UNIVERSITÀ DI MODENA E REGGIO EMILIA

Dipartimento di Ingegneria "Enzo Ferrari"

Master's Degree in Computer Engineering
Data Engineering and Analytics

Enhancing Financial Time Series Analysis

Design and Implementation of a Web Application for Efficient Machine Learning Dataset Generation

Supervisor:
prof. Francesco Guerra

External supervisor:
Roberto Landi - Axyon AI

Candidate:
Vittorio Nutricato

Academic Year 2022-2023

To my family, my friends
and those who have been close to me

“Certo, chi nasce incatenato non sa che cosa è la libertà,
ma anch’egli sa cos’è il coraggio.
Un coraggio che tu nemmeno puoi immaginare.
Il coraggio di portare ogni giorno un carico più pesante senza curvare le spalle,
il coraggio di continuare a vivere per sé, per chi si ama.”

Lo Scudo di Talos, Valerio Massimo Manfredi

Acknowledgements

I am deeply indebted to Professor Francesco Guerra for his guidance and support throughout my academic journey. He's been a source of valuable advice, patiently steering me through the complexities of my studies, even when the path ahead seemed shrouded in uncertainty.

My heartfelt gratitude also extends to the Axyon AI team, who embraced me with open arms and welcomed me into their vibrant community. Roberto Landi, my Axyon AI advisor, has been an invaluable mentor, his constant support and insightful guidance instrumental in shaping my professional growth. I would also like to mention Andrea Bisi and Davide Brembilla from Axyon AI, thanking them for their patience and continuous commitment in advancing this project with me.

In conclusion, I would like to express my sincere appreciation to several individuals who have made an important contribution to my personal and academic adventure. For this reason, I dedicate a heartfelt embrace to my entire family and friends, who have always been by my side. As I reflect on my journey, I am overwhelmed with gratitude for the countless individuals who have enriched my personal and academic life. I am forever grateful for the unwavering support of those who have touched my life. Their contributions have been instrumental in shaping my journey.

Summary

The development of the web application "Talos" is a new project by Axyon AI company with the main goal of simplifying the dataset generation process in "Datasmith" by leveraging a user-friendly Graphical User Interface (GUI). Datasmith is the Axyon AI project aimed at creating datasets useful for both model training and inference and internal company experiments. Moreover, Talos will be used to track the main Datasmith activities, such as the historical and live ingestion of Feature Groups on the AWS SageMaker Feature Store directly from the web app panel. To generate a dataset, it is necessary to know several properties:

- The investable universe we are considering, identified by the "Feed code"
- The types of included Instruments
- The Features to be included in the dataset
- The contextual Features for the dataset
- The time period the dataset should cover
- The target (for training datasets)

All the listed information and properties describing the dataset are included in a YAML file that contains all the Feature Groups with their corresponding Features present in the Feature Store. This YAML file represents the dataset "Genome" through which the actual dataset can be generated once ready.
Generating the dataset Genome is currently a slow, error-prone, and semi-manual operation due to the nature of the YAML file, which contains a long list of Features and Instruments. Talos aims to expedite and strengthen the Genome generation process, providing Datasmith users with a tool to monitor all major activities related to the Feature Store.

The web application comprises a backend developed using the Django framework, and a frontend, developed thanks to the Nuxt framework.
In addition, the application is able to handle LDAP (Google) integration of Axyon AI users and that of AWS for dataset generation.
Talos is implemented on the company's internal server as a modification of the multidocker solution currently adopted for other projects. Despite not using a serverless architecture, a microservices/multidocker solution offers a simple way to achieve modularity and flexibility.

Talos is a project that evolved during my internship and will have a future even afterwards with new functionalities. The thesis project was therefore divided mainly into two phases:

- **Lygos:** development of a page that shows, for each Feed in production, the list of Feature Groups that are part of it. The page includes an ingestion tracker showing the ingestion status of Feature Groups in production and, for each Feature Group, the list of Feeds utilizing it.
- **Byzantium:** Talos guides users through the process of generating the Genome of a dataset, allowing them to choose Feature Groups, Features, Feed Code, and all the previously listed properties.

Talos currently provides the user, directly from the graphical interface, with the opportunity to generate a dataset with a one-day time interval in order to verify that the Genome is consistent. The next phase after the thesis project includes the possibility of enabling the generation of the complete dataset once the consistency of the test dataset is verified.

Sommario

Lo sviluppo dell' applicazione web "Talos" è un nuovo progetto dell'azienda Axyon AI con l'obiettivo principale di semplificare il processo di generazione dei dataset in "Datasmith" sfruttando un'interfaccia utente user-friendly (GUI). Datasmith è il progetto di Axyon AI mirato alla creazione di dataset utili sia per l'addestramento di modelli che per l'inferenza e le sperimentazioni interne all'azienda. Inoltre, Talos verrà utilizzato per tenere traccia delle principali attività di Datasmith, come l'ingestione storica e live dei Feature Group sul Feature Store AWS SageMaker direttamente dal pannello della web app. Per poter generare un dataset, è necessario conoscere diverse proprietà:

- L'universo investibile che stiamo considerando, identificato dal *Feed Code*
- Il tipo di Instrument inclusi
- Le Feature che vogliamo inserire nel dataset
- Le Feature di contesto per il dataset
- Il periodo temporale che il dataset dovrebbe coprire
- Il target (nel caso dei dataset di training)

Tutte le informazioni e le proprietà elencate, che descrivono il dataset, vengono incluse in un file di tipo YAML che contiene tutti i Feature Group con le corrispondenti Feature effettivamente presenti nel Feature Store. Questo file di tipo YAML rappresenta il "Genoma" del dataset tramite il quale, una volta pronto, è possibile generare il dataset reale.
Generare il Genoma di un dataset è attualmente un'operazione lenta, soggetta a errori e semi-manuale a causa della natura del file di tipo YAML che contiene un lungo elenco di Feature e Instrument. Talos mira a rendere il processo di generazione del Genoma più veloce e robusto, fornendo agli utenti di Datasmith uno strumento per monitorare tutte le principali attività relative al Feature Store.

La web application è composta da una parte backend, sviluppata utilizzando il framework Django, e da una frontend, sviluppata grazie al framework Nuxt. Inoltre l'applicazione è in grado di gestire l'integrazione LDAP (Google) degli utenti Axyon AI e quella di AWS per la generazione dei dataset.
Talos è implementato sul server interno dell'azienda come modifica della soluzione multidocker attualmente adottata per altri progetti. Anche non utilizzando un'architettura serverless, una soluzione microservizi/multidocker offre un modo semplice per raggiungere modularità e flessibilità.

Talos è un progetto che si è sviluppato nel corso del mio tirocinio ma avrà un futuro anche successivamente con nuove funzionalità. Il progetto di tesi è stato quindi suddiviso principalmente in due fasi:

- **Lygos:** sviluppo di una pagina che mostra, per ciascun Feed in produzione, l'elenco dei Feature Group che ne fanno parte. È presente anche un tracker di ingestione che mostra lo stato di ingestione del Feature Group in produzione e per ciascun Feature Group l'elenco dei Feed che lo utilizzano.
- **Byzantium:** Talos guida gli utenti attraverso il processo di generazione del Genoma di un dataset, consentendo di scegliere i Feature Group, le Feature, il Feed Code e tutte le proprietà precedentemente elencate.

Talos attualmente fornisce all'utente, direttamente dall'interfaccia grafica, l'opportunità di generare un dataset con intervallo temporale di un giorno al fine di verificare che il Genoma sia consistente. La fase successiva al progetto di tesi prevede la possibilità di abilitare la generazione del dataset completo, una volta verificata la consistenza di quello di test.

Contents

List of Figures

Listings

List of Acronyms

ML	Machine Learning
AI	Artificial Intelligence
GUI	Graphical User Interface
UI	User Interface
FG	Feature Group
ORM	Object Relational Mapping

Chapter 1

Introduction

This chapter will initially provide an overview of Axyon AI, the company where my internship was conducted, describing its structure and goals. Subsequently, the reader will be given a general overview of the project, starting from the context that gave rise to the need to carry it out as a thesis project, up to the definition of its phases, the goals to be achieved, and the tools used.

1.1 Presentation of Axyon AI

Axyon AI is an Italian fintech company based in Modena, committed to making the investment management industry more robust through the power of AI. The company provides AI-based solutions to asset managers, hedge funds, and institutional investors, generating predictive insights and identifying alpha opportunities.
Axyon Platform, the company's proprietary platform, is specifically designed for financial time series. It enables the design and development of extremely accurate AI/Deep Learning predictive models. Axyon IRIS, the product offered to the market, is an AI/Deep Learning engine utilizing proprietary technology to consistently provide precise forecasts for the behavior of indices and securities.

As reported on the company's official website [1], Axyon's focus is not on individual models but on designing and implementing a highly performant, efficient, and automated assembly line for strategy production. This process evolves over time through incremental improvements. Currently, Axyon IRIS strategies are powered by sets of supervised learning models, including neural network-based models and tree-based models. Initially, a large number of candidate models are explored using a hyperparameter search algorithm. Subsequently, these models are optimally combined to form a set of AI models used for generating historical (out of sample) forecasts and, ultimately, for production.

Various types of data from multiple sources are employed in the process. The primary data types used to create datasets include end of day (EOD) and intraday market data, fundamental indicators, macroeconomic indicators, related indices or securities, sentiment indicators extracted from news and social media, options data, and analyst forecasts.

The feature selection and hyperparameter optimization process employed for model development involve an in-depth exploration of the search space, whose dimensions depend mainly on the complexity of the dataset and modeling assumptions, determining the set of AI models available to the search algorithm. In most cases, a thorough exploration of this space is computationally unfeasible, necessitating the use of a stochastic optimization method to optimize hyperparameters and feature subsets.

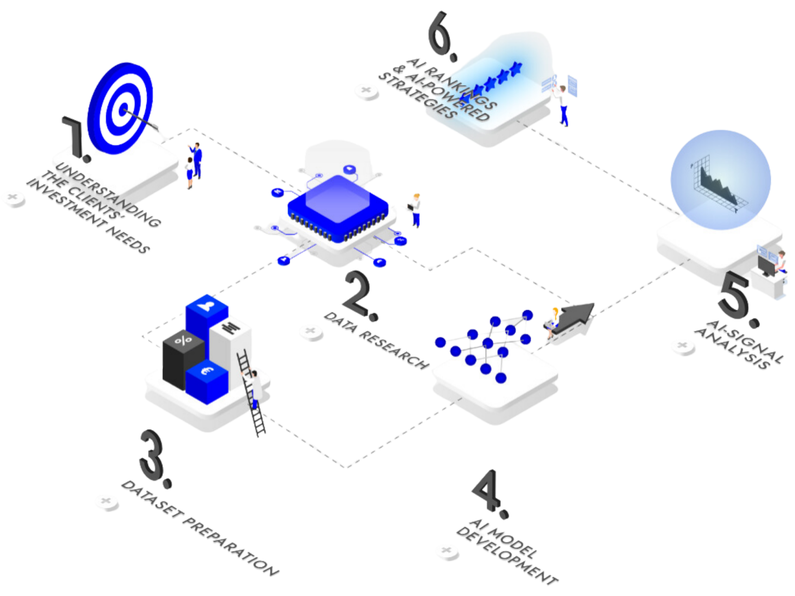

1. UNDERSTANDING THE CLIENTS' INVESTMENT NEEDS
2. DATA RESEARCH
3. DATASET PREPARATION
4. AI MODEL DEVELOPMENT
5. AI-SIGNAL ANALYSIS
6. AI RANKINGS & AI-POWERED STRATEGIES

Figure 1.1: how Axyon AI works [1]

In the figure 1.1, the company's workflow is visually represented, illustrating how data is leveraged and transformed into possible future investment decisions.
In particular, we are talking about AI signals closely correlated to an investable universe, which must be defined as the initial step in the construction of a new strategy, namely a new Axyon IRIS Feed.

The investable universe defines the set of potentially tradable assets. A static investable universe never changes after its definition. A dynamic investable universe may alter daily based on specific market rules. Axyon AI employs two types of investable universes:

- Customized investment universes tailored to client requests
- Investment universes aligned with product development choices

The project undertaken during the internship has had an impact on the third step: "Dataset Preparation", more precisely on the process of generating datasets used in the pipeline.

1.2 Contextualization of the project

As previously highlighted, the data used in the process of generating datasets, imperative for model development, are of different types and come from multiple sources. All these data are collected and stored within an Amazon S3 Data Lake.

Amazon Simple Storage Service (Amazon S3), as mentioned in Amazon S3 website [2], is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customers of all sizes and industries can store and protect any amount of data for any use case, such as data lakes, cloud-native applications, and mobile apps. With convenient storage classes and user-friendly management features, it's possible to optimize costs, organize data, and configure optimal access controls to meet specific business, organizational, and compliance requirements.

The data residing in the data lake undergo processing and validation during the data preparation phase, where any anomalies are rectified through structural adjustments, and missing data is addressed. After data preparation, the data are cleaned and ready to be transformed and engineered to create Features. Consequently, it becomes imperative for Axyon AI that this data, stemming from the Features selection process, maintains consistent storage and structuring within the Amazon SageMaker Feature Store.
In particular, the resulting Features are grouped based on characteristics and application into Feature Groups.
Amazon SageMaker Feature Store, as stated on the dedicated website [3], is a dedicated and fully managed repository for storing, sharing, and managing Features for machine learning models. Features serve as inputs for ML models during training and inference, and their quality is crucial for ensuring the creation of a highly accurate model. SageMaker Feature Store provides a secure and unified store to process, standardize, and use Features at scale throughout the machine learning lifecycle.

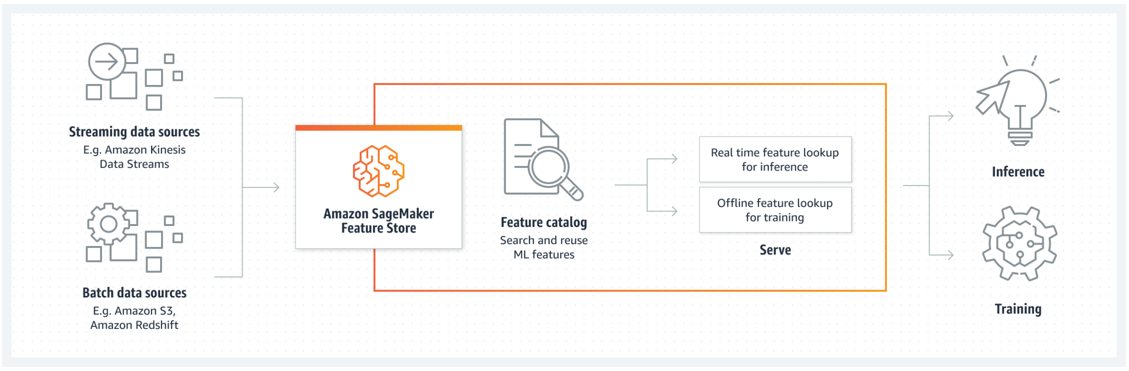

Streaming data sources
E.g. Amazon Kinesis Data Streams
Batch data sources
E.g. Amazon S3, Amazon Redshift
Amazon SageMaker Feature Store
Feature catalog
Search and reuse ML features
Real time feature lookup for inference
Offline feature lookup for training
Serve
Inference
Training

Figure 1.2: how Amazon SageMaker works [3]

Once the Feature Groups find their place on AWS, the final step to conclude the data preparation phase is the formulation of datasets tailored for model training and experimentation within the company. This commences with the careful selection and determination of which Features from the Feature Store are to be incorporated into the dataset.

1.3 Role and functions of Datasmith in Axyon AI

The context in which Talos web application operates is the one just described and is implemented within Axyon AI business process through Datasmith, the project aimed at creating datasets. Datasmith puts into practice all the steps that make up the data preparation phase, specifically:

- Reads raw data from the Datalake (S3 bucket).
- Computes Features on the read data.
- Creates, updates, or deletes Feature Groups hosted on AWS SageMaker Feature Store.
- Generates datasets by retrieving updated time series of Features from the Feature Store, combining them into a dataset based on a configuration file.

At the implementation level, in the Datasmith codebase, a class is created for each Feature Group, which will then be deployed to AWS SageMaker allowing all Features to be specified within the class. Each class contains a method for each Feature that needs to be implemented within the Feature Group.

There are level 0 Features that do not require any other Feature to be calculated but only need data from the data lake, and higher level Features that instead need to be able to use lower level features in order to be processed and inserted on AWS.
Datasmith can recognize the level of Features through the presence of a decorator associated with the method that defines them, so that the Features can be processed in order of level.
There are two types of Feature Groups:

- **Instrument-related:** the value of instrument-related Features changes for each date and for each Instrument. Instrument-related Features may or may not be associated with a Feed Code (Axyon IRIS Feed).
- **Context:** a context Feature value changes for each date but remains the same for each Instrument on that specific date. Each context Feature is associated with a Feed code.

In the case that there are Features associated with a Feed code, then the Features without Feed code will be processed first, in order of level, and then the Features with Feed code.

The information about Feature Groups, collected or produced by Datasmith, is stored within a series of tables with the prefix "sn_datasmith" that were critical to the development of Talos. In particular, the Datasmith tables used during the internship, and which I will explore in more detail during the thesis, are:

- sn_datasmith_feeds_in_feature_groups
- sn_datasmith_feature_groups
- sn_datasmith_feature_group_types

1.4 Objectives of Talos - web Datasmith

The reason that led Axyon AI to develop Talos project is to create a web application capable of monitoring and supporting the dataset generation process managed by Datasmith.

1) The primary objective of the application is to monitor the historical and live ingestion of Feature Groups into the AWS SageMaker Feature Store. This will make information available and clear to all Datasmith users during the process, and will make it easier to identify any errors in the ingestion of historical and especially live data.

The generation of each dataset is based on the previous creation of a configuration file, referred to in the company's process as the dataset "Genome". All the information and properties describing the dataset are then encapsulated in a YAML file representing the dataset Genome. Once prepared, this Genome serves as the basis for generating the actual dataset. Creating the dataset Genome is currently a slow and semi-manual operation prone to error due to the nature of the YAML file containing a lengthy list of Features and Instruments. Presently, users of Datasmith choose and input the properties into the configuration file without any support.

