OGC Testbed-15

Federated Clouds Analytics Engineering Report

Table of Contents

Publication Date: 2019-12-19

Approval Date: 2019-11-22

Submission Date: 2019-10-18

Reference number of this document: OGC 19-026

Reference URL for this document: http://www.opengis.net/doc/PER/t15-D020

Category: OGC Public Engineering Report

Editor: Pedro Gonçalves

Title: OGC Testbed-15: Federated Clouds Analytics Engineering Report

---

OGC Public Engineering Report

COPYRIGHT

WARNING

LICENSE AGREEMENT

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER’S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR’s sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to

indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Chapter 1. Subject

This OGC Engineering Report (ER) documents the results and experiences resulting from the Federated Cloud Analytics task of OGC Testbed-15. More specifically, this ER provides an analysis of:

- The potential for the OGC Web Processing Service (WPS) Interface Standard as an Application Programming Interface (API) to a workflow automation service for managing job execution involving multiple containers in the Scale Data Center Environment;
- Using an implementation of the OGC WPS standard as a general frontend to workflow automation with containers;
- The suitability of the OGC WPS 2.0 standard as an API for Cloud analytics;
- Using OGC Web Services (WS) as analytics data sources and sinks.

Chapter 2. Executive Summary

The work documented in this Engineering Report addresses a broader question on how to leverage Cloud architectures managing automated processing on a cluster of machines combined with using OGC standards.

Focusing on the SCALE Data Center Environment, the ER explores how the OGC WPS 2.0 Standard can be used as a standard API for Cloud analytics for workflow automation.

SCALE was developed at the National Geospatial-Intelligence Agency (NGA) and is targeted for on-demand, near real-time, automated processing of large datasets. SCALE provides management of automated processing on a cluster of machines, allowing users to define jobs that can be any type of script or algorithm, execute them and generate product files.

Currently, SCALE only supports data processing applications compiled for Linux with no user interaction. These data processing applications (called Jobs) are configured as workflows (called Recipes) of simple tasks that consume inputs and produce outputs. Each job performs a single data processing task by creating and executing a process in a Docker container. During task execution, SCALE manages the container creation, deployment and result retrieval processes.

The Testbed-15 Federated Cloud Analytics task deployed a Scale Datacenter Environment and this ER documents the installation, deployment and execution of data processing applications in the SCALE Datacenter environment using an OGC WPS interface. The data processing application can access other OGC services, such as WMS, WFS and WCS, to serve as data sources and sinks. The data discovery process explored in this activity was limited to using WCS but other methods such as the SpatioTemporal Asset Catalog (STAC) or OpenSearch should be investigated in future activities.

This ER clarifies how SCALE jobs can be mapped to WPS processes in a straightforward way enabling a standardized description and execution of SCALE jobs and documents some issues to be addressed concerning complex inputs and literal outputs. The experiments documented in this ER also show that different approaches to execute workflows with SCALE jobs wrapped in WPS processes exist. Future initiatives should investigate these approaches by creating different SCALE jobs that could either be combined to SCALE recipes or executed as single WPS processes.

Most importantly, during this activity the job interface specification (SEED) was used to package job input/output parameters metadata with Docker images that contain discrete processing algorithms. This enables developers to prepare the software in a self-contained package containing all execution dependencies, deploy and execute it in a hosted environment with access to data. The SEED specification has shown a great potential for supporting the discovery and consumption of discrete units of work contained within a Docker image and should be further evaluated in ensuing activities.

2.1. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Contacts

Name	Organization	Role
Pedro Gonçalves	Terradue	Editor
Ziheng Sun	George Mason University	Contributor
Benjamin Pross	52°North	Contributor
Peter Baumann	rasdaman	Contributor

2.2. Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 3. References

The following normative documents are referenced in this document.

NOTE: Only normative standards are referenced here, e.g. OGC, ISO or other SDO standards. All other references are listed in the bibliography.

- OGC: OGC 06-121r9, OGC® Web Services Common Standard (2010) [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2]
- OGC: OGC® WPS 2.0.2 Interface Standard Corrigendum 2 (2018) [http://docs.opengeospatial.org/is/14-065/14-065.html]

Chapter 4. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2] shall apply. In addition, the following terms and definitions apply.

SCALE

Scale Data Center Environment, an open source system that provides management of automated processing on a cluster of machines.

SEED

Seed specification to aid in the discovery and consumption of a discrete unit of work contained within a Docker image by the National Geospatial-Intelligence Agency (NGA)

Container (in the context of containerization)

A container is a standard unit of software that packages up code and all of its dependencies, so that the application runs more quickly and reliably from one computing environment to another.

4.1. Abbreviated terms

- API Application Programming Interface
- CSW Catalogue Service for the Web
- DC/OS Distributed Cloud Operating System
- ER Engineering Report
- JSON JavaScript Object Notation
- NFS Network File System
- REST Representational State Transfer
- STAC Spatio-Temporal Asset Catalogue
- UI User Interface
- URL Uniform Resource Locator
- WPS Web Processing Service
- WS Web Services

Chapter 5. Overview

Section 6 introduces the Scale Data Center Environment for on-demand, near real-time, automated processing of large datasets (satellite, medical, audio, video, …). This section provides a short introduction for the potential of the Scale Data Center for rapidly integrating algorithms written in several programming language (e.g. Java, Python, IDL, Matlab, C/C++) and composing complex algorithms using recipes for advanced data processing.

Section 7 discusses the potential for the OGC WPS Interface Standard as an API to a workflow automation service for managing job execution involving multiple containers in the Scale Data Center Environment.

Section 8 presents the entire workflow from installation, deployment and execution of data processing applications in the Scale Datacenter environment using OGC WPS interface.

Section 9 provides a summary of the main findings and provides further recommendations to advance the architecture, integration and implementation strategies for use of OGC standards in the Scale Data Center Environment.

This ER refers to WPS and OGC API – Processes interchangeably. This is because the OGC API – Processes draft specification emerged from the draft specification of the REST binding of the WPS standard.

Note that this ER includes blocks of text with the title ‘keypoints’ to highlight significant points to be noted by the reader.

For readability this ER will refer to the Scale Data Center Environment as SCALE (in uppercase) and to the Seed specification as SEED (in uppercase).

Chapter 6. SCALE Data Center Environment

SCALE Data Center Environment, developed at the National Geospatial-Intelligence Agency (NGA), is targeted for on-demand, near real-time, automated processing of large datasets (satellite, medical, audio, video, …) allowing the rapid integrating of algorithms written in several programming language (e.g. Java, Python, IDL, Matlab, C/C++) and composing complex algorithms using recipes for advanced data processing.

SCALE is a system that provides management of automated processing on a cluster of machines. SCALE allows users to define jobs, which can be any type of script or algorithm. These jobs run on ingested source data and generate product files. The generated products can be disseminated to appropriate users and/or used to evaluate the producing algorithm in terms of performance and accuracy.

SCALE runs across a cluster of networked machines (called nodes) that process the jobs. Algorithm execution is seamlessly distributed across thousands of CPU cores with Docker providing algorithm containerization and Cluster management software to enable optimum resource utilization. While SCALE can be entirely run on a pure Apache Mesos [1] cluster, it is strongly recommended using the Data Center Operating System (DC/OS). DC/OS (the Distributed Cloud Operating System) [2] is an open-source, distributed operating system based on the Apache Mesos distributed systems kernel. DC/OS manages multiple machines in the cloud or on-premises from a single interface and:

- Deploys containers, distributed services, and legacy applications into those machines; and
- Provides networking, service discovery and resource management to keep the services running and communicating with each other.

The main advantages of DC/OS are to provide service discovery, load-balancing and fail-over for SCALE, as well as deployment scripts for target infrastructures. This stack allows SCALE users to focus on use of the framework while minimizing effort spent on deployment and configuration.

6.1. Authentication

The SCALE web server is built using Django and the Django REST Framework [3]. This approach should in theory allow several authentication mechanisms to be employed. The current deployment, however, seems to be primarily deployed behind a firewall or only available from behind the DC/OS Admin User Interface (UI). SCALE development documentation [1] mentions significant development effort for version 6.0 to enforce authentication to all API endpoints. SCALE currently has limited authorization granularity, only recognizing two levels of access (Authenticated and Staff) and supports the following authentication mechanisms:

- Username / password: The default deployment of SCALE with a single superuser account admin that can add new users through the Django user authentication system.
- GEOAxIS: The National Geospatial Intelligence Agency (NGA) Enterprise Identity and Access management [4]
- Token: Targeted for API clients that prefer to issue a long-lived token as opposed to utilizing cookies set after a login event. An API token may be issued through either a username / password or PKI certificate.

Keypoints	Authentication with username / password and tokens for API access

6.2. Algorithm Development

SCALE is designed to allow development of recipes and jobs for specific tasks while disregarding the complexities of cluster scheduling or data flow management. As long as the processing can be accomplished with discrete inputs on a Linux command line, the processing workflow should be able to run in SCALE.

The SCALE system utilizes Docker containers to run algorithms in an isolated environment considering the following restrictions:

1. Run standalone without any user inputs: algorithms must be fully automated.
2. Fail gracefully: algorithms must capture system faults and failures and report an exit code, as well as log an informative message about the error. If not captured appropriately, failures will appear as a general algorithm error.
3. Not display popups
4. Run on Linux: any external libraries needed must be compiled for Linux.
5. Not have hardcoded paths: necessary file paths should be passable into the algorithm either via a configuration file or passed from the command line parameters.

For the algorithm, SCALE provides the absolute paths of inputs files, an empty output directory, and the requested dedicated resources. SCALE also captures standard output, standard error and exit codes. However, some algorithm modifications might be needed considering that SCALE does not resolve relative paths or provide output file names. Also, SCALE does not automatically create NFS mounts in the Docker container or capture output products not listed in the results manifest and job interface.

The algorithms can be created in several languages or frameworks with a few important considerations:

- C/C++: Compiled on Linux and should provide cmake/makefiles for algorithm
- IDL : Code should be compiled into .sav files with IDL's save command (running with the runtime license) with limited usage of some function calls (e.g. Harris Geospatial's ENVI software [5]) that require special licensing.
- Java: Compiled into .jar files with additional .jar libraries in its own folder
- MATLAB [6]: Compiled into an executable using MATLAB's deploytool [7] or mcc command [8] (using MATLAB's compiled runtime mode) with all required toolboxes specified and available at compile time
- Python: Code in its own folder and needed Python modules previously installed in the Docker container

To facilitate the algorithm integration, preparing a wrapper shell script is advisable to:

- Mount NFS directories for the algorithm to reference

- Setup additional environment variables or append to system paths
- Determine additional command line input arguments for the algorithm

The algorithm (or the algorithm's wrapper script) must generate a JSON document describing the results to convey which products should be archived by SCALE and passed onto other algorithms. This JSON document describes the outputs names, paths and data type. The JSON document also includes geospatial and temporal (start and end date) metadata and optionally can point to a GeoJSON file to store additional information about the result item.

