Nubo: A Middleware for interacting with Cloud Platforms

Saurabh Srivastava
Indian Institute of Technology, Kanpur
ssri@iitk.ac.in

T.V. Prabhakar
Indian Institute of Technology, Kanpur
tvp@iitk.ac.in

ABSTRACT

This paper envisions the idea to build a Middleware which can ease the hassle of interacting with multiple cloud platforms. We propose a platform independent language to perform regular and often mundane tasks on varying underlying cloud platforms. The language called Nubo, will have a syntax resembling languages like English, making it easy to learn and recollect. We argue that Nubo is easily extensible and present a demo interpreter for a language subset as a Proof-of-Concept for evaluation.

KEYWORDS

Privacy Engineering, Artificial Intelligence, Software Engineering

1 INTRODUCTION

Driven by an urge to cut initial costs, and utilise available resources to a greater extent, more and more enterprises are switching to cloud based solutions[20]. The Amazon Web Services became the market leader in offering solutions over a public cloud platform[26]. There are others like Microsoft, Google and IBM, trying hard to catch up. For the enterprises who wish to have a private or hybrid cloud environment, solutions like Openstack[24] are becoming fairly popular.

The solution space in cloud offerings are varying, and are evolving quickly. Any IT Infrastructure team tasked with migrating legacy applications to cloud, or facilitating development of new applications with an intend to be deployed on the cloud face the challenge of getting themselves accustomed to the platform(s) of interest. This often forces the IT team to resist usage of two or more dissimilar platforms for use, since it would force them to learn the intricacies of all of them. Such scenarios in turn, can lead to the problem of vendor locking[22].

In this work, we propose building a middleware called Nubo, which aims to fill this gap. The word "nubo" means "cloud" in a *constructed international language* called *Esperanto*[27]. One of the goals of constructing such a language was an ability to express ideas, irrespective of nationalities. Nubo, aims to perform the same task for various cloud platforms. It is inspired by the popular relational database language called SQL[9]. Even though there are a variety of RDBMS available in the market provided by different vendors, almost all support a set of common commands under SQL to interact with their implementations. Another aspect of SQL that made it the de facto standard across RDBMS solutions is its English-like syntax, making it easy to remember. Likewise, Nubo is planned to provide a uniform view of the underlying cloud platforms to the user, with a syntax that closely resembles some natural language, making it easy to remember. Another aspect of SQL worth pointing out is that different RDBMS solutions may provide features that are not standard across industry. Hence the commands and clauses that are supported under SQL do differ from vendor to vendor. Nevertheless, since the overall SQL still resembles greatly to the ANSI SQL, the users generally do not need to learn a different language for every vendor. Because of similar reasons, we intend to keep Nubo extensible, so that vendor specific commands and clauses can be added easily to the overall language.

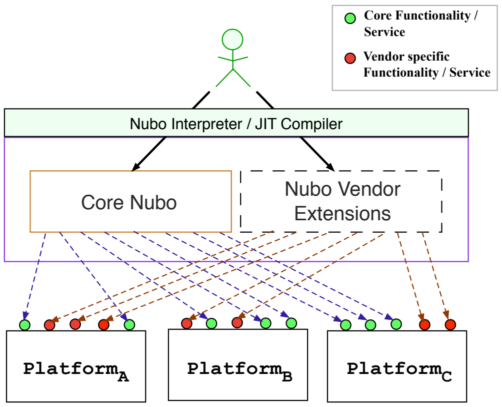
The rest of this paper is organized as follows. We first describe the Nubo Middleware. We enlist the merits of using such a layer, citing its probable applications. We then discuss a particular implementation paradigm which can be used to build such a layer. We do that by defining a basic set of easily extensible grammar rules, which can be parsed by an interpreter to provide the said functionality. We then move over to describing a sample demo interpreter that we built as a proof-of-concept to provide better insights into the intuitions given in the paper. We then wrap up providing a brief overview of previous attempts to build languages for cloud domain and conclude with listing out the opportunities that Nubo can open up in future.