2) Talos aims to make the Genome generation process more efficient and robust by providing Datasmith users with a tool for the selection and configuration of all properties necessary for creating the final dataset. Additionally, Talos also aims to provide a dataset content verification tool, allowing for the analysis of the financial time series within it.

The overarching goal is therefore to develop a web application that can improve the efficiency of the dataset generation process for machine learning, ensuring an enhanced analysis of financial time series at Axyon AI and, more broadly, an improvement in the overall workflow of the company shown in figure 1.1.

1.5 Tools

For the planning and design of the tasks to be carried out during the internship, two highly useful tools were employed: Jira [4], a software development tool utilized as a team for project management, and Moqups [5] for idea sharing and visual collaboration.

At the code level, the web application consists of a backend, developed in Python using the Django framework, and a frontend, crafted with the Nuxt framework in JavaScript. As for accessing data in the company databases, SQL (Structured Query Language) was utilized.

Talos is implemented on the company's internal server as a modification of the current multi-docker solution adopted for other projects. As stated in Docker documentation [6], a container is a standard software unit that encapsulates code and all its dependencies so that the application can run quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, and executable software package that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. Therefore, two separate Docker containers were created, one for the backend and a second for the frontend, which are initialized together through a docker-compose. This solution facilitated modularity and flexibility for the project.

As mentioned earlier, it was necessary to consult and use various tables, accessing existing MySQL databases within the company but also creating a specific one for the web application. DBeaver [7], a cross-platform database tool supporting all major SQL databases such as MySQL and SQLite, has been highly useful for this purpose.

Like all Axyon AI projects, a Git repository was created for Talos on the GitLab web platform, enabling collaborative work and sharing among multiple contributors. GitLab [8] is a version control system that allows saving code in public or private repositories and supports operations like pull (download the code stored in the remote repository locally), push (upload changes made locally to the public repository), and merge (combine proposed changes with the original code). This way, collaboration with other programmers was possible while working independently, avoiding conflicts.

Chapter 2

Analysis and methodology

From the very first day of the internship, the development of Talos was fully integrated into the company's DevOps process, taking shape as a new Epic in the Axyon Ai Gantt chart on the Jira team collaboration platform.
This chapter provides a detailed explanation of the methodology used to advance the project and the decisions made during the design phase.

2.1 Work plan according to the Scrum methodology

The working approach adopted by Axyon AI, and consequently the strategy used in the development of Talos, follows the Scrum methodology: a framework for developing and supporting complex products created by Ken Schwaber and Jeff Sutherland. Its aim is to support individuals and teams in collaboratively creating value.

Axyon Ai chooses to embrace the Scrum methodology primarily because it aligns with a simple but foundational idea of the corporate vision: achieving goals as a team by breaking down the work into small pieces, constantly experimenting with new solutions, and collecting feedback along the way to improve over time, both as individuals and as a team. Additionally, this framework provides an appropriate structure to allow diverse individuals to integrate into the way they work without disrupting individual characteristics and specific needs, which are the assets of every development team.

The values of this framework such as courage, focus, commitment, respect and openness, are all elements that members of a work team should pursue and become particularly important in contexts in which the experiment is essential to make progress, just like the context by Axyon AI.

Scrum operates as an empirical process, making decisions grounded in observation, hands-on experience, and a spirit of experimentation. It is built on three pillars: transparency, inspection and adaptation which support the concept of working iteratively through small experiments and adapting both what you are doing and how you do it as needed.

Getting down to specifics, however, thanks to the Scrum guide provided by the creators of the framework in 2020 [9], we can define the components and phases of this methodology, which precisely describe what happened during my internship.

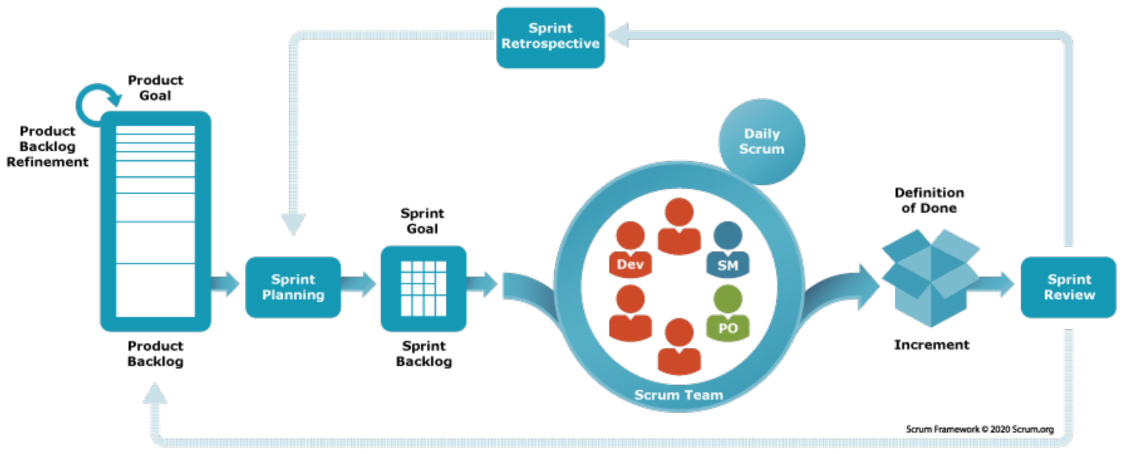


Figure 2.1: Scrum framework [10]

As highlighted in the representation 2.1, the Scrum framework comprises a Team, a Product Owner, a Scrum Master and developers with specific responsibilities. The Scrum Team participates in five events and generates three artifacts during a period of activity called Sprint, lasting two weeks in Axyon.

Scrum includes the role of a Scrum Master to foster an environment in which the Product Owner organizes the work related to a complex problem. Over the course of the Sprint, the Scrum Team transforms a portion of the work into an Increment of value.

Finally, the Team, together with the stakeholders, reviews the results and adapts the work for the next Sprint. Regarding the methodology applied in the development of Talos, the internship covered about ten Sprints, each with distinctive characteristics.

1) Main players of the Sprints:

- The **Scrum Team** constitutes the fundamental entity of the Sprint. While carrying out a project like Talos, assigned mainly to me, I have always been part of a team of around five people, a size small enough to guarantee agility and productivity.
- **Developers** are the team members responsible for pursuing the Sprint objectives.
- The **Product Owner** is responsible for each team's outcome during the Sprint and manages the Product Backlog on the Jira software platform. In Axyon AI, there are two Product Owners, one for the business side and one for the technical side.
- The **Scrum Master** is the person responsible within the team for promoting and implementing the Scrum methodology, acting as a connection between the team and the Product Owners.

2) Main and recurring Sprint events:

- The **Sprint** constitutes the heart of the Scrum methodology, serving as a container for the other events. It is what allowed me to have short/medium term objectives and to constantly evaluate and improve the work done.

- **Sprint Planning** starts the Sprint. It took place every two weeks, on Mondays, during the entire internship period and is divided into two moments: an initial meeting managed by the Product Owners and an actual planning session during the day. During this phase, the various Sprint Goals that each team must pursue are defined and assigned. For Talos, I was almost always responsible for a Sprint Goal.
- The **Daily Scrum** is a meeting aimed at monitoring progress towards the Sprint Goal and in which to organize ourselves as a team on daily work.
- The **Sprint Review,** divided between the business aspect and the technical aspect for Axyon AI, aims to analyze the results of the Sprint and determine future adaptations. The Scrum Team presents the key results of their work to stakeholders and discusses progress towards the Product Goal.
- The **Sprint Retrospective** serves to identify possible improvements in the quality and effectiveness of the Sprint, allowing each Scrum Team to analyze every aspect of the methods adopted in the just concluded Sprint.

3) Sprint Scrum artifacts, designed for transparency and collaboration:

- The **Product Backlog** is a list of needs to improve the company product. In Axyon, it is represented in a Gantt chart on the Jira software platform, where it is possible to insert Epics (Product Goals), but also individual stories and tasks to complete. From the beginning, Talos was listed as an Epic scheduled between September and January.
- The **Sprint Backlog** is a plan developed by and for Developers in each Sprint. In particular, it was my job, together with other developers, to define the Goal on Talos for each Sprint.
- The **Increment** represents the added value of the Sprint by each team.

Respecting the Scrum methodology just described, the first phase of the development of Talos was the design and definition phase of the Epic, which had already been planned before my internship, but which took shape and structure in the first Sprints of September.

2.2 Talos structure and architecture

As already mentioned in the introductory chapter, Web Datasmith is deployed on Axyon AI's internal server using a microservices/multidocker solution, a choice already adopted for other projects. This architectural approach is preferred over the strictly monolithic one as it allows achieving modularity and flexibility, optimizing the use of resources and not excluding the possibility of creating new services in the future.

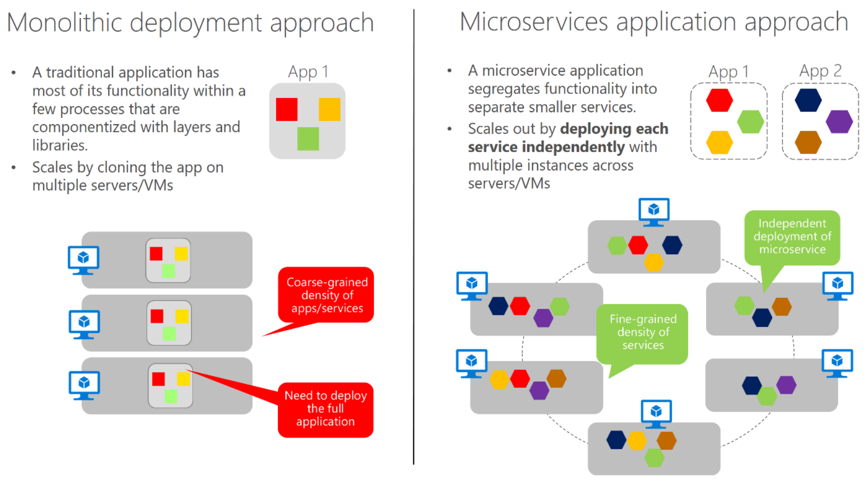

Monolithic deployment approach
• A traditional application has most of its functionality within a few processes that are componentized with layers and libraries.
• Scales by cloning the app on multiple servers/VMs
App 1
Coarse-grained density of apps/services
Need to deploy the full application
Microservices application approach
• A microservice application segregates functionality into separate smaller services.
• Scales out by deploying each service independently with multiple instances across servers/VMs
App 1
App 2
Independent deployment of microservice
Fine-grained density of services

Figure 2.2: monolithic deployment vs microservice approach [11]

Docker, an open source software developed in the Go programming language, facilitates the deployment of containerized software systems. These containers encapsulate the application and all its dependencies, allowing for flexible execution in any environment.
To automate the deployment process and reduce the need for operators with high permissions, an automated deployment pipeline was implemented. Docker represents a *de facto* standard for systems such as the one proposed in Talos.

Additionally, Docker provides tools like Docker Compose, which allows the definition and management of multi-container applications. The architecture of the internal web app Talos, implemented with Docker, consists of a container for the frontend and one for the backend, and can be outlined by considering the following aspects:

1) Frontend container:

- This Docker container encloses the code for the frontend of the application, including HTML, CSS, and JavaScript files.
- Based on a Node.js image, designed to deliver fast and scalable server-side applications and networking.
- Communicates with the backend through HTTP requests or APIs.

2) Backend container:

- This Docker container hosts the backend of the application, including the web server, business logic, and database connection.
- Built upon a base image from Axyon projects, containing the runtime system for the Python programming language.
- Copy and install the Datasmith project inside the datasmith submodule, thus allowing the management of dataset generation as previously illustrated in the chapter.
- Set up your AWS credentials to grant access to the AWS SageMaker Feature Store and S3 bucket.
- Exposes an API used by the frontend to communicate with the backend.

3) Deployer container:

- This Docker container is a service that implements an automated deployment pipeline. The service is based on a Python image and uses Flask to implement the pipeline.

4) Talos network:

- The Talos network is a bridge-type Docker network called "talos-network". The network is used to connect the web app containers.

5) Communication between frontend and backend:

- The frontend sends requests to the backend through HTTP API calls.
- The backend processes the requests, accesses the necessary data, and returns responses to the frontend.

6) Dependency management and isolated environments:

- Each container has its dependencies and libraries, isolated from the host system and other containers, ensuring a cohesive and easily replicable environment.
- Configurations and environment variables are managed through specific Docker configuration files.
- Replicating containers across different machines is facilitated to ensure flexibility and scalability.

In the appendix A it is possible to observe the structure of the project codebase in its entirety, with the subdivision of the various containers and all the necessary configuration files, with particular reference to the docker-compose, which manages the creation of the various environments. The two architectural blocks on which I mainly worked during the internship are the two sides of the web application, backend and frontend.

2.3 Technologies and framework used - backend side

To implement the backend side of Talos, as a team we opted to use the Django framework, known as "The web framework for perfectionists with deadlines".
Django is an open source web framework, written in Python, which stands out for its completeness and power. This choice was motivated by Django's ability to manage the development of complex and highly scalable web applications.
The preference fell on this framework not only for its intrinsic characteristics, but also because thanks to its active community and its widespread adoption in projects of different complexities, it presents extremely complete documentation.[12]

Django adopts a specific version of the MVC (Model View Controller) architecture called Model View Template (MVT). This structure divides the framework logic into three main components: data management (Model), presentation logic (View), and flow control (Template). This approach makes Django extremely flexible and reusable.

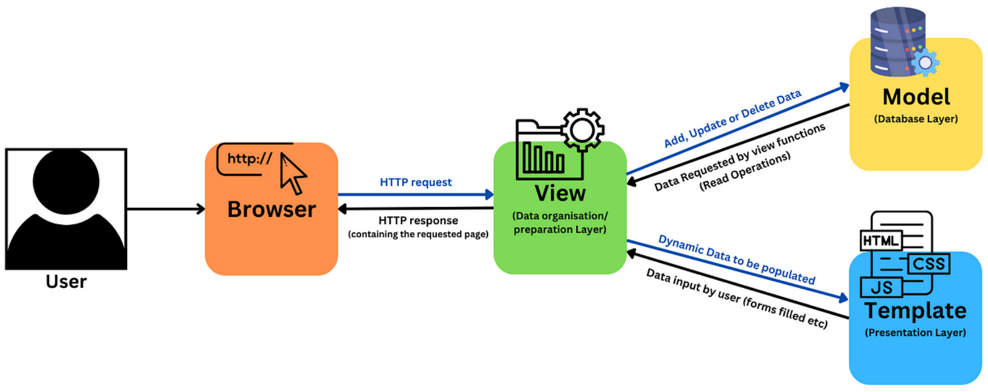

User
http://
Browser
HTTP request
HTTP response
(containing the requested page)
View
(Data organisation/ preparation Layer)
Add, Update or Delete Data
Data Requested by view functions (Read Operations)
Model
(Database Layer)
Dynamic Data to be populated
Data input by user (forms filled etc)
HTML
CSS
JS
Template
(Presentation Layer)

Figure 2.3: Django MVT architecture [13]

In Django's conception, the Controller corresponds to the framework itself, while the Views do not determine how data should be displayed, but rather which data should be displayed. The presentation process is defined in the Templates. However, in Talos, we handled the visualization on the frontend side using another framework, and therefore Django Templates were not employed.

As for data management, Django provides an integrated ORM that simplifies interaction with the database. Object Relational Mapping represents a programming technique that facilitates the integration between software systems based on the object-oriented programming paradigm and RDBMS relational database management systems. Thanks to Django it is therefore possible to define data models in Python, and the framework automatically takes care of the creation or modification of tables in the database as well as CRUD (Create, Read, Update, Delete) operations.

A Django Model defines the structure and behavior of the data necessary for the operation of the web application. Generally, each Model is mapped to a single database table, which is why in Talos we have taken the approach of creating a backend model for each table used. In practice, each Model is implemented as a Python class that inherits from

the django.db.models.Model class. Each attribute of the class represents a database field, defining the data type and all its properties.
After defining or modifying a Model, Django offers the ability to generate "Migrations", which represent changes to the database schema or a single table. These migrations can be applied to the database to reflect changes made to the models. Once data models are created, Django also provides a database abstraction API that allows you to make complex queries without writing direct SQL, with the use of "QuerySets".

If the models deal with the management of data in the database, the presentation logic (View) is the fundamental element for managing user requests and determining how to respond to each request. This process involves two main components: the routing system and the views:

- The routing system, or URL configuration, determines how URLs are mapped to views within a Django application. The main file that handles the mapping is commonly called urls.py and is responsible for routing URL requests to the corresponding views.
- On the other hand, views are responsible for handling URL requests specified in the routing system and can return HTML pages, JSON data, or perform other actions. Views can be implemented as Python functions or classes and when a URL configuration is requested, Django determines which view to call based on the established configurations.

In summary, the routing system and views in Django allow to specify the way in which URL requests must be handled, directing the flow to the corresponding views that determine what must be shown or executed in response to a specific request arriving from the frontend side of the application.

2.4 Technologies and framework used - frontend Side

To create the Talos user interface, as a team we chose to adopt Nuxt, an open source framework based on Vue.js, known as "The Intuitive Vue Framework".
Nuxt goes beyond the core functionality of Vue.js, extending its versatility and greatly simplifying the creation of sophisticated applications. This is made possible thanks to a convention management system, which offers predefined structures for crucial aspects such as automatic routing and server-side rendering (SSR). This approach allowed me to focus on the application logic without having to deal with complex configurations, thus speeding up the development process and improving code consistency.

Before exploring Nuxt, it is essential to understand the fundamental role of Vue.js.
Vue is a progressive framework [14] designed for creating dynamic user interfaces, based on the standard HTML, CSS and JavaScript languages. Its philosophy is founded on a declarative programming model, offering a clear and intuitive approach to structuring applications in a modular way. What primarily guided our decision to use Vue was above all its incremental nature, a feature that will allow the web application to evolve over time with new functionalities.

The Nuxt framework, as reported in the official documentation [15], therefore successfully integrates all the advantages offered by Vue, creating a web development environment enriched with distinctive features and strengths:

- The philosophy of Nuxt.js is based on the concept of conventions, which translate into an organized and easily understandable project structure. This approach favors a rapid learning curve and facilitates collaboration between developers, a characteristic that has been confirmed with the positive experience obtained through Talos.
- Server-Side Rendering is a key feature that provides the ability to render pages server-side before sending them to the client. This aspect allows the complete rendering of pages before they reach the user's browser, significantly improving the user experience and facilitating indexing by search engines.

