VISUALIZATION OF DIFFERENCES IN DIFFERENT VERSIONS OF A SYSTEM

1. Introduction:

It is often beneficial to know the distinctions in the clusters, dependencies, most important components, most dependent components, etc. of different versions of a system for building a new version of the system. We build more up to date versions of a system to make it easier to use, increase effectiveness and functionalities, and make it less complicated. In such scenarios, it is valuable to get the information of the progressions made in each cluster of each version, the bifurcation of dependencies based on the clusters for each version, the components on which maximum number of components rely upon, and the components which are dependent on maximum components. Our project is about visualization of each of these parts, i.e. our project is divided into 4 parts which we will examine in detail further. We have used RELAX architecture recovery method. Our tool is built to simplify the work for a developer and architect working on building a new version of any system.

2. Problem Statement:

While building a new version of a system, the following decision-making challenges are faced:

1) How to decide what all changes to make in this latest version for improvement over the previous ones?
2) In the latest version of the system, what all interdependencies for each cluster are present?
3) Which components if removed from the previous version will cause major architectural change?
4) Which components if expelled from the previous version will make the architecture loosely coupled?

3. Solution and Scope:

PART-1

To decide the changes to make in the latest version, we can get an overview by looking at the changes made in previous versions and compare whether these modifications reflect success or failure of the altered version. In simple words, if a version X is proved to be a success/failure compared to version Y, then what all changes/differences were found in the clusters [Graphics, IO, Networking, SQL, No Match] of the former version compared to the latter one. We can keep a track of these changes and its corresponding effects for multiple versions and use machine learning algorithms to find out accurately the impact of each change. For this purpose, we have made a Comparison Tool which checks the added, removed, and identical components of two versions at a time. It is visualized using D3.js in the form of Cluster Representation.

This visualization makes it easier to see the distinctions in a given cluster of two different versions of the system.

PART-2

Here, we check the dependencies of files and their corresponding clusters. If there is a file A that depends on file B, file C, file D, then we check the cluster in which file A belongs and similarly the clusters in which file B, file C, file D belong. Similarly, we do this for multiple versions and compare. This helps us getting the overview of the dependencies for multiple versions. Through this comparison, the first level matrix solves the issue of scalability. It is elegantly displayed using Matrix Representation.

PART-3

We frequently need to find out the list of components that have maximum number of components dependent on it in the previous versions. For example, if there is a component A that has maximum number of components dependent on it in version X, and the same component A may or may not have maximum number of components dependent on it in version Y. So, with this change in number of dependencies on that component A we can figure out the impact on the system by the above changes. Hence, we can find out the importance of change in components' dependencies. So, we find the top most crucial components that have maximum components dependent on them and make sure to minimize changes in them if minimum architectural change is required. It is represented using bar charts.

PART-4

A typical problem in altering an existing system is the risk of causing cyclic dependencies and coupling it tightly. In order to dodge this problem, it is better to have the data from previous versions of the components that depend on maximum number of components. These components could potentially cause tight coupling. Consolidating this with part-3 problem, it brings cyclic dependencies, resulting in deadlocks and inefficient usage of resources. Hence, as a solution, we keep track of top most components which are dependent on maximum number of components for version X and the same top most components' dependency number for version Y. This way we can decide which change to incorporate for number of dependencies of top most components based on its impact by making modifications. This correlation is again represented using bar charts.

