Building Big Data Applications

An Analysis of Cask Data Application Platform

November 2014

White Paper

Email: bdg@qburst.com | Website: www.qburst.com

Contents

Introduction

This white paper attempts to review the *Cask Data Application Platform (CDAP)* and introduce the reader to the open-source cluster management software, *Cask Coopr*. The aim is to provide detailed information about the functionality of *CDAP*, explaining the platform stack and how the data and application virtualizations are used to develop Big Data applications. To better understand the utility of this tool, both the data and application virtualizations are exemplified with code snippets from a real application scenario.

Finally, the paper documents a list of advantages and disadvantages observed while developing an end-to-end application from a real dataset of energy disaggregation, using *CDAP* as the main programing framework and environment.

Cask Data Application Platform

About CDAP

As established in the official documentation, *CDAP* is "Virtualization for Hadoop Data and Apps." [1]

"*CDAP* brings virtualization to Hadoop data and applications. *CDAP* provides data virtualization by creating logical representations of data in datasets, and provides application virtualization through standardized containers with runtime services." [2]

CDAP provides two types of virtualizations.

Data Virtualization

This type of virtualization, as established by *Cask*, offers "logical representations of physical data as *CDAP* datasets within the *CDAP* runtime environment." [3]

The key features are:

- Streams for data ingestion.
- Reusable libraries for common Big Data access patterns.
- Data availability to multiple applications and different paradigms.

Application Virtualization

The application virtualization allows the "applications to be deployed as *CDAP* containers within the *CDAP* runtime environment." [4]

Application virtualization provides the following features:

- Integrated transactions and ingestion capabilities; processing engines that provide partitioning, ordering, and one-time execution.
- Full development lifecycle and production deployment.
- Standardization of application across programing paradigms.

CDAP makes use of four basic abstractions:

- **Streams** for real-time data collection from any external system.
- **Flows** for performing elastically scalable, real-time stream or batch processing.
- **Datasets** for storing data in simple and scalable ways without worrying about details of the storage schema.
- **Procedures** for exposing data to external systems through stored queries.

Platform Stack

The platform stack of *CDAP* provides a clear distinction between infrastructure components and application code. Primarily, it works as a middle-tier application, providing simple, high-level abstractions to gather, process, store, and query data.

