Computing and Informatics, Vol. 43, 2024, 1416–1431, doi: 10.31577/cai_2024_6_1416

SCALABLE CLOUD APPLICATION DEPLOYMENT SERVICE FOR VERSATILE CLOUD SERVICE DEPLOYMENT AND CONFIGURATION

Ondrej Habala*, Martin Šeleng

Institute of Informatics
Slovak Academy of Sciences
Dúbravská cesta 9
845 07 Bratislava, Slovakia
e-mail: {ondrej.habala, martin.seleng}@savba.sk

Michal Habala, Ľubor Stuhl

Demtec, s.r.o.
Hraničná 18
821 05 Bratislava, Slovakia
e-mail: {michal.habala, lubor.stuhl}@demtec.sk

Michal Staňo, Ladislav Hluchý

Institute of Informatics
Slovak Academy of Sciences
Dúbravská cesta 9
845 07 Bratislava, Slovakia
e-mail: {michal.stano, ladislav.hluchy}@savba.sk

Abstract. We present a cloud management service called RAIN. It has been designed specifically for versatility and scalability of operation, allowing for the processing of a large number of requests at the same time. Its operation is transactional and controlled by a workflow of operations forming one requisition. Requisitions

* Corresponding author

and their operations can be executed in parallel, allowing for high throughput and scalability of the controlled cloud environment(s). The service is being used in day-to-day operations in a commercial environment. It is also designed for high failure tolerance, which is necessary when operating on third party cloud infrastructures. It has been developed and actively used for several years now, giving us a mature tool with many important features added over time, allowing for practical day-to-day operations. The architecture of the service is open and easily extendable to allow the inclusion of new cloud services of various types – PaaS providers as well as providers of higher-level services. The service is accessed via an asynchronous REST API. It allows the caller to resume execution and not wait for cloud deployment operations to take an arbitrary amount of time to finish, receiving progress updates via a simple callback REST API.

Keywords: Cloud computing, cloud application deployment, cloud automation, REST service

Mathematics Subject Classification 2010: 68-U35

1 INTRODUCTION

Hundreds of millions of cloud instances of all types are being run every day. Cloud computing has become the mainstream method of deploying computing infrastructure and applications. The market in cloud services is worth hundreds of billions of dollars, and data stored in the cloud range in the zettabytes [1]. On the cloud, architectures of systems able to process exascale amounts of data are being designed and deployed [2, 3]. In this environment, manual deployment and configuration of cloud infrastructure and cloud applications is unthinkable, and numerous automation tools for these tasks have been developed. We will talk more about some of them in Section 2.

We have also developed one such tool, and are actively using, maintaining and expanding it. It is a generic system for cloud infrastructure as well as application deployment and (re)configuration, its name is RAIN (for Robotized Automatic Instance Nagger). It has the form of a service with a REST API. The folowing are the reasons why we have developed another such tool:

- At the time we have started the development (2017), many of the currently existing tools were not yet available.
- We needed a tool which would be highly scalable.
- We needed an infrastructure-agnostic tool which could work with any cloud infrastructure from any provider, and mix them freely (which can be a difficult task, often requiring specialized methodologies [4]).

- We needed a flexible, easily extendable tool which would potentially cover more than cloud infrastructure and applications management.
- Since the tool was to be part of a web-based infrastructure, we preferred a service with a REST API instead of a command-line tool.
- We needed a tool which would be transactional yet provide sufficient feedback, including some parameters of the created infrastructure or application.
- As the tool was meant for the deployment and configuration of commercially offered applications, high fault-tolerance was of paramount importance – a failure to flawlessly deploy the customer's required application could lead to the loss of the customer's business.

In the end, after analyzing the existing systems, we have come to the conclusion that using an existing system and developing for it the required plugins for the infrastructure and applications which we would need, while having to learn the internals of the selected system, would be more cumbersome than developing a lightweight backbone service tailored precisely for our needs, and then extend it with the plugins which we will need. This approach also allowed us to research and address cloud automation challenges better than if we would use an existing system, which has already fixed design choices and trade-offs.

In Section 2 we will provide a brief state-of-the-art of the most used cloud infrastructure and application deployment automation tools. In Section 3 we will describe the architecture of RAIN. In Section 4 we will explain the internal workings of RAIN, the most important concepts it uses and the most used plugins which access various infrastructure and application components. In Section 5 we will define the methods of parametrization of the cloud management process, which is the main tool with which RAIN users can influence their workflows. In Section 6 we provide a summary and outlook towards the future of RAIN.

