Voyager User Documentation

UE ProCom FISEA3
Datamining of Ansible Code

MARGUINAUD Estelle, AAGOUR Fouad, DIARISSO Abdoulaye, GRENET Maxime, BLEVIN Sarah

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Table of Contents

---

1. Introduction

Voyager is a powerful tool designed to collect datasets of Ansible roles from Ansible Galaxy and analyze structural changes between role versions. By leveraging APIs and metadata extraction, Voyager provides researchers and developers with a robust framework to study version histories and repository structures. More details about Voyager can be found in the corresponding research article and R.Opedebeeck’s git repository.

https://cris.vub.be/ws/portalfiles/portal/76353754/vub_tr_soft_21_03.pdf

https://github.com/ROpdebee/Voyager.git

Voyager++ builds upon the original Voyager tool, addressing obsolescence and dependency issues while introducing significant new capabilities. Notably, Voyager++ includes a dedicated datamining stage, enabling advanced analysis for identifying trends and insights in datasets. This updated version is tailored for student projects and research contributions, providing a flexible pipeline for data collection and analysis. Its updated framework ensures compatibility with modern systems and simplifies the workflow for users. It was designed for research project For-CoaLa, which aims to formally certify Ansible based on a subset of Ansible code which includes the features most used by the community of users.

2. System Requirements

- **Operating System:** Compatible with systems running Python 3.8 or higher.
- **Hardware Requirements:** No specific hardware requirements, but sufficient processing power and memory are recommended if the user intends to collect the entirety of the Ansible Galaxy ldataset.
- **Dependencies:**
    - Python >= 3.8
    - Poetry

3. Installation

- **Downloading Voyager:** Clone the Voyager++ repository or obtain the source code.

  https://github.com/NotArobase/Voyager.git

- **Installation Steps:**
    1. Navigate to the Voyager++ directory:

       ```
       cd /path/to/voyager++
       ```

    2. Install dependencies using Poetry:

       ```
       poetry install
       ```

- **Verification:** Ensure the installation is successful by running:

  ```
  poetry run -- python main.py –help
  ```

  This should display all general options and commands available in the pipeline, as described in section 6.3 Command Options.

4. Getting Started

- **First Launch:** Once installed, Voyager++ can be executed using Poetry to access its primary functionality.
- **Initial Setup:** No special configuration is required beyond installing dependencies.
- **Sample Walkthrough:** Run the following command to scrape data from Ansible Galaxy:

  ```
  poetry run -- python main.py --progress --report --dataset
  my_data galaxy-scrape
  ```

This creates a dataset named `my_data` and begins harvesting data while showing progress and generating a report.

---

5. Project Structure

The Voyager++ codebase is organized into several directories and modules that enable flexibility and maintainability. Below is an overview of the structure, highlighting key components and their purpose.

Root Directory

- `main.py`: The entry point for the application, where the command-line interface (CLI) is defined. This file parses user commands and triggers the pipeline stages.
- `config.py`: The class MainConfig serves as the global configuration for the entire tool. It extends Config (standard click class). Specialized configurations (classes ExtractRoleMetadataConfig, CloneConfig, ExtractStructuralModelsConfig) extend MainConfig to add options for specific tasks, such as cloning repositories or extracting metadata.
- `README.md`: A markdown file providing an overview of the project, installation instructions, and usage examples.
- `pyproject.toml`: The configuration file for managing Python project dependencies and settings. It is used by Poetry to define the package and handle dependencies.
- `poetry.lock`: The lock file generated by Poetry to ensure consistent installation of dependencies across different environments.
- `mypy.ini`: The configuration file for `mypy`, a static type checker for Python. It defines the settings for type checking and the directories to be checked.

`models/`

Contains the data models used by Voyager++, defining the structure of the data that will be processed at various stages. These models are implemented using Python classes with support for serialization and deserialization.

`pipeline/`

This directory holds the stages of the Voyager++ pipeline, each encapsulated as a class that defines specific functionality.

- `base.py`: Contains the abstract base class `Stage`, which is subclassed by each specific stage in the pipeline. The base class defines the general interface for running stages and reporting results.
- `collect/`: Contains the stages responsible for collecting data from external sources.
  1. `galaxy_scrape.py`: Implements the functionality for scraping data from Ansible Galaxy and collecting role information.

