

# Intégration de la tâche de vectorisation à la plateforme EIDA

## Cahier des charges

### Objectifs scientifiques

EIDA est une plateforme dédiée à l'étude des diagrammes astronomiques, dont l'ambition est de former un véritable SI rassemblant des sources de traditions diverses et les outils permettant leur étude. Dans ce cadre, s'élabore bloc par bloc une pipeline pour le traitement automatique des sources historiques visuelles dans le but de produire un outil de recherche s'appuyant sur des techniques de *computer vision* comme assistant à l'exploration d'un corpus de diagrammes. Les tâches d'extraction d'objets et de recherche de similarité sont déjà implémentées, et il s'agit maintenant d'inclure la tâche de vectorisation des diagrammes extraits. Cette approche permettrait non seulement l'automatisation de certaines étapes de l'élaboration d'une édition critique, mais ouvre également la voie à une recherche basée sur le contenu géométrique de ces diagrammes.

### Axe 1 : Implémentation de la vectorisation à l'API

#### Une API

Dans une logique de développement modulaire, [l'API](#) permet de déplacer les tâches accomplies par les modèles de vision et nécessitant une puissance de calcul importante sur le GPU. Les inférences avec les modèles de détection, extraction des similarités et vectorisation ne se déroulent pas directement dans l'application pour ne pas entraîner de ralentissement général de l'exécution sur le serveur web. Le recours à une API garantit la sécurité des échanges entre le serveur web et l'API.

L'API, récemment refondue et dockerisée, inclut déjà l'extraction automatique et la recherche de similarité (*similarity retrieval*), il s'agira donc d'implémenter le module de vectorisation.

#### Une approche modulaire

Dans une logique open-source et modulaire, on veut permettre à l'utilisateur·ice de choisir les fonctionnalités qu'il souhaite implémenter tout en gardant une seule API cohérente.

Une seule API est déployée, mais les utilisateur·ices peuvent choisir d'activer uniquement les modules pertinents pour leurs besoins spécifiques. Cette modularité se reflète dans la structure des dossiers, où chaque module (ou tâche) peut être isolé dans son propre environnement. Chaque dossier contient les fichiers nécessaires pour exécuter une tâche spécifique.

L'utilisation de fichiers de configuration (.env) permet de définir facilement quels modules doivent être activés ou désactivés. Les utilisateur·ices peuvent alors personnaliser leur instance locale de l'API, en facilitant la gestion et la maintenance.

```
# api/.env  
  
# apps (folder names) to be imported to the API  
INSTALLED_APPS=similarity,extraction
```

## Docker : gestion de la mise en production

Lorsqu'un service comme Redis ou Celery fonctionne sans être conteneurisé, il partage l'environnement système avec d'autres processus. Cela signifie que les ports utilisés par ces services sont potentiellement accessibles à sur la même machine, ce qui peut entraîner des conflits de port et perturber le fonctionnement normal de ces services.

En conteneurisant les services, on crée un environnement isolé pour chacun d'eux. Les conteneurs utilisent des espaces de noms réseau distincts, ce qui signifie que chaque conteneur a sa propre pile réseau virtuelle, y compris ses propres interfaces, adresses IP et tables de routage. De cette manière, les ports utilisés par Redis, Celery, ou tout autre service, sont confinés à l'intérieur du conteneur et ne sont pas accessibles aux autres processus sur le même GPU ou même sur la même machine physique.

En outre, les conteneurs offrent également d'autres avantages en termes de gestion des dépendances et de la configuration. En encapsulant chaque service dans son propre conteneur, il devient plus facile de gérer les versions des bibliothèques, les paramètres de configuration et les dépendances spécifiques à chaque service, sans craindre d'interférences avec d'autres services ou processus.

## Nouveau module de vectorisation : description haut niveau

L'API reçoit une requête HTTP de la part de l'application EIDA lorsque l'utilisateur·ice envoie une requête de vectorisation sur un témoin ou un ensemble de diagrammes. Un fichier JSON contenant une liste d'URLs est généré par l'application et envoyé à un API endpoint qui le lit et enregistre les images qui seront ensuite traitées par le modèle. Les

fichiers SVGs générés seront retournés à l'application en réponse à la requête. L'application reçoit, lit et écrit les fichiers, qui pourront alors être visualisés par l'utilisateur, selon différentes modalités (cf. Annexe A).

## Types d'input

### a. Manifeste entier EIDA

Après validation des annotations d'un témoin dans la plateforme, un bouton apparaît pour lancer l'inférence avec le modèle de vectorisation, en envoyant un fichier JSON à l'API. Une fois la tâche de vectorisation terminée un bouton apparaît pour visualiser les vectorisations.

Exemple de structuration du JSON envoyé à l'API pour un témoin entier :

```
{
  "doc_id": "wit14_img14_anno14",
  "model_name" : "0036",
  "images": {
    "wit14_img14_0029_255,363,341,614":
    "https://eida.obspm.fr/iiif/2/wit14_img14_0029.jpg/255,363,341,614/full/0/default.jpg",
    "wit14_img14_0029_604,355,313,379":
    "https://eida.obspm.fr/iiif/2/wit14_img14_0029.jpg/604,355,313,379/full/0/default.jpg",
    "wit14_img14_0030_250,393,540,587":
    "https://eida.obspm.fr/iiif/2/wit14_img14_0030.jpg/250,393,540,587/full/0/default.jpg",
    "wit14_img14_0033_627,426,327,429":
    "https://eida.obspm.fr/iiif/2/wit14_img14_0033.jpg/627,426,327,429/full/0/default.jpg",
    "wit14_img14_0033_231,393,382,427":
    "https://eida.obspm.fr/iiif/2/wit14_img14_0033.jpg/231,393,382,427/full/0/default.jpg",
    "wit14_img14_0035_299,352,332,350":
    "https://eida.obspm.fr/iiif/2/wit14_img14_0035.jpg/299,352,332,350/full/0/default.jpg",
    ...
  }
}
```

Il faut laisser ouverte la possibilité de spécifier un modèle dans les données JSON traitées par l'API, même si à l'heure actuelle un seul modèle est disponible. Cette évolution est motivée par les développements en prévision qui pourraient introduire un choix parmi plusieurs modèles dans un *formulaire de lancement de traitement*.

Une variable de configuration doit être définie pour spécifier le modèle par défaut. Si aucun modèle n'est précisé, cette variable sera utilisée et écrite dans le JSON généré.

## b. Une image particulière ou une liste personnalisée d'URLs

Il est aussi nécessaire de penser au lancement de la vectorisation sur un ou plusieurs diagrammes choisis. Quand il sera implémenté, on pourra aussi effectuer ce choix sur l'interface de visualisation des *crops* de page. Le formulaire de lancement de traitement permettra de lancer des actions sur ce sous-ensemble de données choisi.

Les données envoyées seront structurées différemment.

Exemple de structure du JSON pour une image simple :

```
{
  "set_id": "XXX",
  "model_name" : "0036",
  "images": {
    "wit14_img14_0029_255,363,341,614":
    "https://eida.obspm.fr/iiif/2/wit14_img14_0029.jpg/255,363,341,614/full/0/default.jpg"
  }
}
```

## Modèle

[Le script d'inférence implémenté](#) est basé sur [DINO-DETR](#) modifié pour prédire les primitives pré-déterminées sur les diagrammes astronomiques au préalable extraits des manuscrits.