2 STATE OF THE ART OF THE IT INFRASTRUCTURE AUTOMATION TOOLS, APPLICATION MANAGEMENT AND TOOLS FOR INTEGRATION WITH DEVELOPMENT PROCESS

In this section we describe several existing automation tools for infrastructures and applications deployment. Also, we deal with tools responsible for the application management and with tools used for integration with the development process.

One of the most popular and known tool is the Ansible [5]. Ansible is an open source IT automation engine that automates provisioning, configuration management, application deployment, orchestration, and many other IT processes. Ansible is an automation tool for IT infrastructure automation and server management. It automates various configuration tasks, application deployment, and software delivery. Ansible uses YAML files to define the infrastructure state, and establishes remote access over SSH.

SaltStack [6] is an open-source tool for configuration management, orchestration, and remote execution. The tool automates jobs in large-scale infrastructures from a single site. SaltStack utilizes a master-slave architecture with a central control server for remote hosts.

AWS CloudFormation [7] is an Amazon Web Services tool for provisioning and managing AWS resources. The solution uses a templating approach to describe IaC through YAML or JSON files.

GitLab CI/CD [8] (Continuous Integration and Continuous Delivery) is a built-in GitLab CI/CD tool. The tool automates software development and deployment. GitLab automates building, testing, and deploying code.

CircleCI [9] is a cloud-based CI/CD platform for streamlining development workflows. The tool automates building, testing, and deploying applications at different software development levels than Ansible.

Azure DevOps Server [10] (previously Microsoft Team Foundation Server) is a Microsoft platform for collaborative development. The tools suite provides functionalities to maintain the full application lifecycle management (ALM) process.

Azure Automation [11] is a cloud-based automation solution by Microsoft Azure. The service automates tasks across Azure cloud services to simplify resource management and deployment.

Terraform [12] is an open-source IaC tool which uses human-readable language to define and provide infrastructure resources such as virtual machines, storage, networks, and other cloud and on-the-premises resources.

Rudder [13] is an open-source automation tool for streamlining IT infrastructure and configuration management. The tool helps teams manage large-scale IT infrastructures through a centralized platform. The Rudder is an agent-based architecture with a web-based visual interface.

Chef Infra [14] is an Infrastructure as Code (IaC) framework for automating IT infrastructure configuration and maintenance. The tool uses DevOps infrastructure automation in both small and large-scale environments. Chef uses a domain-specific language called Ruby DSL for describing system states.

Jenkins [15] is a CI/CD tool for automating, building, and testing applications. The tool automates software development workflows, including code compiling, testing, and packaging software. Jenkins uses a master-slave architecture to execute tasks on remote machines.

Puppet[16] is an open-source IaC tool for managing infrastructure configuration and deployment. The solution uses declarative code to create and implement the desired system state. Puppet has an agent-based architecture and focuses on managing resource states.

CFEngine [17] is an automation and configuration management tool for maintaining large IT infrastructures. The open-source tool helps ensure compliance and consistency across distributed systems. CFEngine uses an agent-based architecture that constantly evaluates the system's current state compared to the desired results.

Semaphore [18] is a cloud-based CI/CD platform for streamlining application delivery. The tool aims to simplify processes in the CI/CD pipeline with automated workflows.

The tools mentioned in this chapter, and many others, provide subsets of the capabilities which we need for our infrastructure and application deployment needs, but none of them provide the whole set. Therefore, we have designed our tool, RAIN, that is described in the following chapters.

