# 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]](https://paperpile.com/c/wYkhtl/MYgQ).

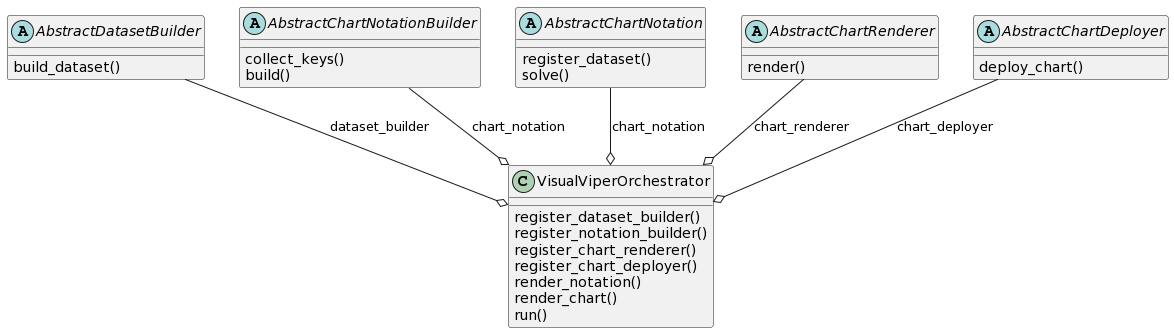


Figure X: High-level Class Diagram of System Architecture

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]](https://paperpile.com/c/wYkhtl/lINo).
* **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.

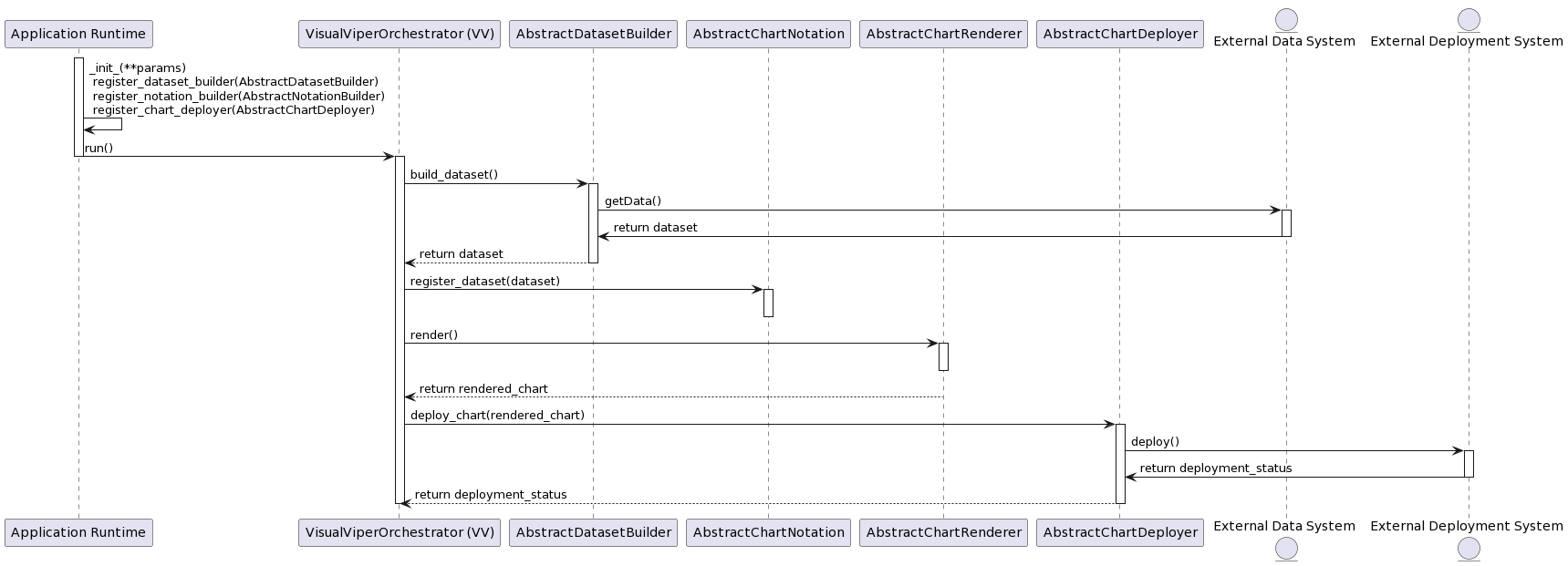


Figure X: Sequence Diagram for Chart Creation and Deployment in Visual Viper Framework