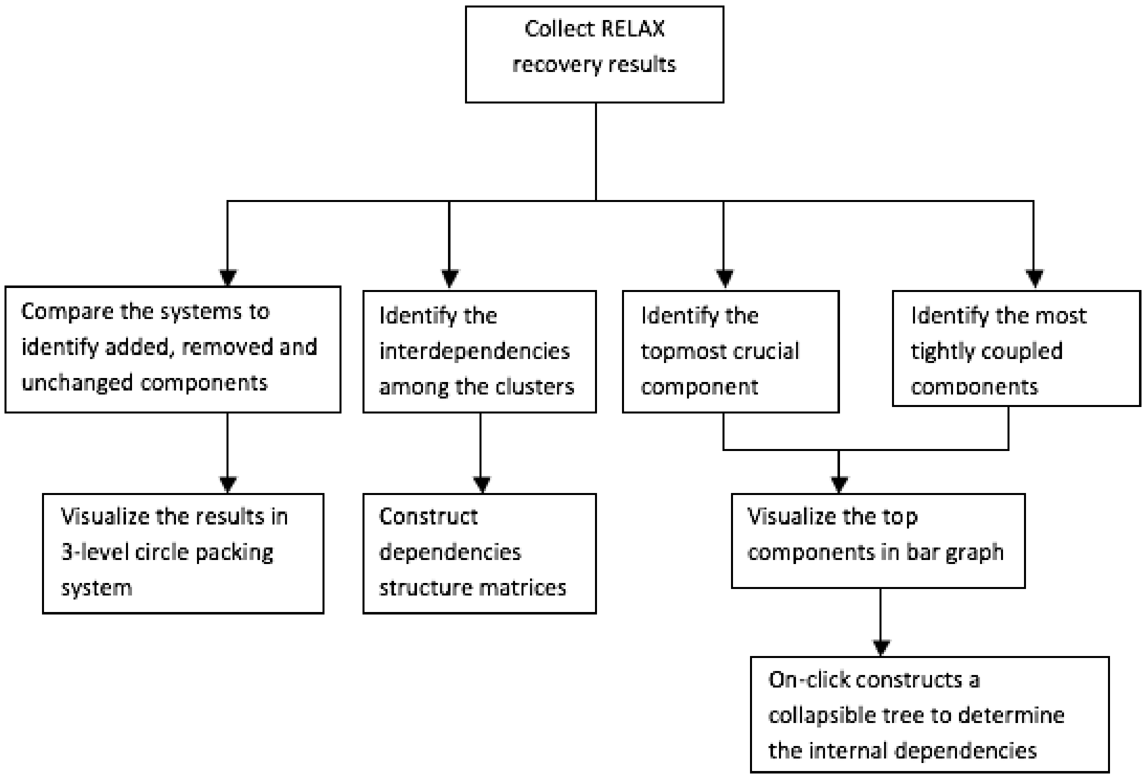

Collect RELAX recovery results
Compare the systems to identify added, removed and unchanged components
Identify the interdependencies among the clusters
Identify the topmost crucial component
Identify the most tightly coupled components
Visualize the results in 3-level circle packing system
Construct dependencies structure matrices
Visualize the top components in bar graph
On-click constructs a collapsible tree to determine the internal dependencies

Figure 3.1 Overview of Implementation of Solution

4. Architectural Style:

The underlying architecture for this tool is Client-Server architecture. The recovered architecture dependencies and cluster files will be the input to our server side which will process and give outputs based on requested section (out of 4 parts), and will display it in the form of a webpage. Here, client and server exist in the same system. The client side will initiate the communication to the server by just a button click and the server will process and deliver the output accordingly. We have used separation of concerns methodology for the server side, which provides four services based on four different concerns.

The backend part uses Python language and contains four functionalities in one source code file. The frontend part uses Highcharts and D3.js as visualiazation tools; languages like HTML, CSS, JavaScript; plugins like jQuery, bootstrap. The outputs are represented as Clusters, Matrices, and Bar Charts.

