A Web Service for Cloud Metadata

Michael Smit, Przemyslaw Pawluk, Bradley Simmons, Marin Litoiu
York University, Canada
{msmit,ppawluk,bsimmons,mlitoiu}@yorku.ca

***Abstract*—Descriptive information about available cloud services (i.e., metadata) is required in order to make good decisions about which cloud service provider(s) to utilize when deploying an application topology to the cloud. Presently, there are no uniform mechanisms for describing these services. Further, there is no unifying process that aggregates this metadata from the set of cloud providers and makes it available to a user in a programmatic fashion from a single location. This paper presents a methodology for and an implementation of a service-oriented application that provides relevant metadata information describing offered cloud services via a uniform RESTful web service. The data provided by this service is automatically acquired and mapped to a standard ontology. Community members can submit performance benchmarks using a metrics agent that submits metrics via a web service. Several example applications using this API to help users select resources are presented.**

***Keywords*-web service, RESTful, metrics, instances, intercloud, cloud, metadata**

I. Introduction

The choice of which cloud service providers (and which resources offered by those providers) to deploy an application topology to is a highly complex and difficult question to answer. A variation of this question is also encountered at runtime when an application's elasticity policy [1] requires that cloud resources be added to or removed from the application deployment.

A growing number of providers (CloudHarmony lists 99[1]) offer widely varied services, describing their offerings according to self-defined methodologies. For example, Amazon uses what they refer to as EC2 Compute Units (ECU) and number of cores to express the CPU capabilities of what it calls *instances*, while Rackspace defines CPU as a (unpublished) proportion of the physical host machine with a fixed number of virtual cores for what it calls *flavors*[2]. Choosing from among these various providers and the hundreds of instance options can be difficult.

In the long-term, there is movement toward interoperability, a desire to avoid provider lock-in, and an interest in moving seamlessly from one cloud to another. *Federated clouds*, where a variety of clouds are accessed uniformly, are enabled in part by projects like Apache Deltacloud[3] and Apache libcloud[4] which provide uniform access for deploying and accessing cloud instances. The need for interoperability and intercloud protocols [2] and a vision of utility-oriented cloud federation [3] have been described previously. We have proposed an intercloud broker [4] that would make such decisions autonomically. Each of these projects expresses a need for uniform access to metadata about cloud providers and the resources they provide. Even for the most basic information (what instances are offered, how are they configured, what do they cost), little information is available. Amazon provides a JSON version of their prices only. Although libcloud offers a `list_sizes` API call, it is implemented by hard-coding even the most basic information (CPU, RAM) in the source[5]. Rackspace provides an API to list their instances, but it includes only RAM, local storage, and the name.

We believe an initial step toward solving this problem is a metadata service listing the available cloud services, their properties, and some basic cross-cloud metrics for comparing instances. This paper describes such a service along four main dimensions: selecting information, acquiring information, representing that information, and accessing that information (Section II). Selecting information requires identifying what metadata is useful from available information and organizing that information in a usable manner. We identify metadata in three categories: provider-level metadata, resource-level properties, and resource-level metrics. Information is acquired automatically using a combination of existing APIs and using a cloud abstraction layer (Deltacloud) to start instances and acquire information from them directly. Our collection efforts are made possible by a RESTful API [5] for submitting new metrics, which could also be used by the community to contribute. Information is represented in a relational database using a simple information ontology designed to facilitate cross-provider comparisons. Information is accessed using a RESTful API, fully documented and available online to any user. Information can also be accessed using a service that renders API calls in HTML for viewing by users, or by using one of the two applications implemented to demonstrate the RESTful API (the Instance Browser and the Instance Search AJAX applications).

This paper describes four novel contributions. First, our identification and organization of metadata about cloud providers and the resources they offer is provider-agnostic and extensible. Second, we introduce a functional RESTful API to provide programmatic access to this metadata, and to receive updated information from the community. This service is like the central metadata repository called for in efforts to standardize service measurements (e.g. [6]). Third, we describe a novel method for acquiring metadata automatically, testing the cloud resources empirically to obtain configurations and

[1] http://cloudharmony.com/clouds
[2] We'll adopt the term *instances* as it is more common.
[3] http://deltacloud.apache.org
[4] http://libcloud.apache.org
[5] https://github.com/apache/libcloud/blob/trunk/libcloud/compute/drivers/

performance benchmarks. Fourth, we present two applications that consume information from the API to support searching and browsing available cloud computing resources.

