Towards a Reference Architecture with Modular Design for Large-scale Genotyping and Phenotyping Data Analysis: A Case Study with Image Data

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Amit Kumar Mondal

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

With the rapid advancement of computing technologies, various scientific research communities have been extensively using cloud-based software tools or applications. Cloud-based applications allow users to access software applications from web browsers while relieving them from the installation of any software applications in their desktop environment. For example, Galaxy, GenAP, and iPlant Colaborative are popular cloud-based systems for scientific workflow analysis in the domain of plant Genotyping and Phenotyping. These systems are being used for conducting research, devising new techniques, and sharing the computer assisted analysis results among collaborators. Researchers need to integrate their new workflows/pipelines, tools or techniques with the base system over time. Moreover, large scale data need to be processed within the time-line for more effective analysis. Recently, Big Data technologies are emerging for facilitating large scale data processing with commodity hardware. Among the above-mentioned systems, GenAp is utilizing the Big Data technologies for specific cases only. The structure of such a cloud-based system is highly variable and complex in nature. Software architects and developers need to consider totally different properties and challenges during the development and maintenance phases compared to the traditional business/service oriented systems. Recent studies report that software engineers and data engineers confront challenges to develop analytic tools for supporting large scale and heterogeneous data analysis. Unfortunately, less focus has been given by the software researchers to devise a well-defined methodology and frameworks for flexible design of a cloud system for the Genotyping and Phenotyping domain. To that end, more effective design methodologies and frameworks are an urgent need for cloud based Genotyping and Phenotyping analysis system development that also supports large scale data processing.

In our thesis, we conduct a few studies in order to devise a stable reference architecture and modularity model for the software developers and data engineers in the domain of Genotyping and Phenotyping. In the first study, we analyze the architectural changes of existing candidate systems to find out the stability issues. Then, we extract architectural patterns of the candidate systems and propose a conceptual reference architectural model. Finally, we present a case study on the modularity of computation-intensive tasks as an extension of the data-centric development. We show that the data-centric modularity model is at the core of the flexible development of a Genotyping and Phenotyping analysis system. Our proposed model and case study with thousands of images provide a useful knowledge-base for software researchers, developers, and data engineers for cloud based Genotyping and Phenotyping analysis system development.

Acknowledgements

First of all, I would like to express my heartiest gratitude to my respected supervisors Dr. Chanchal K. Roy and Dr. Kevin A. Schneider for their constant guidance, advice, encouragement and extraordinary patience during this thesis work. I am deeply indebted to Dr. Banani Roy, the Research Associate of our group, for her direct supervision of the projects of this thesis work and helping shape the thesis in general. Without three of them, this work would have been impossible.

I would like to thank Dr. Gord McCalla, Dr. Mark Keil, and Dr. Khan A Wahid for their willingness to take part in the advisement and evaluation of my thesis work. I also thank to Dr. Kevin Stanley and William Van Der Kamp for their help.

Thanks to all of the members of the Software Research Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Md. Saidur Rahman, Manishankar Mondal, Masudur Rahman, Kawsar Wazed, Shamima Yeasmin, and Muhammad Asaduzzaman.

I am grateful to the Department of Computer Science of the University of Saskatchewan for their generous financial support through scholarships, awards and bursaries that helped me concentrate more deeply on my thesis work.

I thank the High Performance Computing team of the University of Saskatchewan for their support. I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me to reach at this stage. In particular, I would like to thank Gwen Lancaster, Greg Oster, and Heather Webb.

I also thank to my wife Joyshree Mallick for her sacrifices. I express my heartiest gratitude to my father Anukul Chandra Mondal and my mother Arpita Rani Mondal who are the architects of my life. Their endless sacrifice has made me reach at this stage of my life. My parents, and my wife have always inspired me in completing my thesis work.

Disclaimer

Chapter 4 in this thesis dissertation has been published in the 2017 International Conference on Software Architecture (ICSA 2017) with title, *"Towards a Reference Architecture for Cloud-Based Plant Genotyping and Phenotyping Analysis Frameworks"*. I significantly contributed in analysis, implementation and writing of the paper under the direct supervision of Banani Roy.

I dedicate this thesis to my father, Anukul Chandra Mondal, whose inspiration helps me to accomplish every step of my life.

Contents

List of Tables

List of Figures

List of Abbreviations

SCUBA	Self Contained Underwater Breathing Apparatus
LOF	List of Figures
LOT	List of Tables

Chapter 1

Introduction

1.1 Motivation

Problems in the lifecycle of software development, maintenance, and adaptation with continuous changes are well defined by the researchers. Ren et al. [102] report that 90% of software life cost is related to maintenance. Over the last decades, software researchers have been working for more effective development methodologies as architectural models [35]. Software systems for various domains have various challenges in terms of development, maintenance, and adaptation. Data to analyze in scientific research is ever growing. Management, storage, processing, analysis, and sharing of data impose a great challenge to the researchers. In light of making scientific computations reproducible to the end users, computer scientists [111] have been building flexible tools. With the rapid advancement of computing technology, various scientific research communities have been extensively using both desktop and web-based software tools as eLab. For example, Genotyping and Phenotyping analysis systems work on various types of data of large volume: Genome data, Geospatial data, Image data, sensors data, and so on. In this domain, the necessity of cloud based systems is becoming popular for collaboration and research. Some of the existing systems are: Galaxy [49], iPlant Collaborative [83], GenAP [68], and so on. To process large data, Big Data frameworks are emerging that leverage the cluster processing. Among them, many data-scientists are using Spark [7], Hadoop [2], Pig [96], Google-Data-Flow [11]. Still, Big Data platforms cannot be easily integrated with the interactive data-analytic framework. Research on Big Data, design methodologies and software architecture are evolving with the emergence of the latest technologies.

Nonetheless, usual development model might not be a good fit for this domain. For instance, Aniche et al. [15] illustrate that the mostly used model-view-controller (MVC) architecture has several issues with a cloud-based system. Likewise, Torres [121] suggests in his article that "Context is the King" for developing a suitable software architecture. Therefore, more effective design methodologies are warranted for cloud based Genotyping and Phenotyping analysis systems. Another key thing to remember is that in these systems, heterogeneous large scale data are processed and analyzed. In a recent study, Saltz et al. [107] report an interesting finding that even experienced software engineers and data engineers are facing challenges for large scale data analysis. Consequently, more abstract and flexible frameworks are required to develop that can reduce the efforts of the developers. Considering the challenges and importance of Genotyping and

Phenotyping analysis, in this thesis, we present a case study for developing a reference architecture and modularity model for this Genotyping and Phenotyping analysis systems.

1.2 Problem Statement

Many cloud based systems are using traditional development methodologies such as model view controller architecture, three tier architecture, and infrastructure model. However, in the domain of Genotyping and Phenotyping analysis, only a brief description of their reference architecture is available for a few of them. Recently, tools in other domains are being developed following layer based and workflow based architectures for large scale data analysis. Unfortunately, none of the works provide extensive study for the design methodologies and guidelines for the Genotyping and Phenotyping analysis systems. Therefore, an important research question "How could cloud based Genotyping and Phenotyping analysis system that supports large scale data analysis be flexibly designed and maintained with minimal efforts?" is required to answer.

To find the answer, an analytical study of the architectural changes in various development phases of the candidate systems is essential to extract a knowledge-base. After that, complex architectural properties should be extracted in order to devise an effective architectural model. Finally, a technique for modularizing complex components with the cloud and Big Data technologies is warranted. However, none of the existing studies focused on the study of the above-mentioned directions for Genotyping and Phenotyping analysis systems explicitly.

1.3 Our Contribution

Following the above concerns, we performed a few case studies on architectural model and modularisation.

First Study

Perfect architecture design is still a challenging task for the developers of emerging computer-supported domain. A few studies show that even the popular MVC architecture creates a serious problem during the development of a web based system. Genotyping and Phenotyping analysis systems are large and complex in nature. Identification of recurrent problems related to system architecture, and devising a feasible architectural model is essential to reduce maintenance and development effort. To this end, we present an empirical study of the architectural changes and issues in the development phases of existing systems in this domain. We present a key-term based technique to detect architectural change related activities from development artifacts. Then, employing this technique, we extract commits related to architectural changes. We detect the subcomponents which are frequently changing architecturally from those extracted commits. We found these entities are different from usual customer or business based software systems for Genotyping and Phenotyping. Moreover, we figure out that architectural quality metrics (which are directly related to maintenance and bugs/issues) in various releases are also unstable to adopt continuous changes as these systems follow the traditional model. In addition, those releases are prone to more bugs. Therefore, our empirical study shows

that architectural changes are influential during the development phases of the Genotyping and Phenotyping analysis systems. Our proposed technique will be helpful for the project managers and system architects for the architectural analysis and maintenance activities, and automatic summary generation of architectural changes. Furthermore, our empirical study prompts a road map to develop new design principles and concrete architectural model for adopting continuous changes in this domain.

Second Study

Reproducible computation in the domain of plant Genotyping and Phenotyping is challenging in various perspectives especially handling the large volume of data. Various tools and systems have been developed to automate the scientific workflows and support the computational needs of this domain. In this study, we review a number of widely used systems (i.e., Galaxy, iPlant Collaborative, GenAp, and LemnaTec) in the domain of plant Genotyping and Phenotyping using the scenario-based architectural analysis method (SAAM). In particular, we focus on how different stakeholders are using these systems in a variety of scenarios and to what extent the systems support their needs. Our SAAM analysis shows that the existing systems have shortcomings. For example, they are limited in their support for high throughput processing of large amounts of heterogeneous types of data, virtual plant simulation, plugin integration, and so on. Based on our findings we propose a reference architecture along with a preliminary evaluation in the subject domain. The reference architecture and its evaluation is aimed at helping developers/architects create suitable architectural designs and select appropriate technologies when developing plant Genotyping and Phenotyping systems.

Third Study

With the rapid advancement of Big Data platforms such as Hadoop, Spark, and Dataflow, many tools are being developed that are intended to provide end users with an interactive environment for large-scale data analysis (e.g., IQmulus). However, there are challenges using these platforms. For example, developers find it difficult to use these platforms when developing interactive and reusable data analytic tools. One approach to better support interactivity and reusability is the use of micro-level modularisation for computation-intensive tasks, which splits data operations into independent, composable modules. However, modularizing data and computation-intensive tasks into independent components differ from traditional programming, e.g., when accessing large scale data, controlling data-flow among components, and structuring computation logic. In the final study, we present a case study on modularizing real world computation-intensive tasks that investigates the impact of modularization on processing large scale image data. To that end, we synthesize image data-processing patterns and propose a unified modular model for the effective implementation of computation-intensive tasks on data-parallel frameworks considering reproducibility, reusability, and customization. We present various insights of using the modularity model based on our experimental results from running image processing tasks on Spark and Hadoop clusters.

1.4 Related Publications

Some parts of this thesis have been published previously. Architectural change related commit extraction in our first study is motivated by the key-term based valuable information mining from the software artifacts discussed in the second paper.

- Banani Roy, Amit K. Mondal, Chanchal K. Roy, Kevin A. Schneider and Kawser Wazed, "Towards a Reference Architecture for Cloud-based Plant Genotyping and Phenotyping Analysis Frameworks", In Proceedings of the *International Conference on Software Architecture (ICSA 2017)*, Gothenburg, Sweden.

- Amit K. Mondal, M. Masudur Rahman and Chanchal K. Roy, "Embedded Emotion-based Classification of Stack Overflow Questions Towards the Question Quality Prediction", In Proceedings of the 28th *International Conference on Software Engineering and Knowledge Engineering (SEKE 2016)*, California, USA.

In Chapter 2 we discuss some background related to the cloud architecture of Genotyping and Phenotyping analysis systems, and Big Data technologies. Chapter 3 describes architectural change analysis of existing Genotyping and Phenotyping analysis systems. Chapter 4 elaborates our study for a conceptual architectural model. Chapter 5 presents a case study for modularising computation-intensive programs with Big Data platforms. In Chapter 6 we conclude our thesis and discuss about our future work.

Chapter 2

Background

2.1 Genotype and Phenotype Analysis Frameworks

Genotyping [119] is the process of determining differences in the genetic make-up (genotype) of an individual by examining the individual's DNA sequence using biological assays and comparing it to another individual's sequence or a reference sequence. It expresses the variant form of the given genes an individual has inherited from their parents. Traditionally genotyping is the use of DNA sequences to define observable traits of a population. In plant genetics and crop improvement programs, the study of Genomic variation is an essential task [43]. In many cases, phenotype differences are directly related to DNA polymorphisms such as genetical linkage to its causative factor or relationships between individuals in the populations [43]. In phenotyping analysis, image processing plays an influential role [52]. Plant genotyping and phenotyping is important for ensuring global food security. Plant genotyping and image-based plant phenotyping involve the generation and management of large amounts of data [32]. Genotyping involves the generation of DNA sequencing applying different methods (such as next generation sequencing) and results on a huge volume of data from around the globe. Plant phenotyping is the appraisal of complex plant traits including growth, development, tolerance, resistance, architecture, physiology, ecology, yield and the basic measurement of individual quantitative parameters that form the basis for the more complex traits [106]. Plant genotyping and phenotyping analyses involve numerous steps including physical plant sample collections, data curation, data conversion into different steps for generating users' expected end results, and making analysis results available to researchers and practitioners if needed. For plant genotype and phenotyping analysis, the most valuable data includes Genome and sequence data, Geospatial data, sensor data, and image data. The ultimate objective of a cloud based framework is to make scientific computation reproducible (along with accessibility and transparency) to the users with minimal technical knowledge.

From the perspective of the users, the core features of a cloud-based Genotyping and Phenotyping analyses framework are presented in Table 2.1. Among them features F1 and F2 are considered the most architecturally critical features. Feature F3 and F6 are considered as interactive and reproducible data-analysis. With the advancement of Big Data technology, for large scale data, features F1, F2, and F3 have gained new momentum in computer science research. Before starting the development of the system, these features are used to construct a logical structure of the system considering the challenges, feasibility and existing technologies.

Table 2.1: Core feature model of cloud-based Genotyping and Phenotyping analyses frameworks

Serial	Feature name	Description
F1	Data processing	Mixed types and various volumes of data are processed
F2	Data management	Storage management of mixed types and various volumes of data
F3	Workflow decomposition	Drag-drop working UI for workflow and pipelines composition for analyzing the data
F4	Third party app communication	Inter-communication with third party service, app and library for various utilities including visualization
F5	Collaboration	Messaging, Audio, video communication among multiple data-users
F6	Share data and workflows	Export import decomposed workflows and pipelines and result data
F7	Plugin integration	Tools, apps, plugins addition with the system

This logical structure guides the project manager, software architect, programmers, and stakeholders in various ways during the full development and maintenance phases.

2.2 Reference Architecture

Software architecture [18, 101] is formally defined as the fundamental structures of a software system, the design guidelines of such structures, and the details description of these structures. Each structure comprises of [18, 101, 40] software elements, connection and communication among them, and properties of both elements and relations, logic for the introduction and configuration of each element. Corazza et al. [40] define software architecture as the partition of classes into groups (based on close relations) for automatic clustering based on the structure. Czibula and Serban [55] also consider the base of software architecture as groups of program files such that files within a group are similar to one another and different from those in other groups. A number of studies [14, 89] illustrated the importance of reference architecture (RA). A well-defined software reference architecture facilitates a project team reusing architectural knowledge and components in a systematic way [14]. Nakagawa et al. [89] define a reference architecture as *"an architecture that encompasses the knowledge about how to design concrete architectures of systems of a given application [or technological] domain; therefore, it must address the business rules, architectural styles (sometimes also defined as architectural patterns that address quality attributes in the reference architecture), best practices of software development (for instance, architectural decisions, domain constraints, legislation, and standards), and the software elements that support development of systems for that domain. All of this must be supported by a unified, unambiguous, and widely understood domain terminology"*. For instance, Castelan et al. [36]

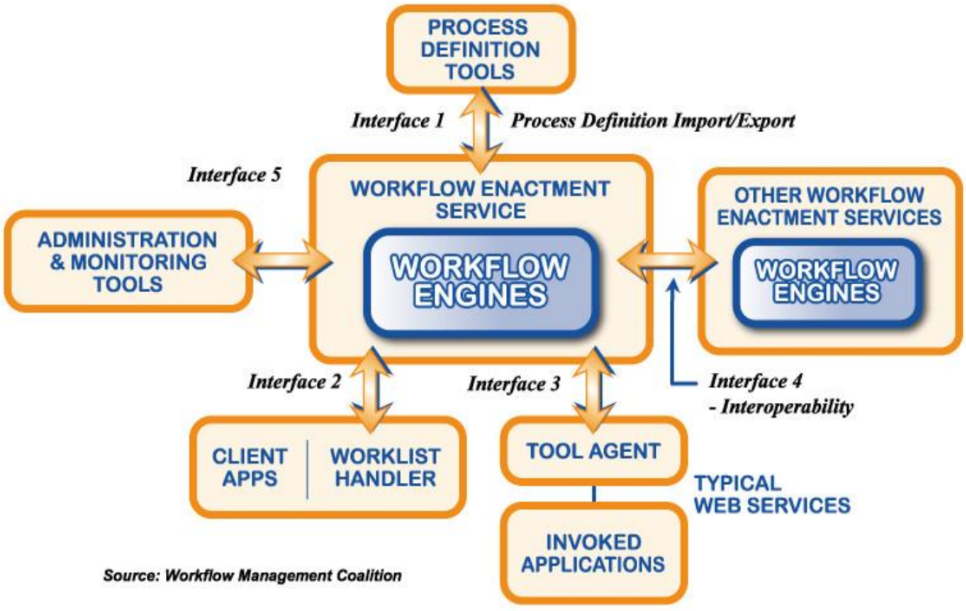

Figure 2.1: A reference architecture for workflow learning applications. Adapted from Maldonado et al. [36].

develop a concrete architecture for a workflow learning system following the reference model (shown in Figure 2.1) defined by the workflow management coalition. They present the RA with Mobile, Cloud and Collaborative functionalities adapting design science research methodology in an elegant way so that a team can follow this pattern to develop workflow management system for other domains as well.

Most of the popular cloud based systems such as Galaxy [49], iPlant Collaborative [83] followed traditional MVC and three tier-architectural patterns. Up to June 2017, they only briefly provide architectural documentation on how they implemented the systems. But, no standard design guidelines (as presented in [36]) have been studied yet to present a common reference architecture. Moreover, most of these systems are struggling to provide an interactive environment for large scale data processing. Although Galaxy project attempted to follow MVC architectural pattern, it is developed into four high-level components:

- The toolbox: Handles bioinformatic tools, plugins, workflows, and so on
- The job manager: Runs and manages the data processing jobs
- The model: Handles data store and I/O
- The web interface: GUI and user interaction

Therefore, it appears from the Galaxy project documentation that they have not specified any strict development strategy. iPlant Collaborative presents their system architecture on cloud-service point of view. iPlant Collaborative is a combination (shown in Figure 4.5) of three different services implemented with different technologies and hard to follow their architecture. GenAP [68] has extended the Galaxy project and added

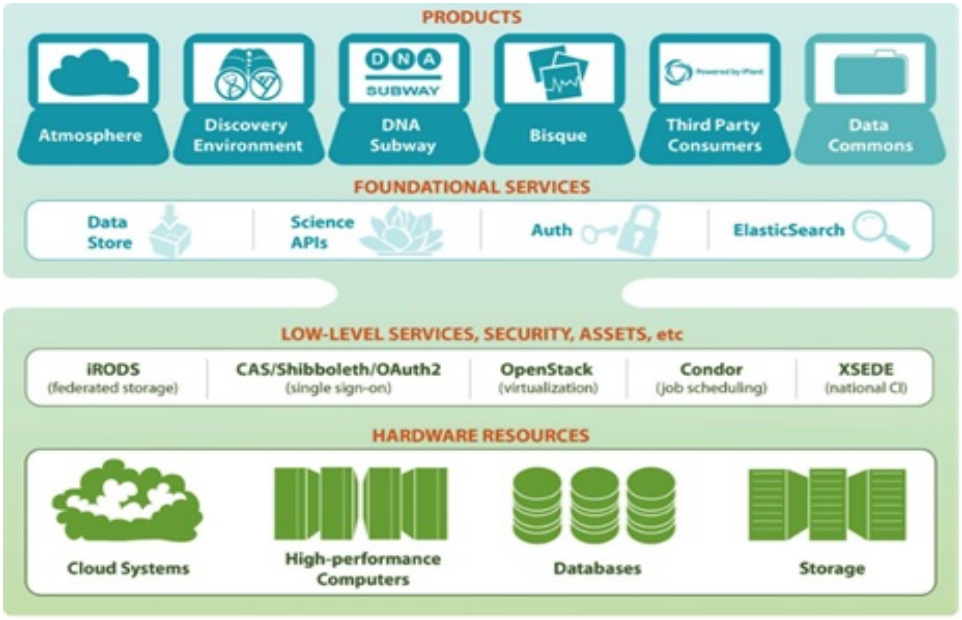

Figure 2.2: Official architecture of iPlant Collaborative. Adapted from Merchant et al. [83]

some extra services to interface with Big Data technologies and has similar documentation for the Galaxy part. Infrastructure part of GenAP is described by the development team as shown in Figure 4.8.

Overall, complexities of these potential systems development are expected to be high for both development and maintenance perspective. Therefore, feasible design guidelines and reference architecture are essential to follow. In this thesis, we attempt to analyze the stability of that architecture during the development phases, extract architecture on components level, and propose a reference architectural model.

2.2.1 Architectural Analysis

Various techniques [78, 35, 34, 58, 126] exist to extract meaningful insights, information about maintenance related activities, and issues of a software structure. One of the fruitful methods is to analyze development artifacts [46, 62, 65, 87, 39, 90]: source code, code-base from release to release, commit messages, bugs, and issues. In order to analyze issues, existing literature [39] also report the importance of examining development history. However, manual analysis of thousands of artifacts is infeasible. Consequently, either a small portion is selected or search space is reduced. We attempted to reduce search space during the architectural analysis. Williams and Carver [126] create a *Software Architecture Change Characterizing Scheme* (SACCS) to analyze architectural changes. They presented the reason for changes in architecture: motivation, type, size, impact on static, impact on dynamic properties and effect on requirements (functional and non-functional). Many of these important SACCS properties might be used to detect architectural activities from the artifacts. Moreover, to analyze design quality of a system, different metrics are available [112, 33, 34, 98]. We revisited this literature for studying the stability of the architecture of the Genotyping and Phenotyping analysis systems.

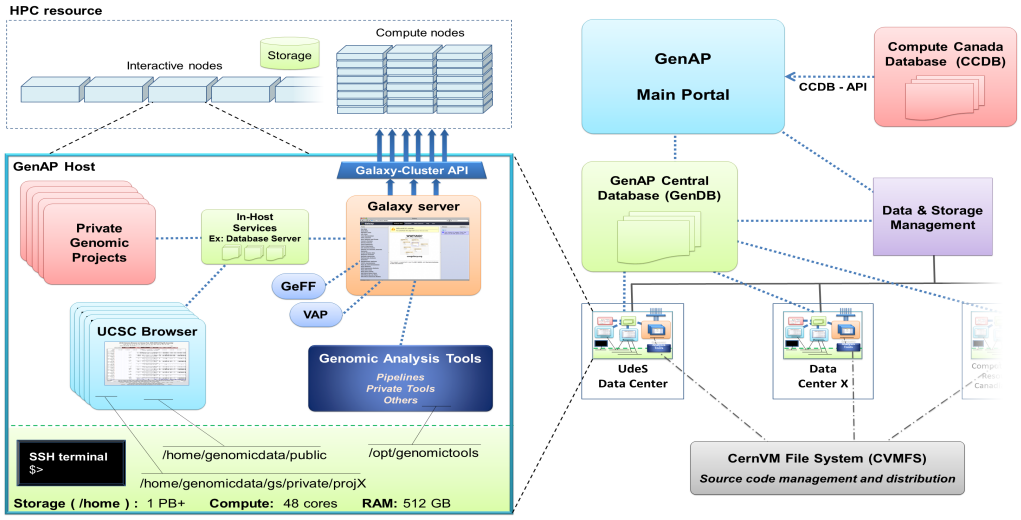

Figure 2.3: Official architecture of GenAp that add an extra layer for Big Data cluster. Adapted from [68]

Both for evaluating an architecture and re-engineering an architectural model there are a number of key attributes [13, 61, 60] to be considered: maintainability, portability, functionality, reliability, efficiency, modifiability, modularity, and reusability. With this in mind, an underlying architecture and design patterns of a domain can be identified from existing candidate systems using various analysis techniques [60, 38, 104, 69, 29, 105, 61, 6, 100]. Patidar and Suman [100] surveyed a number of methods for architectural evaluation from existing systems. One of the most widely used methods is proposed by Kazman et al. [60, 61] called *Scenario-based Architecture Analysis Method* (SAAM). SAAM is a pioneering work to evaluate how an architecture adopts the domain functionality and other nonfunctional qualities. SAAM is the base method that considers most of the previously mentioned quality attributes. Other methods such as ATAM, ALMA, SAAMCS, SBAR, ALPSM, ESAAMI focused on a few of the specific quality attributes individually. Notably, most of these methods are the modification of SAAM and might require extra inputs for evaluation (e.g., specification, historical maintenance activities, and so on). SAAM has been applied to numerous case studies for identifying design problems: global information systems, air traffic control, and so on [6, 38]. Considering the context, we followed SAAM to extract architectures and design problems of the existing Genotyping and Phenotyping analysis systems. The modified steps we adopted for evaluating the candidate architectures and devising a new architecture based on SAAM is presented in Figure 2.4.

2.3 Large Scale Data Analysis with Big Data Platforms

2.3.1 Big Data Technology

Over the past few years "Big Data" has been a buzzword in computing technology. Big Data is the advancement of cloud computing. Big Data technologies are being used in earth sciences, social sciences, IoT (internet

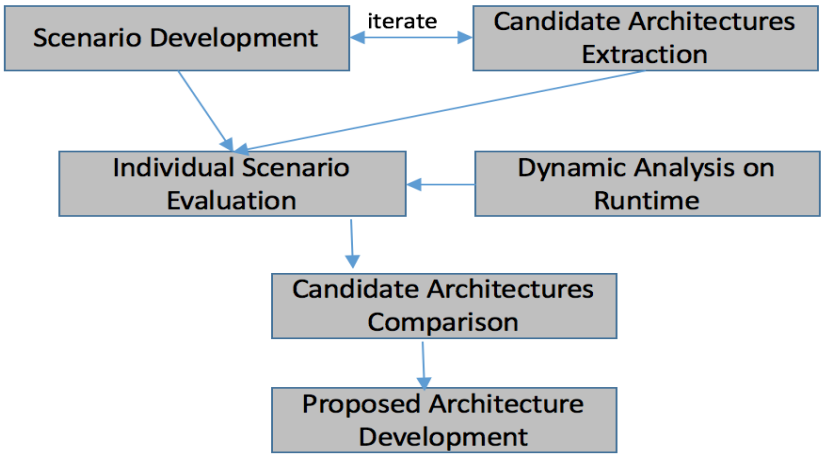

Figure 2.4: A methodology for the evaluation and development of a reference architecture based on SAAM. Adapted from [60]

of things), astronomy, business, government, industry, journalism, and so on [120]. Marr [81] describes Big Data using 5Vs:

(i) Volume- Ever growing data that cannot be handled with usual technologies

(ii) Velocity- Fast generation and transmission over the network

(iii) Variety- Various formats, models, and structures

(iv) Veracity- Quality and transparency

(v) Value- Should be valuable for decision making

In Genotyping and Phenotyping, analysis data involves Terabytes of Genomic data, Geospatial data, Image data and sensors data. These data-sets are expected to grow faster over times. Therefore, Big Data technologies can handle these datasets. Various platforms, techniques, and models are emerging [81, 99, 63] for handling Big Data problems. Distributed processing and storage is at the heart of Big Data, and this is possible with commodity hardware. Various Big Data technologies classified by Pääkkönen and Pakkala [99] are presented in Table 2.2.