Keypoints	Applications must be compiled for Linux with no user interaction and produce a result manifest with metadata limited to date and geospatial box

6.3. Algorithm Container

Once a standalone algorithm is capable of generating a results manifest, a container can be created.

A container is a standard unit of software that packages up code and all of its dependencies, so that the application runs more quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine. Containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Docker containers should be as small as possible. The Docker containers are pulled and cached on the host the first time it is used and will update when the cache no longer matches the Docker registry. Excessively large files will unnecessarily fill up the host machine's disk space requiring the host machine's entire cache to be reset.

Docker containers in SCALE normally have no knowledge of other containers running and cannot share resources or data across containers. By default, the Docker containers will not have access to files or NFS mounts on the host machine, unless a predefined persistent disk is mounted to the containers. Nevertheless, the output of a container can be tied to the input of another container using recipes and the outputs defined in the algorithm's results manifest file. Files are conveyed among containers following the configuration in SCALE recipes, and the final results will be kept according to the information in results manifest. If the recipe and result manifest are missing, there will be no result files being stored after the jobs are done.

Keypoints	SCALE jobs should use recipe to be linked into a workflow to pass on their outputs to others input variables. Result manifests must be created to get the final results.

6.4. Job Types and Jobs

Jobs represent the various algorithms or units of work that get executed in SCALE. The jobs are created from definitions named *job-type*. *job-type* is a JSON structure that defines the interface for executing the job's algorithm. The JSON document describes the algorithm's inputs and outputs, the docker image that contains the job algorithm, as well as the command line details for how to invoke the algorithm. *job-type* definitions are created by submitting the JSON document to SCALE API through HTTP POST actions. Jobs are running instances of Job Type that are triggered by calling the SCALE `new-job` HTTP API.

Here's an example *job-type* definition:

```
{
    "name": "read-bytes",
    "version": "1.0.0",
    "title": "Read Bytes",
    "description": "Reads x bytes of an input file and writes to output dir",
    "category": "testing",
    "author_name": "John_Doe",
    "author_url": "http://www.example.com",
    "is_operational": true,
    "icon_code": "f27d",
    "docker_privileged": false,
    "docker_image": "geoint/read-bytes",
    "priority": 230,
    "timeout": 3600,
    "max_scheduled": null,
    "max_tries": 3,
    "cpus_required": 1.0,
    "mem_required": 1024.0,
    "disk_out_const_required": 0.0,
    "disk_out_mult_required": 0.0,
    "interface": {
        "output_data": [
            {
                "media_type": "application/octet-stream",
                "required": true,
                "type": "file",
                "name": "output_file"
            }
        ],
        "shared_resources": [],
        "command_arguments": "1024 ${input_file} ${output_file}",
        "input_data": [
            {
                "media_types": [
                    "application/octet-stream"
                ],
                "required": true,
                "partial": true,
```

```
                "type": "file",
                "name": "input_file"
            }
        ],
        "version": "1.1",
        "command": ""
    },
    "error_mapping": {
        "version": "1.0",
        "exit_codes": {}
    },
    "trigger_rule": null
}
```

Important element definitions:

Job Type Definition Element	Description
docker_image	The container image with the algorithm
interface.input_data.name	Name of the input data parameter that is used to chain SCALE workflows
interface.output_data.name	Name of the output data parameter that is used to chain SCALE workflows

Keypoints	The SCALE job type is defined by submitting a JSON document to the SCALE *job-type* API. Job types can then be chained together and triggered to create processing jobs.

6.5. Recipe Types and Recipes

Recipes represent a graph/workflow of jobs that allow jobs to depend upon one another and for files produced by one job to be fed as input into another job. Recipes are instantiated from recipe types. Recipe types are created by submitting a JSON document to the SCALE *recipe-type* API. Here's an example of recipe type definition.

```
{
    "definition": {
        "input_data": [
            {
                "media_types": [
                    "application/octet-stream"
                ],
                "name": "input_file",
                "required": true,
                "type": "file"
            }
        ],
        "jobs": [
            {
                "dependencies": [],
                "job_type": {
                    "name": "read-bytes",
                    "version": "1.0.0"
                },
                "name": "read-bytes",
                "recipe_inputs": [
                    {
                        "job_input": "input_file",
                        "recipe_input": "input_file"
                    }
                ]
            }
        ]
    },
    "description": "Read x bytes from input file and save in output dir",
    "name": "read-byte-recipe",
    "title": "Read Byte Recipe",
    "version": "1.2.0"
}
```

Important element definitions:

Job Type Definition Element	Description
input_data.name	The name of an input object from a workspace
jobs	List of jobs and their input/output chains in this recipe

Keypoints	Jobs can be composed into workflows called recipes which are specified with a JSON document and *recipe-type* SCALE API. SCALE will chain inputs and outputs of several jobs in a recipe.

6.5.1. Workspaces

Workspaces in SCALE represent storage areas for job/recipe inputs and outputs. Separate workspaces are normally used for saving input and output of SCALE workflows respectively, so that the new generated files do not trigger another round of workflow execution by listening for changes to the input workspace. NFS backed workspaces allow distributed storage to be used by multiple SCALE nodes.

```
{
    "description": "workspace-input",
    "json_config": {
        "broker": {
            "host_path": "/mnt/nfsshare/input",
            "type": "host"
        }
    },
    "name": "workspace-input",
    "title": "workspace-input",
    "base_url": "http://cloud.csiss.gmu.edu/scale_files"
}
```

6.6. Job Type Standardization

To support the discovery and consumption of a discrete unit of work contained within a Docker image, the NGA Research developed a general standard named SEED.

Supported by new version of SCALE (version 6), the job types have a SEED manifest that takes the place of the old job interface Job Interface. Likewise, the SEED manifest describes how to run a job: What inputs it expects, what outputs it produces, and how to invoke the algorithm and complements it with a definition of the runtime processing, memory and storage requirements of a discrete unit of work.

A SEED image is essentially the following:

- A Docker Image
- Follows a naming convention of `[name]-[version]-seed`, e.g. `snapapp-1.1.1-seed`
- Contains a SEED manifest as Docker label

The SEED standard is intended to support both simple and complex job packaging. To that end the standard allows for sensible defaults to take the place of a fully specified manifest. The following examples identify both a minimal Seed use and a more realistic, fully exercised standard.

The following JSON shows a minimal Seed manifest demonstrating the simplest possible Seed definition for a Random Number Generator Job.

```
{
  "seedVersion": "1.0.0",
  "job": {
    "name": "random-number-gen",
    "jobVersion": "0.1.0",
    "packageVersion": "0.1.0",
    "title": "Random Number Generator",
    "description": "Generates a random number and outputs on stdout",
    "maintainer": {
      "name": "John Doe",
      "email": "jdoe@example.com"
    },
    "timeout": 10
  }
}
```

The previous manifest would be serialized as a label in a Dockerfile snippet like

```
FROM alpine

ENTRYPOINT /app/job.sh

LABEL
com.ngageoint.seed.manifest="{\"seedVersion\":\"1.0.0\",\"job\":{\"name\":\"random-
number-gen\",\"jobVersion\":\"0.1.0\",\"packageVersion\":\"0.1.0\",\"title\":\"Random
Number Generator\",\"description\":\"Generates a random number and outputs on
stdout\",\"maintainer\":{\"name\":\"John
Doe\",\"email\":\"jdoe@example.com\"},\"timeout\":10}}"
```

A more complex example would be Image watermark job taking a single image and returning with watermark applied using SEED definition.

```
{
  "seedVersion": "1.0.0",
  "job": {
    "name": "image-watermark",
    "jobVersion": "0.1.0",
    "packageVersion": "0.1.0",
    "title": "Image Watermarker",
    "description": "Processes an input PNG and outputs watermarked PNG.",
    "maintainer": {
      "name": "John Doe",
      "email": "jdoe@example.com"
    },
    "timeout": 30,
    "interface": {
      "command": "${INPUT_IMAGE} ${OUTPUT_DIR}",
      "inputs": {
```

```
        "files": [
          {
            "name": "INPUT_IMAGE"
          }
        ]
      },
      "outputs": {
        "files": [
          {
            "name": "OUTPUT_IMAGE",
            "pattern": "*_watermark.png"
          }
        ]
      }
    },
    "resources": {
      "scalar": [
        {
          "name": "cpus",
          "value": 1
        },
        {
          "name": "mem",
          "value": 64
        }
      ]
    },
    "errors": [
      {
        "code": 1,
        "name": "image-Corrupt-1",
        "description": "Image input is not recognized as a valid PNG.",
        "category": "data"
      },
      {
        "code": 2,
        "name": "algorithm-failure"
      }
    ]
  }
}
```

The Seed would be serialized as a label in a Dockerfile snippet as:

```
FROM alpine

ENTRYPOINT /app/watermark.py

LABEL
com.ngageoint.seed.manifest="{\"seedVersion\":\"1.0.0\",\"job\":{\"name\":\"image-
watermark\",\"jobVersion\":\"0.1.0\",\"packageVersion\":\"0.1.0\",\"title\":\"Image
Watermarker\",\"description\":\"Processes an input PNG and outputs watermarked
PNG.\",\"maintainer\":{\"name\":\"John
Doe\",\"email\":\"jdoe@example.com\"},\"timeout\":30,\"interface\":{\"command\":\"\${I
NPUT_IMAGE}
\${OUTPUT_DIR}\",\"inputs\":{\"files\":[{\"name\":\"INPUT_IMAGE\"}]},\"outputs\":{\"fi
les\":[{\"name\":\"OUTPUT_IMAGE\",\"pattern\":\"*_watermark.png\"}]}},\"resources\":{\
"scalar\":[{\"name\":\"cpus\",\"value\":1},{\"name\":\"mem\",\"value\":64}]},\"errors\
":[{\"code\":1,\"name\":\"image-Corrupt-1\",\"description\":\"Image input is not
recognized as a valid PNG.\",\"category\":\"data\"},{\"code\":2,\"name\":\"algorithm-
failure\"}]}}"
```

Worth noticing that SEED supports arbitrary parameter outputs where JSON keys will be mapped to specific output properties, including definition of optional/mandatory, parameter names, types (string, number, etc). Contrary to SCALE, SEED allows the captures resulting outputs by file name pattern, supporting multiple files and append the appropriate mime type automatically. This feature removes the need for the SCALE result manifest, and the output files only need to be placed into the correct (and provided) path in the container.