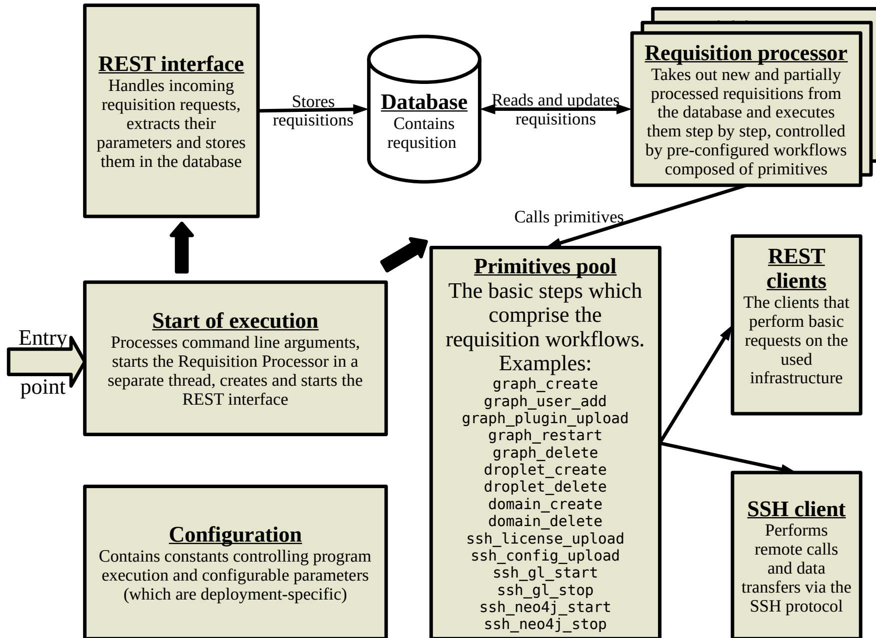

Figure 1. Architecture of RAIN cloud infrastructure and application deployment and configuration tool

3 RAIN ARCHITECTURE

RAIN is a scalable, extendable cloud infrastructure and application deployment, configuration and management service. It offers a RESTful API and is transactional – it executes the required task, returns metadata about the created infrastructure, but does not store any state for later reuse. In Figure 1 we show the high-level architecture of RAIN. It is divided into two main parts.

The first part is formed by the REST interface which handles incoming requests. This part of RAIN runs synchronously with the request originator (the client), and is therefore designed to handle the incoming request and return quickly the control back to the client. It only registers the request – creates a requisition, and notifies

the caller of the result of this registration – success, or failure if the request is malformed.

The second part, running asynchronously from the REST interface, is the requisition processor. It takes a requisition (a single request for some infrastructure work) input parameters, selects the appropriate workflow, executes it and stores the final cloud infrastructure metadata in the database. The processor can run in multiple independent instances in order to increase the throughput of RAIN. Since most of the time during requisition processing is spent waiting on the cloud infrastructure, running multiple requisition processors can significantly increase the responsiveness of the system. The different requisition processor instances are observing resource exclusivity, so no two requisition processor instances work on the same requisition, or relater requisitions requesting modifications of the same resource, which must be processed sequentially.

The database is the connecting point between the interface which communicates with the client, and the requisition processors which communicate with the cloud infrastructure. Incoming requisitions are stored in the database, and requisition processors access and process them asynchronously. The database stores information about all requisitions in the form of key-value pairs – requisition parameters.

Expandability and multi-infrastructure support are provided by the pool of so-called primitives. A primitive in our case is one operation, for example a PaaS instance creation, or a reconfiguration step. When access to new infrastructure is necessary, a new set of primitives for handling this infrastructure can be written and added into RAIN, using simple rules – every primitive has the same signature, and is stored in a source file in a pre-configured directory. New primitives can make use of existing ones, or of the underlying infrastructure of a hierarchy of REST client classes for different infrastructures.

4 REQUISITIONS, WORKFLOWS, OPERATIONS, PARAMETERS AND PRIMITIVES

By experimenting with cloud automation, we have arrived at a set of connected concepts which serve us well and which we believe are optimal for the class of cloud automation tools to which the RAIN belongs – services with high scalability, fault tolerance and responsiveness necessary for the commercial environment, and easy extensibility.

In Figure 2 we present the relationships between the main concepts used in the internal model of RAIN. The unit of work required by the user is called a `requisition`. A requisition is initiated by calling RAIN's REST interface. A requisition consists of a set of steps – a `workflow`. Each workflow consists of one or more `operations`. An operation is implemented by a `primitive`, which is a function programmed by RAIN maintainers, offering a prescribed signature. A primitive consumes and produces the parameters of a requisition.