2 THE NUBO MIDDLEWARE

Adding a layer to an existing Software System is generally a decision that requires severe introspection. The reason for the same comes from the body of knowledge in the domain of Software Architecture and Engineering. At an abstract level, a software layer can be viewed as a trade-off between certain quality attributes[16] of the overall system. The addition of an extra layer often introduces a penalty on the *performance* attribute, making up on the fronts of other traits such as *modifiability*, *usability* or *maintainability.* Building a layer of Nubo aims at achieving a similar trade-off.

Figure 1 depicts the concept of the Nubo layer in a nutshell. Every platform exposes a set of features or services. These service may have different names, different communication protocols, as well as different workflow. In any case, it may still be possible to consolidate some of these features under a same name. For example, both *EC2*[13] by AWS and *Nova*[24] by Openstack are essentially a scalable *Compute* service. The Nubo Layer consists of two components. The **Core Nubo** component presents a uniform, consolidated view of common services offered by all vendors, e.g. the compute service. The **Nubo Vendor Extensions** component leaves space for implementing constructs that can provide access to any vendor specific features or services, not offered by others, e.g. AWS Lambda[14].

The user interacts with either an interpreter or a just-in-time compiler for Nubo. The commands that belong to the core Nubo have no differences in their syntax or clauses, irrespective of the platform for which the command is intended. For most cases, it should suffice to set a *platform type* when the interpreter starts, or via a separate command to change the intended platform type later on. For some peculiar cases, where a *cross-platform* deployment scenario is in place, the same can be provided at per command basis too. For commands that belong to the Vendor Extensions, the interpreter will raise an alarm, if the command is attempted for a



Core Functionality / Service
Vendor specific Functionality / Service
Nubo Interpreter / JIT Compiler
Core Nubo
Nubo Vendor Extensions
Platform$_A$
Platform$_B$
Platform$_C$

Figure 1: The abstract representation of Nubo Layer

platform it in not meant to be. The Nubo layer essentially provides a *per-command, per-platform* implementation for all the Core Nubo commands, and a *platform only* implementation for commands in Vendor Extensions. The implementation may use services provided by the vendor such as REST API or SDK to fulfil the requests.

The Nubo layer may perform some or all of the below functions:

(1) The layer hides a number of mundane tasks required to implement an abstract functionality, and fill them with operations to perform these tasks in a *default* fashion. For example, in order to create a VM on an IaaS platform, the users are required to create a Key-pair, define a set of rules to allow or restrict traffic on the machine, create subnets and routers etc. In the demo application we've built, we do the same over AWS.
(2) The layer provides an opportunity for *heuristics-based* decision making for resource creation and manipulation on the underlying cloud platform. For example, the layer may choose an availability zone that is near to the user's geographical region when creating a VM. It can also be used to provide enhanced Quality of Service, such as by replication of resources across multiple availability zones (in case the underlying platform doesn't support it implicitly).
(3) The layer can implement *optimization techniques* meant for better resource utilization and reduction of costs for certain use-cases. For example, the layer can implement a kind of *lazy execution* paradigm for some operations. In the demo application, we give an example of such a scenario by decoupling the process of "creating a VM" from "switching it on" - two steps that are combined by AWS implicitly.
(4) The layer can handle scenarios of cross-platform resource creation by deploying parts of a deployment stack over multiple cloud platforms. It can also be useful in scenarios like *cloud bursting*[21]. This can ease the complexities for enterprises wishing to have a Hybrid Cloud environment, or the deployment chooses resources dynamically from some marketplace (e.g. AWS EC2 spot instances[1]).
(5) An unconventional advantage of building such a layer may come in the field of training. If we go back to our database analogy, the usage of SQL greatly simplifies the aspect of learning how to interact with Relational Databases. Compare that to other, more formal and detailed methods like Relational Algebra[10], and one can easily figure out why SQL became so popular. Since Nubo heavily borrows its overall structure from SQL, it can be an easy starting point for someone interested in knowing more about Cloud Computing in general, without requiring to go through a more formal and detailed method.