Keypoints	SEED is a new specification of the SCALE **Job Type** description and metadata with potential for independent usage in other environments to support the discovery and consumption of a discrete unit of work contained within a Docker image.

Chapter 7. OGC Web Processing Service and Scale

This section describes how SCALE jobs and recipes can be wrapped as Web Processing Service (WPS) processes.

The mapping between SCALE and WPS is described using the following WPS 2.0 operations:

- GetCapabilities
- DescribeProcess
- Execute
- GetStatus
- GetResult

The draft OGC API - Processes specification was used in Testbed-15. The Processes API maps the WPS 2.0 operations to REST endpoints and uses a JSON encoding for requests and responses.

7.1. GetCapabilities

The GetCapabilities operation of a WPS instance returns general service information and the available processes. In the OGC API - Processes this operation returns a set of links, for example the link to the list of processes.

7.1.1. Mapping to SCALE

The SCALE datacenter defines two types of resources that are suitable to be mapped as WPS processes:

- Jobs
- Recipes

The SCALE documentation states that "Jobs represent the various algorithms or units of work that get executed in Scale. Recipes represent a graph/workflow of jobs that allow jobs to depend upon one another and for files produced by one job to be fed as input into another job." [1: https://ngageoint.github.io/scale/docs/architecture/jobs/index.html, accessed 9/3/2019]

SCALE jobs and recipes are described by job types and recipe types.

As at the time of writing this ER there were no suitable SCALE recipe types available. Therefore, this ER concentrates on describing the mapping between SCALE job types and WPS processes.

A list of available SCALE job types can be retrieved using the following endpoint URL:

```
{SCALE_base_URL}/job-types/
```

This will return a list of job types encoded in JSON. An abbreviated example is shown below:

```
{
    "count": 2,
    "next": null,
    "previous": null,
    "results": [
        {
            "id": 10,
            "name": "curl",
            "version": "1.0",
            "title": "Curl Downloader",
            "description": "The working Curl Downloader",

            ...

        },
        {
            "id": 14,
            "name": "imagemagick-json",
            "version": "1.0",
            "title": "imagemagick-json",
            "description": "A job running ImageMagick and outputting resulting pixel
data as JSON array",

            ...

        }
    ]
}
```

The complete description of the elements of a job type can be found in the job type SCALE documentation [2: https://ngageoint.github.io/scale/docs/rest/v6/job_type.html#v6-job-type-list, accessed 9/3/2019].

The following table shows the mapping between the WPS process summary and the SCALE job type description.

Table 1. Mapping between WPS process summary and SCALE job type

Process summary element	**SCALE Job type element**
id	`Composition of job type elements: name, version and revision`
title	`title`

Process summary element	SCALE Job type element
description	`description`
version	`Composition of job type elements: version and revision`

The WPS API endpoint URL to obtain a list of processes is as follows:

```
{WPS_base_URL}/processes/
```

A complete example of process summaries encoded in JSON is shown below:

```
[
  {
    "id": "scale-algorithm-curl-1.0-r2",
    "title": "Curl Downloader",
    "description": "The working Curl Downloader",
    "version": "1.0-r2",
    "jobControlOptions": [
      "async-execute",
      "sync-execute"
    ],
    "outputTransmission": [
      "value",
      "reference"
    ],
    "links": [
      {
        "href": "/scale-algorithm-curl-1.0-r2",
        "rel": "process description",
        "type": "application/json",
        "title": "Process description"
      }
    ]
  },
  {
    "id": "scale-algorithm-imagemagick-json-1.0-r1",
    "title": "imagemagick-json",
    "description": "A job running ImageMagick and outputting resulting pixel data as
JSON array",
    "version": "1.0-r1",
    "jobControlOptions": [
      "async-execute",
      "sync-execute"
    ],
    "outputTransmission": [
      "value",
      "reference"
    ],
    "links": [
      {
        "href": "/scale-algorithm-imagemagick-json-1.0-r1",
        "rel": "process description",
        "type": "application/json",
        "title": "Process description"
      }
    ]
  }
]
```


The elements *jobControlOptions* and *outputTransmission* are added automatically by the WPS server. The mapping of WPS job control options and output transmission modes to SCALE is

described in more detail in sections Execute, GetStatus and GetResult.

7.2. DescribeProcess

The DescribeProcess operation of a WPS instance returns more detailed information about a process.

7.2.1. Mapping to SCALE

As mentioned in the previous section, SCALE jobs are described by job types. A description for a single job type can be retrieved using the following endpoint URL:

```
{SCALE_base_URL}/job-types/{job-type-id}
```

The job type id can be found in the list of job types (see previous section).

An abbreviated example of the job type response is shown below:


```
{
    "id": 10,
    "name": "curl",
    "version": "1.0",
    "title": "Curl Downloader",
    "description": "The working Curl Downloader",
    "category": "test",
    "author_name": "Alexander Lais",
    "author_url": "https://www.solenix.ch",
    "is_system": false,
    "is_long_running": false,
    "is_active": true,
    "is_operational": true,
    "is_paused": false,
    "icon_code": "f0ed",
    "uses_docker": true,
    "docker_privileged": false,
    "docker_image": "alaisslxogc/scale-curl:latest",
    "revision_num": 2,

    ...

    "created": "2019-08-30T10:02:54.140366Z",
    "archived": null,
    "paused": null,
    "last_modified": "2019-08-30T14:03:14.708923Z",
    "interface": {
        "settings": [],
        "output_data": [
            {
```

```
                "required": true,
                "type": "file",
                "name": "output_file"
            }
        ],
        "env_vars": [],
        "shared_resources": [],
        "command_arguments": "${url} ${job_output_dir}",
        "input_data": [
            {
                "required": true,
                "type": "property",
                "name": "url"
            }
        ],
        "version": "1.4",
        "command": "/entrypoint.sh",
        "mounts": []
    },

    ...
}
```

The complete list of elements can be found on the version 5.4 of SCALE Job Type documentation [3: https://github.com/ngageoint/scale/blob/5.4.0/scale/docs/rest/job_type.rst, accessed 9/3/2019].

The single job type description has some elements from the shorter description in the job type list (see previous section). More importantly, the single job type description has the additional elements *input_data* and *output_data*. These are needed to complete the WPS process description.

WPS specifies three different input and output types:

- literal,
- complex and
- bounding box data

For more information about these types, please refer to the WPS 2.0 standard, section 7.3: Data Types [4: http://docs.opengeospatial.org/is/14-065/14-065.html#21]

The following tables show the mapping between the WPS processes and the SCALE job type description regarding inputs and output types:

Table 2. Mapping between WPS process and SCALE job type inputs

WPS input type	**SCALE Job type input data**
literal	`"type": "property"`
complex	`"type": "file " | "type": "files"`
bounding box	`not mapped`

Table 3. Mapping between WPS process and SCALE job type outputs

WPS output type	SCALE Job type output data
literal	`not mapped`
complex	`"type": "file " | "type": "files"`
bounding box	`not mapped`

The WPS endpoint URL to obtain a process description is as follows:

```
{WPS_base_URL}/processes/{process-id}
```

The process id needs to exist in the list of processes (see [PROCESS_LIST]).

An example of a process description encoded in JSON is shown below:

```
{
  "id": "scale-algorithm-curl-1.0-r2",
  "title": "Curl Downloader",
  "description": "The working Curl Downloader",
  "version": "1.0-r2",

  ...

  "inputs": [
    {
      "id": "url",
      "title": "url",
      "description": "url",
      "input": {
        "literalDataDomains": [
          {
            "valueDefinition": {
              "anyValue": true
            }
          }
        ]
      },
      "minOccurs": 1,
      "maxOccurs": 1
    }
  ],
  "outputs": [
    {
      "id": "output_file",
      "title": "output_file",
      "description": "output_file",
      "output": {
        "formats": [
          {
            "default": true,
            "mimeType": "application/octet-stream"
          },
          {
            "default": false,
            "mimeType": "application/octet-stream",
            "encoding": "base64"
          }
        ]
      }
    }
  ]
}
```

The *url* input data element from the SCALE job type description is mapped to a literaldata element.

The WPS literal input type defines additional elements that further describe the input such as *default value* or *unit of measurement* (see WPS 2.0 standard, section 7.3.2 Literal Data [5: http://docs.opengeospatial.org/is/14-065/14-065.html#25]). These elements have no expression in the SCALE job type description.

The *minOccurs* element in the WPS process description is set to "1" if the element *required* for the respective input is set to "true" in the SCALE job type description.

The *maxOccurs* element has no expression in the SCALE job type description and is thus set to "1" by default.

For WPS complex data, the format needs to be specified, especially the MIME type. The MIME type has an expression in the SCALE job type description, namely the *media_type(s)* element of input or output data. Note that in the example ([JOB_TYPE_RESPONSE]) the *media_type* element is missing for the output data. As a fallback, the WPS instance adds the following two generic formats to the output:

```
{
  "default": true,
  "mimeType": "application/octet-stream"
},
{
  "default": false,
  "mimeType": "application/octet-stream",
  "encoding": "base64"
}
```

7.3. Execute

The Execute operation of a WPS executes a process.

7.3.1. Mapping to SCALE

A new job is queued in SCALE by posting job data encoded in JSON to the queue endpoint URL:

```
{SCALE_base_URL}/queue/new-job/
```

The specification for job data in JSON can be found on version 5.4 of the SCALE Job Data documentation [6: https://github.com/ngageoint/scale/blob/5.4.0/scale/docs/architecture/jobs/job_data.rst].

A complete example of a job data JSON is shown below:

```
{
    "job_type_id": 10,
    "job_data": {
        "version": "1.0",
        "input_data": [
            {
                "name": "url",
                "value":
"http://ows.rasdaman.org/rasdaman/ows?service=WMS&version=1.3.0&request=GetMap&layers=
BlueMarbleCov&bbox=-90,-
180,90,180&width=800&height=600&crs=EPSG:4326&format=image/png&transparent=true&styles
="
            }
        ],
        "output_data": [
            {
                "name": "output_file",
                "workspace_id": 1
            }
        ]
    }
}
```

The parameter *workspace_id* is mandatory for outputs of type file or files. This is so SCALE knows where the result should be stored. As the endpoint-URL for submitting new jobs is generic, the *id* of the job type also needs to be specified.