We begin by describing the design (Section II) and implementation (Section III) of our service, including the organization and automated acquisition of metadata. The sample applications are presented in Section IV. Finally, we describe work related to cloud metadata (Section V), discuss the current challenges and future directions for the design of cloud metadata services (Section VI), and conclude the paper (Section VII).

II. Designing a Cloud Metadata Service

A cloud metadata service provides information about the properties of cloud-related services. This metadata encompasses various IaaS, PaaS, and SaaS services (we illustrate IaaS compute services in our current implementation[6]). The primary consideration is the intended use of this information: selecting a provider, and a resource offered by that provider, based on some criteria. We exclude the actual resource acquisition decision (RAD) problem [4] and consider only the information required to make the decision. The actual decision can be made by an automated tool (e.g. [4]) or by the user themselves using a user interface to the information.

Given this problem statement, we identified the requirements of a cloud metadata service as follows:

1) Be accessible programmatically
2) Contain information to help compare cloud service providers
3) Store and represent information in a provider-agnostic fashion
4) Be extensible to include new information
5) Update automatically or semi-automatically (minimal manual curation)
6) Provide accurate information (no reliance on untrusted user-submitted information)
7) Scale to large-scale deployments

The following subsections describe meeting these requirements, which we categorize as Choosing Relevant Metadata, Acquiring Metadata, Representing Metadata, and Providing Access to Metadata. This service has been implemented, and can be accessed at `http://cloudymetrics.com`. Our implementation will be used as a running example; it currently focuses on IaaS compute services, though future versions will include other cloud services.

A. Choosing Relevant Metadata

Given our requirements, we need sufficiently detailed information in order to compare providers. This implies a need to move beyond provider-specific advertisements and metrics to something standard. We cannot produce metrics that are too abstract, because abstraction implies existing preferences and weighting on particular metrics. It must also be information that we can acquire automatically (or semi-automatically) while being verifiably accurate. This data must be extensible as standard metrics are adopted (e.g. SMI [6]).

Our service considers three primary types of information: provider-level metadata, resource-level properties, and resource-level metrics. We provide examples of each as implemented currently, but fully expect these lists to expand as other cloud services are included in our implementation.

Provider-level metadata considers information applicable to all resources of a provider, and the properties of the provider itself. For example, in our implementation we include:

- *billing_time* denotes the smallest increment of time billed (e.g. hourly). Prices will be reported by hour regardless of this value.
- *cpu_bursting* is a boolean value indicating whether the CPU has a hard cap (false) or can "burst" to higher levels if the resources are available (true).
- *bandwidth_[in/out]_price* records the price in dollars of the first monthly 2TB of bandwidth in/out[7].
- *bandwidth_price_scales* is a boolean value indicating whether the price of bandwidth changes some time after the 2TB mark.

Resource-level properties include constant reportable properties about the *resources*. These numbers will be the same each time they are retrieved. For IaaS compute services, the resources are instances; for IaaS storage services, the resources would be storage accounts, and so on. Because the current focus of our implementation is on IaaS compute services, the properties reported are instance-level properties. Our implementation also assigns each unique instance a canonical name that uniquely identifies it within the database. For example, an Amazon `m1.small` instance in the us-east availability zone running linux is named `m1.small.us-east.linux`. The complete list of properties is as follows:

- *name* is the unique name of the instance, used internally as a unique identifier.
- *display_name* is the name used by the provider to describe the instance.
- *cpu_name* denotes the name of the processor used in this instance. This is typically the Processor Brand String from the `cpuid` command.
- *cpu_clock* represents the clock frequency of the processor as reported, in MHz.
- *cpu_cores* is the number of cores available.
- *cpu_proportion* represents the proportion of each CPU core guaranteed to be available to this instance. This value is not always available.
- *memory* denotes the amount of memory (in MB).
- *local_storage* designates the amount of local disk including used and free (in GB).
- *price* is the cost of this instance in dollars per hour[8].