3 IMPLEMENTING A BASIC NUBO LAYER

The task of implementing a language involves a number of steps. We divide the task into the following phases:

(1) *Lexical Analysis* of the character stream, to produce distinctly identifiable tokens such as keywords and identifiers.
(2) *Syntax Analysis* of tokens to verify that they do not appear out of place, and follow some language pattern.
(3) *Semantic Analysis* of the statements to see if they make sense or not in the language context.
(4) *Implementation* of the operations that are requested in the statement.

Before proceeding any further, we must underline the fact that the implementation of the above steps presented by us is "one" plausible way to build the Nubo layer. There can be other, possibly better implementations that may come up in future.

3.1 Proposed Nubo Grammar

We combine the first two steps defined above into a single step and term the output as **Nubo Grammar**. We have chosen to write the grammar using a well known format called Extended Backus–Naur Form or EBNF in short. We define the language in a top-down fashion. The starting production for Nubo is:

program : *statement*+;

meaning that a Nubo program consists of *one or more* statements. A statement can be one of the following type:

statement : *create* | *delete* | *attach* | *detach* | *modify* |
use | *show* | *filter* | *turn*;

The different statements include:

create : CREATE *any_entity* NAME *having*? ';';
delete : DELETE *any_entity* NAME ';';
attach : ATTACH *attachable_entity* NAME TO
attached_to_entity NAME ';';
detach : DETACH *attachable_entity* NAME FROM
attached_to_entity NAME ';';
modify : MODIFY *modifiable_entity* NAME *set* ';';
use : USE *usable_entity* NAME ';';
show : SHOW (*any_entity* NAME |
ALL *any_entity*) ';';
filter : FILTER (ALL)? *any_entity* *with* ';';
turn : TURN (ON | OFF)
switchable_entity NAME ';';

The statements use some sub-statements or clauses:

having : HAVING *attribute_value_pair*
(',' *attribute_value_pair*)+ ';';
set : SET *attribute_value_pair*
(',' *attribute_value_pair*)+ ';';
with : (NOT)? WITH *attribute_comparison_pair*

```
        ( ',' attribute_comparison_pair )+ ';';
attribute_value_pair : attribute '=' value ';';
attribute_comparison_pair :
        attribute ('>' | '<' | '>=' | '<=' | '=') value ';';
attribute : CORES | STORAGE | IMAGE | IP | FS |
        CAPACITY | KEY_FILE | ADDRESS |
  ALL_TRAFFIC | IN_PORT_ON | IN_PORT_OFF |
    OUT_PORT_ON | OUT_PORT_OFF | CIDR ';';
value : NUMBER | STRING | BOOL |
        ALLOW | DISALLOW ';';
```

The *attach*, *delete* and *modify* statements create, destroy and manipulate resources on the cloud respectively. The *use* statement puts in use, a particular project, under which other resources reside. The *show* statement displays details about a particular resource (e.g. the memory, storage, cores etc. of a VM) whereas the *filter* statement displays a list of all resources of a particular type, that follow some criteria (e.g. all VMs with 2GB memory). The *attach* and *detach* statements form and remove associations between two entities respectively (e.g. a Key-pair with a VM). The *turn* statement is used to turn on/off a resource (e.g. a VM).

The tokens written in the *lower* case are processed during the *syntax analysis*, whereas those written in CAPITAL case are defined for the *lexical analysis* phase.

The lexical tokens are defined in such a way that the language becomes *case insensitive.* To save space, we are not putting all those definitions here, but they are of the form:

```
KEYS : [Kk] [Ee] [Yy] [Ss];
```

In essence, this allows us to consider "keys", "Keys", "KEYS" or even "kEyS" as the same lexical token. There are some lexical tokens, where we can allow the user to write a different word altogether, yet meaning the same thing. For example, the rule for keyword TURN can be written like this:

```
TURN : [Tt] [Uu] [Rr] [Nn] |                // accepts "turn"
       [Ss] [Ww] [Ii] [Tt] [Cc] [Hh];       // or "switch"
```

This means that the user can write a phrase "turn on vm" or "switch on vm", but the language considers it the same. This is a nice and clean way to allow additions of *synonyms* for operations and entities within the language framework, without changing anything upstream. This can also mean that we can add words from languages other than English, or add colloquial words for entities and operations to the overall language model. We discuss more on this when we talk about extensibility of Nubo.