Le modèle est entraîné sur des images synthétiques puis sur un dataset authentique et spécifique de 303 diagrammes astronomiques de traditions diverses, datant du XII<sup>e</sup> au XVIII<sup>e</sup> siècle, et annoté de plus de 3000 lignes, cercles et arcs par les chercheur·ses.

Il est nécessaire de prendre en compte la possibilité de choisir entre plusieurs modèles, celui-ci étant amené à évoluer (*fine-tuning* et prédiction de nouvelles primitives).

L'implémentation du modèle se fait par import de l'algorithme, du dernier checkpoint et de son fichier de configuration dans l'API. Le répertoire /vectorization contient les fonctions qui construisent le modèle. Il s'agira de rédiger les fonctions et les tâches *Dramatique* associés aux endpoints de l'API qui traitent les inputs, téléchargent les images et lancent l'inférence sur les données.

## Output

La sortie sans traitement du modèle de vectorisation, pour chaque image, est un fichier NPZ.

Les fichiers NPZ sont des archives compressées au format ZIP et permettant de stocker plusieurs arrays NumPy de manière compacte, chaque array étant associé à une clé ou

un nom. Dans notre cas, chaque array correspond à une primitive prédite et contient ces coordonnées. Un traitement de ces premiers résultats du modèle permet d'écrire des SVGs.

Plusieurs options s'offrent alors :

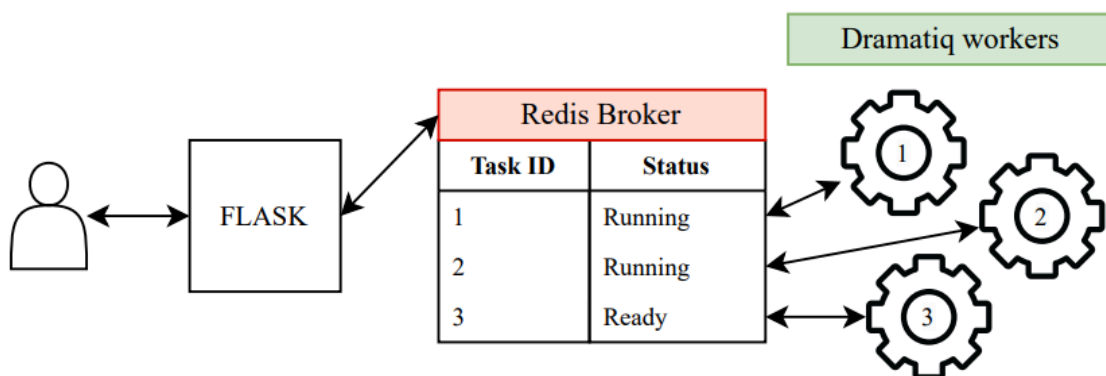
- générer ou non les fichiers NPZ ?
- quel format de fichier envoyer à l'API ?
- si les NPZ sont générés, les enregistrer ou non ? où les enregistrer ?
- si les NPZ sont générés, où lancer l'écriture des SVGs ?

La solution la plus optimale, et donc celle envisagée est l'écriture directe des SVGs dans l'API. Les fonctions de post processing de l'inférence permettent de sortir des résultats dans un format spécifique à InkScape (syntaxe et rajout des namespaces). Des SVGs sont donc générés et envoyés en sortie. Ils sont enregistrés temporairement dans l'API et supprimés par un *Cron Dramatiq* (cf. ci-dessous) déclenché après une semaine.

## Task queues

Le modèle de vision exige des ressources computationnelles importantes. L'implémentation dans l'API doit prendre en compte ces exigences de performance et garantir en conséquence une scalabilité adéquate.

Pour éviter la surcharge et gérer les multiples requêtes, l'API dans sa nouvelle version dockerisée utilise un nouveau système de threading. Il s'agit du task manager [Dramatiq](#), assurant les tâches de fond et l'exécution des requêtes reçues en imposant leur traitement successif. Dramatiq fonctionne avec [Redis](#), magasin de clés/valeurs NoSQL, en mémoire et Open-Source, il constitue une DB intermédiaire qui attribue les tâches aux workers. Dramatiq gère la bonne tenue des tâches, faisant l'intermédiaire entre les workers et le système de login.



## Sécurité

L'API utilise un décorateur pour restreindre les hôtes autorisés à envoyer des requêtes aux endpoints : seuls l'hôte obspm.fr peut envoyer des requêtes pour l'inférence, évitant ainsi un risque de surcharge par des requêtes envoyées depuis un autre hôte.

Pour protéger les données en transit et assurer une communication sécurisée entre les clients de l'application VHS (qui tourne sur les serveurs de l'ISCD) et l'API, un chiffrement est mis en place via un proxy. Un proxy est un routeur qui fournit une passerelle, une couche supplémentaire entre les utilisateur·ices et le GPU. Ce proxy s'assure que toutes les communications sont cryptées, empêchant ainsi toute interception ou altération des données.

## Interaction avec l'utilisateur·ice dans EiDA

### 1. Avec le manifest

- Développement court-terme : via le formulaire automatique de Django et via l'interface **Witness**

L'interface administrateur de Django propose un formulaire de lancement de traitement sur un ensemble de témoins sélectionnés : le rajout d'un traitement dépend de l'écriture d'une @admin.action.

Select Witness to change

Search

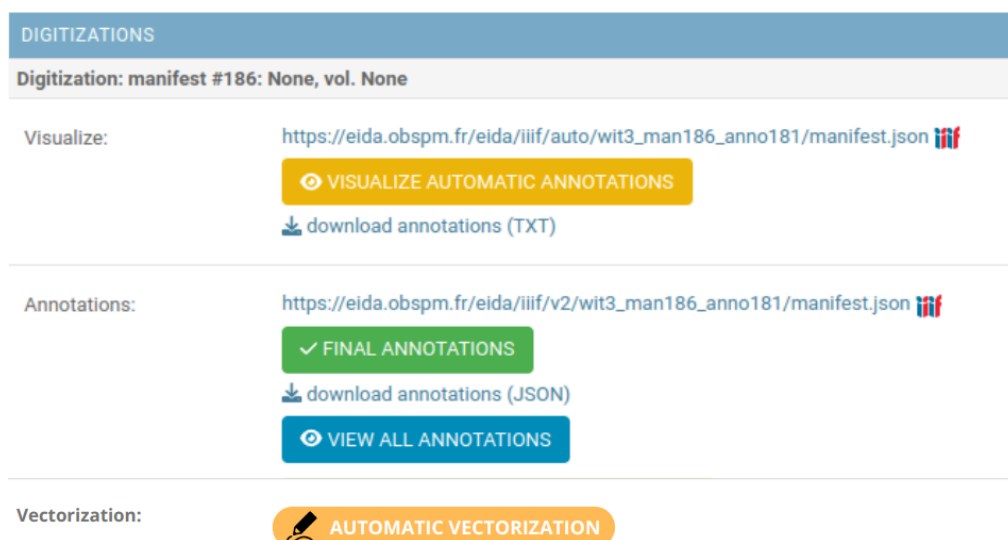
Action: [dropdown] Go 3 of 100 selected

ID	DATES	WORK	HISTORICAL ACTORS	DIGITIZ
1	France	801-900	Ptolemy   Almagest	Digitiza
2	terio de El Escorial	-	-	Digitiza
3	Chinois 4957	Paris   Bibliothèque nationale de France	-	Digitiza
4	Lat. 7214	Paris   Bibliothèque nationale de France	-	Digitiza
5	Lat. 16211	Paris   Bibliothèque nationale de France	-	Digitiza

Add "Compute vectorization"

Pour chaque **Witness**, la **Digitization** choisie est la première. Autant de requêtes que de **Digitization** sont envoyées à l'API.