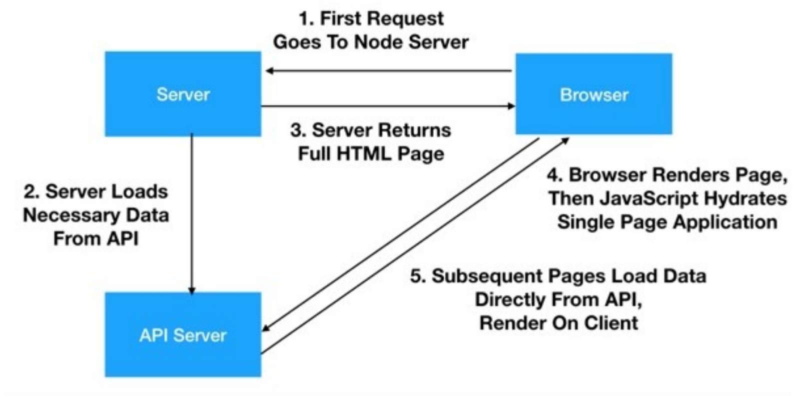

1. First Request Goes To Node Server
Server
Browser
3. Server Returns Full HTML Page
2. Server Loads Necessary Data From API
4. Browser Renders Page, Then JavaScript Hydrates Single Page Application
5. Subsequent Pages Load Data Directly From API, Render On Client
API Server

Figure 2.4: universal Nuxt JavaScript architecture [16]

- Automatic Routing greatly simplifies route management through an automatic system based on folder and file conventions. It was therefore easy to organize the code in an intuitive way by having the corresponding routes dynamically generated by the framework itself. This approach minimizes the need for manual configurations allowing for faster, more cohesive development.

To complete the panorama of technologies employed in developing the frontend of the web application, it was decided to integrate the Vuetify design framework into the Vue.js and Nuxt.js ecosystem.
Vuetify [17] is a complete user interface framework built right on top of Vue.js, designed to provide a rich set of predefined UI components that respect Google's material design principles. Its Vue.js specific design allows for seamless integration with Vue's state management system, leveraging the framework's reactive capabilities for dynamic state management.
The vast range of ready to use components, including buttons, cards, navigation bars and forms, guaranteed the project flexibility and customization, significantly simplifying the development of Talos.

The synergistic integration of Nuxt.js, Vue.js and Vuetify in the frontend of the web application represented a decisive step in the development path of the project. The combined use of these technologies has provided a solid foundation for the creation of a modern, responsive user interface that combines the required performance with well structured code and an appealing user interface

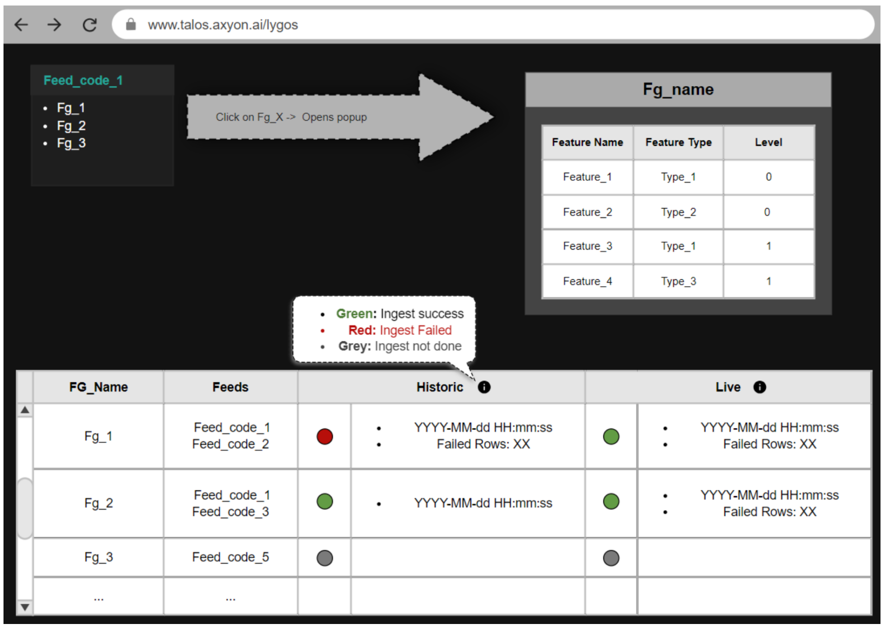
Chapter 3

Lygos implementation

As reported in the chapter 1, the first reason that leads to the need for Axyon AI to develop Talos is to easily consult the availability of Features within the AWS Feature Store through a Graphic User Interface. This enables the information to be accessible and clear to all the protagonists of the business process. The implementation of these functionalities throughout the project is referred to as "Lygos" and is described in this chapter.

3.1 Mockup of the page in the web application

Initially, during the Lygos implementation phase of the project, together with the Axyon AI team we developed a mockup for the interface, trying to incorporate all the properties necessary for the correct functioning of the Feature Store tracking page.



www.talos.axyon.ai/lygos

Feed_code_1

- Fg_1
- Fg_2
- Fg_3

Click on Fg_X -> Opens popup

Fg_name

Feature Name	Feature Type	Level
Feature_1	Type_1	0
Feature_2	Type_2	0
Feature_3	Type_1	1
Feature_4	Type_3	1

- **Green:** Ingest success
- **Red:** Ingest Failed
- **Grey:** Ingest not done

FG_Name	Feeds	Historic		Live	
Fg_1	Feed_code_1 Feed_code_2	🔴	• YYYY-MM-dd HH:mm:ss • Failed Rows: XX	🟢	• YYYY-MM-dd HH:mm:ss • Failed Rows: XX
Fg_2	Feed_code_1 Feed_code_3	🟢	• YYYY-MM-dd HH:mm:ss	🟢	• YYYY-MM-dd HH:mm:ss • Failed Rows: XX
Fg_3	Feed_code_5	⚫		⚫	
...	...				

Figure 3.1: mockup of the monitoring page

In particular, from the mockup represented in figure 3.1, it is possible to identify three areas on which I focused my work, in temporal order, for the implementation of the Lygos phase:

- The **Feed monitoring** section at the top of the mockup 3.1 is represented by a series of cards, (one as example in the figure) each containing information on the Axyon AI Feeds. Each card must associate all the Feature Groups used for the Feeds in production.
- A **Feature Group ingestion tracker** table, located at the bottom of the mockup 3.1, is intended to provide further details about the Feature Groups, specifying in which Feeds they are used.

  In the "Historic" section of the table, it is expected to display the ingestion status in the Feature Store of historical data related to Feature Groups, checking for the presence or absence of failed rows.

  In the "Live" section, on the other hand, the data ingestion status from the day preceding the current date must be displayed, identifying any anomalies.
- The table displayed at the top right of the figure 3.1, which appears with a pop-up window, shows the individual **Feature Groups composition** listing all the Features that are part of it. This window appears upon clicking the name of a Feature Group in the cards or in the table.

From an implementation point of view, regarding the backend side of the application, the use of the Django framework allowed the creation of a single app called "lygos" within the project, guaranteeing an organized and modular structure.
For this reason, a separate directory has been created for the monitoring page which contains its own models, views, etc. Outside of this folder there are configuration files common to all apps, such as "settings.py", with any models common to the entire project included in the folder called "common".

The lygos app was generated using the specific command "python manage.py startapp lygos", which automatically generated the basic structure, including the "models.py", "views.py" and "urls.py" files. After creation, the lygos app was configured in the "settings.py" file in the ”INSTALLED_APPS” section.

As for the frontend aspect of Lygos, a file named "lygos.vue" has been created inside the "pages" folder. This file represents the graphical interface of the monitoring page, which also makes use of components present in the "components" directory.
The codebase structure just described of this implementation phase, frontend and backend, can be observed in appendix A.

3.2 Feed monitoring

The data relating to the Axyon AI Feeds and the associated Feature Groups, to be inserted in the cards of the Feed monitoring section, are contained in the Datasmith table ”sn_datasmith_feeds_in_feature_groups”, whose structure can be analyzed in the Django Model included by appendix B.
During the fetch phase on the frontend side of the "lygos.vue" page, data is requested through an HTTP request performed using the Axios library. Axios provides a simple and clean interface for making HTTP requests. It can be used to perform GET, POST, PUT, DELETE and other requests. In this case it is used to make a GET request, the resulting data from this request is inserted into the variable ”fg_in_feeds”.

```
async fetch() {
        let [res_feeds, res_fgroups] = await Promise.all([
            this.$axios.get("/api/v1/lygos/feeds-list"),
            this.$axios.get("/api/v1/lygos/fg-ingest"),
        ])
        this.fg_in_feeds = res_feeds.data;
        this.tableData = res_fgroups.data;
```

Listing 3.1: fetch of lygos.vue page in Vue.js

The HTTP request is received on the backend side through the "feeds-list" url pattern, which directs the HTTP request to the "feeds_list" view. The latter simply queries the ”sn_datasmith_feeds_in_feature_groups” table via its Django Model and returns the data to the frontend in the form of JsonResponse.
The query executed on the table groups the data by "feed_code" and aggregates all the Feature Groups by associating them with the corresponding "feed_code".

From the user's point of view, information on Axyon AI Feeds is presented through a series of customized cards that slide within a group of slides, thus giving the end user the possibility to navigate between the various cards, as clearly illustrated in the figure 3.2.

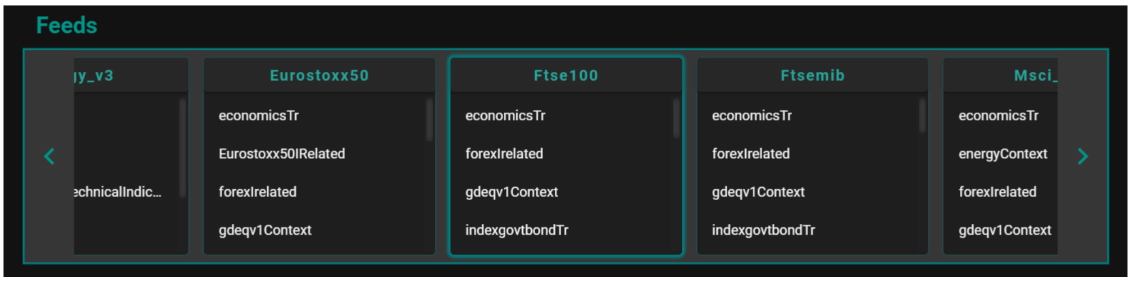


Figure 3.2: Feeds monitoring section

Each card, representing a single Feed, can be selected as illustrated in the figure 3.2. When a Feed is selected, the table in the "Feature Group ingestion tracker" section is filtered, showing only the Feature Groups associated with that Feed.

3.3 Feature Groups ingestion tracker

The table that monitors the ingestion status of Feature Groups in the AWS Feature Store consists of several columns. The data for these columns is requested from the frontend side in the "lygos.vue" page during the fetch phase through an HTTP request and is subsequently assigned to the "tableData" variable, as shown in 3.1

The first two columns of the table provide information about Feeds that use a particular Feature Group. This information is obtained through a query similar to the one performed on the "sn_datasmith_feeds_in_feature_groups" table in the previous section 3.2, except that the data are grouped by "fg_name_id" rather than "feed_code".

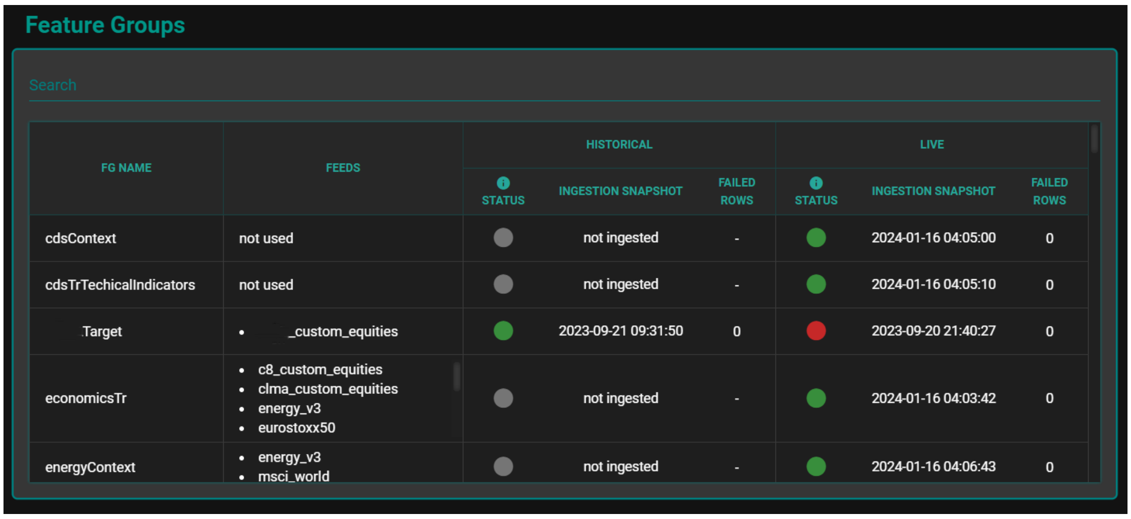
For the columns relating to data ingestion in the Feature Store, a query is executed on the ”sn_datasmith_feature_groups” table using the Django Model specified in appendix B. In particular, the columns relating to the name of the Feature Groups, the temporal snapshot of the historical and live data, as well as the number of rows that were not properly ingested into the Feature Store, for both historical and live data, are selected.

This data is then processed to determine the status of the Feature Group, merged with the first two columns obtained with the first query, and sent to the frontend as a JsonResponse.

```python
    fg_groups = get_feeds_in_feture_groups("fg_name")
    feature_groups_table = get_fg_ingest()
    fg_table = pd.merge(
        feature_groups_table,
        fg_groups,
        left_on="fg_name_suffix",
        right_on="fg_name_id",
        how="outer",
    )
    fg_table["historical_status"] = fg_table.apply(
        lambda row: "success--text"
        if (row["historical_ingestion_snapshot"] and
            row["historical_failed_rows"] == 0)
        else (
            "error--text"
            if (
                row["historical_ingestion_snapshot"]
                and row["historical_failed_rows"] > 0
            )
            else "accent--text"
        ),
        axis=1,
    )
    fg_table["live_status"] = fg_table.apply(
     ...
    )
```

Listing 3.2: Django view "fg_ingest" in Python

From the user's perspective, Feature Groups ingestion information is displayed within a Vuetify v-data-table component. Various modifications and customizations have been made to this component in order to ensure optimal graphic presentation and complete flow of the data within it, as can be seen in the figure 3.3.



Feature Groups

Search

FG NAME	FEEDS	HISTORICAL			LIVE		
		STATUS	INGESTION SNAPSHOT	FAILED ROWS	STATUS	INGESTION SNAPSHOT	FAILED ROWS
cdsContext	not used	⚪	not ingested	-	🟢	2024-01-16 04:05:00	0
cdsTrTechicalIndicators	not used	⚪	not ingested	-	🟢	2024-01-16 04:05:10	0
[redacted]Target	• [redacted]_custom_equities	🟢	2023-09-21 09:31:50	0	🔴	2023-09-20 21:40:27	0
economicsTr	• c8_custom_equities • clma_custom_equities • energy_v3 • eurostoxx50	⚪	not ingested	-	🟢	2024-01-16 04:03:42	0
energyContext	• energy_v3 • msci_world	⚪	not ingested	-	🟢	2024-01-16 04:06:43	0

Figure 3.3: Feature Groups ingestion tracker section

Historical case:

- Green -> time snapshot present, failed rows = 0
- Red -> time snapshot present, failed rows != 0
- Gray -> not ingested or otherwise

Live case:

- Green -> today's time snapshot present, failed rows = 0
- Red -> today's time snapshot present, failed rows != 0
- Red -> snapshot present but not from the current date
- Gray -> not ingested or otherwise

When a card is selected in the Feed monitoring section, the data present inside the variable "tableData" is filtered, keeping only the Feature Groups associated with the selected Feed active. This mechanism is also activated if you enter text in the search field at the top of the table. However, it is important to note that the search is performed on all columns and not just the one containing the Feature Group name.

3.4 Feature Groups composition

If a Feature Group is selected in one of the cards in the Feed monitoring section or directly in one of the table rows in the ingestion tracker section, a pop-up with the description of the Feature Group will be opened. Inside the opened window, there is a table that illustrates the composition of the Feature Group, listing all the Features belonging to it.
Each Feature is accompanied by information such as the type of data it contains, the level (discussed in section 1.3) and whether or not it is associated with a specific Feed.
The necessary data is requested from the frontend side on the Lygos page using the "OpenPopup" method. This method is triggered when you select a Feature Group and make an HTTP request, including the selected Feature Group as a parameter.

```
async OpenPopup(fg_i) {
            var data = {};
            data.fg_name_suffix = fg_i;
            const response = await
               this.$axios.$get("/api/v1/lygos/fg-details", {
               params: data });
            this.popupData = response;
            this.searchPopup = '';
            this.isPopupOpen = true;
            this.popupTitle = fg_i;
        }
```

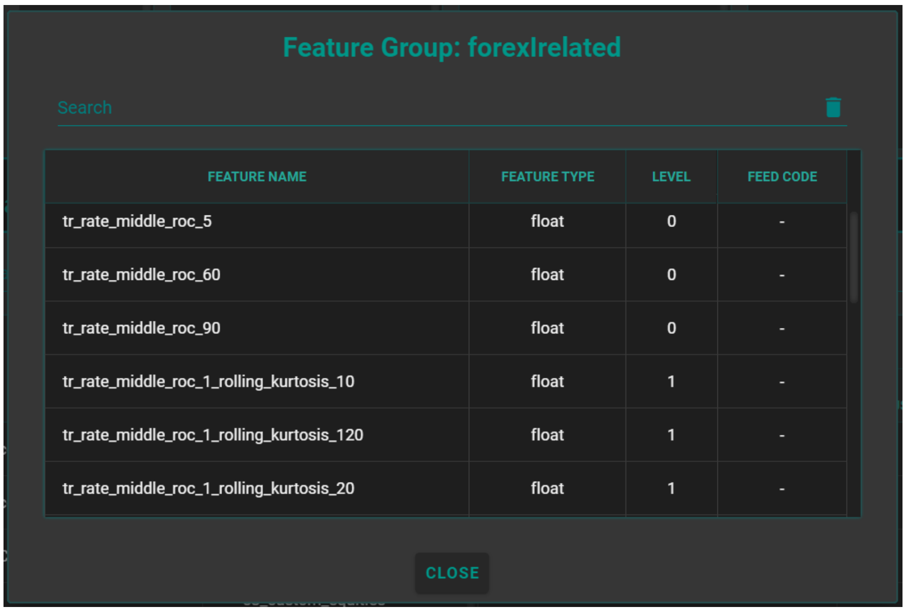
Listing 3.3: OpenPopup method of lygos.vue page in Vue.js

The request is received from the backend side via the "fg-details" url pattern, which routes the HTTP request to the "fg_details" view.
The latter searches for the class relating to the selected Feature Group within the Datasmith codebase, listing the description of all the Features contained in the class, as shown in 3.4. At this point, the data is returned to the frontend using JsonResponse.

```
@api_view(["GET"])
def fg_details(request) -> JsonResponse:
    fg = request.query_params.get("fg_name_suffix")
    module_name = "datasmith.features." + fg
    class_name = fg.capitalize()
    module = importlib.import_module(module_name)
    class_var = getattr(module, class_name)
    feature = class_var()
    fg_descr = feature.describe()
    fg_descr = [{"feature_name": key, **value} for key, value in
       fg_descr.items()]

    for f_dict in fg_descr:
        if f_dict["feed_code"] is None:
            f_dict["feed_code"] = "-"
    return JsonResponse(fg_descr, safe=False)
```

Listing 3.4: Django view "fg_details" in Python

In the GUI managed by the Vue page "lygos.vue", when the "isPopupOpen" variable becomes true, a Vuetify v-dialog component is triggered. This component acts as a container for a v-data-table component, which lists all the Features found in the Datasmith codebase class. Also in this case there is a text search field to filter the table.



