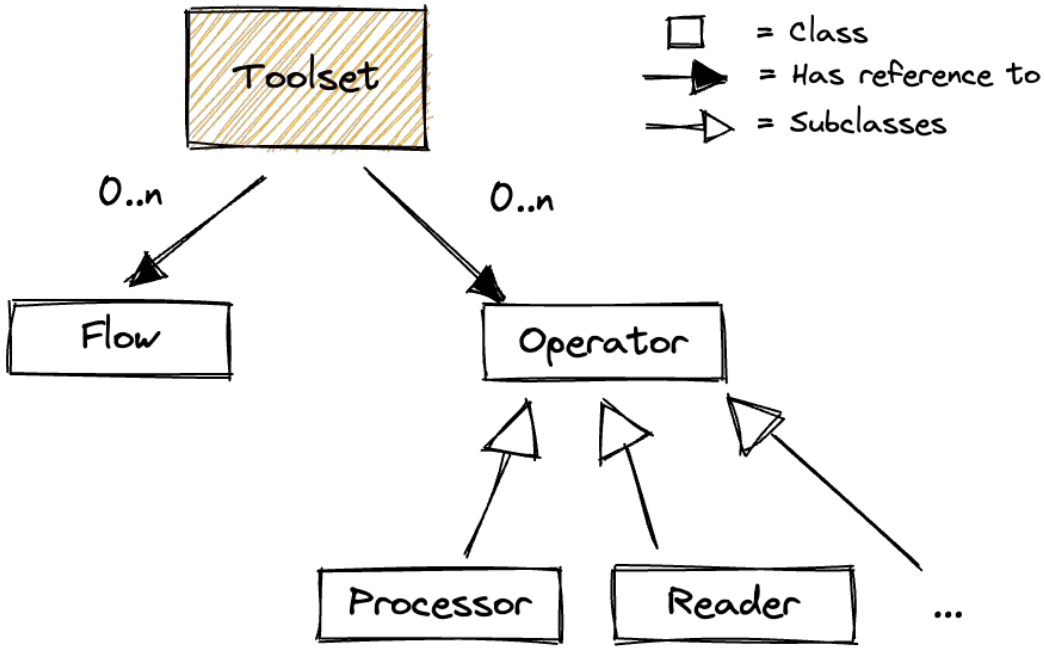
ENGINEERING



- As an engine…
  - Based on Message Queue (**MQ**) architecture
    - (Just like CLARIN-PL's WFE)
  - Distributable (No K8s needed)
- As a library…
  - Clients & workers `import orcanlp`
  - Provides abstractions to wrap existing tools/models -> interoperable
  - Focus on dev-friendliness
    - E.g. 'Everything-as-code' -> rely on IDE assistance
  - Utilities for setting up (e.g. project generation)