SCALE jobs are only asynchronously executed, meaning that the response is returned immediately with information about how to obtain status updates.

The response body is a job description with additional information about the input and output data. Also, a HTTP header named *location* is returned that contains an URL pointing to status information about the process execution.

A client needs to poll the status location regularly to check the status of the execution.

The execution of a process using the OGC API - Processes is very similar.

The endpoint URL for the execution is as follows:

```
{WPS_base_URL}/processes/{process-id}/jobs
```

As the process id is part of the URL, there is no need to repeat it in the execute JSON.

A complete example of a WPS execute JSON is shown below:

```
{
    "inputs": [
        {
            "id": "url",
            "input": {
                "value": {
                    "inlineValue":
"http://ows.rasdaman.org/rasdaman/ows?service=WMS&version=1.3.0&request=GetMap&layers=
BlueMarbleCov&bbox=-90,-
180,90,180&width=800&height=600&crs=EPSG:4326&format=image/png&transparent=true&styles
="
                }
            }
        }
    ],
    "outputs": [
        {
            "id": "output_file",
            "format":{
               "mimeType" : "application/octet-stream"
            },
            "transmissionMode": "reference" ①
        }
    ],
    "mode" : "sync",
    "response" : "document" ②
}
```

In addition to the input and output specification, there are three WPS specific parameters:

- transmissionMode
- mode
- response

The *transmissionMode* parameter determines whether the WPS should return the data inline in the response JSON or as Web-accessible resource.

The *mode* parameter determines whether the WPS should execute the process synchronously or asynchronously. This is independent of the execution of SCALE jobs, which is always asynchronous.

The *response* parameter determines whether the WPS should return the output data wrapped in JSON or not (see GetResult).

The response has a HTTP header named *location* that contains an URL pointing to status information about the execution, similar to SCALE.

7.4. GetStatus

The GetStatus operation of a WPS returns status information about the execution of a process.

7.4.1. Mapping to SCALE

Status information about a SCALE job can be obtained by requesting the endpoint URL for the specific job execution:

```
{SCALE_base_URL}/jobs/{job-id}/
```

The response body is a job description with the additional information about the input and output data and information about the job status.

An abbreviated example of a job status JSON is shown below:

```
{
    "id": 33,
    "job_type": {
        "id": 10,
        "name": "curl",
        "version": "1.0",
        "title": "Curl Downloader",
        "description": "The working Curl Downloader",

    ....

    },
    "node": {
        "id": 2,
        "hostname": "192.168.1.242"
    },
    "error": null,
    "status": "RUNNING",①
    "priority": 1,
    "num_exes": 1,
    "timeout": 30000,
    "max_tries": 3,
    "cpus_required": null,
    "mem_required": null,
    "disk_in_required": 0.0,
    "disk_out_required": 0.0,
    "is_superseded": false,
    "root_superseded_job": null,
    "superseded_job": null,
    "superseded_by_job": null,
    "delete_superseded": true,
    "created": "2019-09-03T13:40:22.443960Z",
    "queued": "2019-09-03T13:40:22.503457Z",
    "started": "2019-09-03T13:40:22.884730Z",
    "ended": null,
    "last_status_change": "2019-09-03T13:40:22.884730Z",
    "superseded": null,
    "last_modified": "2019-09-03T13:40:23.745541Z",

    ....

}
```


① shows the status of the job. Besides RUNNING, the possible states are: COMPLETED, BLOCKED, QUEUED, FAILED, CANCELED and PENDING

Once the job has finished successfully, the status is set to COMPLETED and information about the output is added to the job description. An abbreviated example is shown below:

```
{
    "id": 33,
    "job_type": {
        "id": 10,
        "name": "curl",
        "version": "1.0",
        "title": "Curl Downloader",
        "description": "The working Curl Downloader",

    ....

    },
    "node": {
        "id": 2,
        "hostname": "192.168.1.242"
    },
    "error": null,
    "status": "COMPLETED",

    ...

    "ended": "2019-09-03T13:40:39.744206Z",
    "last_status_change": "2019-09-03T13:40:39.744206Z",

    ....

    "results": {
        "output_data": [
            {
                "name": "output_file",
                "file_id": 20
            }
        ],
        "version": "1.0"
    },
    "recipes": [],

    ...

    "outputs": [
        {
            "name": "output_file",
            "type": "file",
            "value": {
                "id": 20,
                "workspace": {
                    "id": 1,
                    "name": "workspace-input"
                },
                "file_name": "BlueMarbleCov.png",
                "media_type": "image/png",
```

```
                "file_type": "PRODUCT",
                "file_size": 855244,
                "data_type": [],
                "is_deleted": false,
                "uuid": "a5ce1f570c734a1f7fedea2866fb5e92",
                "url": "http://myserver,org/output/BlueMarbleCov.png",①
                "created": "2019-09-03T13:40:38.583206Z",
                "deleted": null,

                ....

            }
        }
    ]
}
```

① Output-URL

If the job execution failed, the error-element of the response is populated, as shown below:

```
{
    "id": 34,
    "job_type": {
        "id": 10,
        "name": "curl",
        "version": "1.0",
        "title": "Curl Downloader",
        "description": "The working Curl Downloader",

    ....

    },
    "node": {
        "id": 2,
        "hostname": "192.168.1.242"
    },
    "error": {
        "id": 2,
        "name": "unknown",
        "title": "Unknown",
        "description": "The cause of this failure is unknown. Please see the job log
for more information and report this error to the Scale development team at
https://github.com/ngageoint/scale so they can correct the issue.",
        "category": "SYSTEM",
        "is_builtin": true,
        "created": "2015-03-11T00:00:00Z",
        "last_modified": "2015-03-11T00:00:00Z"
    },
    "status": "FAILED",
    "priority": 1,

    ...

}
```


Status information about a WPS process execution can be obtained by requesting the respective endpoint URL. For example:

```
{WPS_base_URL}/processes/{process-id}/jobs/{job-id}
```

An example response for a running job is:

```
{
  "status": "running",
  "jobID": "2a49446a-305a-4c68-8116-6a7d85d154c7",
  "links": [
    {
      "href": "/processes/scale-algorithm-curl-1.0-r2/jobs/2a49446a-305a-4c68-8116-
6a7d85d154c7",
      "rel": "self",
      "type": "application/json",
      "title": "this document"
    }
  ]
}
```


An example response for a successful job is:


```
{
  "status": "succesful",
  "jobID": "2a49446a-305a-4c68-8116-6a7d85d154c7",
  "links": [
    {
      "href": "/processes/scale-algorithm-curl-1.0-r2/jobs/2a49446a-305a-4c68-8116-
6a7d85d154c7",
      "rel": "self",
      "type": "application/json",
      "title": "this document"
    },
    {
      "href": "/processes/scale-algorithm-curl-1.0-r2/jobs/2a49446a-305a-4c68-8116-
6a7d85d154c7/results",
      "rel": "result",
      "type": "application/json",
      "title": "Job result"
    }
  ]
}
```


An example response for a failed job is:

```
{
  "status": "failed",
  "jobID": "2a49446a-305a-4c68-8116-6a7d85d154c7",
  "links": [
    {
      "href": "/processes/scale-algorithm-curl-1.0-r2/jobs/2a49446a-305a-4c68-8116-
6a7d85d154c7",
      "rel": "self",
      "type": "application/json",
      "title": "this document"
    },
    {
      "href": "/processes/scale-algorithm-curl-1.0-r2/jobs/2a49446a-305a-4c68-8116-
6a7d85d154c7/results",
      "rel": "exception",
      "type": "application/json",
      "title": "Job exception"
    }
  ]
}
```


The job status of WPS and SCALE can be mapped as follows:

Table 4. Mapping between WPS process and SCALE job stati

WPS process status	SCALE job status
accepted	QUEUED
running	RUNNING
successful	COMPLETED
failed	FAILED
Not mapped	BLOCKED
Not mapped	CANCELED
Not mapped	PENDING

7.5. GetResult

The GetResult operation of a WPS returns the result of a process execution.

7.5.1. Mapping to SCALE

As shown in [SCALE_STATUS_COMPLETED], if a SCALE job is completed, the *outputs* section of the job description is populated with information regarding the job outputs. The output files are stored in SCALE workspaces, from where they can be used by other jobs in workflows or can be downloaded by clients via a Web-accessible URL.

If a WPS process execution is completed, the status information response is populated with a link to the outputs. This is shown in [WPS_SUCCESSFUL_JOB].

Each output can be requested from a WPS in two different ways:

- As a reference to a Web-accessible resource and;
- Inline in the GetResult response. The output type needs to be specified in the execute request (see [WPS_EXECUTE_JSON], 1).

The following example shows a WPS GetResult response encoded in JSON. The output was requested as reference.

```
[
  {
    "id": "output_file",
    "value": {
      "href": "/results/2a49446a-305a-4c68-8116-6a7d85d154c7/result"
    }
  }
]
```

For the WPS implementation used in Testbed-15, the outputs were stored in the WPS and therefore the output-URL points to a WPS endpoint. Another approach could be to return the Web-accessible URL of the output file in the SCALE workspace.

An example of an output requested as value is shown below:

```
[
  {
    "id": "output_file",
    "value": {
      "inlineValue": "iVBORw0KGgoAA ... ORK5C"
    }
  }
]
```

Single outputs can also be requested without the JSON-wrapper. For this, the *response* element needs to be set to "raw" in the execute JSON (see [WPS_EXECUTE_JSON], 2).

Keypoints	SCALE jobs can be mapped to WPS processes in a straightforward way enabling a standardized description and execution of SCALE jobs. However, some issues will need to be addressed concerning complex inputs and literal outputs.

Chapter 8. Processing Workflow

This section presents the entire workflow from installation, deployment and execution of data processing applications in the SCALE Datacenter environment using an OGC WPS interface.

8.1. SCALE Data Center Installation

A SCALE data center installation has two stages. First the DC/OS operating environment must be configured. Then SCALE is deployed on DC/OS.

8.1.1. DC/OS Background

DC/OS is an open-source, distributed operating system based on Apache Mesos distributed systems kernel. DC/OS runs on a cluster of machines. Each machine is called "a node". There are two types of nodes: master and agent. Master nodes coordinate the cluster operation while agent nodes execute distributed application workloads. DC/OS coordinates distribution of computing tasks among the available nodes. DC/OS applications are made up of workloads (or services) that are isolated into docker containers. DC/OS offers capabilities for workload scheduling, resource allocation, service discovery, workload collocation, load balancing and software defined networking.