Feature Group: forexlrelated

FEATURE NAME	FEATURE TYPE	LEVEL	FEED CODE
tr_rate_middle_roc_5	float	0	-
tr_rate_middle_roc_60	float	0	-
tr_rate_middle_roc_90	float	0	-
tr_rate_middle_roc_1_rolling_kurtosis_10	float	1	-
tr_rate_middle_roc_1_rolling_kurtosis_120	float	1	-
tr_rate_middle_roc_1_rolling_kurtosis_20	float	1	-

Figure 3.4: Feature Group composition section

Chapter 4

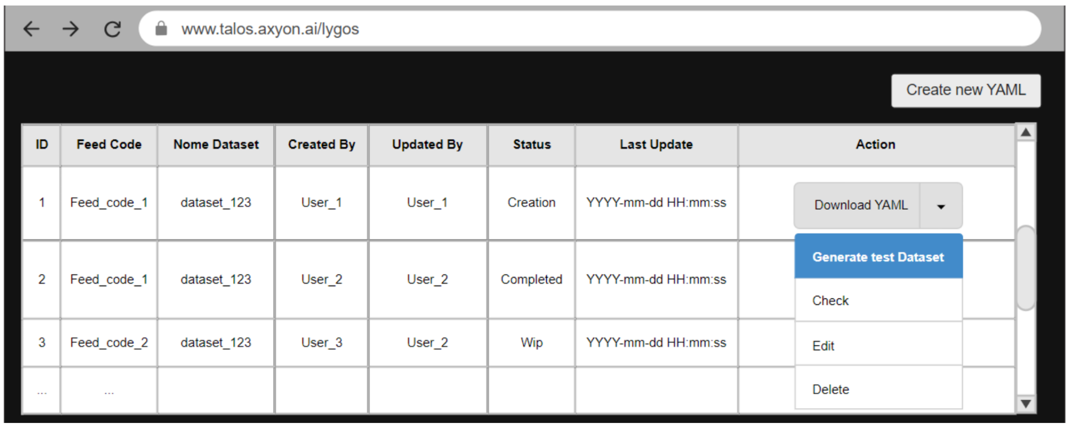
Byzantium implementation

As mentioned in the introductory chapter 1, the second objective that led Axyon AI to the development of Talos is to make the process of generating the Genome of a dataset faster and more robust, providing Datasmith users with a web application for selection and choice of all the properties necessary for the creation of the final dataset.
The implementation phase of this goal during the project is called "Byzantium" and will be described in this chapter.

4.1 Mockup of pages in the web application

Initially, in the implementation phase of the project called Byzantium, in collaboration with the Axyon AI team, we created a mockup for the graphical interface. The objective was to incorporate all the properties necessary for the correct functioning of the Genome generation process of a dataset. In this phase, the state is programmed on two main pages:

1) A first page for managing and displaying each Genome, created by one of the different users, to be able to modify, delete, or test it through the test dataset generation.



ID	Feed Code	Nome Dataset	Created By	Updated By	Status	Last Update	Action
1	Feed_code_1	dataset_123	User_1	User_1	Creation	YYYY-mm-dd HH:mm:ss	Download YAML
2	Feed_code_1	dataset_123	User_2	User_2	Completed	YYYY-mm-dd HH:mm:ss	
3	Feed_code_2	dataset_123	User_3	User_2	Wip	YYYY-mm-dd HH:mm:ss	
...	...						

Figure 4.1: Genome generation index page

From the mockup illustration provided in the figure 4.1, it is evident that essential informational data for each Genome are included in the table, with a specific indication of whether a Genome has been subsequently modified from its creation and by which user.
In the "Action" column, various operations can be performed on the Genome. In particular, by selecting the "Check" option, a pop-up window is opened, providing a summary of the Genome and, if it exists, the generated test dataset view.

2) A second page for Genome generation, guiding the user through the creation process with a step-by-step approach. The Genome creation page remains the same, but its components alternate during the generation process to ensure the input of all necessary properties.
In the mockup figures 4.2 and 4.3 are visible two examples of some steps of the process.

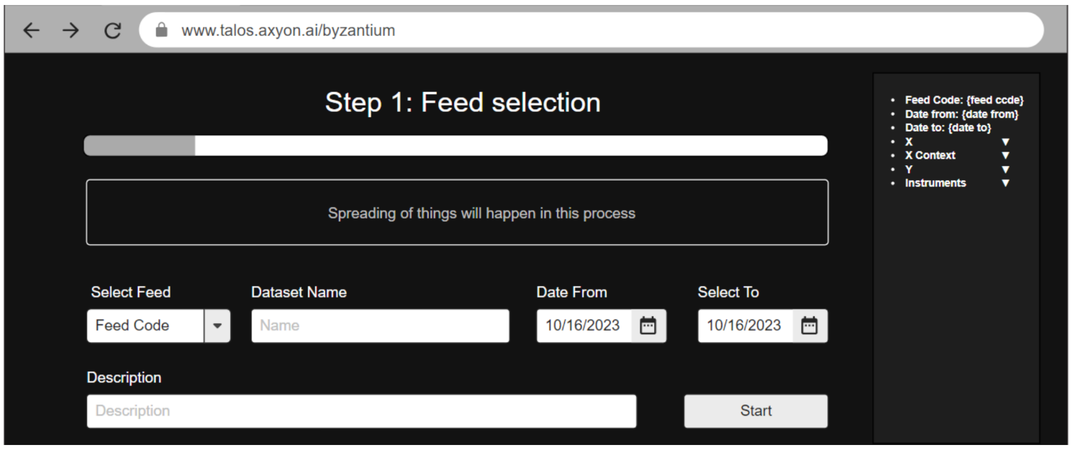


Figure 4.2: Genome generation step 1

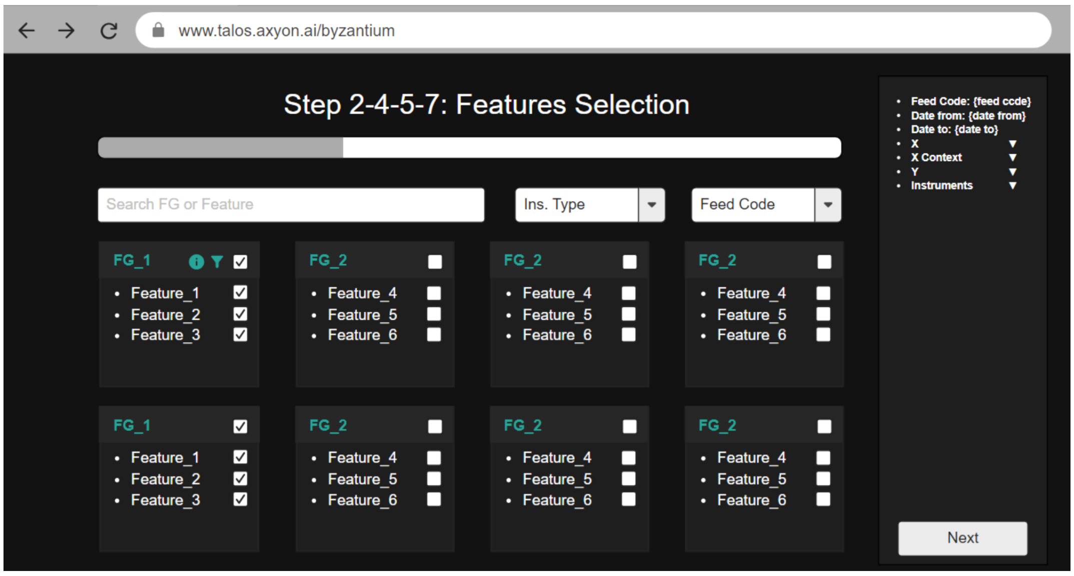


Figure 4.3: Feature selection step 2,4,5,7

On the right side of the page in the mockup, a representation of the Genome being generated remains consistently visible, while in the center, the various steps of the process unfold. Specifically, the steps that guide the user in generating a complete Genome are:

- Step 1: Feed Selection
- Step 2: Instrument related Features Selection
- Step 3: Instruments Mapping
- Step 4: Context Features Selection
- Step 5: Custom Context Features Selection
- Step 6: Context Assets Selection
- Step 7: Target Selection
- Step 8: Recap

From an implementation perspective, similarly to what has already been done in the Lygos phase, concerning the backend of the application, the use of the Django framework has allowed for the creation of a single app named "byzantium" within the project, ensuring an organized and modular structure.

As for the frontend side of Byzantium, two files named "index.vue" and "create-genome.vue" have been created within the "pages\byzantium" directory. The first file represents the graphical interface of the monitoring page for each genome, while the second contains the page that guides the user through the genome generation process. The latter extensively utilizes components found in the "components\byzantium" directory to implement the various steps of the process.

The structure just described for this implementation phase, both for the frontend and the backend, can be observed in appendix A.

4.2 Structure of a dataset Genome

Before going into the details of the implementation of the two pages illustrated in the mockup of the previous section, it is essential to provide a more in-depth description of what the Genome of a dataset represents in the Axyon AI development context and how it is structured.

To generate a dataset useful for model development in Axyon AI, it is necessary to understand several properties:

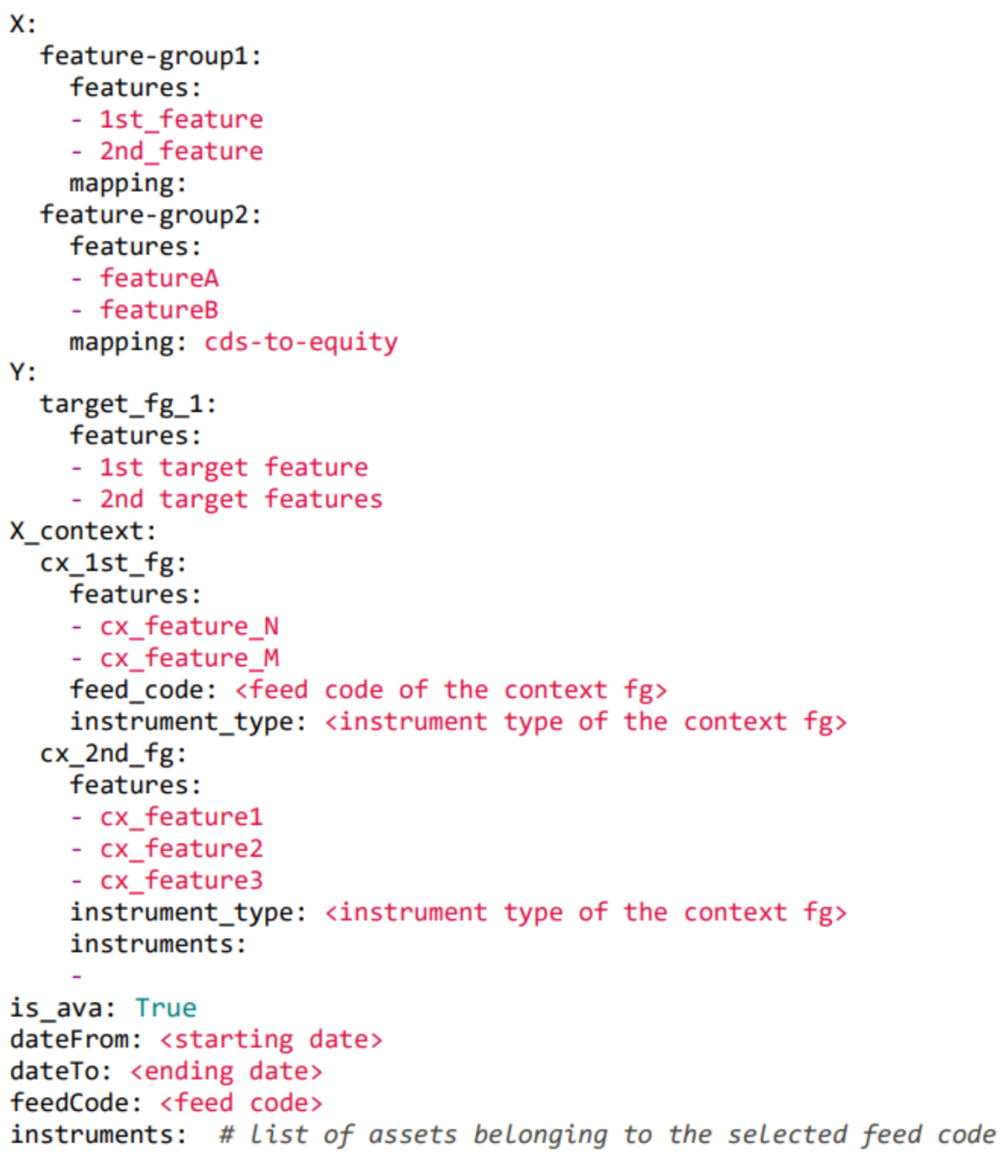
- **Investible universe:** identified by the Feed code, the investible universe defines the set of potentially tradable assets (Instruments).
- **Time period:** specifies the time period that the dataset should cover.
- **Dataset type:** determines whether the dataset to be generated should be of the AvA (Asset vs Asset) type or not.
- **Instrument related Features:** they specify the "instrument-related" Features that are desired to be included in the dataset.
- **Context Features:** they specify the "context" Features that are desired to be included in the dataset.

  They can also be specified in this section the "instrument-related" Features that are intended to be used as context Features (identified as "custom context Features"). In this case it is necessary to specify which Instruments, associated with the Feature Group, to use as a context. This choice is managed by Talos in step 6 (context assets selection).
- **Target:** indicates the target Features in the case of training datasets.
- **Included Instruments:** provides the list of assets belonging to the selected Feed.

For the instrument-related Features, it is possible to include the "mapping" property for each selected Feature Group in the Genome. This property is employed when you want to incorporate Features of different instrument types respect to the one used for Instruments identified by the "instruments" key inside the Genome.

The mapping relies on the structure of the "sn_instruments_attributes" table (whose configuration is outlined in the Django models in appendix B) to establish a link between the two types of Instruments. Therefore, the mapping value must coincide with the "attribute_code" of the mentioned table.

This information is input into a .yaml file, representing the DNA and identity of the dataset to be generated. This file is called the dataset's Genome, and once it's ready, it is possible to generate the actual dataset. The structure of the output .yaml file follows the scheme described in the figure 4.4, but it can become a very long file, full of Features and Instruments.



```yaml
X:
  feature-group1:
    features:
    - 1st_feature
    - 2nd_feature
    mapping:
  feature-group2:
    features:
    - featureA
    - featureB
    mapping: cds-to-equity
Y:
  target_fg_1:
    features:
    - 1st target feature
    - 2nd target features
X_context:
  cx_1st_fg:
    features:
    - cx_feature_N
    - cx_feature_M
    feed_code: <feed code of the context fg>
    instrument_type: <instrument type of the context fg>
  cx_2nd_fg:
    features:
    - cx_feature1
    - cx_feature2
    - cx_feature3
    instrument_type: <instrument type of the context fg>
    instruments:
    -

is_ava: True
dateFrom: <starting date>
dateTo: <ending date>
feedCode: <feed code>
instruments:  # list of assets belonging to the selected feed code
```

Figure 4.4: Genome structure in a .yaml file

If intending to generate an ordinal dataset instead of AvA, the "is_ava" key can be removed from the Genome. To create the Genome of a test dataset, as we mentioned in the previous section, it is essential to specify a time period of just one day within the .yaml file.