Table 2.2: Classified Big Data technologies. Adapted from [99]

Data Collection and Storage	**Data analysis**	**Benchmarking**
Hive, HBase, HDFS, VoltDB, Voldemort, CouchDB, Cassandra, MongoDB, Graph, EV-cache, G-Store, Virtuoso, Pregel,GraphLab, Chukwa, Kafka, SparQL	Hadoop, MapReduce, Spark, Storm, FlumeJava, Pig, MrRunner, Dyrad, D-steams, Avatara, Mahout, Google-Data-Flow	LinkBench, BigBench, BigDataBench, Asterix, Flink

Among large collections of frameworks, map-reduce based distributed cluster processing (Hadoop, Spark, Google-Data-Flow) is widely used to process large scale data with commodity hardware. MapReduce [41] is a simple and powerful programming model that facilitates easy parallelization and distribution of large-scale

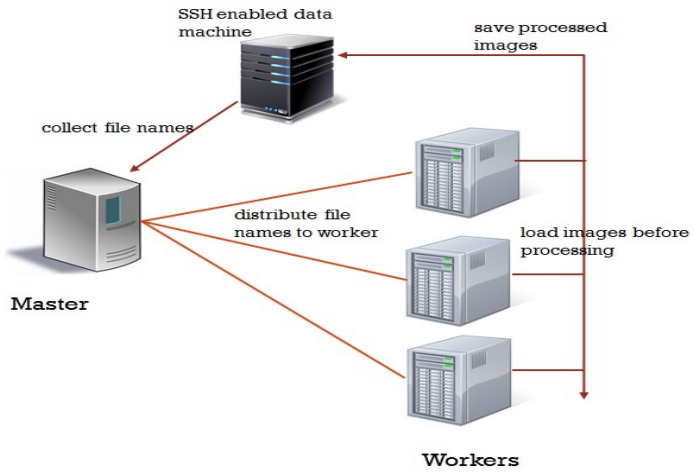

Figure 2.5: Processing of images with cluster

computations achieving high performance on large clusters. MapReduce framework shields us from many complexities [41] of distributed cluster processing such as parallelization, fault-tolerance, data distribution, and load balancing. A cluster is a loosely/tightly connected (secure way) multiple computers [80] over the network that can process a task simultaneously. For the distributed storage and accessing of data [99], HBase, HDFS, Cassandra, and so on platforms are being used. However, an open-source map-reduce based platform, Spark (along with Hadoop and HDFS) is optimized and the mostly used [117, 79] map-reduce framework for Big Data cluster. A Spark cluster [136] consists of a master node and multiple worker nodes as shown in Figure 2.5. Here, we call cluster management technology, map-reduce programming frameworks and distributed data storage combinedly as Big Data platforms. Cluster manager technologies such as HDP Sandbox [3], and Cloudera [57] are useful for flexible managing and configuring clusters. Job scheduler [57] (such as Yarn and Mesos) are also used for multiple jobs running into the cluster. Big Data platforms such as Hadoop [2], Spark [7], Google Data-flow [11], and so on provide us a high-level abstract interface for implementing distributed-cluster processing of data (mainly text data). However, researchers and programmers are trying to develop unified frameworks for their relevant domains for further abstraction for many others complex data processing such as Genome sequence, Image, and Geospatial data.

2.3.2 Unified Frameworks for Large Scale Data-processing

Unified programming framework is a combination of [54] denotational semantics, operational semantics and algebraic semantics for abstract programming that reduces complexities and efforts. Researchers and developers are developing unified frameworks [125, 92] for the relevant domains on top of the parallel and map-reduce programming API for further abstraction and flexibility of Big Data processing. For example, SparkSeq [125] is based on Hadoop-BAM [92] data frameworks. Hadoop-BAM is created to overcome the issues of map-reduce implementation and attempted to include all data formats in bioinformatics. A unified framework for large scale Geospatial data analysis, SpatialHadoop [45] added three more layers and two

abstract components on top of Hadoop to drive efficient map-reduce based processing of GIS data. KIRA [139] is written using SEP library on top of Spark and FITS data model for analyzing astronomical objects. FITS is a new data structure that embeds photometric and spatial calibration information together with image origin metadata with the images. However, for large scale image processing such unified framework with Big Data platforms is not readily available.

2.3.3 Development Strategies of Big Data Analytic Systems

To better understand the design guidelines as an architectural model, it is important to analyze the recent trends of application development that use the Big Data technologies. A number of systems have been attempted to develop an interactive and flexible analysis system for large scale data (in scientific research) utilizing the Big Data technologies. Among them, some applications follow a workflow based modularity architecture [70, 115] whereas others follow a layered based architecture [130, 67, 76]. In the workflow based modularity architecture, applications are designed using a special data model which is much different than the traditional model view controller model. For example, IQmulus [70] architecture is heavily dependent on data-analysis workflows. High-level components, job manager, processing services, and so on are designed focusing the on-the-fly workflow composition. Still, users need to learn a considerable amount of script for composing workflows for GIS. Similarly, GrayWulf [115] handles two types of workflows: (i) one is for the data manager, and (ii) another is for the end users. The architectural model is based on these workflows composition. However, through GrayWulf, a smaller amount of processed result can be shared and retrieved in the cloud. Another application, IABDT [67] followed multilayer architecture and primarily used Hadoop-ImageBundle (HIB) for performing basic operations on image data for few cases utilizing Hadoop ecosystem. All of these development strategies provide useful knowledge-base for devising a flexible design methodology and guidelines for cloud based Genotyping and Phenotyping analysis systems. Nonetheless, more effective architectural model and design guidelines are required for the Genotyping and Phenotyping analysis system development utilizing the Big Data technologies.

2.4 Conclusion

In this chapter, we have discussed Genotyping and Phenotyping analysis systems, architectural structure of some of the existing Genotyping and Phenotyping analysis systems, and reference architectures of some of the applications in other domain that utilize Big Data technologies. We have seen that reference architecture, design guidelines, and unified frameworks are being used for the development and maintenance of the software systems. We observed that existing studies do not provide concrete experiments on the research question "How could cloud based Genotyping and Phenotyping analysis system that supports large scale data analysis be flexibly designed and maintained with minimal efforts?". We have conducted three studies to find the answer to this question. These studies have been elaborated in the following chapters.

Chapter 3

Architectural Change Analysis of Genotyping and Phenotyping Systems

3.1 Introduction

Software requirements [19] are being changed as change is inevitable. To reflect the changes, sometimes system structure (e.g., components, infrastructure, interface) is broken which is related to the architectural changes, and it is considered an expensive [19] task. If the architecture of a system is not defined intelligently at the early stage of the development, the system undergoes changes and changes cause issues. For example, in an empirical study, Aniche et al. [15] demonstrate that for the Web-based system development the most popular architectural model, MVC increases change and defect-proneness, and many experiences the impacts as severe problems. Therefore, stable architecture is a pressing need to adapt the continuously changing user requirements.

With the advancement of computing technology, many scientific developers are developing more complex system than the traditional software system. Recently, cloud based collaborative systems are emerging for Genotype and Phenotyping data analysis. Genotyping and Phenotyping analysis has an enormous effect to solve many risks related to the whole mankind. Three high-level requirements of a Genotyping and Phenotyping analysis system are: (i) scalability, (ii) reusability, and (iii) composability. The core features of this system are described in Table 2.1. The features list represents that Genotyping and Phenotyping analysis systems [68, 27, 83] are large and complex to design and implement. For example, Galaxy [49] project has 1190 Python files, and 175 modules (excluding web interface) and components until March 2017 for the Genotype data analysis. So far, 138 developers are involved in the parallel development of Galaxy. App creation and integration are few of the major features of Galaxy. If we only consider the app handling module of Galaxy, the micro-architecture involved for the functionalities consist of many components as shown in Figure 3.1. A simple structural change for adopting new requirements might increase the development and maintenance efforts significantly to the project managers and developers.

Above all, maintenance and development efforts of these systems are expected to be high and required to be handled the complexities arisen from continuous changes and the introduced bugs. Software developers need domain centric [121] design principles and architectural model easily adaptable to the changes. Due

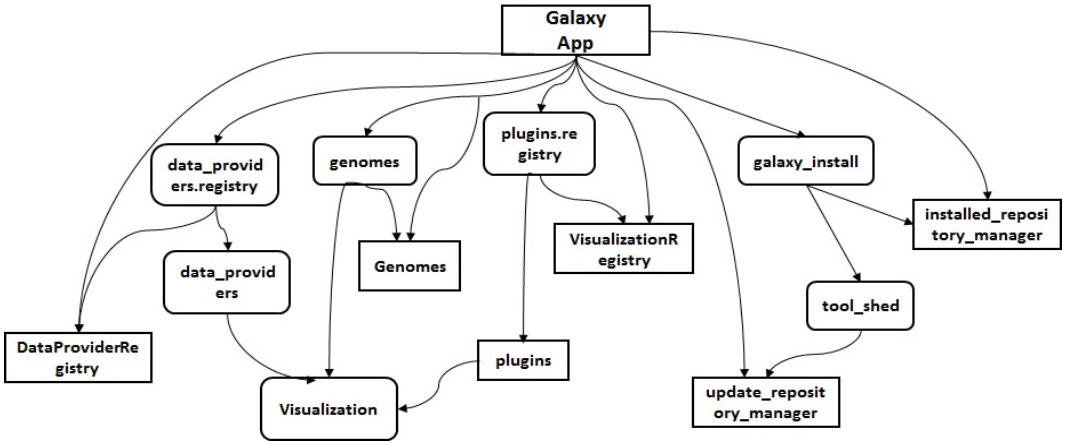

Figure 3.1: Micro-architecture of Galaxy for apps handling module

to the huge number and frequency of changes that mature systems undergo, software maintenance has been regarded as the most expensive phase of the software lifecycle [126, 102]. Torres [121] provide an empirical study on the importance of considering the context for designing a software system. Therefore, more research is warranted to analyze the context and identify challenges before developing a feasible architectural model for Genotyping and Phenotyping analysis systems. Furthermore, during development, architectural change analysis of the previous versions is essential for maintenance and decision making for future changes. To that end, existing studies [126] extracted the benefits of architectural change analysis: (i) Change understanding and architecture analysis, (ii) Build historical baseline of software change data, (iii) Development group changes based on impact/difficulty level, (iv) Facilitate discussion among developers, and (v) Facilitate change difficulty/complexity estimation. Earlier studies [46, 62] focus on to the analysis of the effects of requirement changes, code changes, and import changes from the software development history. A number of studies [112, 59, 19, 13] focus on to the architectural evolution and stability analysis in other domains. Maffort et al. [78] represent a technique for mining architectural violations from high-level specifications and version history. In this study, we conduct analysis on architectural changes from various perspectives such as structural changes, design quality metrics, bug-proneness, and intentional change activities.

Developers sometimes express their intention in the description of a commit which could be useful for architecture-level change impact analysis, the reason of changes, candidate components of change, and traceability analysis (e.g., co-evolution of both the requirements and the architecture). There could be thousands of commits in the development phases of a system, and manually analyzing all these commits is not feasible. Therefore, automatic techniques are essential in order to consider various perspectives. However, to the best of our knowledge, no study directly [90, 126, 58] focuses the architectural changes and their impact during the development lifecycle of Genotyping and Phenotyping analysis systems. Moreover, we do not find any automatic technique yet to identify architectural change related activities from software artifacts. To that end, we also focus on a technique for detecting deliberate architectural commits to reduce search space for more intuitive analysis from a large collection of development history.

In this study, we first detect and filter structurally changed releases. Then we extract architectural change related commits using a key-term-graph based technique. We found most of the releases that undergo structural changes contain intentional commits for changing the architecture. After that, by measuring and analyzing design quality metrics, we observed those releases are unstable in terms of design; bugs are also introduced in those releases. Finally, from the architectural change related commits, we identified the frequently changed subcomponents and noticed that actions performed by these components are different from software systems for business and service organizations. This technique is valuable for the architecture traceability analysis [19] and decision making for selecting reusable components [16]. Therefore, our empirical study verifies that the existing Genotyping and Phenotyping analysis systems undergo architectural changes significantly and the traditional architectural model appears to be problematic for developing these systems. All things considered, we focus on to the following research questions:

RQ1. Are the architectural changes significant during the development lifecycle of plant Genotyping and Phenotyping analysis systems?

RQ2. How to automatically analyze development history related to architectural changes from the development artifacts of Genotyping and Phenotyping analysis systems?

RQ3. What sub-components of Genotyping and Phenotyping analysis systems are prone to changes and cause issues?

At a higher level view, software architecture is of two type[126]: (i) logical or static, and (ii) runtime or dynamic. Our study focuses on static architecture (the commits detection technique identify the run time changes as well). To answer **RQ1**, we detected static changes from release to release, and calculated static dependencies related to the changed components. Additionally, we measured design quality metrics of those releases using state-of-the-art tools and techniques. To answer **RQ2**, we develop a technique to automatically identify commits to extract the recurrent activities related to architectural changes. To answer **RQ3**, we analyze architectural change related commits and source codes associated with them detected by our proposed key-term-graph based technique. Our study would be helpful for the system architect to develop design principles and concrete architectural model for implementing variable (easily adaptable to continuous changes) systems. The project manager could use the presented technique for the effortless analysis of the architectural change related commits and bugs. Moreover, frequently changed sub-components analysis technique can be used as a subtask to repair, restructure and refactor the architecture for future requirement changes. Furthermore, our study would be a starting point for the researchers to focus more on the development challenges of Genotyping and Phenotyping analysis systems. In the next sections, we subsequently describe our experimental methodologies.

Table 3.1: Candidate open-source projects for our study (May, 2017)

Project	**#Commits**	**#Developers**	**#Release**	**Data source**
Galaxy	26,782	138	47	https://github.com/galaxyproject/galaxy
ImageJ	9,080	15	77	https://github.com/imagej/ imagej
iPlant Collaborative (Discovery Environment)	7,972	8	39	https://github.com/cyverse-archive/DE
Bisque	2,876	> 15	8	https://biodev.ece.ucsb.edu/projects/bisquik
GenAP	1000	> 10	10	https://bitbucket.org/mugqic/mugqic_pipelines

3.2 Dataset Collection

We have collected code-base, commit history, and releases of some of the popular open-source projects related to Genotyping and Phenotyping analysis. Each of the projects contains thousands of commits and many releases. The sources of the code-bases of these open-source projects are mainly GitHub[1] and Bitbucket[2]. The selected projects for our study are Galaxy, iPlant Collaborative, and ImageJ which mainly work on Genomic data and Image data along with other users centric data (for few cases GIS data). The description of the collected datasets is presented in Table 3.1. Another two projects shown in the Table 3.1, GenAP and Bisque are related to Galaxy and iPlant Collaborative. Our first candidate system, Galaxy provides a cloud environment for Genotyping analysis (few private instances also provide image based phenotyping) [49]. One of the important features of Galaxy is that it has history and workflow composition along with export and import options which are unique to Galaxy compared to the other tools. GenAP is an extension of Galaxy to support large scale data analysis, data hub management pipeline, and options of separate working environment creation for the users. The second candidate system, iPlant Collaborative [83] provides a cloud framework for both Genotyping and Phenotyping analysis for plant science and agriculture. In addition, iPlant Collaborative facilitates flexible app creation, image annotation, Google map service for the images, and so on, which are unique to iPlant Collaborative. In fact, iPlant Collaborative is a combination of some separate applications: Discovery Environment (DE), Bisque, Atmosphere, and so on. Please note that in our study we consider only the Discovery Environment part of iPlant Collaborative. The third candidate system, ImageJ [110] is a desktop application for various image processing and analysis tasks. ImageJ provides unique

[1]https://github.com/
[2]https://bitbucket.org/

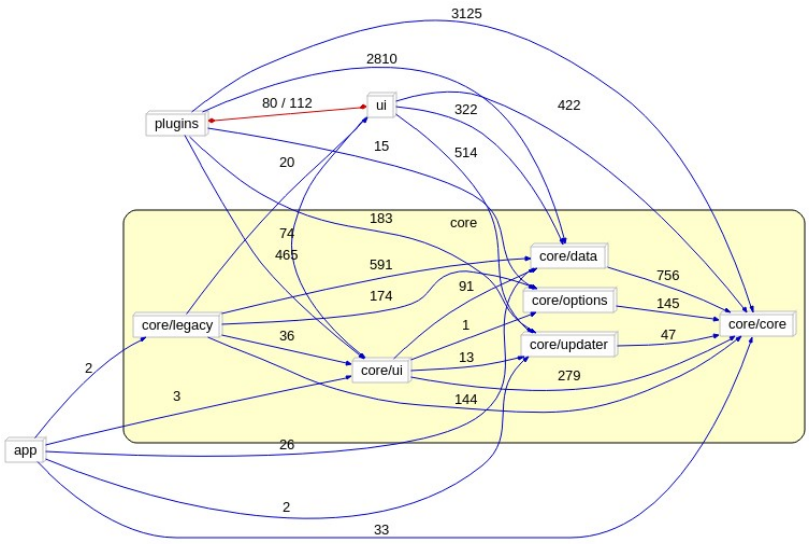

Figure 3.2: Abstract view of ImageJ architecture. High-level modules are: *core, app, ui, and plugins. core* module is decomposed into *legacy, ui, data, options, updater, and core* high-level components further.

options for integrating plugins for image analysis tasks. The abstract and major features of these systems are shown in an earlier Table 2.1. Therefore, these candidate systems cover many functionalities of Genotyping and Phenotyping data analysis.

3.3 Architectural Properties

In this section, we briefly discuss architectural properties from existing literature. Automated analysis of the stability and evolution of the architecture of a large system has several challenges [19] due to manual intervention required in many cases. Software architecture [18, 101] refers to the fundamental structures of a software system, the guidelines of creating such structures, and the documentation of these structures. Each structure comprises of software elements, relations among them, properties of both elements and relations, and logic for the introduction and configuration of each element. Corazza et al. [40] define software architecture as partitions of classes into groups (based on close relations) for automatic clustering based on the structure. Czibula and Serban [55] also consider the base of software architecture as groups of program files such that files within a group are similar to one another and different from those in other groups. Figure 3.2 demonstrates the high-level view of ImageJ architecture.

Williams and Carver [126] present an empirical study about architectural properties and changes by analyzing about 130 research works. Please note that we have adopted most of the description of this section from their study. At a very high-level, architectures are described in terms of their logical (static) structure and their run-time (dynamic) structure [126]. The logical views include dependency relationships, layers, inheritance structure, module decomposition, and source structure abstractions as shown in Figure 3.3. The

runtime views include control flow processing, repository access, concurrent processes, component interaction, distributed components, and component deployment abstractions [126]. Logical changes affect system structure and consist of changes to systems, subsystems, modules, packages, classes, and relationships. Class hierarchy changes consist of modifications to inheritance views. Class signature changes describe alterations to system interfaces [126]. Change can be made to UML diagrams where each diagram type will signify the nature of changes made to it: class diagrams (i.e., add/delete attributes, change attribute, add/delete method, change method, add/delete relationship, change relationship, add/delete class, and change class), sequence diagrams and state charts [126]. A more general description includes changes to entities (i.e., classes and modules), relations and attributes [126]. Other types of architecture changes include: kidnapping, splitting, and relocating. Kidnapping is moving an entire module from one subsystem to another. Splitting involves dividing the functions of a module into two or more distinct modules. Relocating involves moving functionality from one module to another. All of these actions have a non-trivial impact on the maintenance and development of a system.

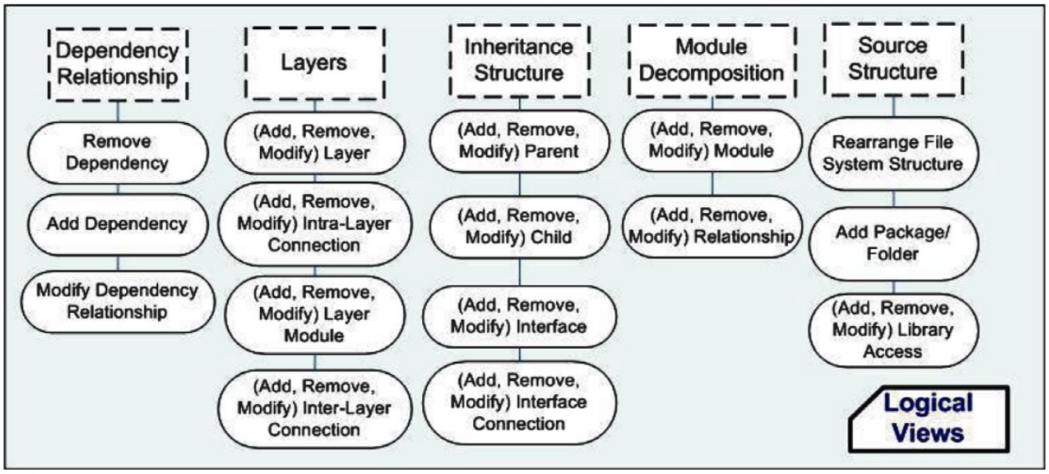

Figure 3.3: Logical view of static architecture. Adapted from [126].

In this work, we mainly focus our analysis on the logical and static architecture. Considering knowledge-base from the literature [18]-[35], initially, we consider the architectural changes as kidnapping, removing modules, merging or splitting of high-level packages/modules/components (as a folder) of a system, and merging or splitting of program files [126]. Addition of new files or modules is the continuation of existing structure, thus at the first step, we ignore this action. We will calculate the impact of those changes as the number of program files affected and the number of places affected. In this study, we also analyze recurrent activities related to those changes using attributes of the *Software Architecture Change Characterization Scheme* (SACCS). SACCS describes the change's motivation, type, size, impact on static, impact on dynamic properties and effect on requirements (functional and non-functional). The detailed change characteristics identify specific changes that must be made to the major architectural views. The knowledge base of these SACCS attributes are important to identify specific reasons and intention about why an architecture is chang-

Table 3.2: SACCS characteristics and their attributes. Adapted from [126].

SACCS Category	Attributes
Motivation	Enhancement, Defect
Source	Resource constraint, Law/government regulation, Policy, Stakeholder request
Criticality	Risk, Time, Cost, Safety, Requested
Developer experience	Minimal, Localized, Extensive
Granular effect	Functional/module, Subsystem- subset attributes: micro-architecture changes, Architectural– subset attributes: restructuring, refactoring, architecturally significant change, structural changes, macro-architecture changes System
Category	Corrective– subset attributes: intensive evolution; Perfective– Subset attributes: performative, groomative, reductive, enhansive, anticipative, evolutive, design evolution; Preventative; Adaptive– subset attributes: extensive evolution
Properties	Static, Dynamic
Features	Devices, Data access, Data transfer, System interface, User interface, Communication, Computation, Input/output
Quality attributes	Usability, Reliability, Functionality, Portability, Availability, Maintainability, Scalability, Efficiency– subset attributes: performance change
Logical	Dependency relationship, Layers, Module decomposition– subset attributes: coupling between modules; Source structure– subset attributes: header file changes; Inheritance structure– subset attributes: inheritance deviation
Runtime	Control flow processing; Concurrent processes; Distributed components ; Repository access; Component interaction; Component deployment

ing; we also utilize these attributes in our study to extract key-terms for architectural commits. Following this, we mainly consider the major concepts of architectural change analysis methodologies from the SACCS characteristics and attributes presented in Table 3.2 as deliberate changes. However, attributes of some of the SACCS are dependent on user-requirements, stakeholders, and developers. Consequently, in our study, we exclude those properties: Source, Criticality, Developer experience.

3.4 Architectural Change Analysis from the Development History

In this section, we will discuss in details about the architectural changes and their impact in various releases of the candidate projects. All of the steps of our study are demonstrated in Figure 3.4. The major steps of our analysis study are:

(a) Structural change detection

(b) Architectural commits detection and analysis

(c) Architectural quality metrics calculation and evaluation

In the discussion section, we will present bug-proneness and frequently changed sub-components along with their implications.

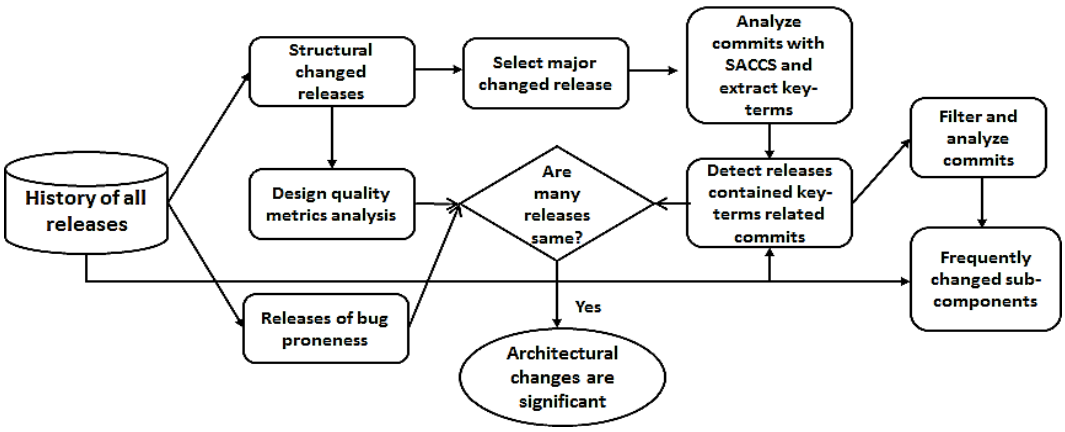

Figure 3.4: Overall steps of our analysis study

3.4.1 Structural Change Detection

We have written a program to create a dictionary consisting of components (with relative directories) and all the programs contained in each component for a release of the candidate systems. Our program identified high-level static architectural changes: kidnapping, splitting, relocating from the dictionary and AST (abstract syntax tree) parsing. The total number of releases changed of the systems are presented in Table 3.3. The releases that contain static architectural changes are presented in the Figures 3.5, 3.6, and 3.7 with the number of sub-components that are the candidates of the high-level changes. We also calculated the weights of the structural changes as the number of dependent files (partial static dependency defined in [78]) of the changed components similar to *FAN-IN* count described by Aversano et al. [16]. This step is similar to the retrospective analysis presented by Jazayeri [59]. So far, Galaxy undergoes 19 release changes; iPlant Collaborative (Discovery Environment) undergoes changes for 11 releases, whereas ImageJ undergoes changes for 8 releases. Among 19 changed releases of Galaxy we found 10 release contain substantial changes, for iPlant Collaborative (Discovery Environment), 6 have major changes and ImageJ has 3 major release changes. Although no rules-of-thumb are defined for measuring the impact of changes explicitly, Williams and Carver [126] report the impact of changes to one component, several components, and whole architecture as low, medium, and high respectively. In our context, we consider minor changes if the program dependency counts < 10 of the affected sub-components, and significant changes if dependency count $>= 10$.

From Table 3.5, we observed that the highest number of components (14) related to changes are found in Galaxy version 16.04. Among them, the sub-component of *pulsar* called the client, and actions sub-component of Galaxy lib are relocated, removed, or renamed to the next release which is used by many other

Table 3.3: Number of releases that changed structurally

Project	**# Changed Releases**	**#Kidnapping,Remove, Relocate**	**# Modules added**
Galaxy	19	14	5
iPlant Collaborative	11	9	2
ImageJ	8	4	4

important components. Pulsar is a library for an event driven framework to build scalable network program. In the latest version (17.01), we found this component is implemented in a single file-module (that means the micro-structures have been merged). The *action* component contains functionalities of tool configuration handling.

Furthermore, we studied all the components of all the changed releases. Our investigation found that these subcomponents are associated with: tools, tool shed, job handling, data, and web-request handling.

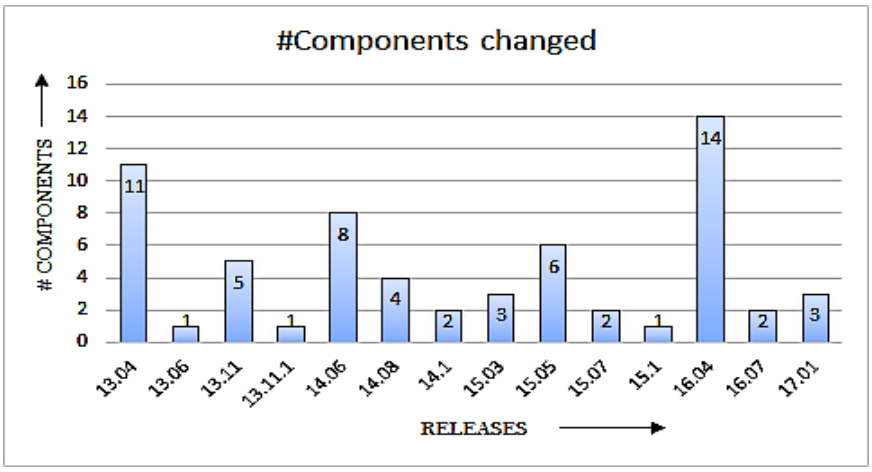

Figure 3.5: Components changed in Galaxy releases (from each previous adjacent release)

From the release 1.9.4 (ui) to the release 1.9.5 (ui) of iPlant Collaborative (Discovery Environment) 20 components are changed. Among them, the largest is the *events* sub-component of the *disk-resource* module. This component implements Google GWT library for resource and file event handling on the web.