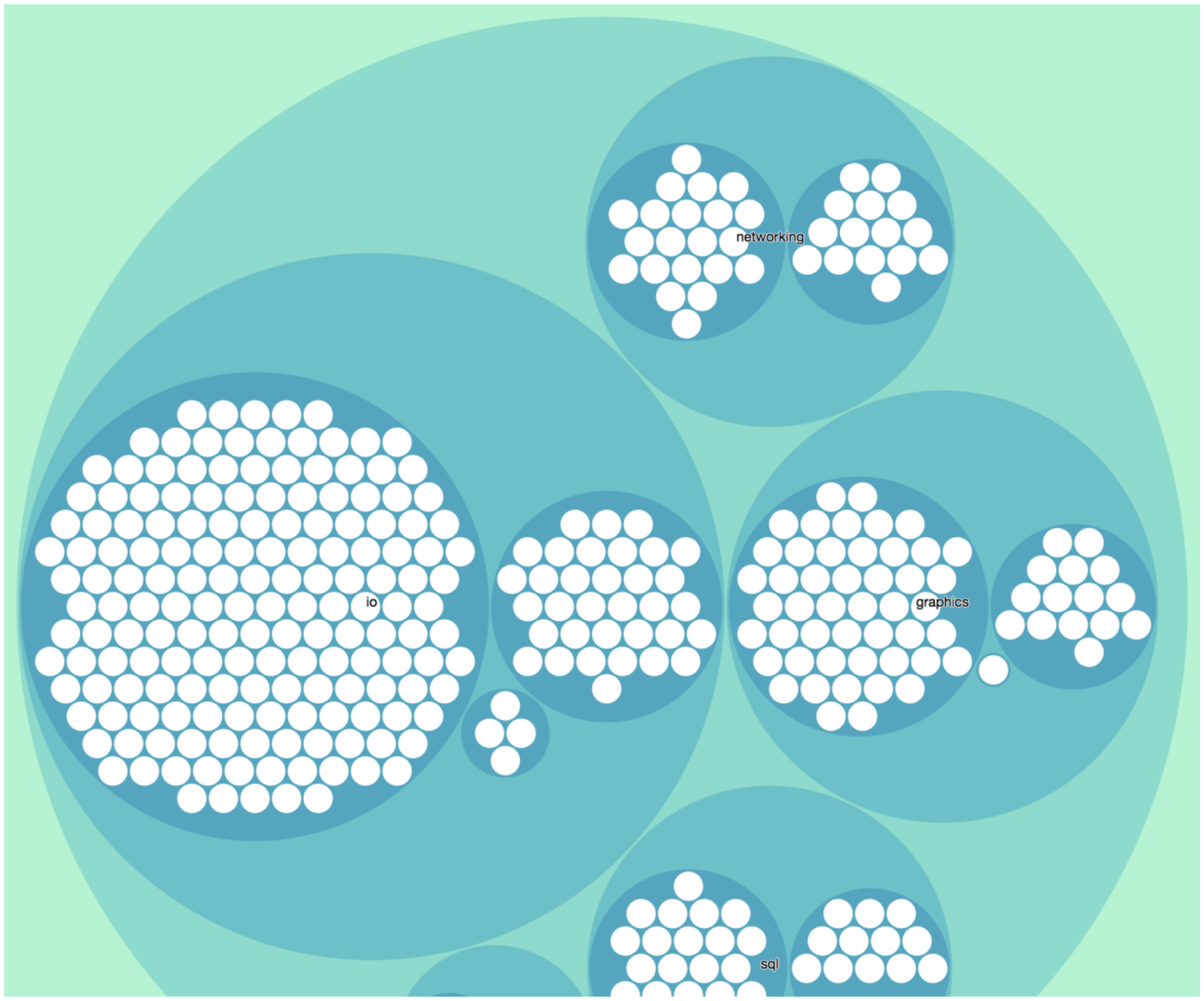


Figure 4.1 3-Level