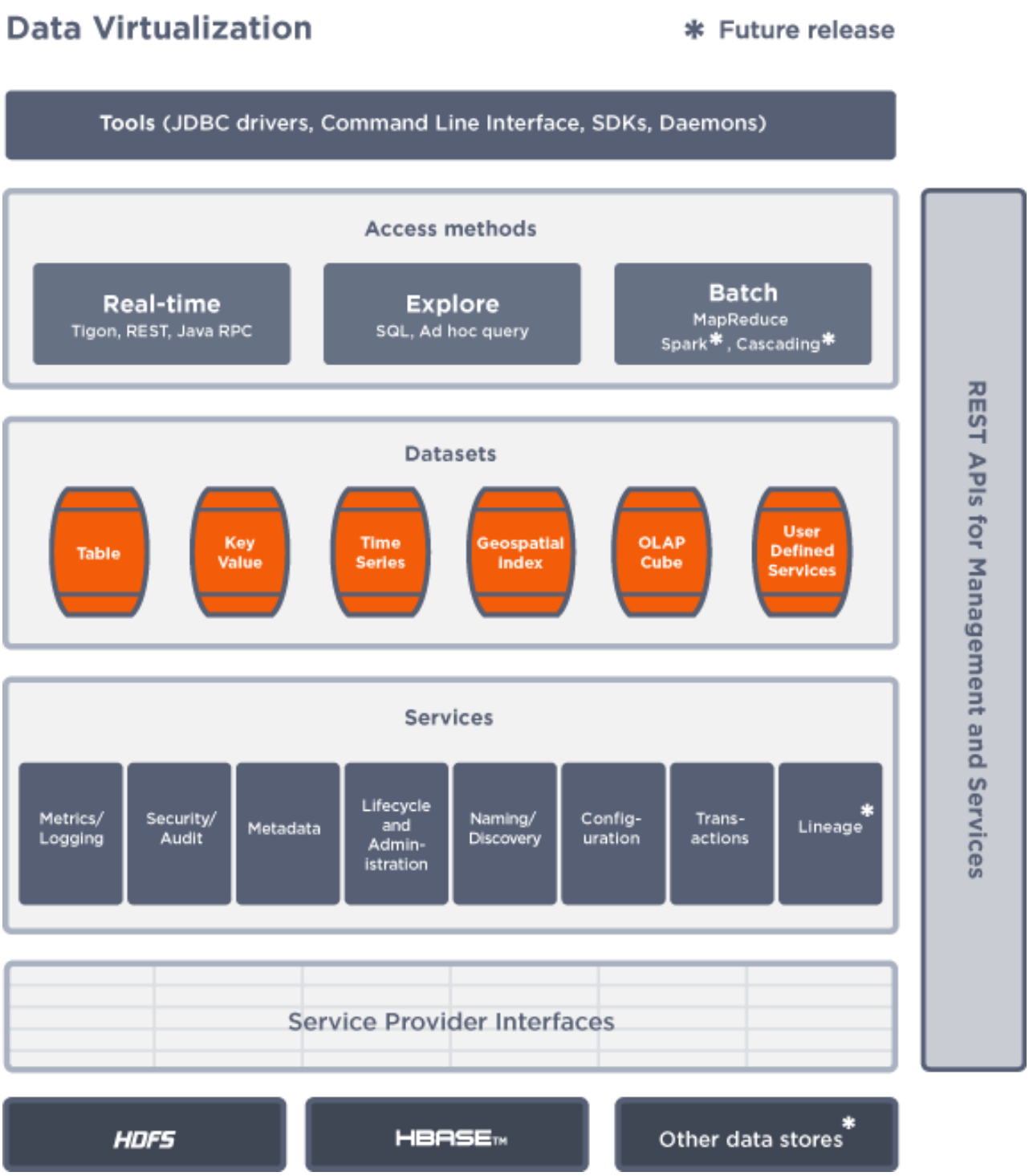

Figure 1: Cask's Data Virtualization Architecture [3] *(Source: Cask.co)*

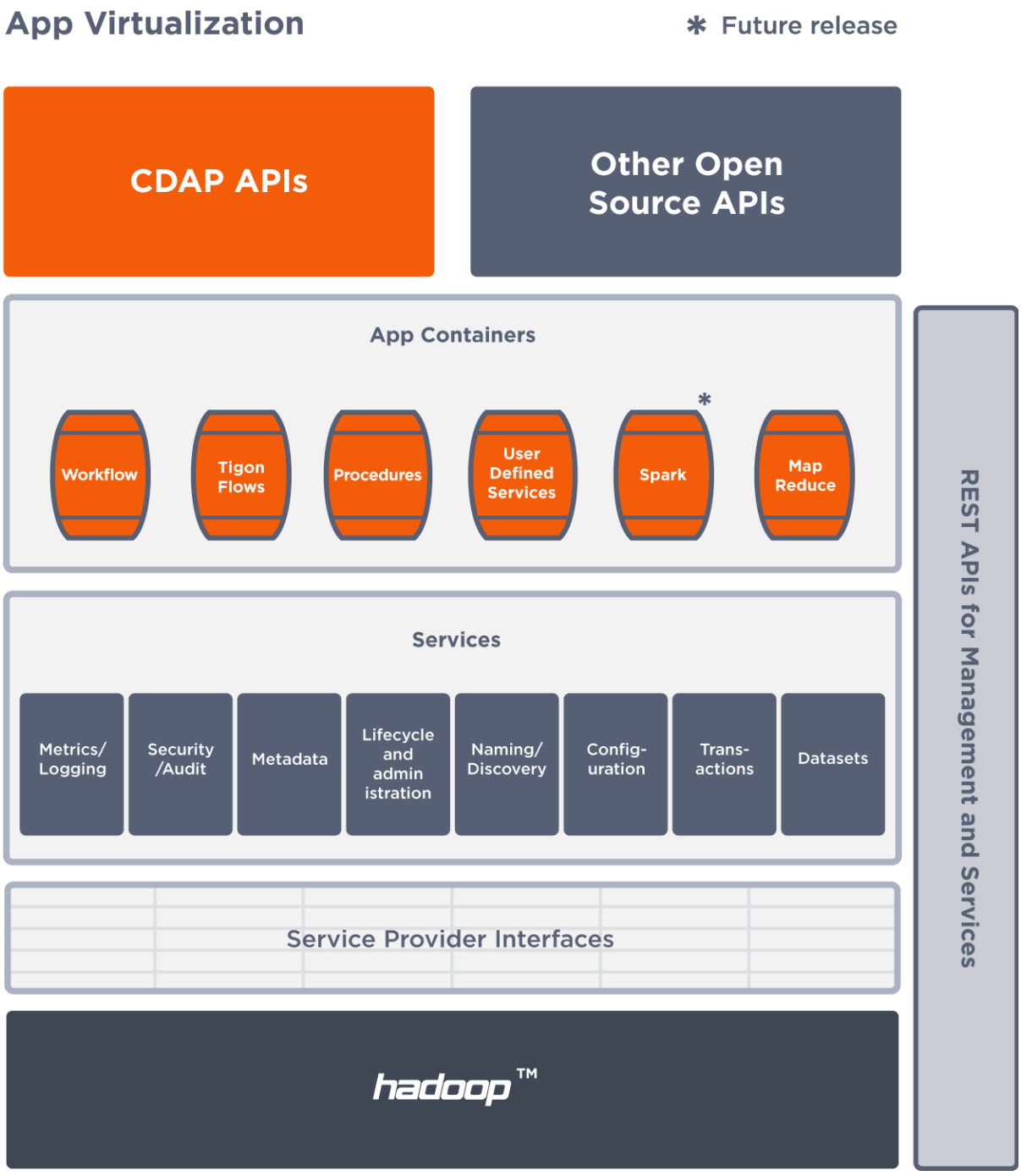

Figure 2: Cask's Application Virtualization Architecture [4] *(Source: Cask.co)*

Everything starts with *Streams; Streams* are in charge of collecting logs while *Flows* perform basic aggregation and real-time analysis. For more advanced aggregation, *MapReduce jobs* and *Workflows* are used. In fact, *Workflows* are used to execute a series of *MapReduce jobs*. *Procedures* are the main abstraction to query data. A more detailed explanation of each of these abstractions can be found in the following sections.

Creating a New Application

Let us try to understand the functionality of *CDAP* and its main abstractions in the context of a real application scenario.

Consider the problem of collecting, processing, storing, and querying data from energy disaggregation datasets, which contain values of low frequencies per device per second from a set of houses. [5]

It is important to note that *CDAP* uses *Builder-like* methods to create and configure applications. The following code creates and configures our sample application.

```
public class Application extends AbstractApplication {

    @Override
    public void configure() {
       setName("EnergyDisaggregation");
       setDescription("Energy disaggregation datasets analysis
using Cask CDAP
       addStream(new Stream("lowFreqStream"));
       createDataset("lowFreqTimeTable",
TimeseriesTable.class);
       addFlow(new LowFreqFlow());
       addProcedure(new LowFreqProcedure());
    }
}
```

This code extends from the *AbstractApplication* class, specifying the application metadata as well as declaring and configuring each of the application elements (name and description). Overriding the configure method, we can add *Streams*, create *Datasets*, and add *Procedures*.

Collecting

In order to inject data from external systems, *CDAP* makes use of *Streams*. *Streams* are specified in the application metadata as:

```
addStream(new Stream("lowFreqStream"));
```

The names used for *Streams* should be unique across *CDAP* as they are shared with all existing applications. *Streams* can be created programmatically through the management dashboard or using the command line.