Studying of all components of all the releases of iPlant Collaborative (Discovery Environment) related to change we observe most subcomponents are associated with: dynamic UI, client services, data-model, tools, and messaging.

Similarly, for ImageJ, in the most changed release 2.0.0-beta4, *pipesentity* and *core* module are the two largest subcomponents that are changed structurally. Pipesentity is used for handling different attributes during pipeline execution. Core/module is used for handling plugins and scripts, as well as workflows, which are directed acyclic graphs consisting of modules whose inputs and outputs are connected.

Finally, we also investigated all subcomponents related to the changes of all the releases of ImageJ. Most

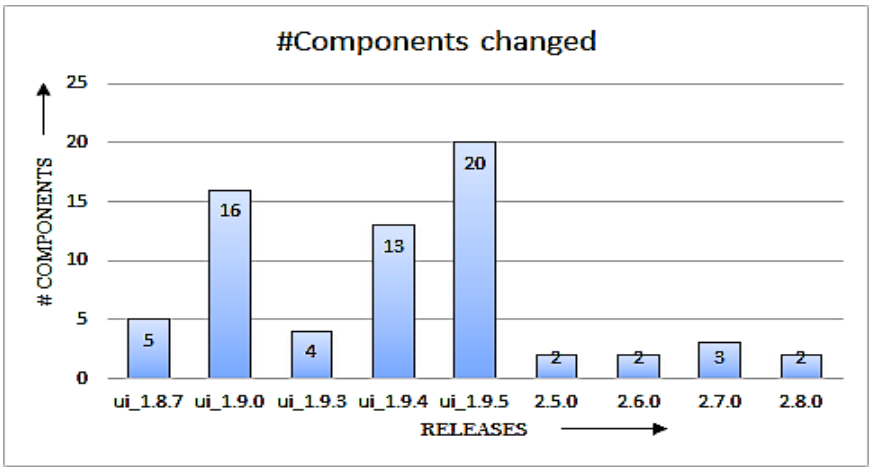

Figure 3.6: Components changed in iPlant Collaborative (Discovery Environment) releases

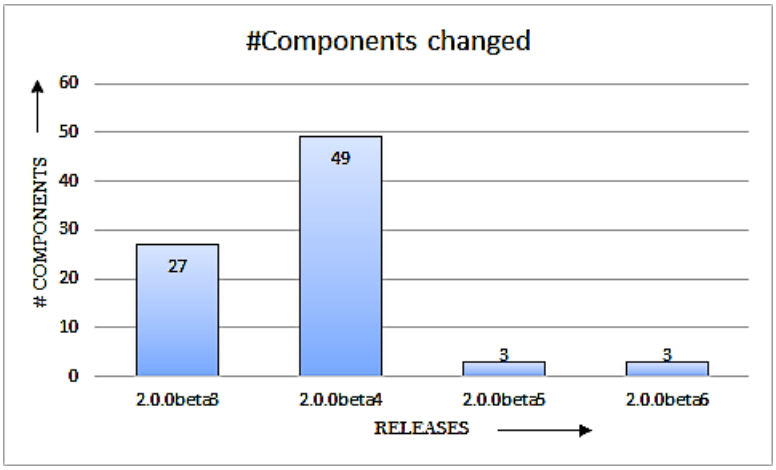

Figure 3.7: Components changed in ImageJ releases

of the sub-components are associated with workflows/pipelines, tools, plugins, data-handling, services, I/O, and UI. Therefore, this experiment provides us the information that static architectures of these systems are changing significantly from release to release due to unstable design. However, more detail information is necessary such as original reason or intention of the changes, and the sub-components of the systems that are related to deliberate changes. In the next section, we will present a technique for mining that information from commit history of the releases.

3.4.2 Architectural Commits Detection

Each release of the projects has commits that contain changes at the code level and short description about what has been changed. Therefore, from the commit messages, we can extract valuable knowledge-base about the intention of the changes. Experts and architect can study patterns of information contained in this history to understand why a piece of code evolved, interested piece of source code or to find and

Table 3.4: Sample commits that contain the intention of architectural changes

#	commit id	Text	Key-terms
1	..1ea4a4336441d87ba	**Migrate** imagej.ext **classes** to imagej base **package** (We leave only InstantiableException, temporarily, to avoid an issue with old versions of the ImageJ updater.) This is part of an effort to **make** the ImageJ package **structure simpler** and easier to understand	migrate classes package
2	..2ab8235d023f0297	**Add interface** for **objects** housed by a **UI** This UIComponent interface is shared between InputPanel and InputWidget,and could potentially be useful for other composition-style UI (i.e,"has a" UI component rather than "is" one) in the future.	add interface objects UI (not architectural change)
3	..81b6f6bcffec0f90c	Tweaks to the recent Tool Shed **API enhancements**, making them more RESTful.	API enhancements

debug faults. Even many attributes of the *Software Architecture Change Characterizing Scheme* could be extracted from the commit messages. We leverage the commit information to analyze architectural change activities as SACCS. However, it is a non-trivial task to manually analyze thousands of commits. Existing literature [39] illustrated the importance of reducing search space. We also focus on reducing search space and automatizing the commits analysis. To that end, after structural change detection, we select the most changed release from each project: 16.04 from Galaxy, ui_1.9.5 from iPlant Collaborative, and 2.0.0beta4 from ImageJ. These releases contain almost $\sim$ 1500 commits. We analyzed the commit messages and found that the developers present their intention in the commit messages; thus commit messages are the primary artifacts for the automatic extraction of the architectural change related insight and reason of changes without communicating with the stakeholders, developers, and requirement specification. After careful analysis, we found some interesting natural language patterns consisted of two to four words in a sentence (many of them are present in SACCS attributes) that express the intention of the developers about architectural changes; we call them as co-occurred terms. All co-occurred terms should be present in a sentence to express the architectural change. For example, the terms: make, structure, and simpler (contained in the sample #1 in Table 3.4) express that the structure of a module has been changed to make it more simple. Thus, we found $\sim$ 200 commits express architectural changes explicitly. Some samples of commit messages are presented in Table 3.4. We use the knowledge base of SACCS [126] attributes presented in Table 3.5. We also observe that some sentences do not mean architectural change even though the co-occurred terms exist along with some other terms (i.e., UI, visualization, display). Some co-occurred terms are ambiguous about explicit architectural change. We classified these co-occurred terms into three types: (i) explicitly represent architectural changes, (ii) implicit, and (iii) do not represent architectural change, are presented in Table 3.5.

Table 3.5: Representational co-occurred terms of architectural change (we have identified 120 co-occurred terms). Presence of these terms in a sentence are most likely to express architectural commits.

Co-occurred-terms	Non-terms	Implicit-terms
Design improvement	Visualizations decompose into	distributed object store free space
Decouple function	Repository dependency hierarchy	component review approval status
Enhance process	Enhance installing dependency	Processing state runner
Refactoring controller	Enhancement displaying	Performance improvement
run unless	Enhance setting to	Adjust logic module
Job state	Merge changes	Merge job changes
refactoring job preparation	Enhance display	Dependency resolution
new-style applications module	Merged in	Context dependency resolution

After a thorough investigation of the 1500 commits, we found that two to four terms are enough to represent architectural change activities in the commit messages for most of the cases. So far we extracted 100 co-occurred terms that may represent architectural change related activities.

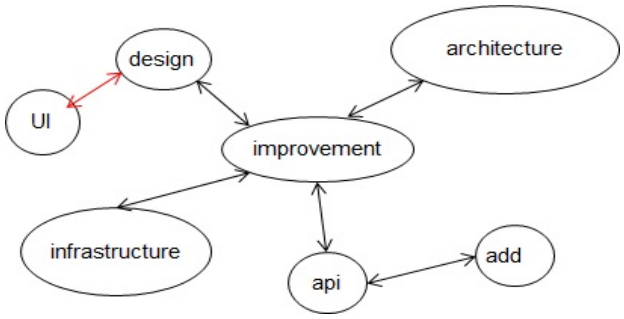

Figure 3.8: Key-term Graph related to architectural changes (red edge means not an architectural change)

Key-terms (according to the context) are being used in many techniques for mining significant information from software artifacts. For instance, in a published work [86], we utilized key-terms for predicting question quality in the developers' Q/A forum. We represent the key-terms as a graph (hence term-graph), where vertices correspond to terms, and edges correspond to co-occurrence between the two terms. Specifically, edges are drawn between vertices if the vertices co-occur within a sentence of a commit message. Termgraph is used [26] in information retrieval for detecting valuable information from natural texts. In this study, after collecting those co-occurred terms we first generate a termgraph using the co-occurred terms. A sample term-graph is shown in Figure 3.8. During scanning all the commits, a temporary term graph of each sentence of a commit message is generated. After that, we search a path in the temporary graph from each edge contained in the base graph. If the temporary graph contains a path then that sentence is the candidate sentence representing the architectural change commit. However, we see many cases where key-terms does

not exactly represent architectural changes such as #2 sample containing UI related terms as shown in Table 3.4. Consequently, we discard those sentences as decision-making sentence about the commit. We set the weight of each edge of the base graph corresponding to the number of the co-occurred terms that represent actual architectural change. First, an edge containing two terms is matched in the base graph. Then we check the weight of the edge and consider the sentence as a candidate if it contains a similar number of terms corresponding to the weight in the adjacent connected edges in the base graph.

Listing 3.1 Algorithm for detecting commits representing intentional architectural change

```
1  LC <— {} #List of architectural related commits
2  Collect co-occurred key-terms, Tkey
3  Gkey = developWeightedTermGraph(Tkey)
4  For commit C in Cm do
5      Sc = Sentences(C)
6      For sentence S in SC do
7              Ts <— Important terms in sentence S
8          Gts = developTermGraph(Ts)
9          If(valuable path from Gkey is in Gts)
10             LC <— add(C), exit from second For.
11     End For
12 End For
13 Return LC
```

Listing 3.1 represents the architectural commit messages detection algorithm. Here, weight is important to track the number of nodes of a path for filtering non-representative commits. Our technique detects 278 architectural commits from 26,284 Galaxy commits, 23 from 7,905 iPlant Collaborative (Discovery Environment) commits, and 127 from 9,800 ImageJ commits. Detected commits in major structural changed releases are presented in Table 3.6. Most of the static architectural changed releases contain these commits. We manually verified these detected commits and found relevancy about architectural changed related description. After manual analysis at the source code level, we found 57% to 77% of the detected commit messages reflect the deliberate architectural changes (average 66.33%). This is very promising as some commit messages are falsely identified due to the cross natural language effect, for example, *"Merge changes from other branches"* originally means merging two commits that may not be architectural components merging. The number of false positive can be reduced, but in that case, important commits may be skipped. Many architectural change commits those do not contain explicit natural language about the change may not be detected with this technique. In future, we will conduct more analysis for this issue considering different constraints. However, as the commits are detected by information in SACCS attributes, it should be said that the architecture of Genotyping and Phenotyping systems are changed due to enhancement, defect repair, restructuring, dependency changing, scalability, technology change, and so on. Furthermore, once these com-

Table 3.6: AC commits extracted by our technique

Galaxy (65%)		iPlant(DE)(57%)		ImageJ (77%)	
Version	**AC commits**	**Version**	**AC commits**	**Version**	**AC commits**
v13.02	19	/ui/1.9.0	8		
v13.04	39	/ui/1.9.4	3	v2.0.0-beta4	127
v13.08	20	/ui/1.9.5	5		
v14.06	69	2.5.0	3		
v14.08	36	2.6.0	4		
v14.10	35				
v15.01	22				
v15.05	19				
v16.04	19				

mits are detected from thousands of commits, project manager and software architects can conduct further analysis; for example, developers profile can be developed for those are more aware (or expert) about good architectural design.

3.5 Design Quality Metrics Calculation and Evaluation

In this section, we will discuss the design quality of the releases of the candidate systems. Aversano et al. [16] utilized Core Design Instability (CDI) and Core Calls Instability (CCI) for measuring the stability of the core module of a system architecture. However, recently, a number of metrics have been proposed to evaluate design quality of a complete system. To verify whether the logical/static changes reflect the architectural properties we evaluated various design quality metrics [21, 127, 112, 33, 34, 98]: Design Rule Hierarchy (DRH) depth [127, 33, 34], independence level (IL) [112], Package Cyclic Dependency (PCD) and Improper Inheritance [98]. To calculate those metrics we used state-of-the-art tools: SciTools (Understand) [3] and Titan [129]. SciTools produces dependency matrix (DSM) of all program files of a project, and Titan tool uses that matrix to calculate other design quality metrics. We will briefly discuss the Design Rule Hierarchy, Independence Level, and Package Cyclic Dependency as follows:

DRH (Design Rule Hierarchy): The unique feature of a Design Rule Hierarchy clustering is that files in the same layer are decoupled into modules that are mutually independent of each other. Here, independence means that changing or replacing one module will not affect other modules in the same layer. Design Rule Hierarchy is calculated through Augmented Constraint Network (ACN) and Design Structure Matrix (DSM)

[3] https://scitools.com/trial-download-3/

[21, 127] of a project. A Design Structure Matrix is a square matrix in which rows and columns are labeled with design dimensions where decisions are made; a marked cell emphasizes that the decision on the row depends on the column [127]. DSM can capture the concept of modules and design rules, as well as their decoupling effects. To detect design rules and independent modules within a software system, few prior works define a clustering algorithm, Design Rule Hierarchy (DRH) [127, 33, 34], which clusters a system's files into a hierarchical structure with n layers, where layer 1 contains the most influential files, typically interfaces or abstract classes that have many dependencies. Formally, a Design Rule Hierarchy is a directed acyclic graph (DAG) [127] where each vertex models a program file; each file is defined as a set of design decisions that should be made together. Edges in the graph model an assumption relation: an edge (u, v) models that the decision v assumes decision u. A change in the choice of u may cause a change in the choice for v. The layers within the Design Rule Hierarchy obey the following rules:

- Layer 0 is the set of program files that assume no other decisions.
- Layer i (i $\geq$ 1)is the set of all program files that assume at least one decision in level $i-1$ and assume no decisions at a layer higher than $i-1$. Within any layer, no program assumes any decisions in another program of the same layer. Hence, the program files within the same layer can be completed independently and in parallel.
- The highest layer is the set of independent modules. No decisions outside of these modules make assumption about any decisions within these modules

Independence Level (IL): Sethi et al. [112] proposed a metric called Independence Level (IL) to measure the portion of a system that can be decoupled into independent modules within the last layer of its Design Rule Hierarchy: the better modularized a system is, the larger the proportion of files in the last layer. Independence Level is calculated from DR (Design Rule) cluster. The modules in layer n, that is, the bottom layer of the Design Rule Hierarchy are truly Independent Modules because each can be improved or replaced without influencing any other parts of the system. Design rule theory suggests that more independent modules create more option values in a system.

Package Cyclic Dependency (PCD): Tosin et al. [98] illustrated that cyclic dependent components are more defect-prone than non-cyclic dependent components. Therefore, Package Cyclic Dependency (sometimes called cross-module dependency as well [84]) metric is used to predict defect of a system architecture. Package Cyclic Dependency is formally defined as:

- Given two packages P_1, P_2 of the DSM, there exists files f_1 in P_1 and f_2 in P_2. Now, let there are two files f_j in P_2 and f_i in P_1 exist, if $depend(f_1, f_j)$ and $depend(f_2, f_i)$, then these two packages are said to be created a Package Cycle (which is called a cyclic dependency between the packages).

Similarly, improper inheritances [84] also indicate the issue within a system structure. Above all, all of these metrics represent the architectural quality of a software, and detecting changes in their values represent

significant architectural changes. We select the releases of the candidate projects which have higher structural changes that are previously identified programmatically. The value of the design metrics of the releases of Galaxy, iPlant Collaborative, and ImageJ are presented in Figure 3.9, 3.10, and 3.11 respectively.

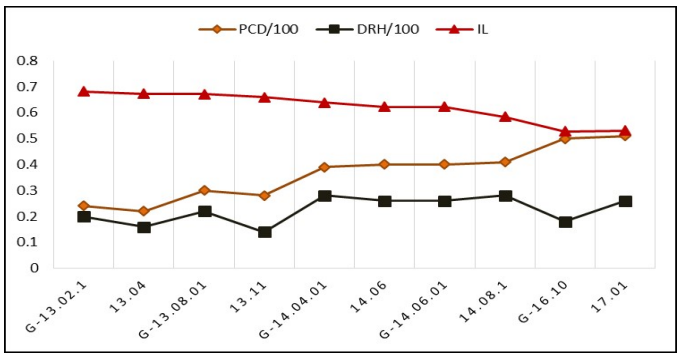

Figure 3.9: Architectural quality metrics in different releases of Galaxy. (Here, DRH is Design Rule Hierarchy, PCD is Package Cyclic Dependency, and IL is Independence Level).

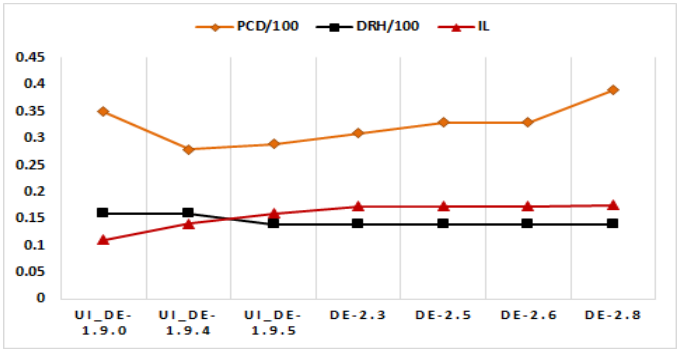

Figure 3.10: Architectural quality metrics in different releases of iPlant DE

From the chart in Figure 3.9, we observe that for most of the releases of Galaxy project, there are changes of Design Rule Hierarchy, Package Cyclic Dependency, and Independence Level measurement; iPlant Collaborative and ImageJ have almost steady value for most of the releases. From the charts, we notice that Package Cyclic Dependency of Galaxy and iPlant Collaborative grows with respect to the age of the projects that indicate the negativity of the design; by contrast Package Cyclic Dependency of ImageJ decreases at the latest releases. Initially, the value of the Independence Level of Galaxy was good but decreases in the latest releases (means negativity). On the other hand, Independence Level value was little for iPlant Collaborative and ImageJ, but slightly increases for iPlant Collaborative and significantly increases for ImageJ which represents the positivity of the design. However, Design Rule Hierarchy level of Galaxy and ImageJ are unstable for various releases. We also calculated the Improper Inheritance metric of all the releases and found significant value. Average values of IL of those releases for Galaxy, iPlant, and ImageJ are 0.621, 0.158 and 0.373 respectively. Average values of PCD for them are 32.4, 36.4 and 34 respectively. Although no value is defined as a standard threshold the higher values of IL and lower values of PCD indicate better

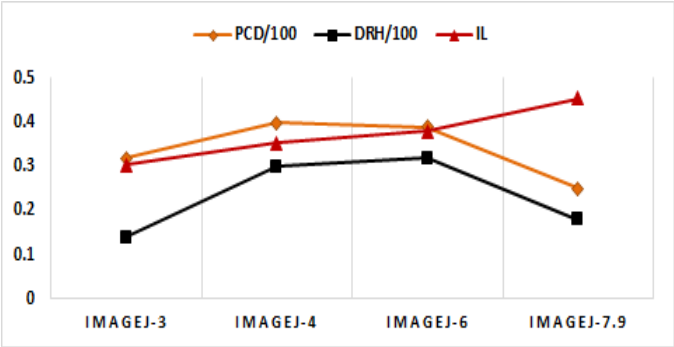

Figure 3.11: Architectural quality metrics in different releases of ImageJ

Table 3.7: Detected bug related commits in various releases

Galaxy		iPlant Collaborative(DE)		ImageJ	
Version	**Bug commits**	**Version**	**Bug commits**	**Version**	**Bug commits**
v13.04	96	/ui/1.9.0	43	v2.0.0-beta4	167
v13.08	68	/ui/1.9.4	3	v2.0.0-beta5	43
v13.11	11	/ui/1.9.5	15	rewritten-trunk	608
v14.06	116	v2.3.0	16		
v14.08	113	v2.5.0	37		
v14.10	105				
v15.01	158				
v15.05	220				
v16.04	312				

design of a system. Having said that, comparing these design quality metrics, it appears that design quality of ImageJ tends to be better than those of Galaxy and iPlant Collaborative. Notably, ImageJ is a smaller desktop-based system, and handle only localized image data. Another key thing to remember is that Galaxy and iPlant Collaborative (Discovery Environment) use multiple technologies and programming languages which might increase the complexity. This analysis result verifies that our detected logical structural changes of these projects are significantly related to architectural quality changes. All things considered, the design quality metrics reveal that these projects have significant architectural issues which affect maintenance and development efforts. Moreover, in terms of modularity, there is space for the improvement.

3.6 Case Study and Discussion

Analysis study presented in the previous sections identify the releases of the candidate systems that have been changed architecturally. Furthermore, we detect bug related commits in these releases using technique presented by Audris and Lawrence [85]. The bug related commits are presented in Table 3.7. This technique detects 1,997 bug related commits (among 5,230) for major changed releases of Galaxy, 115 commits from iPlant Collaborative, and 800 bugs from ImageJ. In the latest releases of Galaxy and iPlant Collaborative, bugs are increased, that means maintenance efforts are also increased. However, ImageJ is exceptional in this regard but bugs are still produced. A number of studies [85, 74] relate bug-proneness with bad design. Therefore, this also verifies that architectural issues create other issues. Above all, we analyzed architectural changes from the various perspectives: structural changes, commits that contain the attributes of *Software Architecture Change Characteristic Scheme*, various architectural quality metrics, and bugs-proneness. All of these analyses verify that existing Genotyping and Phenotyping analysis systems are unstable as they follow traditional architecture. This instability might be introduced with a change of requirements and new specifications, and developers and managers might have faced new challenges.

As a case study of our proposed architectural change commits detection technique presented in Section 3.4.2, we detected sub-components from the filtered commits from the associated code-base that are frequently changed due to SACCS attributes. Top 20 detected sub-components in various releases of Galaxy, iPlant Collaborative, and ImageJ are presented in Table 3.8. From the project properties, we also attempt to map the subcomponents with their major tasks in the respective projects which are also presented in Table 3.8. This information is also useful for the selection and enhancement of reusable components (from existing systems) for developing Genotyping and Phenotyping analysis system. The components (skipping UI and Test components) involved in the frequent architectural changes are related to Tools/ APP, Data-transfer/ IO, Data-processing, Plugins/3rd party tool integration, Threading, Job Scheduling, Pipelines/Workflows. Additionally, during sub-components analysis, we found the structure of some third party packages (i.e., LWR, pulser, gou, gin) used in the projects are significantly changed, which indicates that available technologies also prompt architectural changes. Therefore, more stable and feasible architecture is essential to be defined, especially focusing the dynamic nature of these subcomponents and technologies.

3.7 Related Work

Existing studies have focused on architectural evolution and issues during software development phases in various domains (business, industry, development tools, and so on). Nowadays, cloud based Genotyping and Phenotyping analysis systems are emerging, and have multi-complex structure. Despite this, a little effort had been given for the analysis of architectural issues in this domain.

Cobodon et al. [39] conducted an empirical study over 217 developers and reported that the most

Table 3.8: Frequent architecturally changed components and their main actions in the project

Rank	*Galaxy*	Major action	*iPlant*	Major action	*ImageJ*	Major action
1	api	app handle	lib	core actions+	display	visual
2	util	data op+	presenter	file viz	plugin	plugin handle
3	tools	tools handle	desktop	viz	assign	data op
4	galaxy	mvc op+	pq	database	tools	tool handle
5	tool_dependencies	handle dependent libraries	gou	go util	event	user action
6	recipe	handle dependent lib	gin	web routing	overlay	data op
7	dataset_collections	data op+	diskResource	data storage	module	I/O handle
8	parameters	tool param	fileViewers	viz	pivot	viz
9	controllers	web control	iplant	dependent subsystems	debug	logging
10	galaxy_install	associated tool	util	data op+	options	data stream
11	lwr_client	system job	de	pipeline handle+	core	processing service+
12	grids	tool repo	services	system service	util	data op +
13	runners	multi job	go-hostpool	host manage	restructure	data strcuture
14	collectl	job tracker	cells	viz	api	plugin
15	instrumenters	job plugin	deployed Components	viz	imglib	img operation
16	blue	viz	impl	web service	interactive	viz
17	tool_shed_unit_tests	Tool repo test	config	vendor config	io	I/O
18	jobs	job manage	base	collaboration	service	data manage service
19	workflow	workflow	certs	database	data	data op
20	base	toolbox	stub	viz	thread	job run

Here, "op"- operations, "viz" - visualization, "+"- more operations, "repo"-repository

developers want to investigate past development history to improve the next challenges. They report that, 58% developers are interested to investigate architectural evolution, but find it difficult to analyze as fully automatic technique is not yet developed. A number of studies focus on the analysis of software development history [46, 62, 65, 87, 39, 90] for extracting significant insight about software artifacts. Their analyzed artifacts are requirement specification, source code changes, and commit messages. In our study, we analyze commits, architecture, and sub-components from the history of open-source systems in this domain.

Nazar et al. [90] stated different techniques for automatic summarization of software artifacts from development history, however, no technique for summarizing architecture is reported by them. Fluri et al. [46] studied source code changes from history and found that change type patterns do describe development activities and affect the control flow, the exception flow, or change the API. Jim and Lee [62] studied the effect of IMPORT change from development history and found that import change is a significant factor in change prediction and change coupling analysis. Moreno et al. [87] proposed an automatic technique for release generation from change history. However, they did not include architectural change summary due to lack of automatic technique. Klepper et al. [65] propose a semi-automatic release note generation for different viewers (customer, project manager, tester, developer). This work did not provide any indication of architectural changes. Our proposed architectural commits detection technique can be used for generating automatic architectural change summary.

Williams and Carver [126] develop a Software architecture change characterizing scheme for the developers and maintainers to help decision making on potential changes can be made, given the development constraints and architectural complexity. Through a systematic review, they identified a number of important facts related to the software architecture change: (i) How are software architecture elements and relationships used when determining the effects of a software change (ii) How is the architecture affected by functional and non-functional changes to the system requirements (iii) How is the impact of architecture changes qualitatively assessed (iv) What types of architecture changes can be made to common architectural views. However, they have not provided any experiment with real world projects. We consider their facts during our analysis. In another study, Jamshidi et al. [58] identify research works on architectural evolution study, identify shortcomings in the architectural evolution study technique. Based on the lacking of existing techniques of architectural evolution they suggested to architectural evolution study from software repository using practical and systematic context and change impact of it. Our study explicitly follows the guidelines and insight about architectural change characteristics directed by these works.

A few studies focused on to the automatic software change and impact analysis from development artifacts. Behnamghader et al. [24] analyzed the impact of each commit (their dataset contains $\sim$ 20,000 commits) to the main module of a system as error proneness. They describe it as quality evolution analysis of a system. However, an automated technique is essential to reduce search space. In another study, Nejati et al. [91] proposed a technique for reducing search space to analyze change impact from requirement statement and design elements; where requirements and design information is essential as input into their technique. In our study, we propose a technique to reduce search space for analyzing architectural changes from the historical commit messages.

A number of studies [112, 59, 19, 13, 78] focused on to the architectural stability from the requirement specifications, pre-defined rules, design elements, and retrospective. We conducted our study with the development artifacts at the sub-component level explicitly using both retrospective and commits. Various research works [21, 127, 112, 33, 34, 98] focused on identifying the design quality metrics that directly represent the quality of an architecture. A few studies [85, 74] propose a technique for identifying bug related commits. We also utilized those techniques to identify architectural change impact from release to release of the existing Genotyping and Phenotyping analysis systems.

Finally, to the best of our knowledge, our study is the first that reports an extensive analysis of architectural changes and issues from the development history of the Genotyping and Phenotyping analysis systems.

