SWE621 – HW3

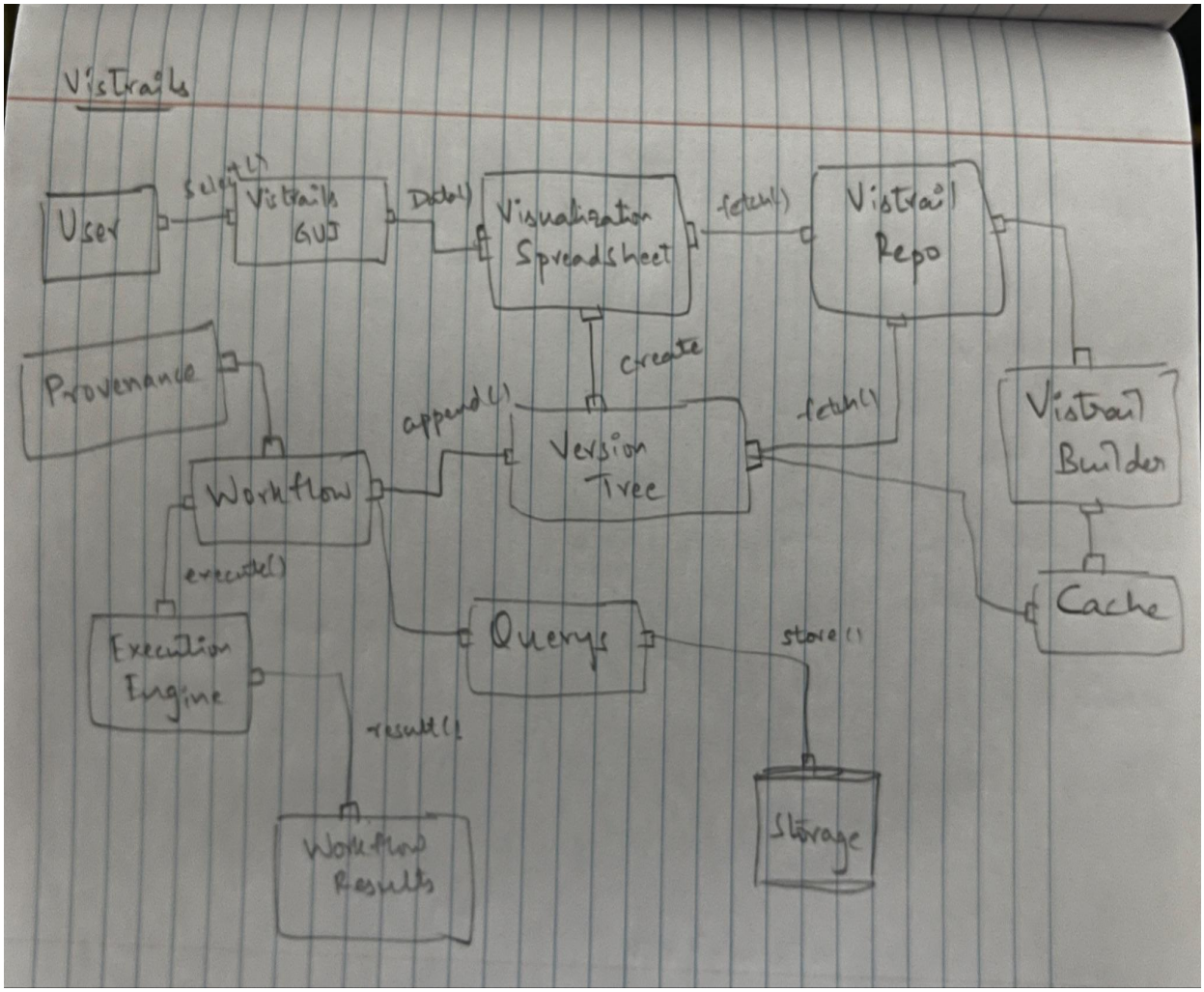
Group:
Venkata Krishna Chaitanya Chirravuri **(G01336659)**
Sai Prashanth Reddy Kethiri **(G01322333)**

