U.PORTO

CICLO DE ESTUDOS
MESTRADO EM INFORMÁTICA MÉDICA

Visual Viper: A Portable Visualization Library for Streamlined Scientific Communication

Mariana Canelas Pais

M

2023

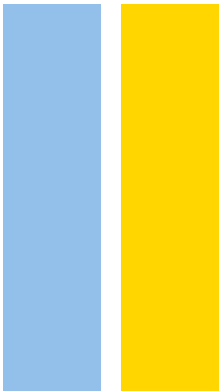


Table of Contents

Abstract

Background

The healthcare industry is seeing a digital revolution, resulting in an ever-growing influx of data. This transformation creates an urgent need for efficient and automated data visualization tools. Visual Viper (VV) aims to meet this demand by offering an automated Python library that streamlines the complex and often time-consuming process of creating data visualizations for scientific communication.

Aim

The aim of this study is to outline the development of VV, assess its performance and adaptability, explore its modular design and development methodologies, and establish its practical applications in healthcare research.

Methods

Built using Python, VV employs Vega-Lite for high-level interactive graphics. The library is structured with modular, extensible architecture, developed with object-oriented programming (OOP) and test-driven development (TDD) practices. Docker containerization ensures a consistent development environment, and GitLab version control, aligned with Semantic Versioning, streamlines collaborative development. Native CI/CD capabilities of GitLab further enrich the development process. VV operates environment-agnostically and offers serverless deployment options.

Results

VV includes various interconnected components, each responsible for specific tasks ranging from data retrieval to chart rendering. Four main classes ('DatasetBuilder', 'ChartNotationBuilder', ‘ChartRenderer’, and 'ChartDeployer') encapsulate the respective functionalities, thus aiding in code maintenance and extension.

Evaluation metrics, captured using Monday.com and Python’s time library, showed that while VV required a longer initial setup time (2h vs. 0.5h), it outperformed manual methods in

"Time-to-Final-Chart" (2h9min vs. 14h54min) for a project involving 72 spreadsheets. Adjusted metrics accounting for task fatigue and human intervention also favor VV, especially for larger and ongoing projects.

VV effectively minimizes manual labor, ensures data visualization consistency and fosters best practices in scientific communication. Current limitations include a focus on mostly specific organizational workflows and visualizations.

Conclusion

VV presents a robust, automated, and customizable solution for data visualization. It holds promise for significantly enhancing scientific communication efficiency within the healthcare sector, with its modular and scalable design paving the way for future developments.

Resumo

Contexto

O sector da saúde está a ser alvo de uma revolução digital que resulta no aumento crescente de dados. Esta transformação cria uma necessidade urgente de ferramentas de visualização de dados eficientes e automatizadas. Visual Viper (VV) tem em vista responder a esta necessidade, oferecendo uma biblioteca Python que automatiza e simplifica o processo complexo e muitas vezes demorado de criação de visualizações de dados para comunicação científica.

Objetivo

O objetivo deste trabalho é descrever o desenvolvimento do VV, avaliar o seu desempenho e adaptabilidade, explorar o seu desenho modular e metodologias de desenvolvimento utilizadas, bem como estabelecer as suas aplicações práticas na investigação em saúde.

Métodos

Construído com recurso à linguagem Python, o VV emprega Vega-Lite para gráficos interativos de alto nível. A biblioteca é estruturada com arquitetura modular, extensível, desenvolvida com práticas de programação orientada a objetos (OOP) e desenvolvimento orientado por testes (TDD). A utilização de contentores Docker garante um ambiente de desenvolvimento consistente, e o controle de versão utilizando GitLab em conjugação com o sistema de Versionamento Semântico, simplifica o desenvolvimento colaborativo. As capacidades nativas de CI/CD do GitLab enriquecem ainda mais o processo de desenvolvimento. O VV opera de forma agnóstica de ambiente e permite opções de implementação sem servidor.

Resultados

O VV inclui vários componentes interconectados, cada um responsável por tarefas específicas que vão desde a leitura de dados até à renderização de gráficos. Quatro classes principais - 'DatasetBuilder', 'ChartNotationBuilder', 'ChartRenderer', e 'ChartDeployer' - encapsulam as respectivas funcionalidades, facilitando a manutenção e extensão do código.

As métricas de avaliação, capturadas com recurso ao Monday.com e a biblioteca ‘time’ do Python, mostraram que embora o VV tenha exigido um tempo de configuração inicial mais longo (2h vs. 0.5h), superou os métodos manuais em "Time-to-Final-Chart" (2h9min vs. 14h54min) para um projeto que envolvia 72 folhas de cálculo. Métricas ajustadas que incluem o efeito da fadiga da tarefa e a intervenção humana, também favorecem o VV, especialmente para projetos maiores e contínuos.

O VV efetivamente minimiza o trabalho manual, garante a consistência da visualização dos dados, e promove boas práticas de comunicação científica. Atualmente, as limitações incluem um foco em fluxos de trabalho e visualizações tendencialmente específicas da organização.

Conclusão

O VV apresenta uma solução robusta, automatizada e personalizável para a visualização de dados. Promete melhorar significativamente a eficiência da comunicação científica dentro do setor de saúde, com seu design modular e escalável abrindo caminho para desenvolvimentos futuros.

Acknowledgements

As I reach this significant milestone in my academic and personal journey, I feel compelled to pause and honor those who have served as irreplaceable pillars of support and inspiration.

Firstly, my deepest gratitude goes to my family. To my son, António, your arrival in the middle of my Master's program has been the most beautiful and motivating challenge of my life. You are a constant source of inspiration, a daily reminder of why I sought this new path. To Ricardo, your love has been an unwavering presence, fundamentally shaping how we've faced the challenges and triumphs in our lives. To my parents, your support and love helped me refine my commitment. Thank you for challenging me and for supporting my choices. To my siblings, who have been invaluable in providing support and in creating a sense of home, no matter the distance that separates us.

My friends, who have been unwavering champions of my aspirations, continually push me to break boundaries. Tthank you for all the joy and fulfillment you bring into my life.

I reserve a profound sense of gratitude for my academic advisors, Professor Tiago Taveira-Gomes and Professor Ricardo Cruz-Correia. It was under your mentorship that I discovered a new career path where my clinical experience can merge with technological innovation. You did not just guide me academically, you instilled in me the confidence to break norms and follow this less-conventional career trajectory, for which I am incredibly passionate.

To my colleagues in the Master's in Medical Informatics program, your camaraderie has been invaluable. The diversity and multi-disciplinary nature of our cohort have not only expanded my horizons but also deeply enriched my perspective.

A special note of appreciation goes to my colleagues at MTG. Your innovative spirit and commitment to excellence have contributed greatly to my professional growth and have continually inspired me to strive for the best.

Finally, my heartfelt thanks go to the entire team at MEDCIDS. Your warm welcome and consistent support have been instrumental in shaping me both academically and as a human being.

Figures

Listings

Abbreviations

API: Application Programming Interface

AWS: Amazon Web Services

CI/CD: Continuous Integration/Continuous Deployment

CLI: Command Line Interface

GB: Gigabyte

GUI: Graphical User Interface

HR: Hazard Ratio

HTML: HyperText Markup Language

IEEE: Institute of Electrical and Electronics Engineers

IDE: Integrated Development Environment

JSON: JavaScript Object Notation

KPI: key performance indicators

OOP: Object-Oriented Programming

RAM: Random Access Memory

REST: Representational State Transfer

SVG: Scalable Vector Graphics

TDD: Test-Driven Development

UML: Unified Modeling Language

VSCode: Visual Studio Code

VV: Visual Viper

YAML: Yet Another Markup Language

XML: Extensible Markup Language

XP: Extreme Programming

Introduction

Healthcare is generating an unprecedented amount of data due to the rise in digital technology [1,2]. This data, ranging from patient records to complex genetic information, holds value for various studies, including those related to real-world evidence. However, the sheer volume of this data makes manual chart generation and updating increasingly impractical. As such, automation is becoming essential for efficient data interpretation in healthcare.

As healthcare increasingly digitizes, the sector is inundated with a complex array of data that professionals and researchers must make sense of. While visualization tools exist, they often don't address the specific needs of healthcare data or scale well with big data challenges [3,4]. Moreover, the manual effort involved in using these tools remains significant. Therefore, there's a growing demand for an automated and scalable solution capable of simplifying the generation and deployment of relevant visualizations.

Objectives

The aim of this project is to conceive, architect, develop and evaluate Visual Viper (VV), a Python library aimed to automate the creation of data visualizations in the healthcare sector. This work will provide a description of each phase, from initial requirement gathering and system architecture design to coding, testing, and evaluation. Limitations will be discussed, along with suggestions for future enhancements.

To provide a comprehensive understanding of the scope of this project, the following objectives are enumerated:

- Conduct an initial requirement analysis to identify the specific needs and constraints that VV aims to address.
- Outline the architecture of VV while adhering to best practices in software development.
- Implement the designed architecture of VV, emphasizing its modular and extensible nature.
- Apply and critically analyze software development methodologies such as object-oriented programming and test-driven development in the creation of VV, considering their impact on the code's quality, maintainability, and extensibility.

- Implement, test, and evaluate the features that VV offers for data retrieval, transformation, and visualization, with a specific focus on retrieving data from Google Sheets, creating Forest Plots, and deploying visualizations to Miro Board and Google Drive.
- Conduct performance testing on VV to assess its efficiency and scalability, especially when handling large healthcare datasets.
- Review and identify areas of improvement within the current version of VV, setting the stage for future iterations and enhancements.
- Assess the tool's success in automating the data visualization process in healthcare research, measuring its effectiveness in facilitating scientific communication.

The project seeks to fill a critical gap in the existing tools for automating the generation of healthcare data visualization. By automating the often labor-intensive and complex process of generating custom visualizations, VV aims to significantly improve the efficiency of scientific communication in healthcare. It also introduces a modular and extensible architecture, enabling the library to adapt to diverse data sources and evolving visualization needs, thereby extending its lifespan and relevance.

Thesis Overview

This thesis is organized as follows:

- **Introduction**: This section encompasses the background of the study, problem statement, purpose, research objectives, and justification. It serves to establish the context and significance of the research.
- **Background**: This section provides a concise assessment of the literature relevant to data visualization.
- **Methodology**: This section details the methodologies adopted. It covers aspects like requirement analysis, user stories, and scope of the project. Core principles such as modularity, extensibility, and usability are also elaborated in separate subsections.
- **Development Approach**: Offers an overview of the development approach and programming paradigms employed. It discusses possible development approaches and discusses the use of Object-Oriented Programming and Test-Driven Development.
- **Development Environment and Tools**: In this section, the various tools utilized during the development process are covered in detail. This includes aspects of Docker

containerization, version control through GitLab, and CI/CD pipelines. Furthermore, this section elucidates the build automation process and discusses the Makefile in detail.

- **System Architecture**: Provides an in-depth description of the system's architecture. It discusses the high-level architecture, key classes, component interactions, and data flows among components.
- **Implementation Details**: This section delves into the technical nuances of the project's implementation.
- **Workflow Demonstration**: This section provides a demonstration of how the VV system operates in a real-world context. The aim is to convey both the utility and the user experience of the system.
- **Evaluation Results**: This section analyzes VV's performance on the specific use cases of Google Sheets data retrieval, Forest Plots creation, and deployment to Miro Board and Google Drive.
- **Discussion**: This section serves as a platform to review the research findings and to propose future recommendations.
- **Conclusion**: Summarizes the research and outlines the contributions made by the study.
- **References**: Lists all sources cited throughout the document.

Background

This section provides a concise assessment of the literature relevant to data visualization, emphasizing its role in healthcare research, as well as a brief description of some technical foundations of software development. For this non-systematic review we used several academic resources such as PubMed, IEEE Xplore, Scopus, and arXiv.org.

Data Visualization

Data visualization serves various aims, such as exploration, interpretation, and communication of data, by harnessing human visual perceptual abilities [5]. The field is inherently complex, integrating elements of creativity, technology, and social knowledge to achieve its goals. This complexity echoes the diverse requirements and challenges seen in healthcare research, where visualization tools must be both scientifically rigorous and accessible for diverse stakeholders.

Historical Development and Evolution

Historically, visualization techniques have been distributed mainly as stand-alone applications or specialized libraries. This practice is particularly prevalent for niche or highly specialized visualization methods. However, over time, there has been a shift towards generalization and abstraction. Developers have distilled components from these specialized solutions to create general-purpose frameworks. These frameworks assist in crafting custom visual representations, providing a more flexible toolset for different applications, including healthcare research [5].

Design Choices

To appreciate the role of visualizations in today's research landscape, it's critical to analyze their evolution and the importance of their design. Over the last three centuries, charts, graphs, and equivalent visual representations have become primary mediums for quantitative communication [6] .

Despite their increasing use, visualization designers have to navigate through multiple decisions. This includes the choice of visual encoding and styling, which significantly influences the aesthetics and perception of a graphic [7]. Unfortunately, though principles of effective visualization design have seen significant development, many contemporary charts exhibit substandard design choices that interfere with comprehension and aesthetic appeal. However,

incorporating automated design methods based on established visual design principles can improve the effectiveness and consistency of visualizations, particularly for analysts working with their own data [8].

Graphics Grammars

One foundational line of research in this field has been the systematic study of structural theories of graphics, which is thought to have been introduced by Bertin [9]. In this research statistical graphics were deconstructed into their basic elements such as rectangles, lines or points. This in turn led to the development of graphical languages. These languages enable a wide array of graphical representations through the combination of simple geometric primitives and transformations [5].

In the field of data visualization, the term "Graphics Grammars" refers to a methodological approach to the creation and manipulation of visual displays using structured, syntax-like rules and principles. Derived from language theory, grammar here doesn't pertain to linguistic rules; instead, it represents a system of structures and transformations that directs the visual representation of data. Graphics grammars enable a more systematic and succinct specification of graphics, which can be a real advantage in large-scale, complex data visualization projects.

Low-level grammars such as Protovis [10], D3 [11], and Vega [12] are often beneficial for explanatory data visualization or creating customized analysis tools due to their primitives offering fine-grained control. For exploratory visualization, however, higher-level grammars like ggplot2 [13], are usually preferred for their conciseness over expressiveness. Another example of a higher-level grammar is Vega-Lite, that provides a more concise interface than the lower-level Vega language, making systematic enumeration and ranking of data transformations and visual encodings more manageable [14]. A summary of various graphics grammars and their characteristics is provided in Table X.

Table X: Summary of Graphics Grammar Languages

Language	Level	Implementation	Notable Features
Protovis	Low	JavaScript	No longer under active development, the responsible team is now maintaining D3.js (see below).
D3.js	Low	JavaScript	Capable of generating interactive data visualizations, including transitions and tooltips, using web technologies. Typical use

			cases include the creation of custom visualizations.
Vega	Low	JavaScript/ TypeScript	The visualization is defined in a JSON format. Typical use cases include the creation of explanatory figures, with high degree of customization.
ggplot2	High	R	Part of the tidyverse, a collection of R packages designed for data science. Based on the concept of the "Grammar of Graphics," initially proposed by Leland Wilkinson. Widely-used in the academic community.
Vega-Lite	High	TypeScript	Enables the use of higher-level grammar, defined using JSON format, that is compiled to Vega specifications. Typical use cases include the creation of quick exploratory data visualizations.
Vega-Altair	High	Python	Leverages the Vega-Lite JSON specification and creates a declarative Python API for the creation of visualizations.
Matplotlib	Low	Python	One of the most popular python libraries for data visualization. Notable for extensive customization and ability to generate 2D and 3D Plots.

Visualizations in the Study Lifecycle

Visualizations have a vital role throughout the lifecycle of any research study. They provide key insights during crucial stages such as [15]:

- **Protocol development**: Visualizations aid in analyzing design and data issues clearly and objectively, ensuring study accuracy.

- **Diagnostics**: They assist in verifying if all prerequisites for the study have been met, including the requirements set by the chosen statistical methods.

- **Results**: Visual data representations help to interpret research outcomes enhancing communication and understanding.

The role of visualizations throughout various stages of a research study underscores the necessity for specialized tools capable of adapting to the complex demands of healthcare research. VV aims to address some of these needs by automating the visualization creation process.

Technical Foundations and Development Paradigms

This section will outline some key principles that have guided the architecture and functionalities of robust and scalable software systems.

Modularity

The idea of modularity has a long history in the field of software development. As early as 1970, Gouthier and Pont outlined the critical elements of system modularity in their textbook on system program design, stating that well-defined project segmentation ensures each task forms a distinct program module. This clarity in definition streamlines the implementation, testing, and even maintenance phases of development, making it easier to trace errors and deficiencies to specific system modules [16].

Parnas' seminal paper in 1972 further evolved the philosophy by introducing the concept of "information hiding" in modular programming, laying the groundwork for what later came to be termed as high cohesion and loose coupling [17,18]. This evolution was particularly important for large codebases, offering a framework that allows modules to be written, reassembled, and replaced without needing to reassemble the entire system [17].

Beyond the code itself, the systematic reuse of software modules offers a series of additional benefits. This approach not only improves software dependability but also reduces process risks and accelerates development cycles [19]. These advantages are particularly important in healthcare settings where the need for reliable and timely solutions is ever-present.

The importance of conceptual integrity in software design shouldn't be underestimated either. In his 1975 book, "The Mythical Man-Month: Essays on Software Engineering," Brooks advocated for the architecture of a system to be designed by a single mind or a small, cohesive team to ensure a consistent and well-thought-out framework [20].

Overall, the benefits of a modular design approach are far-reaching. They contribute collectively to enhancing productivity and software quality, significantly reducing both time-to-market and development costs [19]. In fields like healthcare, the positive impacts of adopting a modular design philosophy can be particularly impactful.

Object-Oriented Programming (OOP)

OOP has been a common paradigm for solving complex tasks through interactions between objects. It allows for greater flexibility, better quality coding techniques, and enhanced productivity [21,22]. With the project's complexity and the need for a clear, modular structure, OOP becomes an ideal choice. OOP languages like C++, Python, and Java have dominated software development, making them crucial for both current and future applications [23,24].