3.8 Conclusion

In this study, we present an analysis of architectural changes from the development artifacts of the releases of Genotyping and Phenotyping analysis systems. At first, we identified and filtered out the most structural

changed releases and analyzed them. After that, we detected intentional commits about architectural changes using our proposed key-term-graph based technique. We observed that most of the releases filtered in the first step have such commits as well as bugs. Finally, we measured the design quality metrics of the releases and noticed the instability from release to release of the candidate systems. Additionally, we figured out the subcomponents frequently changed (most of them are different compared to usual systems) architecturally from the detected commits. Our experiment illustrates that existing Genotyping and Phenotyping analysis systems go through major architectural changes (hence unstable) mainly for tools/app, data-processing, plugins, workflows modules. That is to say, a new architectural model is required focusing the adaptable nature to the continuous changes which will reduce the maintenance and development efforts of the Genotyping and Phenotyping analysis systems. The knowledge-base of our analysis study is helpful for devising a stable architectural model and modularity design. The summary of frequently changed components and subcomponents also helps to develop use-cases in our next study.

Chapter 4

A Conceptual Reference Architecture

4.1 Introduction

Plant Genotyping and Phenotyping analyses involve numerous steps including physical plant sample collections, data curation, data conversion into different steps for generating users' expected end results, and making analysis results available to researchers and practitioners if needed. There are a number of challenges involved in automating the process of plant genotyping and phenotyping, e.g. reproducibility of experiments, high throughput processing of large amounts of data in various formats (e.g. structured, semi-structured and unstructured), identification of appropriate meta-data for the diverse uses of the data, collecting, abstracting, and loading data into easy accessible structures. Fortunately, several frameworks such as Galaxy [49], GenAp [68], iPlant Collaborative (or iPlant) [83], and LemnaTec [4] are already available to tackle many of the challenges. These technologies also attempt to tackle other problems, such as security, workflow management and accessibility of public datasets.

Architectural stability analysis presented in the previous chapter reveals that core components of the existing systems are changing significantly. Consequently, more analysis is required in terms of scenarios and micro-architectures in order to find issues and scopes of improvement in details. In this chapter, we investigate and analyze the architectures of the candidate frameworks by combining a reverse engineering process (using various tools and manual analysis) and SAAM [61], which is a scenario-based architecture analysis method. Both software architectures and scenarios are important tools for understanding a systems' behaviour. In our investigation, we attempt to understand the four candidate frameworks and determine their strengths and weaknesses by doing a comparative analysis with a set of scenarios using SAAM. According to the features described in Table 2.1, the abstract view of a such a system from the perspective of the users is presented in Figure 4.1. Here, users are mainly various computational biologists and agriculturists. But, developers face complex scenarios as there are working dependencies among various experts groups. For instance, image processing group develop algorithms, data-scientists use those algorithms for developing work-flows or pipelines, and the developers of the main cloud framework use those pipelines to integrate. Consequently, both users and developers confront with totally different contexts compared to other domains such as BI (business intelligence) systems [99]. Moreover, a large collection of data (e.g., crops field images) are required to be processed on daily basis for effective analysis. To process large data, Big Data frameworks

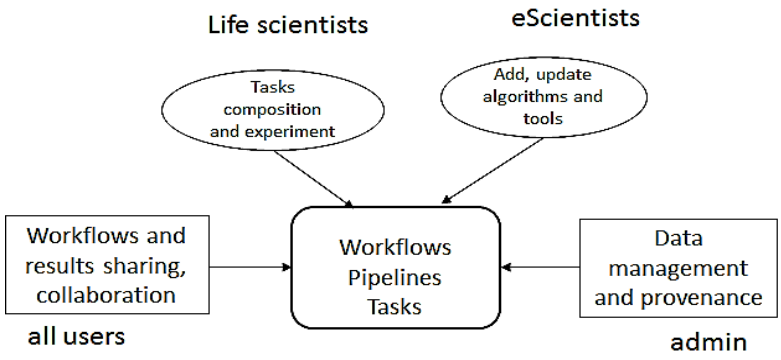

Figure 4.1: Abstract view (from a user) of a Phenotyping framework

are emerging that leverage the cluster processing. Among them, many data-scientists are using Spark [7], Hadoop [2], Pig [96], Google-Data-Flow [11]. Research on Big Data, design methodologies, and software architecture are evolving with the emergence of the latest technologies.

For an integrated solution, it is expected that a plant phenotyping and genotyping system should support (i) a data-centric model [11] capable of handling large and heterogeneous data with support for high throughput data processing, (ii) a software component model [70] supporting loosely coupled interactions among different independent components of the systems and (iii) an infrastructure model [9] defining services of the system across various machines in the cloud. However, our SAAM analysis reveals that none of the candidate architectures fully support these three models. For example, GenAp only partially supports the data-centric model and the infrastructure model, services of the Galaxy system are not separated into independent modules that are easily replaceable and pluggable without redeploying the whole system, and core subsystems of iPlant run on different machines that cause performance degradation. Consequently, the candidate systems are not flexible enough to support some advanced features needed for plant phenotyping, such as high-throughput image analysis, geo-spatial data analysis, scientific and intelligent workflow management, and virtual plant modeling and simulation.

Following the challenges of core components and scenarios, we propose a conceptual architectural model for cloud-based Genotyping and Phenotyping framework. The proposed architecture is a combination of data-centric model, component integration model, and infrastructure model. For developing the reference architecture we emphasized both inductive and deductive reasoning [99] which include bisecting existing systems from various dimensions and analyzing published papers in other big-data domains (this is a strong point of realizing a reference architecture). Some of the elements in the reference architecture are also adopted from various existing solutions. We map our proposed model with published works in various other domains. To the best of our knowledge, no cloud-based reference architecture exists in the subject domain that can handle the data centric, software component and infrastructure models for large volume data.

Along with discovering capabilities of the candidate frameworks, we also discuss our experience with SAAM. In this study. For example, while applying SAAM to four different architectures supporting similar

functionalities, we realized that when SAAM is applied to already built systems, product analysis is one of the important steps. It increases an evaluation team's insights and understanding of the systems.

In short, there are four major contributions of this study:

- Scenario based architectural analysis of four popular systems for plant genotyping and phenotyping
- Product analysis identified as an important/explicit step in SAAM
- The design of a conceptual reference architecture
- Mapping our proposed architectural model with the published works
- A proof of concept subsystem implementation based on the reference architecture

4.2 Related Work

Software architectural evaluation provides assurance to developers that their chosen architecture will meet both functional and non-functional quality requirements [105]. There have been different architectural evaluation methods proposed [38, 104] including surveys and comparison frameworks [69, 29, 105]. Among the different methods SAAM, proposed by Kazman et al. [61], is one of the most used methods. SAAM has been applied to numerous case studies: global information systems, air traffic control, and so on [6, 38]. SAAM identified different design problems in these systems and provided guidelines for fixing them.

Other methods [100] such as ATAM, ALMA, SAAMCS, SBAR, ALPSM, ESAAMI are based on SAAM and focused on a few of the specific quality attributes individually. ATAM is an architecture trade-off analysis method for availability, maintainability, security; SAAMCS is devised focusing complex scenarios in specific problems; ALMA is for architecture level modifiability analysis; ALPSM is the architecture level prediction of balancing maintainability; SBAR is scenario-based architecture reengineering for reliability and performance; ESAAMI is the extension of SAAM for reusable knowledge base in the domain; SACAM focused software architecture comparison method for selecting an architecture. In general, architectural evaluation is a central element during the entire software life cycle [97] and has been used for identifying architecturally significant requirements [23], for analyzing and evaluating architectures [38], for managing architectural knowledge [17], for architecture documentation [48], for architecture design [105], and for checking architecture/implementation conformance [108]. There are also recent works that use architectural evaluation methods for improving software product line management [97], for ensuring the security of a software system [63], understanding the sustainability of software architecture [69], and even early architecture evaluation for large-scale distributed systems [137]. Given that genotyping and phenotyping domains deal with terabytes of data, the architectures of the frameworks in these domains should be evaluated whether they meet not only the relevant quality attributes but also the challenges of Big Data and the specific requirements of the problem domain.

Unfortunately, to the best of our knowledge, there are no studies that evaluate the architectures or model a common reference architecture of Big Data frameworks in the subject domain. From the described

documentation, it appears that the candidate systems use MVC and three-tier model. However, Aniche et al. [15] illustrate that the mostly used model-view-controller (MVC) architecture has several issues with a cloud-based system. Models [71] are the major components of the system application that instrument the core tasks along with database handling. Controllers [71] are used as a way of interaction mechanism between the model and corresponding views (the interactive user interfaces). Some of the issues of MVC are called fat repositories (problematic when third party services are essential to handle), the promiscuous controller (critical when various categories of processing logic are required to handle), meddling service (complexities arise when there are data services other than only relational database services), and so no. It appears that these types of concerns are prevalent in this domain than other studied domains. However, in recent years, studies of some advanced reference architectures for other cross-domain systems that handle mixed and large data are available [42, 11, 9, 135, 75, 70, 99, 10]. Highlights of these studies include standardizing on a data-centric model, an infrastructure model, and a software component model that can adapt to the challenges of mixed data computation. We adopted them to introduce a common reference architecture for a cloud-based plant genotyping and phenotyping analysis system. Undoubtedly, these systems work on large and mixed data for diverse and varied sets of use cases. Demchenko et al. [42] discuss a number of aspects of why a data-centric model is necessary. Pääkkönen and Pakkala [99] define some reference architectures partially based on a data-centric model. Recently, Apache Beam (Google DataFlow) [11] has been developed based on a strict data model for processing massive-scale, unbounded, out-of-order data on the cloud.

Recently, a number of studies [42, 99, 70] propose new reference architectures for the modular systems for Big Data application. Among them, some applications follow a workflow based modularity architecture [70, 115] whereas others follow a layered based architecture [130, 67, 76]. Multilayer pattern is also followed by Tomominer [130], IBM Pairs [76] (for GIS data analysis), Bolstar [88], and so on. Bolster basically propose a straight-forward architecture and technology used in each layer for Big Data processing. Hierarchical modularity is followed by a National Security system [63] in which top level contains three modules– (i) Big Data application module, (ii) Framework provider module, and (iii) Cross-cutting module. Each of the modules has sub-modules. While the analytic module is the core development module, it is described as compact and no design rule is presented. Pääkkönen and Pakkala [99] extracted reference architecture of Facebook, Twitter, LinkedIn, Netflix, BlockMon. All of them are designed considering temporary storage of processed stream data in various stages and what kind of technology is suitable for each stage. A recently proposed framework, MDEF [53] is developed for redeployment flexibility of statistical models which implicitly followed data processing pattern. Infrastructure model is adopted by one of our candidate systems GenAP [68], as well as by HBase [132] and HVistrail [138]. Although HVistrail is for workflow systems, its main concern is how to orchestrate with high-performance computing environment from the application interface. In this thesis, we explicitly defined data-centric modularity orchestrating other modules in the system. We also include components to deal with the data interface, virtual plant modeling, and generic, semi-generic and customizable plugins integration on-the-fly. It is also necessary to describe a cloud infrastructure that fulfills

Table 4.1: Some Important Properties of the Candidate Frameworks

#	Property Text	Galaxy	iPlant	GenAp	LemnaTec
1	What kind of cloud API is used for allocating virtual resources?	CloudMan	Atmosphere	Galaxy Cluster API	Scanalyzer3D
2	What is the name of the cloud service provider?	Amazon Ec2	Atmosphere Server	Canada HPC	SUSE Linux
3	Does it provide Phenotype image analysis/ storage tools?	No	Yes, Bisque	No	Yes
4	Does it offer third party API support?	Yes, BioBlend	Agave API	MUGQIC	-
5	What kind of scheduling algorithm is used?	SLRUM	HTcondor	PBS scheduler	-
6	What is the underlying database?	SQLite	MongoDB	Spark SQL	SQLite
7	How large data files are stored?	No BigData yet	HPI Cluster (IROD, FUSE)	CVMFS	LemnaBase
8	How meta data is stored?	Simple XML	Sematic Web(iRODS)	ADAM(R)	CSV/SQL
9	Does it support scientific workflow?	Yes	Yes (Taverna engine)	Yes	Yes
10	Does it give access to publicly available datasets or analysis results?	Yes	Yes	Yes	Yes
11	Does it support MapReduce for distributed computing?	No	No	Yes through pipeline	No
12	Does it support applications through web portals?	Yes	Yes	Yes	No
13	Does it fully dependent on third party developers?	Yes	No	Yes	No
14	What kind of visualization tools it offers?	IGV+UCSC +IGB	CoGe	Same as Galaxy	LemnaMiner
15	Does it support independent 3rd party tool integration?	Command Line App	Command Line App	Command Line App	No
16	Does it support role based security management?	Through Admin Panel	Through Admin Panel	Through Admin Panel	No

the diverse requirements within the reference architecture. We adopt a heterogeneous cloud infrastructure [82, 9, 135, 75] for our conceptual architecture. Our analysis study is helpful for illustrating the challenges of developing a phenotyping system that handles Big Data, and developers will be able to implement such a system in a more advanced way by being more aware of the challenges.

4.3 Analysis Methodology

We adapted the architectural evaluation methodologies proposed by Kazman et al. [61, 60] and Roy et al [104] to evaluate the state-of-the-art platforms for genotyping and phenotyping. Our methodology combines different techniques in order to find out differences among the candidate platforms' architectures, including reverse engineering their software architectures, and scenario-based architectural and dynamic analysis. Reverse engineering software architectures [37, 122] alone do not provide a deep understanding of architectures. If the extracted architectures are refined with evaluation methods, they become more understandable to developers or architects [104]. Keeping this in mind, we extracted architectures of the candidate frameworks using tools and evaluated them using SAAM [61].

Major steps of our developed methodology for evaluating candidate architectures is described as follows:

Step 1. Product Analysis: In this step, we studied the candidate frameworks by using their executables, source code, online documentation, videos, asking questions of the product developers, and presenting

and demonstrating them to various practitioners. As an outcome of the step, we devised a question answering table (Table 4.1) to guide the comparative analysis of the candidate products. This table is helpful to learn about the products and their important differences quickly. The SAAM authors, Kazman et al. [60] used this step in their architectural analysis implicitly. In this study, we make this step explicit in our proposed methodology. Our understanding is that when SAAM is applied to an already implemented system, it is an effective step to execute during the architectural analysis. Moreover, this step is also helpful for developing various usage scenarios.

Step 2. Candidate Architecture Extraction: In this step, we extracted the architectures of candidate frameworks using various tools such as PyDev, PyCharm, ObjectAid and Eclipse since most of the apps are Python based. These tools generate UML class diagrams based on the class relationship within each module. However, class diagrams alone are not enough to extract an architecture since it contains other scripts (such as Linux shell scripts). Therefore, we also manually analyzed the source code ourselves to mine crucial attributes and used the built-in Python script ModuleGraph to identify dependencies among different source code modules. Finally, we obtained UML activity diagrams based on our analysis.

Step 3. Scenario Development: In this step, we elicited scenarios (or use cases). Each scenario is a representation of a particular quality attribute and expresses tasks [61] illustrating the kinds of activities the system must support and the kinds of anticipated changes to be made to the system over time. These scenarios represent tasks relevant to different roles such as end user/customer, marketing, system administrator, maintainer and developer.

Step 4. Individual Scenario Evaluation: For each scenario, we determine if it is direct or indirect. A direct scenario describes behaviour that can be supported without any modification in the system whereas an indirect scenario describes behaviour that is not supported by the system. In order to support an indirect scenario, the system must undergo some changes. In this study we discuss changes that are required in different modules/components of candidate systems to support the elicited indirect scenarios. This step allows us to understand the interactions between different modules of the candidate systems.

Step 5. Candidate Architecture Comparison: In this step, we compare different architectures by identifying the outstanding properties of the different architectures along with their strengths and weaknesses.

4.4 Product Analysis

As we mentioned earlier, in this step we first explore the features of the candidate frameworks. Our first candidate system, Galaxy, provides support for biomedical research. One of the important features of Galaxy is that it has History and Workflow options which are exclusive in their operations compared to other tools. Recently, Galaxy is hosted in a cloud environment too which helps a user to use it through the web. The second candidate system, GenAP [68] tried to design a more interactive database analysis framework over Galaxy. GenAp is a replication of Galaxy, but has a datahub management pipeline and individual server

Table 4.2: Scenario-based comparison of the candidate frameworks

Properties	#	Scenarios	Galaxy	iPlant	GenAp	LemnaTec
Usability	S1	A biologist or clinician wants to explore their data using tools available in the web-based platform.	Direct, well Structured	Direct, well structured	Direct, updated and well structured	Mainly Desktop
	S2	A computational biologist wants to use state-of-the-art pipelines to analyze their data without the burden of installing and maintaining all of the associated bioinformatics tools.	Direct, well Structured	Direct, well structured	complex initial setup	Complex
	S3	A sequencing centre pushes raw data into a Web based project created by a user to facilitate data access and analysis.	Indirect; Client, Scripts and Lib codes need to be changed	Indirect; Services, Atmosphere and Models code need to be updated	Partially supported; use unified data interface	Limited, not web
Flexibility	S4	A computational agriculturist wants to annotate an image (leaf, root) with paint and brush (if possible), and also wants to save and update the annotation.	Indirect; Client, Tools, Scripts and Lib codes need to update	Direct, Well Structured	No tools found	Support
	S5	A developer wants to plug-in his image analysis tool or pipeline for doing image analysis, registration and segmentation.	Indirect; codes of Config, Cron, Tools and Scripts need to be updated	Direct	Less supportive, difficult to integrate	Difficult
	S6	A developer wants to upload an intermediate processed work flow data in the system for further analysis.	Direct, Updated and Well Structured	Direct but needs to be improved, not user friendly	Direct, updated and well structured	Not Supported
Security	S7	A computational biologist wants to add a new RNA analysis tool HISAT in the existing system.	Direct but not well structured	Direct and user friendly	Direct but not well structured	Not Supported
	S8	A phenotypic researcher wants to work with a visual Map interface to automatically extract environmental information (weather, soil info, landscape etc.) of a certain region and integrate it with plant traits analysis.	Indirect, Client, Config, Lib and Scripts codes need to be worked out	Indirect, input parameter list needs to be updated for Migration and Model components	Not supported yet	Not supported yet
	S9	A user wants to share data from a project with another scientist to do joint data processing and exploration.	Secured	Secured	Secured	Not secured
Perform.	S10	A computational agriculturist wants to convert a file from one format to another supported and useful format within a reasonable amount of time.	Faster	Faster	Slower than Galaxy	–
Modifiability	S11	Get an update after execution of each of the steps of a workflow or image processing pipeline.	Indirect; Need to add in Client, Config, Lib and Workflow code design	Indirect; Models and Core functionality need to be updated	Source Code not analyzed	Not Supported yet
	S12	A user wants to modify any part of code base of an available workflow and wants to save it in his private history as a new version of the workflow.	Not Available; Need to modify Config, Lib and Workflow code design	Not available; Models, Core and Service code design need to change	Not available	Not available

instance creating facilities. For dedicated plant phenotyping, potential candidate frameworks include: iPlant [83], and LemnaTec [4] which are the third and fourth candidate systems respectively. iPlant supports a number of applications related to plant phenotyping. In addition, iPlant facilitates flexible app creation, image annotation, Google map service for the taken image, and HPC options, which are exclusive in their operations. LemnaTec is a desktop application which provides exclusive options for non-invasive plant data collection and life cycle observation of plant through imaging. This framework fails to provide support for collaborative research as it is a standalone desktop application.

As part of the product analysis steps, we did some dynamic analysis to learn the products' features and performance. For example, for evaluating the scenario S9 (Table 4.2), we consider the use case *Import data from 3rd party tool and convert the imported file into a different format.* For Galaxy and GenAp, we imported BED format genome file from UCSCB and converted it into GFF format. For iPlant, which does not provide automatic import, we put the BAM file into our workplace and converted it to BED format. Although both of them took < 10 minutes, Galaxy is more flexible than iPlant. However, GenAp is two times slower than Galaxy.

Along with exploring different features, we also develop a question asking framework (shown in Table 4.1) for summarizing different important features. The table helps us to figure out some important difference between the candidate frameworks quickly and help us in our scenario-based architectural analysis described in Section 4.6.

4.5 Scenario Development

As discussed in Section 4.3, we have carefully selected 12 scenarios (shown in the third column of Table 4.2). We selected the scenarios based on our product analyses and conversations with various bio-computation researchers, agriculturists, and image processing researchers who work with the agriculture group [1] at U of S. We also collected few scenarios from the GenAP website. These scenarios show important usage of the system while reflecting various quality attributes such as usability, flexibility, security, and performance by different stakeholders, such as computational agriculturists, computational biologists and developers. For example scenario *[S2:] A computational biologist wants to use state-of-the-art pipelines to analyze their data without the burden of installing and maintaining all of the associated bioinformatics tools* expresses usability of the system for data analysis by the computer biologist. On the other hand, the scenario *[S7:] A computational biologist wants to add a new RNA analysis tool HISAT in the existing system.* represents "requires to integrating" facility hence necessitating system flexibility.

[1] http://p2irc.usask.ca/

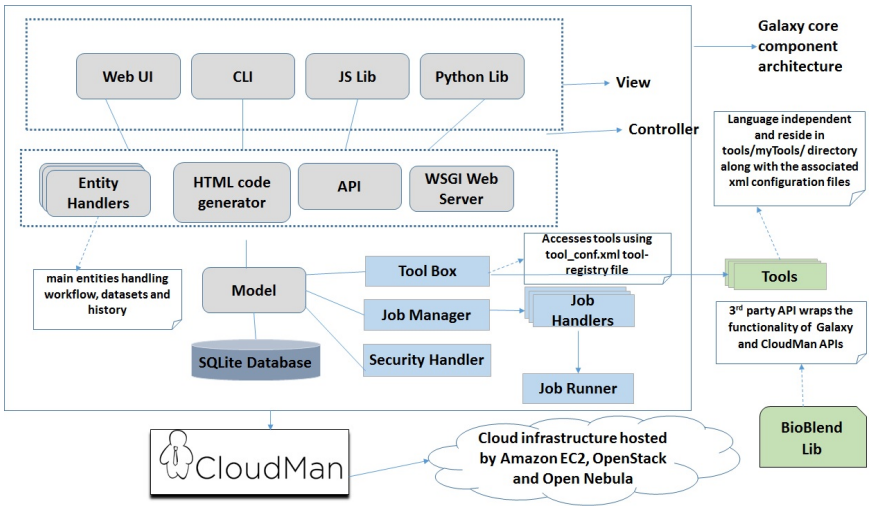

Figure 4.2: Galaxy Cloud Architecture

4.6 Extraction of Candidate Architectures and Scenario Analysis

In this section, we briefly describe our extracted architectures of candidate frameworks that handle plant genotyping and phenotyping along with our scenario-based evaluation findings.

4.6.1 Galaxy

The Galaxy [49] architecture consists of reusable software components as shown in Figure 4.2 (extracted as described in Section 4.3, Step 2). Galaxy infrastructure is developed using the Model View Controller pattern[2] as shown in Figure 4.2. The Model is responsible for adding and retrieving items from the relational database SQLite. Data entities handling workflow, datasets and history are mapped on to SQLite database tables. There is a controller for each of the main entities handling workflow, histories, and datasets. One of the important controllers is the API enriched with libraries for various programming languages helping the developers in building different applications and tools in the Galaxy platform, e.g. BioBlend [116] is developed using the Galaxy API. The Job Manager is responsible for executing jobs with the allocation of resources and keeping track of jobs by adding sequence number whereas the ToolBox collects and manages all types of tools in a language independent manner using the tool_conf.xml registry file. In addition, Galaxy allows third party visualization tools as plug-ins.

4.6.1.1 Individual Scenario Evaluation of Galaxy Architecture

In this step (discussed in Section 4.3), we executed the scenarios onto the extracted Galaxy architectural artifacts including the conceptual architecture (Figure 4.2) and on derived UML diagrams (e.g. Figure 4.3). From different scenarios discussed in Table 4.2 we noticed that scenarios S1,S2, S6, S7 and S10 are directly accessible from Galaxy architecture. The rest of the scenarios (such as S3, S4, S8, S11 and S12) are indirect

[2]http://www.laputan.org/pub/papers/POSA-MVC.pdf

scenarios that are not supported by Galaxy but can be integrated or updated with present architecture by modifying the system. For a complete understanding on what changes are required to add those scenarios with present architecture are stated in Table 4.2.

For scenario S3, our study shows that Galaxy does not support processing of heterogeneous data yet. It mainly works on a processed dataset which is already collected and manually inserted in a SQLite database without having any unified data input mechanism. To add this feature, it would be necessary to change Galaxy's code in Client, Scripts and Lib modules. Or a wrapper can be added to support a Unified Data Interface in Galaxy architecture. One of our candidate architectures, GenAP, which incorporated their architecture with the Galaxy public instance, has already added this feature in their work.

Similarly, scenario S4 addressing image processing behaviour is also not available in Galaxy. Galaxy is mainly focused on searching for a data in a flat file database (SQLite) with a user friendly query. Galaxy developers' focused on manually extracted (by a researcher or a biologist) genotype and phenotype information from different gene and chromosomes. Thus, they did not focus much on image processing pipelines or analysis. But Galaxy supports different options to adapt this behaviour. First, Galaxy developers need to build a tool which covers the functionality of S4. Then they can add the tool in Galaxy's Toolshed or like other available tools they can integrate the tool in their web version by changing the Galaxy source code in Client, Scripts and Lib modules. Or they can provide direct web-based support for this scenario. S5 relates to integration of image processing pipelines and as with S4, the main Galaxy instance still needs to work on it. Galaxy's code needs to be changed for Config, Cron and Tools modules. As Galaxy is an open source project, a group of developers have developed BioMina [1] based on Galaxy to support image analysis. But they do not do image registration or segmentation which is required for phenotyping analysis.

S6 focuses on workflow management and by analyzing the source code of Galaxy, we noticed that its workflow management system is well-structured. S11 and S12 also deal with workflow or the image processing pipeline. For S11, in Galaxy, all the submitted tasks and jobs of a user developed workflow run as a background process and the user gets an update once the whole process executes or fails. If it fails, Galaxy shows logs which is not user friendly. To make it more usable, it is would be good to inform the user after execution of each step as a flash message or as a notification. To do this feature needs to be updated in Galaxy's Client and Workflow code design. For scenario S12, from our study, it is observed that sometimes a researcher or a biologist needs to modify steps or the number of time a tool is executed in an existing fully developed workflow. To add this flexibility, Galaxy's workflow architecture and code should be updated.

In order to determine the coupling among different components in Galaxy, we worked on the direct scenario S7 where we look into how a tool can be added to Galaxy. Basically, same approaches discussed for S4 is followed to add the tool. From the activity diagram of Figure 4.3 we notice that tool integration involves many actions, such as tool shed configuration, data, datatype and metadata handling and loading utility library for tools from the toolbox. Each of the actions interact with various modules shown in Figure 4.4. For this scenario, it is clear that Galaxy system interchanges execution flow with numerous components,

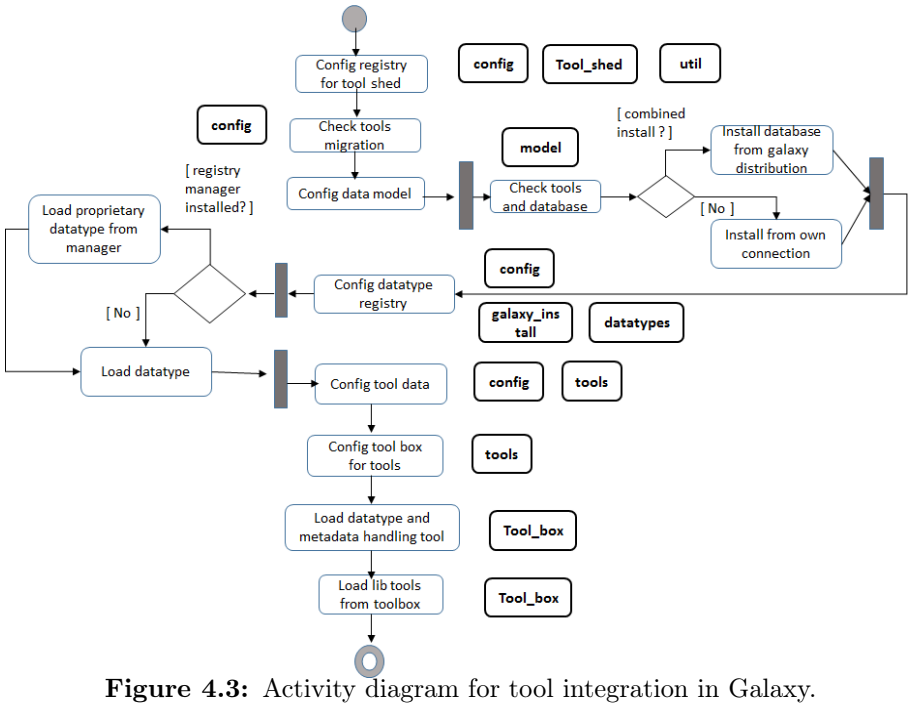

Figure 4.3: Activity diagram for tool integration in Galaxy.

and hence, we conclude it is a tightly coupled system.