La vectorisation des images peut aussi être lancée via l'interface de gestion d'un **Witness** : un bouton apparaît après l'étape de validation des annotations au niveau de la **Digitization**.



Il est nécessaire, en outre, de prendre en compte le temps de réponse de la requête du point de vue de l'utilisateur·ice. La fin d'un traitement lancé par un·e utilisateur·ice est marqué de l'envoi d'un mail. En outre, une tooltip prévient l'utilisateur·ice du bon lancement du traitement, ou de l'erreur de lancement, le cas échéant. L'apparition du bouton de visualisation des vectorisation dans l'interface signifie expressément la fin du traitement et le retour des résultats.

Sur le modèle des modules d'extraction et de similarités, une condition sur l'apparition du bouton de visualisation des vectorisations a déjà été implémentée. Si au moins un diagramme du **Witness** a été vectorisé, le bouton de visualisation des vectorisations est activé, et l'utilisateur·ice peut consulter les résultats.

La condition d'apparition de la visualisation s'appuie sur la méthode de classe suivante :

```
def has_vectorization(self):  
    # if there is at least one SVG file named after the current digitization  
    if len(glob(f"{SVG_PATH}/{self.get_ref()}*.svg"))  
        return True  
    return False
```

Python

Concernant les conditions de lancement du traitement (sélection dans l'interface admin ou disparition du bouton) : il faut empêcher le lancement de l'action si toutes les images du **Witness** ont bien un SVG associé. De cette manière, la vectorisation peut être

relancée facilement par l'utilisateur·ice si l'inférence initiale n'a pas ramené l'intégralité des résultats attendus. En conséquence les deux boutons (visualisation et lancement) ne s'excluent pas nécessairement l'un l'autre, en prévision de la manipulation des objets à un niveau de granularité plus fin (création et lancement de traitements sur un set de diagrammes).

La condition du lancement s'appuie sur la méthode de classe suivante :

```
Python
def has_all_vectorization(self):
    # if there is as much svg files as there are images in the current digitization
    if len(glob(f"{SVG_PATH}/{self.get_ref()}_*.svg")) == len(glob(f"
{IMG_PATH}/{self.get_ref()}_*.jpg")):
        return True
    return False
```

Les évolutions planifiées pour la plateforme devraient permettre une meilleure gestion de l'administration et l'état des traitements sur un ensemble de document donné, avec l'extension d'une table qui répertorie les actions (**Treatments**) effectués sur chaque entité ; elle permettra de trouver une trace des objets traités ensemble, et la liste des actions effectuées par un·e utilisateur·ice sur cet ensemble.

b. Développement long-terme : via le formulaire de lancement des traitements

Quand il sera implémenté, il sera possible de lancer la vectorisation via *Formulaire de lancement de Treatment*.

Une fois des enregistrements sélectionnés dans un panier, l'utilisateur·ice aura accès à une page pour lancer un traitement sur les documents sélectionnés, en sélectionnant des paramètres : choix du modèle, threshold... (reste encore à déterminer).

## 2. Avec un diagramme

Dans le cadre de la refonte de l'application, il a été évoqué le besoin de manipuler les diagrammes unitaires. En annexe B, un tableau résume les avantages et inconvénients des différentes solutions envisagées pour ce faire. Après discussions, l'idée de rajouter un niveau de granularité au modèle de données a été abandonnée. L'identification et la manipulation continueront de passer par une référence et les solutions proposées par l'API IIIF, et un unique diagramme sera déjà considéré comme un **RegionSet**.

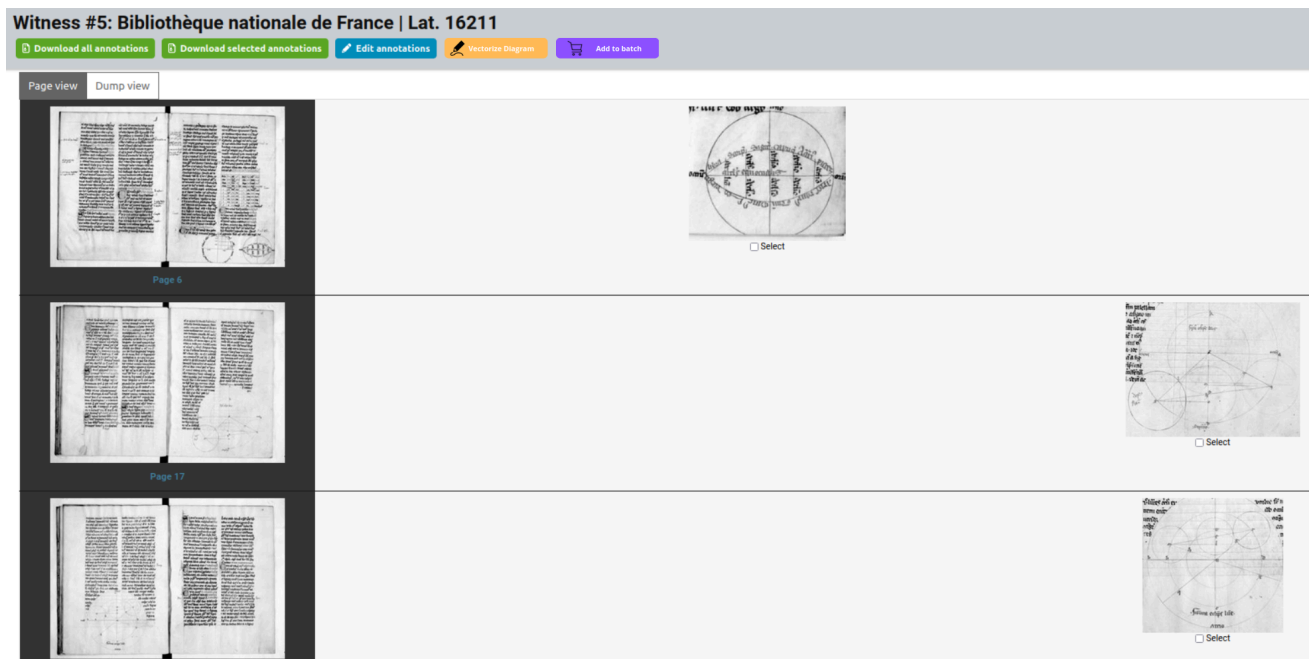
## 3. Avec un batch de diagrammes

Les développements prévues pour la plateforme permettront de créer des "batches" d'éléments issus de plusieurs témoins. Ces éléments pourraient être sélectionnés via la



même interface de visualisation des crops de diagrammes, grâce à un bouton “Add to Batch” ou “Add to Set”. Un **Set**, serait lié à un utilisateur·ice. Son exploitation se fait via l’interface de lancement de traitements.