Circle Representation for Cluster Changes

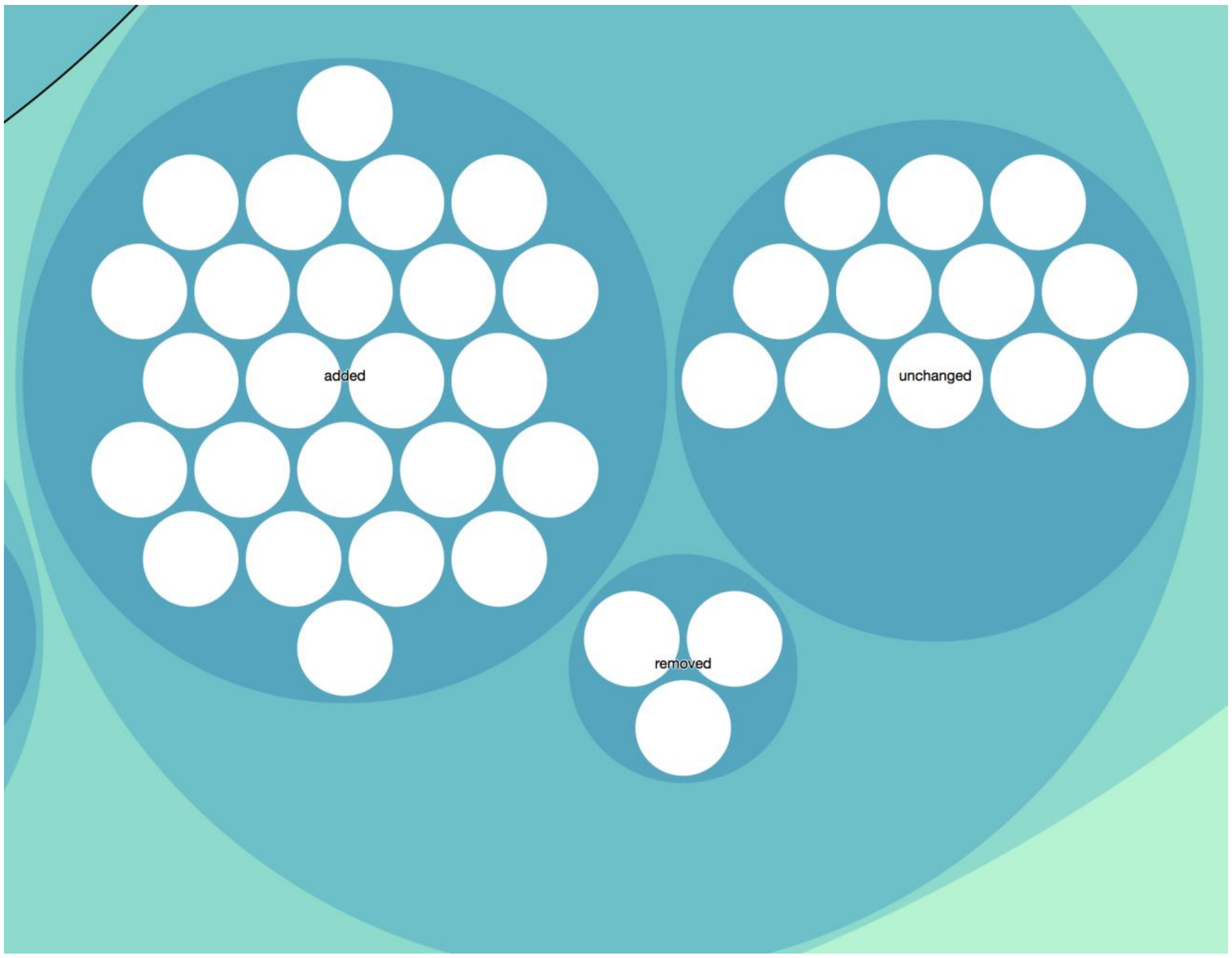