Resource-level metrics include measurable values about the resources. Unlike fixed properties, these metrics may

[6]It is our intention to extend this to PaaS and other IaaS services

[7]2 TB is empirically a number that covers most use-cases, while still being a fairly constant cost even for providers that offer volume discounts

[8]This value may be computed when a per-hour rate is not provided.

vary with each measurement. They are primarily micro-benchmarks designed to measure only one component of a system's performance (CPU speed, storage write speed, DNS update time, etc.). While resource-level properties are theoretically comparable, when provider-level properties are taken into account the comparisons do not work in practice. The metrics provide another avenue for comparison, without abstracting away the detail (like the Amazon Elastic Compute Unit or the CloudHarmony Cloud Computing Unit do). These leaves the decision about which properties are important to the application or user choosing the provider and resource. The metrics currently reported include micro-benchmarks of network speed, disk reads and writes, CPU processing, and boot time; specifically, we include:

- *ping_time*: the time required to ping www.google.com (averaged over 10 attempts, with the DNS lookup already cached).
- *local_read*: the speed (in MB/s) of reading local instance storage; calculated using `hdparm`[9] non-cached reads, averaged over three tests.
- *remote_read*: the speed (in MB/s) of reading remote storage (like Amazon EBS) where applicable; calculated using `hdparm` non-cached reads, averaged over three tests.
- *local_write*: the speed of writing to local instance storage, calculated by writing 1 GB to a file using `dd` from `/dev/zero`.
- *remote_write*: the speed of writing to remote storage (like Amazon EBS) where applicable, calculated by writing 1 GB to a file using `dd` from `/dev/zero`.
- *dl_speed*: the download speed (in MB/s not Mbps), calculated by downloading a 512MB file from a fixed location.
- *cpu_bench_<benchmark>*: A set of (fairly) standard CPU benchmarks as implemented by the `hardinfo`[10] tool: blowfish, cryptohash, fibonacci, n-queens, fft, and raytracing. Smaller is better.
- *startup_time*: how long it takes from requesting this instance to being able to access it via SSH.

B. Acquiring Metadata

As discussed previously, only some providers provide API access to information about their instances, and when they do it is to small subsets of relevant information. We specified a requirement to acquire information automatically (manual curation of metadata is time-consuming and therefore expensive); however, we also require accurate information. To meet this requirement, we used a service-oriented solution that allows a loosely- and temporarily-coupled infrastructure to obtain and report these metrics.

We begin with the resource-level properties that are available to us programmatically; the prices from Amazon, the basic CPU/memory specifications from Rackspace, etc. We then augment with provider-level metadata that we enter manually; though this is manual curation, it is for a small number of providers and is not expected to change frequently.

We acquire the remaining information (missing resource-level properties and all resource-level metrics) automatically by starting each instance and obtaining the information empirically; we also allow community submissions. There are three primary components to this automatic metadata acquisition service system (AMASS).

The **metadata submission endpoint** allows accepts programmatic submissions of JSON-formatted text. It is implemented as a REST endpoint (`http://api.cloudymetrics.com/api/submit/metrics`) that accepts `POST` submissions. This information is stored temporarily and presented to an administrator, who can approve it for addition. A desired format is specified, permitting any property or metric name, and requiring one special field, the name of the instance. An optional field allows the remote user to specify an authentication code to verify their identity. Our automated centralized metadata collector is assigned an authorization code. Codes can be distributed to the community in various ways; a small-scale solution would be off-line informal trust agreements; at a larger scale, a framework for establishing or negotiating trust can be employed (e.g., [7], [8]).

While community members can employ any programmatic client they wish to submit metrics, for convenience a **metadata gathering script** allows end users to gather and submit metrics to the metadata submission endpoint described above. This free download from the metadata website runs a series of system commands to acquire properties and metrics, formats them according to the specified JSON format, then submits them. This tool allows for a large community of users to provide metrics and properties for a variety of providers and instances; any submitter temporarily joins a loosely-coupled distributed architecture. The script is used by the centralized metadata collector.