4.3 Dataset Genome generation overview

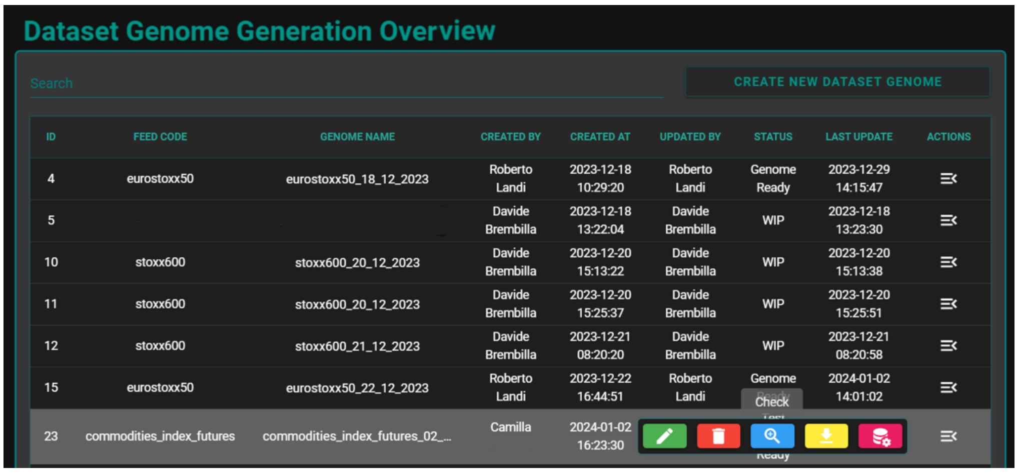
To keep track of the data for each Genome created by Axyon AI users, a new table has been created in the company's database. The generation of this table was made possible through a migration carried out via the Django Model contained in the "byzantium" app, as shown below in 4.1. The most important column is represented by the "preview" attribute, which is designed to contain all the properties of the Genome described in the section 4.2.

```python
class TalosYaml(models.Model):
    id = models.AutoField(primary_key=True, null=False)
    feed_code = models.CharField(max_length=255, null=False)
    name = models.CharField(max_length=255, null=False)
    created_by = models.CharField(max_length=255, null=False)
    updated_by = models.CharField(max_length=255, null=True)
    status = models.CharField(max_length=255, null=False)
    created_at = models.DateTimeField(null=False)
    last_update = models.DateTimeField(null=True)
    description = models.TextField(null=True)
    preview = models.TextField(null=True)
    class Meta:
        managed = True
        db_table = "talos_yaml"
```

Listing 4.1: Django Model in Python - bizantium directory

The data from the "talos_yaml" table is requested by the frontend in the "index.vue" page during the retrieval phase through a GET HTTP request, performed using the Axios library. The backend app simply receives the request and, through the "yaml_list" view, executes a read query on the table, returning the data to the frontend as a JsonResponse.

At this point, similar to what was done in the Talos monitoring page, the data is displayed to the user within a Vuetify v-data-table component, which is customized to adapt to the specific use case.



Dataset Genome Generation Overview

Search

CREATE NEW DATASET GENOME

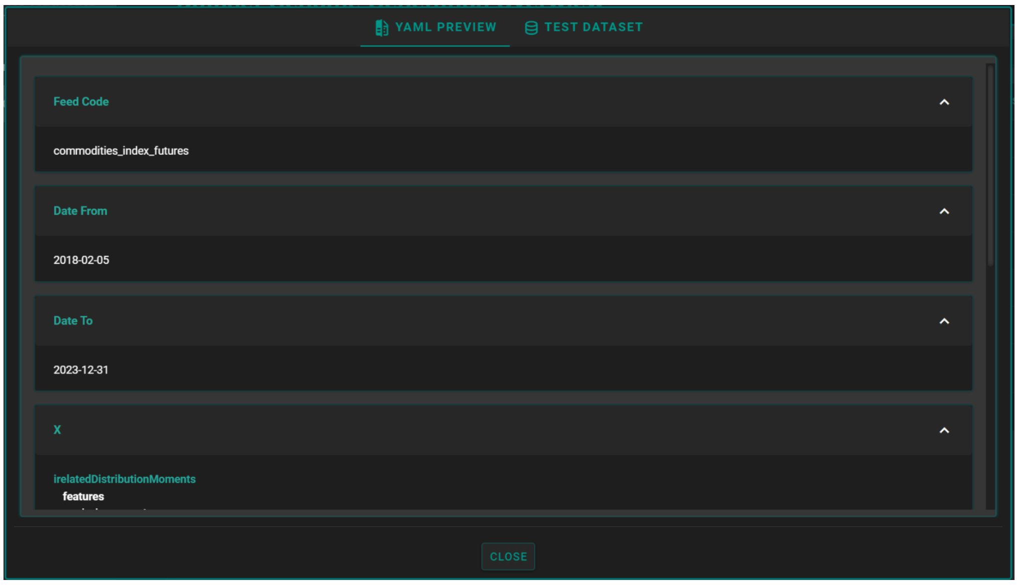
ID	FEED CODE	GENOME NAME	CREATED BY	CREATED AT	UPDATED BY	STATUS	LAST UPDATE	ACTIONS
4	eurostoxx50	eurostoxx50_18_12_2023	Roberto Landi	2023-12-18 10:29:20	Roberto Landi	Genome Ready	2023-12-29 14:15:47	
5			Davide Brembilla	2023-12-18 13:22:04	Davide Brembilla	WIP	2023-12-18 13:23:30	
10	stoxx600	stoxx600_20_12_2023	Davide Brembilla	2023-12-20 15:13:22	Davide Brembilla	WIP	2023-12-20 15:13:38	
11	stoxx600	stoxx600_20_12_2023	Davide Brembilla	2023-12-20 15:25:37	Davide Brembilla	WIP	2023-12-20 15:25:51	
12	stoxx600	stoxx600_21_12_2023	Davide Brembilla	2023-12-21 08:20:20	Davide Brembilla	WIP	2023-12-21 08:20:58	
15	eurostoxx50	eurostoxx50_22_12_2023	Roberto Landi	2023-12-22 16:44:51	Roberto Landi	Genome Ready	2024-01-02 14:01:02	
23	commodities_index_futures	commodities_index_futures_02_...	Camilla	2024-01-02 16:23:30		Ready		

Figure 4.5: dataset Genome generation overview table

The "actions" column in the table shown to the user is adjusted based on the Genome status, displaying various action possibilities.

- **WIP (Work in Progress):** the Genome has been created and is present in the database table, but it is not yet complete (there are some mandatory properties). The possible actions are only "Edit" and "Delete".
- **Genome Ready:** the Genome has been created, and all its fundamental properties have been inserted. Therefore, all actions are available: "Edit", "Delete", "Download", "Check" and "Generate Test Dataset".
- **Generating Test Dataset:** the process of generating the test dataset has been initiated but is not yet completed.
- **Test Dataset Ready:** the test dataset is ready, allowing all actions. In particular, with "Check" it is possible to view the test dataset preview.
- **Test Dataset Failed:** the generation of the test dataset was unsuccessful, but all actions are still possible. With "Check" it will also be possible to view any error messages collected during the generation process.

The possible action "Check", which we will also see better in the next paragraphs, allows for a preview of the Genome created up to that point. In practice, as shown in the figure 4.6, it provides the same basic functionalities as the recap step that also concludes the Genome generation process. However, in this case it serves the user to observe their own work, as well as the work of other users, and potentially decide to modify it.



Feed Code

commodities_index_futures

Date From

2018-02-05

Date To

2023-12-31

X

irelatedDistributionMoments
features

Figure 4.6: Genome preview with "Check" action

By selecting the "CREATE NEW DATASET GENOME" button (a v-btn component) located at the top right of the table in the figure 4.5, the user is directed to the second page of byzantium for creating a new Genome.

4.4 Genome generation process

The creation of a new Genome, as well as the process of editing an existing Genome within the table shown in the previous section, is entirely managed at the frontend level by the "create-genome.vue" page. This page is responsible for presenting all available options to the user regarding the information contained in the Genome structure, as detailed in 4.2. From a graphical perspective, but not only, the page can be divided into three separate but closely interconnected areas:

- The dynamic Genome recap, always present on the right side of the screen, allows the user to monitor their choices in real-time. This functionality is handled by an external component to the page, "Recap.vue," which enables the user to proceed or go back in the Genome generation process.

  It is called dynamic because it's composed of various panels that automatically open and close based on the progress status of the process. This allows the user to see the relevant section of the process at any given time.

  This component expands and fills the entire screen in the final phase of the process, when the user is prompted, in step 8, to review the choices made and confirm the completion of the process.

- The display and control of steps at the top of the screen are managed directly within the page through a Vuetify v-stepper component, keeping track of the progress status of the process. This section, during Genome creation, simply serves a display function. However, during the editing of an existing Genome with a status of "Genome Ready", it also serves a control function. This allows the user to directly select the desired step without following the predefined order of steps.

- The selection of properties to include in the Genome, in the central part of the screen, is managed by components external to the page that alternate depending on the step the user chooses to be in. For this reason, an ad hoc component was created to manage every single step listed in the mockup section 4.1.

```
import FeedSelection from
    '/components/byzantium/FeedSelection.vue'
import FeatureSelection from
    '/components/byzantium/FeatureSelection.vue'
import Recap from '/components/byzantium/Recap.vue'
import Mapping from '/components/byzantium/Mapping.vue'
import AssetsSelection from
    '/components/byzantium/AssetsSelection.vue'
```

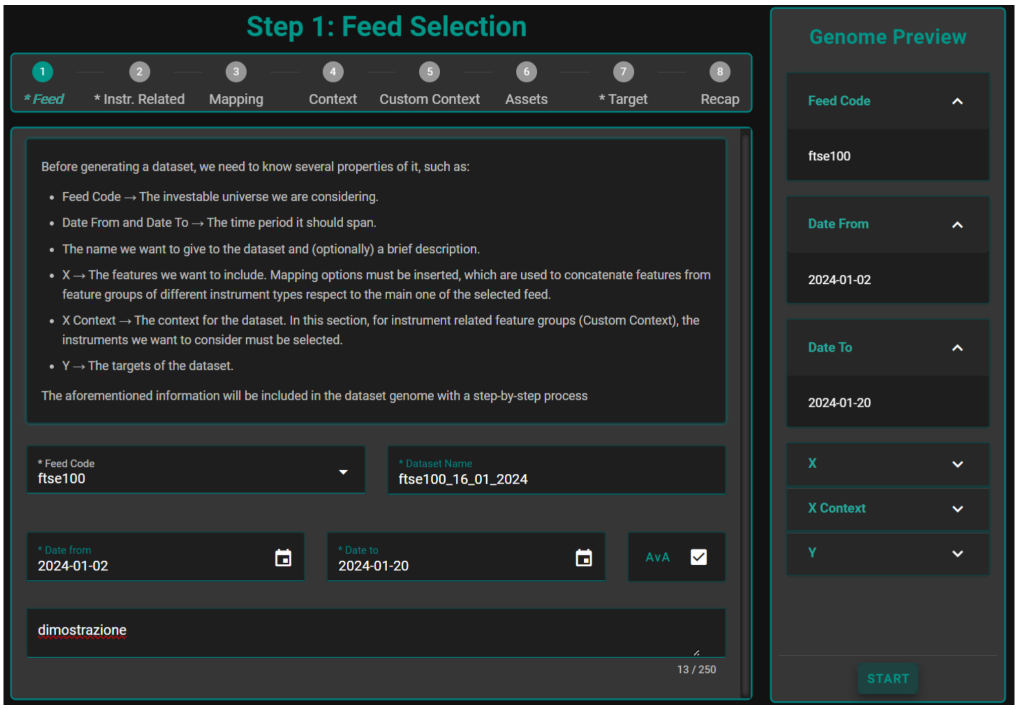
Listing 4.2: create-genome.vue imports

When transitioning from one step to another, whether during the creation phase via the recap panel or through the stepper during the editing phase, a function in the frontend page sends a POST HTTP request to the backend using the Axios library. This request is processed by the Django views "yaml_insert" and "yaml_edit", which execute a query on the "talos_yaml" table in the database to update or create the new Genome.

The transition from one step to another may not always be possible as some properties are mandatory to create or modify a Genome. Specifically, the following are always required: the Feed code of the investible universe, the name, the time period the dataset must cover, at least one instrument-related Feature, and at least one target Feature. Mandatory fields and steps are marked with an asterisk.

4.4.1 Step 1: Feed selection

In the first step of generating the dataset Genome, in addition to a brief description of the process, the main fields such as the Feed, the name, the time period, and optionally, a description of the dataset the user wants to create are required. This first step is visible in the figure 4.7



Step 1: Feed Selection

1 * Feed — 2 * Instr. Related — 3 Mapping — 4 Context — 5 Custom Context — 6 Assets — 7 * Target — 8 Recap

Before generating a dataset, we need to know several properties of it, such as:

- Feed Code → The investable universe we are considering.
- Date From and Date To → The time period it should span.
- The name we want to give to the dataset and (optionally) a brief description.
- X → The features we want to include. Mapping options must be inserted, which are used to concatenate features from feature groups of different instrument types respect to the main one of the selected feed.
- X Context → The context for the dataset. In this section, for instrument related feature groups (Custom Context), the instruments we want to consider must be selected.
- Y → The targets of the dataset.

The aforementioned information will be included in the dataset genome with a step-by-step process

* Feed Code	ftse100
* Dataset Name	ftse100_16_01_2024
* Date from	2024-01-02
* Date to	2024-01-20
AvA	✓
	dimostrazione
	13 / 250

Genome Preview

Feed Code	ftse100
Date From	2024-01-02
Date To	2024-01-20
X	
X Context	
Y	

Figure 4.7: step 1 - create Genome

If one of the required fields is missing during the transition to the next step, an error message is displayed indicating which field has not been filled in, preventing the user from proceeding further. The same would happen during the editing phase if one of the mandatory fields were eliminated.

The list of Feeds available for selection is determined through a GET HTTP request when fetching data by the "FeedSelection.vue" component.
On the backend side, this request is handled by the Django view "feeds_list", which queries the "sn_datasmith_feeds_in_feature_groups" table, just as it was done for the Feeds monitoring page.

4.4.2 Step 2: instrument related Features selection

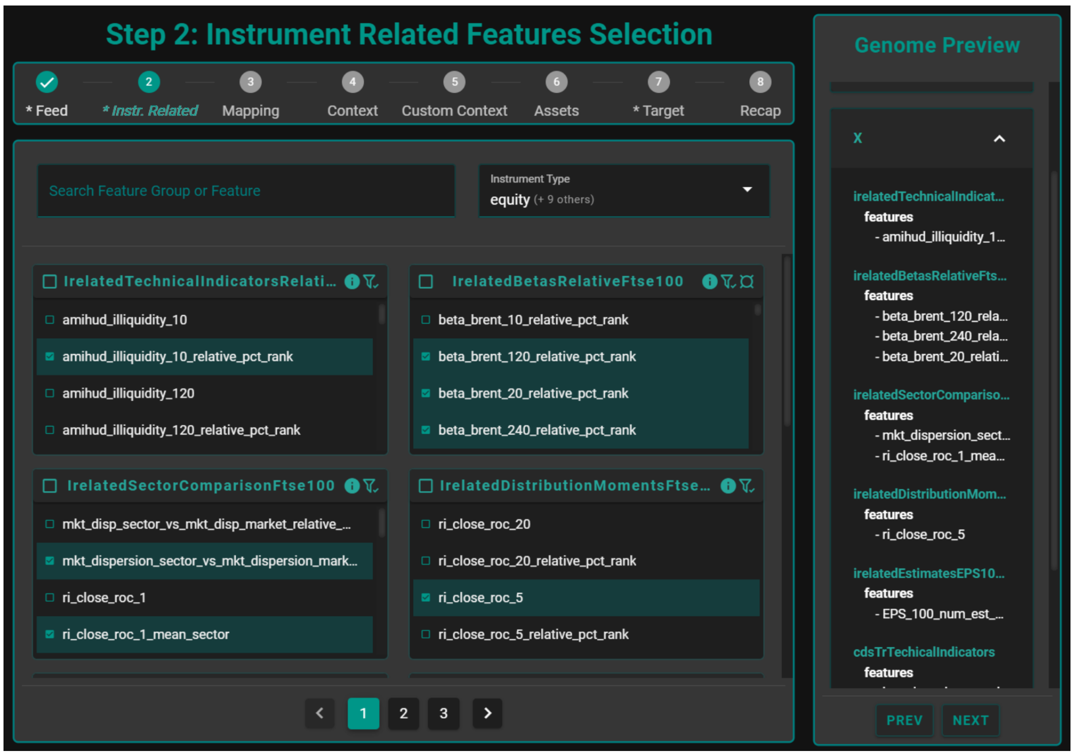
In the second step of generating the dataset Genome, users are asked to input instrument-related Features by selecting from a series of Feature Groups represented through vertically scrolling cards. The data within these cards is retrieved during the fetch of the "FeatureSelection.vue" component through a GET HTTP request. On the backend, this request triggers the Django view "feature_selection," which in turn executes a query on the "sn_datasmith_feature_groups" table to obtain the desired Feature Groups.

In this case, only the Feature Groups with the same Feed code as the one of the Feed chosen in step 1 or with null Feed code are selected.
Subsequently, utilizing the Datasmith codebase, all features belonging to the individual Feature Groups are obtained. This information is then returned to the frontend via JsonResponse from the Django view found below 4.3.

```python
@api_view(["GET"])
def feature_selection(request) -> JsonResponse:
    """Get the features given the input parameters.
    Args:
        request: request containing the parameters featureType and, optionally, currentFGs and feedCode

    Returns:
        JsonResponse: a dictionary containing the selected feature groups
    """
    feature_type = request.query_params.get("featureType")
    current_fgs = request.query_params.get("currentFGs")

    if feature_type != "custom_context":
        feedCode = request.query_params.get("feedCode")
    else:
        feedCode = None
    feature_groups = get_feature_groups(feature_type, feedCode, current_fgs)

    fg_dir = os.path.dirname(features.__file__)
    fg_list = [
        file[:-3]
        for file in os.listdir(fg_dir)
        if file.endswith(".py") and file != "__init__.py"
    ]

    feature_groups = (
        feature_groups.query("fg_name_suffix␣in␣@fg_list")
        .apply(get_feature_group_description, axis=1)
        .to_dict("records")
    )
    return JsonResponse(feature_groups, safe=False)
```

Listing 4.3: feature_selection Django view

As can be seen in the figure 4.8, in the header of each card, along with the name of the Feature Group, there are also icons that provide useful information to users.


Step 2: Instrument Related Features Selection

Instrument Type
equity (+ 9 others)

IrelatedTechnicalIndicatorsRelati...
☐ amihud_illiquidity_10
☑ amihud_illiquidity_10_relative_pct_rank
☐ amihud_illiquidity_120
☐ amihud_illiquidity_120_relative_pct_rank

IrelatedBetasRelativeFtse100
☐ beta_brent_10_relative_pct_rank
☑ beta_brent_120_relative_pct_rank
☑ beta_brent_20_relative_pct_rank
☑ beta_brent_240_relative_pct_rank

IrelatedSectorComparisonFtse100
☐ mkt_disp_sector_vs_mkt_disp_market_relative_...
☑ mkt_dispersion_sector_vs_mkt_dispersion_mark...
☐ ri_close_roc_1
☑ ri_close_roc_1_mean_sector

IrelatedDistributionMomentsFtse...
☐ ri_close_roc_20
☐ ri_close_roc_20_relative_pct_rank
☑ ri_close_roc_5
☐ ri_close_roc_5_relative_pct_rank

Genome Preview

irelatedTechnicalIndicat...
features
- amihud_illiquidity_1...

irelatedBetasRelativeFts...
features
- beta_brent_120_rela...
- beta_brent_240_rela...
- beta_brent_20_relati...

irelatedSectorCompariso...
features
- mkt_dispersion_sect...
- ri_close_roc_1_mea...

irelatedDistributionMom...
features
- ri_close_roc_5

irelatedEstimatesEPS10...
features
- EPS_100_num_est_...

cdsTrTechicalIndicators
features

Figure 4.8: step 2 - instrument-related Features selection

In particular, hovering the mouse over the information icon allows users to read a description of the FG. The filter-shaped icon indicates that the Feature Group has the same associated Feed code selected for the Genome being created, while the generic currency symbol indicates that the Feature Group has conversion enabled, information retrieved from the Datasmith codebase.