Aperçu de l’interface de visualisation des extractions : il est déjà possible de switcher entre une vue “dump” (uniquement les extractions) et la vue ci-dessous qui donne aussi le contexte de la page dans une colonne sur la gauche.




## Interaction avec les admins

En cas d’échec de la vectorisation (ou bien pour la relancer avec un nouveau modèle), il est nécessaire de créer un autre endpoint dans l’application servant, pour un certain témoin, à relancer la vectorisation après la déléition des images sur l’API. Si de nouveaux SVGs correspondant aux régions découpées arrivent dans EIDA, elles écraseront automatiquement les anciennes.

## Workflow

### D’un witness

1. Requête utilisateur·ice dans l’application EIDA;
2. Création d’un fichier JSON : l’image correspondant au diagramme est identifiée par une URL construite conformément aux standards de l’API Image de IIIF, ce qui permet de ramener uniquement la portion extraite;

- Parser le.s manifeste.s et ramener les URLs dans une liste;
3. Envoie du JSON via requête HTTP POST à l'API endpoint (un JSON par manifeste);
  4. Appel de la fonction Python;
    - a. Vérification du JSON;
    - b. Enregistrement des images issues du JSON pour leur traitement;
    - c. Récupération des données pour lancer la tâche;
  5. Lancement de l'inférence;
    - a. Traitement des images par le modèle : écriture des SVGs;
-  Convention de nommage : le SVG correspondant porte le même nom que l'image JPG;
- b. Envoi d'un zip contenant les fichiers XML ;
  6. Récupération des SVGs au endpoint de l'application ;
  7. Dézip, lecture et écriture des SVGs en backend dans l'application EiDA;
    - a. Vérification dans le répertoire de résultats (dossier SVG dans /mediafiles) que deux fichiers du même nom n'existent pas. S'il existe déjà un fichier du même nom, il est écrasé par le nouveau.
    - b. Apparition du bouton de visualisation des vectorisation sur l'interface du witness

## D'un batch

À ce stade du développement, il est logique de relier l'implémentation de la vectorisation à la manipulation d'ensembles personnalisés d'éléments. Le modèle de données est désormais enrichi de deux nouvelles entités : **Set** et **Treatment**.

Le set est un ensemble de documents regroupés dans le "panier". L'entité **Treatment** va donc admettre différents types d'objets traités (pas seulement les numérisations, comme c'est le cas aujourd'hui). Les objets du **Set** seront reliés à l'action réalisée via des paramètres stockés en JSON dans l'entité **Treatment**. On distingue les **RegionSets** (sélection de diagrammes) des **DocumentSets** (sélection de témoins ou d'œuvres ou de séries ; la sélection d'une œuvre ou d'une série ruisselant sur l'ensemble des témoins à laquelle elle est liée).

DocumentSet est une nouvelle entité de la base de données, reliée aux tables **Witness**, **Serie** et **Work** par une relation many-to-many. Treatment est aussi une instance liée à un set et par son intermédiaire aux entités **Witness**, **Serie** et **Work**. Au moment de l'enregistrement du **DocumentSet**, une méthode va récupérer les id des **Witness**, **Serie** et **Work** qu'il contient dans un JSON. Ce JSON est parsé pour récupérer le nombre et les *id* de chacun des objets du **Set**, et à partir de ça constituer une liste de témoins, et les URLs de leurs images.

Une variété d'action peuvent être accomplies via un **Treatment** (export, détection de similarité, vectorisation, etc.). Côté utilisateur-ice : le traitement sera lancé via un formulaire modulaire pouvant s'étendre avec l'ajout de nouveaux modules. Une fois le traitement lancé, le workflow est le même. Ce qui change c'est la structure du JSON envoyé à l'API : c'est un dictionnaire de dictionnaire / et les données renvoyées à l'appli : c'est un ZIP avec un niveau d'arborescence en plus.

Pour ce faire, la création d'un nouvel endpoint capable de parser le JSON d'entrée est nécessaire. Ainsi, une seule requête sera envoyée et reçue, évitant ainsi l'envoi d'une requête par témoin, ce qui pourrait surcharger l'API.

Le premier endpoint servira pour les **RegionSet**. Du côté de l'application, une gestion de la présence ou de l'absence de fichiers SVGs sera intégrée, permettant ainsi une vectorisation partielle du document. Il sera également possible d'écraser ces fichiers SVGs en lançant une tâche de vectorisation sur l'ensemble du témoin, par exemple.

## Stockage

Appli : Les fichiers SVGs seront stockés dans le répertoire /médiafiles, dans un sous-dossier SVG. La réception des SVGs devra vérifier l'existence de ce dossier et le créer si besoin.

API : Les images téléchargées et les SVGs générés seront supprimés au bout de 7 jours grâce à une *Redis Cron*.

## Tests

Afin de garantir la qualité, la fiabilité et la robustesse de la fonctionnalité de vectorisation au sein de l'application, en assurant que chaque composant et l'intégration globale fonctionnent correctement, des tests doivent être effectués. Ils vérifieront que la communication avec l'API et l'inférence avec le modèle se font de manière optimisée et que les résultats sont conformes aux attentes.

Tests d'intégration :

- flux complet vers l'API et retour
  - conformité entre le nombre d'objets en requête et le nombre de fichiers SVGs reçus
  - conformité du format du JSON envoyé à l'API
  - les fichiers générés sont bien formés
- mesurer le temps de réponse de l'API pour mesurer
  - en environnement de développement
  - en production

## Axe 2 : Les résultats : visualisation et exploitation

À court terme, on envisage une visualisation des prédictions avec une faible interaction et sans enregistrement des résultats des actions.

### Échelle du **Witness**

Une fois l'inférence avec le modèle de vectorisation effectuée, apparaît un bouton pour l'overview des vectorisations.

On envisage la création de deux vues différentes, sur le modèle des visualisations des extractions.

1. *dumps* / page de thumbnails : présentation du diagramme et de la vectorisation côte à côte.
2. un tableau qui "contextualise" la vectorisation en affichant la page associée

Au click sur les vectorisations affichées sur les pages ci-dessus :

3. Visualiser l'image en JPG recouverte par le SVG. Interactions envisagées :

- zoom in and out
- choix d'ajouter ou enlever des layers (JPEG et SVG)
- Traitements sur les layers
  - toggle opacity
  - all lines in black

#### **Comment ?**

Utiliser JS pour modifier les classes CSS.

- Export
  - zip SVG + JPG pour ensuite pouvoir effectuer plus de traitements dans InkScape
  - export en JPEG en gardant les modifications

#### **Comment ?**

Utiliser JavaScript pour dessiner l'image sur un élément <canvas>, puis appliquer les traitements (opacité, couleur) et exporter le

canvas en image JPG. Le SVG modifié est transformé en un objet Blob, et une URL est créée pour ce Blob pour le téléchargement.