Processing

Flows

Flows conform to *Flowlets*, which are the basic units in a *Flow*. Together, they build a directed acyclic graph. *Flowlets* pass data objects between one another and each of them is able to perform individual data operations. *Flows* offer ACID operations.

Given below is the implementation of the main *Flow* of the example application.

```
public class LowFreqFlow implements Flow{
    @Override
    public FlowSpecification configure() {
       return FlowSpecification.Builder.with()
               .setName("LowFreqFlow")
               .setDescription("Analyze low frequencies")
               // Processing logic is written in Flowlets
               .withFlowlets()
                   .add("parser", new
LowFreqParseFlowlet())
               // Wire the flowlets into a DAG
               .connect()
                   // Data that is sent to the Stream is
sent to the parses Flowlet
                   .fromStream("lowFreqStream").to("parse
r")
               .build();
    }
}
```

As you can see, the name, description (with or without *Flowlets),* and connections are defined before building the *Flow* itself.

Flowlets

Flowlets are the basic building blocks of a *Flow* and represent individual processing nodes within a *Flow*. *Flowlets* receive data objects as input. They can then execute data operations to finally emit data objects, if desired.

```
public class LowFreqParseFlowlet extends AbstractFlowlet{
    public static final String HOUSE_COLUMN = "house";
    public static final String DEVICE_COLUMN = "device";

    @UseDataSet("lowFreqTimeTable")
    private TimeseriesTable lowFreqTimeTable;

    // Annotation indicates that this method can process
incoming data
    @ProcessInput
    public void process(StreamEvent event) throws
ChangedCharSetException {
        Map<String,String> headers = event.getHeaders();

        String houseColumn = headers.get(HOUSE_COLUMN);
        String deviceColumn = headers.get(DEVICE_COLUMN);
        String lowFreqRow =
Charsets.UTF_8.decode(event.getBody()).toString();
        String[] lowFreqData = lowFreqRow.split("\\s+");

        lowFreqTimeTable.write(new
TimeseriesTable.Entry(Bytes.toBytes(houseColumn+":"+devic
eColumn),Bytes.toBytes(lowFreqData[1]),Long.valueOf(lowFr
eqData[0])*1000));
    }
}
```

The above code belongs to the main *Flowlet* class of the sample application; the class extends from the *AbstractFlowlet.* When a *StreamEvent* is injected, it is processed by extracting the body and in this case, creating a time-series *Table*.

There are many additional options to configure *Flowlets,* such as processing handlers with the same name or sending additional information about input data. For a deeper explanation of these concepts, please review the official documentation.

Storing

Datasets are in charge of storing and retrieving data. They provide higher-level abstractions, and generic and reusable common data patterns. These abstractions allow the users to easily perform data storage operations, avoiding the utilization of low-level APIs. The core *Dataset* is a *Table*. These *Tables* are optimized for efficient storage of semi-structured data, different schemas, or sparse data.

CDAP provides a set of useful *Datasets* such as key/values, indexed *Tables*, and time-series *Tables,* which is the one we have used in the sample application. Creation of custom datasets is also possible, when needed.

Querying

Procedures are the abstractions in charge of querying and retrieving results from *Datasets*. They make synchronous calls to *CDAP* from external systems. A *Procedure* implements and provides an API, requiring a name and a set of arguments to query the data.