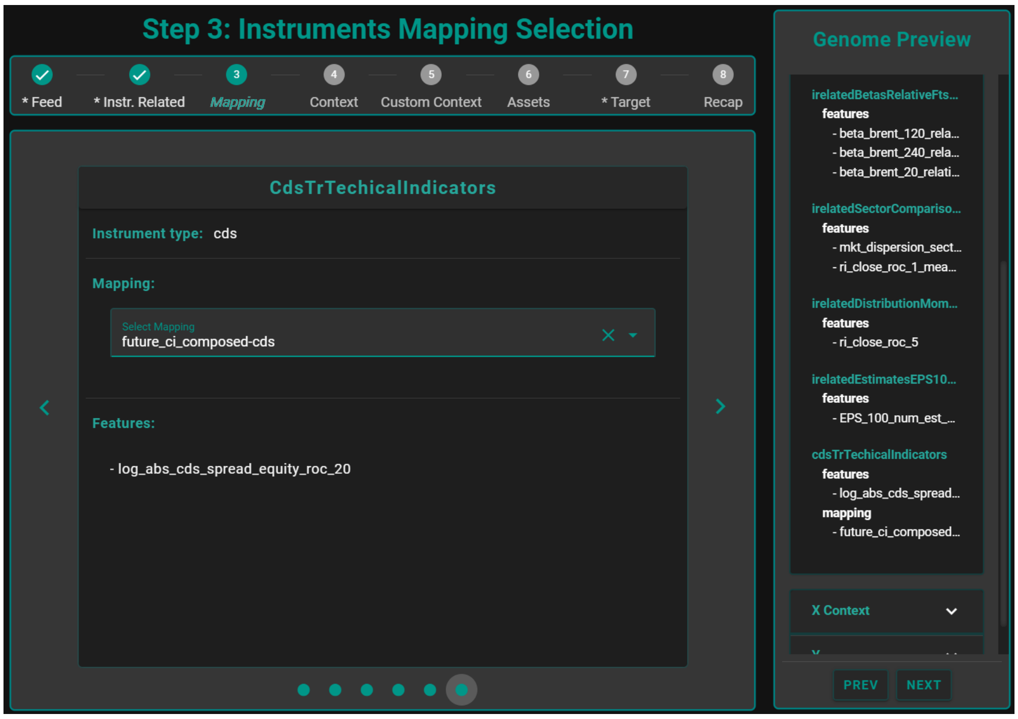
In some cases, there are many Feature Groups to choose from, so there are two filtering options available to simplify selection for users. One involves a text search field that filters the cards based on matches in the names of the Feature Groups and Features. The other is a filter for instrument type, allowing users to select one or more instrument types from those present in the Feature Groups. Each Feature Group indeed has an attribute for instrument type within the "sn_datasmith_feature_groups" table.
These filters allow searching among all available cards, not just those shown on the screen at that moment.

4.4.3 Step 3: instruments mapping selection

In the third step of generating a dataset's Genome, users are prompted to input the mapping for the Feature Groups selected in the previous step that do not have the same instrument type as the main instrument type of the chosen Feed code in step 1.

To achieve this, first, the main instrument types of the Feed (usually only one) are retrieved through an HTTP request managed by the "mapping" view, which operates on the "sn_feed_instrument" and "sn_instrument" tables of the Axyon AI database. Subsequently, only the Feature Groups with an instrument type different from those retrieved from the backend are shown to the user because they require a mapping operation.

Then, through a second HTTP request of type GET, the mapping options for all the Feature Groups displayed to the user are obtained via the Django view "mapping_options," which operates on the "sn_instrument_attribute" table, searching for a match with the instrument types of the FG. The structure of these tables can be observed in appendix B.



CdsTrTechicalIndicators

Instrument type: cds

Mapping:

Select Mapping
future_ci_composed-cds

Features:

- log_abs_cds_spread_equity_roc_20

irelatedBetasRelativeFts...
features
- beta_brent_120_rela...
- beta_brent_240_rela...
- beta_brent_20_relati...

irelatedSectorCompariso...
features
- mkt_dispersion_sect...
- ri_close_roc_1_mea...

irelatedDistributionMom...
features
- ri_close_roc_5

irelatedEstimatesEPS10...
features
- EPS_100_num_est_...

cdsTrTechicalIndicators
features
- log_abs_cds_spread...
mapping
- future_ci_composed...

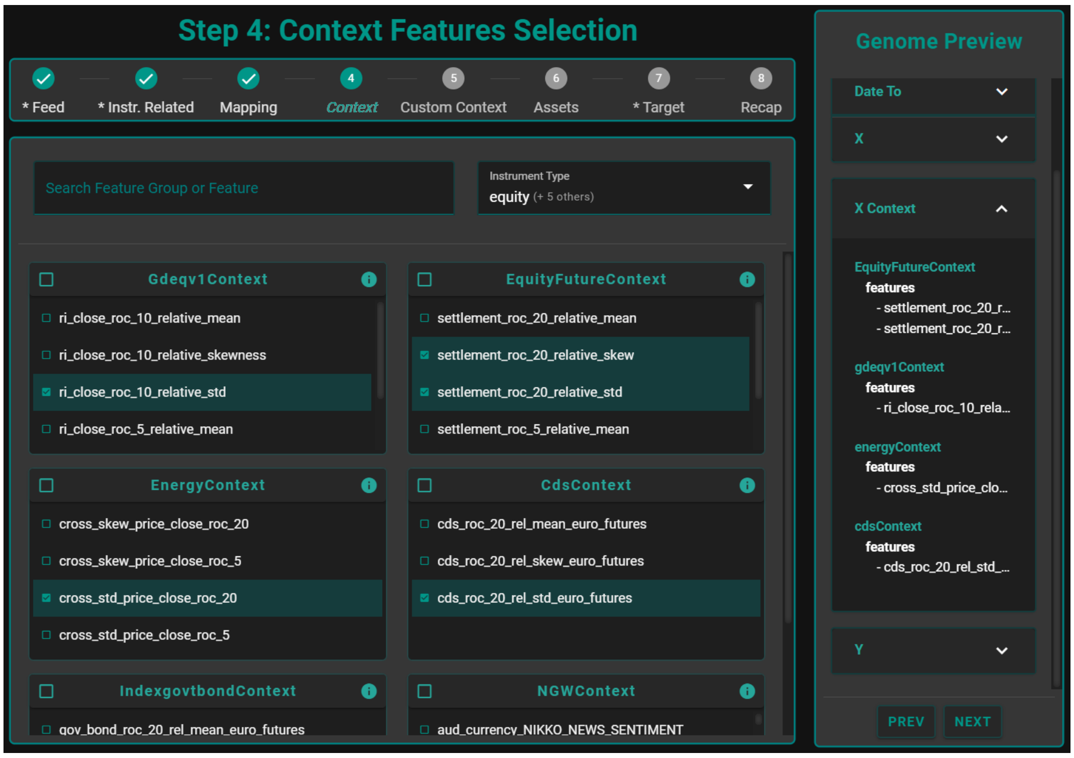
Figure 4.9: step 3 - Instruments mapping selection

From the user's perspective, as illustrated in the figure 4.9, the Feature Groups for which mapping is required are displayed one at a time through a card slider. This makes it clear how many Feature Groups require mapping. When a mapping option is selected in the selection component, the mapping key appears in the dynamic preview on the right side of the page under the name of the selected Feature Group.
If for a Feature Group that requires mapping there are no mapping options available, the user is shown a clear message inviting them to check if that Feature Group is necessary, offering them the possibility to remove it because it probably shouldn't have been inserted.

4.4.4 Step 4: context Features selection

In the fourth step of generating a dataset's Genome, users are prompted to input "context" type features by selecting from a series of Feature Groups represented through vertical-scrolling cards. A context Feature value changes for each date but remains the same for each Instrument on that specific date.

The data contained in the cards is retrieved during the fetch of the "FeatureSelection.vue" component through an HTTP GET request, similar to what occurred in step 2. The main difference is that, in this case, on the backend, only Feature Groups of type context are selected from the "sn_datasmith_feature_groups" table, as shown in the "feature_selection" view in 4.3.



Instrument Type
equity (+ 5 others)

Gdeqv1Context
- ☐ ri_close_roc_10_relative_mean
- ☐ ri_close_roc_10_relative_skewness
- ☑ ri_close_roc_10_relative_std
- ☐ ri_close_roc_5_relative_mean

EquityFutureContext
- ☐ settlement_roc_20_relative_mean
- ☑ settlement_roc_20_relative_skew
- ☑ settlement_roc_20_relative_std
- ☐ settlement_roc_5_relative_mean

EnergyContext
- ☐ cross_skew_price_close_roc_20
- ☐ cross_skew_price_close_roc_5
- ☑ cross_std_price_close_roc_20
- ☐ cross_std_price_close_roc_5

CdsContext
- ☐ cds_roc_20_rel_mean_euro_futures
- ☐ cds_roc_20_rel_skew_euro_futures
- ☑ cds_roc_20_rel_std_euro_futures

IndexgovtbondContext
- ☐ gov_bond_roc_20_rel_mean_euro_futures

NGWContext
- ☐ aud_currency_NIKKO_NEWS_SENTIMENT

EquityFutureContext
features
- settlement_roc_20_r...
- settlement_roc_20_r...

gdeqv1Context
features
- ri_close_roc_10_rela...

energyContext
features
- cross_std_price_clo...

cdsContext
features
- cds_roc_20_rel_std_...

Figure 4.10: step 4 - context Features selection

As clearly seen in the example shown in the figure 4.10, when there are fewer than ten Feature Groups, and therefore fewer than ten cards, they are not divided into separate pages, allowing for complete scrolling on a single page.

The Features, both in this step and in the other Feature selection steps, can be individually selected or the entire Feature Group can be chosen by clicking on the checkbox at the top left of the card.

4.4.5 Step 5: custom context Features selection

In the fifth step of generating a dataset's Genome, users are given the opportunity to include additional context Features by selecting from a series of Feature Groups represented, like in the previous steps, through vertical-scrolling cards.

The data contained within these cards is retrieved during the fetch of the "FeatureSelection.vue" component via an HTTP GET request, similar to what occurred in the previous step. The main difference is that, in this case, on the backend all Feature Groups of instrument-related type are selected from the "sn_datasmith_feature_groups" table, including those with a different Feed code than the one chosen for the Genome in step 1.

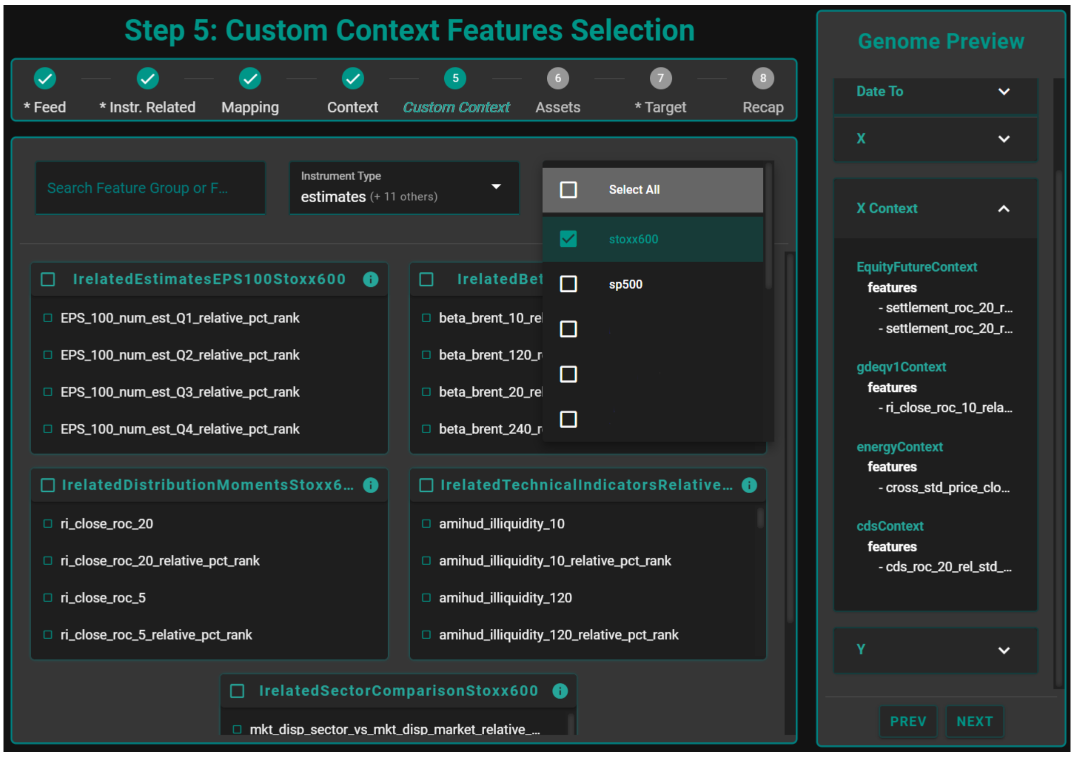

Step 5: Custom Context Features Selection
* Feed
* Instr. Related
Mapping
Context
Custom Context
Assets
* Target
Recap
Search Feature Group or F...
Instrument Type
estimates (+ 11 others)
Select All
stoxx600
sp500
IrelatedEstimatesEPS100Stoxx600
EPS_100_num_est_Q1_relative_pct_rank
EPS_100_num_est_Q2_relative_pct_rank
EPS_100_num_est_Q3_relative_pct_rank
EPS_100_num_est_Q4_relative_pct_rank
IrelatedBet
beta_brent_10_re
beta_brent_120_r
beta_brent_20_re
beta_brent_240_r
IrelatedDistributionMomentsStoxx6...
ri_close_roc_20
ri_close_roc_20_relative_pct_rank
ri_close_roc_5
ri_close_roc_5_relative_pct_rank
IrelatedTechnicalIndicatorsRelative...
amihud_illiquidity_10
amihud_illiquidity_10_relative_pct_rank
amihud_illiquidity_120
amihud_illiquidity_120_relative_pct_rank
IrelatedSectorComparisonStoxx600
mkt_disp_sector_vs_mkt_disp_market_relative_...
Genome Preview
Date To
X
X Context
EquityFutureContext
features
- settlement_roc_20_r...
- settlement_roc_20_r...
gdeqv1Context
features
- ri_close_roc_10_rela...
energyContext
features
- cross_std_price_clo...
cdsContext
features
- cds_roc_20_rel_std_...
Y
PREV
NEXT

Figure 4.11: step 5 - custom context Features selection

As shown in the example in the figure 4.11, in this step, there are no longer Feature Groups with a null Feed code or one that matches the Feed code associated with the Genome selection. Therefore, an additional filtering mode has been introduced to filter the cards by Feed code, allowing users to choose from those associated with the available Feature Groups in the cards. This significantly helps users because, by selecting all the Feed Codes and not just one, the options for selection are broader.

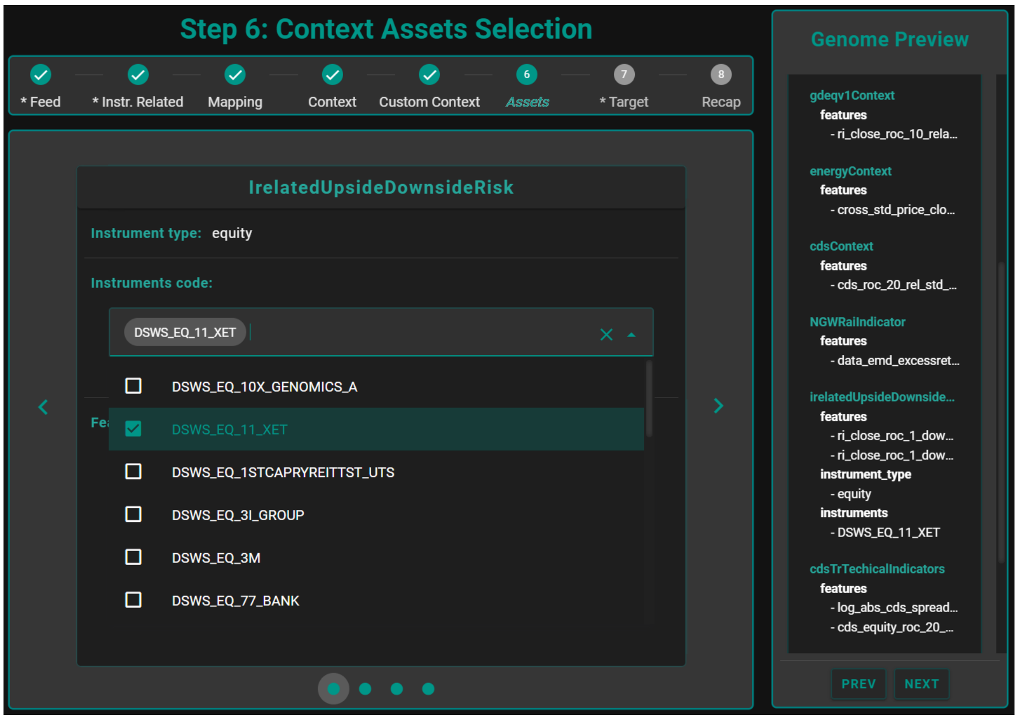
The Features selected in this step are still part of the Genome's contextual Features, so they will be included in the "X Context" section of the dynamic preview on the right.
They are called "custom" because they originate as instrument-related Features and thus, to be used as context, they require the selection of individual assets to be included in the context in the next step.