## Export des vectorisations

Sur les deux pages de visualisation des vectorisation, un bouton "export all" pourra déclencher l'exécution d'une *view* pour exporter à la fois les annotations et les SVGs du manifeste observé. Des checkbox disposées sous chaque vectorisation permettent d'exporter un échantillon.

## Prévision d'exploitation des vectorisations :

Post-processing à développer :

- recherche de similarités sur les résultats de la vectorisation
- éditions des résultats de la vectorisation

Futures features (idiomes) qui pourraient être extraits (implémentations futures) :

- élargissement du spectre des primitives détectées (ellipses et points), et ajout de la détection des couleurs et de l'épaisseur du trait.
- détection et transcription du texte et des labels
- *edge detection*
- tagging manuel et annotations SAS
- tagging automatique :
  - graduation
  - mono ou polychromatique

## Prévision concernant la plateforme d'édition :

À long terme, on envisage la création d'une véritable plateforme d'édition des SVGs pour la correction des prédictions du modèle , la création *manuelle* de SVGS à partir d'images raster, et différents types d'édition des diagrammes.

Les besoins :

Les Layers :

- Une même image => des existences parallèles => des layers ou représentations multiples :
  - Raster Image
  - SVG, avec encodage de l'information géométrique, de la couleur etc.
  - *Edge Detection* (tracés sur fond blanc)
  - Text et label detection (+ implémentation HTR) : transcription des textes et lettres entourant les diagrammes, et extraction des symboles (les planètes par exemple)
  - Des classifications automatiques et taggings manuels
- Pouvoir naviguer dans l'épaisseur d'une même image, en autorisant une bascule et un alignement des différents layers.
- Implémenter des fonctions de recherche avancées basées sur des métadonnées qui découlent du manuscrit témoin, issu des classifications/tagging au niveau du diagramme, ou issu de la fouille dans les formats *computable*.

scénario A : Édition diplomatique (fidèle au témoin)

#### Action sur

#### Besoins

*Edge Detection* (JPEG) **OU**  
SVG + labels

- Tracer une couche SVG sur l'image JPEG pour corriger les défauts de la numérisation OU utiliser le SVG et corriger les défauts de la transcription automatique.
- embarquer la donnée : certitude

scénario B : Transcription (correction des prédictions)

#### Action sur

#### Besoins

- SVG
- Labels

- modifier, rajouter ou supprimer des primitives
- modifier la couleur des primitives
- modifier la transcription du texte ou rajouter du texte

- recording : garder un record des changements effectués (système de versionning type git ?)

scénario C : Édition scientifique (reconstitution par le·a chercheur·se)

### Action sur

- Nouveau SVG

### Besoins

- On a un appareil critique (X témoins du même diagramme / type de diagramme) (cf. Annexe C)
- On dessine une nouvelle version de ce diagramme "from scratch", mais pour nous aider on a sûrement besoin de pouvoir basculer entre les témoins pour les placer "en transparence" dessous. Sans oublier la possibilité de rajouter des labels (texte).
- On veut relier automatiquement EI manuellement chaque élément signifiant (idiome) de ce diagramme à l'apparat critique. **Il est nécessaire de mettre en œuvre un système de calques.**
- On veut pouvoir organiser et observer l'apparat critique pour chaque idiome sémantique.
- On veut pouvoir avoir une interaction avec celui-ci : pouvoir switch d'une représentation à une autre, la placer en transparence du diagramme édité, etc.

scénario D : Édition corrigée (cohérence cis-à-vis de la proposition)

### Action sur

- Nouveau SVG

### Besoins

- Un type particulier d'édition scientifique
- Éventuellement embarquer plus de métadonnées concernant la proposition textuelle.



Trois grands blocs de l'interface d'édition scientifique :

- dessin
- collation
- consultation

Deux grandes orientations :

- collation manuelle
- fonctionnalité de recherche et collation automatique (si automatisation permettre la correction manuelle)

La fonction de recherche repose sur des scripts effectuant une recherche automatique de similarités dans la sélection de diagrammes à partir des primitives et des métadonnées des SVGs, en ciblant un motif. Il faudra alors statuer sur ce qui constitue le motif signifiant: décidant alors d'inclure ou non dans la recherche les couches de sens plus fines (la couleur et l'épaisseur) ou des métadonnées englobantes (classification automatique ou tagging). La recherche peut reposer sur le contenu encodé dans les fichiers SVG, grâce aux coordonnées, aux attributs couleur (code hexadécimal) et épaisseur du trait (les trois modulo un Delta à déterminer), grâce aux transcriptions du texte.

Cette solution présuppose la qualité des SVG générés automatiquement, et donc la possibilité de les corriger en amont. Il est donc crucial de disposer d'un outil d'édition fonctionnel avant d'envisager la collation et l'établissement de l'édition critique. Le·a chercheur·se devra en outre connaître son appareil critique afin de pouvoir le corriger, et l'interface autoriser cette correction manuelle.

[Ici une modélisation de l'interface](#) : voir annexe C pour des captures d'écran.

## Benchmark des solutions techniques

### 1. Gestion de la donnée

Un nouvel élément `EditedDiagram` pourrait être créé dans le modèle de données. L'édition serait :

- soit un fichier SVG (identifiant `ed_{x}`) stocké en backend sur le serveur.
- soit écrit en dur dans la base de données.

`EditedDiagram` est lié à un `RegionSet` (batch de diagrammes découpés issus de différents `Witnesses`).

Les EditedDiagrams seront liés à un utilisateur qui pourra ou non les rendre publics, et qui sera le seul avec les droits d'édition.

## Dessin

L'élément SVG est dessiné ou modifié dans l'interface d'édition. D'autres actions peuvent être effectuées via l'interface. Il est possible d'embarquer des métadonnées dans les sous-éléments de `<svg>` et aussi de rajouter du balisage pour traiter plusieurs éléments comme une seule unité sémantique (`<g/>` ou `<symbol/>`). Il est aussi possible d'associer un identifiant unique aux éléments ou groupes d'éléments.

## Collation

Un JSON par EditedDiagram, écrit en dur dans la base de données, pourrait faire correspondre à un idiome signifiant (que ce soit une primitive, une couleur etc.) un ensemble d'autres objets.

Il permet de trier les sources selon leur représentation ou non sur un élément sémantique du SVG.

## Consultation

Parser ce fichier permet de trier les sources entre *is\_present* ou pas + charger les régions (URLs IIIF) et leurs représentations. Une *sandbox* peut être créée pour effectuer des comparaisons.

### 2. Visualisation et actions basiques

Les développeurs du modèle (Syrine Kalelli) fournissent des fonctions de post-traitement des SVGs (bibliothèques PIL et Matplotlib) pour embarquer l'identifiant de l'image matricielle correspondante, ce qui sert de métadonnée et de superposer très simplement les deux entités dans le DOM.