2. custom_scrape : Implements the functionality for scraping data from Ansible Galaxy and collecting role information ; but using a custom schema to filter out unnecessary data.
3. `clone.py`: Handles the cloning of Git repositories for the discovered roles, as part of the repository analysis pipeline.

- `extract/`: Includes the stages related to data extraction from the sources.
- `datamine/`: Holds the `DatamineStage` stage, which dynamically loads and executes an external datamine algorithm script of the user's choosing. Also holds a set of « default » datamining scripts which have yielded relevant results in our case.

data/

This is the directory where the results of the Voyager++ pipeline stages are stored. Each stage generates output in the form of files, directories, or reports.

- `GalaxyScrape/`: Contains the scraped data from Ansible Galaxy, including a summary file `index.yaml` and individual role data files.
- `GalaxyMetadata/`: Stores extracted metadata, such as `Roles.yaml` and `Repositories.yaml`.
- `RepositoryMetadata/`: Stores git metadata extracted from the cloned repositories.
- `StructuralModels/`: Contains YAML files for each role, detailing the structural model for each version of a role.
- `Datamine/`: The output directory for the `DatamineStage` stage, where the analysis results are stored, as defined in the external algorithm script used when running the stage.

tests/

Contains unit tests for various modules and pipeline stages, ensuring the correctness of the functionality. **Obsolete.**

utils/

Utility functions and helper modules that are used across various stages and components.

- `logging.py`: A centralized logging configuration for Voyager++, ensuring consistent logging across all stages.
- `file_operations.py`: Helper functions for reading and writing files, handling directories, and managing file paths.

services/

Contains services that interact with external systems or APIs, such as the Ansible Galaxy API.

voyager_api/

Contains the code of the rest API server for Voyager.

---

6. Usage

6.1 Basic Operations

- **Scraping Data:**

  ```
  python main.py --progress --report --dataset my_data galaxy-
  scrape
  ```

  Harvest data from Ansible Galaxy into a dataset named `my_data`.

  Output : A new folder my_data/GalaxyScrape is created, with an index.yaml containig the list of all the collected roles, and multiple roles_x.json files containing the collected data about each role. The number of roles per file is set to 250 by default. The fomat of the data is defined in models/galaxy_schema.py .

- **Scraping Data (Custom) :**

  ```
  python main.py --progress --report --dataset my_data custom-
  scrape -–schema my_schema_path
  ```

  Harvest data from Ansible Galaxy into a dataset named `my_data`, using the custom json schema found at path `my_schema_path`

  Output : A new folder my_data/CustomScrape is created, with an index.yaml containig the list of all the collected roles, and multiple roles_x.json files containing the collected data about each role. The number of roles per file is set to 250 by default. The fomat of the data is defined in my_schema_path. More details in section 7- Customization > Easy Customization functionalities > Custom Scrape.

- **Extracting Metadata (requires : Scraping Data):**

  ```
  python main.py --dataset my_data extract-role-metadata
  ```

  Extract metadata from harvested API pages into the Galaxy metadata schema.

  Output : A new folder my_data/GalaxyMetadata is created, with and index.yaml acting as a summary of the contents of the folder, and the Roles.yaml and Repositories.yaml files, containing the extracted metadata of all the scraped roles and corresponding git repositories (if found). The data is extracted based on the models defined in /models.

6.2 Advanced Operations

- **Cloning Repositories (requires : Extracting Metadata):**

```
python main.py --report --dataset my_data clone
```

Clone repositories discovered in the metadata (if found via the Git API).

Output : A new folder my_data/Repositories is created. Inside, one subfolder per repository is also created. Each subfolder will contain the code of the role as posted by the owner of the repository. It usually contains:

`main.yml`: Ansible role definitions or configurations.

Additional directories and files, such as:

- `tasks/`
- `templates/`
- `README.md`

- **Extracting Git Metadata (requires : Cloning Repositories):**

```
python main.py --dataset my_data extract-git-metadata
```

Extract git repository metadata, including commits, branches, and tags.

Output: A new folder my_data/RepositoryMetadata is created. It contains: an index.yaml file which is the list of the repositories that were analyzed ; and one subfolder per user. In this subfolder, each repository has its own yaml file containing the metadata of the repo, extracted based on the models defined in /models.