While OOP offers many advantages, it is not without limitations. Complexity control remains a challenge, especially when these codes are updated to cover future requirements [21,25]. This complexity often results from the very features that make OOP powerful: polymorphism, inheritance, and encapsulation.

To manage this complexity, several design principles and patterns have been introduced. The Gang of Four's design patterns provide robust frameworks for addressing recurrent design issues, focusing on creational, structural, and behavioral patterns. These patterns help in making OOP code more manageable, reusable, and maintainable [26]. Furthermore, the Unified Modeling Language (UML) has been instrumental in providing a general-purpose language for visualizing, specifying, constructing, and documenting the artifacts of software systems. It aids both developers and business stakeholders throughout the software modeling process [27]. To enhance code quality and manage complexity, principles such as the SOLID principles have been proposed, promoting design that is easy to manage and scale [24,28].

Test-Driven Development (TDD)

The field of software development offers a variety of methodologies aimed at optimizing code quality, increasing efficiency, and promoting teamwork. Among these, Test-Driven Development (TDD) is notable for its iterative approach that integrates programming, unit testing, and code refactoring. Based on the review on the impact of TDD on program design and software quality, as well as the educational benefits for the author, we have selected TDD as a methodology for our software development project.

TDD promotes the writing of automated tests before the actual production code is developed. This proactive approach has been shown to lead to projects of higher quality that are completed in a shorter period compared to traditional methods. One added benefit is the generation of a regression-test suite as a natural outcome, minimizing the need for manual testing while allowing

for earlier error detection and quicker remediation. Traditional software development often involves considerable time and resources dedicated to debugging in later stages. TDD, however, facilitates testing early in the design cycle, significantly reducing the time and financial resources spent on debugging [29]. This can mitigate some of the complexity control issues that are often seen in OOP [23,25].

In the TDD methodology, refactoring plays a crucial role, enabling ongoing improvements in the internal structure of the code while preserving its external behavior. This is beneficial for code maintainability and long-term project viability [30]. TDD encourages modular code, which aligns with the OOP principle of high cohesion and loose coupling introduced by Parnas [17,24]. This makes it easier to maintain and extend the system, thus enhancing productivity, which represents a key advantage of OOP.

TDD is versatile, compatible with a range of software development paradigms such as Agile, Scrum, XP, and Lean. This adaptability offers flexibility in project management approaches [31].

Python Programming Language

The landscape of programming languages is ever-changing, but Python has consistently shown remarkable growth both in the educational sector and the industry at large. Python's straightforward syntax and robust set of tools position it as an ideal language for educational settings, particularly for those new to programming. It's the go-to introductory language at many top-tier universities, facilitating a seamless transition from basic mathematical reasoning to intricate coding tasks. When compared to languages like Java and C++, Python offers easier code writing, thanks in part to its clean syntax, which is especially beneficial for educational environments [32].

According to RedMonk's programming language rankings, as depicted in Figure X, Python has ascended to the second position as of 2023, right behind JavaScript. This ranking reflects a combination of GitHub repositories and Stack Overflow discussions, providing insights into both code usage and community discussion. The methodology doesn't claim to offer a statistically valid representation of current usage but rather aims to provide insights into potential future adoption trends [33].

A comprehensive survey by Stack Overflow, which gathered responses from 89,184 software developers across 185 countries, revealed that Python is the second most popular programming

language in 2023, trailing only behind JavaScript (64% to 49%, respectively). Python emerged as the most favored language among non-professional coders. In the educational sector, Python's impact was also pronounced: 57% of student developers reported using Python, a figure that closely trails JavaScript's 61%, further underscoring Python's growing significance in educational settings. Note that in our the interpretation of the survey data categories such as HTML/CSS and SQL were deliberately excluded from this analysis due to their unique and complementary roles in software development [34] .

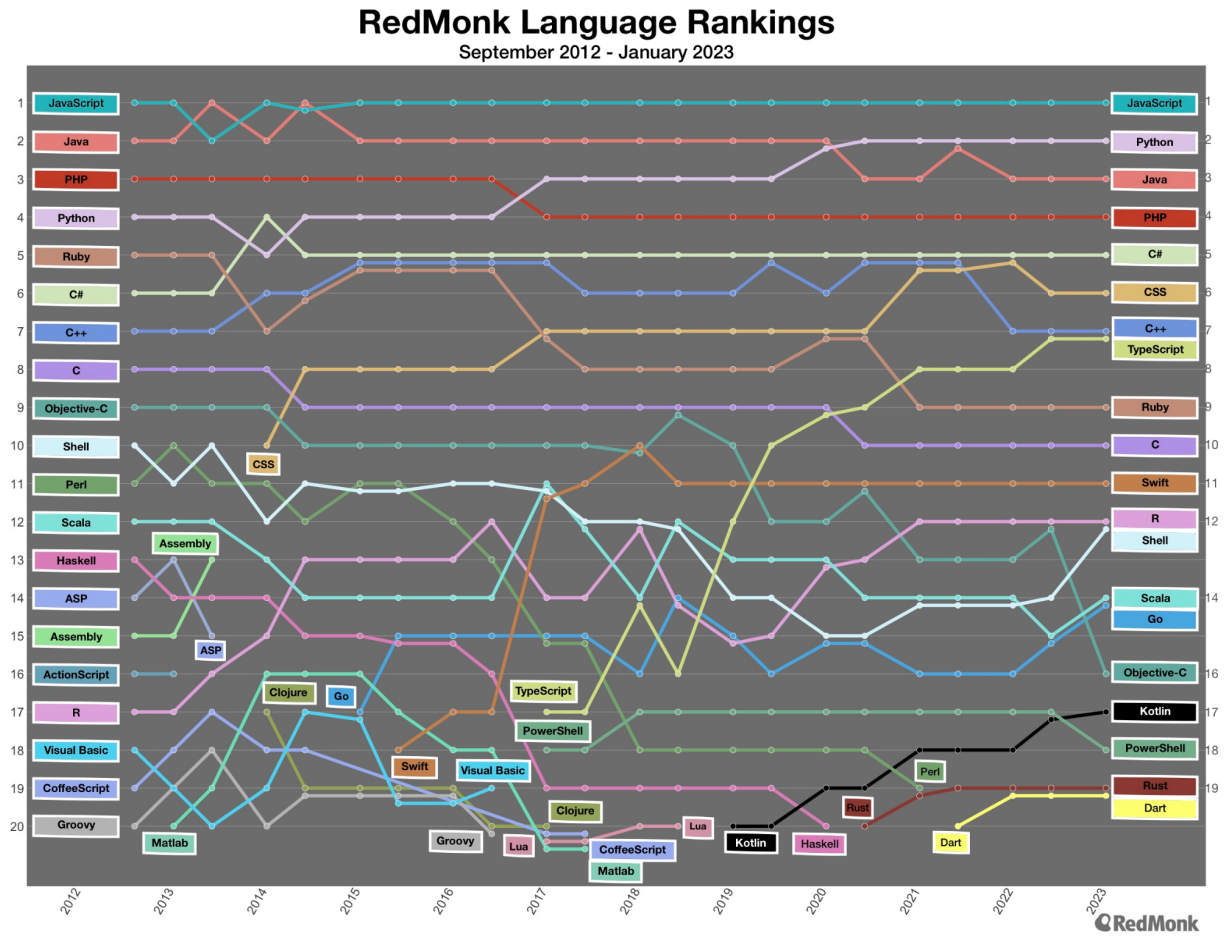


Figure X: Programming Language Popularity Over Time, adapted from RedMonk's 2023 Q1 Programming Language Rankings [33].

Docker for Containerization

With Docker, each part of an application, along with its dependencies and libraries is packaged together in a container. This ensures that the application runs uniformly regardless of where the container is deployed [35].

An alternative to Docker is to manage dependencies manually or use a virtual environment such as Python's native venv. While such options can work, they are not as robust as Docker when it comes to encapsulating an application and its environment. Specifically, they do not provide the level of isolation or the ease of deployment that Docker offers [36].

Advantages of Using Docker

- **Isolation**: Containers operate in isolation, ensuring that each service is unaware of the other and runs independently.
- **Version Control for Environments**: Much like source code, the Docker environment can be version controlled, enabling easy rollback and updates.
- **Scalability**: Docker makes it easier to create a distributed system, facilitating the application's scaling without a hassle.
- **Easy Deployment**: With Docker, the development environment can be precisely replicated in the production system, minimizing deployment errors.
- **Cross-Platform**: Docker containers can run anywhere, on any machine that has Docker installed, regardless of the underlying operating system.

Disadvantages of Using Docker

- **Learning Curve**: Docker has a learning curve, and initial setup can be complex.
- **Resource Intensive**: Containers may consume more resources than native applications when running multiple instances.
- **Overhead**: For simpler applications, the advantages of Docker may not justify the resource overhead and the complexity it introduces.

Methodology

The methodology section serves as a roadmap detailing the design, development, and evaluation of the VV Python library. The objective here is to offer comprehensive insights into the technical aspects of VV, elucidating the rationale behind various design and architectural choices, as well as the methods used for implementation and assessment. Given that this library aims to bridge a gap in healthcare data visualization, especially in handling big data and providing customizable solutions for automation, it is crucial to understand the techniques and technologies that make it both functional and scalable.

This section will start by explaining the basic ideas behind the VV project. Then, we'll get into the actual development aspects, including our use of Object-Oriented Programming (OOP) and Test-Driven Development (TDD). Lastly, we will explore how the library was evaluated, describing the metrics and methods used during the evaluation phase.

Requirement Analysis

This section outlines the key functions and quality features expected of the VV system. It provides a set of clear requirements that will guide the design and implementation stages of the project. To enhance the system's effectiveness and ease of use, we present selected use cases that illustrate how VV will interact with other systems for better integration in the broader data visualization landscape. In short, this section sets the foundational requirements that will guide the development efforts.

User Stories

User stories serve as a vehicle for capturing product functionality from the end user's perspective. These stories encapsulate discrete system features in a format that is easy to read and understand by both non-technical stakeholders and the development team [37,38]. In the context of VV, a system designed to automate the rendering of graphical charts from clinical research data, the user stories described here are aimed to outline the essential features and functionalities that satisfy the needs of different roles involved in clinical research.

Scope

The following user stories are specifically tailored to the needs of clinical researchers, medical writers, data analysts, and system administrators who are the key stakeholders of the VV system. They focus on tasks related to data visualization, report generation, and system management within the context of clinical research.

Stakeholder Definitions

- **Clinical Researcher**: A professional conducting clinical studies.
- **Medical Writer**: A professional responsible for creating documents that describe research results, product use, and other scientific dissemination outlets.
- **Data Analyst**: A person responsible for interpreting complex clinical data sets.
- **Data Scientist**: A professional who uses scientific methods, processes, algorithms, and systems to extract insights and knowledge from structured and unstructured data.
- **System Administrator**: A person responsible for managing and maintaining the system infrastructure, including VV.

Story 1: Batch Rendering of Clinical Charts

- **As a** clinical researcher/data scientist,
- **I want** VV to batch render multiple charts from automatically generated clinical reports,
- **So that** a large volume of data can be visually represented quickly and efficiently.
- **Acceptance Criteria**:
    - System should be able to accept multiple clinical reports as input.
    - System should be able to render charts in batches without manual intervention.
    - Rendered charts should accurately represent the data from the clinical reports.
    - System should provide an option for selecting the types of charts to be rendered (bar, line, etc.).
    - Batch rendering process should complete within a reasonable time frame (e.g., under 5 minutes for 50 reports).

Story 2: Deployment to Miro for Triage

- **As a** clinical researcher/data scientist,
- **I want** VV to deploy rendered charts directly to specific boards in Miro,
- **So that** they can be quickly triaged alongside tabular reports.

- **Acceptance Criteria**:
    - System should integrate with Miro API.
    - System should be able to send rendered charts to specified Miro boards.
    - Rendered charts should appear on the Miro boards in a layout that facilitates triage.
    - Charts should be deployed to Miro boards without manual intervention.

Story 3: Export Charts for Research Documents

- **As a** medical writer,
- **I want** VV to export rendered charts in formats suitable for academic manuscripts, posters, and other research documents,
- **So that** the visual data complements the written content.
- **Acceptance Criteria**:
    - System should offer multiple export formats such as PNG, JPEG, SVG, etc.
    - Exported charts should maintain high resolution and quality.
    - System should allow for batch export of multiple charts.

Story 4: Inclusion of Supplementary Material

- **As a** clinical researcher/data scientist,
- **I want** VV to render charts that can be included as supplementary material when publishing,
- **So that** we can increase the transparency of our research.
- **Acceptance Criteria**:
    - System should allow rendering of charts that are suitable for supplementary material in terms of quality and resolution.
    - System should allow for easy categorization or labeling of such charts for supplementary material.
    - Charts should be exportable in a format accepted by major research publications.

Story 5: Automated Data Retrieval

- **As a** data analyst,
- **I want** VV to automatically retrieve data from predefined clinical report formats,
- **So that** I don't have to manually input data for chart rendering.

- **Acceptance Criteria**:
    - System should be able to identify and read predefined clinical report formats.
    - System should accurately extract relevant data fields from these reports.
    - Data retrieval should happen automatically through API calls.

Story 6: Customization of Chart Types

- **As a** clinical researcher/data scientist,
- **I want** to specify the type of chart (bar, line, scatter, etc.) VV should render,
- **So that** the chart is most appropriate for the data being represented.
- **Acceptance Criteria**:
    - System should offer a range of chart types (bar, forest plot, survival, etc.).
    - Users should be able to easily select the desired chart through configuration.
    - Rendered charts should accurately represent the selected chart type.

Story 7: Logging and Monitoring

- **As a** system administrator,
- **I want** VV to keep logs of all chart rendering activities,
- **So that** I can monitor system performance and troubleshoot issues.
- **Acceptance Criteria**:
    - System should maintain logs for each chart rendering activity.
    - Logs should include timestamps, types of charts rendered, and any errors or warnings.
    - Logs should be easily accessible for review and analysis.

Story 8: Re-run Chart Rendering with Updated Data

- **As a** clinical researcher/data scientist/medical writer,
- **I want** to re-run chart rendering when new data is available,
- **So that** my visual representations are always up-to-date.
- **Acceptance Criteria**:
    - System should allow for easy updating of data sources.
    - Users should be able to initiate re-rendering without having to redo the entire setup.

Non-functional Requirements

The non-functional requirements for VV aim to outline the quality attributes the system should possess. These are essential aspects that define how well the system performs its functions rather than what functions it performs. They encompass characteristics like modularity, error handling, and auditability, among others. These requirements are especially critical in ensuring that VV is not only functional but also efficient, maintainable, and adaptable to various environments and use-cases. Below is a list of the non-functional requirements we deem essential for the system:

System Architecture

- **Modularity**: The system should be modular to allow for easier debugging and updating of individual components.
- **Extensibility**: Designed in a way to easily allow the addition of new functionalities.

Usability and User Experience

- **Configurability**: Users should be able to easily configure chart rendering options regardless of the environment (API, module, terminal).
- **Environment Agnosticism**: Should be usable as an importable Python module, accessible via web API, or through the terminal.

Reliability

- **Error Handling**: The system should be able to gracefully handle errors and exceptions, providing useful error messages.

Maintenance and Support

- **Documentation**: All code should be well-documented, and system documentation should be easily accessible for maintenance activities.
- **Auditability**: Should provide logging features to keep track of data processing and rendering activities.

Technical Foundations and Development Paradigms

The objective of VV is the automation of data visualization, helping with the challenges in handling large and complex data sets common in healthcare. Concurrently, the project serves an educational purpose, offering the developer a framework to explore and learn fundamental

software development paradigms. This educational aspect makes it crucial to ensure that the project adheres to established coding practices and methodologies, making it both a practical tool for data visualization and a case study in applying robust software development principles.

The following sections will delve into the specifics of these foundational principles, revealing how they guided the choices in architecture and functionalities in VV.

Modularity

In VV, modularity is a fundamental element guiding our design approach. This ensures that each module is a self-contained unit with well-defined interfaces, enhancing both reusability and portability, attributes highly valued in specialized fields like healthcare informatics [39].

Object-Oriented Programming (OOP)

In the VV library, OOP serves as a pivotal architectural choice, both for the developer's educational enrichment and the system's overall functionality and extensibility. Employing OOP facilitates encapsulation, which allows for the bundling of data and methods that operate on that data within single units or classes.

OOP also uses inheritance, enabling code reusability and abstraction. For instance, different types of charts, be it a bar chart, a forest plot, or a survival plot, can be represented as individual classes. These classes can contain methods to set chart properties, draw axes, and render the data. Since each chart type may have common characteristics such as a title or axes labels, inheritance allows these shared features to be abstracted into a parent class. Specific chart types can then inherit from this parent class, enabling them to reuse common code while still allowing for their own specialized features. Furthermore, different deployment targets, like cloud storage or Miro boards, can also be abstracted into separate classes, encapsulating the methods required for deploying visualizations to these locations. This makes the system adaptable and easier to integrate with new deployment options as needs evolve.

Test-Driven Development (TDD)

TDD serves as a rigorous verification mechanism that aligns with the project's objective of delivering a reliable and high-quality tool. Furthermore, for the author, who is in the process of learning software development, employing TDD provides a valuable educational experience. This

hands-on exposure is expected to be invaluable in future projects and particularly beneficial when collaborating within larger teams that also utilize TDD.

To implement TDD in this project, we selected pytest as the testing library for its feature-rich environment, ease of use, and compatibility with various Python frameworks. It provides detailed failure reports to streamline debugging, and its straightforward syntax is especially beneficial for those new to TDD [40].

Evaluation Metrics and Methods

The evaluation phase for the VV Python library was designed to assess both the functional capabilities of the library and its impact on workflow efficiency. The key performance indicators (KPIs) used for this evaluation were "Time to First Chart Draft" and "Time to Final Chart," designed to capture the time-efficiency gains enabled by the VV library.