Here is the implementation of the *getDailyDeviceComparison* handle. It performs a daily comparison between the devices of the *Datasets*, determining which ones spend more and which ones less.

```
  @Handle("getDailyDeviceComparison")
    public void getDailyDeviceComparison(ProcedureRequest
request, ProcedureResponder responder) throws IOException,
ParseException {
        int device = 0;

        String end = request.getArgument(END_DATE_ARG);
        String house = request.getArgument(HOUSE_ARG);

        if(end == null) {

responder.error(ProcedureResponse.Code.CLIENT_ERROR,
UNSPECIFIED_END_DATE);
            return;
        }
        else if(house == null) {

responder.error(ProcedureResponse.Code.CLIENT_ERROR,
UNSPECIFIED_HOUSE);
            return;
        }

        // Parse Begin date
        SimpleDateFormat dateFormat = new
SimpleDateFormat(DATE_FORMAT);
        Date endDate = dateFormat.parse(end);
```

```
        long time = endDate.getTime();
        List<List<TimeseriesTable.Entry>> deviceEntries = new
ArrayList<List<TimeseriesTable.Entry>>();

        for(int i=0;i<DEVICES_HOUSE_1.length;i++) {
            List<TimeseriesTable.Entry> tableEntries =
lowFreqTimeTable.read(Bytes.toBytes(house+":"+DEVICES_HOUSE_1[i
]), time - TimeUnit.MILLISECONDS.convert(1, TimeUnit.DAYS),
time);
            deviceEntries.add(tableEntries);
        }

        Map<String,Object> lowFreqMap;
        List<Map<String,Object>> lowFreqMaps = new
ArrayList<Map<String, Object>>();

        for(List<TimeseriesTable.Entry> entries :
deviceEntries) {
            lowFreqMap = new TreeMap<String, Object>();
            double freqSum = 0;
            int freqSize = 0;

            for(TimeseriesTable.Entry entry : entries) {
                freqSum = freqSum +
Double.valueOf(Bytes.toString(entry.getValue()));
                freqSize++;
            }

            if(freqSize > 0) {
                lowFreqMap.put(DEVICE_ARG,
DEVICES_HOUSE_1[device]);
                lowFreqMap.put(FREQUENCY, (freqSum /
freqSize));
            }
            else {
                lowFreqMap.put(DEVICE_ARG,
DEVICES_HOUSE_1[device]);
                lowFreqMap.put(FREQUENCY, 0.0);
            }
            lowFreqMaps.add(lowFreqMap);
            device++;
        }

        // Send response in JSON format
        responder.sendJson(ProcedureResponse.Code.SUCCESS,
lowFreqMaps);
    }
```

HTTP REST API

One of the best ways to interact with CDAP is using the *HTTP REST API*. It offers interfaces to accomplish a multitude of tasks. Given below is a brief description of the main uses of the API and the most common calls used in the sample application.

Stream

It supports the creation of *Streams*, as well as sending and reading of events.

Dataset

With the *Dataset* calls, you can list, create, delete, and truncate Datasets.

Query

The *Query* interface allows you to execute SQL queries over *Datasets* in asynchronous mode. The structure of a query should be sent as a JSON string.

```
{ "query": "<SQL-query-string>" }
```

Procedure

The *Procedure* interface allows you to call the methods of the application's Procedures.

CDAP Client

Used to deploy or delete applications and manage the lifecycle of *Flows*, *Procedures, MapReduce jobs, Workflows,* and *Custom Services*.

Logging

You can use the *Logging* interface to download the logs generated by an application.

Metrics

The *Metrics* interface can be used to check the metrics gathered during the execution of applications.

Monitor

The *Monitor* interface provides a way to inspect the status of the system services used by CDAP.

Example Calls

Inject one record:

```
curl -X POST -d "1303132930 225.57" --header "Column: mains"
http://localhost:9999/v2/streams/lowFreqStream
```

Get a specific record:

```
curl -X GET --header "Column: mains"
http://localhost:9999/v2/tables/lowFreqTable/rows/1303132930
```

Truncate a dataset:

```
curl -X POST
http://localhost:9999/v2/streams/lowFreqStream/truncate
```

Execute a procedure:

```
POST
http://localhost:9999/v2/apps/EnergyDisaggregation/procedures/LowFreqProcedure/methods/getDailyLowFreq
```

Operating Modes

CDAP can be operated in three different modes. In our example application, we have explored the standalone mode.

In memory

This is generally used while running *CDAP* unit tests. In this mode, the underlying Big Data structure is emulated using in-memory data structures.

Standalone

The local mode provides a fully operational *CDAP,* emulating the Big Data infrastructure on top of the local file system. The *CDAP* binds only to the local host, and hence is not available for remote access.

Distributed Data Application Platform

The distributed data application platform runs in a fully distributed mode. It also provides a distributed and highly available Hadoop infrastructure.