There is one specific lexer rule that we would like to mention before we move ahead. It is the rule for defining a NUMBER in Nubo. Unlike usual numbers, Nubo numbers can take a *quantifier* like "M" or "G". The basic idea behind it is to allow users to specify numbers in terms of computing world units such as Megabyte or Gigabyte. The rule for a NUMBER looks something like this:

```
NUMBER : ( [0-9]+ (.)? | [0-9]* [.] [0-9]+ )
         ( MEMORY_UNITS )?;
MEMORY_UNITS : [Mm] | [Gg] | [Tt];
```

This allows us to take inputs like "2", "2.", "2.5" as well as "2.5g" as valid numbers. Of-course, this does put on some extra workload in the semantic analysis, to check if the input is semantically correct (e.g. "memory = 2g" makes sense, "cores = 5m" doesn't).

The *_entity rules used in different statements, just group *entities* into various groups, based on the applicability of the operation on them. To save space, we are not putting all those definitions here, but they are of the form:

```
any_entity : PSPACE[1] | VM | SUBNET | ROUTER |
             VOLUME | KEYS | POLICY;
```

By decoupling the allowed entity types into a separate group using the *_entity rules, we make it fairly easy to add new operations to the language. A new operation can creates a new *_entity rule to point to, making other operation remain exactly how they have been. This also allows rolling out language features in stages, instead of being required to implement all at the same time. For instance, if a new operation applies to all entities in the system, yet we wish to allow the implementation to go one at a time, we can create a rule for the same, and keep adding entities to it, as and when we have implemented the functionality.

The overall grammar can be summarized as follows:

- A Program consists of one or more statements.
- A statement implements a consolidated feature or service, like creating a resource or managing associations between two resources.
- Each statement begins with a *verb*, followed by *entity types* and *entity names*, with optional *quantifiers* and *prepositions* concluding in a possibly empty list of *attribute-value pairs.*
- We've defined a set each for representing *entities* (resources and projects), their *attributes* (like a VM's memory, or a Subnet's CIDR), and the *values* that attributes can take.
- **Note** that the first two steps only check for statements that are *syntactically* correct. They may still be incorrect semantically (e.g. an attribute-value pair like "ip = ALLOW" is syntactically correct, but semantically incorrect).

3.2 Extending the proposed grammar

We discussed the existence of **Nubo Vendor Extensions** in Section 2. The Vendor Extensions are basically a way to extend Nubo for non standardized features and services. To do so, we must take a moment, and make a comment about the *extensibility* of the language grammar. To add a new operation to the language, we just add it to the list of alternatives for a *statement* rule. Although the statement itself, can have any peculiar syntax (not necessarily following the general description above) as per the requirements, the current grammatical constructs may be easily reused to select a subset of entities and receive attribute values for them. In this regard, our discussions on the usage of *_entity rules and the sample scenario presented for accepting customized numeral inputs in Section 3.1 can be helpful to grasp the overall extensibility of the language.

We summarize the typical steps that can be taken to add a new operation to the existing grammar:

(1) Create a rule for the new operation to be added to the language grammar. For example,
```
demo_op : DEMO_OP demoable_entity NAME ';';
```
If any sub-statements or clauses are required, create a rule similarly for them too.

(2) Create a rule to include the entities that are going to be affected by this new operation. For example,
```
demoable_entity : VM | VOLUME;
```

[1]**PSpace** or Private Space is just another term for **Project**

This allows a step by step inclusion of the functionality, say one entity at a time.
(3) Create lexer rules for the operation keywords, and any new constructs or entities that will be added to the language. Refer to Section 3.1 for sample implementations.
(4) Finally, add the operation rule to the rule that selects one of the statements in the language:
statement : *create* | *delete* | *attach* | *detach* | *modify* |
use | *show* | *filter* | *turn* | ***demo_op***;

We still however, need an efficient way to parse this grammar, and execute the operations they represent. In the next section, we go into details of how this can be done in a neat way. We will discuss a sample interpreter implementation, that can parse and perform an end-to-end operation on a small subset of the above grammar.