Time to First Chart Draft

This metric captures the time needed from receiving the initial dataset to generating the first draft of a chart. For the manual method, this involves gathering values for relevant measures, preparing a Vega-Lite JSON definition, populating the JSON with the data and adjusting necessary parameters.

Time to Final Chart

This metric gauges the time from the receipt of the initial data to the point where the chart is exported in the appropriate format (e.g., SVG) and uploaded to a platform like Google Drive and included in a Miro board for further analysis and comparison. This encompasses the entire lifecycle of chart production and is intended to capture any efficiency gains that may be achieved through the VV library.

Data Sources for Evaluation

The primary data source for these evaluations is time-tracking data from MTG Research and Development Lab activities. This data focuses on chart development for academic papers and is an integral part of our methodology. It has been recorded using a tracker within the Monday.com platform, which is the project management tool employed by the company for all R&D activities.

This time-tracking data from past projects, where chart generation was performed manually, serves as a comparative baseline for evaluating the VV Python library's effectiveness.

Simulation for Adjustment for Fatigue and Human Intervention

To provide a comprehensive evaluation of the VV Python library's efficiency in chart creation, we extended our analysis by including a simulation that includes considerations for task fatigue and additional human intervention for validation. For this exercise, we focused on the "Time-to-Final-Chart" metric, which captures the total time needed to finalize a chart, accounting for all adjustments and confirmations.

The analysis was conducted using R (version 4.2.3) [41], and visualizations were generated using the ggplot2 package [13].

Development Environment and Tools

In this section, the development environment and tools used in the construction of VV are discussed. The selection of this environment went beyond mere technical suitability for the project requirements; it also served as an educational framework for the author. The project was not only an exercise in software development for clinical research but also a formative experience in employing modern software development tools and practices. Thus, the choices made were influenced both by their ability to efficiently realize the project's goals and their pedagogical utility in skill acquisition. Through the development process, the author gained valuable insights into effective software development practices.

Development Environment

The development environment consisted of a macOS Ventura machine, running version 13.3, powered by an Apple M2 Pro processor with 16GB RAM.

For the code editing, Visual Studio Code (VSCode) Version 1.81.1 (Universal) was chosen as the Integrated Development Environment (IDE), as depicted in Figure X.

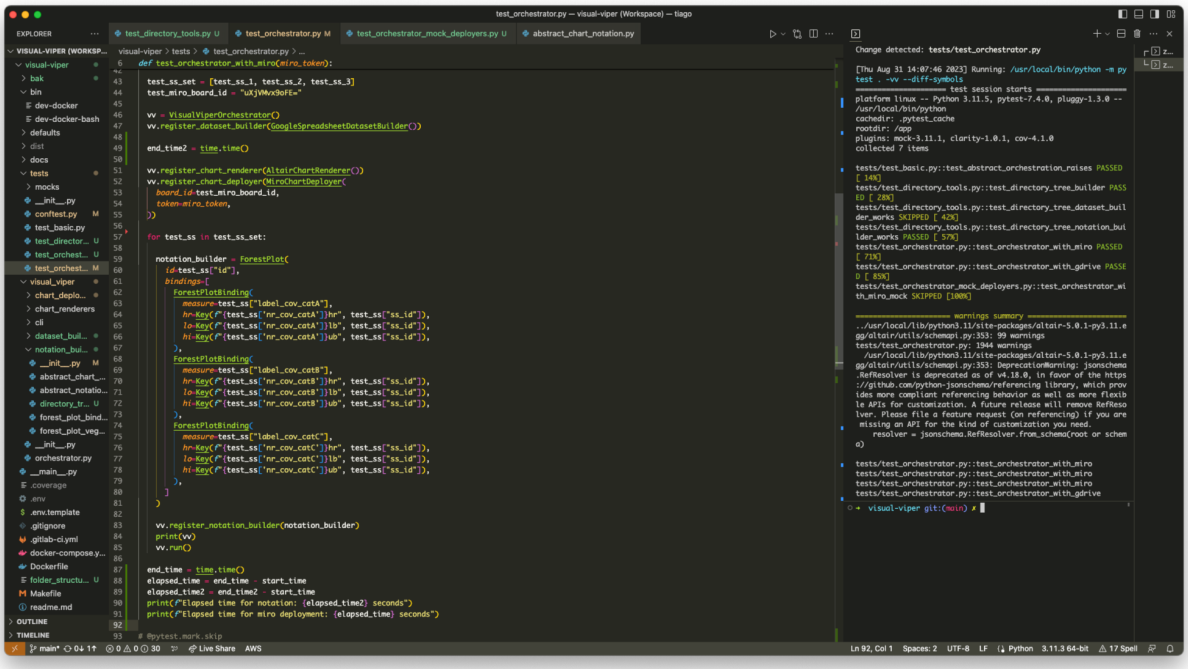


Figure X: Screenshot of the development environment in Visual Studio Code, showcasing the editor's interface, code structure, and various extensions for enhanced productivity. The split terminal on the right side illustrates the integrated development and testing workflow.

The choice of VSCode was influenced by its extensive feature set, including code auto-completion, debugging tools, and an active extension marketplace. Particularly beneficial was the use of the VSCode Live extension, which facilitated live coding sessions for tutoring and collaborative development.

Docker for Containerization

The use of Docker for containerization was a strategic decision aimed at creating a consistent and isolated environment for development and deployment. Figure X provides a screenshot of the Docker Graphical User Interface (GUI), where the operational status of the running 'visual-viper' container is displayed.

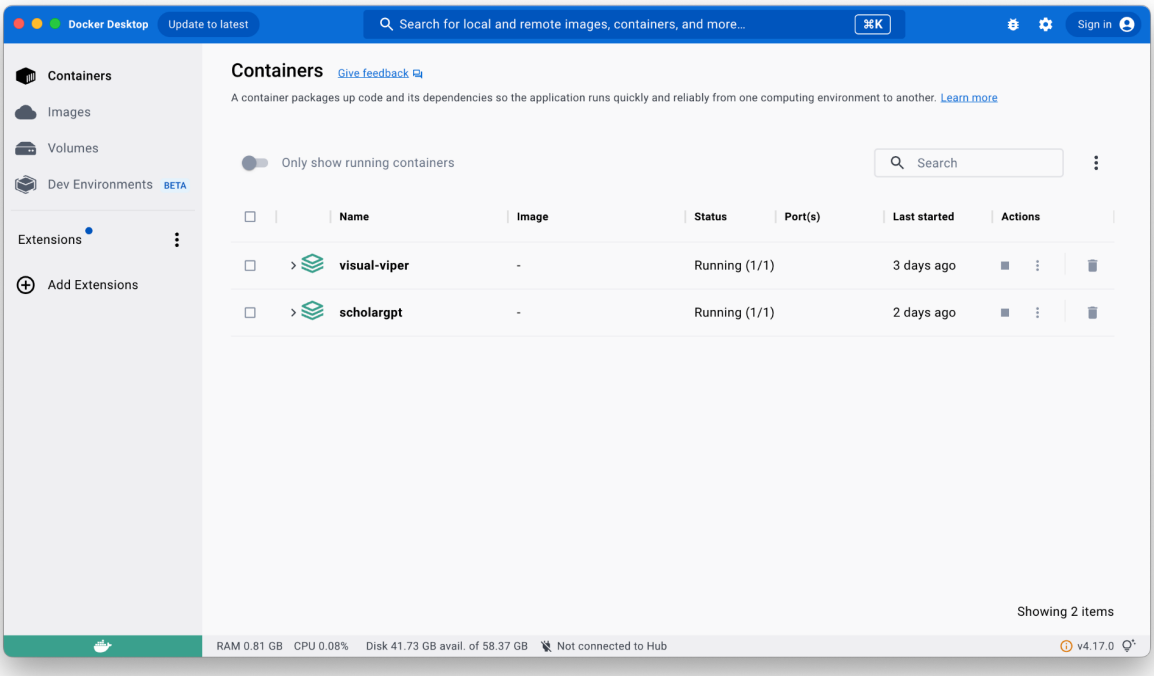


Figure X: Screenshot of the Docker Graphical User Interface (GUI), displaying the running 'visual-viper' container and indicating its operational status.

Using Docker was not just about setting up a convenient environment for code development. It also served as a practical way to learn about important modern practices in software engineering, such as containerization and DevOps. This hands-on experience was valuable for both the project's success and educational objectives, making Docker an optimal choice for this project.

Version Control

GitLab (Version 16.3) was employed as the platform to host the remote repository for this project, in conjunction with the version control system Git (Version 2.39.2, Apple Git-143).

We adhered to Semantic Versioning 2.0.0 for labeling the versions of our project [42]. Figure X displays the GitLab badge for version number 0.0.1.

Figure X: GitLab badge for version number 0.0.1.

The repository followed a simplified branch structure comprising two most important branches:

- **main**: Served as the repository for code deemed ready for production.
- **feature**: Used exclusively for the development of new features or improvements.

The primary driver behind the selection of GitLab for version control and remote repository hosting was its compatibility with the technology stack currently in use at the author's workplace. This alignment not only ensured a seamless integration but also leveraged existing organizational workflows.

Furthermore, GitLab was advantageous for several other reasons, such as:

- Collaboration Features: The platform supports functionalities such as merge requests, code reviews, and issue tracking that enhance team collaboration.
- CI/CD Integration: GitLab's native support for Continuous Integration and Deployment pipelines enriched the development process, with further elaboration in the CI/CD subsection.

Continuous Integration and Deployment (CI/CD)

The implementation of Continuous Integration and Deployment (CI/CD) pipelines is central to modern software development practices. It allows for seamless code integration, testing, and deployment, thereby accelerating the development cycle and reducing the time to market. For this project, GitLab's native CI/CD capabilities were utilized to fulfill these objectives. Listing X shows the GitLab CI/CD Configuration YAML file that was used for automated testing and deployment.

Note that all make commands used in this pipeline are elaborated upon in the Build Automation subsection.

CI/CD Configuration

The CI/CD pipeline was configured using a .gitlab-ci.yaml file, which specifies the environment and commands that GitLab's CI/CD runners should execute. The pipeline was designed to run on a Python 3.10 environment and included two main jobs: test and pages.

Before Script and Dependencies

The before_script section provides the initial setup, which includes updating package lists and installing the FreeTDS dependency required for the project. Following this, the make install command sets up the necessary Python packages.

Test Job

The test job runs the test suite and generates a code coverage report. It uses the make test-ci script, capturing the code coverage percentage as well as producing a JUnit XML report. These artifacts are then stored and can be accessed for further analysis.

Pages Job

The pages job runs only on the main branch and is responsible for generating project documentation. The documentation is built using the make doc command and the output HTML files are moved to the public directory. This ensures that the latest version of the documentation is always available on the project's GitLab Pages. Further details on documentation generation can be found in the Documentation section.

```yaml
image: python:3.10

default:
  before_script:
    - apt update
    - apt install -y freetds-dev
    - make install

test:
  script:
    - make test-ci
  coverage: '/TOTAL.*\s+(\d+%)$/'
  artifacts:
    when: always
    paths:
      - dist/test/junit.xml
    reports:
      junit: dist/test/junit.xml
      coverage_report:
        coverage_format: cobertura
        path: dist/coverage/coverage.xml

pages:
  only:
    - main
  script:
    - make doc
    - mv dist/docs/html public
  artifacts:
    paths:
      - public
  only:
    - main
```

Listing X: GitLab CI/CD Configuration YAML file for Automated Testing and Deployment

Pedagogical Implications

The opportunity to configure and operate a CI/CD pipeline through GitLab has valuable educational benefits, offering the opportunity to understand the principles of automated testing and deployment in the realm of software engineering. Figure X provides a snapshot of a successfully executed CI/CD pipeline, illustrating that all stages were completed.

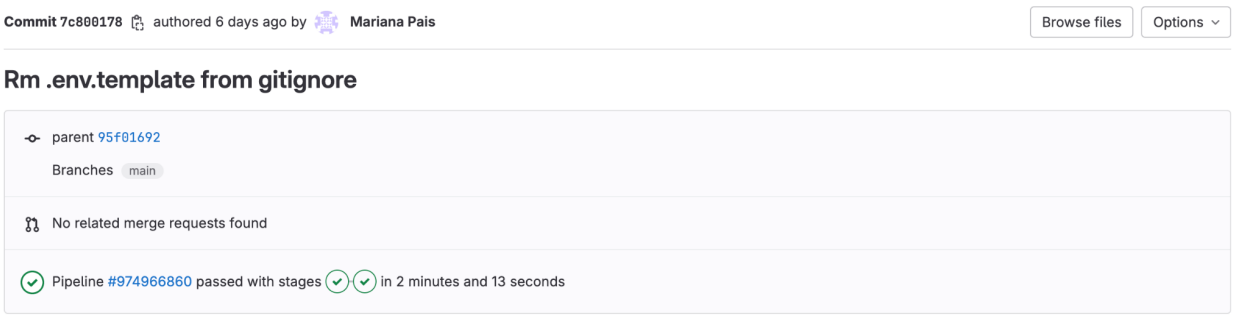


Figure X: Snapshot of a successfully executed CI/CD pipeline for commit 7c800178 on the main branch, illustrating that all stages passed in a duration of 2 minutes and 13 seconds.

Build Automation

In this subsection, the utility of build automation is discussed, focusing on the role of the Makefile in project development. Build automation offers both convenience and standardization, aiding in the quick execution of repetitive tasks and ensuring that all collaborators are using the same set of commands.

Makefile

The Makefile serves as the framework for build automation in this project. It consists of shorthand commands that encapsulate complex or multi-step tasks into a single-line command. These commands serve multiple purposes within the development cycle, from setting up Docker containers to running tests and generating documentation. The structure and details of the Makefile used in the project are displayed in Listing X.

```
docker:
	bin/dev-docker


install:
	python3 -m pip install -q -r requirements.txt
	python3 setup.py develop


run:
	python3 . run


# Shorthand commands for development

dev:
	ENV=dev \
	bash -c 'ptw -c . -- -vv --diff-symbols '


# Shorthand commands for test

test:
	ENV=test \
	bash -c 'pytest . -vv --diff-symbols --cov-report=html:dist/coverage --cov visual_

test-ci: install
	ENV=test \
	bash -c 'pytest . -vv --diff-symbols --junitxml dist/test/junit.xml --cov-report=xm


# Shorthand commands for documentation

doc:
	sphinx-build docs dist/docs/html

dev-doc:
	ptw --runner 'sphinx-build docs dist/docs/html' --ext py,rst


# Shorthand commands for pushing

push:
	git add .
	git commit -m "minor push"
	git push
```

Listing X: Extract from the Makefile, illustrating shorthand commands for various development tasks.

Commands Overview

Docker Configuration

- docker: This command starts the Docker container as specified in the bin/dev-docker file.

Project Installation

- install: Installs all the Python package dependencies and runs the setup script for the project.

Project Execution

- run: Executes the application using Python 3.

Development Commands

- dev: A shorthand for running the project in the development environment. This is particularly useful for quickly testing changes during development.

Test Commands

- test: Executes the unit tests for the application, while also generating an HTML-based code coverage report.
- test-ci: Executes unit tests and prepares the necessary files for CI/CD pipelines. Specifically designed to be run in a CI/CD environment.

Documentation Commands

- doc: Builds the project documentation.
- dev-doc: Builds the project documentation and watches for changes, automatically rebuilding when a change is detected.

Push Commands

- push: A shorthand for adding, committing, and pushing code changes to the remote repository.

Choice of Programming Language and Visualization Libraries

Choosing the right programming language and libraries is crucial for a project's success. These tools affect not just how quickly a project can be developed but also how easily it can be updated or expanded in the future. In this section, we explain why we chose Python and Vega Lite for the Visual Viper (VV) library, focusing on their features, community support, and fit for this project's needs.

Python

Python was chosen for its widespread adoption in the field of data science. It is a high-level, interpreted language that is not only easy to write but also read. Python's large and active community means that a plethora of libraries and tools are readily available for tasks ranging from web development to machine learning. Importantly, Python is open-source, offering an extra layer of flexibility and community engagement.

Vega Lite

We've selected Vega-Lite as our visualization tool influenced by various factors, most importantly API/tool design and level of abstraction. Vega-Lite operates in a framework-agnostic manner and predominantly uses a declarative JSON format for specifying visualizations. This format allows for readability, easy storage, and can even be automatically generated by other tools. Unlike framework-specific libraries that require prerequisite knowledge about frameworks like React or Angular, Vega-Lite offers greater flexibility in deployment [43].

Vega-Lite offers a high-level grammar of graphics that's adequate for both explanatory and exploratory data visualizations. It is based on a JSON format that's platform-independent, thus allowing it to be readily used across various applications. Importantly, Vega-Lite supports various interaction techniques, something often lacking in existing high-level languages. This enables us to construct interactive dashboards and data presentations without delving into low-level code [14]. Vega-Lite's approach enables quick creation of both simple and sophisticated visualizations using a concise grammar [44].

Vega-Lite is designed to be expressive yet concise. It allows for an algebra to compose single-view specifications into multi-view displays, something that expands its application in complex data

visualization scenarios. Its high-level interaction grammar, based on visual elements or data points chosen when input events occur, adds to its expressiveness [14].