Pros and Cons

Pros	Cons
Since CDAP offers an infrastructure solution, there is no need to spend time looking for applications and configuring them individually. The time to start working on the application itself is way shorter than working with individual sets of tools.	CDAP offers data and application abstractions. However, based on our example, we believe that for more complex and custom scenarios, it would be better to use the classic approach of handling the data and tools individually, instead of abstractions.
CDAP provides simple data APIs to virtually perform all the actions required–development, production, monitoring, and maintenance. The HTTP API offers a very easy way to expose the data from CDAP.	The set of *Tables* provided lack advanced processing methods such as filtering. We noticed this while using the time-series *Table*, which offers a very simple scan method.
It provides different operating modes. So, it is possible to develop an application using the standalone mode and deploy it using the fully distributed mode to get full performance and scalability.	While injecting data into the platform, the web application showed a very low response.
The use of well-defined abstractions to handle the collection, processing, storage, and querying reduces the complexity and length of the code involved.	
It is useful to have reusable libraries for common Big Data access patterns. In our example, we used the time-series *Table*, which provided structure and methods that fitted our scenario.	
The web application provides very useful tools to track operations involved in collecting, analyzing, storing, and querying data. It offers a dashboard and monitors to track the behavior of applications.	
It is very easy to load and run a new application using the web application.	

Cask Coopr

About Coopr

As stated by the creators, the *Cask Coopr* provides "Clusters with a click." [6]

"*Cask Coopr* is a cluster management software that manages clusters on public and private clouds. Clusters created with *Coopr* utilize templates of any hardware and software stack, from simple standalone LAMP-stack servers and traditional application servers like JBoss, to full Apache Hadoop clusters comprised of thousands of nodes. Clusters can be deployed across many cloud providers (Rackspace, Joyent, and OpenStack) while utilizing common SCM tools (Chef and scripts)." [7]

Main *Cask Coopr* systems are:

- Server
- Provisioner
- UI

Server

The *Server* stores and manages metadata around providers, services, and cluster templates. It exposes and handles the web services supported by *Cask Coopr*.

Provisioners

The *Provisioners* take queued tasks and execute them. Once the tasks are completed, the *Provisioners* report the task status back to the *Server.*

UI

The *UI* has two main views: Admin and User. The *Admin UI* allows system administrators to configure providers, disk images, machine hardware types, and software services.

The *User UI* permits the creation of instances of clusters from the available cluster templates. The user has the ability to perform create, delete, amend, update, and monitor operations.