Reference System – VisTrails
Additional Systems – Visualization Tool Kit (VTK), matplotlib

Part -1

VisTrails

Component connector Diagram



The components here are **User, Vistrails GUI, Visualization Spreadsheet, Vistrails Repo, Vistrails Builder, Cache, Version Tree, Queries, Workflow, Execution Engine, Provenance, WorkflowResults, Storage .** The situation we have considered is plotting a graph from the spreadsheet and creating through the workflow execution. The User interacts with the Vistrails GUI and create data Visualization Spreadsheet, then It can also be fetched from the Vistrails Repo. and from the spreadsheet the version tree with the pipeline execution is created and the workflow can be saved as part of the provenance infrastructure. Then the workflow can be retrieved with queries and stored in the storage. Then the workflow is executed in the execution engine and the workflow results are generated. The plotting and selection of type of plots are done in the workflow itself.

Architectural styles followed by VisTrails are flexible provenance architecture along with layered architecture followed by pipeline execution. VisTrails transparently tracks changes made to workflows, including all the steps followed in the exploration. The

system can optionally track run-time information about the execution of workflows (*e.g.,* who executed a module, on which machine, elapsed time *etc.*). One of the key components of any system supporting provenance is the serialization and storage of data. For this they have added a db layer. The db layer is composed of three core components: the domain objects, the service logic, and the persistence methods. The domain and persistence components are versioned so that each schema version has its own set of classes. This way, we maintain code to read each version of the schema. There are also classes that define translations for objects from one schema version to those of another. The service classes provide methods to interface with data and deal with detection and translation of schema versions. Workflow systems support the creation of pipelines (workflows) that combine multiple tools. As such, they enable the automation of repetitive tasks and result reproducibility.

Goals achieved by these architecture styles are, Provenance is recognized as a central challenge to establish reliability and provide security. In scientific workflows, provenance is considered essential to support experiments' reproducibility, interpretation of results, and problem diagnosis. We use templates and a meta-schema to define both the object layout (and any in-memory indices) and the serialization code. The meta-schema is written in XML, and is extensible in that serializations other than the default XML and relational mappings VisTrails defines can be added. Automatic translation is key for users that need to migrate their data to newer versions of the system. VisTrails maintains a copy of code for each version, the translation code just needs to map one version to another. VisTrails caches intermediate results to increase efficiency in exploration. It does so by reusing pieces of pipelines in later executions. We can also perform sequential execution of multiple pipelines through VisTrail Builder and VisTrail Spreadsheet.
Importance of provenance architecture in this system is that by analyzing a provenance repository, we can determine common workflow structures, identify common sequences of workflow changes, gain insight into how different people solve problems, and help users debug common errors and bottlenecks. Such analysis can also enable new techniques like auto-completion for workflows. Introducing db layers to VisTrails architecture has automated the process of code serialization, where writing code in XML used to be tedious and repetitive. Workflows (pipelines) have a number of advantages compared to scripts and programs written in high-level languages. They provide a simple programming model whereby a sequence of tasks is composed by connecting the outputs of one task to the inputs of another.

Constraints:
Technical constraints: they are relevant to provenance collection and management. It involves interoperability at the syntactic level between provenance records generated by

heterogeneous systems, security issues including provenance integrity, access control and privacy. Performance, reliability and scalability constraints are also important because we are in a cloud context mandating from a provenance service to be scalable and available. Add to that trust and reliability about track elements provided by external systems/services during their creation or their transfer.

Data Constraints: With file-based references, accessing relational databases in workflows may also lead to problems for reproducibility. For example, when workflow consumes data from a database, its results may depend on the data. When the database is updated, the results of the workflow may no longer be reproduced, as database updates may have changed the data that is used by the workflow. Reproducibility, in this case, is more challenging because there is an inherent mismatch between workflow and database models: while workflows are stateless and deterministic, databases are stateful - new states reflect the changes applied to the database.

Cache Constraints: Vis Trails caches by default, so after a workflow is executed, if none of its parameters change, it won't be executed again. If a workflow reads a file using the basic module File, VisTrails does check whether the file was modified since the last run. It does so by keeping a signature that is based on the modification time of the file. And if the file was modified, the File module and all downstream modules (the ones which depend on File) will be executed.