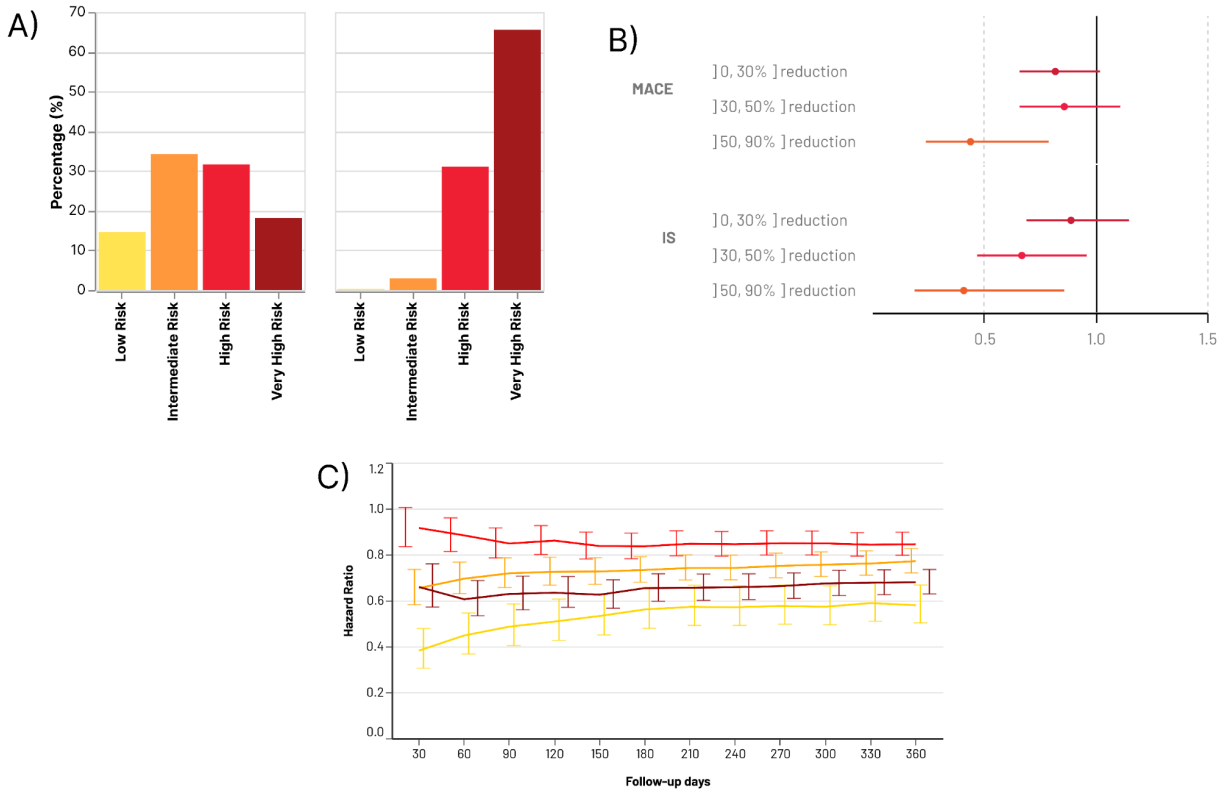


Figure X: Examples of Charts Generated by the Author Using Vega-Lite. Note that in these examples, some graphical details such as legends have been omitted to simplify the visualizations and highlight the most relevant features for the given context. A) A bar chart presented by the author in an oral communication in a national conference [45]. B) A Forest Plot featured in a moderated poster session at an international conference [46]. C: A line chart with error bars that represents the adjusted hazard ratio and respective confidence interval at various time-points, stratified by cohorts, published in a peer-reviewed paper [47].

Documentation

The documentation for the Visual Viper (VV) library was developed using Sphinx, a documentation generator that transforms reStructuredText sources into HTML, LaTeX, PDF, and other formats. This comprehensive guide aims to assist users and developers in understanding the functionalities and architecture of VV.

The documentation is structured into the following key sections:

1. **Getting Started**
    a. How it works
    b. Requirements
    c. Installation
    d. Configuring .env
    e. Commands
    f. Make commands
2. **Architecture**
    a. User Workbench
    b. Package
3. **Development**
    a. Development guidelines
4. **Support**
    a. Glossary
    b. Contacts

The documentation is accessible online at https://visualviper.mtg.pt/ and is tightly integrated into our development pipeline. Specifically, it's hosted on GitLab Pages, ensuring seamless compatibility and automatic updates with each code commit. This integration with GitLab CI/CD serves a dual purpose: it automates the documentation build process and ensures that the documentation is always aligned with the most recent changes to the codebase (Figure X).

Furthermore, we've leveraged AWS Route 53 to route traffic to our custom domain.

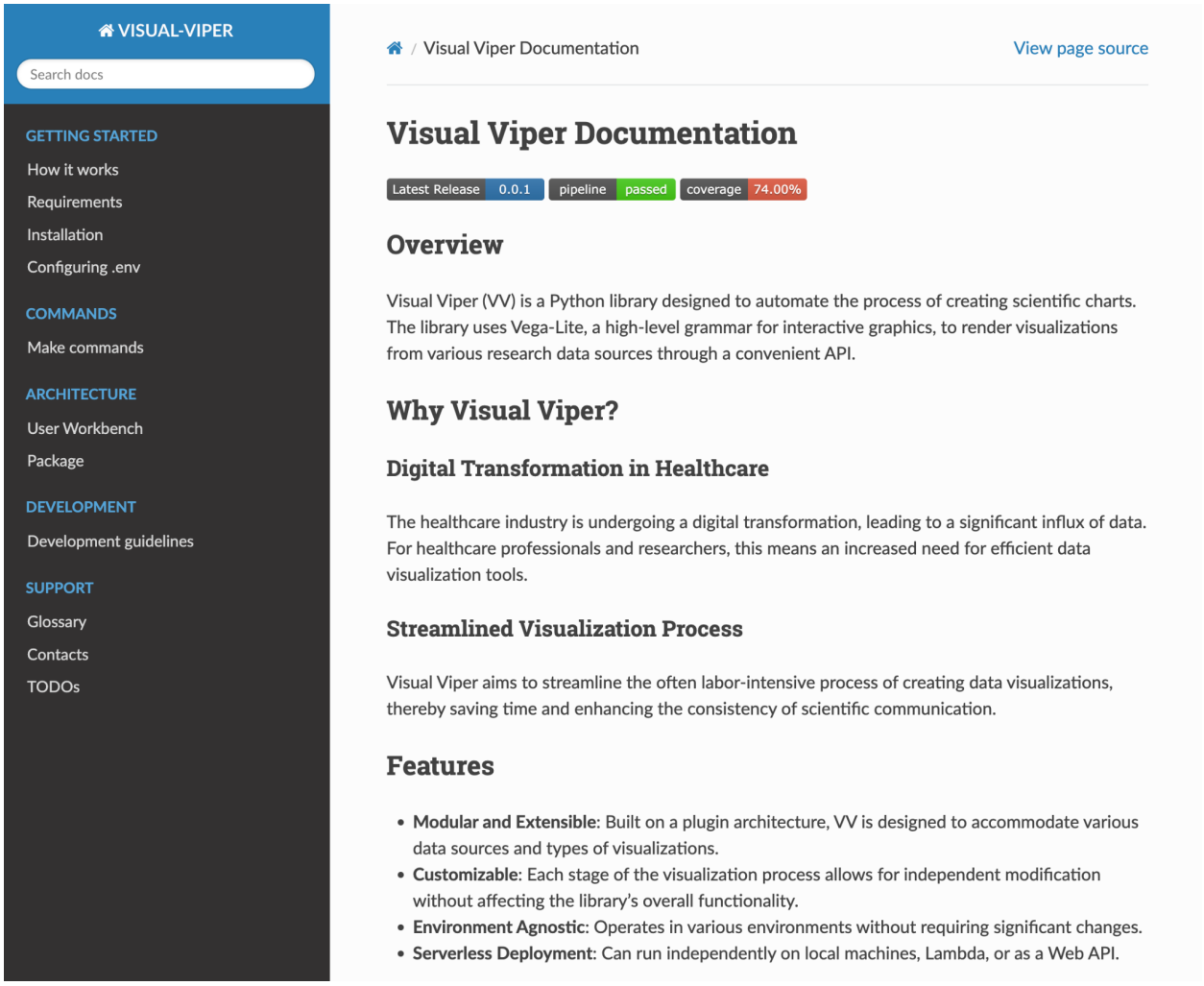


Figure X: Screenshot of the Visual Viper (VV) Documentation Interface

System Architecture

High-level Architecture

To facilitate a comprehensive understanding of the system's architecture, this section presents a high-level overview of the primary classes and their interactions. Figure X below provides a simplified visual representation of the class structure and their relationships. It's important to note that this diagram is an abstraction intended to clarify the core architectural elements; it does not depict every attribute or method within these classes. The diagram has been constructed using PlantUML [48].

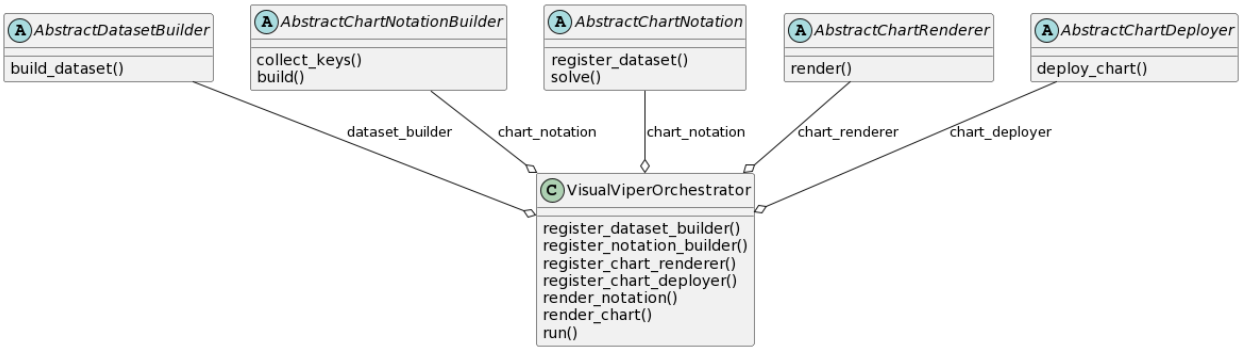


Figure X: High-level Class Diagram of System Architecture

The architecture of the VV system is designed to be both modular and extensible, adhering to the principles of OOP. This design allows for high cohesion among components, low coupling between modules, and promotes scalability. To elaborate on the components that constitute this architecture, we have categorized them into Abstract Classes, Concrete Implementations, and an Orchestrator Class.

Key Classes and Components

The Abstract Classes act as templates or interfaces, specifying what actions must be performed but not how to perform them. Concrete Implementations are subclasses that provide the specific 'how-to', the logic and the behavior. The Orchestrator Class serves as the orchestrating agent that ties these different components together into a cohesive, functioning system.

Abstract Classes

- **AbstractDatasetBuilder**: Provides the framework for constructing datasets.

- **AbstractChartNotation**: Functions as the foundational class for handling chart notations. It provides the methods for registering datasets and solving elements.
- **AbstractChartRenderer**: Serves as the interface for chart rendering mechanisms.
- **AbstractChartDeployer**: Serves as the base class for all chart deployment mechanisms.

Concrete Implementations

- **GoogleSpreadsheetDatasetBuilder**: Specially designed to build datasets from Google Spreadsheets.
- **AltairChartRenderer**: A concrete implementation of AbstractChartRenderer, which specifically uses Vega-Altair for rendering charts [49].
- **GdriveChartDeployer** and **MiroChartDeployer**: These are specialized implementations of AbstractChartDeployer designed to deploy charts on Google Drive and Miro, respectively.

Orchestrator Class

- **VisualViperOrchestrator**: This class manages the interaction between the various components. It references a DatasetBuilder, a ChartNotationBuilder, a ChartRenderer, and a ChartDeployer. This allows the orchestrator to manage the flow of operations.

Component Interactions

The VisualViperOrchestrator serves as the fulcrum around which the entire architecture revolves. It dynamically links to various components, directing the flow of data and operations throughout the system. Subclasses of AbstractDatasetBuilder, AbstractChartNotationBuilder, AbstractChartRenderer, and AbstractChartDeployer, can be plugged into the orchestrator, thereby fulfilling the design goals of modularity and extensibility.

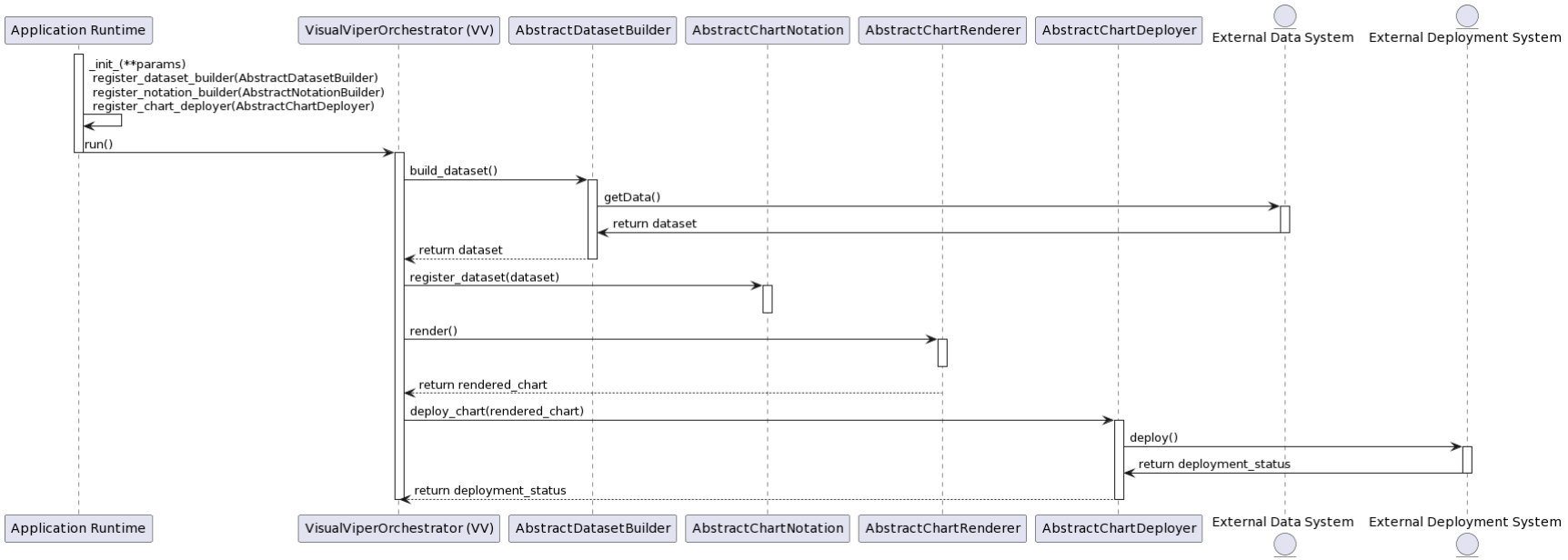


Figure X: Sequence Diagram for Chart Creation and Deployment in Visual Viper Framework

Sequence of Operations

To provide a more concrete understanding of the interactions between components, Figure X presents a sequence diagram illustrating the flow of operations in a typical use case. This diagram was also constructed using PlantUML.

In this sequence diagram:

1. The Application Runtime initializes the VisualViperOrchestrator and registers the required components: AbstractDatasetBuilder, AbstractChartNotation, AbstractChartRenderer, and AbstractChartDeployer.
2. The VisualViperOrchestrator initiates the dataset construction process by calling the build_dataset() method on an AbstractDatasetBuilder object. This object may retrieve data from an external system, abstracted here for generality.
3. Upon successful dataset construction, the VisualViperOrchestrator registers the dataset with AbstractChartNotation for further processing.
4. The VisualViperOrchestrator then invokes the render() method on an AbstractChartRenderer object to create the actual visual representation.
5. Finally, the VisualViperOrchestrator calls the deploy_chart() method on an AbstractChartDeployer object, deploying the rendered chart to an external system.

This sequence of operations encapsulates the VV system's core functionality while emphasizing its modularity and extensibility. It serves as an exemplar flow, illustrating how the system components interact to accomplish the data visualization task.

Description of Components

In this section, we elaborate on the various components of our system, their roles, and how they interact. To give you a comprehensive understanding, we've included a directory structure in Figure X.

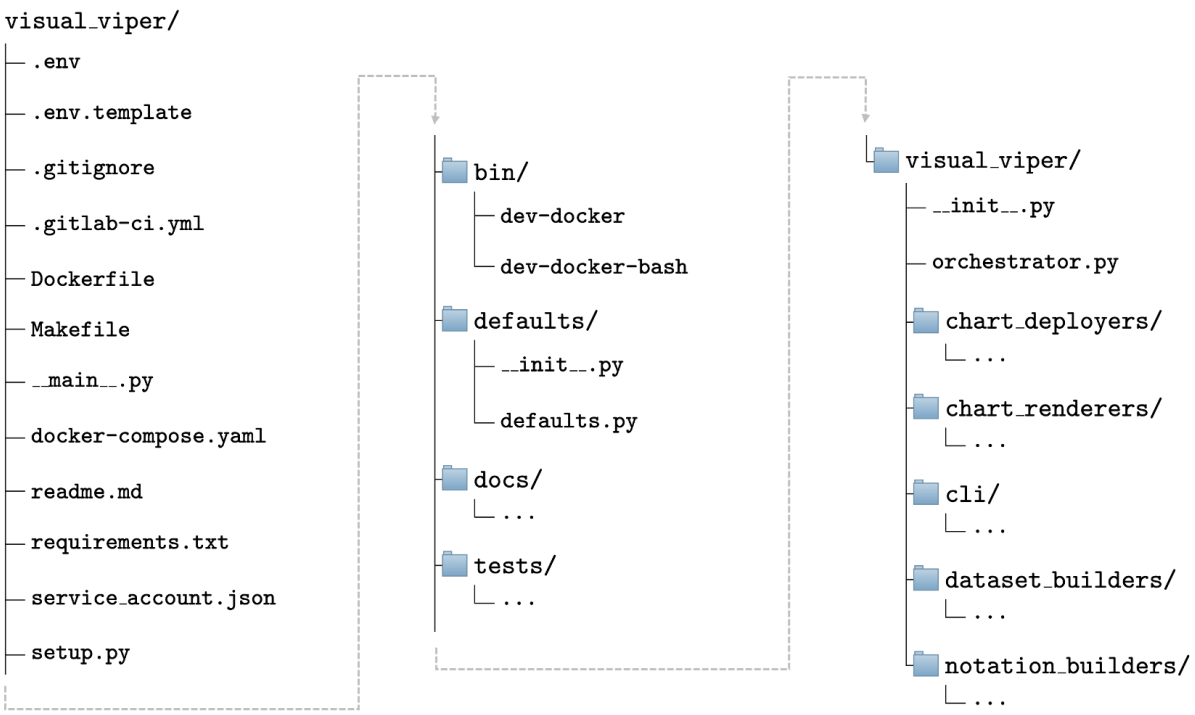


Figure X: Directory structure of the project. The directory structure and the following graphical diagram were generated using VV's directory description and LaTeX diagramming plugins (not described in the current work). For brevity, certain folders have been excluded or their contents omitted from this diagram.