Pour une interaction avec l'image vectorielle :

#### a. Intégration de l'élément graphique au HTML

	<a href="#"><code>&lt;canvas&gt;</code></a>	<a href="#"><code>&lt;svg&gt;</code></a>
Caract.	Description : Un élément pour dessiner des graphiques avec JavaScript, utilisant une méthode de dessin par pixels.	Description : Un langage de balisage pour décrire des graphiques vectoriels. Mode de Rendu : Vectoriel, ce qui signifie que les graphiques sont

	<b>Mode de Rendu</b> : Bitmap, ce qui signifie que les graphiques sont rendus pixel par pixel. <b>Utilisation</b> : Principalement pour les graphiques animés, les jeux, et tout ce qui nécessite un redessin fréquent.	décrits en utilisant des formes géométriques. <b>Utilisation</b> : Idéal pour les images statiques ou celles qui nécessitent une manipulation DOM, comme les diagrammes ou les graphiques interactifs.
Limites	Complexité : Nécessite une gestion manuelle de beaucoup de détails (redessin, gestion de l'état des objets, etc.).	Moins performant que <code>&lt;canvas&gt;</code> .

Utilisation de `<svg>` pour la manipulation dans le DOM et réécriture dans un `<canvas>` pour des manipulations plus complexes comme l'export en JPEG.

## b. scripting du SVG avec du JavaScript et des classes CSS

<https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial>

### 3. Éditeur

**Le langage** JavaScript semble être adapté à l'implémentation d'un éditeur dans l'application EiDA :

- Environnement d'exécution : langage natif du web et performant pour celui-ci
- Facilité et Rapidité de Développement : L'écosystème de JavaScript offre des bibliothèques comme D3.js, Snap.svg, et des frameworks comme React, Vue ou Svelte pour une manipulation éléments du DOM.
- Web-Oriented : Les bibliothèques JavaScript sont conçues pour le web, facilitant la création d'interfaces interactives et la manipulation DOM.

### Propositions de librairies :

Le besoin principal : performance pour gérer des contenus multimédia.

- [Fabric.js](#) : support pour la sélection, la modification et la transformation d'objets. Bien que Fabric.js soit basé sur Canvas, il peut être utilisé pour la manipulation d'objets graphiques.
- [SVG.js](#) : Facile à utiliser avec une API claire et concise. Ajouter SVG.js à votre projet en incluant le script dans les templates Django. Utiliser les vues Django pour servir les données SVG et interagir avec l'éditeur via AJAX.

- React + [React Konva](#) : permet de manipuler des graphiques vectoriels en utilisant l'architecture de composants de React. Bénéficie de la gestion de l'état et de la réactivité de React. Utiliser Django pour servir les templates de base et gérer les composants React via un bundle JavaScript. React peut interagir avec l'API Django pour les opérations CRUD sur les SVGs.
- Svelt : créer des composants réactifs et réutilisables. La syntaxe est simple et lisible côté développeur et il offre de très bonnes performances côté utilisateur·ice, en particulier dans la manipulation des éléments multimédia. Il se distingue par son approche pour le rendu des composants.

### **Compilation au Moment de la Construction**

Il s'agit de la caractéristique principale de Svelte. Au lieu de faire tout le travail dans le navigateur, Svelte déplace une grande partie de la compilation côté serveur. Comment ?

- **Développement** : Écriture d'un composant en utilisant la syntaxe Svelte, qui ressemble beaucoup à du HTML, CSS et JavaScript combinés.
- **Compilation** : Le compilateur Svelte prend ce code source et le transforme en JavaScript vanilla très optimisé.
- **JavaScript Vanilla** : Le code résultant est un JavaScript pur sans la surcharge des bibliothèques ou framework, ce qui signifie que le code est plus rapide.

### **Réactivité Automatique**

Svelte propose un système de réactivité automatique. Les variables déclarées dans les composants deviennent réactives par défaut :

- En déclarant une variable réactive avec la syntaxe spéciale **\$**, Svelte réactive automatiquement l'interface utilisateur chaque fois que la variable change.
- Utilisation des **stores** pour partager des états entre composants (en effet ceux-ci peuvent être profondément imbriqués, et l'utilisation de stores évite de faire descendre ou remonter l'état des variables entre les divers composants).

## Axe 3 : Modularisation de l'application

**Objectif général :** Développer une architecture modulaire pour l'application permet une gestion flexible et évolutive des différentes fonctionnalités. La modularisation facilitera la maintenance, l'ajout de nouvelles fonctionnalités et l'adaptation aux besoins spécifiques des utilisateur·ices.

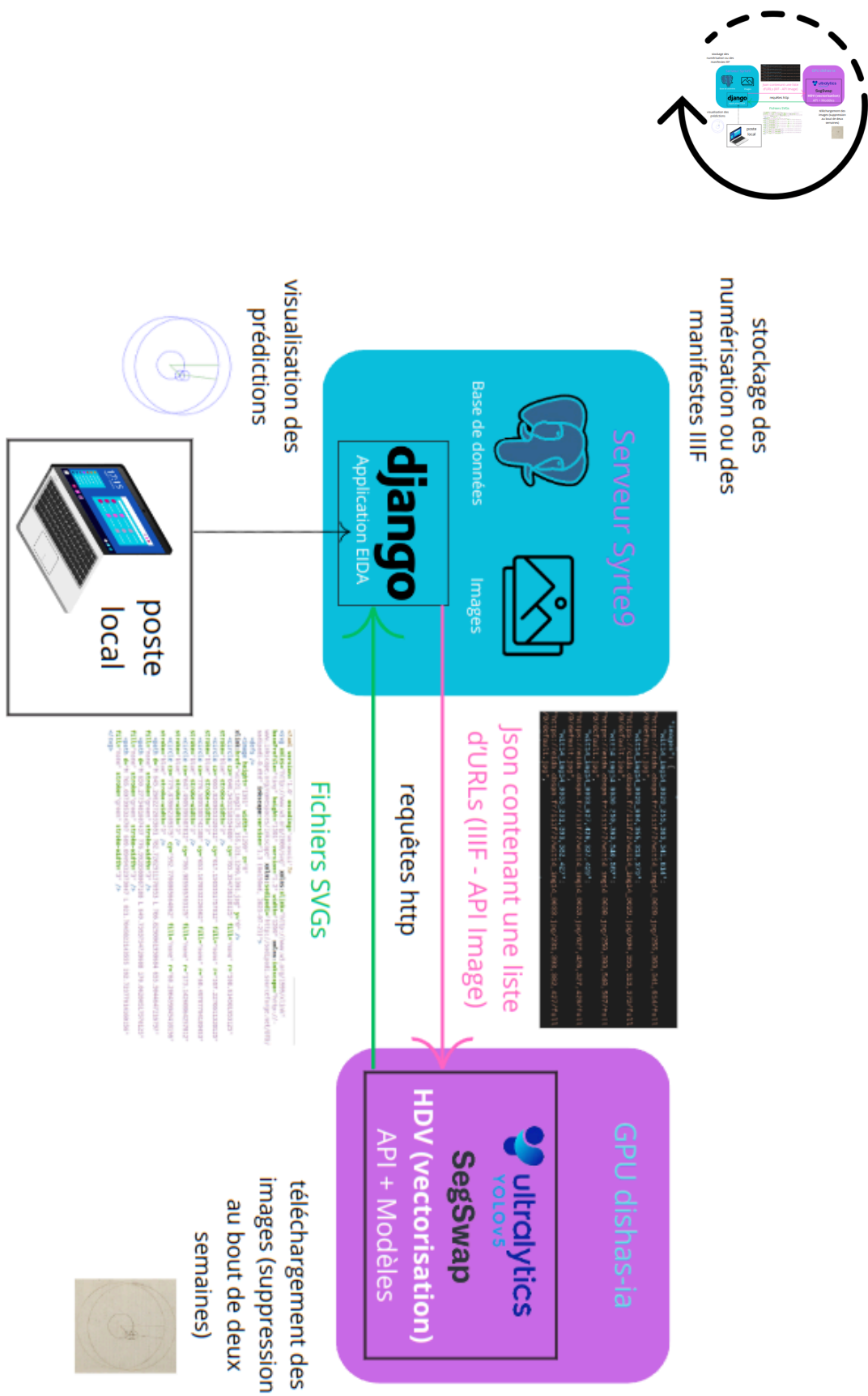
- Flexibilité et personnalisation : permet d'ajouter ou non les modules lors de l'installation, selon les besoins de l'utilisateur·ice. Il peut configurer l'application selon des besoins spécifiques.
- Évolutivité : une architecture modulaire standardisée et normalisée l'ajout de nouvelles fonctionnalités, sans perturber les tâches déjà présentes.
- Maintenance : simplifie le débogage