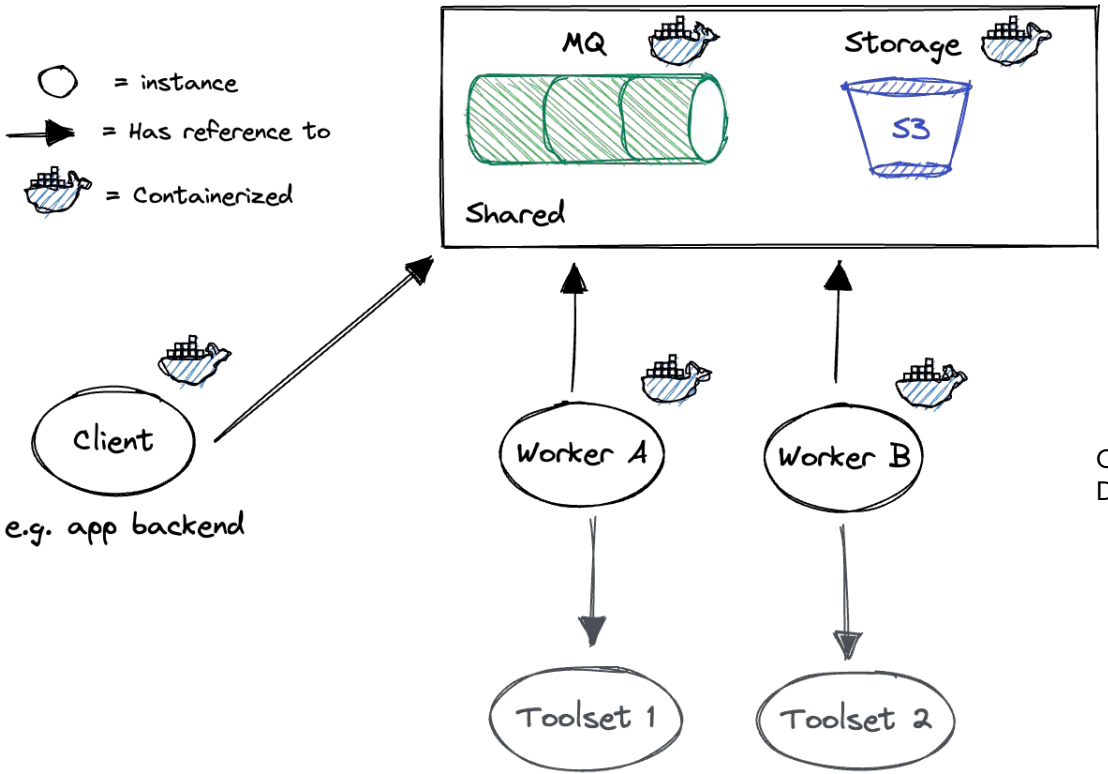
Main user-facing abstractions

ENGINEERING



Deployment Setup

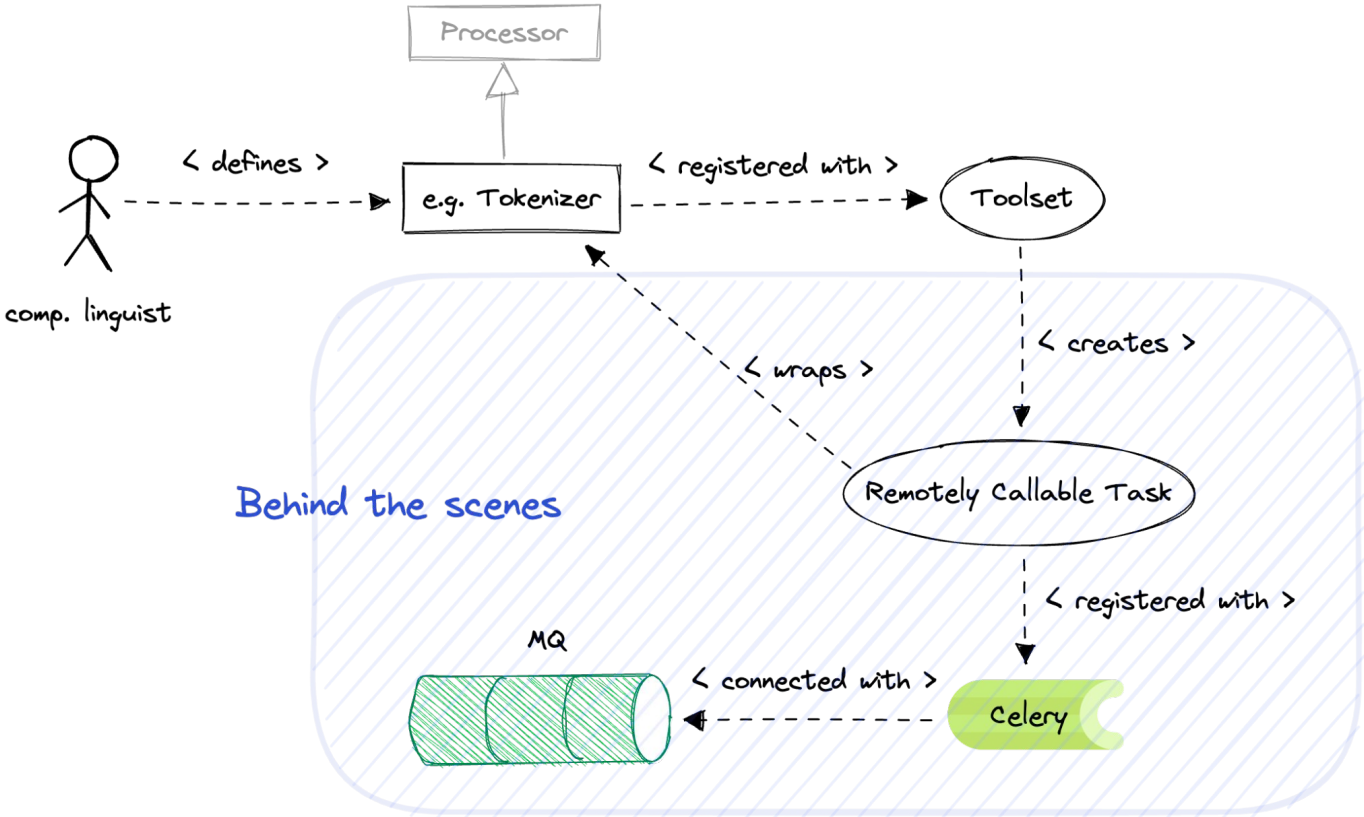
ENGINEERING



Orchestration:
Docker **Compose** or **Swarm Mode**

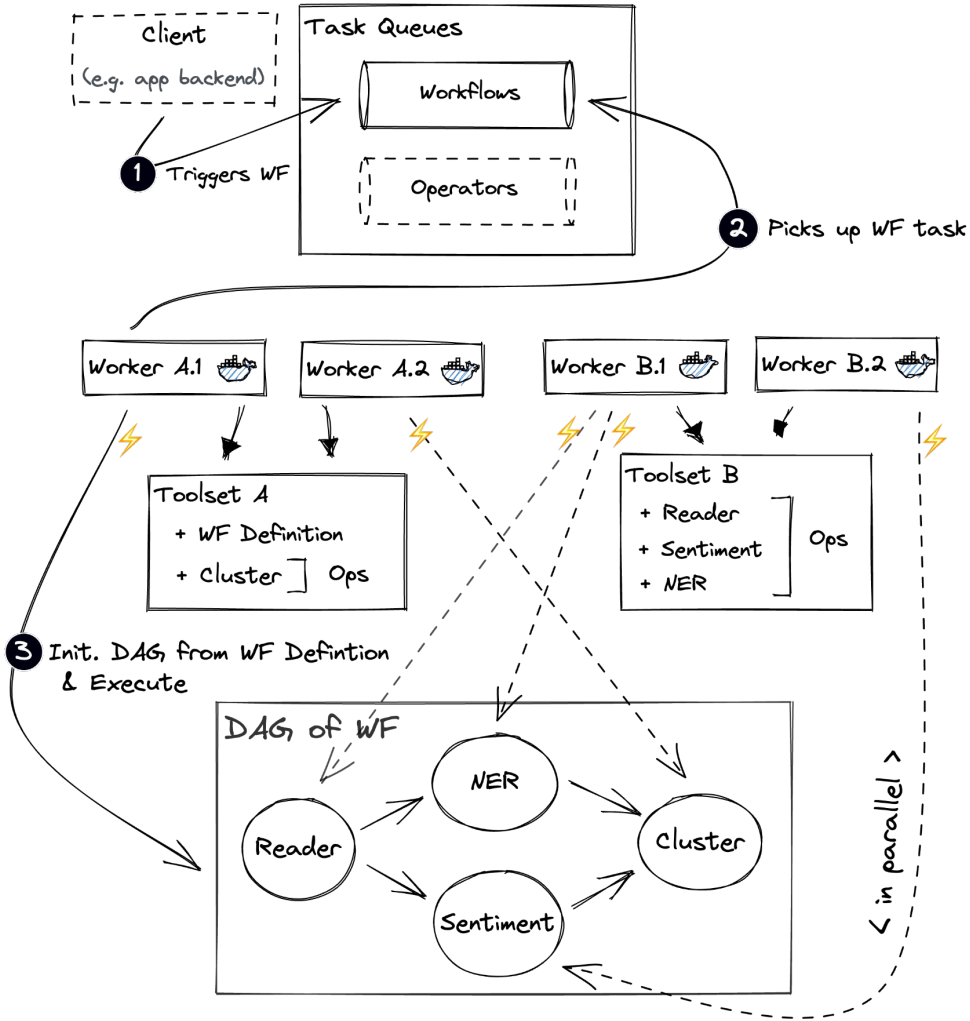
Hiding details of distributed system

ENGINEERING



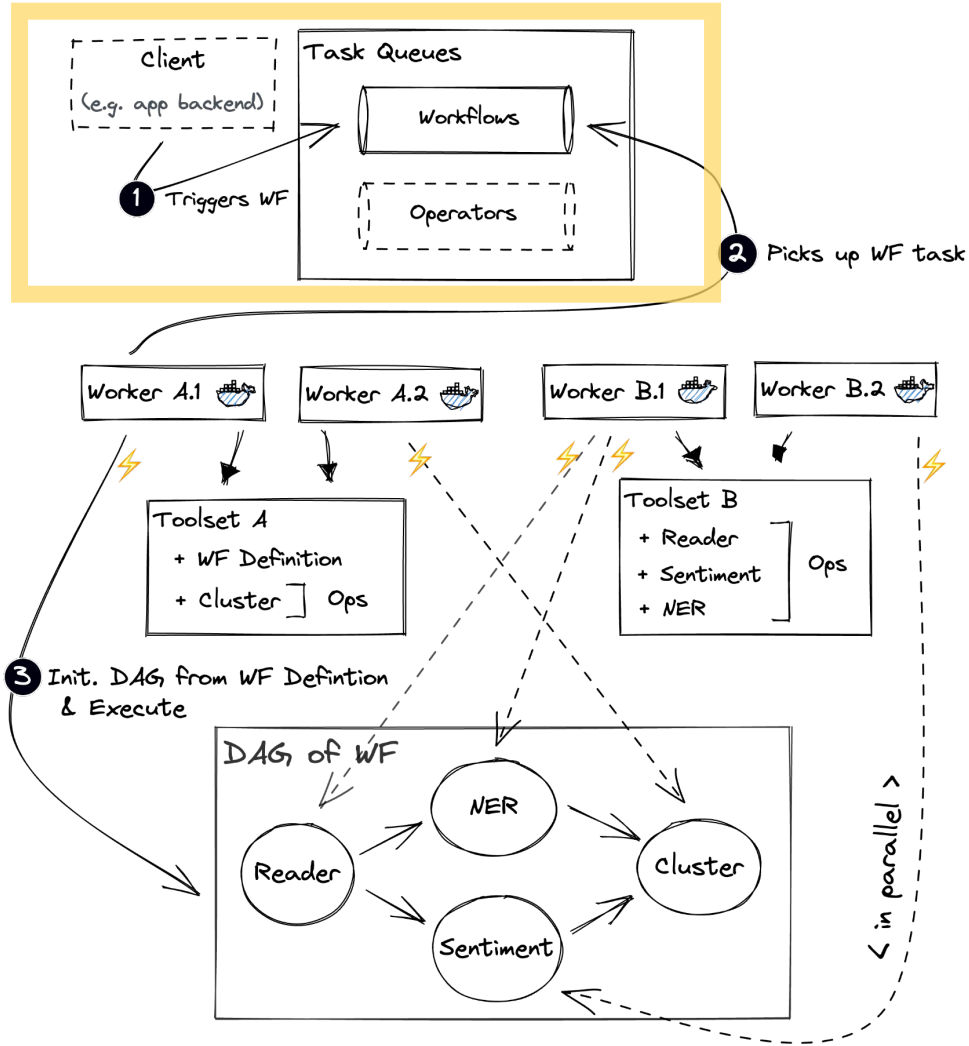
Lifecycle of a Workflow

ENGINEERING



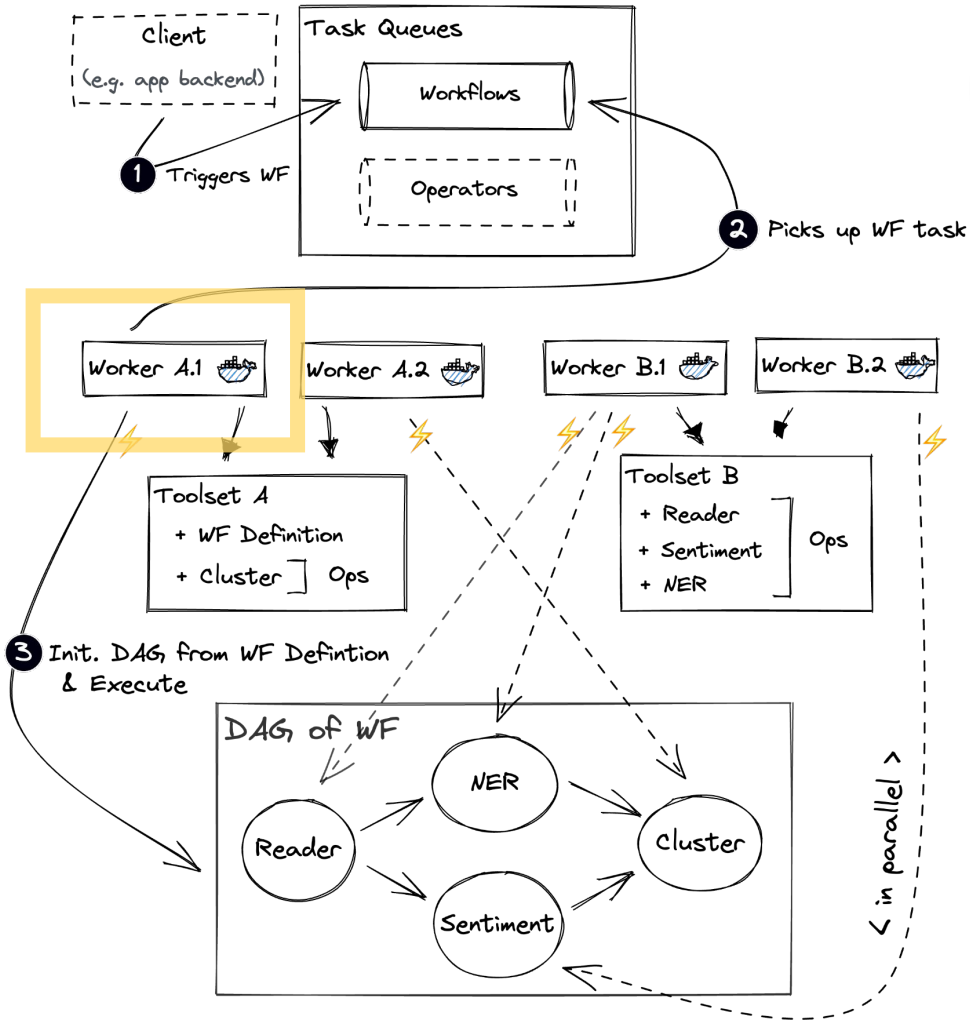
Lifecycle of a Workflow