S8 describes important parameters for phenotyping research and analysis. Although Galaxy has a phenotype association tool available, it does not support weather, climate or soil information for phenotyping analysis. To support this, Tools, Lib and Client modules, specially in phenotyping association tool, need to be updated.

In summary, our architectural analysis reveals that Galaxy's underlying architecture is modularized. The modules in the architecture are tightly coupled as different modules interact with each other in order to execute a scenario while maintaining moderate separation of concern. The public instance of Galaxy does not support image-based phenotyping as it does not offer image processing pipeline. This can be added through Toolbox with source code modification as discussed above. However, we have noticed that the image processing pipeline has been added in one of the galaxy servers, called Image Analysis and Processing Toolkit[3]. At present, one of the Galaxy instances, called BioMina, [1] supports image analysis.

4.6.2 iPlant Collaborative

The architecture of iPlant Collaborative [83] is a combination of independent applications as shown in Figure 4.5 and discussed as follows.

Atmosphere is a cloud service that allows users to launch their own virtual machines, whereas Discovery Environment (DE) is the primary web interface and platform to access the powerful computing, storage, and analysis application resources. DNA Subway makes high-level genome analysis broadly available to students

[3]http://cloudimaging.net.au

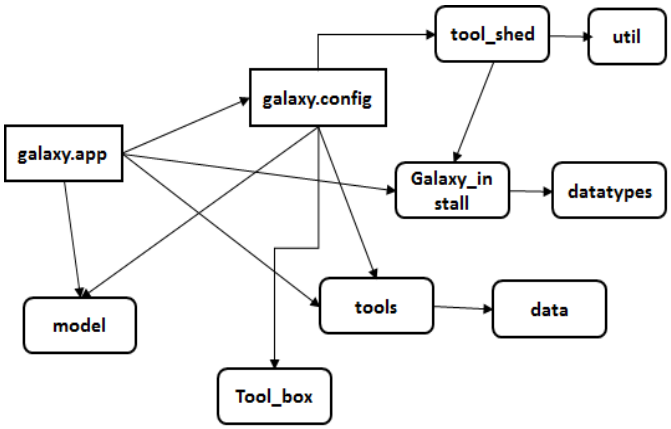

Figure 4.4: Component interaction diagram for tools integration in Galaxy

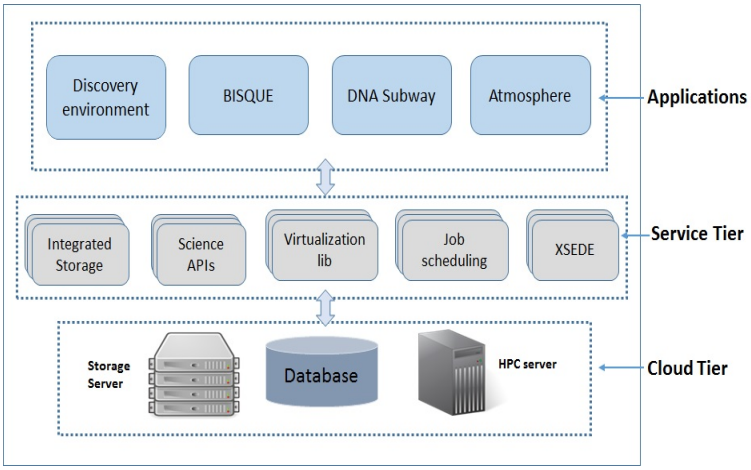

Figure 4.5: iPlant Collaborative architecture

and educators, and Bisque is the image analysis tool [73]. This platform preserves different scientific APIs links. If users need to use any of the scientific API, they can easily go to the link and access that. For job scheduling, they have used HTcondor and for creating cloud VM machines, they have used OpenStack. iPlant used two steps for storing data, e.g., a central database for smaller datasets for meta data and some private datasets and IRODS for storing images.

4.6.2.1 Individual Scenario Evaluation of iPlant Architecture

For iPlant, scenarios S1, S2, S4, S5, S6 and S7 are directly supported whereas the rest of the scenarios are indirect and can be added or updated in their architecture (see Table 4.2 for details).

For S3, as of Galaxy, iPlant does not support raw data input directly. IPlant developers worked on a processed dataset. To support this, the source code of Atmosphere, Models and Services modules need to be changed. Moreover, Discovery Environment source code needs to be redesigned.

Scenarios S6 and S7 focus on the workflow management and the tool integration process respectively. Workflow and tools integration for iPlant are implemented in DE, and it is a Java project. We use a free tool

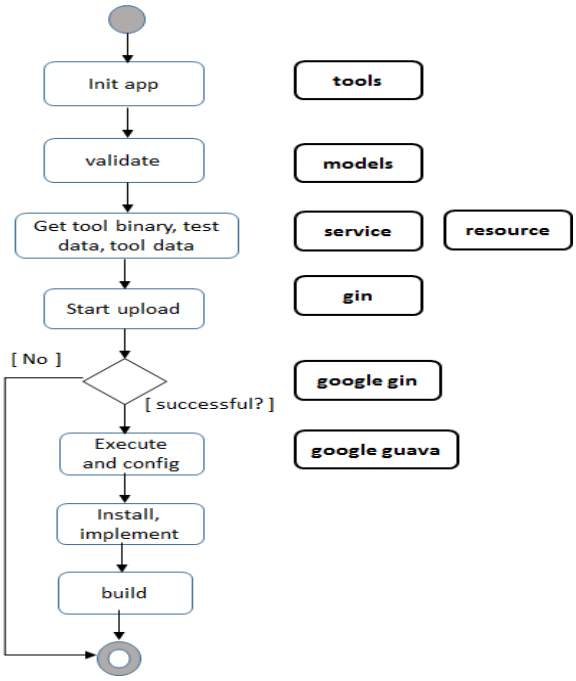

Figure 4.6: Activity diagram for tool integration in iPlant DE

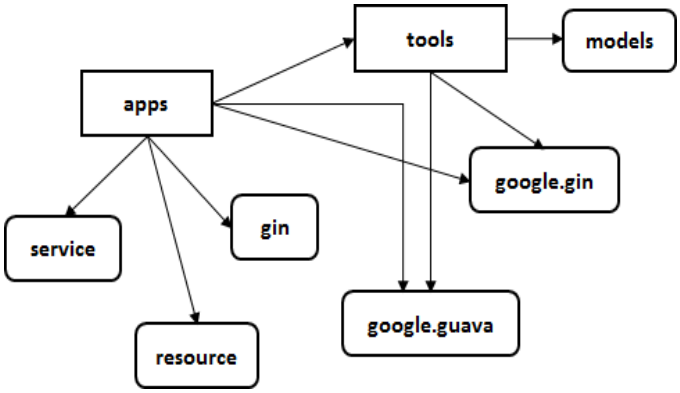

Figure 4.7: Interaction of components for tool integration in iPlant

ObjectAid to extract class dependency diagrams then manually draw the module interaction diagram (Figure 4.7) from the activity diagram (Figure 4.6). DE uses Google GIN API for handling some critical actions of app handling such as building and executing apps. The interaction diagram in Figure 4.7 only presents some abstract modules, but in the source code level, each of the modules has numerous sub-packages. However, from the source code analysis and class dependency diagram, we did not find any reference class or package of workflow implementation. It appears that the workflow is implemented as an app creation way (there is an option in DE for creating an app like workflow), and there is no special module for workflow handling in Discovery Environment. One user can upload her working workflow or create it in the discovery environment.

After a careful observation of the architecture and source code, we noticed that DE of iPlant has complex modules interaction, and thus imposes heavy coupling (even it is hard to find the reference code of workflow implementation). Apart from this, iPlant uses some third party APIs for managing tools and apps at the code

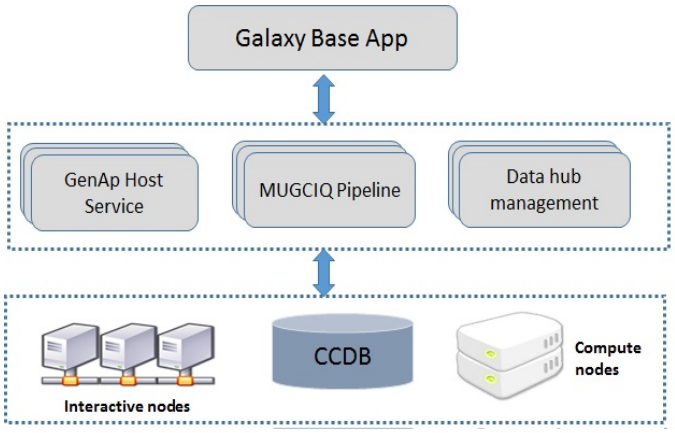

Figure 4.8: GenAP architecture

level. From source code analysis of DE, Atmosphere and Bisque, we observe that there is little interaction among these software systems. Although this is convenient as a loosely coupled system, some of them are developed in different technologies which makes it harder for a developer to modify source code and add new functionalities. Besides, various application servers need to be deployed for various technologies. There is no separate module for the workflow management which violates the separation of concern principle in a system design.

4.6.3 GenAp

GenAp [68] integrates its whole system with one of the Galaxy instances which is locally maintained with a focus on Big Data handling. GenAp offers its own MUGQIC pipeline tools which do not require any command-line knowledge. For each project creation, they communicate with Galaxy over HTTP which is a performance issue for GenAp. As discussed in Section 4.4, our dynamic analysis reveals that GenAp is much slower than other frameworks. They develop a central database and store smaller files and indexing data in that storage. At the same time, GenAp communicates with open source database for reference data. For their own large size files, they have stored them in a different large size database. And they used CVMFS for managing their file system. Comparatively, it is an easy architecture but is fully dependent on 3rd party developers[4].

4.6.3.1 Individual Scenario Evaluation of GenAp Architecture

The majority of the source code of GenAp is based on the Galaxy project, and workflow and tool integration are similar to Galaxy. Therefore, most of the scenarios are similar to Galaxy. However, GenAp has an additional data hub pipeline MUGQIC. So for the scenario S2 (details in Table 4.2), we evaluated Illumina pipeline processing for workflow execution. By following the similar steps to Galaxy and iPlant, we obtained an activity diagram for scenario S2 (Figure 4.9). Running Illumina pipelines involves a number of actions

[4]https://bitbucket.org/mugqic/mugqic_pipelines/src

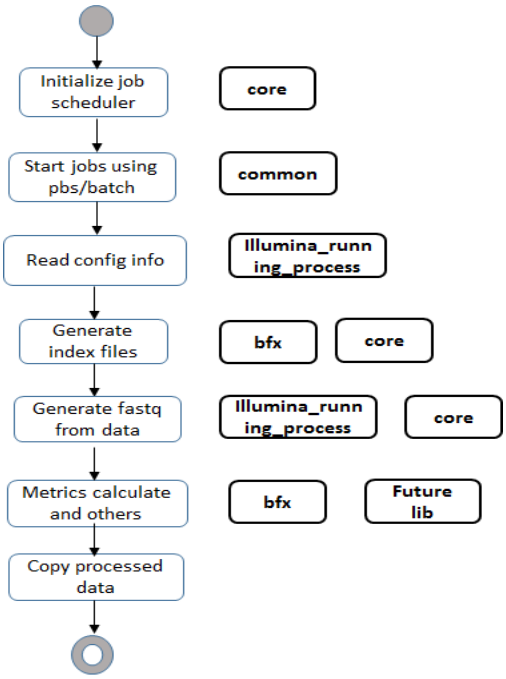

Figure 4.9: Activity diagram of Illumina pipeline service

such as generating index files, fastaq conversion, metrics calculation, even blast tool execution. In summary, from source code analysis and the activity diagram analysis of Illumina processing, we noticed that every step of all the pipelines are executed through a job scheduler; therefore this DataHub pipeline for the GenAp tool can be considered as a loosely coupled system. As GenAp is based on Galaxy, it also does not support the message queuing architecture.

4.6.4 LemnaTec Software

LemnaTec OS [4] is a well-known plant phenotyping system. An integrated set of modules provides rich functionality to control any hardware configuration and then record and analyze the resulting sets of data. LemnaTec's modular architecture is shown in Figure 4.10.

4.6.4.1 Scenario-based analysis of LemnaTec Architecture

As LemnaTec is commercial desktop software, its source code is not available, so we studied system documentation and explored only LemnaShare features and extracted the abstract architectural view. Like other analysis, we tried to evaluate the LemnaTec architecture based on our 12 scenarios (mostly provided by the LemnaTec development team) which is discussed in Table 4.2. We found scenarios S1, S2, and S4 are directly supported in the LemnaTec architectural modules. However, others are mostly absent or the developers did not provide explicit information about the features or the architectural design.

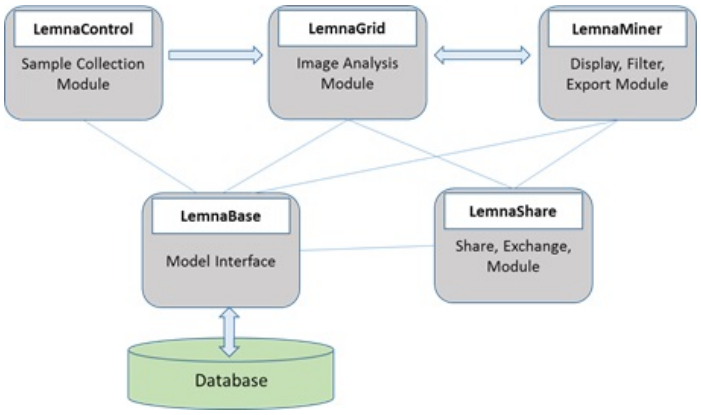

Figure 4.10: LemnaTec image analysis (base) plant phenotyping architecture

4.7 Comparison of Candidate Architectures

We compared the four candidate architectures based on our individual scenario evaluation and findings of the dynamic analysis. Our comparison result is shown in Table 4.2. In the table, we also showed the mapping of the scenarios to different quality attributes, such as Usability, Flexibility, Security and Performance. From the table, we see that iPlant supports most of the scenarios, except scenario S3, S8, S11 and S12, whereas Galaxy, GenAp and LemnaTec do not have support for many scenarios. If we consider S3 and all kinds of data interchange operations, none of them have full supportive components. We found that GenAP supports heterogeneous and open cloud infrastructure modules such as data-read write operations from users' data-machine (distributed storage too); others do not. For the performance scenario S9, we noticed that Galaxy and iPlant are much faster than GenAp since GenAp communicates with the Galaxy server via Http request. In addition, during our scenario analysis, we revealed that in all the four candidate architectures, components are tightly coupled as none of the architectures support message queuing for components' interactions. Also for high performance none of the architectures support image bundling or grouping using the MapReduce technology for high performance. Moreover, for scenario S4, we found that Galaxy and GenAp do not provide image analysis pipelines to measure out a plant leaf growth, whereas iPlant and LemnaTec have the support of this behavior.

We extensively analyzed the micro-architectures of the topmost frequently changed components with SciTools (Understand) [5], interaction and activity diagrams of some of the scenarios with the help of PyDev, PyCharm, ObjectAid and Eclipse, and ModuleGraph to identify the high-level relationship among components. This enables us to identify architectural issues, re-usable model, and scope for further improvement of the micro-architectural point of view. We reviewed a number-of-studies related to the architecture of cloud-based systems utilizing Big Data technologies. We were unable to determine any defined data flow model interface or message queuing module at the architecture level in any of the candidate architecture. However, all of them have components for mixed data model, and Infrastructure as service module (although

[5]https://scitools.com/trial-download-3/

Table 4.3: Comparison with Advanced Architecture

Tools	UDI	MDM	DCM	BDM	RCWM	PIM	HIOM	ISM
GenAp	∼	∼	⊙	⊕	⊙	⊙	∼	⊕
Galaxy	∼	∼	⊙	⊙	⊙	∼	⊙	∼
iPlant	∼	⊕	⊙	⊙	⊙	⊙	⊙	⊕

Here, UDI - Unified data interface, DCM - Data-centric model

DFM- Data flow model component, BDM- Big Data module

RCWM - Real-time collaborative workflow module, PIM - Plugin integration module

HIOM - Heterogeneous and open cloud infrastructure module

ISM - Infrastructure as service module.

⊕ - Full, ∼ - Variant/partial, ⊙ - Not exist or found

Galaxy has complex structure). Nonetheless, some other advanced components presented in Table 4.3 are not included in the architecture of all these tools. Therefore, this analysis leads us to define a reference architecture for advanced cloud-based plant Genotyping and Phenotyping systems.

4.8 The Conceptual Reference Architecture

Off-the-self reference architecture would be valuable for stakeholders, eScience developers, and researchers of plant genotyping and phenotyping system. Unfortunately, no common reference architecture (RA) is available that can serve as a design guideline considering the wide-spread support of cloud-based genotyping and phenotyping analysis. Moreover, in the previous sections, we have shown that the components of the systems are changing significantly, major components do not follow the well defined rule, and many scenarios are not supported by the candidate systems. However, we tried to extract design architectures of these systems from the development history and scenario evaluation described in previous sections. Adapting those use-cases, scenarios, and requirements of the P2IRC project of the University of Saskatchewan [5] we propose a conceptual reference architecture. A reference architecture can be of two [99] types - (i) High Level and (ii) Subset of System Functionality Level. Here we consider the high-level model and adopt reference architectures [70, 99] from cross-domain systems (those which handle heterogeneous and large data). Additionally, we observed that provision of only a software component model of such cloud-based systems is not sufficient to define a reference architecture; therefore, we include other important aspects of our proposal. The conceptual reference architecture diagram is presented in Figures 4.11 and 4.12. In fact, our conceptual architecture is a set of design guidelines and selection of solution patterns described as follows:

4.8.1 Data-centric Model

Recently, researchers have started to follow specific data-centric model [42, 11] for efficiently analyzing, sharing, and handling a varied set of massive data in the cloud system. Table 4.4 represents a number of applications (mainly scientific data analysis) for large-scale data processing and their architectural model. Likewise, Table 4.5 shows a number of frameworks for developing large-scale data analytic tool. Developing most of these applications and frameworks, implicitly, data-model plays an influential role. That is to say, defining a standard and uniform model for the management, processing, and interchange of data for a cloud system is one of the most important aspects of a reference architecture. Cloud-based Genotyping and Phenotyping analysis systems work with structured, unstructured and semi-structured data including - Genome dataset, Geospatial dataset, and Image dataset. These miscellaneous types of data are processed and analyzed by various workflows and pipelines. Recently, researchers have focused on specific data models (as shown in Table 4.5) for more effective and collaborative workspace. A common data-flow model is useful for implementing discoverable and shareable data processing logic as well. Thus, we recommend extracting and defining common data processing pattern for each category of data (such as images, Genome sequences) from features and specifications so that it can easily be instrumented to common workflows and pipelines in the system. Moreover, data can come as an input to the cloud-based system from heterogeneous sources (e.g., Genome data from GenBank [6]) in the open cloud. In our architecture, we recommend using a unified data interface (Figure 4.11) for interchanging input and processed data between the system and the user source. Various data-storage mechanisms should be defined for the cloud system based on the categories of data rather than a common storage for all categories of data.

4.8.2 Software Component Model

Recently, the modular architectural paradigm for designing a cloud system that handles large and heterogeneous data has been in a different direction [70, 99, 10]. Components and subcomponents of the system are updated, replaced or plugged in by the open community during the lifetime of the product. In order to develop a flexible and adaptable system with this dynamic context, the components must be independent unlike traditional MVC or three-tier architecture. Here independent means loosely coupled, smoothly replaceable and pluggable. Basically, all other major components should be orchestrated with and converge to data-centric model.

Workflow/pipeline module, data-centric module (DCM), and unified data-interface are at the heart of our proposed component model as shown in Figure 4.11. All other modules should just interact and utilize the actions provided by these core modules. DCM1, DCM2 are corresponding to image data processing, genome data processing engines and so on. Workflow module facilitates on-the-fly workflow and pipeline composition combining tasks instrumented in DCMs and contains workflow description language (WDL)

[6] ftp://ftp.ncbi.nlm.nih.gov/genbank

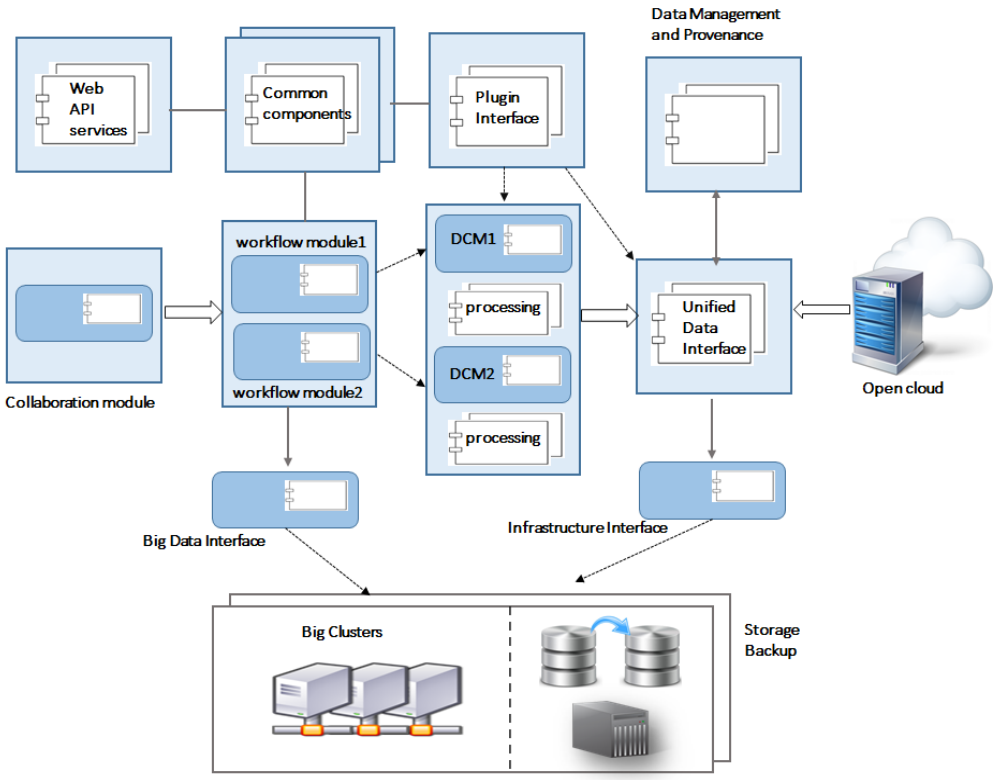

Figure 4.11: Software component model (DCM means data centric module). High-level component categorization.

module as well. Workflow module runs composed workflow's using DCMs to cluster through Big Data Interface module. Collaboration module works at the workflow module level. There should be no common gateway for interacting with UI. Individual modules (i.e., web services API, common components, plugin interfaces, Data provenance, WDL) interacts with web pages or mobile application (We have web-based plant modeling module, and remote sensor module in the common components part). This will facilitate easily replaceable and pluggable modules without redeploying the whole system. Apart from these, we also recommend providing API libraries to easily develop workflows, pipelines, and plugins integrable to the base system. A module that automatically integrates dependable libraries on-the-fly while uploading the plugins (shown in Figure 4.14) by the users would add more extensibility of the system. The last but not the least principle is that the multiple workflows, pipelines or data-analysis job execution should be handled by message-queuing [10] module (MQM) irrespective of whether they run in the base server or in the cloud-cluster.

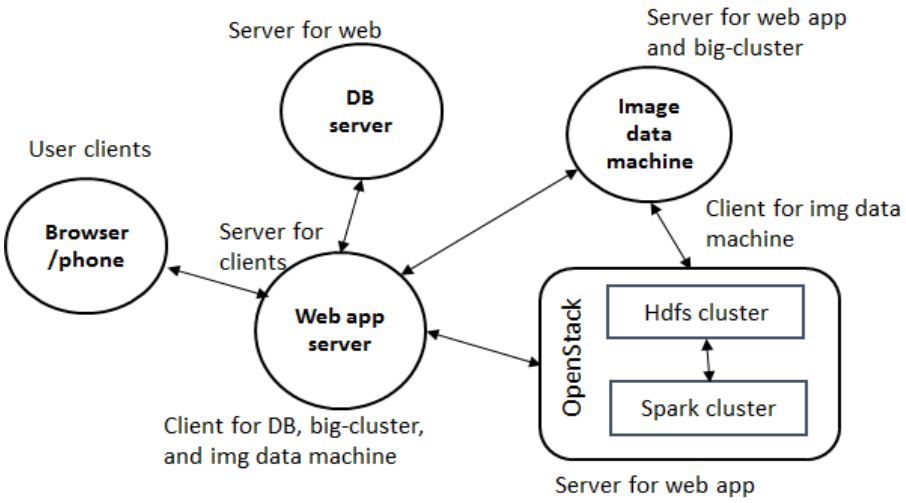

Figure 4.12: Heterogeneous cloud infrastructure model

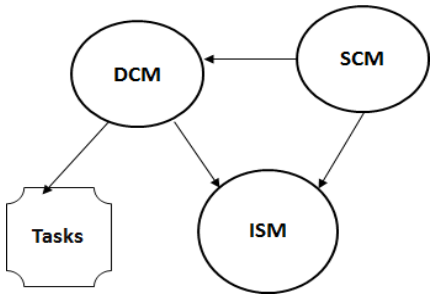

Figure 4.13: Influence of designing the models. SCM (software-component model) is influenced by ISM (infrastructure model) and DCM (data-centric model).

4.8.3 Infrastructure Model

Cloud-based Genotyping and Phenotyping analysis systems need to interact with various machines including open infrastructures in the cloud. Thus, the infrastructure model is an important part of designing such systems [135, 75] to firmly define which parts of the infrastructure serves which services, which parts are open to public access, which parts are restricted to access, and what are the communication protocols among various parts. We recommend a heterogeneous cloud infrastructure model for Cloud-based Genotyping and Phenotyping analysis systems presented in Figure 4.12. Here heterogeneous means different sections of the backbone machines are in different network model as well as there are diverse communication protocols among the parts. Heterogeneous cloud [75] consists of two critical entities: (i) Cloud, and (ii) Big Data cluster. Cloud part is composed of an application server and a data storage server, and Big Data cluster is a combination of distributed storage nodes and a number of connected machines (Hadoop and Spark ecosystem is widely used). Yu et al. [75] suggest using cloud and Big Data cluster on the same physical servers. However, analyzing the effectiveness of cloud-based Genotyping and Phenotyping analysis, we recommend using separate physical machines for them.

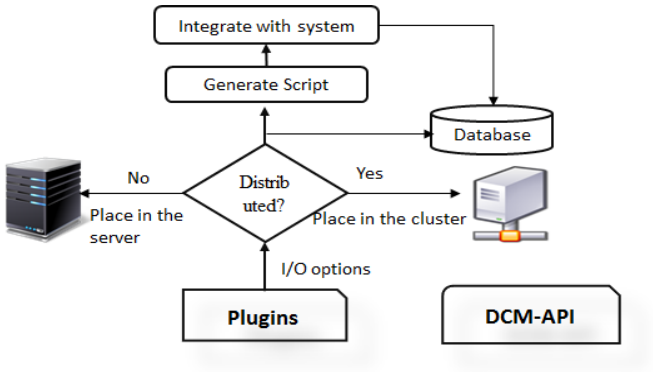

Figure 4.14: Plugin integration diagram for Genotyping and Phenotyping analysis system

Table 4.4: Unified frameworks for large data processing

Name	Base Model	Data	Framework
SparkSeq [125]	Hadoop−− >BAM	Genome	Spark
BIOSpark [64]	Hdfs−− >SFile	numeric, image	Spark
HIPI [118]	HDFS−− >HIB	Image	Hadoop
KIRA[139]	SEP−− >FITS	Astronomy(image)	Spark
StormCV[8]	Topology−− >stream	Image Stream	Storm
ICP [44]	P-Image−− >Big-Image	Image	Distributed
PCML[114]	Layer−− >location	Cartography	Usual parallelisation

From the above discussion of our proposed model, we observed that the models are influenced by each other. The influence diagram is shown in Figure 4.13. In the modulrarity model section, we will see how DCM is influenced by Big Data framework hence ISM. We have developed a prototype system following the proposed architectural model.