Goals achieved by these constraints:

The provenance information is stored in a structured way. You have a choice of using a relational database (such as MySQL or IBM DB2) or XML files in the file system. The system provides flexible and intuitive query interfaces through which you can explore and reuse provenance information. You can formulate simple keyword-based and selection queries (e.g., find a visualization created by a given user) as well as structured queries (e.g, find visualizations that apply simplification before an isosurface computation for irregular grid data sets). So, provenance architecture with constraints provides security of data, efficiency in processing and producing results upon querying.

VisTrails assumes fundamentally that a pipeline is a dataflow. This means that pipeline cycles are disallowed, and that modules are supposed to be free of side-effects. This is obviously not possible in general, particularly for modules whose sole purpose is to interact with operating system resources. In these cases, designing a module is harder – the side effects should ideally not be exposed to the module interface. VisTrails provides some support for making this easier. VisTrails caches intermediate results to increase efficiency in exploration. It does so by reusing pieces of pipelines in later executions.So, caching constraints achieves reusability, reproducibility, efficiency.

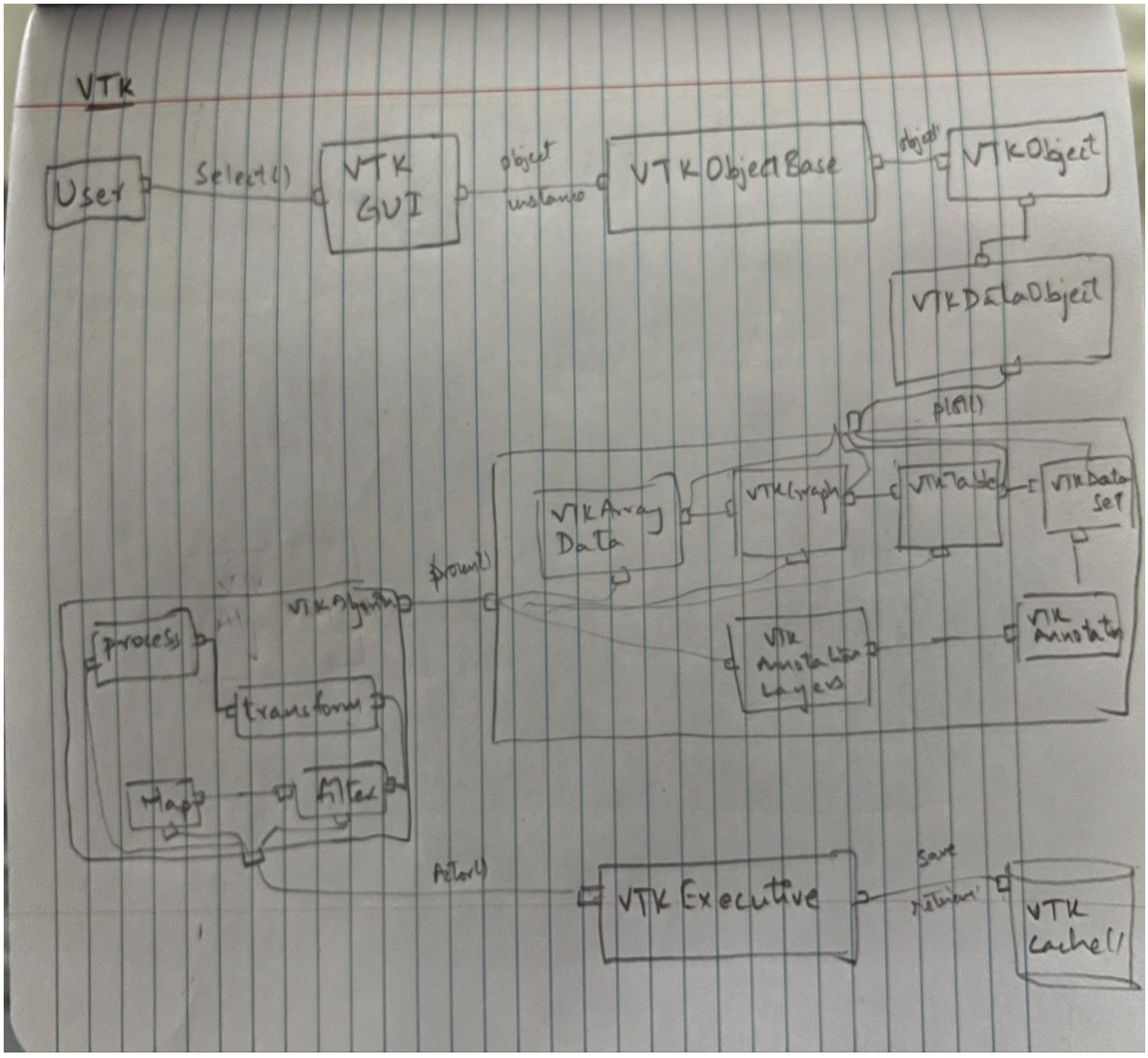
By maintaining detailed provenance of the exploration process - both within and across different versions of a dataflow - it allows scientists to easily navigate through the space