ENGINEERING



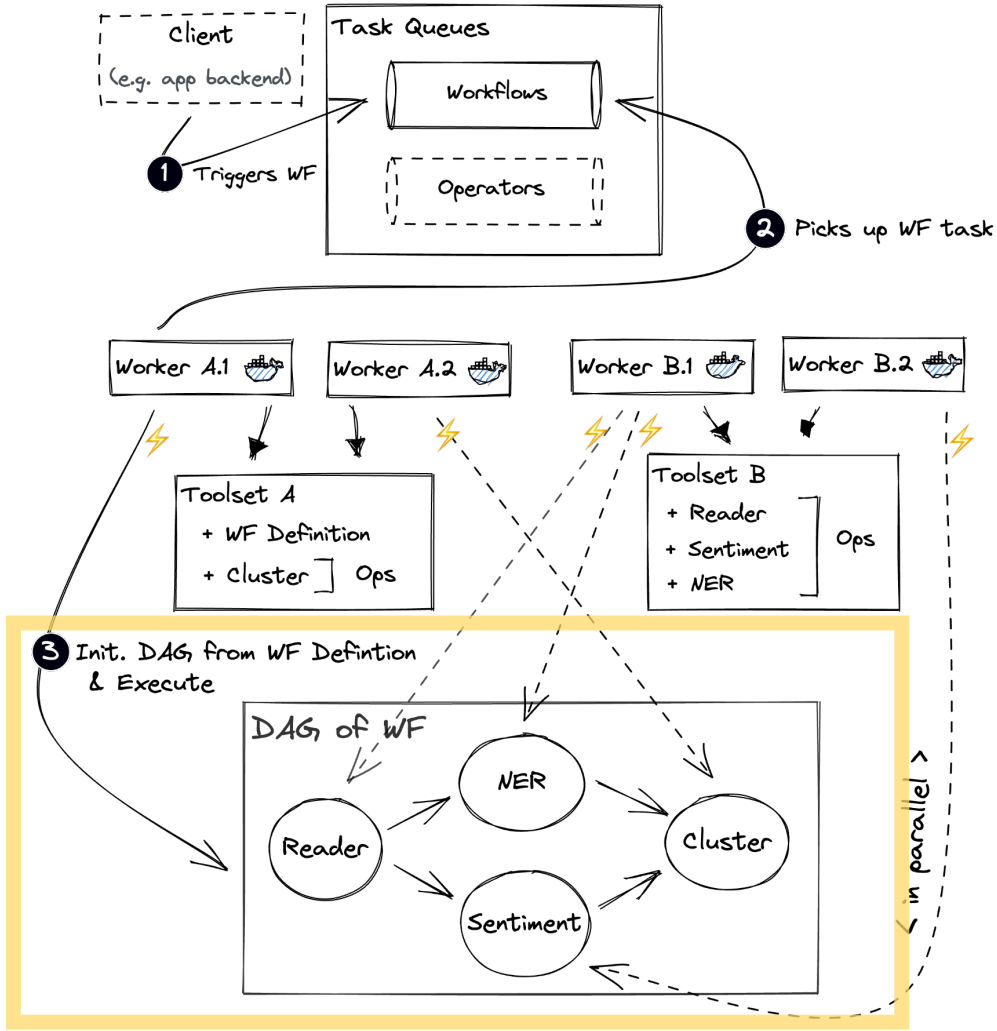
Lifecycle of a Workflow

ENGINEERING



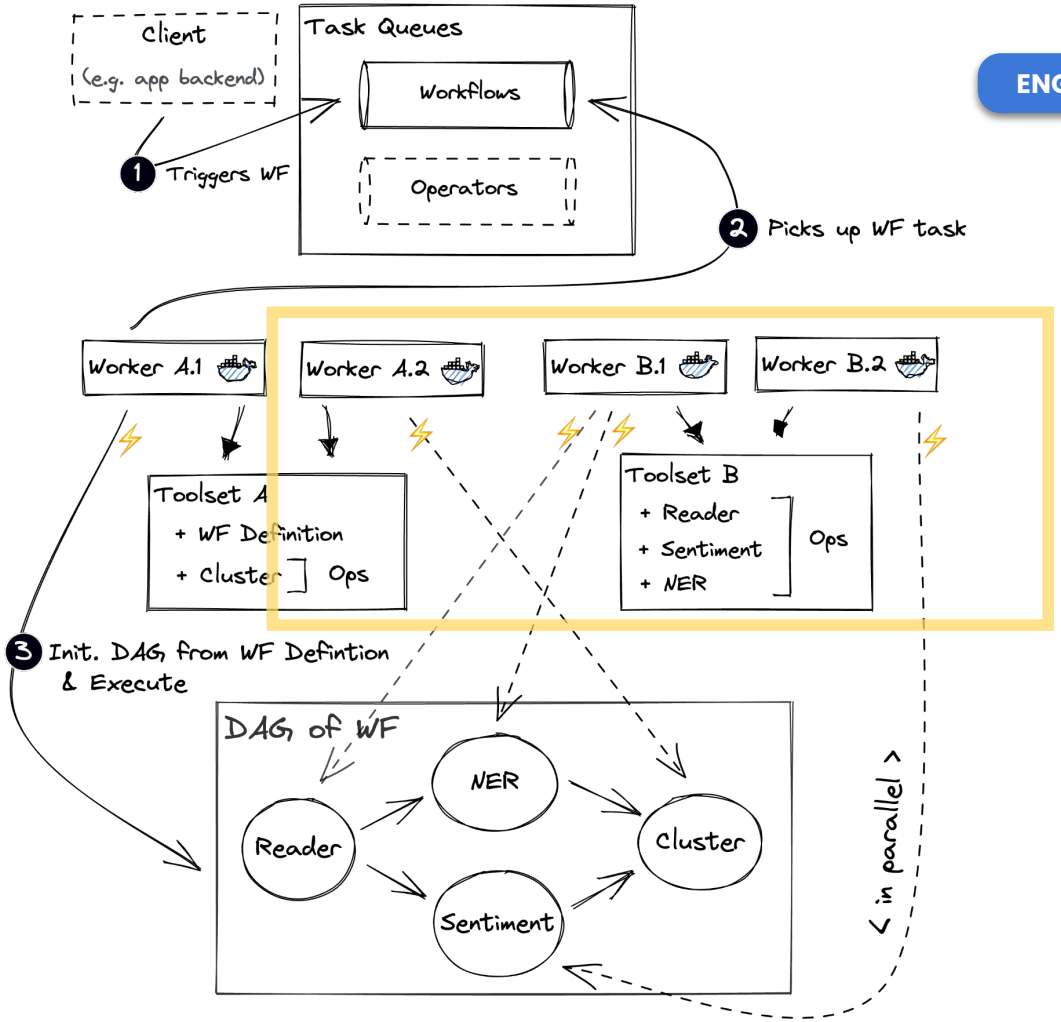
Lifecycle of a Workflow

ENGINEERING



Lifecycle of a Workflow

ENGINEERING



Defining an **Operator**

ENGINEERING

```python
class TextClusterer(BaseProcessor):

    meta = BaseProcessor.Meta(
        title="Document Clusterer",
        desc=(
            "Identify groups of similar documents based on the text they contain.",
        ),
        needs=["text"],
        assigns=["tag.cluster_id"],
        langs=None
    )

    @dataclass
    class Cfg(BaseProcessor.Cfg):
        hierarchical: bool = dataclasses.field(
            metadata={"desc": "Find groups *within* groups"}
        )
        number_of_clusters: Optional[int] = None

    def __call__(self, docarray: OnDiskDocArray) -> None:
        from sklearn.cluster import KMeans
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.pipeline import make_pipeline

        if self.cfg.number_of_clusters:
            raise NotImplementedError

        k = self.cfg.number_of_clusters or 20  # TODO: search for optimal no. clusters.

        pipeline = make_pipeline(TfidfVectorizer(), KMeans(n_clusters=k))
        texts = docarray[:, "text"]
        cluster_idxs = pipeline.fit_predict(texts)
        docarray[:, "tags.cluster_id"] = cluster_idxs
```