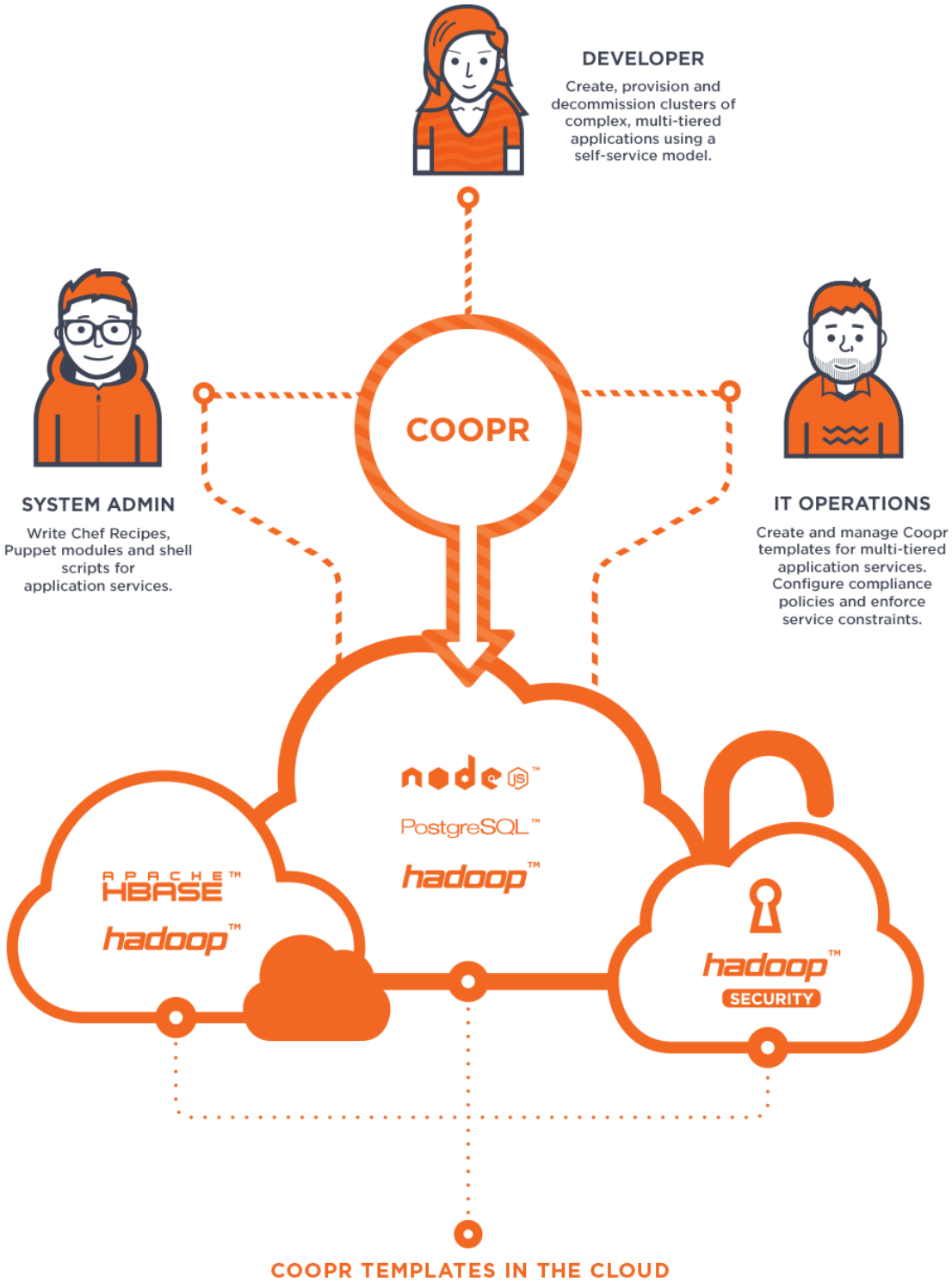

Figure 3: Cask's Coopr Workflow [6] *(Source: Cask.co)*

Architecture

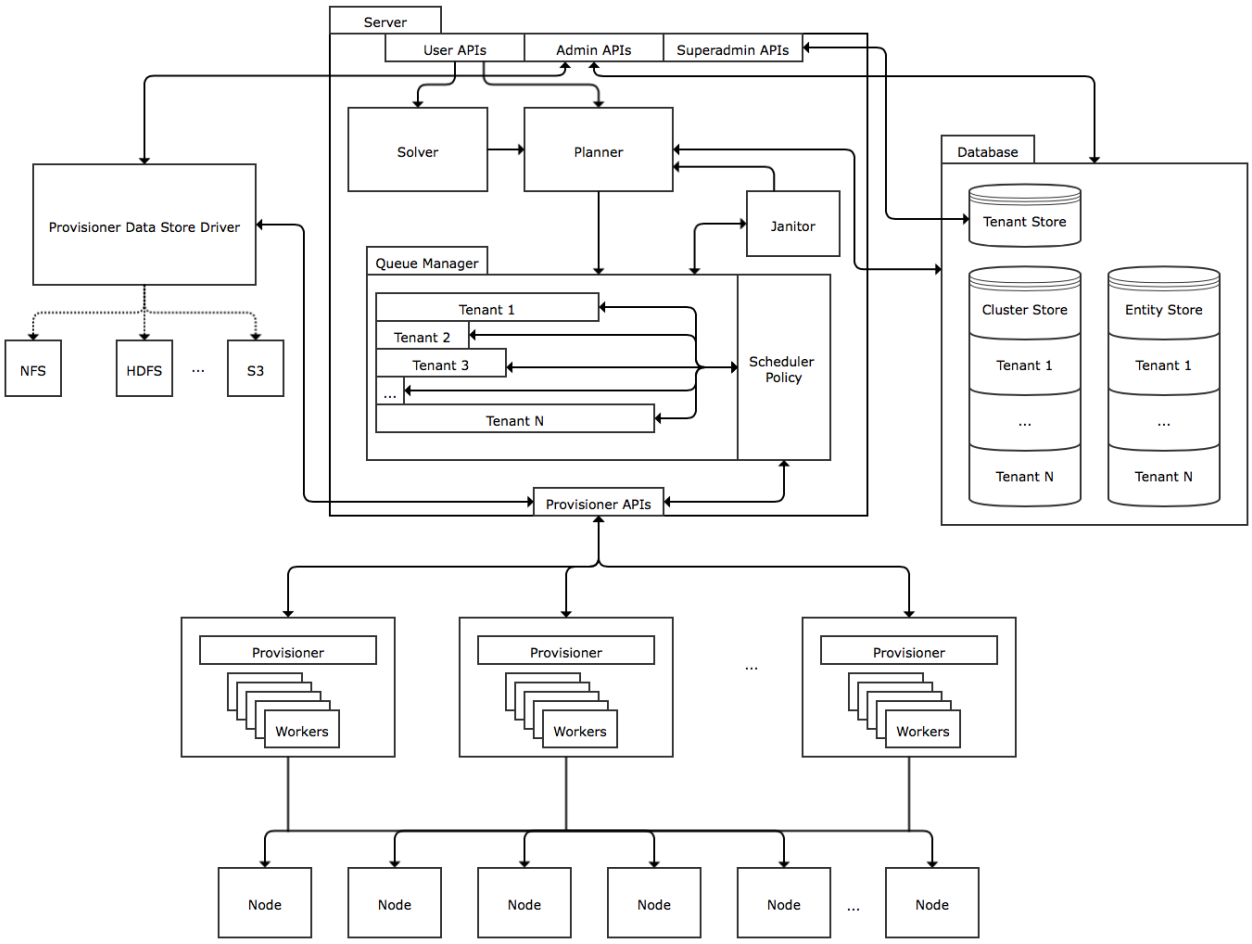

Figure 4: Cask's Coopr Architecture [7] *(Source: Cask.co)*

Conclusion

CDAP introduces a new approach to Big Data application development, making use of data and application abstractions instead of individually handling the underlying data and technologies involved.

We discovered that *CDAP* enables existing teams to start creating Big Data applications faster. This is because the virtualizations provided negate the need for learning the complexities of each technology. Teams can also focus more on solving business challenges, as they do not need to invest a lot of time learning the tools involved.

However, we believe that for very specific and custom scenarios that require advanced functionalities and complex data operations, it would be better to use the classic approach of accessing data and using tools individually.

References

[1] Cask (2014). *Cask Data Application Platform*. http://cask.co/products/#cdap

[2] Cask (2014). *CDAP Whitepaper*. http://docs.cask.co/collateral/current/cdap-whitepaper-1.0.0.pdf

[3] Cask (2014). *Cask Data Application Platform*. Data Virtualization. http://cask.co/products/#cdap

[4] Cask (2014). *Cask Data Application Platform*. App Virtualization. http://cask.co/products/#cdap

[5] REDD (2011). *The Reference Energy Disaggregation Dataset*. http://redd.csail.mit.edu/

[6] Cask (2014). *Cask Coopr*. http://cask.co/products/#coopr

[7] Cask (2014). *Cask Coopr Architecture*. http://docs.cask.co/coopr/current/en/overview/architecture.html

This document is published for educational purposes only. All trademarks, service marks, trade names, product names and logos appearing in this document are the property of their respective owners. QBurst is not liable for any infringement of copyright that may arise while making this document available for public viewership. If you believe that your copyright is being violated, please contact us promptly so that we may take corrective action.