4 NUBO DEMO INTERPRETER

We implemented a small subset of the overall commands in Nubo as part of a demo interpreter. The interpreter is available for evaluation as a public GIT repository[5]. The source code is available under the MIT license[15]. It is built as a Maven[2] project, which makes the process of building it from scratch fairly straightforward. The tool requires a subscription on the AWS platform. In thoery, we could have chosen any public cloud platform for doing the same. The decision to choose AWS is inspired by two factors. First, AWS is arguably the most popular among developers. A demo interpreter over the same can hence be more easily appreciated. In particular, our arguments about lazy execution and default behaviour can be easily grasped by a regular AWS user. Second, AWS has modes of operation, where students and researchers can easily obtain significant amount of credits to create virtual resources. The AWS Free Tier[4] and the AWS Educate[3] facilites can be used to get access to the platform, for the sake of evaluating the interpreter.

For implementing the Proof-of-Concept interpreter, we choose a popular and robust tool called **ANTLR**[23] (**AN**other **T**ool for **L**anguage **R**ecognition). Antlr is a *parser generator* that takes as input, a grammar written in EBNF, and produces Java source files, which can parse the given grammar. Antlr does almost everything short of the actual semantic interpretation of the language. A sample parse tree produced by Antlr for a given statement can be seen in figure 2.

We used Antlr's *listener* interface to implement the demo interpreter. Antlr produced a lexical analyser, and a parser for Nubo from the given grammar files. The listener interface provides methods which are essentially *callbacks* invoked when the parser discovered a specific language construct. For a sample Nubo statement:

CREATE PROJECT *my_proj*;

the callbacks invoked during the parsing of the statement are as shown in figure 3. There are two callbacks associated with each grammar construct which are invoked when the parser reaches and leaves the corresponding node in the parse tree. A method called `visitTerminal` is called for parsing all occurrences of a `TerminalNode`. Clearly, this paradigm might end up producing too many methods to implement. Antlr therefore, produces a *base listener* class file, which provides empty definitions for all these methods. It only leaves us then, to extend this class, and override only those methods, that we wish.

The demo interpreter makes use of AWS Free Tier services for showing the Proof of Concept. The resources that can be created in the Free Tier are limited, but suffice to explain simple use-cases. We now summarize the implementation details of the demo parser:

(1) **Performing tasks behind the scene:** The demo interpreter attempts to cover a simple use-case for a new user on cloud platforms, i.e. creating a Virtual Machine with some common parameters. As discussed in section 2, we used the demo interpreter to showcase some merits that such a layer can exhibit. One such merit was that such a layer can implement some default mundane operations on behalf of the user, without the user coming to know about them. For instance, when a *create* statement for a *project* (similar to the previous example), is parsed by the layer, the demo Nubo interpreter:
   - creates a *VPC* or Virtual Private Cloud in AWS
   - creates a *subnet* within the VPC
   - creates an *Internet Gateway* and attaches it to the VPC
   - creates a *Routing Table* and associates it to the VPC
   - creates a *Route* that routes all traffic originating or ending in the created subnet to the attached Internet gateway
   - configures the subnet such that it attaches a *Public IP* automatically to any VMs created on it.

   The interpreter makes HTTP Requests using the AWS Java SDK to perform each one of these steps, transparently from the user. Please note that we need the Internet gateway and the routing system, so that the newly created VM is accessible to the user at the start. Similarly, when asked to create a VM, the interpreter quietly creates a default *Key-pair* and a *Security Group* that allows SSH communication to go through.
(2) **Making smart decisions on behalf of user:** Different cloud platforms have different ways to initialize a VM. In any case, almost every platform expects the user to select an *image template* and an *instance type* before the VM can be created. The demo interpreter:
   - keeps an offline archival of instances meta-data and images (permitted under the Free Tier) available for use
   - takes up a generic user query to find an image, e.g. a query like "Ubuntu 14"
   - makes a keyword search over the name, description and other information available in the images' meta-data, and tries to find a suitable match for the user's requirements
   - tries to find an instance type with specifications closest (higher or same) to the user requirements.