**Description des besoins :** Pouvoir proposer une utilisation du code à des projets de recherches et des équipes diverses, ayant des corpus différents de celui d'EiDA, et des besoins numériques sur une échelle allant de la simple agrégation des données de la recherche à l'ajout de modules personnalisés. Les efforts tendent vers la facilitation de l'installation et du déploiement, et vers une amélioration graphique de l'interface graphique.

**Mise en œuvre concernant la vectorisation :** Au lieu d'un seul dossier webapp/ on aura donc autant de dossiers que de tâches implémentées (extraction/, similarity/ et vectorization/ pour le moment. Chaque module dispose de ces fichiers utilitaires (utils/), des fichiers principaux (task.py et views.py), et d'un fichier de configuration pour déterminer les paramètres spécifiques à l'implémentation l'implémentation de chaque module. Un fichier de configuration générale gère les paramètres de haut-niveau :

- nom de l'application (APP\_NAME)
- installation des modules souhaités (INSTALLED\_APP)
- ontologie
- liste des types de witness
- modalités de recherche

## ANNEXE A : Fonctionnement du module Vectorisation



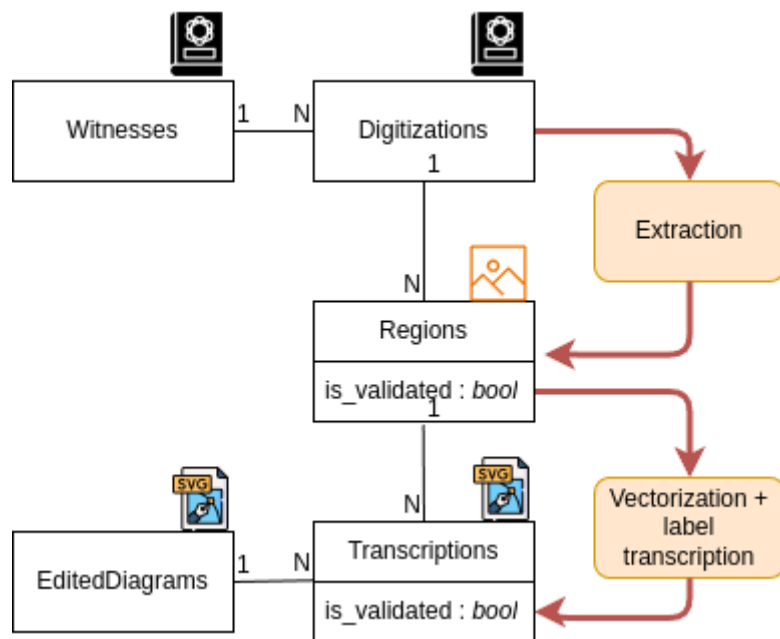
ANNEXE B : Pour ou contre l'implémentation d'une table `GraphicalElement` dans le modèle de données. Ce tableau est extrait du cahier des charges rédigé par Ségolène Albouy en accord avec l'équipe d'ingénierie VHS/EiDA.

	Avec Graphical Element	Sans Graphical Element
+	<ul style="list-style-type: none"> <li>→ Respecte la logique des autres unités de description (Witness / Series)</li> <li>→ Plus facile de sélectionner différents enregistrements dans un panier</li> <li>→ Plus facile d'effectuer des recherche avec la même interface que pour Witness / Serie</li> <li>→ Effectue un tri dans le bruit de toutes les SAS-anno</li> </ul>	<ul style="list-style-type: none"> <li>→ Plus simple à mettre en oeuvre</li> <li>→ Pas de redondances</li> <li>→ Indépendant du mode d'extraction de la boundingbox</li> <li>→ Naturel de créer autant de zones dans une image que nécessaire (pas besoin de niveaux enchevêtrés)</li> </ul>
-	<ul style="list-style-type: none"> <li>→ Statut doublon/confusant avec SAS-anno</li> <li>→ Modes de création opposés (via une SAS-anno ou par formulaire)</li> <li>→ Risque d'avoir des centaines de milliers d'enregistrements dans la base</li> <li>→ Plus difficile à lier naturellement à un Witness</li> </ul>	<ul style="list-style-type: none"> <li>→ Objet hétérogène à traiter dans la gestion de panier</li> <li>→ Plus difficile à enrichir à l'aide de métadonnées</li> <li>→ Plus dépendant de SAS ce qui pourrait constituer une dette logicielle dans le futur</li> </ul>
	<b>~ 2 semaines temps plein</b>	<b>~ 1 semaine temps plein</b>
	Importation d'image unique	
	Import dans un formulaire à part ⇒ ajout d'une digitization + lien (opt) à un Witness	Import via un Witness ⇒ ajout d'une digitization (partielle)
	Ajout au panier	
	Import cohérent avec les autres entités Witness et Series	Garder en mémoire dans le panier une référence (img_page_coord)
	Taguer des Images	