Defining an **Operator**

1. Inherit from Operator base class

1

ENGINEERING

```python
class TextClusterer(BaseProcessor):

    meta = BaseProcessor.Meta(
        title="Document Clusterer",
        desc=(
            "Identify groups of similar documents based on the text they contain.",
        ),
        needs=["text"],
        assigns=["tag.cluster_id"],
        langs=None
    )

    @dataclass
    class Cfg(BaseProcessor.Cfg):
        hierarchical: bool = dataclasses.field(
            metadata={"desc": "Find groups *within* groups"}
        )
        number_of_clusters: Optional[int] = None

    def __call__(self, docarray: OnDiskDocArray) -> None:
        from sklearn.cluster import KMeans
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.pipeline import make_pipeline

        if self.cfg.number_of_clusters:
            raise NotImplementedError

        k = self.cfg.number_of_clusters or 20  # TODO: search for optimal no. clusters.

        pipeline = make_pipeline(TfidfVectorizer(), KMeans(n_clusters=k))
        texts = docarray[:, "text"]
        cluster_idxs = pipeline.fit_predict(texts)
        docarray[:, "tags.cluster_id"] = cluster_idxs
```

Defining an **Operator**

1. Inherit from Operator base class
2. **Metadata-as-code**

1

ENGINEERING

```python
class TextClusterer(BaseProcessor):

    meta = BaseProcessor.Meta(
        title="Document Clusterer",
        desc=(
            "Identify groups of similar documents based on the text they contain.",
        ),
        needs=["text"],
        assigns=["tag.cluster_id"],
        langs=None
    )

    @dataclass
    class Cfg(BaseProcessor.Cfg):
        hierarchical: bool = dataclasses.field(
            metadata={"desc": "Find groups *within* groups"}
        )
        number_of_clusters: Optional[int] = None

    def __call__(self, docarray: OnDiskDocArray) -> None:
        from sklearn.cluster import KMeans
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.pipeline import make_pipeline

        if self.cfg.number_of_clusters:
            raise NotImplementedError

        k = self.cfg.number_of_clusters or 20  # TODO: search for optimal no. clusters.

        pipeline = make_pipeline(TfidfVectorizer(), KMeans(n_clusters=k))
        texts = docarray[:, "text"]
        cluster_idxs = pipeline.fit_predict(texts)
        docarray[:, "tags.cluster_id"] = cluster_idxs
```

Defining an **Operator**

1. Inherit from Operator base class
2. Metadata-as-code
3. Config schema

ENGINEERING

1

2

3

```python
class TextClusterer(BaseProcessor):

    meta = BaseProcessor.Meta(
        title="Document Clusterer",
        desc=(
            "Identify groups of similar documents based on the text they contain.",
        ),
        needs=["text"],
        assigns=["tag.cluster_id"],
        langs=None
    )

    @dataclass
    class Cfg(BaseProcessor.Cfg):
        hierarchical: bool = dataclasses.field(
            metadata={"desc": "Find groups *within* groups"}
        )
        number_of_clusters: Optional[int] = None

    def __call__(self, docarray: OnDiskDocArray) -> None:
        from sklearn.cluster import KMeans
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.pipeline import make_pipeline

        if self.cfg.number_of_clusters:
            raise NotImplementedError

        k = self.cfg.number_of_clusters or 20  # TODO: search for optimal no. clusters.

        pipeline = make_pipeline(TfidfVectorizer(), KMeans(n_clusters=k))
        texts = docarray[:, "text"]
        cluster_idxs = pipeline.fit_predict(texts)
        docarray[:, "tags.cluster_id"] = cluster_idxs
```

Defining an **Operator**

1. Inherit from Operator base class
2. Metadata-as-code
3. Config schema
4. **Wrap tool/core logic in `__call__()` body**

ENGINEERING

```python
class TextClusterer(BaseProcessor):

    meta = BaseProcessor.Meta(
        title="Document Clusterer",
        desc=(
            "Identify groups of similar documents based on the text they contain.",
        ),
        needs=["text"],
        assigns=["tag.cluster_id"],
        langs=None
    )

    @dataclass
    class Cfg(BaseProcessor.Cfg):
        hierarchical: bool = dataclasses.field(
            metadata={"desc": "Find groups *within* groups"}
        )
        number_of_clusters: Optional[int] = None

    def __call__(self, docarray: OnDiskDocArray) -> None:
        from sklearn.cluster import KMeans
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.pipeline import make_pipeline

        if self.cfg.number_of_clusters:
            raise NotImplementedError

        k = self.cfg.number_of_clusters or 20  # TODO: search for optimal no. clusters.

        pipeline = make_pipeline(TfidfVectorizer(), KMeans(n_clusters=k))
        texts = docarray[:, "text"]
        cluster_idxs = pipeline.fit_predict(texts)
        docarray[:, "tags.cluster_id"] = cluster_idxs
```