   The scope of the demo interpreter is fairly restrictive (in terms of available resources), but the idea can still be gleaned from the implementation of the interpreter.
   Another smart paradigm that the interpreter deploys is *lazy execution*. In general, cloud platforms merge the operations of "creating" and "launching" a VM. This is due to the fact that in most cases, a first time boot-up of the machine is necessary, to bake in certain user specific information in the VM (e.g. the RSA Key for access to the machine). Nubo decouples these two operations. However, the cloud platform still only allows the combined process of "laucnhing" a VM.

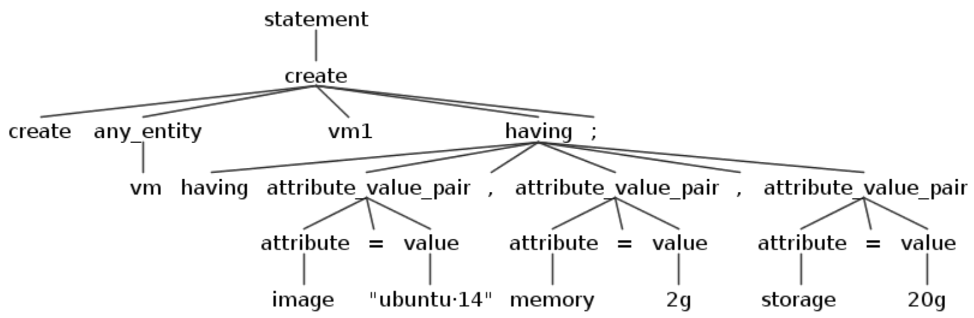
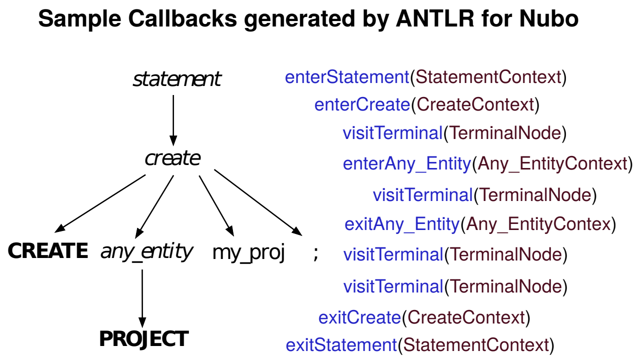


Figure 2: An example Parse Tree for a Nubo Statement



```
enterStatement(StatementContext)
  enterCreate(CreateContext)
    visitTerminal(TerminalNode)
    enterAny_Entity(Any_EntityContext)
      visitTerminal(TerminalNode)
    exitAny_Entity(Any_EntityContext)
    visitTerminal(TerminalNode)
    visitTerminal(TerminalNode)
  exitCreate(CreateContext)
exitStatement(StatementContext)
```

Figure 3: Parser callbacks for Nubo statements

To imitate this as a two-step process, Nubo creates a template, created from the user specifications as described above. It will also perform the background mundane tasks, like creation of keys and groups, but will not issue a call to launch the VM. The actual launch is issued when the user "switches on" the VM for the first time. This means no resources are occupied on the cloud at start, until the user wishes to use them.

(3) **Supporting users from non English background:** Another useful aspect of such a Middleware could be removing hurdles in front of those who are not native speakers of English language. The advanced features such as a Command Line Interface (CLI) still expect the users to know English language. To show that Nubo can be *natural language independent*, we attempted to support **French** locale, in addition to English in the demo interpreter.