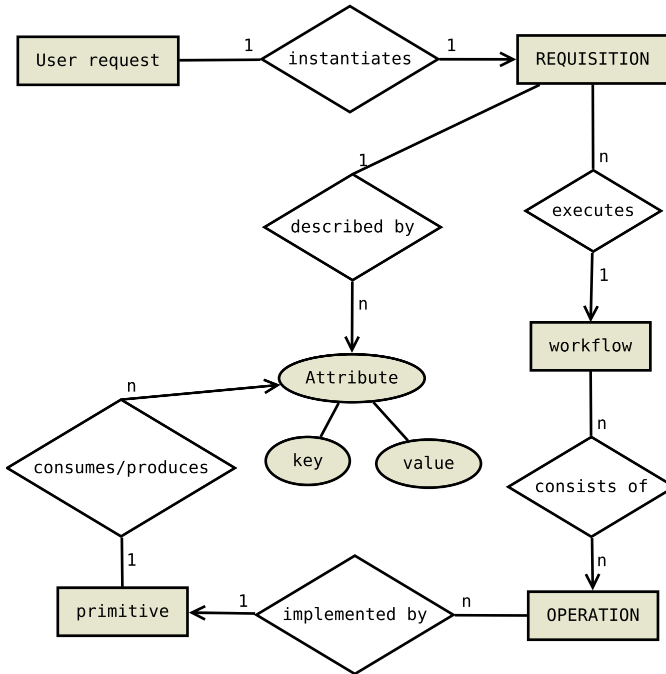

Figure 2. The Entity-Relationship Diagram of the main concepts used in RAIN

Each requisition at any point of time during its processing is described by a set of key-value pairs – `parameters`. The initial parameters are included in the client's REST call which creates the requisition, and more parameters are usually created by each operation in a requisition. Some of those parameters are returned in the (asynchronous) reply to the client's request. They usually describe parameters of the created infrastructure or application component, such as the IP address of a new PaaS instance. Primitives communicate with each other only through these parameters – some produce them, others consume them.

As of this paper, RAIN is a single-user tool, intended to work inside trusted company infrastructure, without any authentication and authorization. There is no *user* concept in the entity relationship diagram (Figure 2). We plan to change it, and allow RAIN to be deployed outside of the trusted infrastructure, as well as to be used by multiple users (or multiple cloud application administrative domains) at the same time. For this purpose we will introduce users in the RAIN database, and tie all requisitions, operations, their parameters and log outputs to a particular user. All of them will be accessible only once that user has authenticated, and each user will have access only to those parts of the database which are tied to them.

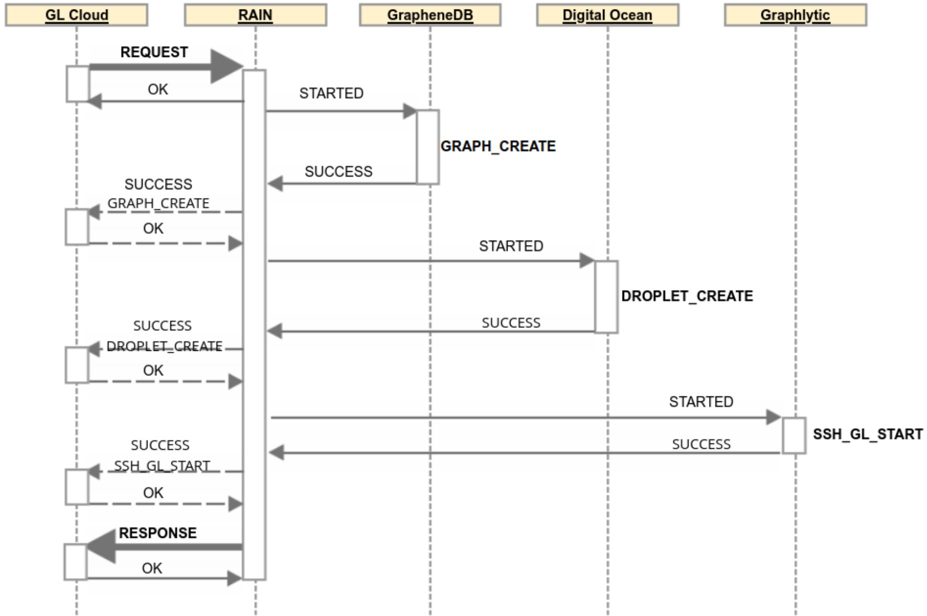

Figure 3. Sequence diagram of the processing of a simple requisition by RAIN

In Figure 3 we illustrate the processing of a simple requisition. This requisition consists of a simple workflow of 3 operations:

- `GRAPH_CREATE` – instantiate a Neo4J [19] graph database in GrapheneDB
- `DROPLET_CREATE` – instantiate new virtual server in the Digital Ocean infrastructure
- `SSH_GL_START` – start the Graphlytic application, using the created droplet to run it and the Neo4J instance to store its data

As shown in Figure 3, the requisition processing is asynchronous from the API call itself. As soon as the requisition's initial parameters (provided in the API call) are stored in the database, the API returns a reply to the user, stating that the requisition has been initiated. The requisition processor then chooses the appropriate workflow, and starts executing its operations. After each operation the user is notified (via a provided callback service) of the requisition's progress. Once the whole workflow is finished, the callback service will receive the final `RESPONSE` with the parameters needed for the use of the created application. This approach allows us to give sufficient responsiveness to the whole, sometimes very lengthy, application creation process, which is very important for a commercial service which has to show the user that the work on their request is processing.