- **Extracting Structural Models (requires : Cloning Repositories, Extract Git Metadata):**

```
python main.py --dataset my_data extract-structural-models
```

Extract structural models for each git tag matching semantic versioning.

Output : A new folder my_data/StructuralModels is created. Inside, one yaml file per role is created, containing the structural model of each role, as defined in /models.

- **Datamining Roles Stage (requires : Extracting Structural Models):**

```
python main.py --dataset my_data datamine-stage –-path
my_script_path --options '{"param1": "value1", "param2": 42}'
```

Analyze the dataset to identify trends and insights based on the structural models extracted from the previous stage. The analysis technique is defined in an external script my_script_name written by the user and stored in their device at my_script_path. More details in section 7- Customization > Easy Customization functionalities > Datamine.

Output: A new folder my_data/Datamine/my_script_name is created. It will contain the output of the analysis, as defined in the external script written by the user. More details in section 7- Customization > Easy Customization functionalities > Datamine.

- **Distilling Structural Diffs:**

  ```
  python main.py --dataset my_data extract-structural-diffs
  ```

  Extract differences between structural model versions.

  Output : A new folder my_data/StructuralRoleEvolution is created. It contains a n index.yaml file which is the list of the roles that were analyzed, and one yaml file per role, highlighting the changes in role structure between versions.

6.3 Command Options

- **Global Options:**
  - `--delete / --no-delete`: Delete the output directory before running.
  - `--force / --no-force`: Force regeneration of cached results.
  - `--output DIRECTORY`: Specify the output directory.
  - `--dataset TEXT`: Define the dataset name (required).
  - `--progress / --no-progress`: Display task progress.
  - `--report / --no-report`: Generate a task completion report.
  - `--help`: Display the help text.
- **Stage-Specific Option:**

  Scrape

  - `--max-roles INTEGER`: Limit the number of roles collected during the `galaxy-scrape` stage. Intended for testing of the Voyager++ tool.
  - `-- schema PATH` (for custom scrape only) : Specifies the file path to the custom schema that will be used to filter out data from the collect stage.

  Datamine

  - `--path PATH` : Specifies the file path to the algorithm script that should be executed during the datamining stage. (required)

    - `--option Optional[Dict[str, Any]]` : This option allows you to pass additional parameters or configuration settings to the external algorithm script.

- **Available Commands:**
    - `clone`: Clone the repositories for discovered roles.
    - `datamine-stage`: Process the datamining stage
    - `extract-git-metadata`: Extract git repository versions.
    - `extract-role-metadata`: Extract metadata from collected roles.
    - `extract-structural-diffs`: Extract structural differences between role versions.
    - `extract-structural-models`: Extract metadata from collected roles.
    - `galaxy-scrape`: Discover roles to populate the dataset.
    - `custom-scrape` : Discover roles to populate the dataset, using custom schema.

6.4 Shortcuts and Tips

- Combine commands for efficiency, such as:

  ```
  python main.py --report --progress --dataset my_data datamining
  extract-structural-diffs
  ```

  Execute the datamining and diff-extraction stages in one command.

  Most stages are « linked », as in one is needed for the next to run, and calling only the last one will result all its dependencies also running if they are not already there (ie if the corresponding folders are missing).

- Use the --help command during a stage to display stage-specific help text, for example :

  ```
  python main.py --dataset my_data galaxy-scrape --help
  ```

---

7. Customization

Voyager++ is built on a flexible pipeline and modular models, making it highly customizable for a variety of use cases. We have also implemented a few modularity functionalities for easier customization.

This section is divided into two parts : first, we will explain the functionalities we have added, designed for an easy start into Voyager++ customization.

Then ; we will explain how to modify Voyager++ code, for users who want to go more in depth.

Easy- Customization functionalities :

Custom Scrape : Users can choose to select which attributes to exclude from the data collection. This is meant as an easy way to filter out unnecessary information about roles, depending on the uses intended for the data, in order to make it mor human read-friendly and to save storage space.

To do this, users can write their own version of the default schema found at models/galaxy_schema.py. This means writing a simple json containing all the attributes available from the Galaxy API as keys, and booleans as values : true if the attribute is to be kept ; false otherwise.