#### 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.

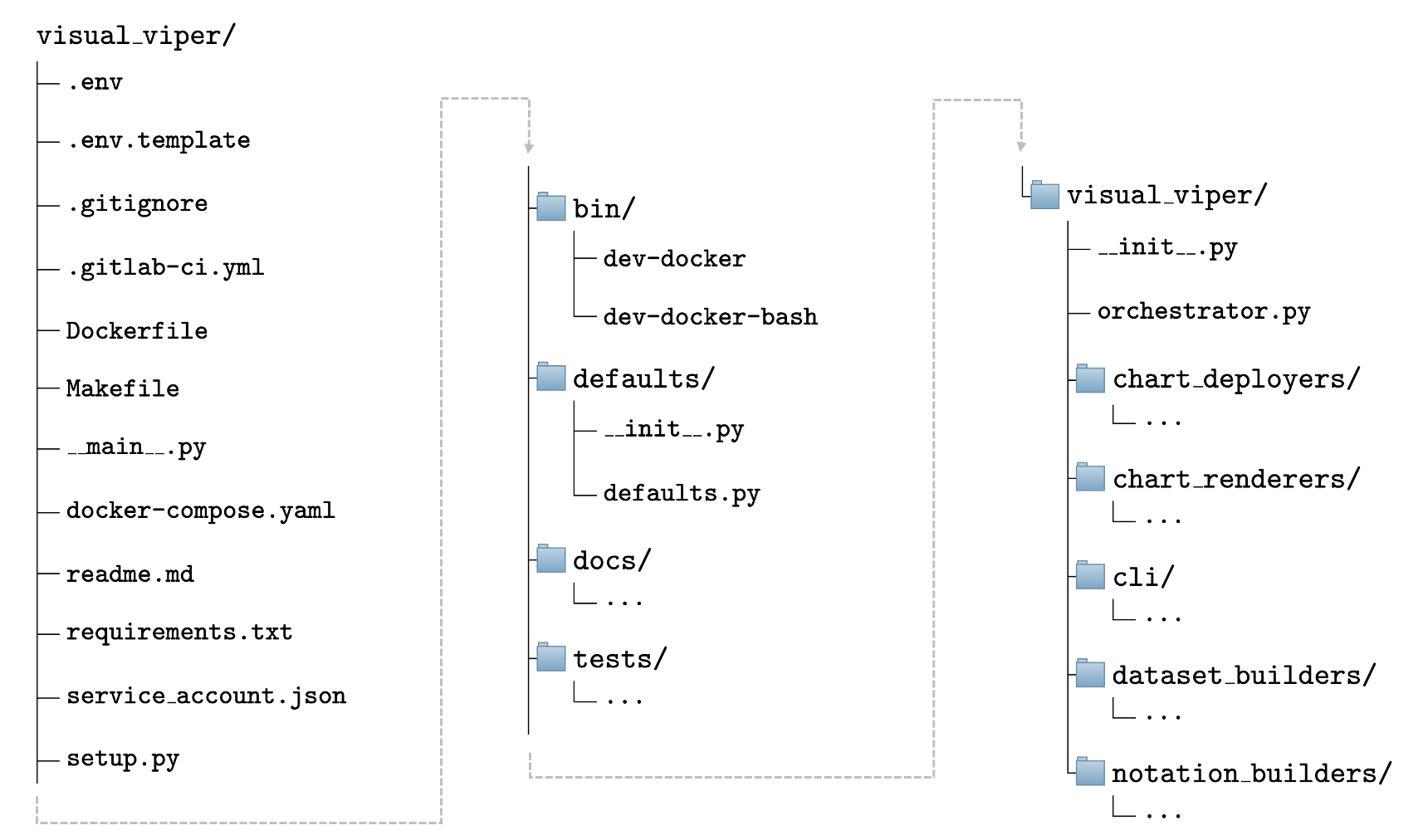


Figure X: Directory structure of the project. The directory structure and the following graphical diagram were generated using VV's directory description and LaTeX diagramming plugins (not described in the current work). For brevity, certain folders have been excluded or their contents omitted from this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

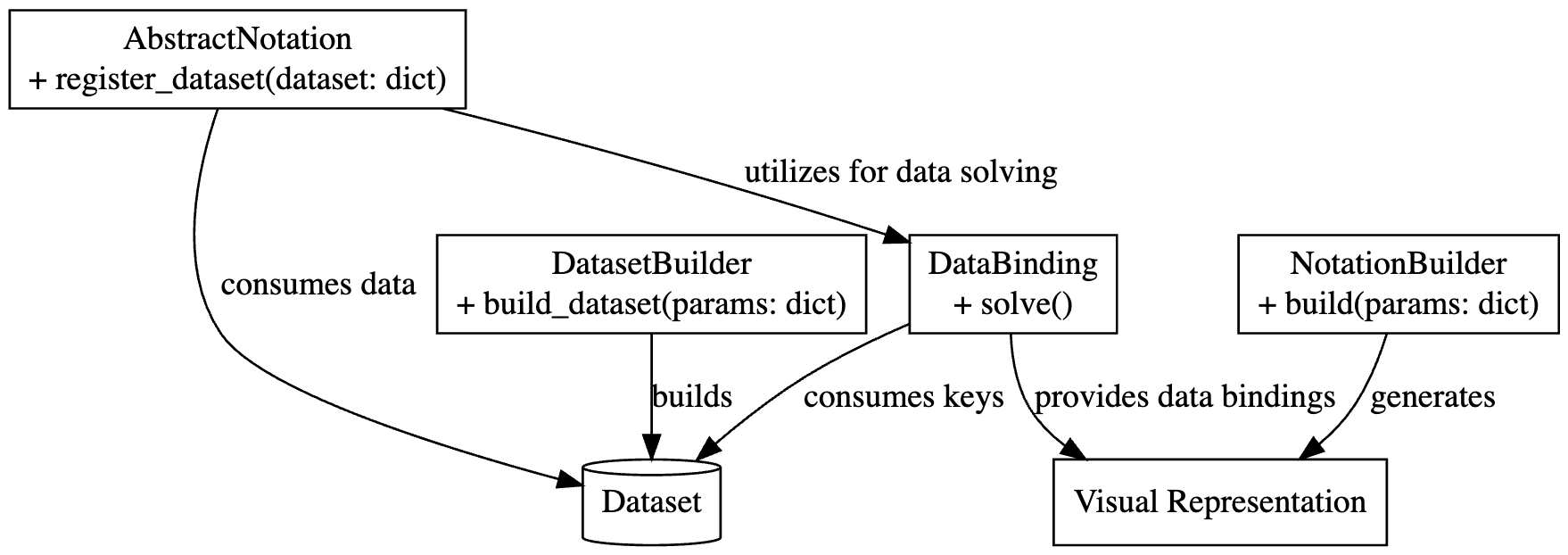


Figure X: Data Flow Diagram of Key System Components of Visual Viper.

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.