of dataflows created for a given exploration task. In particular, this gives them the ability to return to previous versions of a dataflow and compare their results.

Part - 2

VTK

Component Connector Diagram



The components here are User, VTK **GUI, VTKObjectBase, VTKObject, VTKDataObject, VTKAlgorithm, VTKExecutive and VTKCache.** The closest situation

here we considered is plotting of a graph with given scientific data. The User uses the VTK GUI to access the VTKObjectBase where the instance of the VTK Object is instantiated and then respective method for plotting is invoked where the required methods and procedural callings for plotting the graphs will be invoked. All the components in the diagram are strictly used for plotting the 2D graph computing the scientific data. VTKDataSet is specially used for the computation of scientific data. And in case of plotting the polygraphs , we use other components like VTKPoly and required dependencies and continue with the process, transform, map and filter of the dataobject within the VTKAlgorithm and this acts as the Actor() to execute the entire process in a pipeline and the VTKExecutive does the final part of plotting and this can be cached using the VTKCache.

Architectural style followed by VTK is Pipeline architecture. In concept, the pipeline architecture consists of three basic classes of objects: objects to represent data (vtkDataObject), objects to process, transform, filter or map data objects from one form into another (vtkAlgorithm); and objects to execute a pipeline (vtkExecutive) which controls a connected graph of interleaved data and process objects (i.e., the pipeline).