In addition to soliciting input from the community, we have developed an automated approach to gathering metadata and submitting it via the metadata submission service. The **centralized metadata collector** (CMC) identifies instance types in the database with missing information, starts instances, and runs the metadata gathering script on those instances. Using Deltacloud as an abstraction layer allows the CMC to support multiple providers without additional implementation effort. The script completes within approximately 20 minutes, which means this task is not prohibitively expensive. Certain information (e.g. price) is not available from the system itself, but all instance metrics are collected using this service, as well as most instance properties. Although the CMC was designed to be run by the site administrator, in theory any community member could also use the CMC to submit metrics.

The AMASS strikes a reasonable balance between automatic updates and manually-curated information. This approach is extensible; the methodology is not limited to only compute instances.

[9]http://sourceforge.net/projects/hdparm/
[10]http://hardinfo.berlios.de/

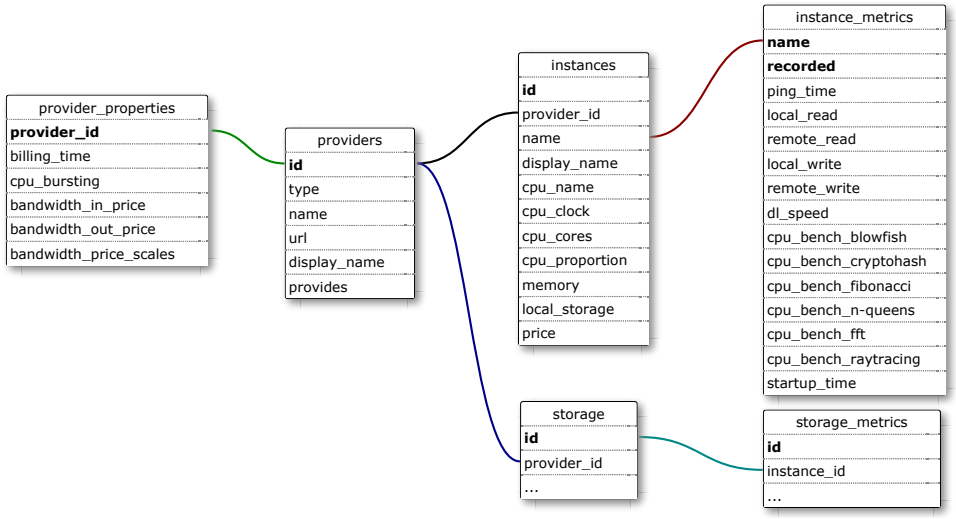

Fig. 1: The as-implemented database schema.

C. Representing Metadata

Representing and accessing information are closely related in a RESTful service, as the REST service paradigm maps URIs to resources. We use a design intended to grow as new metrics, properties. resources, and providers are added (Figure 1).

One table lists providers, including their type and which types of resources they provide. A full set of provider-level metadata is in the provider_properties table. For each type of resource (instance, storage) we maintain the name and properties in one table, with the metrics in a second table (details are provided for instance properties and metrics only). Any number of records are allowed for metrics; the rows are date stamped so the latest metrics can be selected.

This data structure allows the addition of PaaS providers and resources in their own tables, the addition of IaaS resources in their own tables (like storage and private networks) with links to IaaS providers, and the addition of classes of metrics. The RESTful architecture is easily updated as the underlying data model is extended.

D. Providing Access to Metadata

The information is programmatically accessible via a RESTful API that returns JSON; the service is documented at the base service URI, `http://api.cloudymetrics.com/api/`. The documentation provides links to working examples of each REST resource described in this section. For convenience, all API calls can be returned as an HTML table instead (with sortable columns) to assist with application design and debugging: the developer can use any web browser to view the results of an API call in human-readable format (Figure 2). To produce HTML instead of JSON, the invoker uses the endpoint `http://api.cloudymetrics.com/api/`.