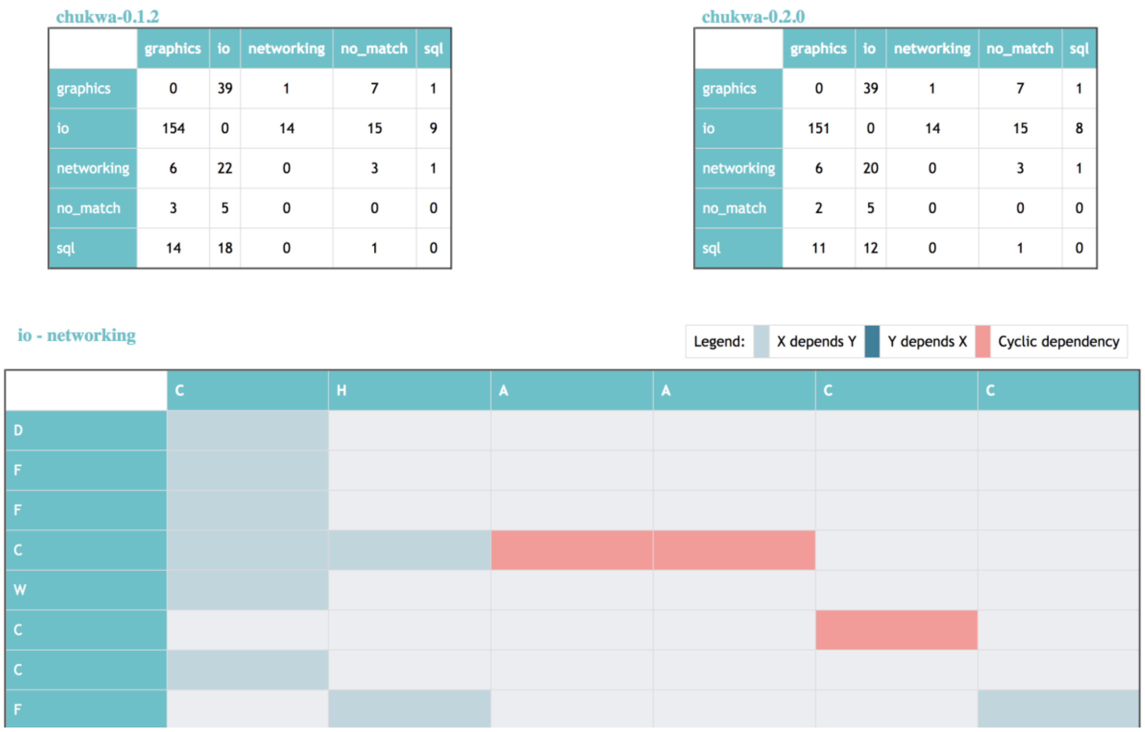