Defining a **Workflow**

(Similar process as before)

ENGINEERING

```python
class HistoricalEntitiesFlow(BaseDAGFlow):

    meta = BaseDAGFlow.Meta(
        title="Historical Dutch Entity Analysis Suite",
        desc=(
            "From scans of historical documents, extract named entities"
            " and use them to cluster documents."
            " Extra: identify entity nationalities.",
        ),
        langs=["nl"]
    )

    @dataclass
    class Cfg(BaseDAGFlow):
        people: bool = True
        places: bool = True
        organizations: bool = True

    def __call__(self) -> Set[Step]:
        ent_types = self.ent_types_from_cfg(self.cfg)
        return {
            Step(id="read", op="OCRReader", toolset="ivdnt-ocr", depends=None),
            Step(
                id="ner", op="HistoricalDutchNER", depends="read",
                cfg={"ent_types": ent_types},
            ),
            Step(id="cluster", op="EntityClusterer", depends="historical-ner"),
            Step(
                id="natcat", op="EntityNationalityCat", depends="ner"
            ),
            Step(
                id="join1", op="FieldJoiner",
                toolset="orca-essentials:v1", depends=["ner", "cluster", "natcat"],
                cfg={"fields": ["text", "tags.ents", "tags.cluster_id"]}
            ),
            Step(
                id="export", op="CsvExporter",
                depends="join1", toolset="orca-essentials:v1",
            )
        }
```

Defining a **Workflow**

(Similar process as before)

1. Inherit from **Flow** base class

```python
class HistoricalEntitiesFlow(BaseDAGFlow):

    meta = BaseDAGFlow.Meta(
        title="Historical Dutch Entity Analysis Suite",
        desc=(
            "From scans of historical documents, extract named entities"
            " and use them to cluster documents."
            " Extra: identify entity nationalities.",
        ),
        langs=["nl"]
    )

    @dataclass
    class Cfg(BaseDAGFlow):
        people: bool = True
        places: bool = True
        organizations: bool = True

    def __call__(self) -> Set[Step]:
        ent_types = self.ent_types_from_cfg(self.cfg)
        return {
            Step(id="read", op="OCRReader", toolset="ivdnt-ocr", depends=None),
            Step(
                id="ner", op="HistoricalDutchNER", depends="read",
                cfg={"ent_types": ent_types},
            ),
            Step(id="cluster", op="EntityClusterer", depends="historical-ner"),
            Step(
                id="natcat", op="EntityNationalityCat", depends="ner"
            ),
            Step(
                id="join1", op="FieldJoiner",
                toolset="orca-essentials:v1", depends=["ner", "cluster", "natcat"],
                cfg={"fields": ["text", "tags.ents", "tags.cluster_id"]}
            ),
            Step(
                id="export", op="CsvExporter",
                depends="join1", toolset="orca-essentials:v1",
            )
        }
```

Defining a **Workflow**

(Similar process as before)

1. Inherit from **Flow** base class
2. Metadata-as-code

ENGINEERING

1

```python
class HistoricalEntitiesFlow(BaseDAGFlow):
```

2

```python
meta = BaseDAGFlow.Meta(
    title="Historical Dutch Entity Analysis Suite",
    desc=(
        "From scans of historical documents, extract named entities"
        " and use them to cluster documents."
        " Extra: identify entity nationalities.",
    ),
    langs=["nl"]
)
```

```python
@dataclass
class Cfg(BaseDAGFlow):
    people: bool = True
    places: bool = True
    organizations: bool = True

def __call__(self) -> Set[Step]:
    ent_types = self.ent_types_from_cfg(self.cfg)
    return {
        Step(id="read", op="OCRReader", toolset="ivdnt-ocr", depends=None),
        Step(
            id="ner", op="HistoricalDutchNER", depends="read",
            cfg={"ent_types": ent_types},
        ),
        Step(id="cluster", op="EntityClusterer", depends="historical-ner"),
        Step(
            id="natcat", op="EntityNationalityCat", depends="ner"
        ),
        Step(
            id="join1", op="FieldJoiner",
            toolset="orca-essentials:v1", depends=["ner", "cluster", "natcat"],
            cfg={"fields": ["text", "tags.ents", "tags.cluster_id"]}
        ),
        Step(
            id="export", op="CsvExporter",
            depends="join1", toolset="orca-essentials:v1",
        )
    }
```

Defining a **Workflow**

(Similar process as before)

1. Inherit from **Flow** base class
2. Metadata-as-code
3. Config schema

ENGINEERING

```python
class HistoricalEntitiesFlow(BaseDAGFlow):

    meta = BaseDAGFlow.Meta(
        title="Historical Dutch Entity Analysis Suite",
        desc=(
            "From scans of historical documents, extract named entities"
            " and use them to cluster documents."
            " Extra: identify entity nationalities.",
        ),
        langs=["nl"]
    )

    @dataclass
    class Cfg(BaseDAGFlow):
        people: bool = True
        places: bool = True
        organizations: bool = True

    def __call__(self) -> Set[Step]:
        ent_types = self.ent_types_from_cfg(self.cfg)
        return {
            Step(id="read", op="OCRReader", toolset="ivdnt-ocr", depends=None),
            Step(
                id="ner", op="HistoricalDutchNER", depends="read",
                cfg={"ent_types": ent_types},
            ),
            Step(id="cluster", op="EntityClusterer", depends="historical-ner"),
            Step(
                id="natcat", op="EntityNationalityCat", depends="ner"
            ),
            Step(
                id="join1", op="FieldJoiner",
                toolset="orca-essentials:v1", depends=["ner", "cluster", "natcat"],
                cfg={"fields": ["text", "tags.ents", "tags.cluster_id"]}
            ),
            Step(
                id="export", op="CsvExporter",
                depends="join1", toolset="orca-essentials:v1",
            )
        }
```