The design of the REST service is closely linked to the representation of the data in the database. The REST service will adapt to match the underlying data model. Each REST URI is designed to intuitively describe the resource it returns; for example, `/iaas/amazon/instances/m1.small/metrics` translates to "for the IaaS-provider Amazon's instances matching m1.small, return metrics". To illustrate the design decisions regarding providing access to the metadata, we describe here the various REST resources available in CloudyMetrics, as follows.

`/iaas`: List all known providers (By name as used in this application, and by the common name, e.g. Amazon AWS.) Currently stored internally but not yet provided via the API is which IaaS service this provider offers: compute, storage, etc.

`/iaas/<provider>`: List the provider-level metadata (see §II-A). The keyword `all` can be used in place of a provider name, here and for all REST resources, to return results for all known providers.

`/iaas/<provider>/instances`: List the resources of type "compute" (known commonly as "instances"), along with all resource-level properties (see §II-A) and the name of the provider.

`/iaas/<provider>/instances/<name>`: List the instances whose display name or assigned unique name match the given string (wildcard matching is provided automatically). All resource-level properties are returned.

`/iaas/<provider>/instances/search`: Searches all instances of the given provider, using a set of parameters provided in a specially-formatted query string. The results returned included the instance provider, name, and all resource-level properties. The query string (i.e. the text following a '?' in the URL) must be a url-encoded set of [field] [operator] [value] triples, joined by &.

- *field*: Any resource-level property or metric name (see §II-A); valid fields can be listed using `/iaas/all/instances/search/list`.
- *operator*: Any of $\sim$, $=$, $==$, $<$, $<=$, $>$, $>=$. If the field is non-numeric, only $=$, $==$, and $\sim$ may be used. The

Results

provider	billing time	cpu bursting	bandwidth in price	bandwidth out price	bandwidth price scales
amazon	hourly	0	0	0.12	1

Fig. 2: The result of calling `/html/iaas/amazon`.

$\sim$ is an "approximately" operator, and is only useful if the value includes wildcards (% or *).

- *value*: The desired value of the field. Must be numeric for numeric fields. If the $\sim$ operator is used, wildcards may be used in the value (* or %). Note the % must be properly url-encoded as %25 .

A field can be included in the query string multiple times to define ranges (e.g. `price<1&price>.1`); all triples are combined using a logical AND operator.

`/iaas/<provider>/instances/search/list`: Lists the valid metrics and properties that can be searched for; because new properties and metrics can be added, applications should periodically obtain an updated list of search fields.

`/iaas/<prvdr>/instances/<name>/metrics`: Lists all known metrics (see §II-A) recorded for the given instance name. This includes each recorded micro-benchmark for that instance, listing the date of the benchmarking and the results. The instance name is wildcarded by default (so using `m1.small`) will return all benchmarks for all instances with "m1.small" in their name.

To limit the metrics shown, an optional GET parameter `show` in a standard query string may be used. Only metrics in a comma-separated no-whitespace list to the `show` parameter will be shown (all metrics will be shown if the parameter is not included). For example, the query string `?show=ping_time,dl_speed` would show only the ping time and download speed of the instance (along with usual information like the name, provider, and recorded time).

`/iaas/<p>/instances/<n>/metrics/<metric>`: Performs the same search as the previous resource (all instances matching `name` from `provider`), but returns only the named metric (along with provider, instance name, and date recorded).

III. Implementation Details

The RESTful service API is implemented in PHP, using the Slim Framework[11]. Applications using this framework list the REST resources they offer, and define a *handling function* to process requests arriving at that resource. Resources are identified by URIs, where each element of the URI can be a string constant or a variable. The variable elements of the API are passed as arguments to the handling function. For example, the command `$app->get('/iaas/:provider', 'process_iaas_properties')` instructs the framework to invoke the function `process_iaas_properties` when it receives requests of the form `GET /iaas/<provider>`, passing the value of `<provider>` as an argument.

[11]http://www.slimframework.com/

We have authored an addition to the Slim Framework that allows us to quickly develop functions to handle new REST resources. This extension requires that handling functions specify an SQL query to acquire the desired results, specify which fields of that query should be filtered out or modified, and then pass control to our extension. The extension executes the query, performs the post-processing, and returns the appropriate JSON or HTML format. If SQL queries are structured to return all columns of a table, changes to the underlying data model will be reflected immediately in the results returned by the resource. New handling functions following this template require only knowledge of SQL.