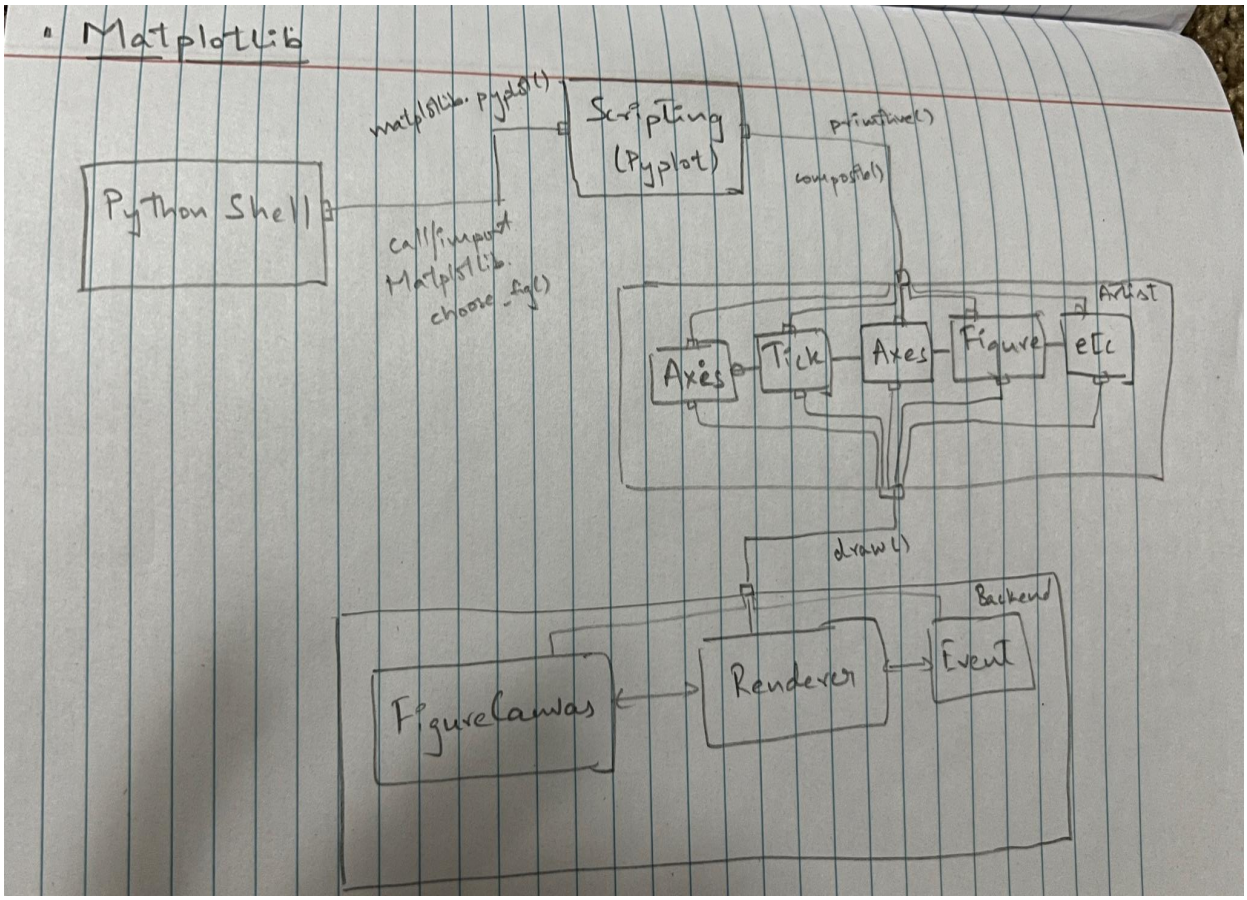
Goals achieved by this architecture style are, visualization data that would not normally fit into memory can be processed. The second is that visualizations can be run with a smaller memory footprint resulting in higher cache hits, and little or no swapping to disk. In order to control the streaming of the data through a pipeline, we must be able to determine what portion of the input data is required to generate a given portion of the output. This allows us to control the size of the data through the pipeline, and configure the algorithms. Pipeline architectures provide a versatile and efficient mechanism for constructing visualizations, and they have been implemented in numerous libraries and applications.
Importance of pipeline architecture in this system is it sometimes saves time with reusability, improves functionality by establishing seamless workflow of data and accelerates data lifecycle processes by automation.

Constraints imposed by this architectural style are, each processor defines zero or more input ports and output ports with certain restrictions on the kind of data and operations supported. The ports are read-only and do not hold references to data objects or information associated with the actual execution of the processor. All data and information for execution are passed to the processor by its executive when a request is made. Interactions happen only through pipes and processes do not make any assumptions about what happens upstream or downstream.

Goals achieved by these constraints: Main goals achieved by pipeline architecture is efficiency, where you can work with multiple workflows at the same time and compare those workflows. As mentioned above, we can also fit visualization data into a smaller memory making the system more scalable to huge datasets. One of the core requirements of VTK is its ability to create data flow pipelines that are capable of ingesting, processing, representing and ultimately rendering data. Hence the toolkit is necessarily architected as a flexible system and its design reflects this on many levels. For example, we purposely designed VTK as a toolkit with many interchangeable components that can be combined to process a wide variety of data. Hence, with the help of pipelines we are able to achieve configurability, extensibility and flexibility. We can also make changes to existing workflows with help of pipeline architecture opening doors for modifiability. With VTK using pipeline architecture, the system achieved design modularity.