Figure 4.2 Cluster Representation of changes in two versions of a system



chukwa-0.1.2

	graphics	io	networking	no_match	sql
graphics	0	39	1	7	1
io	154	0	14	15	9
networking	6	22	0	3	1
no_match	3	5	0	0	0
sql	14	18	0	1	0

chukwa-0.2.0

	graphics	io	networking	no_match	sql
graphics	0	39	1	7	1
io	151	0	14	15	8
networking	6	20	0	3	1
no_match	2	5	0	0	0
sql	11	12	0	1	0

io - networking

Legend: ░ X depends Y █ Y depends X 🟥 Cyclic dependency

	C	H	A	A	C	C
D	░					
F	░					
F	░					
C	░	░	🟥	🟥		
W	░					
C					🟥	
C	░					
F		░				░

Figure 4.3 Two Level Matrix Representation

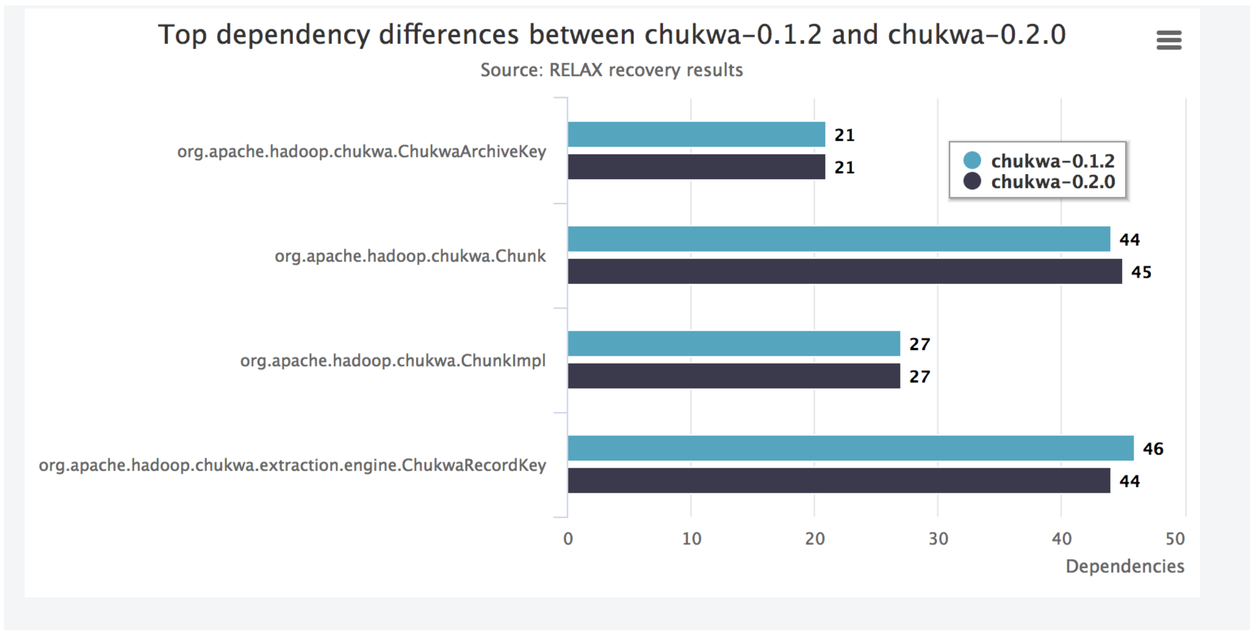


Figure 4.4 Bar Chart Representation for Interdependencies

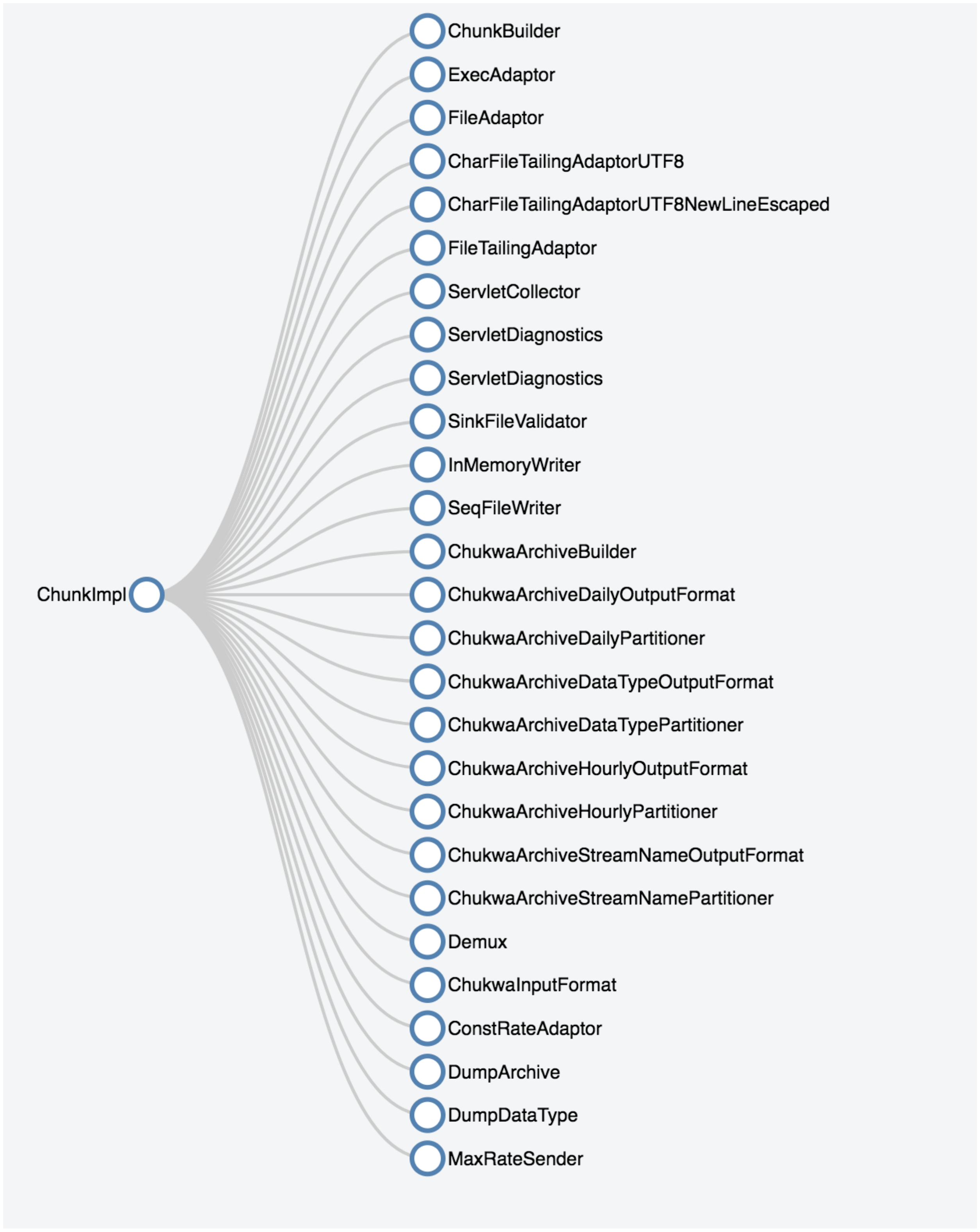

ChunkImpl
ChunkBuilder
ExecAdaptor
FileAdaptor
CharFileTailingAdaptorUTF8
CharFileTailingAdaptorUTF8NewLineEscaped
FileTailingAdaptor
ServletCollector
ServletDiagnostics
ServletDiagnostics
SinkFileValidator
InMemoryWriter
SeqFileWriter
ChukwaArchiveBuilder
ChukwaArchiveDailyOutputFormat
ChukwaArchiveDailyPartitioner
ChukwaArchiveDataTypeOutputFormat
ChukwaArchiveDataTypePartitioner
ChukwaArchiveHourlyOutputFormat
ChukwaArchiveHourlyPartitioner
ChukwaArchiveStreamNameOutputFormat
ChukwaArchiveStreamNamePartitioner
Demux
ChukwaInputFormat
ConstRateAdaptor
DumpArchive
DumpDataType
MaxRateSender

Figure 4.5 Collapsible Tree Representation for Interdependencies

5. Architectural Components:

1) Developer/Architect as Client: The four concerns are raised by the architect or developer are to be resolved. So, the architect/developer requests for services from the server and works as a client in the system.

2) Server: The server hosts the python file and processes the request of the user. It outputs it in the form of clusters, matrices, and bar charts. This output is feasible and easier to understand for the client. So, the server responds the client by resolving the requested concern and presenting it precisely.