5 WORKFLOW DEFINITION

From the point of view of the RAIN user, the most visible component, and practically the only one which is easy to modify, are the workflow definitions.

Currently the workflow definitions are stored as Python scripts inside the directory structure of the RAIN installation – that is, they are static for users without administrator access. This is not a significant problem in a single-user installation of RAIN, but as noted in Section 4, we plan to introduce a multi-user version of RAIN. Not all of the users will have the administrator level access to the RAIN installation, and so they will not be able to update the workflow scripts. Therefore, in the future, we will move the workflow definitions into the RAIN database, and introduce a REST API for working with them. This API will as a minimum allow to

- list available workflow definitions,
- add new workflow definitions,
- remove existing workflow definitions, or
- update existing workflow definitions.

A workflow definition is a Python script, containing one variable named `workflow_list`. This variable is a dictionary, containing pairs of names of workflow definitions and the definitions itself. Each workflow definition is a list of operation definitions. Each operation definition can be either a simple string representing the operation's name, if this operation has no additional parameters, or it can be a tuple consisting of the operation's name (string) and a dictionary of operation parameters. An example is shown in Listing 1.

```
workflow_list =\
{
  "workflow_name_1":
  (
    "OPERATION_1",
    "OPERATION_2",
    "OPERATION_3"
  ),
  "workflow_name_2":
  (
    "OPERATION_1",
    ("OPERATION_4":
      {
        "delay": 60,
        "timeout": 600
      }
    )
  )
}
```

Listing 1. Example of a workflow definition script

In Listing 1, we see two workflow definitions – `workflow_name_1` and `workflow_name_2`. The first one contains three operations – `OPERATION_1`,

OPERATION_2, OPERATION_3. The second workflow definition contains OPERATION_1 and a parametrized OPERATION_4 with two parameters, delay and timeout.

Workflow parameters can be either intended for the control of operation execution (described in Section 5.1), or for the control of the execution of the operation itself. The latter ones are usually different for each operation and their meaning is context-dependent, while the former ones control operation timing, re-execution, failure or success conditions, and failure handling and recovery.

5.1 Basic Operation Parameters

```
1  'REBOOT_APPLICATION':
2  (
3      ('REBOOT_DROPLET', {'continueAfterError': True}),
4       'POWERCYCLE_DROPLET',
5  ),
6  'HARD_POWERCYCLE_DROPLET':
7  (
8      ('POWEROFF_DROPLET', {'continueAfterError': True}),
9      ('SLEEP', {'delay': 20}),
10      'POWERON_DROPLET'
11 ),
12 'RESIZE_DROPLET':
13 (
14     ('POWEROFF_DROPLET',
15         {'condition': (('$appSnapshot', '==', "offline"),)}
16     ),
17     ('SNAPSHOT_DROPLET',
18         {'condition': (
19                         ('$appSnapshot', '==', "offline"),
20                         ('$appSnapshot', '==', "online"),
21                          "or"
22                        )
23         }
24     ),
25     ('POWERON_DROPLET',
26         {'condition': (('$appSnapshot', '==', "offline"),)}
27     ),
28      'DOCKER_DF_SET_GL_MEMORY',
29     ('NEO4J_SET_CONFIG',
30         {'dockerComposeFile': 'docker-compose.yml',
31          'sshLogin': 'redacted'}
32     ),
33     ('SSH',
34         {'commandLine': 'cd gd && docker-compose up --build -d',
35          'sshLogin': 'redacted'}
36     ),
37      'RESIZE_DROPLET',
38      'POWERON_DROPLET',
39     (OpType.DROPLET_WAIT_ACTIVE,
40         {'timeout': 300}
41     ),
42      'SET_SWAP'
43 )
```

Listing 2. Illustration of various workflow parametrization options

Listing 2 illustrates the use of several operation parameters available to the user.

- Line 10 contains the operation POWERON_DROPLET without any parameters.