Key Directories and Their Functional Roles

- **defaults/**: This directory contains the default configuration settings, enabling the system to operate with a predefined set of parameters.
- **docs/**: Comprising comprehensive documentation, this directory aids in the effective utilization and understanding of the system.
- **tests/**: This is dedicated to unit testing.
- **visual_viper/:** This directory encapsulates the core functionalities and classes of the project, which include the orchestrators and Command-Line Interface (CLI) mechanisms (which is still under development).

Alignment with Design Philosophy

The directory structure reflects the project's commitment to modularity and extensibility, design philosophies that are integral to the project. The clear demarcation of responsibilities through

specialized directories, such as those for dataset builders, notation builders, chart renderers, and chart deployers, underscores the project's modular and extensible architecture.

Data Flow among Components

To complement the understanding of the system's architecture, Figure X provides a simplified data flow diagram that outlines the relationships and interactions among key components. The diagram was constructed using the DOT language and serves as a conceptual map for how data is passed and manipulated within the system.

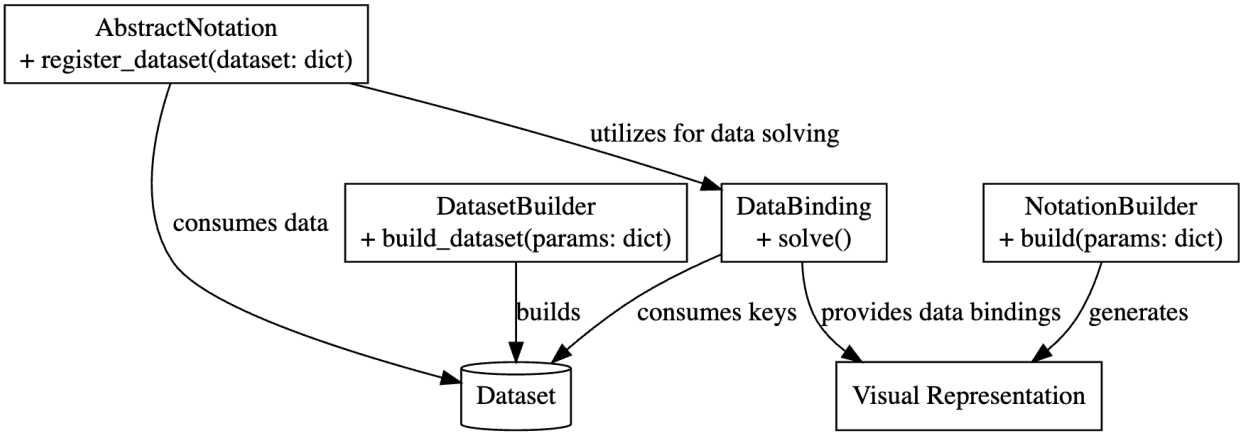


Figure X: Data Flow Diagram of Key System Components of Visual Viper.

As illustrated in Figure X:

- **DatasetBuilder**: Initiates the process by constructing the dataset based on the provided parameters.
- **Dataset**: Serves as the data store which is consumed by both the DataBinding and AbstractNotation classes.
- **NotationBuilder**: Builds the visual representation of the chart, laying out the aesthetics and graphical elements.
- **Visual Representation**: This is the generated graphical layout of the chart, whose appearance is dictated by the NotationBuilder.
- **DataBinding**: Consumes keys from the Dataset to resolve any data dependencies and supplies this resolved data to the visual representation.
- **AbstractNotation**: This class receives data from the Dataset and utilizes the DataBinding class to solve for any data-related calculations.

The DataBinding class plays a crucial role in combining the dataset with its visual representation, ensuring that the data points are correctly mapped onto the chart. On the other hand, the AbstractNotation class establishes the fundamental structure of the chart, including its underlying logic and computations.

This high-level overview allows for easy plug-and-play of different dataset builders, data binding mechanisms, and visual representations, making the system highly modular and extensible.

Modular and Extensible Plugin Architecture

In line with the system's commitment to modularity and extensibility, the architecture of VV features a plugin-based mechanism. This is a crucial subsystem within the broader architecture that enables users to enhance or alter the functionality without changing the core codebase. It facilitates a more dynamic, user-driven ecosystem that aligns with the project's design philosophy. Below we describe the key aspects of this plugin architecture.

Initial Phase Plugins

In the initial phase of development, we aimed to build a set of plugins to meet our most immediate data visualization needs. Specifically, we focused on the following:

- **Google Spreadsheet Data Fetcher**: This plugin will serve the role of a specialized AbstractDatasetBuilder. It will be designed to fetch data from Google Spreadsheets, making it easier for users to source data without manual intervention.
- **Vega-Lite Notation Builders**: A group of specialized AbstractChartNotation plugins will be developed to create notations for Vega-Lite charts. The focus will initially be on generating Forest Plots.
- **Vega-Altair Chart Renderer**: An implementation of AbstractChartRenderer, this plugin will use the Vega-Altair library for rendering the visual representation of the charts.
- **Multi-platform Chart Deployers**: To augment the deployment capabilities, we aimed to create two deployer plugins:
    - **Google Drive Deployer**: Specialized for storing rendered image files in Google Drive, making it convenient for users to access and share their visualizations.
    - **Miro Deployer**: Places the generated charts in Miro boards with a predefined layout, aiding in the interpretation and comparison of the charts.

Our plugin architecture is designed for future expansion, both by our team and external contributors. It allows for:

- **User Customization**: Users can tailor the software to their needs by adding or removing features.
- **Easy Maintenance**: Since the core code is not altered when adding plugins, system updates are more straightforward.
- **Community Input**: The architecture is open to contributions from others, allowing for further enhancements.

This architecture supports the previously described low coupling by allowing independent development and integration of plugins, and high cohesion by ensuring each plugin is a self-contained, focused unit of functionality.

Implementation Details

This section delves into the core classes that make up the architecture of our VV system. Each class has a specific role and set of responsibilities, designed to facilitate modular and extensible software development. It describes the purpose of these classes, their primary methods, and example use cases that highlight their functionality. The intent is to provide a comprehensive understanding of how these components interact to support the system's goals, including adaptability and the ability to accommodate future plugin development by our team or other contributors.

Core Classes and their Responsibilities

In the following subsections, we will examine each core class to detail its role and responsibilities in the system architecture.

The ‘dataset_builders’ Module

The dataset_builders module serves as the core for data acquisition in the Visual Viper Framework. This module offers an abstract class, AbstractDatasetBuilder, designed to be extended for specific data sourcing implementations. Its design promotes low coupling, making it easier to integrate new data sources.

The ‘AbstractDatasetBuilder’ Class

The first class in the architecture is AbstractDatasetBuilder, which is an abstract class acting as a blueprint for all dataset builders. The class declares a method build_dataset(params=None), which subclasses should implement to provide the actual dataset-building functionality (Listing X). This abstract class is crucial in achieving low coupling as it ensures that other components of the system need not know the specific dataset builder that will be used.

```python
class AbstractDatasetBuilder:

@abc.abstractmethod
def build_dataset(self, params=None):
```

```
raise NotImplementedError()
```

Listing X: Code snippet showing the AbstractDatasetBuilder class, which provides a method interface for building datasets.

The ‘Key’ Class

Within the dataset_builders module, there's a simple but critical class named Key (Listing X). This class serves to encapsulate key-value pairs used for data retrieval. The Key class has an initializer that takes two arguments: key and an optional src parameter. Here, key represents the data attribute, while src can be used to specify the data source.

```
Python
class Key():

def _init_(self, key, src=None) -> None:
self.key = key
self.src = src
```

Listing X: Code snippet showing the Key class used for encapsulating data retrieval attributes.

The utility of the Key class becomes more evident when used in conjunction with the notation_builders module, where it plays an instrumental role in linking dataset attributes to visual elements in a chart.

The GoogleSpreadsheetDatasetBuilder Class

Extending the AbstractDatasetBuilder is the GoogleSpreadsheetDatasetBuilder class (Listing X). This concrete implementation utilizes the Google Sheets API to fetch data. The class uses the gspread library and OAuth 2.0 for secure and efficient data retrieval. One of the significant advantages of this class is its ability to handle multiple named ranges across multiple worksheets.

```
Python
import gspread
```

```
from google.oauth2 import service_account as sa
from googleapiclient.discovery import build

from .abstract_dataset_builder import *

class GoogleSpreadsheetDatasetBuilder(AbstractDatasetBuilder):

DEFAULT_SA_PATH = "./service_account.json"
DEFAULT_SCOPES = ['https://www.googleapis.com/auth/drive']

def _init_(self, file_id=None, sa_path=None) -> None:

self.file_id = file_id
self.sa_path = sa_path or self.DEFAULT_SA_PATH
self.auth = sa.Credentials.from_service_account_file(
self.sa_path,
scopes=self.DEFAULT_SCOPES
)
self.dataset = dict()

def build(self, params=None, ws_index=0):

gs = gspread.service_account(self.sa_path)

range_sets = dict()

for el in params["ranges"]:
if not isinstance(el, tuple):
el = (el, self.file_id)
named_range, file_id = el
if not file_id in range_sets:
```

```
range_sets[file_id] = []
range_sets[file_id].append(named_range)
for file_id, ranges in range_sets.items():
sheet = gs.open_by_key(file_id)
worksheet = sheet.get_worksheet(ws_index)
response = worksheet.batch_get(
ranges,
value_render_option="UNFORMATTED_VALUE",
)
response = {
ranges[i]: response[i][0][0] for i in range(len(response))
}
self.dataset.update(response)
return self.dataset
```

Listing X: Code snippet showing the GoogleSpreadsheetDatasetBuilder class, responsible for building datasets from Google Sheets.

The ‘notation_builders’ Module

The notation_builders module encapsulates the logic required for constructing the chart notations and solving data dependencies for the actual visualization. Two abstract classes form the core of this module: AbstractChartNotationBuilder and AbstractChartNotation.

The ‘AbstractChartNotationBuilder’ Class

AbstractChartNotationBuilder is an abstract class that acts as a blueprint for all chart notation builders (Listing X). It declares methods like build(params=None) that subclasses need to implement to provide the actual chart-building functionality. The class uses an internal property bindings, designed to be overridden in subclasses, that links the dataset keys to visual elements in a chart.

The AbstractChartNotationBuilder class also introduces a collect_keys() method, which traverses all the bindings and collects the Key instances, serving as a bridge to the dataset_builders module. This method ensures that all necessary data points can be fetched efficiently from the dataset.

```python
class AbstractChartNotationBuilder:
# ...

def _init_(self, bindings=None, id=None, opts=None):
# ...

@property
def bindings(self):
raise NotImplementedError()

def collect_keys(self, dataset):
# ...

@abc.abstractmethod
def build(self, params=None) -> dict:
raise NotImplementedError()
```

Listing X: Code snippet showing the AbstractChartNotationBuilder class, which serves as the framework for building chart notations.

The 'AbstractChartNotation' Class

The AbstractChartNotation class functions as a complementary element to the AbstractChartNotationBuilder class. This class registers the dataset and contains a solve() method. The solve() method uses instances of the Key class from the dataset_builders module to fetch the necessary data points, thereby linking the chart notation to the actual data (Listing X).

```python
class AbstractChartNotation:

def _init_(self):
self.dataset = {}

def register_dataset(self, dataset):
# ...

def solve(self, el):
# ...
```

Listing X: Code snippet showing the AbstractChartNotation class, which registers the dataset and provides a method for solving notation elements.

The ‘ForestPlot’ Class

The ForestPlot class (Listing X) is a concrete implementation that inherits from AbstractChartNotationBuilder. It specializes in building Forest Plots, a type of chart that is commonly used to visualize grouped data points in a graphical format. The class provides the option to include labels for different measures (hr, lo, hi) and customizes them as needed.

```python
from .abstract_notation_builder import AbstractChartNotationBuilder
from .forest_plot_binding_notation import ForestPlotBinding

class ForestPlot(AbstractChartNotationBuilder):

OPTS = dict(
labels = dict(
  hr="HR",
  lo="CI Low",
  hi="CI High",
```

```
 )
)

@property
def bindings(self):
return [
ForestPlotBinding(
measure="",
hr=self.opts["labels"]["hr"],
lo=self.opts["labels"]["lo"],
hi=self.opts["labels"]["hi"],
),
*self._bindings
]

def build(self, params=None) -> dict:
base_schema = {
"$schema": "https://vega.github.io/schema/vega-lite/v5.json",
"data": {
"values": [
]
},
#...
}
notation = base_schema.copy()
values = [binding.solved_data for binding in self.bindings]
notation["data"]["values"] = values
return notation
```

Listing X: Code snippet showing the ForestPlot class, responsible for building the notation for Forest Plots.

The ForestPlot class overrides the bindings property, providing a default ForestPlotBinding instance that serves as a blueprint for all bindings related to this specific type of chart. It also defines the build(params=None) method to generate the notation for rendering the chart using the Vega-Lite schema.

The ForestPlotBinding Class

This class inherits from AbstractChartNotation and serves to hold and solve the data points necessary for a Forest Plot. Unlike the generic AbstractChartNotation, ForestPlotBinding has additional properties specific to Forest Plots, such as hr (Hazard Ratio), lo (Low Confidence Interval), and hi (High Confidence Interval), as can be seen in Listing X.

The ForestPlotBinding class introduces the data and solved_data properties. The data property returns the initial (unsolved) key-value pairs, whereas the solved_data property uses the inherited solve() method to get the actual data points from the dataset. These properties bridge the gap between data sourcing and data representation in the chart.

```python
import json
from .abstract_chart_notation import AbstractChartNotation

class ForestPlotBinding(AbstractChartNotation):

 def __init__(self, measure, hr, lo, hi) -> None:
  super().__init__()
  self.measure = measure
  self._hr = hr
  self._lo = lo
  self._hi = hi

 @property
 def data(self) -> dict:
  return dict(
   measure=self.measure,
   lo=self._lo,
   hr=self._hr,
```

```
 hi=self._hi,
 )

@property
def solved_data(self) -> dict:
 return dict(
 measure=self.measure,
 lo=self.lo,
 hr=self.hr,
 hi=self.hi,
 )

@property
def lo(self):
 return self.solve(self._lo)

@property
def hr(self):
 return self.solve(self._hr)

@property
def hi(self):
 return self.solve(self._hi)

def items(self):
 yield ("hr", self._hr)
 yield ("lo", self._lo)
 yield ("hi", self._hi)

def __repr__(self):
 return f"hr:{self.hr}, lo:{self.lo}, hi:{self.hi}"
```

Listing X: Code snippet showing the ForestPlotBinding class, which encapsulates the logic for holding and solving data points specific to Forest Plots.

Summary Diagram for the ‘notation_builders’ Module

To sum up the relationships between these classes, please refer to the following class diagram depicted in Figure X.

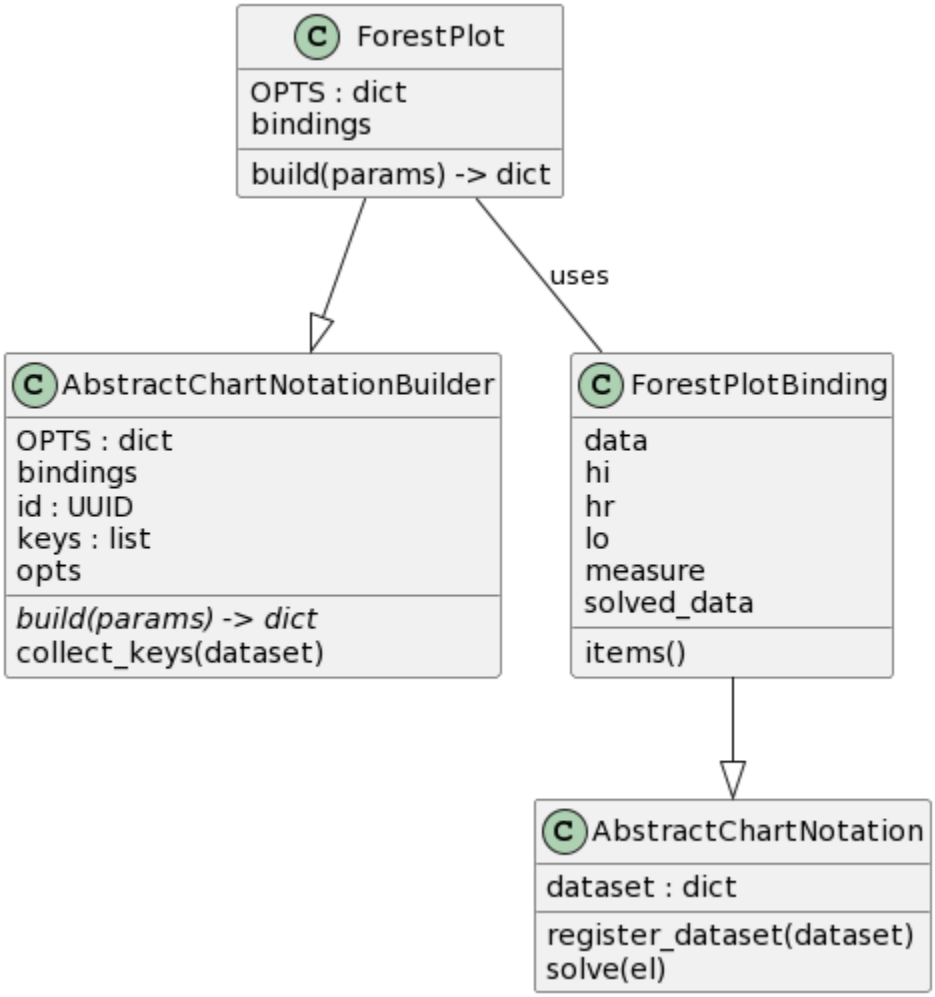


Figure X: Class diagram of the classes included in the ‘notation_builders’ module.