Exemple of schema.json :

```
{
  "commit": false,
  "commit_message": false,
  "created": false,
  "description": true,
  "download_count": false,
  "github_branch": false,
  "github_repo": false,
  "github_user": false,
  "id": true,
  "imported": false,
  "modified": false,
  "name": true,
  "summary_fields": {
    "dependencies": false,
    "namespace": {
      "avatar_url": false,
      "id": false,
      "name": false
    },
    "provider_namespace": {
      "id": false,
      "name": false,
      "pulp_href": false
    },
    "repository": {
      "name": false,
      "original_name": false
    },
    "tags": false,
    "versions": false
  },
  "upstream_id": false,
  "username": false
}
```

This json can be stored anywhere on th user’s device, and be used during the custom stage by using the following command :

```
python main.py --progress --report --dataset my_data custom-
scrape –-schema my_schema_path
```

Only the attributes set to true will appear in the output json in folder my_data/CustomStage. In this example, only description, id and name are collected.

Note : this does not reduce the time required for the collection stage. The Ansible Galaxy API is not modular and we are forced to collect all th attributes at first ; the filter is only applied at the moment where we write the information in the json file.

Custom Datamining Stage : Users can run their own datamining techniques by simply calling their external, Voyager-independant scripts during the datamine stage, when running the following command :

```
python main.py my_dataset datamine-stage --options '{"param1": "value1",
"param2": 42}' --path my_script_path
```

In order to do this, the external script must contain the following functions :

<u>Algo function</u>

The script must define a function named `algo` that performs the actual datamining task. This function is responsible for receiving the configuration, processing data, and returning the results.

```
def algo(config, roles_dir_name: str, options: Optional[Dict[str, Any]] = None):
# Your algorithm implementation here
return result
```

Parameters:

- `config`: The configuration object containing settings passed from the `DatamineConfig` class.
- `roles_dir_name`: The name of the directory where role-related files are stored (e.g., "StructuralModels").
- `options` (optional): Any additional options passed to the script, provided as a dictionary. This can be used to customize how the algorithm behaves.

<u>Store_results function</u>

If your algorithm script involves saving or visualizing results (e.g., generating plots, storing data), you can define an optional function named `store_results`. You can choose to store your data in any format you want to. This function will be automatically called if it exists in the script. The output will be saved in folder my_data/Datamine/filename.

Parameters:

- `results`: The output of the `algo` function (i.e., the results returned by the datamining algorithm).
- `config`: The configuration object, which can be used for file paths and other settings.

- `filename`: The name of the output file or directory where results should be saved.

The script should ideally be Voyager-independant and contain all the logic necessary to run the datamine stage.

The external script can be stored anywhere on the user’s device, with my_script_path being the absolute path to the .py file. The **`--options`** argument is used to pass additional configuration options to the external datamining algorithm script if needed, for example to specify the input data.

Some basic datamining scripts will be available in Voyager++. They can be found in pathToVoyager/pipeline/datamine.

Change Voyager code :

- **Pipeline Flexibility:** Users can adjust, extend, or redefine pipeline stages to tailor the tool for specific research goals. Each stage, implemented as a class, defines its own inputs, outputs, and behavior. Adding a custom stage requires subclassing the `Stage` base class, using a configuration (MainConfig or otherwise, see below) and implementing required methods like `run` and `report_results`.
- **Modular Models:** Data models in Voyager++ are implemented using the `Model` class, which supports serialization and deserialization. This modular approach allows users to:
  - Integrate new data types seamlessly.
  - Implement custom hooks for data transformation using `cattr`.
- **Examples of Customization:**
  - **Adding a New Stage:** Create a new stage by subclassing `Stage`, defining dependencies in `poetry if needed`, and implementing core methods:

```
class CustomStage(Stage[ResultType, ConfigType]):

    dataset_dir_name = …

    def run(self, **kwargs):
        # Custom logic for the stage

    def report_results(self, results: ResultMap):
        # Reporting logic
```

  - **Customizing Data Models:** Add or extend attributes to existing data models and add a dump method if needed:

```
class ExtendedModel(Model):
```

```
                 new_field: str
                @property
                 def id(self) -> str:
                     return self.new_field
                def dump(self, directory: Path) -> Path: #Dump logic
```

- **Extending Configuration Options:** Users can modify the `MainConfig` class to add new global or stage-specific configuration options. These options are parsed automatically from the CLI.