Matplotlib

Component and Connector Diagram:



The components here are the **Python Shell, Scripting, Artist and Backend** where Python Shell is the IDE where a user has a preliminary coding knowledge and imports the matplotlib.pyplot package as the scripting layer. The situation we considered here is plotting a 2D graph, where after importing the package the artist object is instantiated i.e, the artist type of primitive has been instantiated with the axes, axis, and figure , then the actual data plotting happens in the backend layer where the renderer draws the plot at the figure canvas with the corresponding user input event.

Architectural style followed by Matplotlib is layered architecture. The idea behind Layered Architecture is that modules or components with similar functionalities are organized into horizontal layers.As a result, each layer performs a specific role within the application. Matplotlib has three layers: Scripting Layer, Artist Layer and Backend Layer .

Scripting Layer: It is the top layer, designed to make Matplotlib work like other programming scripts. It is a collection of command style functions and is therefore considered the easiest layer to use. Scripting Layer is often known as procedural plotting. This is the top layer in the architecture and is essentially the matplotlib.pyplot interface, which generally automates the process of defining a canvas and defining a figure artist instance and connecting them.

Artist Layer: Artist Layers lie after the Scripting layer , allowing control and fine-tune as many elements as possible in the figure just like an artist paints on the canvas. This layer has one main object, Artist that uses the Renderer to draw on the canvas. This layer particularly enables many customization and is more convenient for advanced plots. All subplots are assigned to an Artist object. Artist Layer is also referred to as object-based plotting, everything visible in the top layer is an instance of the Artist Layer. Artist Layer comprises mainly of two types, one being the primitive type artist such as Line2D, Rectangle,Circle and Text and the other is the composite type such as Axis,Tick ,Axes and Figure.

Backend Layer: Backend Layer is the last and complex layer dealing with communication with other toolkits like wxPython and PostScript. It consists of three abstract interface classes, namely FigureCanvas, Renderer and Event. Event handles the user inputs such as keyboard and mouse, Renderer the base abstract class handles all the drawing responsibilities into the figure canvas and the FigureCanvas is the canvas the figure renders into.

Goals achieved by this architecture style are, by making use of this layered architecture it brings flexibility to the developers to share the script with other developers in the artist layer and for non-programmatic users scripting layer automates the process of defining the FigureCanvas and artist instance, which makes it easy to

use for quick exploratory analysis. Each layer has its own functionality, Backend layer: deals with actual drawing, Artists layer: several of these sit “on top of” the backend, and describe how data is arranged. Scripting layer: creates the artists and choreographs them together into our visual. One of the core architectural tasks matplotlib solves is implementing a framework for representing and manipulating the Figure that is segregated from the act of rendering the Figure to a user interface window or hardcopy.
Importance of layered architecture in this system is it simplifies and speeds up our interaction with the environment in order to build plots quickly. With the help of layered architecture it is easy to keep track of all the objects making the ability to browse the objects easier. It ensures layer independence by offering services from the lowest to the highest layer without specifying how the services are implemented. As a result, any changes made to one layer have no effect on the other levels. Each lower layer contributes its services to the top layer, resulting in a complete collection of services for managing communications and running applications. And most importantly, it is easy to understand and maintain the entire matplotlib system.

Constraints imposed by this architectural style are lacking in built scalability,as the system cannot be scaled as the layered architecture results in a rigid structure of software implementation and highly coupled software groups. A change to a single layer must be verified that it does not crash the entire system. There could be a negative impact on the performance as we have the extra overhead of passing through the layers instead of calling a component directly. The layered architecture sometimes adds complexity to the system as well. There will be interdependence between the layers as one layer depends on the layer above to receive data. And finally parallel processing would be not possible for layered architectural style systems.

Goals achieved by these constraints: Few of the main goals achieved by using layered architecture are building increasingly sophisticated features and logic into the top layers while keeping output devices and backends layers relatively simple. Another goal achieved implementing the layered architecture is it freed developers to refactor the internal object-oriented API several times with minimal impact to most users because the top layer- scripting layer interface was unchanged. And finally the organization of units with specific roles and responsibilities within each layer, this enables each layer to manage its own software dependencies.