- Line 8 contains the operation `POWEROFF_DROPLET` with one parameter `continueAfterError`, which allows the workflow to continue execution even after this operation fails.
- Line 9 contains the operation `SLEEP` with the parameter delay equal to 20. This will pause the execution of the worlfow for 20 seconds before this operation is initiated.
- Line 39 contains the operation `DROPLET_WAIT_ACTIVE` with the parameter `timeout` equal to 300. This means that if this operation is unable to finish in 300 seconds, it will be aborted rather than forcing the whole workflow to wait an arbitrary amount of time.
- Line 29 contains the operation `NEO4J_SET_CONFIG` with two parameters, `dockerComposeFile` and `sshLogin`. These parameters do not influence the execution of the workflow, instead they are operation-specific variables which provide necessary details to the primitive implementing the operation.
- Line 17 contains the operation `SNAPSHOT_DROPLET` with one parameter called `condition`. This allows to make the operation's execution conditional, depending on the current state of some parameters. The details of using conditions will be explained below.

Apart from these parameters, there are several others not shown in Listing 2.

- `repeats` and `repeatDelay` influence how many times will an operation be attempted before the workflow execution is considered unsuccessful, and how long will the workflow execution wait after every failed repeat of the operation before it is attempted again. The default values are 3 repeats, and 30 seconds, but the parameters can change those values. As practice has shown, sometimes it is necessary to allow some operations on infrastructure to be repeated to receive a positive result.
- `overrides` contains a dictionary of parameter names and their values, which can override actual parameter values. Normally, primitive parameters, as shown in Listing 2, operation `NEO4J_SET_CONFIG` on line 29, are used only if such parameter is not already in the list of parameters of the workflow, available in the database. But parameters listed in the `overrides` dictionary have priority even over the parameters existing in the database.

5.2 Operation Execution Conditions

RAIN allows for conditional execution of some operations of the workflow which defines a requisition. This behavior is controlled by a logical formula consisting of conditions and logical operators (*and*, textitor, textitnegation and textitexclusive or) and conditions. It is written in postfix format as a Python list. For example the logical formula $(a > b)$ and $(b > c)$ will be written as $((a > b),\ (b > c), \text{'and'})$.

The conditions in the formula are evaluated to true or false. They can contain variables, constants, and several unary or binary operators, and are written as a list.

In the case of binary operators infix notation is used, that is the operator in between the two operands. Currently available operators are:

- equals, written as $==$ or *eq*. This operator evaluates to *true* if the operands are equal, where comparison is done by applyting the Python operator `==` to the operands.

  Example: $\$parameter == 30$, written as $('\$parameter', '==', 30)$.

- not equal, written as *!=* or *ne*. This operator evaluates to *true* if the operands are not equal, where comparison is done by applying the Python operator `!=` to the operands.

  Example: $\$parameter! = 30$, written as $('\$parameter', '! =', 30)$.

- greater than, written as $>$ or *gt*. This operator evaluates to *true* if the first operand is greater than the second operand, where comparison is done by applying the Python operator `>` to the operands.

  Example: $\$parameter > 30$, written as $('\$parameter', ' >', 30)$.

- less than, written as $<$ or *lt*. This operator evaluates to *true* if the first operand is less than the second operand, where comparison is done by applying the Python operator `<` to the operands.

  Example: $\$parameter < 30$, written as $('\$parameter', ' <', 30)$.

- greater or equal, written as $>=$ or *ge*. This operator evaluates to *true* if the first operand is greater or equal to the second operand, where comparison is done by applying the Python operator `>=` to the operands.

  Example: $\$parameter >= 30$, written as $('\$parameter', ' >=', 30)$.

- less or equal, written as $<=$ or *le*. This operator evaluates to *true* if the first operand is less or equal to the second operand, where comparison is done by applying the Python operator `<=` to the operands.

  Example: $\$parameter <= 30$, written as $('\$parameter', ' <=', 30)$.

- exists, written as *exists* or *ex*. This unary operator evaluates to true if the given parameter exists in the list of parameters available to the operation.

  Example: exists loginName, written as ('ex', 'loginName').

- empty, written as *empty* or *em*. This unary operator evaluates to true if the given parameter is empty (has no value).

  Example: empty loginName, written as ('em', 'loginName').