4.8.4 Mapping With Existing Solutions

As a validation of our proposed model, we map with the published architectural model in various domains. Unified frameworks presented in Table 4.4 are based on modified data-model for large volume which can be implicitly defined as unified data interface. Among existing popular frameworks in the scientific analysis, SparkSeq [125] is based on Hadoop-BAM [92] data frameworks. Hadoop-BAM is created to solve the issue of map-reduce implementation and attempted to include all data formats in bioinformatics. A unified framework for large scale Geospatial data analysis, SpatialHadoop [45] added three more layers on top of Hadoop to drive efficient map-reduce based processing of GIS data. KIRA [139] is written using SEP library and FITS data model for analyzing the astronomical object. All of the evidence prompt that tool development in Big Data platforms requires different design rule and modularity models. Various architectural models for Big Data handling are shown in Table 4.5. HVistrail is similar to our Big Data interface and Infrastructure

Table 4.5: Architectures of various systems that use Big Data technology. Here, [*] represents Facebook, Twitter, LinkedIn, Netflix, BlockMon

Name	Architecture	Data Model	Framework
IQmulus [70]	Workflow based modularity	SpatialHadoop	Hadoop
TomoMiner [130]	Three layer pattern	cryo-ET	Distributed Computing
IABDT [67]	Five layers modularity	HIB	Spark & Hadoop
IBM PAIRS [76]	Three layer pattern	Index-remapping	Hadoop
GreyWulf [115]	Workflow based modules	Pan-STARRS PS1	Distributed data part.
HBase imag [132]	Infrastructure Model	Hdfs-Image	HBase
HVistrail [138]	Multitier Infrastructure	Image	NASA HEC
National Sec.[63]	Hierarchical Modularity	GIS,Video	Kafka, HBase
Bolstar [88]	Multilayers Model	Semantic data	Hadoop, Spark
MDEF [53]	Redeployment modularity	BI-Data	Big Data Tech.
Industrial[*] [99]	Data-staging Architecture	Stream data	Hadoop, Spark

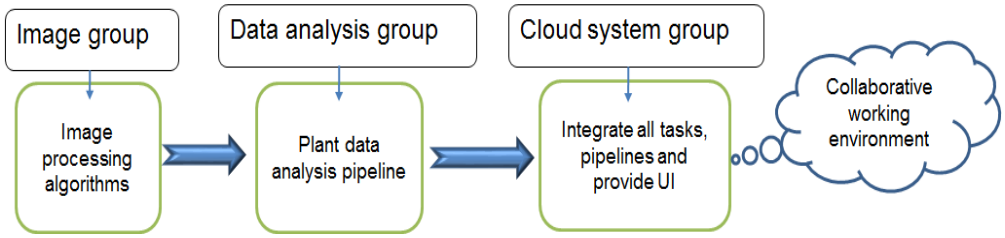

Figure 4.15: Collaborative working environment

interface. MDEF is based on statistical model redeployment facility which implicitly followed data-processing modularity. Apart from this, usual module decomposition technique is being followed by national security system [63], and stream data staging (how processed data is temporarily stored for intermediate stages) model is adopted by a number of industrial systems [99]. However, we could not find any solution that adopted well defined data-centric modularity model and real-time workflow collaboration on it explicitly. Although, code-base of our prototype system is not large enough we measured some architectural metrics. The DRH cluster is 8, IL is 0.906, PCD, and improper inheritance is 0 which mean the architectural model is promising. In the next section, we will present study on data-centric model development.

4.9 Prototype System

We have developed a subsystem for plant phenotyping as a proof of our conceptual architecture. The context of our prototype system is described in Figure 4.15. Since the studied candidate systems do not support high-throughput image analysis pipelines (HIAP), we designed a cloud-based system to support that. In

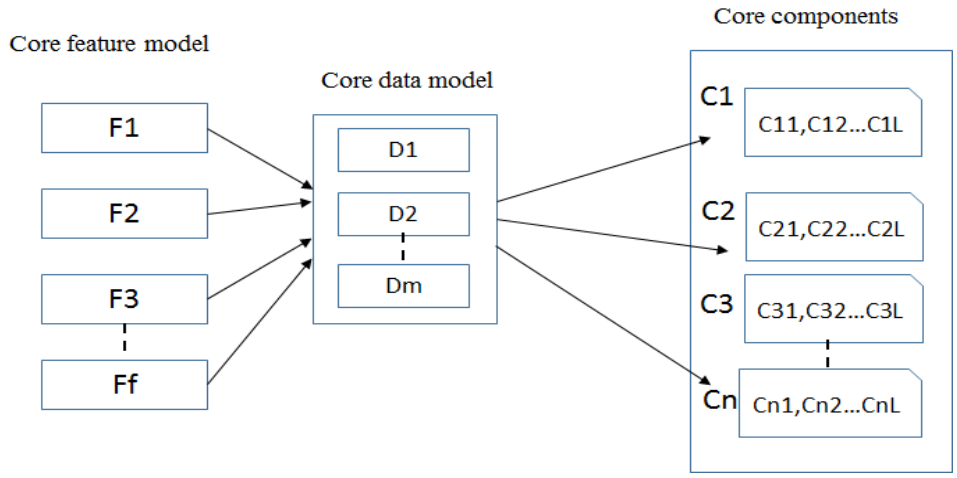

Figure 4.16: Data-centric modularisation of big data analytic tool. Here, F_1 to F_f are core features of a tool, D_1 to D_m are extracted data-centric models (consists of data-storage, I/O, operations, and patterns), C_1 to C_n is first order modular components of the tool, and C_{11} to C_{nL} are micro-level modular components of each first order component.

this section, we briefly discuss our prototype system that provides a few processing pipeline use cases in mapping with our conceptual architecture (full analysis is beyond the scope of this study). First, we set up heterogeneous cloud infrastructure defined in our architecture including a four-node Spark cluster. Then, we define a common data-flow model for image processing pipelines. We followed the strategy presented in Figure 4.16. This model leverages the development of HIAPs executed on Big Data cluster. Additionally, we develop an API library (based on pyspark) for developing high-throughput plugins for our cloud system following the defined data-model. Pre-built HIAPs in our subsystem inherit data-flow interface in the API. This data-model and API library also leverage the effort of scientific researchers to smoothly work with image analysis pipelines/workflows for plant phenotyping (that also ensures flexible collaboration among researchers). Furthermore, we design a unified data access module which can load large image data-set from any remote data machine and other cloud sources. We test the high-throughput image analysis pipeline, plugin integration, and remote data loading use-case with thousands of images with our small Spark cluster. Thus, our subsystem follows both the data-centric model and heterogeneous infrastructure model. Apart from these, we introduce an independent module for plant modeling and simulation based on WebGL. We compare the web-graphics module with traditional graphics library and found that web graphical library is more effective for plant simulation and modeling. In the cloud-based system, producing a plant model with a GL library is more flexible on the server side but less-interactive for usage and has a bandwidth bottleneck on the client side. The independent API, unified-data access module, plant simulation module, HIAPs module, and HIAP plugin uploading without re-deployment of the system indicate that the design of our system follows software component model for plant phenotyping. Moreover, our system is capable of MapReduce distributed processing of large data within the promising time frame. Distributed cluster processing is different than localized data processing. Components of these two types should be separate. It

is better if the cluster processing components are separately placed into master node of the cluster than the web-server. Only the components should be run through a secure protocol and with the Big Data interface. Similarly, in our prototype system, plugin integration module will handle localized and distributed version differently; a block diagram is shown in Figure 4.14. A more detail study on modularity following data processing model is presented in the next chapter with real world image processing tasks.

In summary, we can conclude that our conceptual architecture is a promising one considering the widespread requirements for cloud-based genotype and phenotype analysis. In future, we will update our conceptual architecture in more details at the functionality level, and test the efficacy with various evaluation techniques and upgraded system.

4.10 Conclusion

Plant genotyping and phenotyping analysis is required to handle large datasets and involves numerous steps, from physical plant sample collection to sharing analysis results and scientific workflows. Thus, it is a challenge to develop an efficient and cost effective integrated environment. In this study, we described our analysis of some existing popular tools for genotyping and phenotyping analysis using SAAM, examined weaknesses and strengths of those frameworks, and conducted a comparative analysis (shown in Table 4.2). Our comparative analysis determined that iPlant Collaborative is a strong tool that, unlike the others, provides support for image processing and, using Google Maps, for geo-spatial data. Unfortunately, it is not as flexible as Galaxy for workflow creation. GenAp is based on Galaxy with an extra DataHub server. LemnaTec is a desktop-based commercial tool dedicated to plant phenotyping. Since each of these tools has limitations, we introduced a conceptual reference architecture to better support the broad range of requirements, and developed a subsystem as a proof of concept. We believe that our study would be helpful to overcome the challenges of developing a cloud-based plant Genotyping and Phenotyping analysis system that deals with massive and mixed datasets. In the next chapter, we conduct a case study on micro-level modularity following the data-centric model and component model introduced in the reference architecture.

Chapter 5

Micro-level Modularity of Computation-intensive Programs in Big Data Platforms: A Case Study with Image Data

5.1 Introduction

Data-storage models, data-structures, data-operations, accessing and visualization of large data are complex to handle. Existing literature [53] suggests that significant effort is spent in developing data processing pipelines. Besides, a recent empirical study [107] reports that data engineers are facing great difficulties to work with Big Data platforms. In order to reduce the development efforts and provide better programming flexibility, a few studies attempted to develop more abstract and unified programming interfaces (especially in Bioinformatics and GIS research) [125, 64, 45, 8, 139, 118] as a layer on top of these platforms (e.g., Hadoop and Spark). However, most of them are still in the development phase (e.g., SparkSeq [125]), and some of them only implemented and tested a few specific tasks within a certain domain. Although, a few works implemented large scale image processing tasks [8, 139, 118, 67], they did not provide an analysis study about the underlying challenges and solutions of using these platforms for real world image processing pipelines. Moreover, common unified frameworks are not readily available to implement reproducible image processing pipelines covering a wide area with Big Data platforms. A few researchers [130, 124] have attempted to tune cluster resources for performance optimization of the tasks. Nevertheless, resource enhancement may not provide a feasible solution even with the availability of enough computing power. Therefore, interactive large-scale data-analysis with a Big-Data platform is still a challenging task for the programmers and developers.

Modularization is an important paradigm in software design which provides special program constructs, such as shared data structures or abstract and unified frameworks. Modularization is the action of *"decomposing a system into modules"* [34]. Moreover, modularization is essential for scalable and interactive application development with Big Data platforms [134, 53, 70]. Our focus is on modularising interactive, data-intensive programs so that they operate effectively on map-reduce frameworks in order to support reusability, reproducibility, and customization. Splitting tasks considering large scale data processing and computation logic reusability may have adverse effects when running on Big Data platforms. Well defined rule [128] is

available for modularizing a system into a group of related tasks such as model-view-controller is a high-level pattern which suggests to modularising a software considering data handling (model), business logic on the data (controller), and visualization of the processed result (view). However, a systematic model should be defined for further modularizing a complex and computation intensive task (e.g., extracting texts from a picture), hence we call micro-level modularity. Micro-level modularity has been shown to work successfully on map-reduce frameworks for a number of applications, including machine learning [79, 72] and graph data processing [131, 50]. We are also motivated to support developers of Big Data analytic tools. By separating tasks into further independent micro-components based on data-processing patterns, we hope to develop a unified programming interface that will provide the flexibility for accelerating the development of interactive, re-usable Big Data analytics tools.

Although Big Data platforms hide the complexity of distributed computing, they provide a limited number of methods (e.g., *map, filter, reduce*) for data parallel operations. Adding an extra data processing step with those methods could increase the computation and memory overhead in a significant way. For example, Smith and Albarghouthi [117] discuss the challenge of partitioning computation with data-parallel operators (*map, filter, reduce*). Due to the complexity of partitioning, they avoid optimization of their technique. In order to reduce working efforts, a few authors [134, 53] focused on developing frameworks for running and re-deploying modular jobs and a statistical model provided by the users. Unfortunately, none of them conducted an extensive study on any effective techniques for modularity to examine the impact of modularity on computation-intensive tasks. Moreover, the mechanism of controlling data-flow among intermediate steps is really important for reproducible computation of large scale data. All things considered, we propose a modularity model and observe the behaviors of different applications in terms of modularization. Overall, in our work, we mainly focus on two research questions:

RQ1. How to modularize data and computation-intensive programs to provide a unified abstract framework for developing interactive tools?

RQ2. How does splitting up of run time job-data and processing logic affect the performance of computation-intensive tasks in map-reduce platforms?

In order to answer **RQ1**, we analyzed various open source image processing tools and state-of-the-art image processing techniques that cover a wide range of tasks. We look into the programming models and data-types that are produced during a full image processing task (some of them are presented in Table 5.3). Then, we categorized the image operations and defined a data processing pattern that is fruitful for modularizing the tasks with Big Data platform. After that, we proposed a micro-level modularity model consisting of four major data-parallel modules each having three core layers (the second layer controls parallel data-flow). For answering **RQ2**, we implemented six image processing applications following the proposed modular model. Then we experimented with both the compact and modularised version with various datasets in a Spark cluster. From the experiment, we found that the task modularization affects system's performance and flexibility of pipeline development. Performance varies case by case with some tasks improving, some

decreasing, and others unaffected. For all the cases, it opens up the capability of flexible implementation with data-parallel components. Notably, we also identified the challenges of image processing with data-parallel frameworks from our experimentation. In summary, our case study provides a modularization technique and helpful knowledge-base for interactive tools developers for large scale image processing. Our defined data-pattern and modularity model can be used as a design pattern and design rule in this domain.

The rest of the study is organized as follows. Section 5.2 describes the process of extracting data-processing patterns. Section 5.3 presents our proposed modular model. Section 5.4 provides our experimental results. Section 5.5 provides discussion and some useful insights from the lesson learned. Section 5.6 describes related work. Finally, section 5.7 presents the conclusion and future work of the study.

5.2 Modularising Data-intensive Tasks

In our study we focused on image data since a framework that supports various image processing pipelines is not readily available. On the contrary, few abstract frameworks [139, 118, 44] are being developed for a few specific image processing applications and most of the Big Data frameworks support workflows for text data processing [45, 64]. We conduct our case study following three major strategies: (i) Background and Contextual Analysis, (ii) Data-processing Pattern Extraction, and (iii) Transformation to Data-parallel Components.

5.2.1 Background and Contextual Analysis

To develop a unified framework, understanding the context is essential. To that end, literature review and analysis of various architectures [106, 115, 70, 67], frameworks, tools, techniques, and open-source APIs in the scientific data analysis are essential to determine the exact support needed for the data scientist. Analyzing the recent development strategy of analytic tools for large scale data, we notice that some of the developed applications follow a workflow based modularity architecture [70, 115], whereas others follow a layered architecture [130, 67, 76]. In the workflow based modularity architecture, applications are designed using a special data model which is much different than the traditional model view controller model. For example, the architecture of IQmulus [70], a GIS data processing system, is heavily dependent on data-analysis workflows. High-level components, job manager, processing services etc. are designed focusing on the on-the-fly workflow compositions. Still, users need to learn a considerable amount of script for composing workflows for GIS. Similarly, GrayWulf [115] handles two types of workflows: (i) one is for data manager, and (ii) another is for end-users. The architectural model is based on these workflows composition. However, using GrayWulf, a smaller amount of processed result can be shared and retrieved in the cloud. Another application for image analysis, IABDT [67] followed multi-layer architecture and primarily used HadoopImageBundle (HIB) for performing basic operations on image data. In a recent study, Roy et al. [106] focus on data-centric component development for an application that supports large scale data analysis. Besides, most of

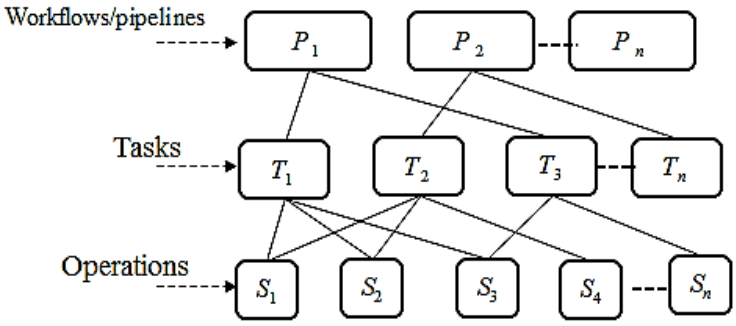

Figure 5.1: Reproducible workflow composition technique (tasks and operations are reused, algorithms are customized).

the unified frameworks to support applications development as mentioned above followed specialized data-models for large scale data processing with distributed clusters. Among existing popular frameworks in the scientific analysis, SparkSeq is based on Hadoop-BAM [92] data frameworks. Hadoop-BAM is created to solve the issue of map-reduce implementation and attempted to include all data formats in bioinformatics. A unified framework for large scale Geospatial data analysis, SpatialHadoop [45] added three more layers on top of Hadoop to drive efficient map-reduce based processing of GIS data. KIRA [139] is written using SEP library and FITS data model for analyzing astronomical objects. All of the evidence suggests that tool development in Big Data platforms requires different design rules and modularity models. Yet, the common obvious advantages of modularization [34] in software development are: (i) Easier to Debug and to Detect Problems, (ii) More Reusable Code, (iii) Better Readability, and (iv) Enhanced Reliability. Debugging time is lengthy during the development of Big Data analytic tools. Most of the time, ultimate problems cannot be detected until the application is run on a live cluster with the full set of data. In summary, for large-scale data analysis the following trends are emerging:

(i) Suitable architectural model [106, 70, 67],

(ii) Work-flow processing and management [95, 77, 70],

(iii) Data-pipelines [22],

(iv) Data-flow management [11, 96, 103],

(v) Data-centric decoupling of programs [50, 131],

(vi) Efficient data-storage model [45, 76], and

(vii) Intelligent modularization [134, 53].

The central objective of all of these paradigms is to make scientific computation reproducible [111] with minimal technical knowledge. Figure 5.1 demonstrates how reproducible workflows/pipelines are constructed. However, large scale image processing domain requires more focus on all of the above-mentioned directions.

Furthermore, in order to understand and develop a knowledge-base, we look into the properties of various open source image processing tools [110, 52, 93, 66, 73, 139, 118, 8] which are presented in Table 5.1.

Analyzing source code of the open-source tools facilitate us more intuitive insight about the implementation of real world image analysis applications, programming models, I/O operations, and data entities that

Table 5.1: Properties of various open-source APIs for image processing tasks

Tool name	Major Focus	Type	I/O	Technology	Limitation
ImageJ	Common tasks, plugins	Desktop(threading)	Local	Java	Localized processing and limited I/O
HTPheno	Plant growth	Desktop	Local	Java	As ImageJ
Fiji	Medical	Desktop	Local	Java	As ImageJ
PlantCV	Plant image analysis	Desktop	Local, database	Python	As ImageJ
Bisque	Crops & biology	Web service api	web storage	Python, C++	Complex for cluster processing
Thunder	Few Crops tasks	Local, Distributed	Local, AWS, Google storage	Spark	Local processing of few tasks, limited I/O
KIRA	Basic image operations	Distributed	hdfs, hive, distributed storage	Spark	Facilitates object extraction
HIPI	Astronomy object extraction	Distributed	hdfs, hive	Hadoop	Few specific tasks, need to write parallel code from scratch
DEDIP	Agriculture	Master-slave server processing	Local	Java	High-learning for basic tasks, no various I/O
MiDas	Brain Images analysis	Batch processing	Web storage	Python, Science Grid	High-learning for basic tasks
ImageHarvest	Plant image processing	Grid Processing	Database, Local	Python	High-learning for few tasks
StormCV	Real Time video analysis	Distributed	Camera stream	Java, ApacheStorm	Limited I/O

analysts, researchers, or end-users might re-use later. We also observe that in image processing tools the following attributes are influential (some of them are identified by Heit et al. [53] in data mining as well): (i) Image pipeline composition, reuse and management, (ii) Image processing workflow modeling language, (iii) Image storage service, (iv) Collaboration between data scientists, (v) Deployment and third party service communication, (vi) Scalability, and (vii) Plugin development and integration. Another key thing to remember is that data structure and computation model of images are complex and diverse [110]. Images (I) consist of different data units (8-bit, 16-bit and 32-bit), formats (TIFF, GIF, JPEG, BMP, DICOM, FITS), or dimensions/channels (2D, 3D, 3-channels and so on). Additionally, we found that many algorithms are operated on an individual image except for machine learning/statistical model and template generations. For developing a desired image processing task, programmers and researchers need to experiment with various combinations of techniques and algorithms (hundreds of algorithms are available) along with parameters tuning for each of the canonical operations. Moreover, these operations are tested on large collections of images multiple times and the experimental setup needs to be stored for the future run. Many core image operations can be found within a popular open-source image processing API called OpenCV [28]. Therefore, if a framework can be devised that also facilitates an automatic transformation of iterative operations for a single image into a parallel (processing with multiple computers) one for multiple images, this will be valuable for data scientists. All of this knowledge-base is useful for unified framework development and reshaping

the modularization for Big Data frameworks. In the following sections, we will present an analysis study of various image processing tasks to extract data-processing patterns and transform the concept into the data-parallel framework.

5.2.2 Program Synthesis and Extracting Data Processing Patterns

From the previous discussion it is persuasive that in large-scale data processing, most of the techniques, algorithms, frameworks, and software models are extensively data-centric [106, 133, 103]. In data-centric development, at first a core-feature model is developed, then data processing patterns are extracted from feature model, targeted technologies and real-world experience, and finally, components are designed and modularized based on the pattern. Therefore, understanding data-processing patterns is an important part of implementing modularised, split and decoupled data processing applications.

Our selected algorithms and techniques cover various image processing tasks in plant science, agriculture, biomedical, astronomy, and general computer vision. A total number of 30 applications we analyzed are presented in Table 5.2. Some of the selected image processing tasks with the major steps and their corresponding produced entities are presented in Table 5.3. The high-level steps are shown in Table 5.3 of extracting texts [56] from a video are: gray-scale conversion and noise removal, feature calculation, detecting text areas, then extract texts from the segmented areas of the video images. We notice that produced output of various steps has different data structures in most of the cases. However, it is necessary to figure out an optimal and unified model that might be fitted for a wide area. Most of the image analysis tasks can be divided into four re-usable tasks (please note that here we consider a single image analysis task, while two or more image analysis tasks are used to compose a complex pipeline like HtPheno [52]). Many tasks have more than four steps. However, in terms of data and program reusability, for image registration (presented in Table 5.3), matching points calculation (S_3) and Homography generation (S_4) can be considered as one logical and independent step. Similarly, in Tomograms generation, refined class objects are reused later, thus S_3 and S_4 can be combined into a single step logically.

Table 5.2: Image processing tasks we analyzed

Area of Application	Number of Image processing Applications
Plant and Agriculture	15
Medical and Biology	5
Astronomy	4
General computer vision	7

Therefore, analyzing the above-mentioned tools and techniques, we categorize the canonical operations of image analysis tasks into major four steps:

Table 5.3: Example of image processing with various steps and produced entities

Steps Output	**Img Clustering**	**Img Registration**	**Text Extraction[56]**	**Pattern detection in Tomograms[130]**
S_1	Grey conversion I_P	Grey conversion I_P	Decomposition I_D	Gaussian Trans. I_G
S_2	Feature extract V_F=$\{F_1, ..Fn\}$	Metrics Calc. V_M=$\{M_1, .M_n\}$	Feature extract V_F	Feature extract V_F
S_3	Train & Model M_M=K-mean	Matching points P_M	Train& Model M=Mlp, An	Cluster objects OB_n
S_4	Grouping C_n {n=1,2.. }	Homography M_H=[..][..]	Area segments I_S=$\{S_1, ..S_n\}$	Refine Class-objs C_n
S_5	–	Warping& align I_r	Text extraction T_r	Generate Tomog. I_T

Here, I_p to I_g - processed image, V_F - feature vectors, I_r - result
C_n - list of classes

Preprocess/conversion (S_1): This step is the first and very common for every image analysis pipeline. This step may produce different kinds of output (such as grayscale image and Canny edge image) based on applied techniques or algorithms, such as Gaussian blurring, wavelet transformation, image contrasting, image enhancement, noise reduction and so on [28]. S_1 for all the tasks in Table 5.3 falls in this category.

Estimate/Extraction (S_2): In this step, different kinds of algorithms such as SURF, SIFT, ORB, HOG [28] are applied for calculating features, metrics, and key points. However, other texture generation techniques are also employed after the feature and keypoint extraction step. This step produces array, vector or list type data-structures. S_2 for all the tasks in Table 5.3 is included in this category.

Model/Fitting (S_3): This step uses extracted features, metrics or composed data for fitting, training or developing models for generating templates based on which final analysis and processing are done. S_3 and S_4 of Image Registration and Tomogram extraction in Table 5.3 fall in this category (S_3 for other tasks).

Analysis/Postprocess (S_4): This final step mainly produces processed images and analysis results based on the generated template or the model in the model-fitting step. The produced results of this step include matched images, extracted objects, clusters of images, and registered images along with statistical results. S_4 of Image clustering and S_5 for other tasks in Table 5.3 fall in this category.

Such a categorization of the operations based on produced data and computation logic would help developers and programmers to wrap image processing tasks into a common data model and abstract frameworks. Furthermore, program synthesis of the above-mentioned image processing tools, their operations, and I/O

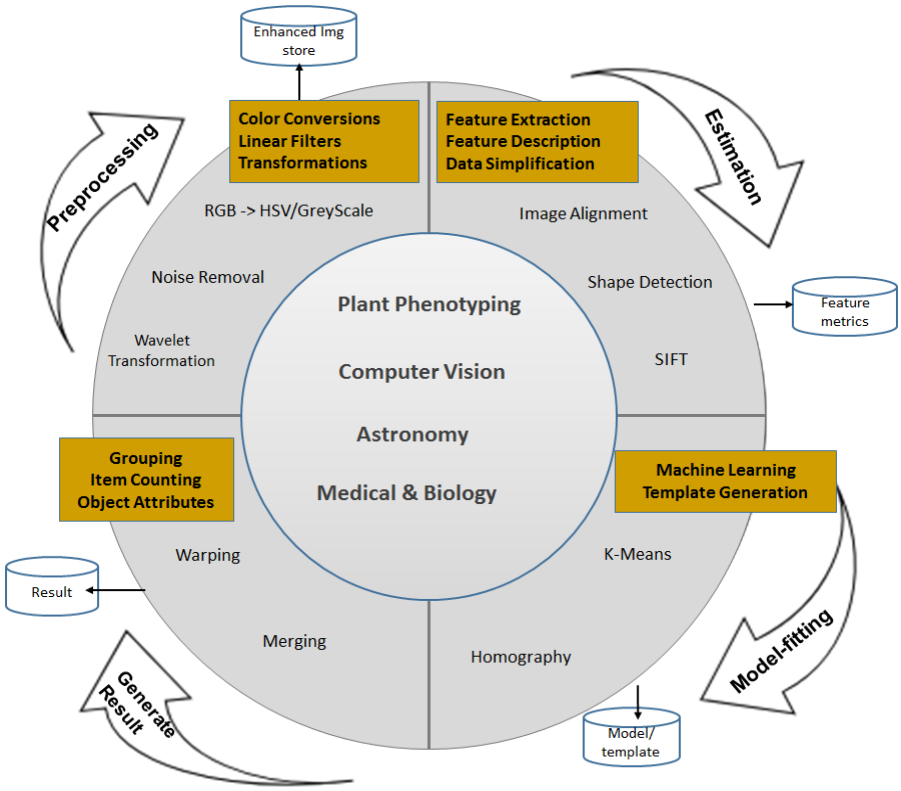

Figure 5.2: Data processing pattern (some are presented in Table 5.3).

operations allowed us to come to a conclusion that produced data in various steps as discussed in Table 5.3 can also be saved for later reuse. Consequently, these tasks should be modularised not only for program reuse but also for data entities re-use. In summary, the data processing pattern for image processing tasks can be described as follows (as shown in Table 5.3):

- Input of Preprocessing step (S_1) is generally $\{I, R_{S1}\}$, where I is raw images and R_{S1} is parameters, produced data are processed images I_P. Parameters may be numeric values, meta-data, vectors, or even raw images.
- The next two steps (Estimation and Model fitting) input are $\{I_P, R_{S(2/3)}\}$, where I_P is the produced entity of the previous step and $R_{S(2/3)}$ represents parameters of these steps.
- However, in some cases the input of the last step (Result generation) is $\{I, I_P, D_M, R_{S4}\}$; where D_M is the model or template generated in the third step, and I_P is the outcome from the first step.
- We observe that many cases, I and I_P are required to flow and retain up to the last step which is handled with disk storage in localized processing.
- For a few cases, images are required to group or bundle during S_3 and S_4 (e.g., image registration and panoramic view generation). Likewise, produced results in image processing tasks have various types.