The ForestPlot class inherits from AbstractChartNotationBuilder, while ForestPlotBinding inherits from AbstractChartNotation. The ForestPlot class uses instances of ForestPlotBinding to build the chart, leveraging the options and methods provided by the parent classes.

Again, this setup ensures low coupling and high cohesion, thus aligning well with the principles of clean architecture.

The ‘chart_renderers’ Module

The chart_renderers module is a pivotal component in the VV Framework responsible for rendering visualizations. The module houses an abstract class, AbstractChartRenderer, which is designed to be extended by specific rendering engines.

The ‘AbstractChartRenderer’ Class

The backbone of the chart_renderers module is the AbstractChartRenderer class (Listing X). It is an abstract class serving as a blueprint for all chart rendering implementations. It declares a method render(notation=None, params=None), which is expected to be implemented by subclasses to provide the actual chart rendering functionality. This design pattern ensures that other system components do not need to be aware of the specific renderer in use, thereby achieving low coupling.

```python
import abc

class AbstractChartRenderer:
"""
Documentation TBD
"""
def _init_(self) -> None:
pass

def render(self, notation=None, params=None):
raise NotImplementedError
```

Listing X: Code snippet showing the AbstractChartRenderer class, which provides a method interface for rendering charts.

The ‘AltairChartRenderer’ Class

Extending the AbstractChartRenderer is the AltairChartRenderer class (Listing X). This specialized class serves as a wrapper for Vega-Altair, utilizing the Altair library to perform the rendering of visualizations. One of its key features is the flexibility of outputting the rendered

chart through a file pointer (fp). This fp can be either a string representing a file path or an in-memory file-like object such as a StringIO object. This offers versatility for different use-cases, including real-time chart generation and embedding charts into web applications.

By overriding the render method, this class takes in a chart notation and a file pointer (fp) parameter. The chart is generated from the notation and saved in SVG format to the location pointed to by fp.

```python
import altair
from .abstract_chart_renderer import AbstractChartRenderer

class AltairChartRenderer(AbstractChartRenderer):
"""
Documentation TBD
"""
def _init_(self) -> None:
super().__init__()

def render(self, fp, notation=None, params=None):
chart = altair.Chart.from_dict(notation)
chart.save(fp, format="svg")
return fp
```

Listing X: Code snippet showing the AltairChartRenderer class, which acts as a wrapper for Vega-Altair and is responsible for rendering charts using the Altair library.

The ‘chart_deployers’ Module

The chart_deployers module serves as the component in the VV Framework that specializes in the deployment of visualizations. This module introduces an abstract class, AbstractChartDeployer, which acts as a blueprint for various chart deployment strategies, including concrete implementations like GdriveChartDeployer and MiroChartDeployer. These implementations provide specialized mechanisms for deploying charts to Google Drive and Miro boards, respectively. The design of the module encourages low coupling, allowing easy integration of different deployment methods without altering the core framework.

The AbstractChartDeployer Class

The foundational class in this architecture is AbstractChartDeployer, an abstract class that defines the standard for all chart deployers (Listing X). It declares a method deploy_chart(buffer, params=None), which is designed to be overridden by subclasses to offer the actual chart deployment functionality.

```python
import io
import abc

class AbstractChartDeployer:
"""
Documentation TBD
"""
@abc.abstractmethod
def deploy_chart(buffer: io.BytesIO, params=None) -> None:
"""
Documentation TBD
"""
raise NotImplementedError()
```

Listing X: Code snippet showing the AbstractChartDeployer class, which provides a method interface for deploying charts.

The ‘GdriveChartDeployer’ Class

Extending the AbstractChartDeployer is the GdriveChartDeployer class (Listing X). This concrete implementation leverages Google Drive's API for the deployment of visualizations. It uses the google-auth and google-api-python-client libraries for secure and authenticated communication with Google Drive.

Python

```
class GdriveChartDeployer(AbstractChartDeployer):

DEFAULT_SA_PATH = "./service_account.json"
DEFAULT_SCOPES = ['https://www.googleapis.com/auth/drive']

DEFAULT_FILE_NAME = "filename.svg"

def _init_(self, folder_id, mime_type=None, sa_path=None, params=None):
self.sa_path = sa_path or self.DEFAULT_SA_PATH
self.auth = sa.Credentials.from_service_account_file(
self.sa_path,
scopes=self.DEFAULT_SCOPES
)
self.drive_service = build('drive', 'v3', credentials=self.auth)
self.folder_id = folder_id
self.file_name    =    params.get("filename")    if    params    else
self.DEFAULT_FILE_NAME
self.mime_type = mime_type


def deploy(self, fp):

files = []

file_metadata = {
'name': self.file_name,
'parents': [self.folder_id],
}

if hasattr(fp, 'getvalue'):
content = BytesIO(fp.getvalue().encode("utf-8"))
```

```
elif isinstance(fp, (str, bytes, os.PathLike)):
with open(fp, 'rb') as file:
content = file.read()
else:
raise TypeError("fp must be a file-like object or a file path")

#...

response = request.execute()

return response.get('id')
```

Listing X: Code snippet showing the GdriveChartDeployer class, responsible for deploying charts to Google Drive.

The 'MiroChartDeployer' Class

Another subclass of AbstractChartDeployer is the MiroChartDeployer class (Listing X). This specialized class is designed for deploying charts to Miro boards. It uses Miro's REST API for communication with Miro boards.

```
class MiroChartDeployer (AbstractChartDeployer):

  DEFAULT_IMAGE_WIDTH=2000
  DEFAULT_IMAGE_X_POSITION=0
  DEFAULT_IMAGE_Y_POSITION=0
  DEFAULT_IMAGE_TITLE="Default Image Title"

  DEFAULT_LAYOUT_COLUMNS=2
  DEFAULT_LAYOUT_COLUMN_SPACING=150
  DEFAULT_LAYOUT_ROW_SPACING=150

  def __init__(self, board_id, token, params=None):
    self.board_id = board_id
    self.oauth_token = token
    self.parent_id = params.get("parent_id") if params else None

    self.image_title = params.get("image_title") if params else self.DEFAULT_IMAGE_TITLE
    self.image_width = params.get("image_width") if params else self.DEFAULT_IMAGE_WIDTH
    self.image_x_position = params.get("image_x_position") if params else self.DEFAULT_IMA
    self.image_y_position = params.get("image_y_position") if params else self.DEFAULT_IMA

    self.layout_columns = params.get("layout_columns") if params else self.DEFAULT_LAYOUT_
    self.layout_x_position = params.get("layout_x_position") if params else self.DEFAULT_I
    self.layout_row_spacing = params.get("layout_row_spacing") if params else self.DEFAULT
    self.layout_column_spacing = params.get("layout_column_spacing") if params else self.D

    self.deployment_counter = 0
    self.row_elements_height = []
    self.last_widget_id = None

  def calc_position(self,last_widget_id=None):

    #...

  def get_widget_attribute(self, widget_id, attribute_path):
    #...


  def deploy(self, fp,
    #...
```

Listing X: Code snippet showing the MiroChartDeployer class, specialized in deploying charts to Miro boards.

The MiroChartDeployer class encapsulates a set of attributes and methods designed to automate the deployment of charts onto a Miro board. Within the class, several attributes warrant particular attention for their role in shaping the class functionality:

- Default Constants: A suite of class-level constants prefixed with DEFAULT_ is defined to establish fallback values for various properties.
- deployment_counter: This attribute serves as a counter of the number of deployments executed through the deploy method.
- row_elements_height: This list-based attribute is specifically designed to capture the height of individual elements within each row on the Miro board. The data stored in this list

informs the layout calculations, facilitating the arrangement of multiple widgets on the board.

- last_widget_id: After each successful deployment, the ID of the last deployed widget is stored in this attribute for later manipulation (namely getting the widget height for layout calculations).

The deploy(fp) method is responsible for actually uploading a chart as an image widget onto a Miro board. It accepts the parameter fp, which stands for file pointer.

```python
class MiroChartDeployer(AbstractChartDeployer):

DEFAULT_IMAGE_WIDTH=2000
DEFAULT_IMAGE_X_POSITION=0
DEFAULT_IMAGE_Y_POSITION=0
DEFAULT_IMAGE_TITLE="Default Image Title"

DEFAULT_LAYOUT_COLUMNS=2
DEFAULT_LAYOUT_COLUMN_SPACING=150
DEFAULT_LAYOUT_ROW_SPACING=150

def __init__(self, board_id, token, params=None):
self.board_id = board_id
self.oauth_token = token
self.parent_id = params.get("parent_id") if params else None

self.image_title = params.get("image_title") if params else
self.DEFAULT_IMAGE_TITLE
self.image_width = params.get("image_width") if params else
self.DEFAULT_IMAGE_WIDTH
```

```
self.image_x_position = params.get("image_x_position") if params else
self.DEFAULT_IMAGE_X_POSITION
self.image_y_position = params.get("image_y_position") if params else
self.DEFAULT_IMAGE_Y_POSITION

self.layout_columns = params.get("layout_columns") if params else
self.DEFAULT_LAYOUT_COLUMNS
self.layout_x_position = params.get("layout_x_position") if params else
self.DEFAULT_IMAGE_X_POSITION
self.layout_row_spacing = params.get("layout_row_spacing") if params
else self.DEFAULT_LAYOUT_ROW_SPACING
self.layout_column_spacing = params.get("layout_column_spacing") if
params else self.DEFAULT_LAYOUT_COLUMN_SPACING

self.deployment_counter = 0
self.row_elements_height = []
self.last_widget_id = None

def calc_position(self,last_widget_id=None):

#...

def get_widget_attribute(self, widget_id, attribute_path):
#...


def deploy(self, fp,
#...
```

Listing X: Code snippet showing the deploy method of the MiroChartDeployer class.

The calc_position method is designed to calculate the position for placing a new image widget on the Miro board according to the parameters defined for a given structured layout such as number of columns and column and row spacing.

The get_widget_attribute method serves the purpose of fetching specific attributes from a widget already deployed on the Miro board. It takes two parameters: widget_id, the ID of the widget from which an attribute needs to be fetched, and attribute_path, a list describing the nested keys to reach the target attribute in the widget's data structure. It is specifically used to get the height of the last widget which is essential for determining how much vertical space a row of widgets will occupy in a structured layout with multiple rows and columns. Specifically, the height attribute helps to calculate the next y-coordinate (image_y_position) for starting a new row of widgets.

In the calc_position method, after each widget deployment, the height of the last deployed widget is fetched and stored in the row_elements_height list. When it's time to move to a new row, i.e., when the number of widgets in the current row equals the predefined maximum number of columns (layout_columns), the maximum height in the row_elements_height list is used to calculate the new y-coordinate.

Workflow Demonstration

To provide a clearer understanding of how the VV library operates in a real-world scenario, this section walks through a complete workflow of data visualization automation. The illustration starts from obtaining data from a specific data source to rendering a chart and ultimately deploying it to a Miro board and Google Drive.

Stage 1: Data Retrieval

In this example, we consider healthcare data obtained from Google Spreadsheets, which serve as our data source (see Figure X). The spreadsheets are organized into named ranges and contain essential metrics for Cox Proportional Hazards Models (see Figure X). These metrics include hazard ratios along with their corresponding confidence intervals for various covariates. The VV library leverages the GoogleSpreadsheetDatasetBuilder class to fetch this data, which is then transformed into a format suitable for chart rendering.

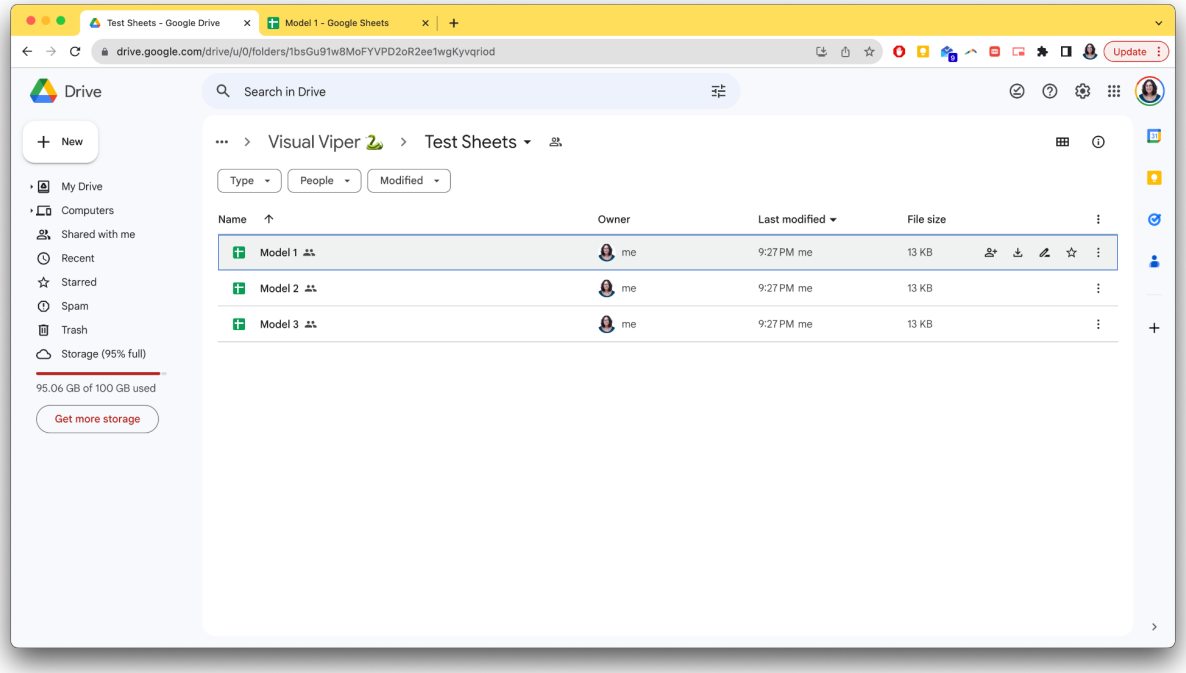
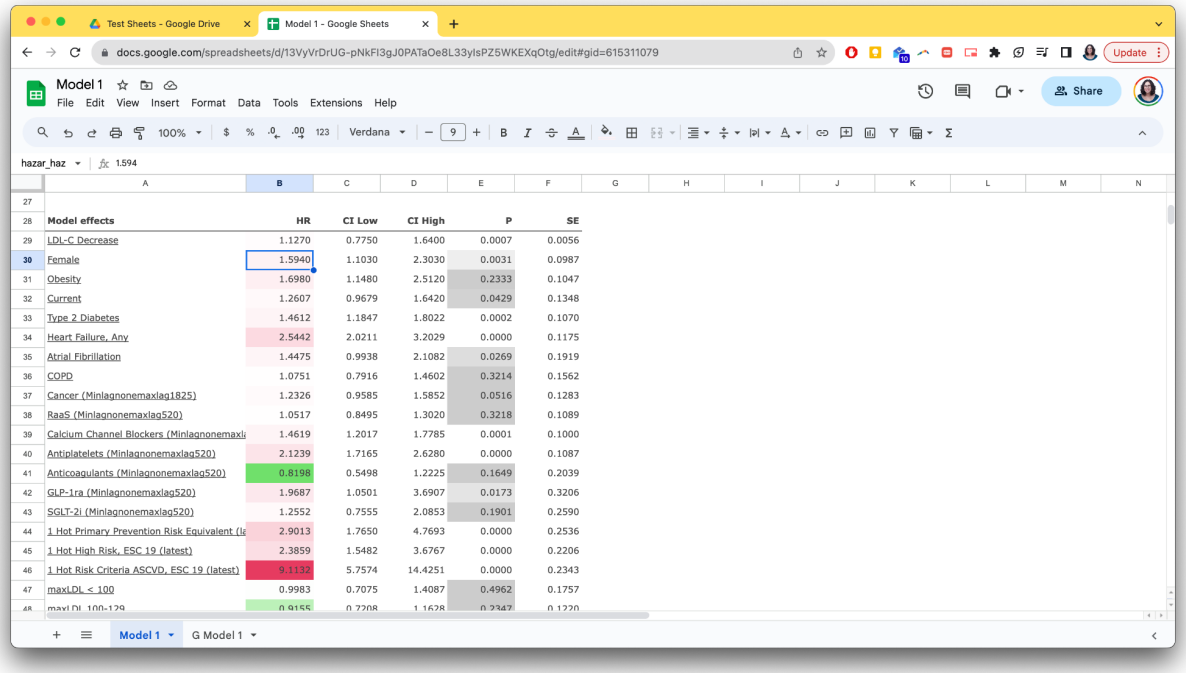


Figure X: Folder Containing Google Spreadsheets for the example.



Model effects	HR	CI Low	CI High	P	SE
LDL-C Decrease	1.1270	0.7750	1.6400	0.0007	0.0056
Female	1.5940	1.1030	2.3030	0.0031	0.0987
Obesity	1.6980	1.1480	2.5120	0.2333	0.1047
Current	1.2607	0.9679	1.6420	0.0429	0.1348
Type 2 Diabetes	1.4612	1.1847	1.8022	0.0002	0.1070
Heart Failure, Any	2.5442	2.0211	3.2029	0.0000	0.1175
Atrial Fibrillation	1.4475	0.9938	2.1082	0.0269	0.1919
COPD	1.0751	0.7916	1.4602	0.3214	0.1562
Cancer (Minlagnonemaxlag1825)	1.2326	0.9585	1.5852	0.0516	0.1283
RaaS (Minlagnonemaxlag520)	1.0517	0.8495	1.3020	0.3218	0.1089
Calcium Channel Blockers (Minlagnonemaxl	1.4619	1.2017	1.7785	0.0001	0.1000
Antiplatelets (Minlagnonemaxlag520)	2.1239	1.7165	2.6280	0.0000	0.1087
Anticoagulants (Minlagnonemaxlag520)	0.8198	0.5498	1.2225	0.1649	0.2039
GLP-1ra (Minlagnonemaxlag520)	1.9687	1.0501	3.6907	0.0173	0.3206
SGLT-2i (Minlagnonemaxlag520)	1.2552	0.7555	2.0853	0.1901	0.2590
1 Hot Primary Prevention Risk Equivalent (la	2.9013	1.7650	4.7693	0.0000	0.2536
1 Hot High Risk, ESC 19 (latest)	2.3859	1.5482	3.6767	0.0000	0.2206
1 Hot Risk Criteria ASCVD, ESC 19 (latest)	9.1132	5.7574	14.4251	0.0000	0.2343
maxLDL < 100	0.9983	0.7075	1.4087	0.4962	0.1757
maxLDL 100-129	0.9155	0.7208	1.1628	0.2347	0.1220

Figure X: Spreadsheet Content for Cox Proportional Hazards Model 1 of the example.