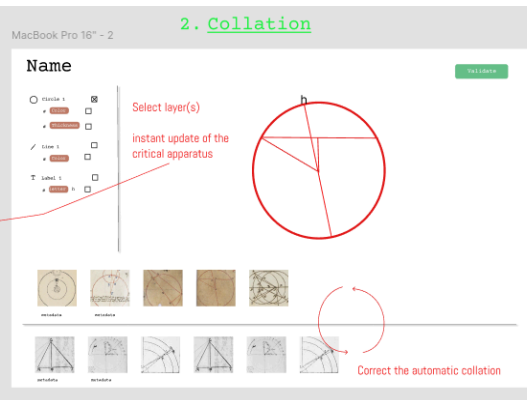
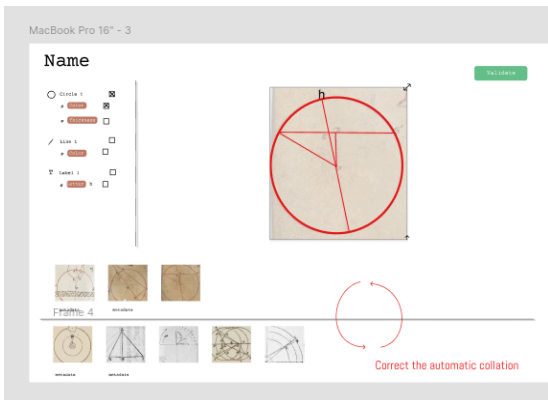
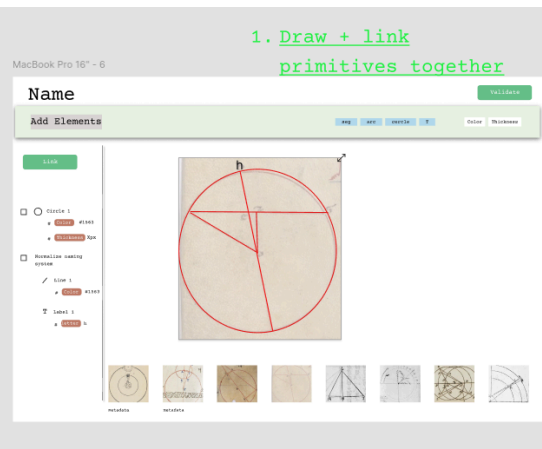
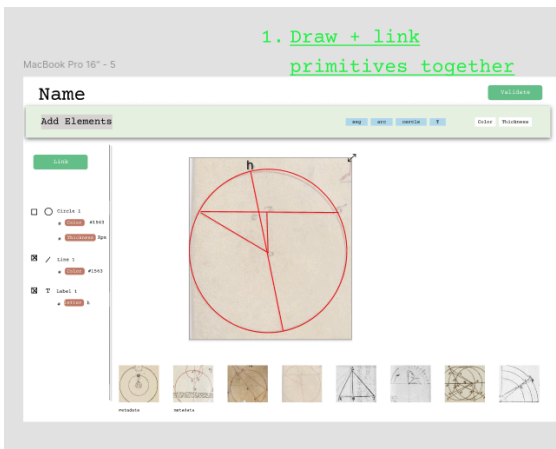
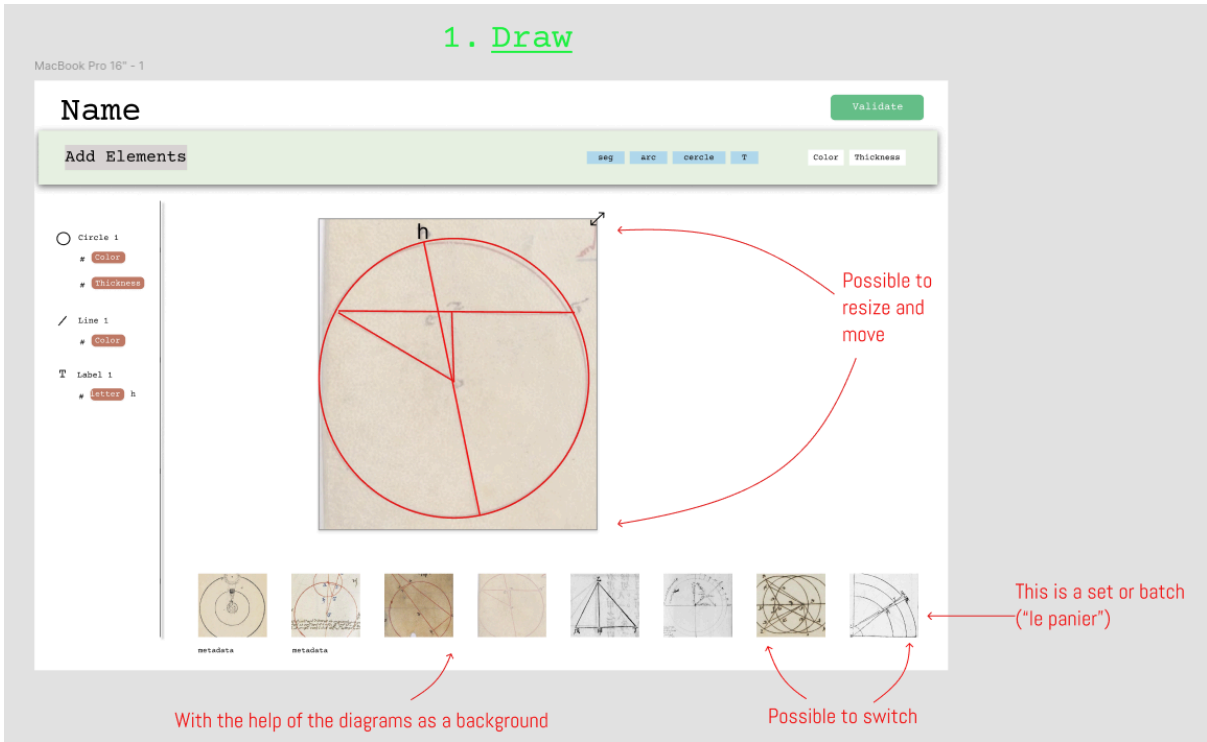
	En liant le modèle GraphicalElement avec le Tag	Soit via Mirador
	Création d'une zone d'image	
	<p>→ Formulaire de GraphicalElement : l'utilisateur importe une image unique et la décrit succinctement, la lie à un Witness (optionnel) &gt; création d'une Digitization (+ création d'une SAS-anno ?)</p> <p>→ Sélection de zones d'image SAS pour lancer une traitement (e.g. vectorisation) &gt; création d'un enregistrement de GraphicalElement &gt; création d'une Digitization au format image (re-télécharger l'image IIIF ?) : Interface à déterminer</p>	Les zones d'images ne sont que créées via une SAS-anno



## ANNEXE C : Modèle logique avec intégration des diagrammes édités



ANNEXE D : Modélisation de l'interface d'édition



### 3. Vizualise Critical Apparatus

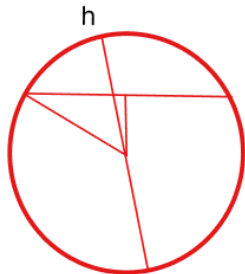
MacBook Pro 16" - 4

Name

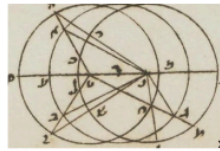
Same idea : select shapes and layers, and instant update of the critical apparatus

different views of each witness

- ☒ Circle 1
  - # Color ☐
  - # Thickness ☐
- ☐ Line 1
  - # Color ☐
- ☐ Label 1
  - # Letter h ☐



SANDBOX => resize, move and compare



close when you are done

metadata

slide

Raster image

SVG

Edge detection

Label Trascript.

view critical apparatus

select

Order by ▾

Witness  
Time  
Place



metadata



metadata