Part - 3

Compare & Contrast

Similar / Different

VTK follows pipeline architecture (pipe & filter) where elements are pipes, filters, read ports and write ports. Coming to matplotlib it makes use of layered architecture, where elements involved in the architecture are layers. In the case of VisTrails, it is a combination of both pipeline execution and layered architecture along with provenance infrastructure. Here, workflows are managed by pipeline architecture, data is being stored in layers which follows layered architecture. Up on these two architectural styles VisTrails maintains a provenance infrastructure that keeps track of changes to the data and the information about the owner of the data.

Pros & Cons

VisTrails

Pros

- Data provenance enables professionals to check the credibility of each piece of given information.
- Provides a persistent storage mechanism that manages input, intermediate, and output data files, strengthening the links between provenance and data.
- Provenance is considered essential to support experiments' reproducibility, interpretation of results, and problem diagnosis.
- Integrity & Security of the data.
- Pipelines enable interactive multiple-view visualizations by simplifying the creation and maintenance of visualization pipelines, and by optimizing their execution.
- Pipelines improve efficiency by making use of cache.
- Goals of reproducibility can be met by making use of the db layer.

Cons

- The provenance data from huge data workflows is too large.
- Different clients / users may have different needs (e.g. provenance described at different levels of detail or no provenance at all). Hence, ‘one size has to fit all’ does not work.
- Time consuming to wrap the experiment into a workflow system if scientists are already using another approach for the execution.

VTK

Pros

- Provides cacheable results
- More scalable with huge datasets.
- Offers more flexibility to process, represent and render the data.
- Users can add their own packages resulting in more configurability and extensibility.

Cons

- Inefficient data representation for certain data types.
- Cannot take advantage of optimized representations.

Matplotlib

Pros

- There is reduced dependency because the function of each layer is separate from the other layers.
- Ability to represent complex forms of data.
- Provides efficient work time by working with multiple workflows at the same time.

Cons

- **Performance Degrades -** having too many layers (data & changes to data passes slowly to higher layers)
- Difficult to assign functionality to the right layer.

Opinion (Which architecture is better?)

Each system has their own set of advantages and disadvantages and each architecture is well designed with their respective systems. But, in my opinion, I feel the architectural style of VisTrails is more efficient and flexible, since we can work with multiple workflows at the same time and the results are cacheable. Provenance infrastructure of this system gives more integrity to the data.

Difference in goals and requirements

The requirements of VTK and VisTrails are the same as they need to work with workflows to process the data and perform 3d visualizations. But they have chosen different architectural styles and meet separate sets of goals like, VisTrails needs to work with multiple workflows at the same time, store and retrieve the results from cache

memory for faster computation. Whereas VTK is much more scalable with huge datasets and also renders 3d images.

Coming to matplotlib it performs 2d data visualization and few cases of 3d data as well. Since it follows layered architecture it is not very scalable. But it achieves its goals like dealing efficiently with 2d data and also provides extensibility.

References

- https://streamsets.com/blog/data-pipeline-architecture-deep-dive/
- https://youtu.be/0m611JTA6l0
- https://kitware.github.io/vtk-examples/site/VTKBook/04Chapter4/
- https://www.cs.uic.edu/~jbell/CS526/Tutorial/Tutorial.html
- https://materials.prace-ri.eu/366/2/vtkPipeline.pdf
- https://itk.org/Wiki/images/8/80/Pipeline.pdf
- https://www.vistrails.org/usersguide/dev/html/intro.html
- https://www.aosabook.org/en/vistrails.html
- https://www.andrew.cmu.edu/user/jiaz/Papers/JiaZhang-VisTrails-HECC.pdf
- http://www.sci.utah.edu/~cscheid/pubs/vt-sciflow.pdf
- https://ieeexplore.ieee.org/document/1532788
- https://www.researchgate.net/figure/VisTrails-Architecture-these-visualization-products-are-generated-in-an-ad-hoc-manner_fig1_221212794
- https://www.researchgate.net/publication/50520348_VisTrails_Enabling_Interactive_Multiple-View_Visualizations
- https://www.vistrails.org/index.php/ProvenanceAnalytics