Defining a **Workflow**

ENGINEERING

(Similar process as before)

1. Inherit from **Flow** base class
2. Metadata-as-code
3. Config schema
4. `__call__()` returns WF `Steps`
    - WF defined **dynamically**, e.g. <- config

1

```python
class HistoricalEntitiesFlow(BaseDAGFlow):
```

2

```python
    meta = BaseDAGFlow.Meta(
        title="Historical Dutch Entity Analysis Suite",
        desc=(
            "From scans of historical documents, extract named entities"
            " and use them to cluster documents."
            " Extra: identify entity nationalities.",
        ),
        langs=["nl"]
    )
```

3

```python
    @dataclass
    class Cfg(BaseDAGFlow):
        people: bool = True
        places: bool = True
        organizations: bool = True
```

4

```python
    def __call__(self) -> Set[Step]:
        ent_types = self.ent_types_from_cfg(self.cfg)
        return {
            Step(id="read", op="OCRReader", toolset="ivdnt-ocr", depends=None),
            Step(
                id="ner", op="HistoricalDutchNER", depends="read",
                cfg={"ent_types": ent_types},
            ),
            Step(id="cluster", op="EntityClusterer", depends="historical-ner"),
            Step(
                id="natcat", op="EntityNationalityCat", depends="ner"
            ),
            Step(
                id="join1", op="FieldJoiner",
                toolset="orca-essentials:v1", depends=["ner", "cluster", "natcat"],
                cfg={"fields": ["text", "tags.ents", "tags.cluster_id"]}
            ),
            Step(
                id="export", op="CsvExporter",
                depends="join1", toolset="orca-essentials:v1",
            )
        }
```

Toolset contribution story

ENGINEERING

1. `$ pip install orcanlp` (soon™)
    - Python **3.7** or later
2. `$ orcanlp init` to generate project structure
3. Define Operators/DagFlow + add to Toolset
4. Track Py dependencies with Poetry & *pyproject.toml*
5. Modify/replace default Dockerfile
    - Non-Py dependencies installed here.
6. `$ orcanlp preflight` to find issues
7. *Seaku-specific* (WIP)
    - Push to any remote on GitHub
    - Open PR in *Seaku* repo adding remote url to toolset index
    - Code review -> build -> deploy

Discussion

ENGINEERING

Strengths

- Initial stress tests promising
- Batch (corpus) processing support
- Lightweight
  - On-demand only
  - No K8s overhead
- Small codebase (< 600 LoC)
- Easy to set-up locally
- Dev-friendly API
  - Clear abstractions
  - Details of remote tasks hidden

Discussion

ENGINEERING

Strengths

- Initial stress tests promising
- Batch (corpus) processing support
- Lightweight
  - On-demand only
  - No K8s overhead
- Small codebase (< 600 LoC)
- Easy to set-up locally
- Dev-friendly API
  - Clear abstractions
  - Details of remote tasks hidden

Weaknesses

- Harder to reason about resource limits
- Elastic scaling less obvious
- Deadlocks possible
  - (So far unobserved)
- Between steps, delay while pushing/fetching data to/from remote storage
  - Underutilization of resources.

Where is the code?

ENGINEERING

Aiming for open-source release in **Q2 2023**.

- Cleaning up code
- Documentation
- Example Toolsets demo'ing best practices
- More tests
- Proper CI/CD

Summary

- 👩‍🎓 We’re building ***Seaku*** - text analytics software for researchers

Summary

- 🧑‍🎓 We’re building ***Seaku*** - text analytics software for researchers

- ⚙️ NLP workflows/pipelines powered by our custom workflow engine → ***orcaNLP***

Summary

- 🧑‍🎓 We're building ***Seaku*** - text analytics software for researchers

- ⚙️ NLP workflows/pipelines powered by our custom workflow engine
  → ***orcaNLP***

- 👷 To grow workflow catalog, make contributing as easy as possible
  -> **dev-friendly** API + tooling

Summary

- 🧑‍🎓 We're building ***Seaku*** - text analytics software for researchers

- ⚙️ NLP workflows/pipelines powered by our custom workflow engine
  → ***orcaNLP***

- 👷 To grow workflow catalog, make contributing as easy as possible
  -> **dev-friendly** API + tooling

- 🚀 Open-source **release** in Q2 next year

FAQs

ENGINEERING

- Q: What about non-Python/older than Python 3.7 tools?
  - A: Include in Docker image & call as subprocess.
- Q: Isn't batch processing memory-intensive?
  - A: DocArrays (corpus objects) are backed by on-disk SQLite DB -> low memory impact.
- Q. How is workflow execution monitored?
  - A: Celery Flower gives us this for free.
- Q: Can a single worker perform multiple Operator tasks at once?
  - A: In principle, yes; but we limit buffer size to 1, so 1 container = 1 task
- Q: How do you avoid dependency conflicts when installing different Toolsets into the Client process environment?
  - A: Toolset-specific deps are skipped (possible because they are only imported within __call__() body)