- At the keyword level it required practically no change in the code. We only provided alternate lexical tokens like below:

  KEYS : [*Kk*] [*Ee*] [*Yy*] [*Ss*] | // *English*
  [*Cc*] [*Ll*] [*Éé*] [*Ss*]; // *French*

  This means that we received the same sequence of callbacks whether the user types *KEYS* or *CLÉS*.
- Even though this may be a simple way to hint users about operations that a statement performs, a better interpreter would need to consider changes in languages at the sentence structure as well. In the demo interpreter, we made one such change in the grammar:

  turn : (TURN (ON | OFF)) |
  COMMENCÉ | ARRÊT)
  switchable_entity NAME ';';

  This means, that the turn command can now take four probable shapes, "*TURN ON*" is equivalent to "*COMMENCÉ*", whereas "*TURN OFF*" is equivalent to "*ARRÊT*". This did require a small if condition to be added to the code, since the decision making in case of English and French now differs slightly.
- Printing messages in French required implementing a standard procedure of keeping locale specific files, with messages in respective languages:

```
cmd_success = Command succeeded   // English
cmd_success = Commandement réussi // French
```

- Here are the outputs produced by the interpreter for English, as well as French locales for similar commands.

  English

  create vm vm1 **having image**="ubuntu 14", **memory**=2g, **storage**=20g;

  Chosen a VM type with 2.0 GB memory and 1 cores.

  French

  créer vm vm1 **ayant image**="ubuntu 14", **memory**=2g, **storage**=20g;

  Choisi un type VM avec 2.0 Go de mémoire et de 1 processeurs.

The demo interpreter has been built in Java over OpenJDK 8 (although it should work with Java 7 with minor changes).

5 RELATED WORK

A number of attempts have been made in past to build languages specific to the cloud domain[12][8][17][7][18]. Cloud# was probably the first detailed attempt to model resources on a cloud. A notable feature of the Cloud#[19] model, was to segregate computational entities from the actions that can be performed over them. CloudML[11] is a description language which can be used by vendors and developers to describe their offerings and requirements respectively. The aspect of CloudML that is worth noting is that they used XML as the underlying technology to build their idea, attempting to provide a uniformity across different vendors, taking a step further towards platform independent Middleware. Cloud DSL[25] is another solution worth mentioning with respect to Nubo, since it inspires to build a common vocabulary of terms to cover a wide range of IaaS offerings. They also tried building a system more usable than the previous attempts, by providing a graphical view of the entities and their relationships.

In our work, we stress more on building a solution over time, which can provide platform independent constructs to the user, keeping scope alive for vendor specific custom behaviour. We reiterate that the Nubo layer is a Middleware, whereas the Nubo implementation we presented is an early attempt to build such a layer. Also, instead of defining a rigid set of language constructs, Nubo expects gradual addition and refinements over a period of time, driven by use-cases. Also, the Nubo layer is intended to encourage development of new innovative implementation patterns and inclusion of more heuristics based decision making so that the overall load at the IT teams in the organisations are reduced. Lastly, Nubo implementation stresses more on a syntax that is intuitive and close to natural languages, so that the overall learning curve for a new user is not steep.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented the idea of a layer that acts like a Middleware between users and cloud platforms. The layer, that we call Nubo aims to relieve the complexities associated with heterogeneity of cloud solutions offered by different vendors, in Public as well as Private Cloud models. We discussed the merits of building such a language, which includes performing mundane and repetitive tasks, making informed decisions on part of users, and implementing smart strategies to optimize overall deployment experience for IT teams in enterprises. We then went on to discuss a plausible set of steps to implement such a layer, including a set of grammar rules in EBNF to capture common constructs in the language, and building a parser over them using Antlr.

In this paper, we presented a basic sketch of Nubo, along with a toy application to see how the language may look like. Some of the future opportunities that Nubo puts forward are:

- **NuboScript:** We mentioned that one of the major inspiration for Nubo was SQL. The analogy can be further extended to build something called *NuboScript* on the lines of PL/SQL[6] by adding basic programming language constructs like conditionals (IF-ELSE, SWITCH-CASE etc.) and iterators (FOR, WHILE, FOR-EACH etc.) to the language, enabling users to fine-tune their resource deployment scenarios.
- **Supporting server-less computing:** The popularity of AWS Lambda shows a growing realisation of providing function level abstractions to users for implementing business logic. Nubo can be extended to provide a generic abstraction of server-less computing, over platforms that do not support it implicitly. The Middleware will create and manage any underlying resources transparently, while the user only cares about a function being performed some where in the cloud.
- **Building applications over Nubo layer:** Vendor-locking often puts constraints on developers to choose sub-par options for certain services from the same vendor, even though better alternative exists being offered by a different vendor. Nubo can evolve as a go-to option to offload deployment challenges, allowing developer to build applications over it, without being pinched by the IT team over choosing platforms.