The AMASS is a loosely coupled system, where the various components are implemented in Java and PHP. The PHP metrics gathering script invokes various command-line utilities to produce the micro-benchmarks. The support for updating the database with submitted metrics after administrator approval is provided by a standalone PHP script. A set of PHP scripts called *populators* are used to acquire information from existing APIs. Additional populators can be added; a library is provided to facilitate development.

The HTML version of all the API resources is produced by formatting the JSON returned as HTML, using the JSON Report library[12]. The HTML returned should be viewable by any HTML5-capable browser.

The metadata is stored in a MySQL database using the schema described in §II-C. Unique ids are defined for all of the major tables, though these are used only internally and are never publicly exposed. These unique ids are the primary key.

To test this implementation, we deployed it to a small web server with 256MB of RAM and 1.6% of four virtual cores of a Quad-Core AMD Opteron Processor @ 2 GHz (burstable), running Ubuntu 10.04. Apache 2.2.8, PHP 5.2.4, and MySQL 5.0.95 were installed. Apache was tuned to run at most 8 threads, limiting the number of simultaneous clients to 8. Using Apache Benchmark[13], we tested API calls with concurrency of 100, sending blocks of 10,000 requests: numbers designed to heavily load the server.

We measured the total throughput for the duration of the test; the results are in Figure 3. The bars show the throughput for various REST endpoints. `amazon` was used as the provider for all tests. `search(2)` and `search(4)` were called with two search terms and four search terms, respectively. `metrics` was called for the `m1.small` instance. The line plot on the same figure shows the number of rows returned by the call. The volume of results returned appears to be

[12]http://ajaxstack.com/jsonreport/

[13]http://httpd.apache.org/docs/2.0/programs/ab.html

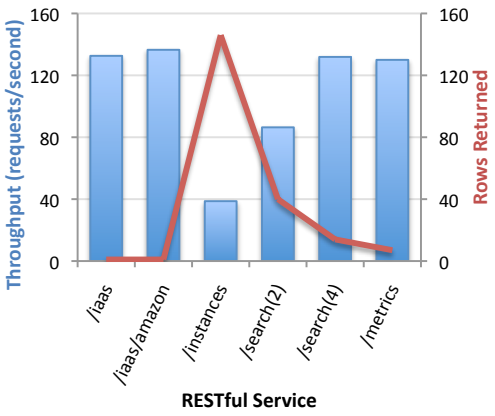

Fig. 3: The throughput for the REST APIs on a small web server.

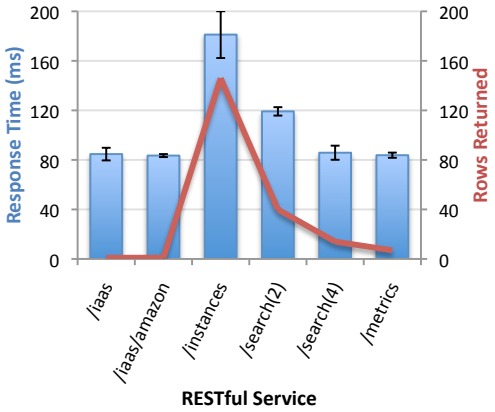

Fig. 4: The response time for the REST APIs on a small web server.

the primary factor affecting throughput, with the `search` endpoint performing better with four terms (and 14 results) than it did with 2 terms (and 40 results). The list of all instances had the slowest performance, returning only 39 results per second.

We also tested the response time of individual requests with the same configuration, except sending one request at a time, 100 times. The results are shown in Figure 4, and show a pattern similar to the throughput numbers.

As the database grows in size, the performance will degrade as more results are returned. An immediate solution is to limit the number of results returned by any API call, and require applications to introduce pagination. At present, however, performance is sufficient; individual requests return in 80-180 milliseconds.