DC/OS has major systems that implement distributed operating system capabilities: Apache Mesos and Marathon. Mesos is "a distributed systems kernel". Like a traditional OS kernel Mesos manages the basic resource (cpu, memory and storage) allocation and task execution. Marathon is "a container orchestration platform". It has a role that in non-distributed operating system is performed by the package manager and the service process system. Marathon uses basic resource and task capabilities provided by the Mesos kernel to implement a distributed application framework. It deploys distributed applications from deployment configurations that define packaging, dependencies and service discovery and load balancing infrastructures. Marathon manages the lifecycle of distributed applications and orchestrates the cluster behavior of multiple services that compose them. Marathon performs application health checks and manages application scaling, graceful service failure and recovery.

8.1.2. DC/OS Installation

Installing DC/OS requires configuring 3 types of nodes (virtual instances): master, agent and bootstrap. The master node directs the cluster operation. The cluster needs the master node to function for distributed applications to run. Therefore, running 3 identical master nodes is recommended. However, the Testbed participants found that in the Testbed demonstration environment a single master works without problems. The second type of node is the agent node. Agent nodes are VM instances where containerized workloads (single run tasks and long-lived services) are executed. DC/OS distributes the tasks and services over the agent nodes seamlessly. The Testbed participants found that to run SCALE, the DC/OS cluster must have at very least 4 agent nodes with 12 CPU cores shared among them. Finally, the third type of DC/OS node is the bootstrap node. The bootstrap VM instance is not part of the cluster operation, but acts as a centralized repository for configuration information. The bootstrap instance is only used for adding new master and agent nodes to the cluster.

In the rest of this section the steps we took to run DC/OS on our Apache CloudStack private cloud with CentOS instances are discussed. DC/OS can also be installed on AWS, Azure and Google clouds and on local on-premises clouds. Detailed directions for DC/OS version 1.13 setup in all environments are provided by D2IQ (formerly Mesopshere) and found here: https://docs.d2iq.com/mesosphere/dcos/1.13/installing/production/

This Testbed activity did not have a local DNS in the private cloud, so each node was configured with full host list in `/etc/hosts` file:

```
192.168.1.160 tb15-vm1
192.168.1.165 tb15-vm2
192.168.1.242 tb15-vm3
192.168.1.246 tb15-vm4
192.168.1.210 tb15-storage
```

Bootstrap Instance Configuration

The bootstrap node was build following DC/OS documentation: https://docs.d2iq.com/mesosphere/dcos/1.13/installing/production/

The following `genconf/config.yaml` file was generated.

```
bootstrap_url: http://192.168.1.189:80
cluster_name: tb15
exhibitor_storage_backend: static
master_discovery: static
ip_detect_public_filename: ip-detect-public
enable_ipv6: false
master_list:
- 192.168.1.248
resolvers:
- 129.174.67.98
use_proxy: 'false'
```

The `bootstrap_url` is the IP address of the bootstrap node. `master_list` contains a single entry because only have one master node was used in this activity. A DC/OS built-in DNS resolver is used which works with a static `/etc/hosts` DNS table.

A host public IP determination script must be hosted on the bootstrap node. The Testbed participants did not have public IPs so an `ip-detect` script was set to return the private IP for the primary network interface:

```
#!/usr/bin/env bash
set -o nounset -o errexit
export PATH=/usr/sbin:/usr/bin:$PATH
echo $(ip addr show ens3 | grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' |
head -1)
```

Master and Agent Node Configuration

For master and agent nodes five t2.xlarge (4 cpu cores, 16GB RAM) instances were created. Each instance runs CentOS 7 which is required by DC/OS. One of the 5 instances was configured as DC/OS master node and the remaining 4 instances were used as DC/OS agent nodes. One of the agent nodes also was configured to be an NFS host used for shared storage for the cluster.

Each machine was configured identically to prepare for DC/OS master or agent installation:

- Stop and disable `firewalld` and `dnsmaq` services
- Install and enable `docker` service
- `groupadd nogroup`
- `groupadd docker`
- Disable SELinux (`setenforce 0`) and edit `/etc/selinux/config` to disable SELinux on reboot
- Configure and mount shared NFS storage

Because there was not an internal DNS service, `/etc/hosts` was configured on each of the 5 DC/OS with IPs and names of all 5 nodes. DC/OS must be able to lookup hostnames of the nodes in the cluster.

The master node must be added first and then agent nodes configured to join the cluster. `dcos_install.sh` script from the bootstrap node is used to add new nodes.

Once DC/OS installation is completed, the DC/OS portal is visible on http://MASTER-IP. DC/OS uses OpenAuth for authentication and on first use the DC/OS portal guides the user to configure their login information. Successfully installed DC/OS system should list 4 healthy agent nodes in the **`Nodes`** tab.

8.1.3. SCALE installation

Once DC/OS is configured and all nodes are in a healthy state, distributed applications can be installed on DC/OS.

SCALE depends on the Elastic Search service application, which must be installed first. Elastic Search is a distributed textual data search engine. The engine is installed as a DC/OS service using Apache Marathon service for deployment configuration.

The Marathon web console can be accessed using this url: http://MASTER-IP/service/marathon. Inside the portal, clicking **`Create Application`** opens a dialog for launching DC/OS distributed applications from JSON specifications. The following JSON will create a distributed Elastic Search application for Scale:

```
{
  "id": "/scale-elasticsearch",
  "container": {
    "portMappings": [
      {
        "containerPort": 9200,
        "labels": {
          "VIP_0": "/scale-elasticsearch:9200"
        },
        "protocol": "tcp"
      }
    ],
    "type": "MESOS",
    "volumes": [],
    "docker": {
      "image": "elasticsearch:2-alpine",
      "forcePullImage": false,
      "parameters": []
    }
  },
  "healthChecks": [
    {
      "gracePeriodSeconds": 300,
      "intervalSeconds": 60,
      "maxConsecutiveFailures": 3,
      "portIndex": 0,
      "timeoutSeconds": 20,
      "delaySeconds": 15,
      "protocol": "MESOS_HTTP"
    }
  ],
  "instances": 1,
  "mem": 2048,
  "cpus": 1,
  "disk": 0,
  "gpus": 0,
  "networks": [
    {
      "mode": "container/bridge"
    }
  ]
}
```

Several minutes are required for Elastic Search deployment to complete. Once that is complete SCALE 5.9.7 can be deployed. This is the configuration to deploy SCALE application using Marathon:

```
{
  "env": {
    "MARATHON_SERVERS": "http://marathon.mesos:8080,https://marathon.mesos:8443",
```

```
    "DCOS_PACKAGE_FRAMEWORK_NAME": "scale",
    "ENABLE_BOOTSTRAP": "true",
    "DEPLOY_WEBSERVER": "true",
    "SCALE_DB_USER": "scale",
    "SCALE_DB_NAME": "scale",
    "SCALE_DB_PASS": "scale",
    "SCALE_VHOST": "scale.marathon.mesos",
    "SCALE_ELASTICSEARCH_URLS": "http://scale-
elasticsearch.marathon.l4lb.thisdcos.directory:9200"
  },
  "labels": {
    "DCOS_PACKAGE_FRAMEWORK_NAME": "scale"
  },
  "id": "/scale",
  "args": [
    "scale_scheduler"
  ],
  "constraints": [
    [
      "hostname",
      "UNIQUE"
    ]
  ],
  "container": {
    "type": "DOCKER",
    "volumes": [],
    "docker": {
      "image": "geoint/scale:5.9.7",
      "forcePullImage": true,
      "privileged": false,
      "parameters": []
    }
  },
  "cpus": 1,
  "disk": 0,
  "instances": 1,
  "mem": 1024,
  "gpus": 0,
  "healthChecks": [
    {
      "gracePeriodSeconds": 300,
      "intervalSeconds": 30,
      "maxConsecutiveFailures": 3,
      "timeoutSeconds": 20,
      "delaySeconds": 15,
      "protocol": "COMMAND",
      "command": {
        "value": "ps -ef | grep 'manage.py scale_scheduler' | grep -v grep >/dev/null"
      }
    }
  ],
```

```
  "networks": [
    {
      "mode": "host"
    }
  ],
  "portDefinitions": [
    {
      "name": "default",
      "protocol": "tcp",
      "port": 10103
    }
  ]
}
```

Some time is required for SCALE to fully come online. Once the deployment is complete, the SCALE Web Console can be viewed at http://MASTER-IP/service/scale

SCALE is composed of several DC/OS services.

- scale
- scale-db
- scale-logstash
- scale-rabbitmq
- scale-webserver

All of the services should be listed in Marathon Web Console and have status `Running`

Keypoints	SCALE is installed and executed on DC/OS distributed operating system. A minimum of 1 master node (instance VM) and 4 agent nodes is required to run SCALE.

8.2. Data Processing Application Deployment

"Data Processing Application Deployment" is concerned with the key consideration that all the processing logic should be implemented in the docker images. SCALE does not execute any specific business logic itself.

To run the processing application in SCALE, the application must be dockerized. This requirement is emphasized.

SCALE data processing applications are configured as workflows of simple tasks that consume inputs and produce outputs. These tasks are called **Jobs** and the workflows are called **Recipes**. Each job performs a single data processing task by creating and executing a process in a docker container. Jobs are defined by specifying a docker image. During task execution, the container creation and deployment is managed and by Scale and DC/OS. One or more jobs are chained together to create a recipe. **Workspaces** are used to store data for job producers and consumers. **SEED** is the job interface specification that is used to package job input/output parameter metadata

with docker images that contain discrete processing algorithms. SEED interfaces are used to define jobs in SCALE.

Deploying distributed data processing applications into SCALE consists of two broad steps: First, adding job definitions in SCALE to define simple processing tasks; and Second, creating workflows that combine multiple discrete job tasks into complex applications. The second step is optional - a basic processing application can be defined with a single processing task.

To create and execute a basic processing task the following steps need to be performed: create workspace, create shared data folder, add job, and run job

The next sections demonstrate how to create and execute a basic job that performs a single task: download data into workspace using curl. The task is executed in a docker container. SCALE handles providing the right execution parameters and DC/OS handles scheduling and running the task docker image on an available agent node when the job is executed.

SCALE is managed using a restful HTTP API. Authentication is handled by HTTP Basic Auth, using a special secret token. The token is provided by `dcos` CLI application. `dcos` is installed on a separate computer that is not part of the cluster, following these directions: https://docs.d2iq.com/mesosphere/dcos/1.13/cli/install/

Once `dcos` is installed, the authentication token can be obtained as follows:

```
dcos auth login
export TOKEN=$(dcos config show core.dcos_acs_token)
```

Now the workflow can use the `$TOKEN` variable to deploy Scale applications using an HTTP API.