Those are a single image, a collection of images or image objects (I_R), list of string or numeric values (L_s), a collection of matrix or vectors (L_M), dictionary (D_S), a tuple of lists (T_L), and so on.

The common data processing model is presented in Figure 5.2. This common pattern is the basis for interactive image analytic tools development for both usual and large scale data. In the next subsequent sections, we will discuss in details how to implement this data-processing patterns into map-reduce frameworks considering a unified programming interface.

5.2.3 Transformation to Data-parallel components

In this section, we discuss how to implement the image processing tasks into modularised and abstract steps in Big Data frameworks following the extracted data-processing pattern. We focus on Apache Spark (with HDFS) implementation which is optimized and the mostly used [117, 79] framework. Here, the data-processing pattern serves as modularity properties. We will use many terms and symbols to avoid frequent use of the phrases in our description (many of them are introduced by Smith and Albarghouthi [117]).

5.2.3.1 Challenges

Recent works [117, 79, 50, 72, 131] with map-reduce frameworks provide firm evidence that map-reduce based implementation is non-trivial for flexible and scalable data processing. Moreover, many applications are not yet a good fit for Big Data platforms using traditional map-reduce techniques due to network induced non-determinism, data shuffling [117], and run-time data increment [79]. For example, researchers are still working to make KNN [79] more feasible for large data with Big Data platforms. Storage files of text and Genome data could be partitioned into further smaller blocks for efficient distributed processing. But data file of each image and associated meta-data is required to treat as a single unit for image processing. Few images among thousands of collection might be corrupted and disrupt the whole processing task. This scenario is also required to handle during large scale processing. However, all the operations in map-reduce based platform (i.e., Spark) should be done with the data parallel components ($\sum_{DP}$): *map(), reduce(), filter(), join(), repartition(), subtractbykey(), count(), collect()* along with $\lambda-$*expressions* (PABS) [117]. All the image operations cannot be easily paralleled with this platform. When data size is big enough, a single additional operation with $\sum_{DP}$ takes a significant amount of time. Moreover, broadcasting data entity frequently to the worker processes might add further overhead. Consequently, programmers are required to be more careful and thoroughly test with a full dataset. For reusable computation, each step should be independent in terms of execution, data sharing, and data storing. Having said that, steps should not be divided arbitrarily like usual programming where a program can be refactored into smaller functions or modules. As we discussed in Section 5.2.2, raw-data, processed data, and external parameters need to flow from one step to another, and this might increase both memory and time overhead (with the number of steps). Apart from these, handling of various types of produced results (as described in the data processing pattern) requires a well-defined rule to store in a distributed environment.

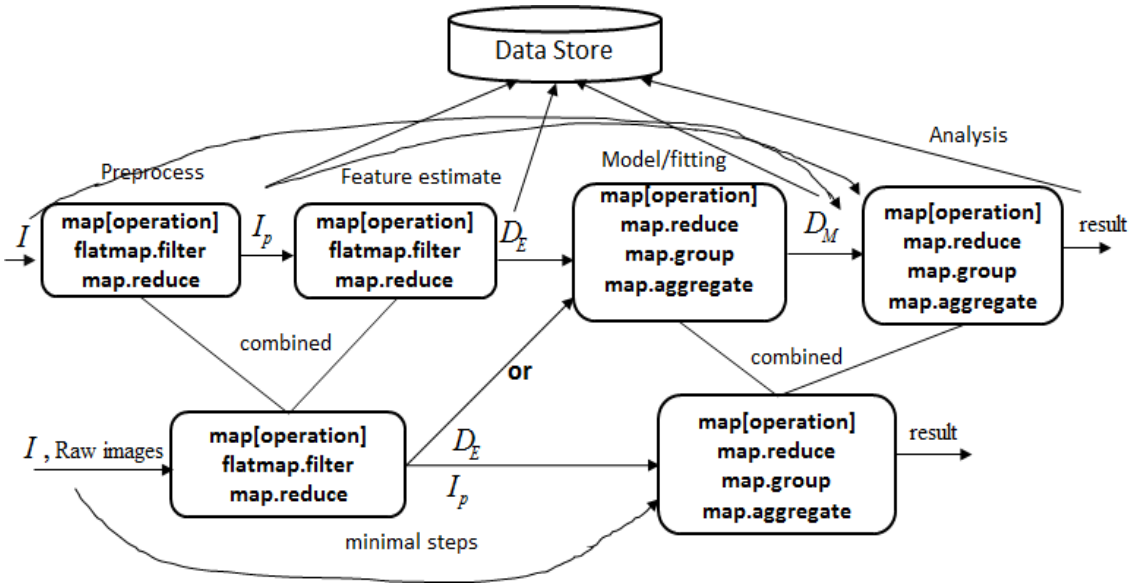

Figure 5.3: Modularisation of image analysis in data-parallel framework (both minimal and optimal split)

5.3 Proposed Modularity Model

Image processing tasks can be implemented in a various number of modularized steps (one or more) with data-parallel frameworks as shown in Figure 5.3. Here we introduce data-parallel module, $M_{DP} = \bigcup_{i=1}^{n} DP_i$ as a combination of one or more data-parallel components in $\sum_{DP}$. The split into M_{DP} is followed by the corresponding data-processing patterns presented in Section 5.2.2. From the analysis of data processing patterns of image processing applications, we identified four canonical steps: S_1, S_2, S_3, S_4. A step is a combination of many operations (some of them are canonical also), and it is essential to detect which operations require parallelism and which parts do not. We can represent $S_i = \{P_{OS}, NP_S\}$, where P_{OS} represents operations that require parallelism, and NP_S represents not parallel. All P_{OS} within a $M_{DP}(S_i)$ should be combined in such a way that the number of $\sum_{DP}$ are minimal (i.e., this rule restricts the modularity of usual computation). However, for a few steps in some cases, run-time data should be partitioned (based on heuristics [79]) for further optimization (as shown in Listings 5.3). A module, M_{DP} must produce a meaningful outcome that can be reused in future either by one of the independent operations in S_1 to S_4 or another task (or pipelines). However, a complete Image analysis task could be implemented with one or two minimal steps in map-reduce frameworks (Figure 5.3). As we observe, in most of the cases the first two steps– S_1 and S_2 can be executed with one component in $\sum_{DP}$. These two steps can be combined into one M_{DP}. Other two steps– S_3 and S_4 require more than one components in $\sum_{DP}$. Another key thing to remember is that in data-parallel components, input entities and parameters are a different thing (Smith and Albarghouthi [117] define them as *arity* and *free variable* respectively). Sometimes, step S_4 requires the input parameters whose values are calculated from either S_1 or S_2. Consequently, S_3 should be in a separate M_{DP}. Similarly, S_4

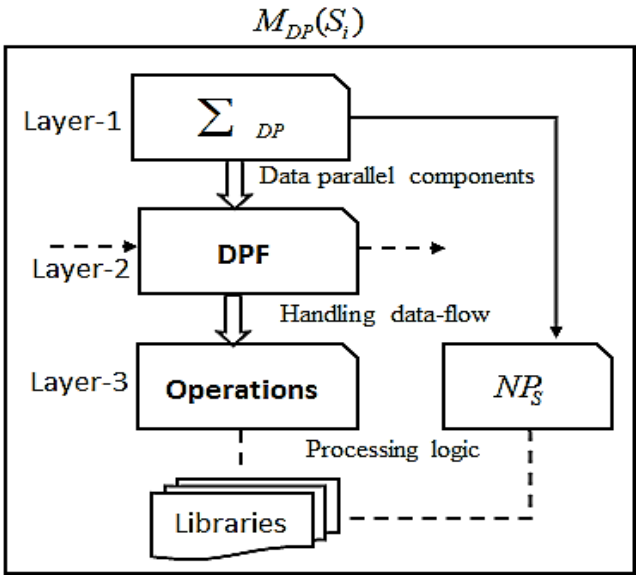

Figure 5.4: Structure of data-parallel module

requires the input parameters calculated from all collective elements from S_3. Therefore, S_4 is separate from S_3. For many image processing tasks, S_3 and S_4 are combined into a single data-parallel step. However, from a reusability and customization perspective, we propose to wrap up the independent meaningful four steps into four M_{DP}– $M_{DP}(S_1), M_{DP}(S_2), M_{DP}(S_3), M_{DP}(S_4)$. As we noticed in data-processing patterns, in many cases outcomes of the steps are required to flow and be retained among intermediate steps (even up to the last step). That poses a challenge to data-parallel implementation as this data flow may increase both run-time memory and execution overhead.

We present a solution considering a common list of data-entities with defined order to link-up data-flows among the M_{DP}. We recommend a three-layers vertical implementation of M_{DP} for image pipelines as presented in Figure 5.4. Layer-1 consists of abstract interfaces and $\sum_{DP}$, layer-2 handles parallel data-flow (DPF) and order of data entities (pseudo code is shown in Listing 5.1), and layer-3 contains S_i on images. Data-parallel operations could be optimized using layer-1 without considering others. Layer-3 also works as a bridge to include image processing libraries (Skimage, OpenCV). Processing logic in this layer can be improved without the knowledge of Layer-1. Components of Layer-1 call components in Layer-2, and Layer-2 call components in the lower layer. Therefore, three layers version of data-parallel module, $M_{DP}(S_i) = \sum_{DP} . > DPF. > P_{OS}[N, R]\{NP_S\}$. Here N is the input entities (similar to RDD elements in Spark) populated by lambda operations (PABS), and R is the list of parameters as described in data-processing patterns, I and I_P can be common in N. Only $\sum_{DP}$ (via PABS) will call P_{OS} through DPF.

This modularity model provides a multidimensional (3x4, three layers and four modules) separation of concerns and dependency inversion principle (which is valuable for parallel development as distributed programming experts and image processing experts are not the same people usually). This will give the tool developer a common programming model to rapidly implement the sequential tasks into a Big Data platform.

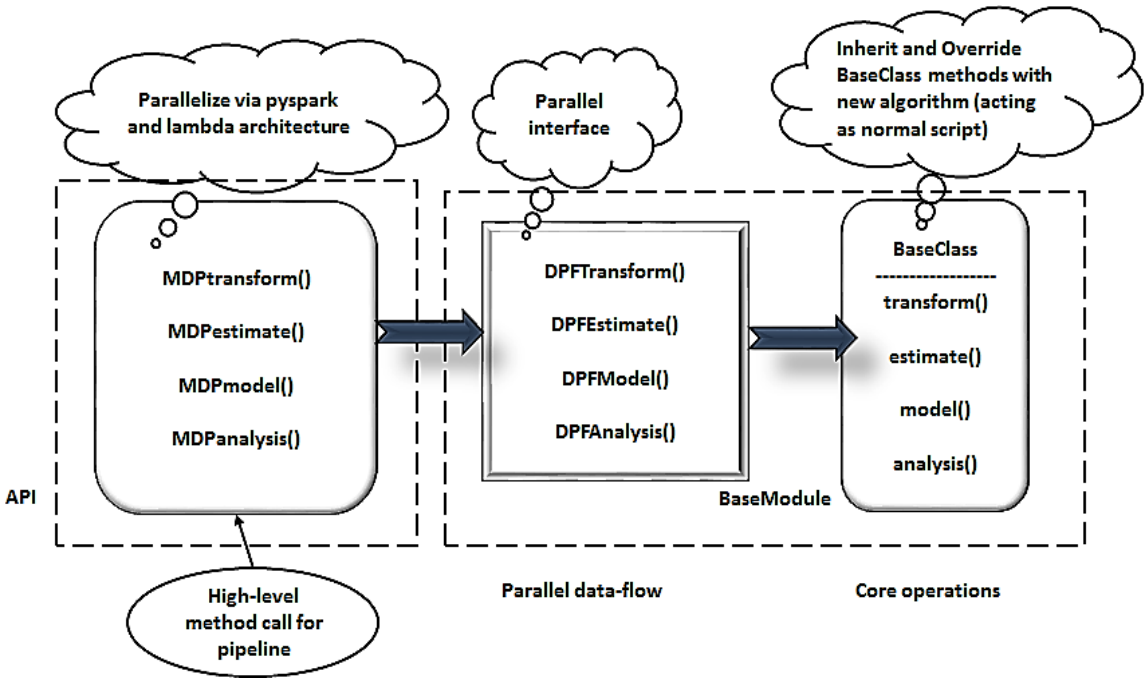

Figure 5.5: Common interface diagram for Image pipelines

Finally, we recommend to save I_R and L_M into distributed storage, other types of result should be stored either in flat storage or databases. A block diagram is shown in Figure 5.5 on how a common programming interface could be utilized using the M_{DP} and processing patterns for interactive workflow/pipeline development. However, in the experimentation phase, we will discuss what will be the impact of modularization and maintain a common list of data entities for each M_{DP}.

Listing 5.1 Pseudo code of DPF of $M_{DP(S_2)}$ and $M_{DP(S_3)}$

```
1  DPFEstimate(N, obj, Rs2)
2      unpack(N)-->im_id, I, Ip
3      metrics = obj.estimate(Ip, Rs2)
4      ...
5      return pack(im_id, I, Ip, metrics)
6  DPFModel(N, obj, Rs3)
7      unpack(N)-->im_id,I, Ip, metrics
8      Dm = obj.model(Ip, metrics, Rs3)
9      ...
10     return pack(im_id, I, Ip, Dm)
```

5.4 Impact Analysis of Modularization

In this section, we will discuss our experiments to observe the modularising effect of the image processing applications based on the modularity model presented in the previous section. Our experimental environment is Spark standalone and Cloudera cluster consists of seven worker nodes with total 58 cores and 56GB RAM. The model of the processors is Intel Xeon L5420 and the speed is 2.50GHz. The master node is configured to 36GB main memory for the cluster driver. We conduct our experiment on three datasets: a set of CANOLA field images (most of them contain flowers, each image size is 1280x720), and two sets of crop field images (each image size is 1280x960). We implemented the programs with Spark-2.0.1 and Python 2.7. We compared the execution time between modularised and compact versions of six image processing tasks: (i) Image matching [47], (ii) Image classification [123], (iii) CANOLA flower count (modified version of the base algorithm [20] for B-Channel), (iv) Object extraction [51], (v) Image registration [30], and (vi) Mosaic image generation [31]. We utilize the OpenCV [28] image processing library to implement the operations in Layer-3. We found interesting behavior of differences between modularised and non-modularised versions. In the case of time-intensive operations, there is a significant performance issue when modularization is used for some tasks (i.e., Image Registration). However, in many cases, there is no significant differences– object separation, matching, and mosaic image generation.

From Table 5.4, we notice that the difference in execution time $\Delta(t)$ for counting flowers from 2K images, $T_{min} - T_{op} = 6.1 - 5.5 = +0.6$ minutes, while for 8.6K images, the modular version overcomes the run-time memory exceeding issue of the master node (with 36GB RAM). That means modularising the tasks facilitates performance optimization for individual step. For image registration, $\Delta(t)$ is about -4 minutes and -13 minutes (slower) for the two datasets for the modular version respectively, thus modularization decreases performance. The performance is affected because of extensive data flow (image bundles) from $M_{DP}(S_1)$ to $M_{DP}(S_4)$. Whereas, the minimal step version has virtually no parallel data-flow. Likewise, for higher dataset, $\Delta(t)$ for modular image classification is -9 minutes. All other cases, $\Delta(t)$ is almost 0, meaning no impact on execution time between modular and non-modular versions. However, we found some challenging tasks during our experiment those are: flower counting, image grouping, and mosaic image generation. With the increment of data-size, execution time and run time memory issues increase (with non-modular version). For 2K images the performance of flower counting is feasible, but we found that for 8.6K images either execution time is unusual or causing memory exceeding issues during run time. Modularization and further splitting up the $M_{DP}(S_3)$ module (as shown in Listing 5.3) of flower counting program solved the memory issue. Moreover, the execution time decreases. However, in mosaic image generation, processing 300 images takes more than 400 minutes (while the well-configured machine takes 210 minutes for the usual program) for both minimal and modular version; with the increment of images, the complexity arises (603 images take more than 600 minutes). Above all, although, for a few tasks modularity increases execution time, we can say that micro-level split (hence modularity) increases the opportunity to further optimization.

Table 5.4: Performance comparison of two versions of the image processing tasks with Spark cluster (execution time is presented including I/O operations).

Tasks	**#Imgs**	T_{min}, **minimal**	T_{op}, **modular**	**#Imgs**	T_{min}, **minimal**	T_{op}, **modular**	**#Min Steps**
Image Matching	2K	3.3mins	3.3 mins	8.6K	13mins	13mins	1
Clustering	2K	11mins	11 mins	4K	18mins	27mins	2
Flower count	2K	**6.1 mins**	**5.5 mins**	8.6K	**Mem issue**	**19** mins	3
Object Extraction	2K	0.8min	0.8 min	8.6K	2.1 mins	2.1 mins	1
Image Registration	0.5K	**9.3 mins**	**13 mins**	1.5K	**27 mins**	**40 mins**	1
Mosaic Image	0.2K	>300mins	> 300mins	0.3K	>400mins	>400mins	2

Apart from these, if we follow common data processing patterns as described in Section 5.2.2, it is possible to write common data-parallel modules (M_{DP}) at the micro-level for the high-level components of large data analytic tools. This not only facilities re-usable module development but also we can write a common abstract interface to work with image processing without much knowledge of data-parallel operations ($\sum_{DP}$) and tuning. If separate operations are implemented as common method signatures (as shown in Figure 5.5) within a class and its object instances are passed through corresponding M_{DP}, then processing logic can be reused or customized (shown in Figure 5.6) willingly without the knowledge of data-parallel components. With our model, the image processing tasks which contains only P_{OS} in S_i can be easily transformed into data-parallel programs without knowing the details of Layer-1 and Layer-2. Furthermore, if R_{S1} to R_{S4} do not depend on the outcome of any of the steps in S_i (and no NP_S) then those image processing tasks can be automatically converted into M_{DP}. Coupled with the data-processing pattern (in Section 5.2.2), our model represents a strong design rule [34] in this domain. For instance, consider a project where one team is working on the web part, one team is working on the efficient large scale data processing support, and another team is working on the image processing part; here dependency inversion principle (depend upon abstractions; do not depend upon concretions [94]) is essential. More sophisticated techniques and algorithms might provide a framework to auto-transform the image pipelines into M_{DP} in future, but this is out of the scope of our study.

5.5 Discussion

From our experimental analysis with six image processing tasks, we identified three challenging tasks to be optimized for Big Data platforms which require more research. Even with small data, they take too many

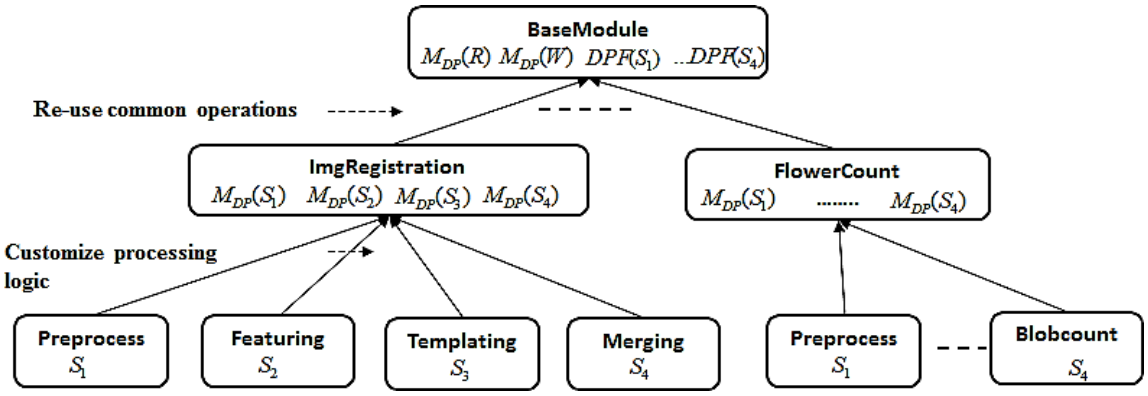

Figure 5.6: Options of reusability and customization. Common modules such as DPF can be placed in BaseModule and tasks (e.g., FlowerCount) can reuse those along with the other modules. Each canonical step (S_i) can be a separate module; they are reused and customized for each of the tasks with new computational logic without the knowledge of upper layer of (M_{DP}).

hours to finish with enough computing power and in-memory capability. We believe that many others image processing tasks are existing yet to be improved to good fit with Big Data platforms.

Listing 5.2 Algorithm for flower counting with Spark

```
1 Pr_RDD <— RDDraw.map(preprocess)
2 Es_RDD <— Pr_RDD.map(histogram, params).cache
3 avg_hist <— Es_RDD.reduce(sumhistogram)/length
4 Cr_RDD <— Es_RDD.map(corrlation, avg_hist)
5 result <— Cr_RDD.map(blobcount, plotmask)
```

Listing 5.3 Optimized algorithm of step 3 of Flower Counting program

```
1 Tmp_RDD = Es_RDD.flatMap(histGrm).zipWithIndex().cache()
2 split = #elements / #maps
3 for i=0 to #maps:
4     start = i * split
5     end = start + split
6     split_hist = Tmp_RDD.filter(in range(start,end)).collect()
7     sum_histogram = sum_histogram + sumHistograms(split_hist)
8 avg_hist = sum_histogram/#maps
9 Cr_RDD <— Es_RDD.map(corrlation, avg_hist)
```

Listing 5.2 represents the usual Spark version of flower count, here line 3 is the most costly step of flower count. Sometimes, processing of small size raw data may exceed the main memory limit during run-time. In our case, for 36 GB RAM of the master node exceeds the memory during executing this step for counting flower [109] of 8.6K images. But, our configuration is a good fit for 2K images. Therefore, this step must

be split up more intuitively for feasible execution with the limited resource. We split the data-parallel components and run-time data in that step to overcome the issue. We choose the value of split of Histogram calculation by the ratio of number of images ($\#elements$) and number of cores ($\#maps$) used in distributing the processing. The tuning version of Canola flower count is presented in Listing 5.3. We also found issues on usual Spark version of mosaic image generation [31] with SIFT features and KNN matching algorithm. The performance of the most recent (July 2017) map-reduce based algorithm we found based on SSIM and K-medoids based technique [113] which is not promising as it takes 314 minutes for 5,679 images of 75x75 size. Moreover, their techniques took extracted features as input. We also observed similar execution time despite trying four different techniques. Our implementation of mosaic image generation with 200 and 300 images (similar to 65,536 images of 75x75 size [113]) takes more than 300 minutes and 400 minutes respectively with color conversion and feature extraction steps. Consequently, mosaic image generation using random selection does not fit with Spark or Hadoop cluster using the traditional data-parallel components. Therefore, the execution steps of mosaic image generation are required to be solved more intelligently which appears to be non-trivial. One of the feasible techniques (among four) is shown in Listing 5.4. However, those techniques do not improve the execution time in a significant way. The reason for the almost constant execution time lies in the matching ratio calculation (computation intensive) and maximum matched image selection dynamically which takes 1.5 to 2 minutes on average in our cluster (the execution happens n-1 times when all the images are merged). Moreover, the merged (larger) image needs to be broadcasted to the workers in every step. A similar execution time problem exists for image grouping for which we used the MLib K-Means API of Spark that is considered to be the most standard one so far. During execution time of image grouping, overall numeric records for 4K images are $\sim$133 million. And the input size raises to 50GB during training K-Means and total time is taking around 18 mins (with minimal steps, but for 8.6K images it takes infeasible time).

We have developed an API for flexible image pipelines development following the proposed micro-level modularity model. In the API, line 1 and 2 in Listings 5.3 and 5.4 are contained in our proposed $M_{DP}(S_1)$ and $M_{DP}(S_2)$ along with few non-parallel operations. Pseudo code in Listings 5.3 and line 3 to line 18 in Listings 5.4 constructed $M_{DP}(S_3)$. $M_{DP}(S_1)$ and $M_{DP}(S_2)$ are directly reused or sometimes operations are customized using inherited class without editing M_{DP} for many image pipelines. We also added two more modules–$M_{DP}(R)$ and $M_{DP}(W)$ for reading images and writing the processed result to/from HDFS and flat storage. We observe one important fact that I/O with HDFS are more efficient (even works are ongoing for further efficiency [57]) but have one bottleneck for images. HDFS operations treat images as text files. Therefore, we need to read and write (data-parallel) the images as text data. However, after processing the data, for many cases, visualization and further usages require another localized program to convert into an image format. For thousands of images, it consumes a significant amount of time. To eradicate the bottleneck, we implemented parallel reading and writing images by the workers to the flat-storage server (ssh protocol). Nonetheless, we notice that the ssh server cannot handle I/O for more than 2,000 images at the same (virtually) time and the program fails.

Listing 5.4 Splitted algorithm of Mosaic image with Spark

```
1  Pr_RDD <— RDDraw.map(preprocess)
2  Fs_RDD <— Pr_RDD.map(faetureExtract)
3  Mosaic <— Fs_RDD.first()
4  Tmp_RDD <— Fs_RDD.zipWithIndex().cache()
5  #split = #elements / split_size
6  traversed.add(Mosaic)
7  do until all_images traversed
8  for i=0 to #split
9      start = i * split_size
10     end = start + split_size
11     Filter_RDD <— Tmp_RDD.filter(not(traversed) and in(start, end))
12     MFeature <— broadcast(faetureExtract(Mosaic))
13     Matched_RDD <— Filter_RDD.map(matchpoints, MFeature)
14     MaxImg <— Matched_RDD.reduce(max(matchratio))
15     if(MaxImg.ratio > previous_ratio)
16         SeletedImg <— MaxImg
17         traversed.add(SeletedImg)
18     Mosaic <— mergeHomography(Mosaic,SeletedImg)
19 result <— Mosaic
```

5.5.1 Lesson learned

In summary, from our case study we extracted the following important insights:

- Still, I/O operations create a bottleneck for optimal image processing with data-parallel frameworks
- Using modular data-processing patterns will reduce the implementation effort and increase the reusability of both the program and the processed entities (in various canonical steps of a task) in data-parallel frameworks.
- Programmers should avoid arbitrary modularisation.
- Programmers should not rely on usual map-reduce concepts and tuning hardware resources only for computation intensive tasks.
- Intelligently splitting up the map-reduce operations and run-time data further might solve the limited resource problems as well as increase performance.
- All image processing tasks may not be a good fit for traditional map-reduce techniques.

5.6 Related Work

A number of studies [79, 72, 113, 131] have pointed out the challenges and problems of implementing computation intensive tasks for scientific data with the abstract data-parallel frameworks in spite of having enough computing resources. To reduce the efforts of the data-scientists for large scale data analysis, some applications and frameworks are being developed [45, 92, 125, 70] for GeoSpatial and Bioinformatic data processing by adding more abstract layers on top of map-reduce frameworks. Despite some progress, they do not support image processing operations. However, a few studies attempted to develop software and tools [44, 67, 130, 66, 25, 139, 118] for large scale image processing for few specific cases. Nonetheless, they do not provide a common framework for diverse image processing pipelines. Our objective is to develop a scalable, unified and abstract framework for developing interactive image processing pipelines for large scale data.

Nowadays, large-scale images are used for analysis in various scientific works and general computer vision. Although a few studies provide techniques [118, 139, 8, 113, 124] for specific image processing tasks with data-parallel frameworks, they do not describe the challenges and optimization techniques to overcome the challenges. In this study, we highlight the challenges of real world large scale image processing tasks as well as recommend optimization technique with data-parallel components.

It is proven that program modularization is a key concept for developing unified frameworks on top of distributed and map-reduce programming environment. However, modularising map-reduce jobs (computation and data-intensive) is still challenging as data-parallel frameworks only provide a limit of few strict API methods. Yang et al. [134] attempted to develop a framework for running modular map-reduce jobs, but users need to provide modularized jobs and dependency information. Recently, Heit et al. [53] proposed a modular architecture for working with statistical models for data mining. However, these studies do not provide any technique of micro-level modularity and impact of modularizing the tasks. In our work, we propose a strategy for modularising large-scale image processing tasks at the micro-level and illustrate the pros and cons of modularising tasks with data-parallel frameworks.

In summary, our study on computation-intensive task analysis strategy, modularity model, and experimental insights will provide the researchers to focus on such challenges extensively for devising better techniques, and developers to consider the insights during large scale image processing tools development.