As can be seen in the provided examples of conditions, an operator can be prefixed by the $ sign. That will force the evaluation of the operator from the list of available parameters, by its name. Otherwise, the operator is treated as a value. This allows the conditional execution of operations, based on the current state of the requisition (by the state we mean the list of parameters at the time of the evaluation of the condition).

The definition of conditions, as described above, is certainly cumbersome. It is currently a prototype, meant to evaluate the feasibility of conditional execution of operations. It has already proven useful, and we are now working on a more practical and streamlined format of conditions, written as strings and using completely the infix notation.

6 SUMMARY AND FUTURE WORK

In this paper, we have described the basic architecture and concepts of a cloud automation tool, designed specifically for commercial operations and successfully working in a commercial environment for several years. Its main attributes are infrastructural agnosticism, high fault tolerance, easy adaptation to new infrastructure, responsiveness and flexibility. The experimentation with the service's design has been ongoing for several years, and currently we believe that the architecture and concepts described in this article serve our purpose well and are worth following in similar endeavours.

During our experimentation with cloud automation, we have identified several important capabilities of the automation tool which significantly increase success rate and customer comfort. These are:

- operation timing – delaying the start of the operation, as well as aborting an operation after a given amount of time;
- failure recovery – repeating a failed operation several times and waiting between consecutive operation executions, as well as ignoring certain errors which do not impact the overall success of the whole requisition;
- conditional operation execution – we have found out that the used cloud infrastructure and services are sometimes controlled in a non-trivial manner, and we need to be able to decide during run-time whether to do certain operations or not, based on the results of the previous operations. That is the reason why we have developed a system of operation conditions (see Section 5.2).

Our future work will be targeting expansion into the new infrastructure of other providers, support for specifics of handling applications using AI methods, and more complex concepts of and capabilities of the currently used infrastructure (automated backups, more complex network configurations and others).

Among the more practical expansion of our cloud automation tool, we plan to introduce multi-user support (described in Section 4). We also plan to better integrate user-side workflow handling by introducing a dedicated REST API component for listing, adding, removing and updating workflow definitions (for details see Section 5). Finally, as mentioned in Section 5.2, we will change the condition notation into a more natural one, describing the logical formula as one string in infix notation.

Acknowledgement

This work is supported by the VEGA grant No. 2/0131/23 and the APVV grants No. APVV-20-0571 and APVV-23-0430.

REFERENCES

[1] AAG IT Services: Headline Cloud Computing Statistics for 2024. 2024, `https://aag-it.com/the-latest-cloud-computing-statistics/`.

[2] Bobák, M.—Hluchy, L.—Belloum, A. S. Z.—Cushing, R.—Meizner, J.—Nowakowski, P.—Tran, V.—Habala, O.—Maassen, J.—Somosköi, B.—Graziani, M.—Heikkurinen, M.—Höb, M.—Schmidt, J.: Reference Exascale Architecture. 2019 15th International Conference on eScience (eScience), 2019, pp. 479–487, doi: 10.1109/eScience.2019.00063.

[3] Meizner, J.—Nowakowski, P.—Kapala, J.—Wojtowicz, P.—Bubak, M.—Tran, V.—Bobák, M.—Höb, M.: Towards Exascale Computing Architecture and Its Prototype: Services and Infrastructure. Computing and Informatics, Vol. 39, 2021, No. 4, pp. 860–880, doi: 10.31577/cai_2020_4_860.

[4] Bobák, M.—Hluchý, L.—Tran, V.: Methodology for Intercloud Multicriteria Optimization. 2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), 2015, pp. 1786–1791, doi: 10.1109/FSKD.2015.7382217.

[5] Hochstein, L.: Ansible: Up and Running. 1st Edition. O'Reilly Media, Inc., 2015.

[6] Myers, C.: Learning SaltStack. Packt Publishing, 2016.

[7] Tovmasyan, K.: Mastering AWS CloudFormation: Plan, Develop, and Deploy Your Cloud Infrastructure Effectively Using AWS CloudFormation. Packt Publishing, 2020.

[8] Cowell, C.—Lotz, N.—Timberlake, C.: Automating DevOps with GitLab CI/CD Pipelines: Build Efficient CI/CD Pipelines to Verify, Secure, and Deploy Your Code Using Real-Life Examples. Packt Publishing, 2023.

[9] Belmont, J. M.: Hands-on Continuous Integration and Delivery: Build and Release Quality Software at Scale with Jenkins, Travis CI, and CircleCI. Packt Publishing, 2018.