Stage 2: Chart Configuration

Once the data is retrieved and prepared, the next step involves defining the chart specifications. For our example, we aim to visualize the metrics from the Cox Proportional Hazards Models in the form of a Forest Plot. VV library allows this by leveraging Vega-Lite, a high-level JSON syntax for generating visualizations.

To accomplish this, the ForestPlot class is employed. This class is a concrete implementation that inherits from AbstractChartNotationBuilder. It specializes in constructing Forest Plots by setting the necessary parameters, configurations, and data values. Moreover, the ForestPlotBinding class plays a vital role. This class inherits from AbstractChartNotation and is designed to hold and resolve the data points essential for a Forest Plot.

The JSON configuration for our Forest Plot, generated by the aforementioned classes, includes specific elements that are essential for visualizing the hazard ratios and their corresponding confidence intervals for the listed covariates (see Listing X). The JSON file lays out not only the type of chart to be generated but also fine-grains the aesthetic details such as titles, subtitles, and axes properties.

The configuration also takes advantage of Vega-Lite's layering capabilities. This enables us to represent multiple elements like the confidence intervals and hazard ratios within the same plot while maintaining visual coherence. Each metric, such as 'Age', 'Sex', 'Obesity', etc., is represented as a horizontal line in the Forest Plot, with markers indicating the confidence interval and a point indicating the hazard ratio. For this example we will use only three covariates.

Listing X: JSON Configuration for Forest Plot.

```javascript
{
 "$schema": "https://vega.github.io/schema/vega-lite/v5.json",
 "data": {
  "values": [
   {"measure": "LDL-C decrease", "lo": 1.127, "hr": 0.775, "hi": 1.64},
   {"measure": "Age", "lo": 1.594, "hr": 1.103, "hi": 2.303},
   {"measure": "Female", "lo": 1.698, "hr": 1.148, "hi": 2.512}
  ]
 },
 "title": {
  "text": "Title 1",
  "fontSize": 12,
  "subtitle": "Subtitle 1"
 },
 "facet": {
  "row": {
   "field": "cohort",
   "header": {
    "labelAngle": 360,
    "labelFontSize": 10.5
   }
  }
 },
 "spec": {
  "encoding": {
   "y": {
```

```
  "field": "measure",
  "type": "nominal",
  "axis": {
   "labelFontSize": 10
  }
 },
 "x": {
  "type": "quantitative",
  "axis": {
   "labelFontSize": 9
  }
 }
},
"layer": [
 {
  "mark": {
   "type": "rule"
  },
  "encoding": {
   "x": {
    "field": "lo"
   },
   "x2": {
    "field": "hi"
   }
  }
 }
 // Additional layers truncated for brevity
]
},
"config": {
 "background": "#F7F7F7",
 "font": "Barlow, Lato, Roboto, sans-serif"
}
```

```
}
```

Stage 3: Chart Rendering

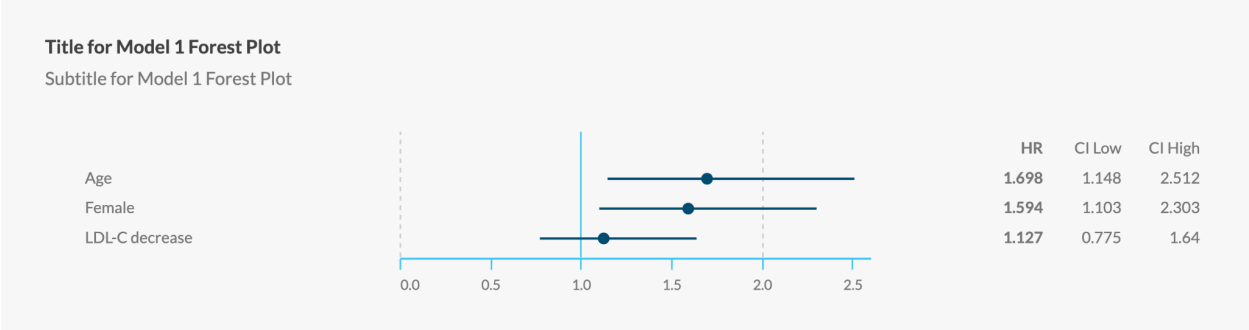
With the data properly set and the chart configuration in place, we are now ready to render the Forest Plot. To achieve this, we make use of altair-save, an external package that integrates with our architecture.

The core class responsible for this task is AltairChartRenderer, which extends the AbstractChartRenderer. This specialized class serves as a wrapper for Vega-Altair, utilizing the Altair library to perform the rendering of visualizations. In this architecture, the AltairChartRenderer takes the JSON configuration produced by ForestPlot and ForestPlotBinding classes and uses it to generate the visual representation of the Forest Plot.

In most of our workflows, the AltairChartRenderer outputs a file pointer (fp), typically an in-memory file-like object such as a StringIO object. This allows for easy manipulation and further use of the chart in the subsequent steps of deployment. However, the renderer is also flexible enough to output the chart as a saved image file, supporting various formats like SVG, for example.

In Figure X, you will find a sample of what the rendered Forest Plot looks like.

Figure X: Rendered Forest Plot for Model 1 of the example.



Stage 4: Deployment

The final stage of the workflow involves deploying the rendered Forest Plot to a Miro board and Google Drive. To achieve this, the VV library employs the specialized classes MiroBoardDeployer and GoogleDriveDeployer.

Both classes automatically handle the upload process, ensuring that the visualizations are transferred to their designated platforms. This streamlined approach makes the visualizations readily accessible for team collaboration (see Figure X).

When deploying to a Miro board, the MiroBoardDeployer class offers additional layout capabilities. Specifically, it arranges the Forest Plots in a grid formation based on a user-defined number of columns. In our example, the Forest Plots are laid out in a two-column grid, facilitating a visually organized comparison of different plots (see Figure X).

For more extensive projects that require the deployment of a large number of Forest Plots, the MiroBoardDeployer is equally capable. It can layout tens of plots on the Miro board in an organized grid, allowing for seamless interpretation and analysis of a more extensive data set (see Figure X for a different example of Forest Plots deployed in Miro with tens of plots).

Figure X: Forest SVG files on Google Drive, uploaded by the Visual Viper agent.

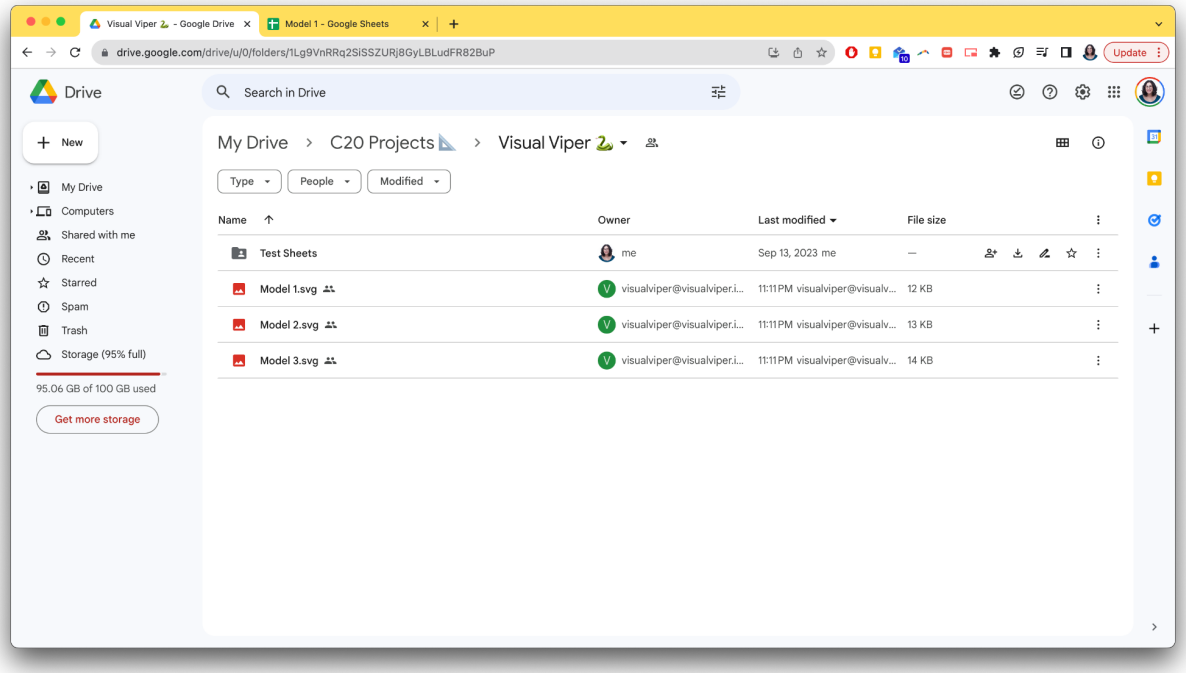


Figure X: Forest Plots for Models 1-3 of the example on Miro Board.

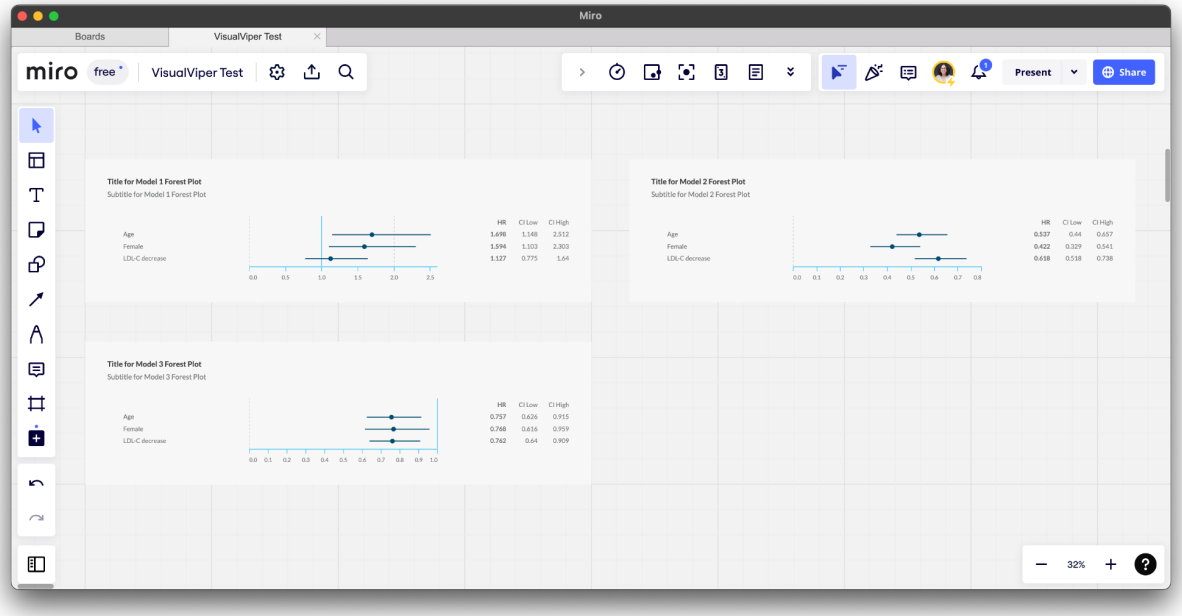
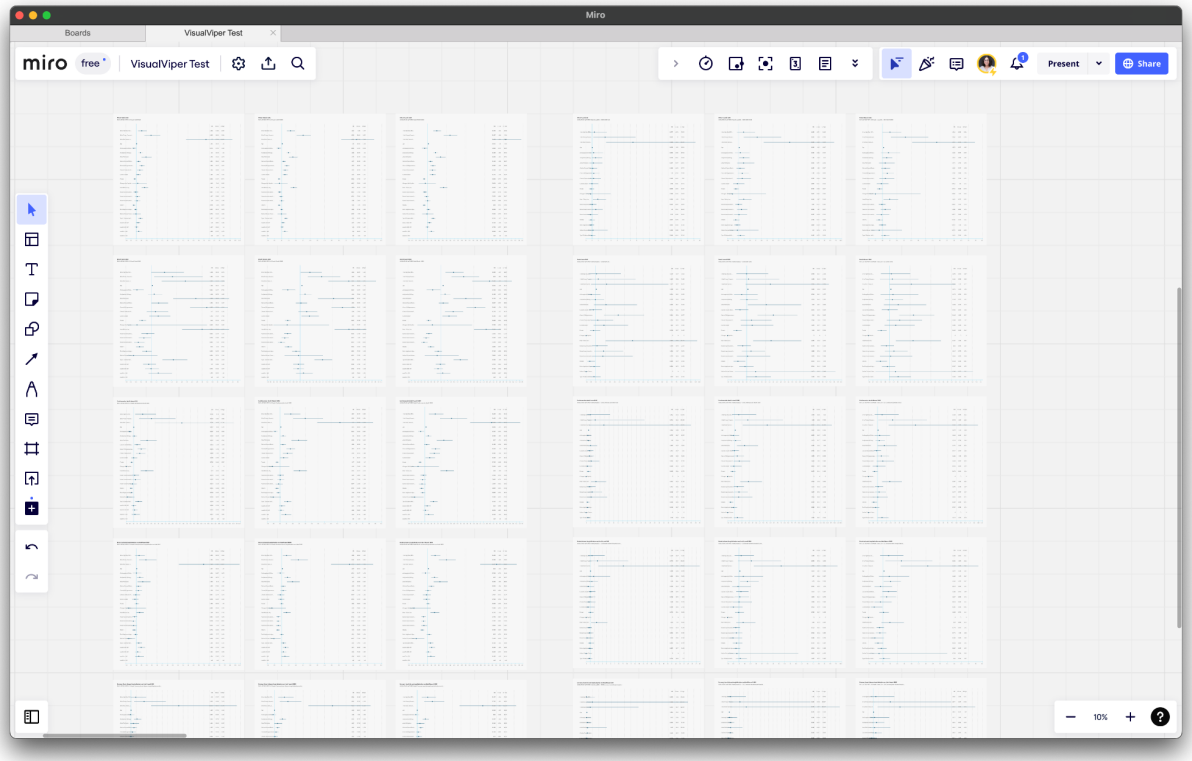


Figure X: Different example of Forest Plots deployed in Miro with tens of plots laid out in a grid.


Miro
Boards
VisualViper Test
miro
free
VisualViper Test
Present
Share
10%

Evaluation Results

This section presents the effectiveness of the VV Python library in improving healthcare data visualization for academic research. We compare its performance against traditional methods. Our focus is on two metrics: "Time-to-First-Chart Draft" and "Time-to-Final-Chart."

The data for this evaluation was collected from a project that involved producing visual representations from a set of 72 spreadsheets. These times were captured using Monday.com, following a well-established practice within the organization where the author works for project management, including time-tracking. Time measurements for VV were taken using Python's time library, by calculating the delta of time between the start and completion of relevant tasks.

Time Decomposition

The total time to complete the project was decomposed into two main categories:

Initial Setup Time

- For a human analyst, this refers to the time spent on organizing the spreadsheets and preparing the necessary files for task completion.
- For the VV system, this means the time required for adequately setting up the software environment and data linkage.

Time per Spreadsheet

- This is the time taken to generate a chart from each individual spreadsheet.

Time Metrics

In Table X we compare the efficiency of manual methods against the Visual Viper Python library specifically for the aforementioned project with 72 spreadsheets.

Table X: Time Metrics Comparing Manual Methods and VV Python Library for a Project with 72 Spreadsheets.

Metric		Manual Methods	Visual Viper
Time-to-First-Chart-	**Initial setup**	0h30min	2h00min

Draft	**Time per spreadsheet**	5min	$<10^{-3}$
	Total time (72 spreadsheets)	6h30min	2h00min
Time-to-Final-Chart	**Initial setup**	0h30min	2h00
	Time per spreadsheet	12min	To Miro: ~ 4 sec To GDrive: ~3 sec
	Total time (72 spreadsheets)	14h54min	2h9min

VV: Visual Viper Library; h: hour; min: minute; sec: second.

Adjustment for Fatigue

To enrich our evaluation, we extend the previous comparison by adding considerations for two essential factors. The analysis was performed using R (version 4.2.3) [41] and the plots were generated using the ggplot2 package [13].

We considered the following factors:

- **Task Fatigue**: It's acknowledged that task fatigue can affect the time taken for task completion in a non-linear manner..
- **Additional Human Intervention**: The output visualizations generated by VV requires additional human intervention for validation of accuracy, a factor not considered in the initial metrics.

In this simulation, we concentrate on the "Time-to-Final-Chart" metric, aiming to provide a more comprehensive view of the time required to produce a finalized chart, inclusive of all adjustments and confirmations.

The time adjusted for fatigue was computed using the equation (1):

$$\text{Adjusted Time} = \text{setup\_time} + (\text{task\_time} \times ix) + (\text{task\_time} \times ix^{\text{fatigue\_rate}}) \quad (1)$$

where ix is the index of the task iteration.

We used bootstrapping with 100 samples, assuming a normal distribution for each variable. The 5th and 95th percentiles (P05 and P95) were calculated to construct 90% Confidence Intervals for our time metrics.