5.7 Conclusion

In this study, we presented a case study on modularising data and computation intensive tasks into micro-level components. Our focus is on large image data as there is a lack of studies on the implications of running a wide variety of image processing tasks on Big Data platforms. We synthesize image data-processing patterns and propose a unified modular model for the effective implementation of computation-intensive tasks on data-parallel frameworks considering reproducibility, reusability, and customization. Our experimental results with

six real world image processing tasks show that splitting and modularising the computation tasks is crucial to utilize the power of Big Data platforms. However, not all tasks show similar performance in execution time after modularising. A few of them need more sophisticated techniques for optimization with data-parallel frameworks. Therefore, our study provides a valuable knowledge-base for abstract and unified frameworks development for large scale data analysis. In future, we will work on techniques for automatic transformation of sequential tasks to data-parallel modules.

Chapter 6

Conclusion

6.1 Concluding Remarks

Plant Genotyping and Phenotyping are important for tackling global food insecurity. In this thesis, we focused on the plant Genotyping and Phenotyping analysis systems that work on diverse data types and size. Cloud-based systems are becoming popular for accessible, reproducible, and transparent computation in this domain. Some of the existing systems are: Galaxy [49], iPlant Collaborative [83], GenAP [68], and so on. However, existing systems do not provide all the core large-scale data processing capabilities. Consequently, more effective and stable solutions and methodologies are required that also reduce maintenance and development efforts.

To that end, in this thesis, we have presented an empirical study on architectural models and modularization strategies for large-scale data processing support in Genotyping and Phenotyping analysis system development. In the first study, we analyze architectural stability and detected sub-components that are frequently changed in the development lifecycle of these systems. We have also proposed a key-term based technique for extracting architectural change related activities for analysis and maintenance purposes. This technique will be helpful for the automatic architectural traceability analysis from the development history of a system as well. The key observations from the first study are: (i) architectures of the existing Genotyping and Phenotyping analysis systems are unstable, and (ii) the core components of these systems are evolutionary due to mainly usability, new scientific algorithms, and technological changes. In the second study, we have extracted a reference architecture, explored design limitations of candidate systems, and proposed an architectural model. We recommend that for a Genotyping and Phenotyping analysis systems development, three design guidelines are useful: (i) data-centric model extraction, (ii) independent component modularization, and (iii) proper infrastructure model following the previous two rules. One of the key observations from the prototype system development following the proposed reference architecture is that the components handling high-volume data processing should be kept into the Big Data cluster (hence separate code-base) to avoid frequently sending those components from the application server during execution. In the third study, we present a case study on developing modularized components for computation intensive programs following the data-centric model extraction. Furthermore, we identified some challenges of large-scale image processing with Big Data platforms. Our last study generated empirical data on the performance of several

image processing tasks with varying choices suggested by our architectural model, and this performance data suggested several special recommendations for micro-level modularization in Big Data platforms. One of the key findings is that the developers should avoid arbitrary modularisation in MapReduce framework for large-scale data processing tasks.

Al-Jaroodi et al. [12] suggested that a suitable software development model is essential for Big Data analytic applications development. Although we emphasized on large-scale Genotyping and Phenotyping data analysis, our proposed architectural model would be helpful for developing cloud-based systems in the domains of Astronomy, Bioinformatic, BioMedical, and GIS. Furthermore, adopting our proposed data-centric model and independent component model (shown in Figure 4.11 and discussed in Chapter 4) could facilitate the flexible development of various kinds of software (e.g., business intelligence systems [53], and national security systems [63]) which mainly focus on data handling, processing, and analysis. Because, data-centric modules are easy to make a design decision and tracking the core components, the independent component model (developed as an API or library) can be easily changed, updated or restructured without affecting much of the whole system. Nowadays, many software systems (including mobile applications) need to handle data accessing from various sources and services, even those systems provide data to the other applications as services. Structuring all of these data handling components into a unified data interface following well-defined rules would provide adaptable and flexible system development (Hadoop-BAM [92] provides a good example of a unified data framework for Bioinformatics).

Performance, security, cost, and disaster recovery are really essential for a large cloud-based system to serve the uninterrupted demand. Our recommended heterogeneous model [135] of service infrastructure would be very effective for the web applications in-general that handle high-volume data storage and processing.

In addition, proposed data-centric modularity model (discussed in Chapter 5) in this thesis might provide a strict rule to modularize components for separation of concerns and parallel development in other contexts where large volume data need to be handled. For example, Facebook, Netflix, Linkedin, and FIU-Miner, described by Pääkkönen and Pakkala [99], focus on processing data in different stages with different platforms. For these systems, unified data interface and data-centric modularity could play an effective role for the developers.

6.2 Limitations and Future Work

In this thesis, for architectural change analysis from the commit history, we propose an automatic technique that explores only intentional commits reflecting architecture changes. We will work to extend the technique for unintentional architectural change analysis as well. However, we have presented an abstract reference architecture, and the software component model requires more intuitive development patterns. In future, we will conduct an empirical study of concrete design rules for the software component model. Moreover, we will focus more on a generic component model for efficient collaboration among the users and stakeholders. In

the second study, we are unable to evaluate the reference architecture extensively because of lack of standard ways. To that end, we will conduct an empirical study for evaluating the architectural model with the software experts and developers in terms of the bad smells [15] of a software. Besides, we have worked only on a unified modularity model for image processing tasks. However, unified modularity model for Genome and GIS data would be fruitful for the design and development of a Genotyping and Phenotyping analysis system, and we will also focus on that direction. Apart from these, we will work on the following issues:

(1) Efficient programming model and data model to overcome the existing challenges: During the experimentation, we identified the challenges for large scale image processing tasks for the map-reduce implementation. We will work on developing an efficient distributed programming model and data-model to overcome the challenges.

(2) Empirical study on the automatic transformation of image processing tasks to data-parallel modules: We will conduct experiments for a rule-based automatic transformation of image processing logic into distributed programming that will reduce the efforts of programmers.

(3) Empirical study on design guidelines of collaborative and visualization tools for large scale data analysis: We will work on the design model for a collaboration and visualization tool for large scale data.

(4) Automatic technique of architectural traceability from commit history: Existing empirical studies report that traceability of architecture is important for developers, but no effective automatic technique is available. In our future work, we will work on the automatic technique of architectural traceability using commit history.

References

[1] Biomina. http://biominavm-galaxy.biomina.be/galaxy/. March, 2017.

[2] The hadoop. http://hadoop.apache.org/. June, 2017.

[3] The hdp sandbox. https://hortonworks.com/products/sandbox/. August, 2017.

[4] Lemnatec. http://www.lemnatec.com/products/. March, 2017.

[5] P2irc project. http://p2irc.usask.ca/. University of Saskatchewan, June, 2017.

[6] Sei-saam: http://www.sei.cmu.edu/architecture/scenario_paper/ ieee-sw2.htm. March, 2017.

[7] The spark. http://spark.apache.org/. June, 2017.

[8] The stormcv. https://github.com/sensorstorm. June, 2017.

[9] Enis Afgan, Brad Chapman, Margita Jadan, Vedran Franke, and James Taylor. Using cloud computing infrastructure with cloudbiolinux, cloudman, and galaxy. *Current protocols in bioinformatics*, pages 11–9, 2012.

[10] Dinesh Agarwal and Sushil K Prasad. Lessons learnt from the development of gis application on azure cloud platform. In *Proceedings of the 2012 IEEE 5th International Conference on Cloud Computing*, pages 352–359, 2012.

[11] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, pages 1792–1803, 2015.

[12] Jameela Al-Jaroodi, Brandon Hollein, and Nader Mohamed. Applying software engineering processes for big data analytics applications development. In *Proceedings of the 2017 IEEE 7th Annual Computing and Communication Workshop and Conference*, pages 1–7, 2017.

[13] Mamdouh Alenezi. Software architecture quality measurement stability and understandability. *International Journal of Advanced Computer Science and Applications*, 2016.

[14] Samuil Angelov, Paul Grefen, and Danny Greefhorst. A framework for analysis and design of software reference architectures. *Information and Software Technology*, pages 417–431, 2012.

[15] M. Aniche, G. Bavota, C. Treude, A. V. Deursen, and M. A. Gerosa. A validated set of smells in model-view-controller architectures. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution*, pages 233–243, 2016.

[16] Lerina Aversano, Marco Molfetta, and Maria Tortorella. Evaluating architecture stability of software projects. In *Proceedings of the 2013 20th Working Conference on Reverse Engineering*, pages 417–424, 2013.

[17] Muhammad Ali Babar, Torgeir Dingsyr, Patricia Lago, and Hans van Vliet. *Software Architecture Knowledge Management: Theory and Practice*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[18] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, M. Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, second edition, 2010.

[19] Rami Bahsoon and Wolfgang Emmerich. Architectural stability. In *Proceedings On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, pages 304–315, 2009.

[20] Neetika Bairwa and NK Agrawal. Counting of flowers using image processing. *International Journal For Technological Research In Engineering*, pages 775–779, 2014.

[21] Carliss Young Baldwin and Kim B Clark. *Design rules: The power of modularity*, volume 1. MIT press, 2000.

[22] Miklós Bálint, Philipp-André Schmidt, Rahul Sharma, Marco Thines, and Imke Schmitt. An illumina metabarcoding pipeline for fungi. *Ecology and evolution*, pages 2642–2653, 2014.

[23] M.R. Barbacci, R.J. Ellison, A.J. Lattanze, J.A. Stafford, C.B. Weinstock, and W.G. Wood. *Quality Attribute Workshops QAWs*. 3rd edition, 2003.

[24] Pooyan Behnamghader, Reem Alfayez, Kamonphop Srisopha, and Barry Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security*, pages 251–262, 2017.

[25] Haresh S. Bhatt and A. K. Aggarwal. A generalized environment for distributed image processing. In *High Performance Computing Systems and Applications*, pages 211–225, 2003.

[26] Roi Blanco and Christina Lioma. Graph-based term weighting for information retrieval. *Information Retrieval*, pages 54–92, 2012.

[27] Daniel Blankenberg, Gregory Von Kuster, Nathaniel Coraor, Guruprasad Ananda, Ross Lazarus, Mary Mangan, Anton Nekrutenko, and James Taylor. Galaxy: a web-based genome analysis tool for experimentalists. *Current protocols in molecular biology*, pages 19–10, 2010.

[28] Gary Bradski. The opencv library. *Software Tools for the Professional Programmer*, pages 120–123, 2000.

[29] H. P. Breivold and I. Crnkovic. A systematic review on architecting for software evolvability. In *Software Engineering Conference*, pages 13–22, 2010.

[30] Lisa Gottesfeld Brown. A survey of image registration techniques. *ACM computing surveys*, pages 325–376, 1992.

[31] Matthew Brown and David G Lowe. Automatic panoramic image stitching using invariant features. *Computer Vision*, pages 59–73, 2007.

[32] Adrian Caciula. Optimization techniques for next-generation sequencing data analysis. In *Scholar Works Georgia State University, Computer Science Dissertations*, 2014.

[33] Yuanfang Cai and Kevin J. Sullivan. Modularity analysis of logical design models. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 91–102, 2006.

[34] Yuanfang Cai, Hanfei Wang, Sunny Wong, and Linzhang Wang. Leveraging design rules to improve software architecture recovery. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*, pages 133–142, 2013.

[35] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software*, pages 191 – 205, 2016.

[36] Edgar Castelán Maldonado, Miguel Ángel Brigos Hermida, and Joaquín Fernández Sánchez. A sofware reference architecture for the design and development of mobile workflow learning applications. In *Proceedings of the 8th International Technology, Education and Development Conference*, pages 6351–6360, 2014.

[37] Elliot J. Chikofsky and James H Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, pages 13–17, 1990.

[38] P. Clements and M. Klein R. K. Kazman. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2002.

[39] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. Software history under the lens: A study on why and how developers examine it. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, pages 1–10, 2015.

[40] Anna Corazza, Sergio Martino, Valerio Maggio, and Giuseppe Scanniello. Weighing lexical information for software clustering in the context of architecture recovery. *Empirical Software Engineering*, pages 72–103, 2016.

[41] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, pages 107–113, 2008.

[42] Yuri Demchenko, Cees De Laat, and Peter Membrey. Defining architecture components of the big data ecosystem. In *Proceedings of the 2014 International Conference on Collaboration Technologies and Systems*, pages 104–112, 2014.

[43] Stephane Deschamps, Llaca Victor, and D. May Gregory. Genotyping-by-sequencing in plants. *Biology*, pages 460–483, July 2017.

[44] Le Dong, Zhiyu Lin, Yan Liang, Ling He, Ning Zhang, Qi Chen, Xiaochun Cao, and Ebroul Izquierdo. A hierarchical distributed processing framework for big image data. *IEEE Transactions on Big Data*, pages 297–309, 2016.

[45] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering*, pages 1352–1363, 2015.

[46] B. Fluri, E. Giger, and H. C. Gall. Discovering patterns of change types. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 463–466, 2008.

[47] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *Proceedings of the 2010 17th IEEE International Conference on Image Processing*, pages 3757–3760, 2010.

[48] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[49] Jeremy Goecks, Anton Nekrutenko, James Taylor, and Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, page R86, 2010.

[50] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 599–613, 2014.

[51] Kostas Haris, Serafim N Efstratiadis, Nikolaos Maglaveras, and Aggelos K Katsaggelos. Hybrid image segmentation using watersheds and fast region merging. *IEEE Transactions on image processing*, pages 1684–1699, 1998.

[52] Anja Hartmann, Tobias Czauderna, Roberto Hoffmann, Nils Stein, and Falk Schreiber. Htpheno: An image analysis pipeline for high-throughput plant phenotyping. *Bioinformatics*, page 148, 2011.

[53] J. Heit, J. Liu, and M. Shah. An architecture for the deployment of statistical models for the big data era. In *Proceedings of the 2016 IEEE International Conference on Big Data*, pages 1377–1384, 2016.

[54] Charles Antony Richard Hoare and He Jifeng. *Unifying theories of programming*, volume 14. Prentice Hall Englewood Cliffs, 1998.

[55] Aftab Hussain and Md. Saidur Rahman. A new hierarchical clustering technique for restructuring software at the function level. In *Proceedings of the 6th India Software Engineering Conference*, pages 45–54, 2013.

[56] M. Z. Islam and A. K. Mondal. Towards a standard bangla photoocr: Text detection and localization. In *Proceedings of the 2014 17th International Conference on Computer and Information Technology*, pages 198–203, 2014.

[57] N. S. Islam, M. Wasi ur Rahman, X. Lu, and D. K. D. K. Panda. Efficient data access strategies for hadoop and spark on hpc cluster with heterogeneous storage. In *Proceedings of the 2016 IEEE International Conference on Big Data*, pages 223–232, 2016.

[58] P. Jamshidi, M. Ghafari, A. Ahmad, and C. Pahl. A framework for classifying and comparing architecture-centric software evolution research. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, pages 305–314, 2013.

[59] Mehdi Jazayeri. On architectural stability and evolution. In *Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, pages 13–23, 2002.

[60] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, pages 47–55, 1996.

[61] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. Saam: A method for analyzing the properties of software architectures. In *Proceedings of the 16th International Conference on Software Engineering*, pages 81–90, 1994.

[62] Jungil Kim and Eunjoo Lee. The effect of import change in software change history. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1753–1754, 2014.

[63] John Klein, Ross Buglak, David Blockow, Troy Wuttke, and Brenton Cooper. A reference architecture for big data systems in the national security domain. In *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*, pages 51–57, 2016.

[64] Max Klein, Rati Sharma, Chris H. Bohrer, Cameron M. Avelis, and Elijah Roberts. Biospark: scalable analysis of large numerical datasets from biological simulations and experiments using hadoop and spark. *Bioinformatics*, pages 303–305, 2017.

[65] Sebastian Klepper, Stephan Krusche, and Bernd Bruegge. Semi-automatic generation of audience-specific release notes. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pages 19–22, 2016.

[66] Avi C Knecht, Malachy T Campbell, Adam Caprez, David R Swanson, and Harkamal Walia. Image harvest: an open-source platform for high-throughput plant image processing and analysis. *Journal of experimental botany*, pages 3587–3599, 2016.

[67] Kerry Koitzsch. "image as big data" systems: Some case studies. In *Pro Hadoop Data Analytics: Designing and Building Big Data Systems using the Hadoop Ecosystem*, pages 235–255, 2017.

[68] Christos Kozanitis and David A Patterson. Genap: a distributed sql interface for genomic data. *BMC bioinformatics*, page 63, 2016.

[69] Heiko Koziolek. Sustainability evaluation of software architectures: A systematic review. In *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS*, pages 3–12, 2011.

[70] Michel Krämer and Ivo Senner. A modular software architecture for processing of big geospatial data in the cloud. *Computers and Graphics*, pages 69–81, 2015.

[71] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, pages 26–49, 1988.

[72] I. Kusuma, M. A. Ma'sum, N. Habibie, W. Jatmiko, and H. Suhartanto. Design of intelligent k-means based on spark for big data clustering. In *Proceedings of the International Workshop on Big Data and Information Security*, pages 89–96, 2016.

[73] Kristian Kvilekval, Dmitry Fedorov, Boguslaw Obara, Ambuj Singh, and BS Manjunath. Bisque: a platform for bioimage analysis and management. *Bioinformatics*, pages 544–552, 2009.

[74] D. Le. Architectural-based speculative analysis to predict bugs in a software system. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering Companion*, pages 807–810, 2016.

[75] Ling Liu. Computing infrastructure for big data processing. *Frontiers of Computer Science*, pages 165–170, 2013.

[76] Siyuan Lu, Xiaoyan Shao, Marcus Freitag, Levente J Klein, Jason Renwick, Fernando J Marianno, Conrad Albrecht, and Hendrik F Hamann. Ibm pairs curated big data service for accelerated geospatial data analytics and discovery. In *Proceedings of the 2016 IEEE International Conference on Big Data*, pages 2672–2675, 2016.

[77] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, pages 1039–1065, 2006.

[78] Cristiano Maffort, Marco Tulio Valente, Ricardo Terra, Mariza Bigonha, Nicolas Anquetil, and André Hora. Mining architectural violations from version history. *Empirical Software Engineering*, pages 854–895, 2016.

[79] Jesus Maillo, Sergio Ramírez, Isaac Triguero, and Francisco Herrera. kNN-IS: An Iterative Spark-based design of the k-Nearest Neighbors classifier for big data. *Knowledge-Based Systems*, pages 3–15, 2017.

[80] Bruce E Mann, Philip J Trasatti, Michael D Carlozzi, John A Ywoskus, and Edward J McGrath. Loosely coupled mass storage computer cluster, 1999. US Patent 5,862,312.

[81] Bernard Marr. *Big Data: Using SMART big data, analytics and metrics to make better decisions and improve performance*. John Wiley & Sons, 2015.

[82] Masood Masoodian and Saturnino Luz. Heterogeneous client-server architecture for a virtual meeting environment. In *Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, 2000.*, pages 67–74, 2000.

[83] Nirav Merchant, Eric Lyons, Stephen Goff, Matthew Vaughn, Doreen Ware, David Micklos, and Parker Antin. The iplant collaborative: cyberinfrastructure for enabling data to discovery for the life sciences. *PLoS biology*, page e1002342, 2016.

[84] Ran Mo, Yuanfang Cai, Rick Kazman, and Lu Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Proceedings of the 2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 51–60, 2015.

[85] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, pages 120–130, 2000.

[86] Amit Kumar Mondal, Mohammad Masudur Rahman, and Chanchal K. Roy. Embedded emotion-based classification of stack overflow questions towards the question quality prediction. In *Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering*, pages 521–526, 2016.

[87] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484–495, 2014.

[88] Sergi Nadal, Victor Herrero, Oscar Romero, Alberto Abell, Xavier Franch, Stijn Vansummeren, and Danilo Valerio. A software reference architecture for semantic-aware big data systems. *Journal of Information and Software Technology*, 2017.

[89] Elisa Yumi Nakagawa, Pablo Oliveira Antonino, and Martin Becker. Reference architecture and product line architecture: A subtle but critical difference. In *Proceedings of the 5th European Conference on Software Architecture*, pages 207–211, 2011.

[90] Najam Nazar, Yan Hu, and He Jiang. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, pages 883–909, 2016.

[91] Shiva Nejati, Mehrdad Sabetzadeh, Chetan Arora, Lionel C Briand, and Felix Mandoux. Automated change impact analysis between sysml models of requirements and design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 242–253, 2016.

[92] Matti Niemenmaa, Aleksi Kallio, André Schumacher, Petri Klemelä, Eija Korpelainen, and Keijo Heljanko. Hadoop-bam: directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, pages 876–877, 2012.

[93] Malia A. Gehan Noah Fahlgren, Maximilian Feldman. A versatile phenotyping system and analytics platform reveals diverse temporal responses to water availability in setaria. *Molecular Plant*, pages 1520–1535, 2015.

[94] Martin E Nordberg. Aspect-oriented dependency inversion. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[95] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, pages 3045–3054, 2004.

[96] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[97] Femi G. Olumofin and Vojislav B. Mišić. A holistic architecture assessment method for software product lines. *Information and Software Technology*, pages 309–323, 2007.

[98] Tosin Daniel Oyetoyan, Daniela S. Cruzes, and Reidar Conradi. A study of cyclic dependencies on defect profile of software components. *Journal of Systems and Software*, pages 3162 – 3182, 2013.

[99] Pekka Pääkkönen and Daniel Pakkala. Reference architecture and classification of technologies, products and services for big data systems. *Big Data Research*, pages 166 – 186, 2015.

[100] A. Patidar and U. Suman. A survey on software architecture evaluation methods. In *Proceedings of the 2015 2nd International Conference on Computing for Sustainable Global Development*, pages 967–972, 2015.

[101] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, pages 40–52, 1992.

[102] Y. Ren, T. Xing, X. Chen, and X. Chai. Research on software maintenance cost of influence factor analysis and estimation method. In *Proceedings of the 2011 3rd International Workshop on Intelligent Systems and Applications*, pages 1–4, 2011.

[103] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. Optimization of complex dataflows with user-defined functions. *ACM Computing Surveys*, page 38, 2017.

[104] Banani Roy and TC Nicholas Graham. An iterative framework for software architecture recovery: An experience report. *Proceedings of the European Conference on Software Architecture*, pages 210–224, 2008.

[105] Banani Roy and TC Nicholas Graham. Methods for evaluating software architecture: A survey. *School of Computing TR*, page 82, 2008.

[106] Banani Roy, Amit Kumar Mondal, Chanchal K Roy, Kevin A Schneider, and Kawser Wazed. Towards a reference architecture for cloud-based plant genotyping and phenotyping analysis frameworks. In *Proceedings of the 2017 IEEE International Conference on Software Architecture*, pages 41–50, 2017.

[107] J. S. Saltz, S. Yilmazel, and O. Yilmazel. Not all software engineers can become good data engineers. In *Proceedings of the 2016 IEEE International Conference on Big Data*, pages 2896–2901, 2016.

[108] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. *ACM Sigplan Notices*, pages 167–176, 2005.

[109] Rajesh S Sarkate, NV Kalyankar, and PB Khanale. Application of computer vision and color image segmentation for yield prediction precision. In *Proceedings of the 2013 International Conference on Information Systems and Computer Networks*, pages 9–13, 2013.

[110] J. Schindelin, C. T. Rueden, and M. C. et al. Hiner. "the imagej ecosystem: An open platform for biomedical image analysis". *Molecular reproduction and development*, pages 518–529, 2015.

[111] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. *Computing in Science & Engineering*, pages 61–67, 2000.

[112] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proceedings of the 2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture*, pages 269–272, 2009.

[113] J. J. Shen, C. F. Lee, and K. L. Hou. Improved productivity of mosaic image by k-medoids and feature selection mechanism on a hadoop-based framework. In *Proceedings of the 2016 International Conference on Networking and Network Applications*, pages 288–293, 2016.

[114] Eric Shook, Michael E Hodgson, Shaowen Wang, Babak Behzad, Kiumars Soltani, April Hiscox, and Jayakrishnan Ajayakumar. Parallel cartographic modeling: a methodology for parallelizing spatial data processing. *Journal of GIS*, pages 2355–2376, 2016.

[115] Y. Simmhan, R. Barga, C. van Ingen, M. Nieto-Santisteban, L. Dobos, N. Li, M. Shipway, A. S. Szalay, S. Werner, and J. Heasley. Graywulf: Scalable software architecture for data intensive computing. In *42nd Hawaii International Conference on System Sciences*, pages 1–10, 2009.

[116] Clare Sloggett, Nuwan Goonasekera, and Enis Afgan. Bioblend: automating pipeline analyses within galaxy and cloudman. *Bioinformatics*, pages 1685–1686, 2013.

[117] Calvin Smith and Aws Albarghouthi. Mapreduce program synthesis. *ACM SIGPLAN Notices*, pages 326–340, 2016.

[118] Chris Sweeney, Liu Liu, Sean Arietta, and Jason Lawrence. Hipi: A hadoop image processing interface for image-based mapreduce tasks. University of Virginia., 2011.

[119] Ann-Christine Syvänen, Katriina Aalto-Setälä, Leena Harju, Kimmo Kontula, and Hans Söderlund. A primer-guided nucleotide incorporation assay in the genotyping of apolipoprotein e. *Genomics*, pages 684–692, 1990.

[120] Alexandru Adrian Tole et al. Big data challenges. *Database Systems Journal*, pages 31–40, 2013.

[121] F. Torres. Context is king: What's your software's operating range? *IEEE Software*, pages 9–12, 2015.

[122] Feliu Trias, Valeria de Castro, Marcos López-Sanz, and Esperanza Marcos. Re-cms: a reverse engineering toolkit for the migration to cms-based web applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 810–812, 2015.

[123] Jinjun Wang, Jianchao Yang, Kai Yu, Fengjun Lv, Thomas Huang, and Yihong Gong. Locality-constrained linear coding for image classification. In *Proceedings of the 2010 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3360–3367, 2010.

[124] Devin A White and Christopher R Davis. A fully automated high-performance image registration workflow to support precision geolocation for imagery collected by airborne and spaceborne sensors. *Advances in Geocomputation*, pages 383–394, 2017.

[125] Marek S. WiewiÃşrka, Antonio Messina, Alicja Pacholewska, Sergio Maffioletti, Piotr Gawrysiak, and MichaÅĆ J. Okoniewski. Sparkseq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. *Bioinformatics*, pages 2652–2653, 2014.

[126] Byron J. Williams and Jeffrey C. Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, pages 31–51, 2010.

[127] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208, 2009.

[128] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[129] Lu Xiao, Yuanfang Cai, and Rick Kazman. Titan: A toolset that connects software architecture with quality analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 763–766, 2014.

[130] Martin Beck Xu, Min and Frank Alber. Template-free detection of macromolecular complexes in cryo electron tomograms. *Bioinformatics*, pages i69–i76, 2017.

[131] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. Processing concurrent graph analytics with decoupled computation model. *IEEE Transactions on Computers*, pages 876–890, 2017.

[132] Yuzhong Yan and Lei Huang. Large-scale image processing research cloud. *Cloud Computing*, pages 88–93, 2014.

[133] Chaowei Yang, Qunying Huang, Zhenlong Li, Kai Liu, and Fei Hu. Big data and cloud computing: innovation opportunities and challenges. *Digital Earth*, pages 13–53, 2017.

[134] Xinsheng Yang, Wei Wang, Lijie Xu, Jie liu, and Jun Wei. Mr-runner: A modularized map-reduce job management tool. In *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, page 19, 2013.

[135] Feng Yu, Casey Stella, and Kriss A Schueller. A design of heterogeneous cloud infrastructure for big data and cloud computing services. *Open Journal of Mobile Computing and Cloud Computing*, pages 1–16, 2014.

[136] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *2nd USENIX conference on Hot topics in cloud computing*, page 95, 2010.

[137] Andrzej Zalewski and Szymon Kijas. Beyond atam: Early architecture evaluation method for large-scale distributed systems. *Journal of Systems and Software*, pages 683–697, 2013.

[138] Jia Zhang, Petr Votava, Tsengdar J Lee, Owen Chu, Clyde Li, David Liu, Kate Liu, Norman Xin, and Ramakrishna Nemani. Bridging vistrails scientific workflow management system to high performance computing. In *Proceedings of the 2003 IEEE 9th World Congress on Services (SERVICES)*, pages 29–36, 2013.

[139] Zhao Zhang, Kyle Barbary, Frank Austin Nothaft, Evan Sparks, Oliver Zahn, Michael J. Franklin, David A. Patterson, and Saul Perlmutter. Scientific computing meets big data technology: An astronomy use case. In *Proceedings of the 2015 IEEE International Conference on Big Data*, pages 918–927, 2015.