IV. Sample Applications

To illustrate the use of the cloud metadata API, we developed two prototype applications. These applications are designed to help an individual make a decision about which cloud provider and what resource offered by that provider is the right one. The user is assumed to have the ability and power to make the decision if provided with accurate

Fig. 5: The starting page of the Instance Search application.

information. (For an example of an application accessing the API programmatically and making this decision automatically on behalf of the user, see [4].)

A. Instance Search

For this application, a familiar search interface is adopted (Figure 5). The user enters a query in the form of triples, [field][operator][value]. Multiple tuples are separated by any amount of white space, and are automatically ANDed together in the query. An auto-complete script makes suggestions for fields and values by making asynchronous background calls to the web service. The format of the triple is based on the `search` REST API, where the field is any resource metric or property (see §II-A), the operator is any of $\sim$, $=$, $==$, $<$, $<=$, $>$, or $>=$, with $\sim$ an "approximately" operator.

An API request is constructed based on the user's query; some basic checking is done client-side, but the query is mostly passed to the server as-is. The query is submitted to the server using an AJAX call, and the results table is updated (Figure 6). The results table updates immediately as new triples are entered; an update can be manually triggered using the "instance search" button (the update is still client-side). Formatted errors are reported to the user. The name of each returned instance is a link to a page listing all of the benchmark results available for that instance type.

This application is available publicly at `http://api.cloudymetrics.com/instance-search/`.

B. Instance Browser

This application addresses a similar use case, but using a different user interface. Because the key terms may be difficult to remember, the interface provides a set of selection boxes to help construct the triples (Figure 7). Changes to any of the selection boxes or text boxes trigger an immediate update to the results table. The display format is similar to the instance search, including the link to a full set of metrics. This application is available publicly at `http://api.cloudymetrics.com/instance-browser/`.

V. Related Work

The problem of discovering a service and the solution of a central index is similar to the service discovery problem in service-oriented architectures, for which the standardized solution is UDDI (Universal Description Discovery and Integration). An OASIS standard[14], UDDI was envisioned as a centralized registry to which service providers would submit

[14]http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

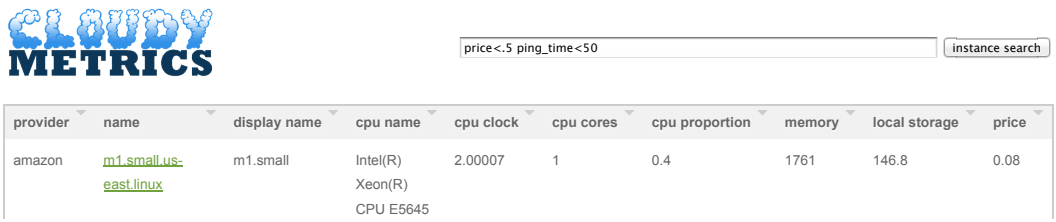

Fig. 6: The results page for the Instance Search application.

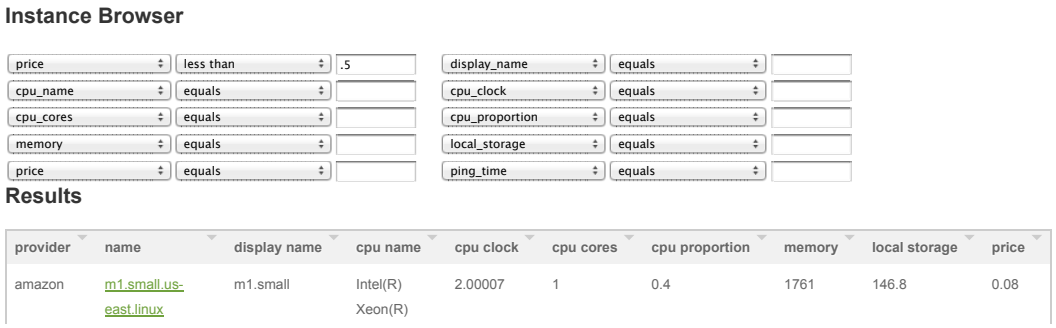

Fig. 7: The starting page of the Instance Browser application.