4.4.6 Step 6: context assets selection

In the sixth step of generating a dataset's Genome, after selecting the custom context Features among the Feature Groups of instrument-related type, users are prompted to input assets to include in the dataset for each chosen Feature Group in the previous step. Users are then presented with a series of cards within a slider, one for each Feature Group selected in step 5. This step is managed by the Vue component "AssetsSeletion.vue".

To determine the assets to choose for each card, first, the Instrument types of each Feature Group are retrieved. Subsequently, an HTTP GET request is made to the backend, providing the obtained list of Instrument types as a parameter.
The request is handled on the backend side by the Django view "assets_selection", which retrieves from the "sn_instrument" table all the codes of the Instruments associated with the received Instrument types as a parameter, grouping them by "instrument type" to ensure a list of codes of the Instruments (assets) for each Instrument type associated with the Feature Groups.

These pieces of information are then returned to the frontend component "AssetsSelection.vue", which provides them to users through a Vuetify component "v-autocomplete", making it easy for users to input assets which are to choose from hundreds of options.



IrelatedUpsideDownsideRisk

Instrument type: equity

Instruments code:

DSWS_EQ_11_XET

- ☐ DSWS_EQ_10X_GENOMICS_A
- ☑ DSWS_EQ_11_XET
- ☐ DSWS_EQ_1STCAPRYREITTST_UTS
- ☐ DSWS_EQ_3I_GROUP
- ☐ DSWS_EQ_3M
- ☐ DSWS_EQ_77_BANK

```
gdeqv1Context
  features
    - ri_close_roc_10_rela...
energyContext
  features
    - cross_std_price_clo...
cdsContext
  features
    - cds_roc_20_rel_std_...
NGWRailIndicator
  features
    - data_emd_excessret...
irelatedUpsideDownside...
  features
    - ri_close_roc_1_dow...
    - ri_close_roc_1_dow...
  instrument_type
    - equity
  instruments
    - DSWS_EQ_11_XET
cdsTrTechicalIndicators
  features
    - log_abs_cds_spread...
    - cds_equity_roc_20_...
```

Figure 4.12: step 6 - context assets selection

As highlighted in figure 4.12, once a user selects an asset, it appears in the dynamic display on the right within the desired Feature Group under the key "instruments". The same happens for all the Feature Groups for which the user needs to input assets, which in the example are 4, as indicated by the dots at the bottom of the slider.

4.4.7 Step 7: target selection

In the seventh step of generating the Genome of a dataset, users are prompted to input the dataset's target by selecting from a series of Feature Groups represented, as in the previous steps, by vertically scrolling cards.

The data within these cards is retrieved during the fetch of the "FeatureSelection.vue" component through an HTTP GET request, similar to what occurred in the previous steps. The main difference here is that, on the backend, all instrument-related Feature Groups from the "sn_datasmith_feature_groups" table are selected, provided they have the "is_target" attribute set to 1 and a Feed code that is either null or matches the one corresponding to the chosen Genome's Feed.

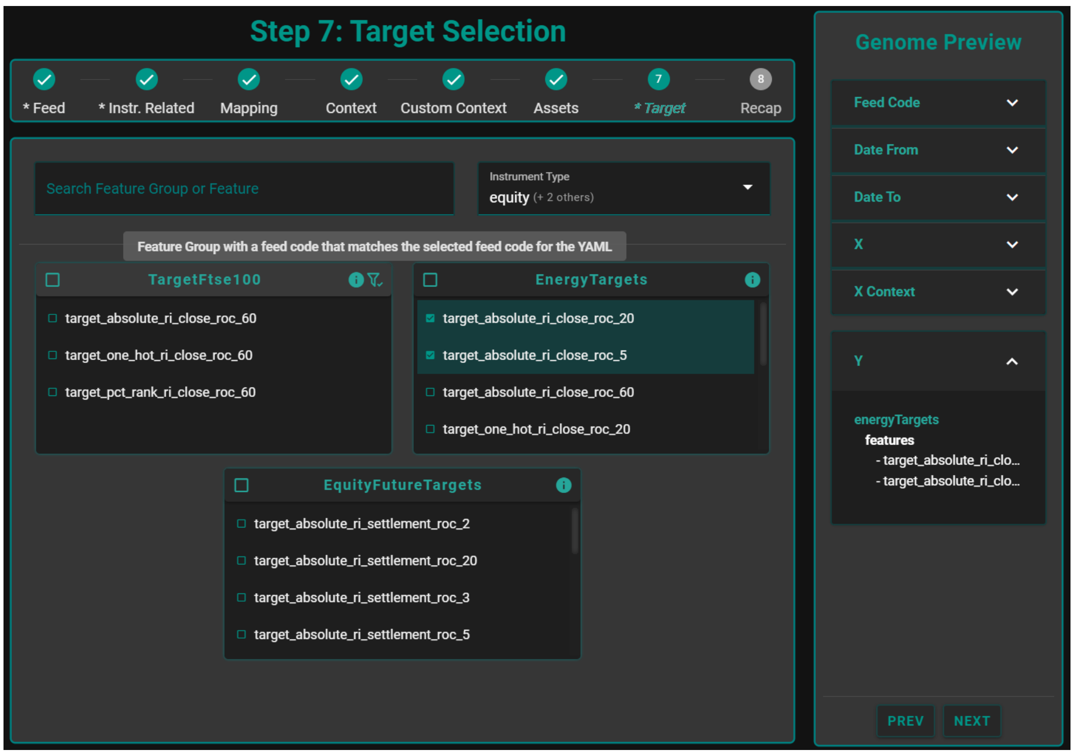

Step 7: Target Selection
* Feed
* Instr. Related
Mapping
Context
Custom Context
Assets
7
* Target
8
Recap
Search Feature Group or Feature
Instrument Type
equity (+ 2 others)
Feature Group with a feed code that matches the selected feed code for the YAML
TargetFtse100
target_absolute_ri_close_roc_60
target_one_hot_ri_close_roc_60
target_pct_rank_ri_close_roc_60
EnergyTargets
target_absolute_ri_close_roc_20
target_absolute_ri_close_roc_5
target_absolute_ri_close_roc_60
target_one_hot_ri_close_roc_20
EquityFutureTargets
target_absolute_ri_settlement_roc_2
target_absolute_ri_settlement_roc_20
target_absolute_ri_settlement_roc_3
target_absolute_ri_settlement_roc_5
Genome Preview
Feed Code
Date From
Date To
X
X Context
Y
energyTargets
features
- target_absolute_ri_clo...
- target_absolute_ri_clo...
PREV
NEXT

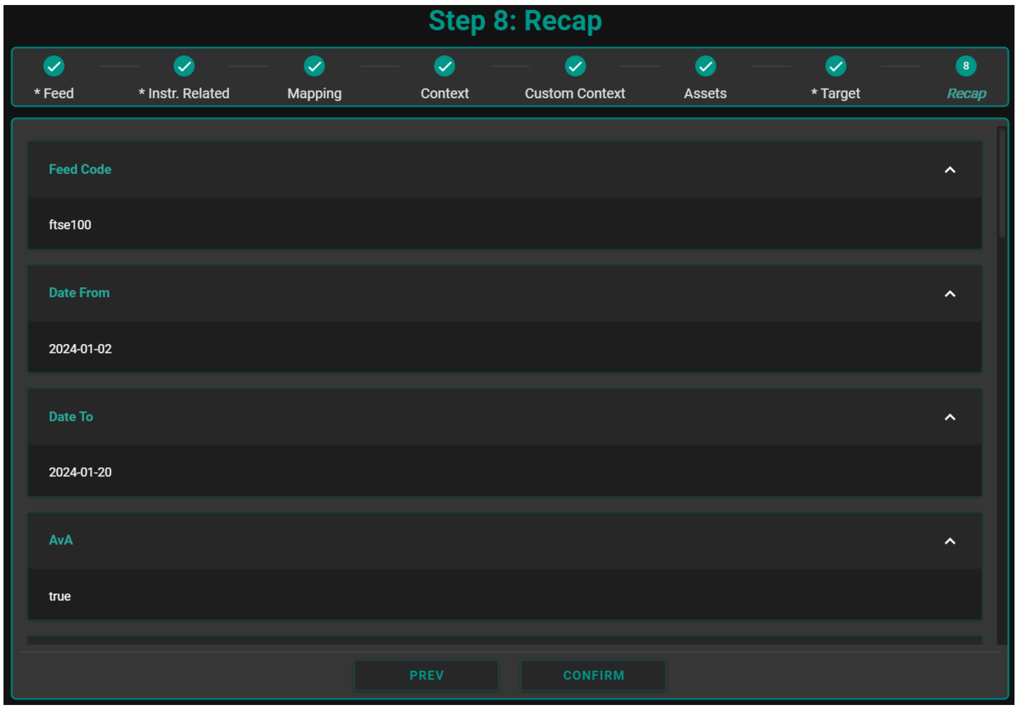
Figure 4.13: step 7 - target selection

As can be seen in the example figure 4.13, the available Feature Groups for selecting target Features are limited compared to previous steps, therefore there is no longer pagination or a Feed code filter, which would be unnecessary.

To further clarify, when a Feature is selected, the "FeatureSelection.vue" component sends the selection data to the page containing the component, in this case "create-genome.vue". It is then the responsibility of the main page to handle the Feature selection by inserting it into the "yaml" variable, which contains all the properties of the Genome.

4.4.8 Step 8: recap

In the final step of the Axion AI dataset Genome generation process, a presentation of the dynamic recap that appears in all steps is provided, but in a more comprehensive and detailed manner. This includes the panel regarding the dataset type (AvA or otherwise) and the list of Instruments associated with the selected Feed for dataset generation through the Genome. In this step, users can confirm the completion of the process or go back through the various steps to make any necessary changes.



Feed Code

ftse100

Date From

2024-01-02

Date To

2024-01-20

AvA

true

Figure 4.14: step 8 - recap

By selecting the confirmation option, the user is instead redirected to the management page "index.vue" described in section 4.3.
Once users return to the "dataset Genome generation overview" page, they have the option to modify an existing Genome, including one that has just been confirmed. In this case, on the "create-genome.vue" page, there is a "mounted()" section responsible for retrieving data from the "talos_yaml" table. This is necessary to display the previously selected properties and, if needed, make changes.

4.5 Test dataset generation

As illustrated earlier in section 4.3, on the page managed by "index.vue" relating to the overview of Dataset Genome generation, users can manage each Genome created. Specifically, various actions can be performed on the Genome based on its current status.
When the Genome's status is "Genome Ready," users can initiate the test dataset generation process. Similarly to the other steps, user selection triggers an HTTP request to the backend of the application, containing the identifying code of the Genome for which the dataset is to be generated. This request is handled on the backend by the Django view "test_dataset", which retrieves all necessary data from the "talos_yaml" table, including the Genome's structure. Subsequently, as evident in the view's code 4.4, two separate folders are created: one to contain the test dataset in parquet format and another to contain the .txt file tracking any errors during the dataset generation process.

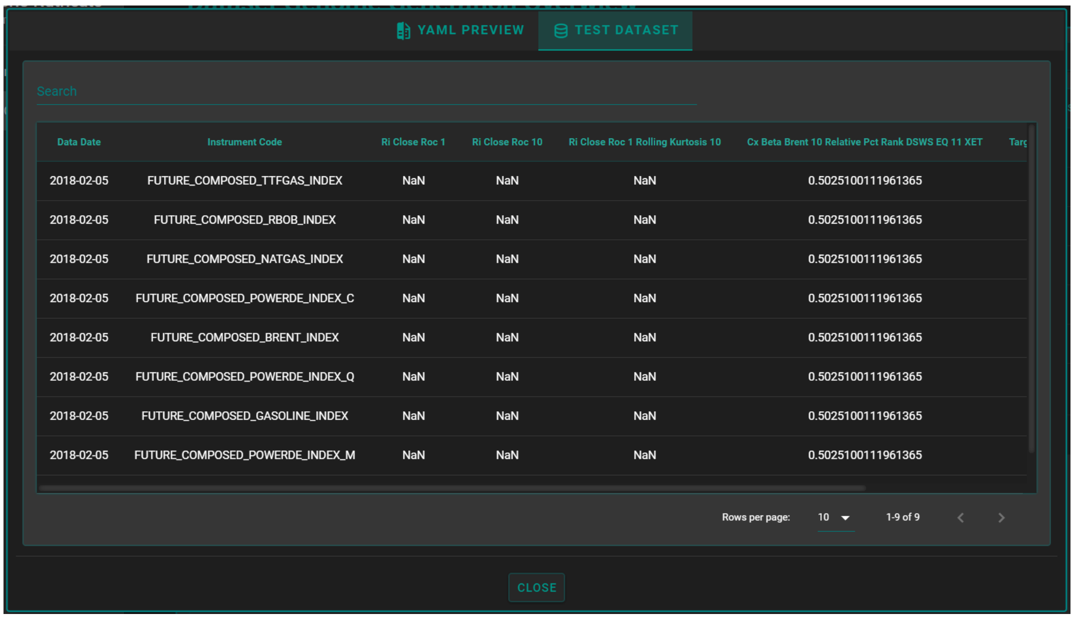
```python
def test_dataset(request) -> JsonResponse:
    id = request.query_params.get("id")
    row = get_yaml_from_id(id)
    preview_json = json.loads(row.preview)
    preview_json["dateTo"] = preview_json["dateFrom"]
    base_folder = "/home/axyon/talos/backend/outputs/"
    output_folder = os.path.join(base_folder, row.feed_code, "logs")
    output_file = os.path.join(output_folder, row.name + ".txt")
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)
    df = None
    edit_talos_yaml_status(id, "Generating Test Dataset")
    try:
        with open(output_file, "w") as log_file, contextlib.redirect_stdout(
            log_file
        ), contextlib.redirect_stderr(log_file):
            df = generate_dataset(preview_json, False)
    except Exception as e:
        with open(output_file, "a") as log_file:
            log_file.write(f"Error during generate_dataset: {str(e)}\n")
        edit_talos_yaml_status(id, "Test Dataset Failed")
    if isinstance(df, DataFrame):
        output_folder = os.path.join(base_folder, row.feed_code, "test_datasets")
        if not os.path.exists(output_folder):
            os.makedirs(output_folder)
        output_file = os.path.join(output_folder, row.name + ".parquet")
        df.to_parquet(output_file)
        edit_talos_yaml_status(id, "Test Dataset Ready")
    return JsonResponse(id, safe=False)
```

Listing 4.4: test_dataset Django view

The test dataset generation process utilizes the entrypoint provided by the Datasmith "generate_dataset" command, which receives the Genome's structure as a parameter and accesses the AWS SageMaker Feature Store directly to retrieve all data related to the Features included in the Genome. To accomplish this, Talos has access to Datasmith's codebase to execute the dataset generation command, as was explained in 2.2.

This process returns the dataset in dataframe format, which is then converted to parquet format. Finally, based on the success or failure of the dataset generation process, the Genome's status is updated to either "Test Dataset Ready" or "Test Dataset Failed".
At this point, in the overview screen of Dataset Genome generation, the Genome's status will appear with one of these two options.

Through the "Check" action, the files generated in the two created folders are retrieved via an HTTP request, and users will be able to explore the test dataset in tabular format in the event of a successful generation process, or read the error or warning messages generated by the process in case of failure, as shown in the figure 4.15.



Data Date	Instrument Code	Ri Close Roc 1	Ri Close Roc 10	Ri Close Roc 1 Rolling Kurtosis 10	Cx Beta Brent 10 Relative Pct Rank DSWS EQ 11 XET	Targ
2018-02-05	FUTURE_COMPOSED_TTFGAS_INDEX	NaN	NaN	NaN	0.5025100111961365	
2018-02-05	FUTURE_COMPOSED_RBOB_INDEX	NaN	NaN	NaN	0.5025100111961365	
2018-02-05	FUTURE_COMPOSED_NATGAS_INDEX	NaN	NaN	NaN	0.5025100111961365	
2018-02-05	FUTURE_COMPOSED_POWERDE_INDEX_C	NaN	NaN	NaN	0.5025100111961365	
2018-02-05	FUTURE_COMPOSED_BRENT_INDEX	NaN	NaN	NaN	0.5025100111961365	
2018-02-05	FUTURE_COMPOSED_POWERDE_INDEX_Q	NaN	NaN	NaN	0.5025100111961365	
2018-02-05	FUTURE_COMPOSED_GASOLINE_INDEX	NaN	NaN	NaN	0.5025100111961365	
2018-02-05	FUTURE_COMPOSED_POWERDE_INDEX_M	NaN	NaN	NaN	0.5025100111961365	

Figure 4.15: "Check" action - test dataset

Chapter 5

Achieved results

At the end of the internship and the drafting of the thesis, it is advisable to conduct a thorough analysis of the work done and the results achieved. This analysis not only includes the aspects of the web application implemented personally but also the functionalities, only briefly mentioned in the introductory chapter, that have been developed in collaboration with the Axyon AI team and to which I have also made a personal contribution.

5.1 Evaluation of the proposed solutions

In the thesis work, the aspects managed by the Axyon team were briefly mentioned, which involve the LDAP (Google) integration to enable access to the web application for company users and the subsequent implementation of custom permissions for the use of Talos functionalities. This aspect is crucial in the development process as it allows only certain users to access the application fully, while others can only use some functionalities.
The solutions proposed for this application management have proven effective and currently allow for the control and monitoring of Talos usage. However, regarding the login graphical interface, there is still room for improvement to align it with other usage pages.

The implementation choices made at the beginning of the internship, concerning the frameworks used and the structure of the pages outlined in the application mockup, have proven to be effective overall. The two main frameworks utilized, Django and Nuxt, have ensured modularity and flexibility in developing Talos. The division of the project between creating Django apps on the backend and crafting individual pages or Vue components on the frontend has facilitated optimal and transparent organization. Thanks to the extensive documentation of the frameworks, I was able to progress quickly in their utilization, focusing immediately on the functionalities to be implemented.

The main challenges encountered during the implementation stemmed from the need to handle large volumes of data frequently requested via HTTP requests from the frontend to the backend, data that then had to be displayed to the user on the graphical interface promptly.
Regarding the number of requests, it emerged that often the requested data remained unchanged, or the same request was made repeatedly during the application's usage.

To address this issue, we adopted caching for certain URLs, as can also be observed in appendix B. In Django, it is indeed possible to implement URL caching using the "cache_page" decorator. This setup allows storing the entire view in the cache for each request, skipping the computation of the view if a cached version is available and valid. This solution has helped reduce the load on the application's backend and the number of database accesses, while keeping some views cached in memory for 15 minutes.