REFERENCES

[1] *Amazon EC2 Spot Instances.* https://aws.amazon.com/ec2/spot/.
[2] Apache Maven Project. https://maven.apache.org/. Accessed: 2018-01-28.
[3] AWS Educate. https://aws.amazon.com/education/awseducate/. Accessed: 2018-04-07.
[4] AWS Free Tier. https://aws.amazon.com/free/. Accessed: 2018-04-07.
[5] Nubo Demo Interpreter. https://bitbucket.org/ssri5/nubodemointerpreter. Accessed: 2018-08-18.
[6] *Oracle Database 12c PL/SQL.* http://www.oracle.com/technetwork/database/features/plsql/index.html.
[7] D. Ardagna, E. Di Nitto, G. Casale, D. Petcu, P. Mohagheghi, S. Mosser, P. Matthews, A. Gericke, C. Ballagny, F. D'Andria, et al. Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 50–56. IEEE Press, 2012.
[8] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. Tosca: Portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.
[9] M. Blasgen, M. Astrahan, D. Chamberlin, J. Gray, W. King, B. Lindsay, R. Lorie, J. Mehl, T. Price, G. Putzolu, M. Schkolnick, P. Selinger, D. Slutz, H. Strong, I. Traiger, B. Wade, and R. Yost. System r: An architectural overview. *IBM Systems Journal*, 38(2.3):375–396, 1999.
[10] E. F. Codd. *The Relational Model for Database Management: Version 2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
[11] G. Goncalves, P. Endo, M. Santos, D. Sadok, J. Kelner, B. Melander, and J.-E. Mangs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 399–406, Nov 2011.
[12] I. Houidi, W. Louati, D. Zeghlache, and S. Baucke. Virtual resource description and clustering for virtual network discovery. In *Proceedings of ICC*, 2009.
[13] A. Inc. *Amazon Elastic Compute Cloud (Amazon EC2).* Amazon Inc., http://aws.amazon.com/ec2/, 2008.
[14] A. Inc. *What Is AWS Lambda?* Amazon Inc., http://aws.amazon.com/lambda/, 2015.
[15] O. S. Initiative et al. The MIT license, 2006.
[16] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010.
[17] G. P. Koslovski, P. V.-B. Primet, and A. S. Charao. Vxdl: Virtual resources and interconnection networks description language. In *Networks for Grid Applications*, pages 138–154. Springer, 2009.
[18] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar. Cloudscale: a novel middleware for building transparently scaling cloud applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 434–440. ACM, 2012.
[19] D. Liu and J. Zic. Cloud#: A Specification Language for Modeling Cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 533–540, July 2011.
[20] C. Low, Y. Chen, and M. Wu. Understanding the determinants of cloud computing adoption. *Industrial Management & Data Systems*, 111(7):1006–1023, 08 2011.
[21] S. K. Nair, S. Porwal, T. Dimitrakos, A. J. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and A. U. Khan. Towards secure cloud bursting, brokerage and aggregation. In *Web services (ecows), 2010 ieee 8th european conference on*, pages 189–196. IEEE, 2010.
[22] J. Opara-Martins, R. Sahandi, and F. Tian. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing*, 5(1):1–18, 2016.
[23] T. Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013.
[24] D. Radez. *OpenStack Essentials.* Packt Publishing Ltd, 2015.
[25] G. C. Silva, L. M. Rose, and R. Calinescu. Cloud DSL: A Language for Supporting Cloud Portability by Describing Cloud Entities. *CloudMDE 2014*, page 36, 2014.
[26] Synergy Research Group. Amazon leads; Microsoft, IBM & Google chase; others trail. https://www.srgresearch.com/articles/amazon-leads-microsoft-ibm-google-chase-others-trail, Aug 2016.
[27] H. Tonkin. *Esperanto, interlinguistics, and planned language*, volume 5. University Press of America, 1997.