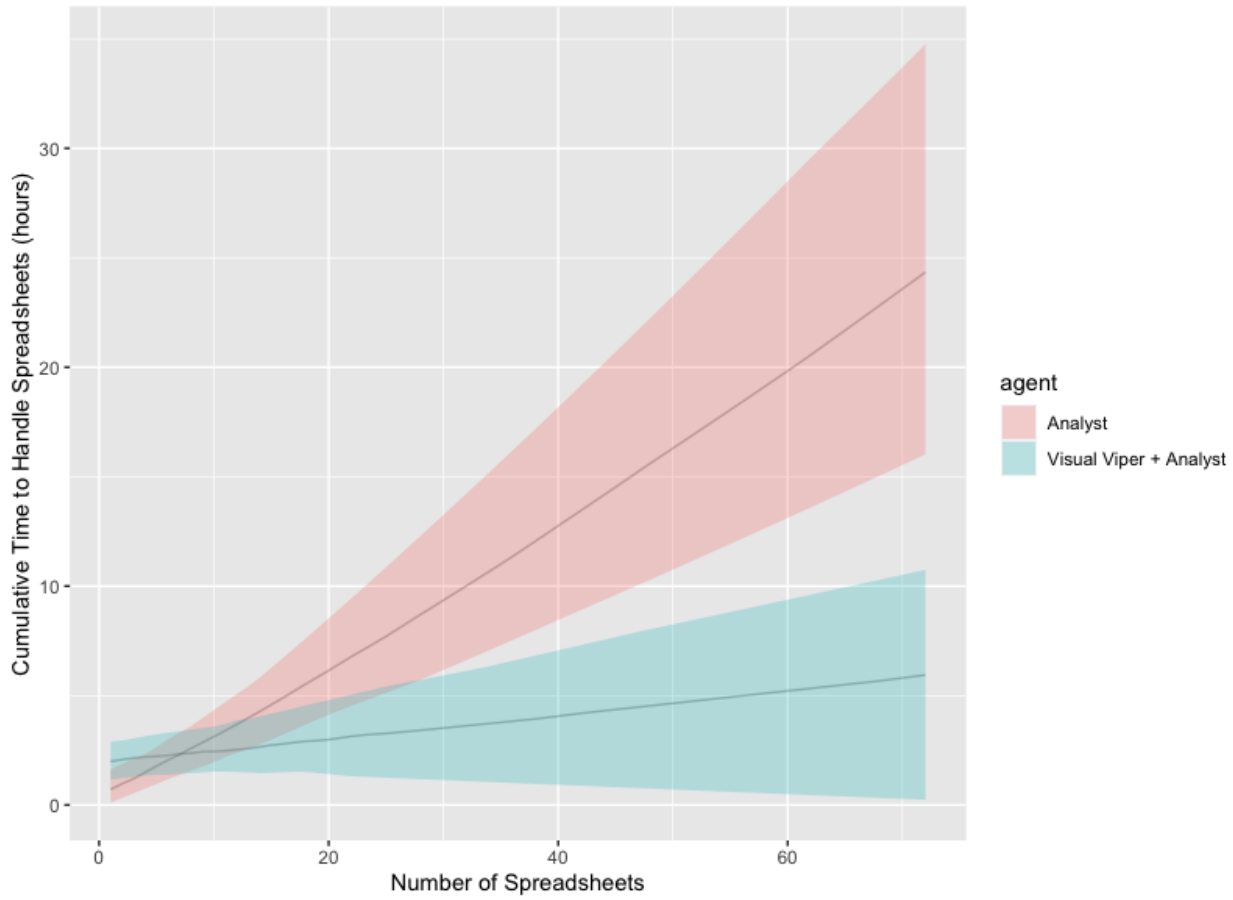


Figure X: Cumulative Time to Handle Spreadsheets for Different Agents

Key Takeaways

The data presented in Table X and Figures X offer significant insights into the operational efficiencies associated with the VV Python library for chart creation in academic research. In particular, the differential impact of using VV in comparison to manual methods becomes more pronounced as the size of the project increases.

Figure X illustrates the cumulative time required to process 72 spreadsheets for both a standalone analyst and an augmented system involving both an analyst and VV. One of the striking observations is the crossover point where VV starts to show a time advantage. While the initial setup time for VV is significantly higher (2 hours compared to 0.5 hours for the analyst), the system starts to outperform the analyst alone at around 8 spreadsheets. By the time 25

spreadsheets are processed, the confidence intervals for the two methods no longer overlap, signaling a clear advantage for VV.

Our adjusted metrics also account for factors like task fatigue and the need for additional human verification of VV's outputs. Even after these considerations, VV holds an advantage in larger projects, both in terms of time efficiency and likely in terms of reduced human error owing to fatigue.

Another significant aspect that adds complexity to this evaluation is the dynamic nature of these data collection processes. Studies are rarely static; they often require adjustments to the design or updating of data. These changes necessitate updating the charts, perhaps multiple times over the course of a study. While the initial setup is a one-time task, adjustments and updates are recurring tasks that continue to consume time. If the initial process is manual and lacks scalability, these frequent updates can quickly become a resource-consuming bottleneck. This is where the growing performance advantages of Visual Viper (VV) become particularly compelling. Our evaluation so far has considered only a single iteration of a project with 72 spreadsheets. In a dynamic study environment requiring frequent adjustments and updates, the scalability advantages of VV could be even more pronounced. Each update in a manual setting can be seen as an iteration that consumes substantial time and resources. VV, which already shows performance benefits in larger projects and single iterations, is likely to magnify these advantages in the context of ongoing, multiple iterations. Therefore, in a continually evolving study, the initial time investment in setting up VV is likely to yield significant long-term savings.

Discussion

The earlier sections provided an in-depth look at the system I've developed, focusing on its architecture, features, and the evaluation metrics that attest to its performance. This discussion aims to offer a comprehensive reflection on this work, examining its current limitations, potential for future development, and the broader implications it could have in academic and healthcare contexts.

Integration in Academic and Healthcare Contexts

The benefits of integrating this system into academic scenarios and other healthcare contexts. The ability to dynamically create and update charts like Forest Plots could be invaluable in both educational settings and medical research. For example, the tool could be integrated into academic courses focusing on statistical methods, epidemiology, or healthcare management, offering students hands-on experience with data visualization. In healthcare settings, the system could aid in real-time data tracking and analytics, which is crucial in making timely and data-backed decisions. The application's modularity and the possibility of developing specific plugins make it highly adaptable to different academic and clinical use-cases.

Deployment Options

As it currently stands, the system operates solely in a local environment. While this setup serves its purpose for small-scale, individual projects, it's limited in terms of scalability and ease of integration into larger workflows. Transitioning to a cloud-based service could effectively address these limitations.

AWS Lambda offers an appealing solution for several reasons. First, it eliminates the need to manage servers or clusters, allowing the focus to remain on code execution. This is particularly beneficial because you only pay for the computation time used, making it a cost-effective choice. Lambda can also automatically respond to code execution requests on any scale, from a few events per day to hundreds of thousands per second, which makes it well-suited for projects with variable demand [50].

Limitations

One limitation of the current system is that the developed plugins are inherently designed to suit the specific workflow requirements of the company where the author works. This could pose challenges in adapting the tool for more generalized use-cases. To enhance the system's utility across various applications, it would be necessary to either develop additional plugins or modify the existing ones to accommodate different configuration parameters.

Another significant limitation remains in terms of deploying charts that handle vector graphics, which would allow researchers to fine-tune the charts intuitively. We initially considered Figma as a potential platform for deployment, but the Figma API is predominantly read-only. It permits only writing comments but restricts manipulating graphical elements directly. This gap opens up a possibility for future work in finding or creating a more versatile platform for chart deployment.

Planned Future Developments

While our focus has been on the Forest Plot plugin due to its prominence in our current large-scale projects, such as one that involves creating 360 Forest Plots, we acknowledge the need for additional chart types. Upcoming releases could include plugins for survival charts, bar charts, and Sankey diagrams.

To make the system more user-friendly, we aim to develop a Command Line Interface (CLI). A CLI would streamline the user experience by providing a straightforward way to configure various system parameters, ideally reducing the initial setup time.

Software Development Learning Insights

Another important outcome of this project is the experience gained in software development methodologies and best practices. While architecting the system, there was an emphasis on employing effective development paradigms and applying established design patterns. Overall, the development process served as a practical case study in applying a blend of software engineering principles, development paradigms, and data structures to create a robust and scalable data visualization tool.

Conclusion

This work has explored the specificities of data visualization in healthcare research, with a particular focus on big datasets and described the development of a data visualization automation tool.

The original contribution of this work lies in the development of a specialized data visualization system designed to meet the specific needs of academic and healthcare settings. While it currently operates in a local environment, it offers a modular architecture that is ripe for future expansion and integration into cloud-based platforms.

The system demonstrated its ability to efficiently create and update complex visualizations, such as Forest Plots, offering substantial advantages in terms of time and resource efficiency.

Importantly, the development process served as an applied case study in employing a range of software development methodologies and best practices, offering significant learning experiences that can inform future work in this domain.

Several limitations were identified, setting the stage for future development that could focus on expanding the types of visualizations supported, increasing scalability, and offering more versatile deployment options.

Ultimately, the insights gained through this work affirm the power of data visualization as a critical tool for data interpretation and decision-making in healthcare research. As this field continues to evolve, it is anticipated that the integration of specialized tools, coupled with advancements in software engineering practices, will further amplify the capabilities of data visualization to serve the complex needs of healthcare research and beyond.

As a final note, it is worth mentioning that the tool developed through this work will be actively leveraged in our scientific communication processes, particularly in the context of real-world evidence. This incorporation not only adds a practical dimension to the academic contributions of this research but also paves the way for a sustained impact on healthcare research and outcomes.

References

1. Coughlin S, Roberts D, O'Neill K, Brooks P. Looking to tomorrow's healthcare today: a participatory health perspective. Intern Med J. 2018;48: 92–96.

2. Dash S, Shakyawar SK, Sharma M, Kaushik S. Big data in healthcare: management, analysis and future prospects. Journal of Big Data. 2019;6: 1–25.

3. El Khatib M, Hamidi S, Al Ameeri I, Al Zaabi H, Al Marqab R. Digital Disruption and Big Data in Healthcare - Opportunities and Challenges. Clinicoecon Outcomes Res. 2022;14: 563–574.

4. Le T, Reeder B, Thompson H, Demiris G. Health Providers' Perceptions of Novel Approaches to Visualizing Integrated Health Information. Methods Inf Med. 2013;52: 250–258.

5. Filonik D, Rittenbruch M, Foth M, Bednarz T. Visualisation Design as Language Transformations - From Conceptual Models to Graphics Grammars. 2019 23rd International Conference in Information Visualization – Part II. ieeexplore.ieee.org; 2019. pp. 18–23.

6. Tufte ER. The Visual Display of Quantitative Information. 1983.

7. Cleveland WS, McGill R. Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods. J Am Stat Assoc. 1984;79: 531–554.

8. Savva M, Kong N, Chhajta A, Fei-Fei L, Agrawala M, Heer J. ReVision: automated classification, analysis and redesign of chart images. Proceedings of the 24th annual ACM symposium on User interface software and technology. New York, NY, USA: Association for Computing Machinery; 2011. pp. 393–402.

9. Wilkinson L. The Grammar of Graphics. Springer New York;

10. Bostock M, Heer J. Protovis: a graphical toolkit for visualization. IEEE Trans Vis Comput Graph. 2009;15: 1121–1128.

11. Bostock M, Ogievetsky V, Heer J. D^3: Data-Driven Documents. IEEE Trans Vis Comput Graph. 2011;17: 2301–2309.

12. Online T. A Visualization Grammar. In: Vega [Internet]. [cited 31 Aug 2023]. Available: https://vega.github.io/vega/

13. Wickham H. Programming with ggplot2. In: Wickham H, editor. ggplot2: Elegant Graphics for Data Analysis. Cham: Springer International Publishing; 2016. pp. 241–253.

14. Satyanarayan A, Moritz D, Wongsuphasawat K, Heer J. Vega-Lite: A Grammar of Interactive Graphics. IEEE Trans Vis Comput Graph. 2017;23: 341–350.

15. Gatto NM, Wang SV, Murk W, Mattox P, Brookhart MA, Bate A, et al. Visualizations throughout pharmacoepidemiology study planning, implementation, and reporting. Pharmacoepidemiol Drug Saf. 2022;31: 1140–1152.

16. Gauthier R, Ponto S. Designing Systems Programs. Old Tappan, NJ: Prentice Hall; 1970.

17. Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM. 1972;15: 1053–1058.

18. Van Vliet H. Software engineering: principles and practice. John Wiley & Sons Hoboken, NJ; 2008.

19. Sun H, Ha W, Teh P-L, Huang J. A Case Study on Implementing Modularity in Software Development. Journal of Computer Information Systems. 2017;57: 130–138.

20. Brooks (Jr.) F. The Mythical Man-month: Essays on Software Engineering. Addison-Wesley Publishing Company; 1975.

21. Singh N, Chouhan SS, Verma K. Object Oriented Programming: Concepts, Limitations and Application Trends. 2021 5th International Conference on Information Systems and Computer Networks (ISCON). 2021. pp. 1–4.

22. Black AP. Object-oriented programming: Some history, and challenges for the next fifty years. Inform and Comput. 2013;231: 3–20.

23. Singh N, Chouhan S, Verma K. Object oriented programming: Concepts, limitations and application trends. TechRxiv. 2021. doi:10.36227/techrxiv.16677259.v1

24. Aniche M, Yoder J, Kon F. Current Challenges in Practical Object-Oriented Software Design. 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). 2019. pp. 113–116.

25. Dessi M. Spring 2.5 Aspect Oriented Programming. Packt Pub.; 2009.

26. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH; 1995.

27. Jacobson L, Booch JRG. The unified modeling language reference manual. Addison-Wesley Professional; 2005. Available: http://debracollege.dspaces.org/bitstream/123456789/404/1/UML%20Reference%20Manual%20by%20James%20Rambaugh.pdf

28. Martin RC. Design Principles and Design Patterns. [cited 29 Aug 2023]. Available: http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf

29. Agha D, Sohail R, Meghji AF, Qaboolio R, Bhatti S. Test Driven Development and Its Impact on Program Design and Software Quality: A Systematic Literature Review. 2023;11: 268–280.

30. Baldassarre MT, Caivano D, Fucci D, Juristo N, Romano S, Scanniello G, et al. Studying test-driven development and its retainment over a six-month time span. J Syst Softw. 2021;176: 110937.

31. Taufiqurrahman F, Widowati S, Alibasa MJ. The Impacts of Test Driven Development on Code Coverage. 2022 1st International Conference on Software Engineering and Information Technology (ICoSEIT). 2022. pp. 46–50.

32. Bogdanchikov A, Zhaparov M, Suliyev R. Python to learn programming. J Phys Conf Ser.

2013;423: 012027.

33. O'Grady S. The RedMonk programming language rankings: January 2023. In: tecosystems [Internet]. 16 May 2023 [cited 28 Aug 2023]. Available: https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/

34. Stack overflow developer survey 2023. In: Stack Overflow [Internet]. [cited 28 Aug 2023]. Available: https://survey.stackoverflow.co/2023/

35. "Home." In: Docker Documentation [Internet]. 22 Aug 2023 [cited 31 Aug 2023]. Available: https://docs.docker.com/

36. What is a container? In: Docker [Internet]. [cited 31 Aug 2023]. Available: https://www.docker.com/resources/what-container/

37. Dalpiaz F, Brinkkemper S. Agile Requirements Engineering with User Stories. 2018 IEEE 26th International Requirements Engineering Conference (RE). ieeexplore.ieee.org; 2018. pp. 506–507.

38. Lucassen G, Dalpiaz F, Werf JMEM van der, Brinkkemper S. The Use and Effectiveness of User Stories in Practice. Requirements Engineering: Foundation for Software Quality. Springer International Publishing; 2016. pp. 205–222.

39. Buzurovic I, Podder TK, Fu L, Yu Y. Modular Software Design for Brachytherapy Image-Guided Robotic Systems. 2010 IEEE International Conference on BioInformatics and BioEngineering. 2010. pp. 203–208.

40. Pytest: Helps you write better programs — pytest documentation. [cited 29 Aug 2023]. Available: https://docs.pytest.org/en/7.4.x/

41. Foundation for Statistical Computing RR. R: A language and environment for statistical computing. RA Lang Environ Stat Comput.

42. Preston-Werner T. Semantic Versioning 2.0.0. In: Semantic Versioning [Internet]. [cited 28 Aug 2023]. Available: https://semver.org/spec/v2.0.0.html

43. Wongsuphasawat K. Navigating the wide world of data visualization libraries. In: Nightingale [Internet]. 22 Sep 2020 [cited 28 Aug 2023]. Available: https://medium.com/nightingale/navigating-the-wide-world-of-web-based-data-visualization-libraries-798ea9f536e7

44. Heer J. Introduction to Vega-Lite. In: Observable [Internet]. 28 Mar 2019 [cited 28 Aug 2023]. Available: https://observablehq.com/@uwdata/introduction-to-vega-lite

45. Gavina C. Lipid Management in pre-diabetes and diabetes - a RWE study of an unselected portuguese population. Congresso Português de Endocrinologia 2023; 2023 Feb 4.

46. Low-density lipoprotein cholesterol reduction and short-term incidence of ASCVD in the population-based cohort study LATINO. [cited 28 Aug 2023]. Available: https://esc365.escardio.org/presentation/267983?resource=abstract

47. Gavina C, Araújo F, Teixeira C, Ruivo JA, Corte-Real AL, Luz-Duarte L, et al. Sex differences in LDL-C control in a primary care population: The PORTRAIT-DYS study. Atherosclerosis. 2023. doi:10.1016/j.atherosclerosis.2023.05.017

48. PlantUML Language Reference Guide. In: PlantUML.com [Internet]. [cited 31 Aug 2023]. Available: https://plantuml.com/guide

49. Saving Altair charts — Vega-Altair 5.1.1 documentation. [cited 31 Aug 2023]. Available: https://altair-viz.github.io/user_guide/saving_charts.html

50. AWS lambda. [cited 31 Aug 2023]. Available: https://aws.amazon.com/pt/lambda/