8.2.1. Deploying Workspaces

Workspaces are storage configurations for job input and output data. All jobs must have a workspace so the first task after obtaining `$TOKEN` is to create a workspace and a shared folder definition for the workspace.

Create `add-workspace.json` file:

```
{
    "name": "workspace-input",
    "title": "workspace-input",
    "description": "workspace-input",
    "base_url": "http://cloud.csiss.gmu.edu/scale_files",
    "is_active": true,
    "json_config": {
        "broker": {
            "host_path": "/mnt/nfsshare/input",
            "type": "host"
        }
    }
}
```

Then create `host-broker.json` shared folder definition file:

```
{
   "version": "1.0",
   "json_config": {
        "broker": {
            "type": "host",
            "host_path": "/mnt/nfsshare/broker/"
        }
   },
   "name": "host_broker",
   "title": "host_broker",
   "description": "host broker folder which will be mounted to every container"
}
```

Then pass the JSON files to Scale to create the resources:

```
# create workspace
curl -X POST -H "Authorization: token=$TOKEN" -H "Content-Type: application/json" -H
"Cache-Control: no-cache" -d @add-workspace.json "http://MASTER-
IP/service/scale/api/v5/workspaces/"

# create host broker folder
curl -X POST -H "Authorization: token=$TOKEN" -H "Content-Type: application/json" -H
"Cache-Control: no-cache" -d @host-broker.json "http://MASTER-
IP/service/scale/api/v5/workspaces/"
```

8.2.2. Create a SCALE Job Type

Using this definition file `new-job.json`:

```
{
```

```
    "name": "curl-3",
    "version": "1.0",
    "title": "Curl Downloader With Output",
    "description": "This is a description of the job",
    "category": "test",
    "author_name": "ZS",
    "author_url": "http://csiss.gmu.edu",
    "is_long_running": false,
    "is_operational": true,
    "is_paused": false,
    "icon_code": "f0ed",
    "docker_privileged": false,
    "docker_image": "appropriate/curl",
    "priority": 1,
    "timeout": 30000,
    "max_tries": 3,
    "cpus_required": 0.5,
    "mem_required": 64.0,
    "mem_const_required": 64.0,
    "mem_mult_required": 0.0,
    "shared_mem_required": 0.0,
    "disk_out_const_required": 64.0,
    "disk_out_mult_required": 0.0,
    "interface": {
        "version": "1.0",
        "command": "curl",
        "command_arguments": "-o ${job_output_dir}/${out_filename} ${url}",
        "input_data": [
            {
                "required": true,
                "type": "property",
                "name": "url"
            },
            {
                "required": true,
                "type": "property",
                "name": "out_filename"
            }

        ],
        "output_data": [
            {
                "media_type": "image/tiff",
                "required": true,
                "type": "file",
                "name": "output_file_tif"
            }
        ],
        "shared_resources": []
    },
```

```
    "configuration": {
        "version": "2.0",
        "settings": {
        }
    },
    "custom_resources": {
        "version": "1.0",
        "resources": {
        }
    },
    "error_mapping": {
        "version": "1.0",
        "exit_codes": {
            "1": "unknown"
        }
    }
}
```

Create the job type in SCALE using HTTP and `new-job.json`:

```
# add job
curl --request POST --header "Authorization: token=$TOKEN" --header "Content-Type:
application/json" --header "Cache-Control: no-cache" --data @new-job.json --url
"http://MASTER-IP/service/scale/api/v5/job-types/"
```

Submitting the job type definition will create a new unique `job_type_id`. The available jobs and their IDs can be retrieved from Scale: `GET http://MASTER-IP/service/scale/api/v5/job-types/`

8.2.3. Execute a Job

Once the job type is defined, it can be executed by specifying the `job_type_id` and input/output parameters using the HTTP `POST queue/new-job` API. To execute the job, the `run-job.json` parameter file must be created:

```
{
    "job_type_id": 17,
    "job_data": {
        "version": "1.0",
        "input_data": [
            {
                "name": "url",
                "value":
"http://ows.rasdaman.org/rasdaman/ows?service=WMS&version=1.3.0&request=GetMap&layers=
BlueMarbleCov&bbox=-90,-
180,90,180&width=800&height=600&crs=EPSG:4326&format=image/png&transparent=true&styles
="
            },
            {
                "name": "out_filename",
                "value": "/tmp/out.tif"
            }
        ],
        "output_data": [
            {
                "name": "output_file_tif",
                "workspace_id": 2
            }
        ]
    }
}
```

Then a new job API can be invoked:

```
curl -X POST -H "Authorization: token=$TOKEN" -H "Content-Type: application/json" -H
"Cache-Control: no-cache" -d @run-job.json --url "http://MASTER-
IP/service/scale/api/v5/queue/new-job/"
```

Job success and failure information and data outputs can be viewed using the Scale Web UI (http://MASTER-IP/service/scale) or using the Scale RESTful HTTP API documented here: http://ngageoint.github.io/scale/docs/rest/v6/job.html

Keypoints	SCALE data processing applications are distributed workflows composed of discrete processing units or task. Processing units are encapsulated in docker containers and are named "jobs" in Scale and "recipes" are executable workflows composed of "jobs" in Scale.

8.3. Discovery for Data Processing Input

Input data is obtained from the rasdaman federation, established between several AWS cloud nodes allocated in the US with nodes of the European Earth Datacube Federation, providing WCS, WMS and WCPS access to Sentinel 2, Temperature and Precipitation datasets.

Data discovery is achieved through the WCS GetCapabilities operation, which can be performed against any server participating in the federation. The GetCapabilities response of any federation participant contains all the data available in the federation, presenting a common, location-transparent information space to the user.

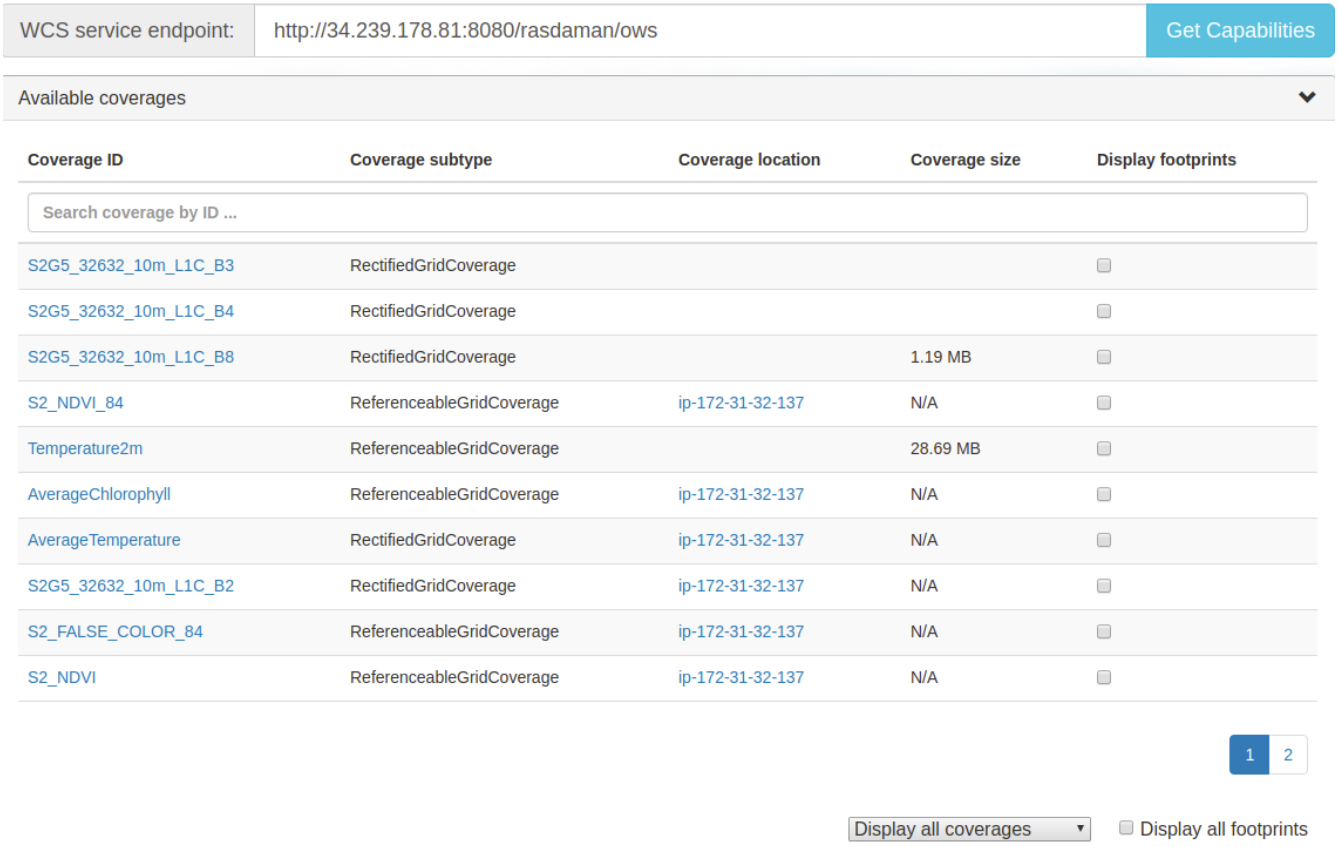

Figure 1. Processed federated GetCapabilities response. Empty coverage location means coverage is local.

Querying the federation for data is done in the same manner. The user is free to send requests against any of the participating servers. The server receiving the query will orchestrate automatically with all members needed (i.e. holding relevant data) to jointly process and come back with the query result.

As a result, SCALE can query any node in the federation to obtain relevant data for further processing. The following queries have been established for obtaining input data:

WCPS NDVI json

```
for $b8 in (S2G5_32632_10m_L1C_B8), $b4 in (S2G5_32632_10m_L1C_B4)
  return encode(
    ($b8 - $b4) / ($b8 + $b4),
  "json")
```

The query computes the NDVI values over an area of Sentinel 2, and encodes the result as json before returning.

WCPS NDVI NetCDF

```
for $b8 in (S2G5_32632_10m_L1C_B8), $b4 in (S2G5_32632_10m_L1C_B4)
  return encode(
    ($b8 - $b4) / ($b8 + $b4),
  "application/netcdf")
```

A second alternative encodes the NDVI data in NetCDF, which allows retaining full coverage information in the response, such as geo-referencing and metadata.