[10] Arora, T.—Shigihalli, U.: Azure DevOps Server 2019 Cookbook: Proven Recipes to Accelerate Your DevOps Journey with Azure DevOps Server 2019 (formerly TFS), 2nd Edition. Packt Publishing, 2019.

[11] McKeown, M.: Microsoft Azure Essentials Azure Automation. Pearson Education, 2015.

[12] Krief, M.: Terraform Cookbook: Efficiently Define, Launch, and Manage Infrastructure as Code Across Various Cloud Platforms. Packt Publishing, 2020.

[13] Mott, L. V.: The Development of the Rudder: A Technological Tale. Texas A&M University Press, 1997.

[14] Vetter, S.—Dhaliwal, N.—Mashhour, A.—Röll, A.—Rosca, L.: IBM AIX Enhancements and Modernization. IBM Redbooks, 2020.

[15] Pathania, N.: Learning Continuous Integration with Jenkins: A Beginner's Guide to Implementing Continuous Integration and Continuous Delivery Using Jenkins. 2nd Edition. Packt Publishing, 2017.

[16] Uphill, T.: Mastering Puppet. Packt Publishing, 2014.

[17] Zamboni, D.: Learning CFEngine 3: Automated System Administration for Sites of Any Size. O'reilly Media, 2012.

[18] Chintale, P.: DevOps Design Pattern: Implementing DevOps Best Practices for Secure and Reliable CI/CD Pipeline (English Edition). BPB Publications, 2023.

[19] Vukotic, A.—Watt, N.—Abedrabbo, T.—Fox, D.—Partner, J.: Neo4j in Action. Manning Publications Co., 2015.

Ondrej Habala is a researcher at the Institute of Informatics of the Slovak Academy of Sciences. He works mainly with distributed computing systems and cloud systems, applying them towards solving domain-specific problems mainly in meteorology and hydrology. He has participated in more than 10 national and international research projects, including EU FP5, FP6, FP7, H2020 and HE projects. He is the author of more than 80 publications in his research field.

Martin Šeleng is a researcher at the Institute of Informatics of the Slovak Academy of Sciences. He specializes in research infrastructures, cloud computing, and machine learning. He has participated in several research projects, including the FP5-FP7, H2020 and HE European research programs. He is the author of over 50 scientific publications.

Michal Habala is the Chief Executive Officer and co-founder at Demtec, s.r.o. He is a visionary in the realm of breakthrough graph visualization tools, leveraging his extensive background as both an IT and business consultant. He holds Master of Science degrees in both software engineering and business management, he possesses a unique blend of technical expertise and strategic insight. With a keen eye for innovation and a passion for solving complex problems, he has dedicated his career to empowering clients with transformative solutions. His unique blend of technical expertise and strategic understanding has enabled him to spearhead the implementation of cutting-edge visualization technologies, changing the way companies analyze and interpret their data.

Ľubor Stuhl is the Chief Technology Officer at Demtec, s.r.o. He is an dynamic IT professional and a versatile full-stack developer with a profound grasp of a broad spectrum of technologies. His academic prowess is underscored by his attainment of his Master's degree in economic informatics. Serving as the Chief Technology Officer (CTO) at Demtec, he shoulders the responsibility for steering the technological trajectory of the company. His expertise extends even further as Demtec's primary AI authority, guiding the company through the uncharted waters of this rapidly evolving field. His role as the company's main AI expert is emphasized by his commitment to pushing the boundaries of technological advancement, propelling Demtec towards a future where AI integration is not just a possibility but a strategic imperative.

Michal Staňo is a machine learning researcher at the Institute of Informatics of the Slovak Academy of Sciences. He specializes in federated and distributed learning systems and blockchain technologies, applying them to solve domain-specific problems. He is currently involved in the AI4EOSC and AI4CC research projects, focusing on federated learning models, blockchain, and cryptocurrency applications using tools like TensorFlow and PyTorch. He has a background in data science, process engineering, and digital technology and is pursuing his Ph.D. in computer science. He has contributed to several research publications in his field.

Ladislav Hluchý is a senior researcher and the head of the Department of Parallel and Distributed Information Processing at the Institute of Informatics of the Slovak Academy of Sciences. He has been active in European research programs since FP4, and has led II SAS team in dozens of research projects in FP4, FP5, FP6, FP7, H2020 and HE. Over his research career he authored over 150 scientific publications. His specialization is distributed information processing, cloud computing and data science.