3) Python file: The python code has the logic for resolving all the concerns. It does JSON parsing after achieving the objective and outputs JSON files which works as an input for the visualization tool files.

4) HTML files: There are multiple html files each for single concern. It takes the JSON file (output of python file) as its input and processes it using visualization tools and plugins, and gives the final output i.e. the final visualization.

6. Conclusion:

The prime goal of this tool is to decrease the workload of the developer and the architect while building a new version of a system. This visualization tool provides strategies to solve four different concerns and is implemented successfully. The decision-making task for the architect is simplified by observing the impacts of each dependency and cluster change in multiple versions of the system. The visualizations created are in the simplest form hence, easy to understand and straightforward to note changes.

7. Limitations:

- ✓ The second level matrix for larger input size can be difficult to peruse. Despite the first level matrix resolving scalability issue, the second level matrix fails to do the same. It is easy to observe the comparison for moderate number of dependency components.

- ✓ Compares only two versions at a time.

8. Future Enhancements:

- ✓ The same tool can be implemented for class files for multiple versions of the system. Hence, we can find out how tightly the files are coupled based on its class dependencies. With this information, we can make changes in the class dependencies such that the classes become loosely coupled and avoid self-loops to expand proficiency.

- ✓ By using the data created by our tool for multiple versions, we can perform various machine learning algorithms to find a pattern in the impact of every change made in the system. This makes it easier to build an efficient version of the system.