WCS RGB NDVI

```
GET http://54.93.148.198:8080/rasdaman/ows?
      SERVICE=WCS&
      VERSION=2.0.1&
      REQUEST=GetCoverage&
      COVERAGEID=S2_NDVI&
      SUBSET=ansi("2018-10-30T00:00:00.000Z")&
      FORMAT=image/jpeg
```

The WCS request above retrieves a subset of the S2_NDVI coverage, an RGB coverage storing the colored NDVI images obtained from Sentinel 2 data, and encodes the results as jpeg.

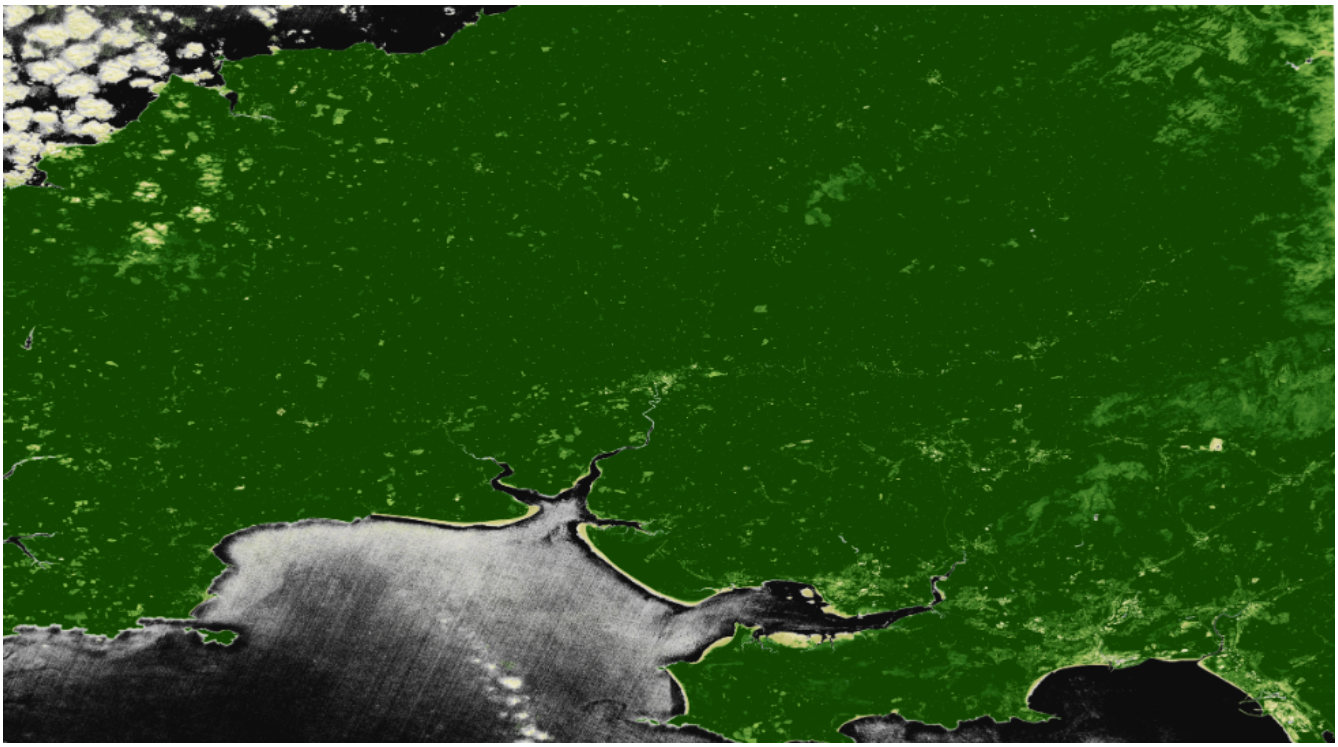

Figure 2. WCS response: NDVI colors from Sentinel 2.

Keypoints	SCALE obtains input data from a rasdaman federation offering WCS and WCPS services. The data discovery process was limited to using WCS's GetCapabilities, as this approach proved to be sufficient for the Testbed purposes. In future, other methods such as STAC or OpenSearch could be investigated. The data access process was realized through WCS's GetCoverage mechanism, as well as WCPS.

8.4. Data Processing Execution

The following sequence diagram shows the execution of a rasdaman query/function that is called by a SCALE job process that is exposed through an implementation of the draft OGC API – Processes specification.

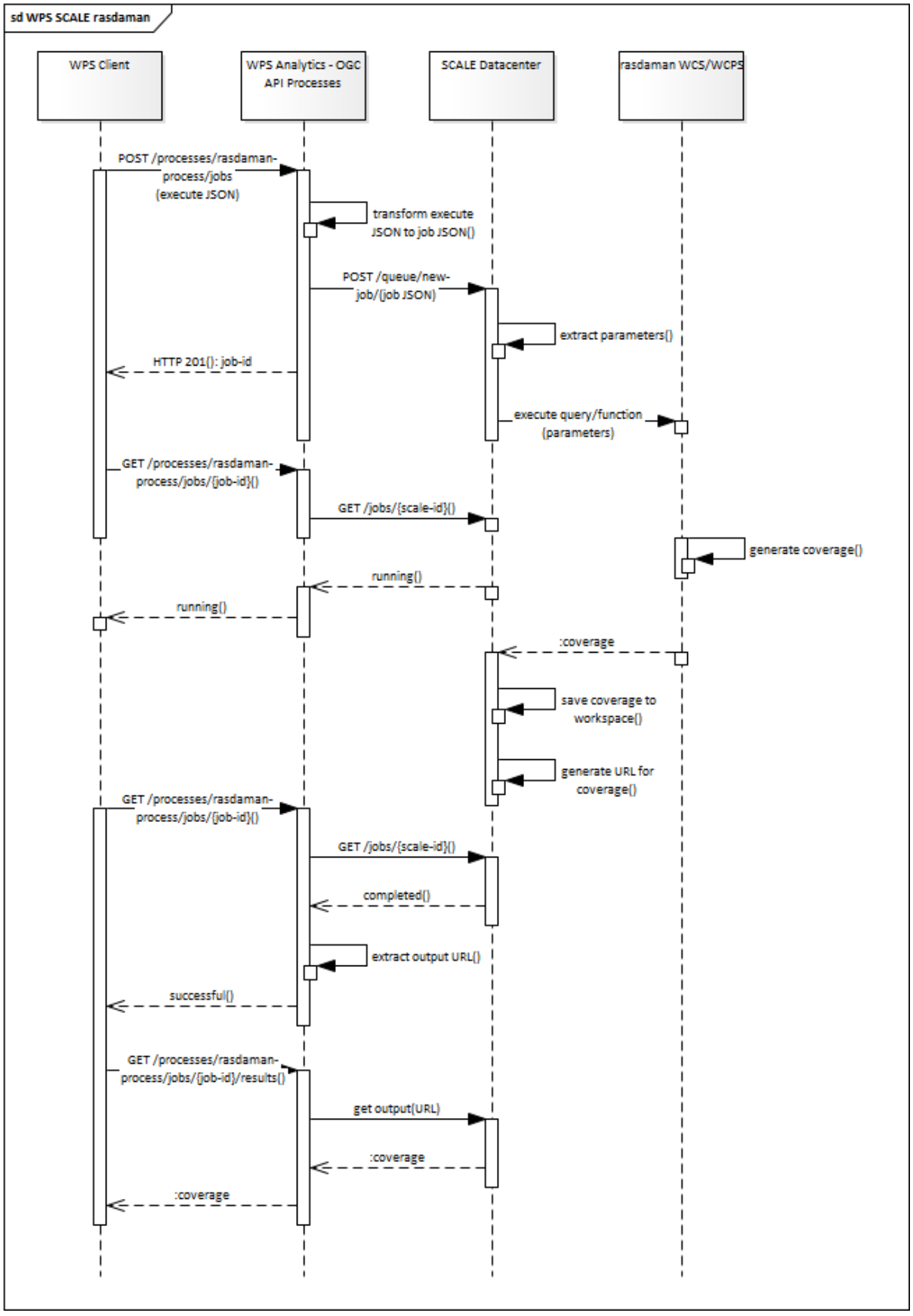

Figure 3. WPS process wrapping a SCALE job connecting to rasdaman

The client sends an asynchronous execute request encoded in JSON to the server implementing the draft OGC API - Processes. The execute JSON is transformed to a JSON request fitting for the underlying wrapped SCALE job. A new SCALE job is queued in the SCALE Datacenter. The parameters for the job are extracted from the JSON request. In the meantime. the client can poll the

status of the process/job. When the SCALE job reaches the top of the queue, it is executed. The job calls a rasdaman WCS/WCPS server with query/function parameters. The rasdaman server generates a coverage based on the parameters and returns the coverage to the SCALE Datacenter. There the coverage is stored in a workspace and a Web-accessible URL is generated. This URL is returned to the WPS along with the next status update. If the result is requested from the WPS, the coverage is downloaded from the SCALE workspace and returned to the client.

As mentioned in section GetResult, the WPS could also just reference the URL of the coverage in the SCALE workspace instead of downloading it. This would reduce network traffic and also storage space on the WPS server. On the downside, the WPS has no influence on how long the output is stored in the SCALE workspace. So, the reference URL could result in a broken link if the SCALE workspace is cleaned.