Regarding the management of the large amount of data to be displayed, this was particularly evident in the "FeatureSelection.vue" component, especially when the number of cards increased significantly, as in step 6, but not only.
To address this challenge, I explored several solutions before adopting the aforementioned pagination of Feature Groups. This approach allows the user to view ten items per page and navigate between pages to view more. It is an effective solution that has been shared within the team.

A proposed solution that still needs some improvement is the header of the card, where the name of the FG is displayed along with some informative icons for the user. There are cases where the names of Feature Groups exceed the available space, and the positioning of the icons still raises some concerns among users. Therefore, while using the application, the best solution to adopt in the future will be identified.

An adopted solution, which I consider effective but believe can be further improved upon personally, concerns the display of possible actions on an existing Genome in the Genome Overview screen managed by the "index.vue" page 4.3. The current solution is not the only one I have developed, but in collaboration with the Axyon AI Team, we have decided to evaluate user feedback on the use of Talos before making a final decision.

5.2 Comparison: initial objectives versus results

The initial goal of the project was to monitor and support the machine learning dataset generation process managed by Datasmith through the design and implementation of a web application. Specifically, the aim was to monitor the historical and live ingestion of Feature Groups into the AWS SageMaker Feature Store, making the information available and clear to all Datasmith users and facilitating the identification of any errors in data ingestion. Additionally, the objective was to enhance the efficiency of the dataset Genome generation process by allowing users to select and configure all necessary properties to create the final dataset, and providing a tool for verifying the dataset content to analyze financial time series within it.

These objectives were pursued and achieved through the Lygos and Byzantium phases, carefully outlined in the thesis document, leading to the development of a web application named Talos. This application integrates into Axyon's AI model development process, making the generation of datasets for machine learning more efficient.

Thanks to the implementation of Talos, every Datasmith user, whether they are a quant analyst, software engineer, or data engineer, now has the ability to easily browse the available Features in the Feature Store and, if necessary, initiate the creation of a dataset. Originally, creating a dataset required multiple corrective iterations on the Genome and, being a manual operation, was inevitably prone to numerous errors. This process has now significantly improved in terms of time efficiency and reducing the number of iterations required to obtain a machine learning dataset ready for AI model production or internal experiments, thus also resulting in an enhancement of financial time series analysis.

An integral part of the web application development, and therefore included in the project objectives, was the creation of backend functionality tests for Talos using the Django framework. Django tests consist of a series of automated procedures executed to verify that various components of the application, such as models and views, are implemented and functioning correctly.
Tests related to the Lygos phase have been implemented and integrated into the project's automated deployment pipeline, while the same process could not be completed for the Byzantium phase due to time constraints. This will be an area to focus on later.

Chapter 6

Conclusions & future prospects

In conclusion, the development of the Talos web application represents a significant advancement within the corporate context of Axyon AI, providing an intuitive and efficient user interface to streamline the dataset generation process for training machine learning models. However, despite the significant progress made so far with Talos, there are still multiple perspectives for development and improvement in the future, partly already mentioned previously. Some of the areas of interest for future development include:

- Improving the user experience by continuing to work on the graphical interface to make it even more intuitive, user-friendly, and in line with users' needs.
- Developing comprehensive and structured Django tests to ensure the proper functioning of the application over time, even with the addition of new functionalities.
- Implementing new functionalities to make Talos increasingly integrated with the evolution of the Datasmith project.

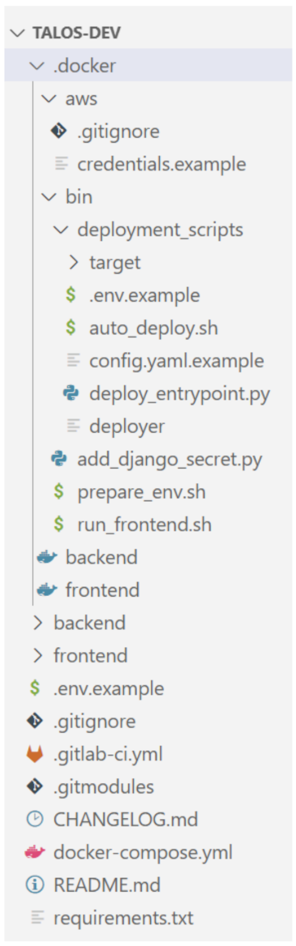
In particular, with the Axyon team, two possible new work phases have already been defined to be scheduled in the company's Product Backlog, which would represent a significant increase in value:

- **Constantinople:** making Talos capable of triggering the full dataset generation script, not just the test one, to complete the process managed by Datasmith.
- **Istanbul:** managing the ingestion of Features on the Feature Store in real-time and allowing Talos users to perform quality checks on the generated dataset viewing the output. The mentioned checks have been developed in Datasmith 1.2.

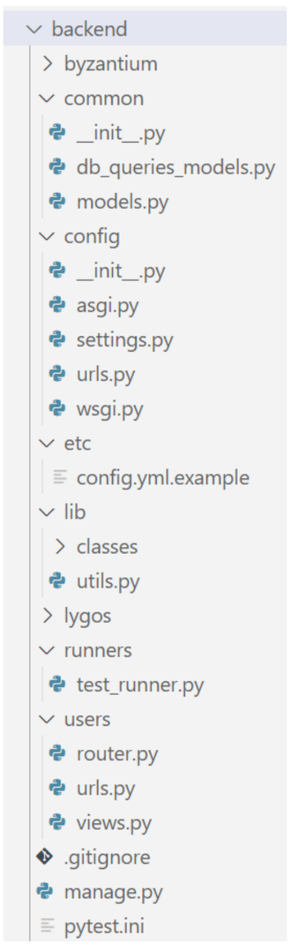
Ultimately, Talos represents an important resource for Axyon AI and has the potential to continue evolving and adapting to the changing needs of the industry. With ongoing commitment to development and optimization, Talos can make a significant contribution to Axyon AI's mission of providing innovative and high-quality solutions in the field of machine learning and financial data analysis.

Appendix A

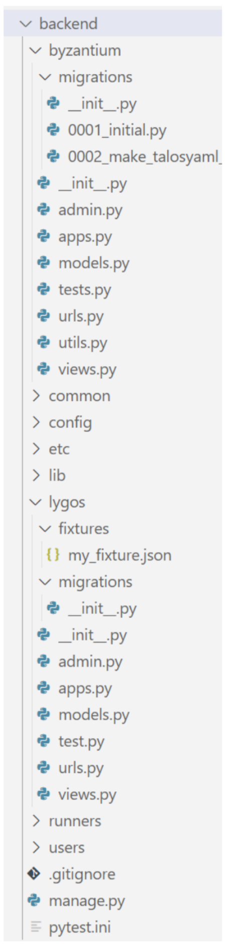
Codebase structure



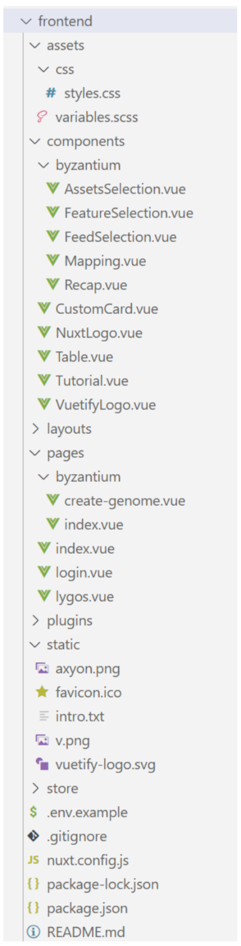
```
TALOS-DEV
  .docker
    aws
      .gitignore
      credentials.example
    bin
      deployment_scripts
        target
        .env.example
        auto_deploy.sh
        config.yaml.example
        deploy_entrypoint.py
        deployer
      add_django_secret.py
      prepare_env.sh
      run_frontend.sh
    backend
    frontend
  backend
  frontend
  .env.example
  .gitignore
  .gitlab-ci.yml
  .gitmodules
  CHANGELOG.md
  docker-compose.yml
  README.md
  requirements.txt
```



```
backend
  byzantium
  common
    __init__.py
    db_queries_models.py
    models.py
  config
    __init__.py
    asgi.py
    settings.py
    urls.py
    wsgi.py
  etc
    config.yml.example
  lib
    classes
    utils.py
  lygos
  runners
    test_runner.py
  users
    router.py
    urls.py
    views.py
  .gitignore
  manage.py
  pytest.ini
```



```
backend
  byzantium
    migrations
      __init__.py
      0001_initial.py
      0002_make_talosyaml_
    __init__.py
    admin.py
    apps.py
    models.py
    tests.py
    urls.py
    utils.py
    views.py
  common
  config
  etc
  lib
  lygos
    fixtures
      my_fixture.json
    migrations
      __init__.py
    __init__.py
    admin.py
    apps.py
    models.py
    test.py
    urls.py
    views.py
  runners
  users
  .gitignore
  manage.py
  pytest.ini
```



```
frontend
  assets
    css
      styles.css
    variables.scss
  components
    byzantium
      AssetsSelection.vue
      FeatureSelection.vue
      FeedSelection.vue
      Mapping.vue
      Recap.vue
    CustomCard.vue
    NuxtLogo.vue
    Table.vue
    Tutorial.vue
    VuetifyLogo.vue
  layouts
  pages
    byzantium
      create-genome.vue
      index.vue
    index.vue
    login.vue
    lygos.vue
  plugins
  static
    axyon.png
    favicon.ico
    intro.txt
    v.png
    vuetify-logo.svg
  store
  .env.example
  .gitignore
  nuxt.config.js
  package-lock.json
  package.json
  README.md
```

Appendix B

Django models

```python
from django.db import models

class SnDatasmithFeedsInFeatureGroups(models.Model):
    fg_name = models.OneToOneField('SnDatasmithFeatureGroups', models.DO_NOTHING, db_column='fg_name', primary_key=True)  # The composite primary key (fg_name, feed_code) found, that is not supported. The first column is selected.
    feed_code = models.CharField(max_length=255)

    class Meta:
        managed = False
        db_table = 'sn_datasmith_feeds_in_feature_groups'
        unique_together = (('fg_name', 'feed_code'),)


class SnDatasmithFeatureGroups(models.Model):
    fg_name_suffix = models.CharField(primary_key=True, max_length=255)
    instrument_type = models.CharField(max_length=255)
    description = models.CharField(max_length=128, blank=True, null=True)
    feature_group_path = models.CharField(max_length=255)
    fg_type = models.ForeignKey('SnDatasmithFeatureGroupTypes', models.DO_NOTHING, db_column='fg_type', blank=True, null=True)
    data_provider = models.CharField(max_length=255)
    feed_code = models.CharField(max_length=64, blank=True, null=True)
    is_target = models.IntegerField()
    complementary_instrument_types = models.CharField(max_length=500, blank=True, null=True)
    historical_ingestion_snapshot = models.DateTimeField(blank=True, null=True)
    historical_failed_rows = models.IntegerField(blank=True, null=True)
    lookback_window = models.IntegerField(blank=True, null=True)
    live_ingestion_snapshot = models.DateTimeField(blank=True,
        null=True)
    live_failed_rows = models.IntegerField(blank=True, null=True)
    ingestable = models.IntegerField()

    class Meta:
        managed = False
        db_table = 'sn_datasmith_feature_groups'


class SnDatasmithFeatureGroupTypes(models.Model):
    fg_type = models.CharField(primary_key=True, max_length=100)
    record_identifier = models.CharField(max_length=255)

    class Meta:
        managed = False
        db_table = 'sn_datasmith_feature_group_types'

class SnInstrument(models.Model):
    instrument_code = models.CharField(primary_key=True,
        max_length=128)
    instrument_type = models.CharField(max_length=32)
    currency_code = models.CharField(max_length=32)
    instrument_source = models.CharField(max_length=32,
        blank=True, null=True)
    dataprovider_main = models.CharField(max_length=128,
        blank=True, null=True)
    bloomberg_code = models.CharField(max_length=128)
    tr_code = models.CharField(max_length=128, blank=True,
        null=True)
    aa_code = models.CharField(max_length=128, blank=True,
        null=True)
    name = models.CharField(max_length=256)
    instrument_sector_code = models.CharField(max_length=128)
    instrument_subsector_code = models.CharField(max_length=45)
    asset_class_code = models.CharField(max_length=128,
        blank=True, null=True)
    sector_code = models.CharField(max_length=32)
    instrument_index_code = models.CharField(max_length=128)
    update_enabled = models.IntegerField()
    update_datetime = models.DateTimeField(blank=True, null=True)
    update_last = models.DateField(blank=True, null=True)

    (#contimue with more)

        class Meta:
        managed = False
        db_table = 'sn_instrument'

```

```
class SnFeedInstrument(models.Model):
    feed_code = models.CharField(primary_key=True, max_length=64)  # The composite primary key (feed_code, instrument_code) found, that is not supported. The first column is selected.
    instrument_code = models.CharField(max_length=128)
    is_target = models.IntegerField(blank=True, null=True)
    quality_score = models.DecimalField(max_digits=10, decimal_places=2)

    class Meta:
        managed = False
        db_table = 'sn_feed_instrument'
        unique_together = (('feed_code', 'instrument_code'),)

class SnInstrumentAttribute(models.Model):
    instrument_code = models.CharField(primary_key=True, max_length=128)  # The composite primary key (instrument_code, attribute_code) found, that is not supported. The first column is selected.
    attribute_code = models.CharField(max_length=128)
    attribute_value = models.CharField(max_length=128, blank=True, null=True)

    class Meta:
        managed = False
        db_table = 'sn_instrument_attribute'
        unique_together = (('instrument_code', 'attribute_code'),)
```

Listing B.1: Django models in Python code - common directory

Appendix C

Django urlspatterns

```python
from django.urls import path
from . import views

urlpatterns = [
    path("feeds-list", views.feeds_list, name="feeds-list"),
    path("fg-ingest", views.fg_ingest, name="fg-ingest"),
    path("fg-details", views.fg_details, name="fg-details")]
```

Listing C.1: urls.py in Python code - lygos app Django

```python
urlpatterns = [
    path("yaml-list", views.yaml_list, name="yaml-list"),
    path("yaml-insert", views.yaml_insert, name="yaml-insert"),
    path("yaml-edit", views.yaml_edit, name="yaml-edit"),
    path("download-yaml", views.download_yaml,
       name="download-yaml"),
    path("delete-yaml", views.delete_yaml, name="delete-yaml"),
    path("mapping", cache_page(60*15)(views.mapping),
       name="mapping"),
    path("mapping-options",
       cache_page(60*15)(views.mapping_options),
       name="mapping-options"),
    path("features-selection",
       cache_page(60*15)(views.feature_selection),
       name="features-selection"),
    path("assets-selection",
       cache_page(60*15)(views.assets_selection),
       name="assets-selection"),
    path("test-dataset", views.test_dataset, name="test-dataset"),
    path("test-dataset-view", views.test_dataset_view,
       name="test-dataset-view"),
    path("instruments", cache_page(60*15)(views.instruments),
       name="instruments"),]
```

Listing C.2: urls.py in Python code - byzantium app Django

References

[1] Axyon AI. *'OUR METHODOLOGY'. axyon.ai.* (accessed Jan. 9, 2024). URL: https://axyon.ai/technology#methodology.

[2] Amazon Web Services. *'Amazon S3'. aws.amazon.com.* (accessed Jan. 15, 2024). URL: https://aws.amazon.com/s3/.

[3] Amazon Web Services. *'Amazon SageMaker Feature Store'. aws.amazon.com.* (accessed Jan. 15, 2024). URL: https://aws.amazon.com/sagemaker/feature-store/.

[4] Atlassian. *'Jira Software'. www.atlassian.com.* (accessed Jan. 15, 2024). URL: https://www.atlassian.com/software/jira.

[5] S.C Evercoder Software S.R.L. *'moqups'. moqups.com.* (accessed Jan. 15, 2024). URL: https://moqups.com/.

[6] Docker Inc. *'what is a container'. docker.com.* (accessed Jan. 15, 2024). URL: https://www.docker.com/resources/what-container/.

[7] DBeaver Corporation. *'DBeaver Documentation'. dbeaver.com.* (accessed Jan. 15, 2024). URL: https://dbeaver.com/docs/dbeaver/.

[8] GitLab B.V. *'Why GitLab'. about.gitlab.com.* (accessed Jan. 15, 2024). URL: https://about.gitlab.com/why-gitlab/.

[9] Ken Schwaber and Jeff Sutherland. *'The 2020 Scrum GuideTM'. scrumguides.org.* (accessed Jan. 15, 2024). URL: https://scrumguides.org/scrum-guide.html.

[10] Scrum.org. *'What is Scrum?'. scrum.org.* (accessed Jan. 15, 2024). URL: https://www.scrum.org/resources/what-scrum-module.

[11] Microsoft. *'Microservices architecture'. learn.microsoft.com.* (accessed Jan. 15, 2024). URL: https://learn.microsoft.com/it-it/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture.

[12] Django Software Foundation. *'Documentation'. docs.djangoproject.com.* (accessed Jan. 17, 2024). URL: https://docs.djangoproject.com/en/5.0/.

[13] Medium. *'Django MVT Architecture'. medium.com.* (accessed Jan. 17, 2024). URL: https://medium.com/@CodeMaple/understanding-django-mvt-architecture-and-view-functions-django-full-course-for-beginners-lesson-39c8da093b44.

[14] Evan You. *'Introduction'. vuejs.org.* (accessed Jan. 17, 2024). URL: https://vuejs.org/guide/introduction.html.

[15] Nuxt. *'Introduction'. nuxt.com.* (accessed Jan. 17, 2024). URL: https://nuxt.com/docs/getting-started/introduction.

[16] LLC KBall. *'7 Frontend Architecture Lessons From Nuxt.js'. zendev.com.* (accessed Jan. 17, 2024). URL: https://zendev.com/2018/09/17/frontend-architecture-lessons-from-nuxt-js.html.

[17] Vuetify. *'Introduction'. v2.vuetifyjs.com.* (accessed Jan. 17, 2024). URL: https://v2.vuetifyjs.com/en/introduction/why-vuetify/#getting-started.

[18] Atlassian. *'Microservices architecture'. atlassian.com.* (accessed Jan. 15, 2024). URL: https://www.atlassian.com/it/microservices/microservices-architecture.