---

8. Voyager API

API Overview

Voyager++ exposes a powerful API to interact with the various stages of the pipeline programmatically. You can use HTTP requests to launch different pipeline stages such as Galaxy Scrape, Custom Scrape, and Datamine, passing parameters that control the execution. The API supports both `GET` and `POST` methods, making it versatile for automation.

Installation

To use the Voyager++ API, you'll need to install the necessary dependencies. Run the following command:

```
pip install fastapi uvicorn
```

Launch the Server

Navigate to the Voyager++ project directory and start the server with `uvicorn`:

```
cd path/to/Voyager
uvicorn voyager_api.main:app --host 0.0.0.0 --port 8000 --reload
```

This will start the API server, and you can now make requests to it via `localhost:8000`.

Example API Requests

Here are some example `curl` commands to interact with the API. You can modify the parameters as needed to suit different datasets, options, or stages. You should also replace the localhost by the ip of the machine you wish to reach if you are not doing local testing.

Launch Galaxy Scrape Stage

To initiate the Galaxy Scrape stage, use the following `POST` request. You can also pass parameters such as the dataset name and options like `delete` and `max-roles`.

```
curl -X POST "http://localhost:8000/launch/galaxy-scrape"
```

To specify a dataset and additional options:

```
curl -X POST "http://localhost:8000/launch/galaxy-scrape?
dataset=your_dataset_name"
curl -X POST "http://localhost:8000/launch/galaxy-scrape?
dataset=your_dataset_name&delete=true&max-roles=100"
```

Launch Custom Scrape Stage

For a custom scrape stage with a specific dataset and schema, use the following:

```
curl -X POST "http://localhost:8000/launch/custom-scrape?
dataset=your_dataset_name&delete=false&max-
roles=5&schema=your_schema.json"
```

Launch Datamine Stage

To trigger the Datamine stage, you can use this `POST` request, where you provide the dataset and script path:

```
curl -X POST "http://localhost:8000/launch/datamine-stage?
dataset=your_dataset_name&delete=false&script_path=/path/to/your/
script.py"
```

You can also pass additional options, such as `num_modules`, as a URL-encoded JSON string in the `options` field:

```
curl -X POST "http://localhost:8000/launch/datamine-stage?
dataset=your_dataset_name&delete=true&script_path=/path/to/your/
script.py&options=%7B%22num_modules%22%3A11%7D"
```

Parameters

- **`dataset`**: The name of the dataset you are processing (e.g., `your_dataset_name`).
- **`delete`**: A boolean flag (`true` or `false`) that determines whether to delete existing output before starting the stage.
- **`max-roles`**: (Optional) Limits the number of roles to scrape (applies to scrape stages).
- **`schema`**: The path to the schema file (only needed for custom scrape).
- **`script_path`**: Path to the script that should be executed (used in the datamine stage).
- **`options`**: (Optional) A URL-encoded JSON string representing additional options. For example, `{"num_modules": 11}`.

API Response

The API returns standard HTTP responses. A successful request returns a 200 status code, while errors will result in a 400 or 500 status code. The body of the response contains additional details, such as the status of the operation and any error messages.

---

9. Troubleshooting

- **Common Issues:**
  - *Issue:* Poetry not recognized as a command.
    - *Solution:* Ensure Poetry is installed and added to the system PATH.
  - *Issue:* Errors during metadata extraction.
    - *Solution:* Check if the dataset directory is properly created
- **Error Codes:** Refer to error messages in the CLI for detailed guidance.

---

10. FAQs

- **Q:** Can I use Voyager without Poetry?
  - **A:** Voyager is optimized for use with Poetry, but manual dependency management is possible. One such version of Voyager without poetry can be found here :

    https://github.com/SarahBlevin/MyVoyager.git

    This version uses a bash to download all dependencies on the user's device or VM (using a python venv is recommended).
- **Q:** Does Voyager support Windows?
  - **A:** Yes, provided Python and Poetry are correctly installed.

---

11. Future features

- A frontend for the Voyager++ tool, using the existing Voyager API. The idea would be for the users to be able to start Voyager++ stages on a distant VM from the frontend, and to be able to check the results from this frontend also. This way ; the brunt of the work would be done on a dedicated VM, while the users would still have access to the results from their own machine.

- Containerization of the Voyager++ tool, which is monolithic in concept at the moment.