Another reason not to download the output onto the WPS server is if the output is used as input for another process in a workflow. Such workflows can be different WPS processes that do not necessarily use SCALE as a processing backend, or the workflow can consist of different SCALE jobs, meaning the workflow is a SCALE recipe. Recipe workflows have the advantage of being able to access the SCALE workspace directly via the file system of the server. This is shown in the following figure:

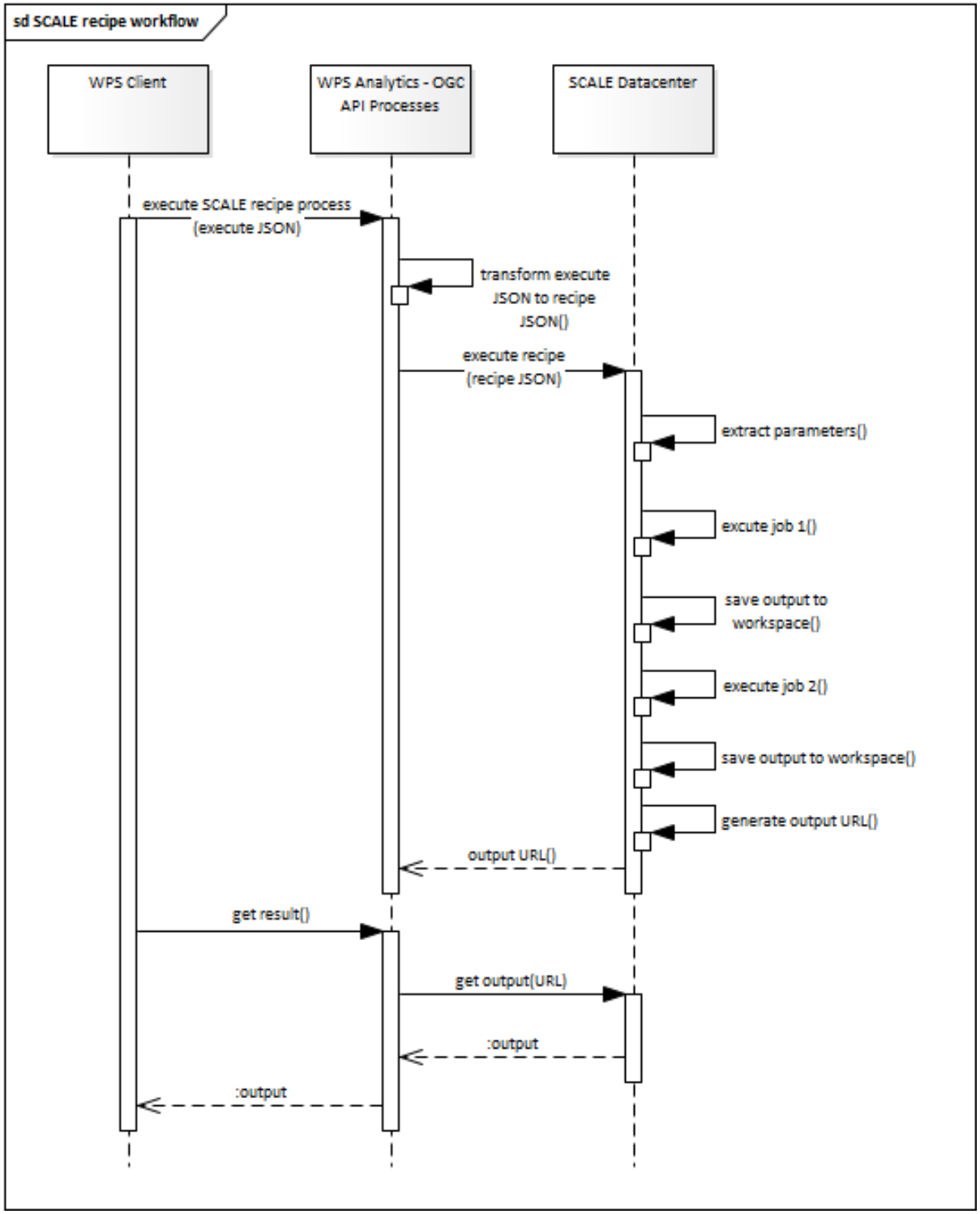

Figure 4. Workflow consisting of a SCALE recipe

Workflows of different WPS processes would still reference the Web-accessible URL of the output in the SCALE workspace, but the next process still needs to download the output from SCALE in order to process it.

The following figure shows a workflow consisting of WPS processes.

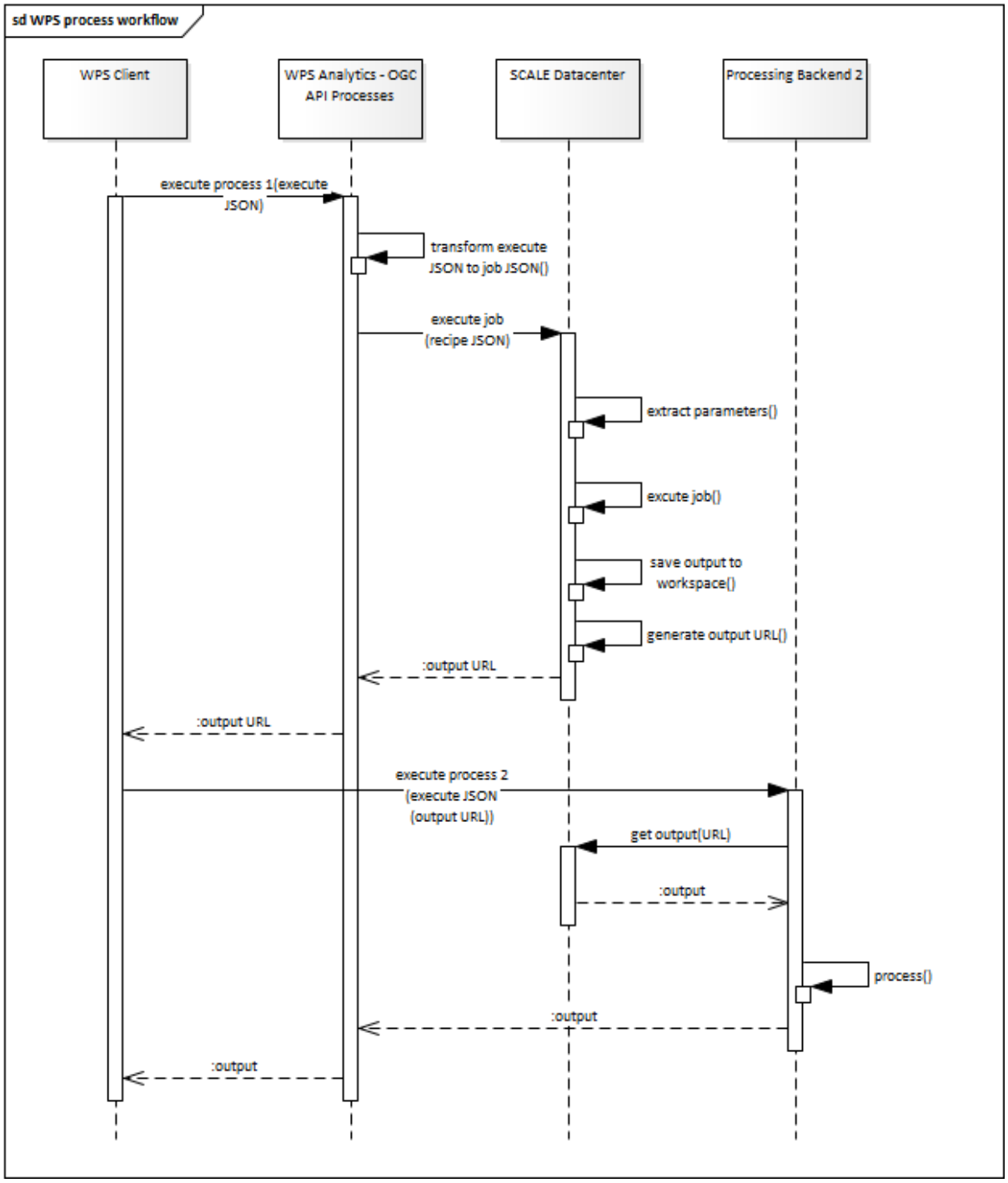

Figure 5. Workflow consisting of different WPS processes

The advantages and disadvantages of the different workflow approaches should be investigated further. Also, the registration of the outputs in a catalogue service should be tested. This could be done by a dedicated SCALE job.

Keypoints	Our experiments showed that different approaches to execute workflows with SCALE jobs wrapped in WPS processes exist. Future testbeds should investigate these approaches by creating different SCALE jobs that could either be combined to SCALE recipes or executed as single WPS processes.

Chapter 9. Main Findings and Recommendations

The experiments performed in Testbed-15 show that SCALE jobs can be mapped to WPS processes in a straightforward way.

SCALE has shown the potential to define processes in Job Types or SEED manifests aggregating:

- Software, packaged in a Docker container.
- result_manifest.json file or specific log output to signal output results to the Scale runtime environment for output extraction.
- Error definition, describing the meaning of exit values for the contained process
- Multiple inputs and outputs.
- Optional inputs and outputs.
- Input Properties (arbitrary values).
- Utilizing Docker volumes and mounts for providing auxiliary data.
- SEED specific definitions.

SCALE also provides the definition of Workflows via Scale Recipes by defining:

- Start and End nodes.
- Mime type declaration and matching.
- Parameter (file or property) mapping.
- Output dependent branching.

The deployment is straightforward by creating a Job Type that uses the Docker container and then calling/invoking it via the queue endpoint. The Docker container needs to be accessible from one of the registries configured in Scale where the images are hosted.

Even if SCALE allows a standardized description and execution of jobs, as well as a standardized check for execution status and result retrieval wrapped in a WPS serivce, some issues were discovered concerning complex inputs and literal outputs.

Topics identified for future work are:

- Mapping complex (file-based) inputs for SCALE jobs.
- Mapping literal outputs for SCALE jobs.
- Mapping WPS bounding boxes for SCALE jobs.
- Handling workflows/SCALE recipes with WPS processes.
- Handling transactional WPS requests with SCALE. This means deploying and undeploying of processes/SCALE jobs.
- Security federation from WPS to SCALE to data servers.

- Handling provenance information in the WPS that is created by SCALE.
- Utilizing of SCALE system metrics to create prognosis of process execution duration

Appendix A: Revision History

Table 5. Revision History

January 18, 2018	**S. Serich**	**.1**	**all**	**additional guidance to Editors; clean up headings in appendices**
September 17, 2019	C. Reed	.1	All	Internal Review complete.
December 18, 2019	G. Hobona	.1	All	Final Review complete.

Appendix B: Bibliography

1. Jonathan Meyer : Authentication in Scale, https://github.com/ngageoint/scale/wiki/Authentication-in-Scale, (2019).
2. D2iQ, Inc.: DC/OS (the Distributed Cloud Operating System), https://dcos.io/, (2019).
3. Encode OSS Ltd.: Django REST framework, https://www.django-rest-framework.org, (2019).
4. Leidos Editorial Team: GEOAxIS Secures Intelligence & Efficiency for the NGA, https://www.leidos.com/insights/geoaxis-secures-intelligence-efficiency-nga, (2017).
5. Harris Geospatial Solutions, Inc.: ENVI image analysis software, https://www.harrisgeospatial.com/Software-Technology/ENVI, (2019).
6. The MathWorks, Inc.: MATLAB, https://www.mathworks.com/products/matlab.html, (2019).
7. The MathWorks, Inc.: MATLAB - deploytool - Compile and package functions for external deployment, https://www.mathworks.com/help/compiler/deploytool.html, (2019).
8. The MathWorks, Inc.: MATLAB - mcc - Compile MATLAB functions for deployment, https://www.mathworks.com/help/compiler_sdk/ml_code/mcc.html, (2019).