XML-based descriptions of their WS-* services. Given the low adoption rates, some large organizations have abandoned UDDI (IBM, SAP, Microsoft[15]) and the OASIS technical committee is no longer maintaining it[16]. Other service discovery solutions (see [9]) exist but have not been adopted. As general service listings, they would not be capable of representing the detail of cloud technology that our approach requires. Additionally, our methodology for automatically acquiring service information is unique.

Han et al. [10] describe a recommendation system (RS) for cloud computing. This approach is most suitable for design-time decisions as it is used statically to provide a ranking of available cloud providers. It is not accessible programmatically via an API, and uses only the basic information gathered manually from the provider's websites.

This project fits with other work in our lab that is similarly aimed at providing a cross-cloud adaptive resource management solution [4], [11]–[13]. As a repository of cloud instance metadata, the metrics service is a critical component of an intercloud solution. The term *intercloud* refers to the notion of a cloud of clouds [14]; our work on a cloud broker [4] was initial step toward this goal.

The idea of cross-provider service measurement has been explored in the context of web services and QoS [15]–[17]. Aggarwal et al. [15] consider measurement specifically in the context of web service composition. Measurement of cloud services is a fairly new area of research.

A consortium, led by Carnegie Mellon and CA, has developed the Service Measurement Index (SMI) as a possible approach to facilitate the comparison of cloud-based business services[17]. The SMI is a hierarchical framework that partitions the description of a service into seven categories (i.e., accountability, agility, assurance, financial, performance, security and privacy and usability) with each category being further refined to a set of attributes. An attribute is then expressed as a set of key performance indicators (KPIs) which specify the requisite data to be acquired for every measure / metric [6]. Our approach is not designed to replace the long-term efforts of this project, which works at a higher-level and once realized may allow for powerful expressiveness. However, as the SMI model requires the cooperation of all participating cloud providers, there is no available timeline for when SMI may be available. Our basic micro-benchmarks are fairly low-level, but are straightforward to acquire without multi-enterprise cooperation, while still offering technical users valuable comparative information. In short, the advantage of our approach is that it exists right now, and could eventually be extended with SMI information once it is available.

There have also been attempts to measure cloud service performance [18]–[20]. In all cases services offered by Amazon EC2 have been compared with services offered by other providers: Rackspace [18] and GoGrid [20]. Our approach, using benchmarks to normalize configurations, has not been seen in the literature.

CloudHarmony offers comprehensive benchmarks of the cloud[18], but does not provide the basic listing of what instances are offered by which providers. We envision using our service for initial acquisition of metadata, filtering out the instances which are unsuitable; the filtered list can be narrowed down further by making fewer, less expensive calls to CloudHarmony. Their service requires a paid subscription, while ours is free to researchers and is open-source so can be deployed locally. Other services like Compare the Cloud[19]

[15]http://uddi.microsoft.com/about/FAQshutdown.htm
[16]http://lists.oasis-open.org/archives/uddi-spec/200807/msg00000.html
[17]http://beta-www.cloudcommons.com/web/cc/about-smi
[18]http://cloudharmony.com/benchmarks
[19]http://www.comparethecloud.net

do not offer direct access to metadata, but rather make manual recommendations based on a survey that gathers the requester's requirements.

VI. Discussion and Future Directions

Though the service is useful in its present form, there are a number of improvements for the immediate future. Considering other IaaS services – such as storage, content delivery networks, and DNS services – as well as adding PaaS services are early priorities. Including additional providers is also important for our goals to be achieved.

The instance applications can be improved in several ways, most notably by improving the parsing of queries in the Instance Search application so that the syntax can be less rigid. There are several aesthetic improvements that would make the Instance Browser feel easier to use. Improving the approach to showing additional metrics would also be useful, rather than requiring opening a new page.

The underlying data model allows for multiple sets of metrics describing the same instance type, because as measured information it is entirely possible to get different results. At present the API returns all recorded metrics, but this concept needs further exploration. Returning the average, or the most recent metric, is too easy a solution. Consideration should be given to expiring or supplanting metrics records, and to presenting statistical information about aggregate metrics (standard deviation, etc.). Though there is immediate value in supplying instance properties like price and hardware configuration, a more developed approach to handling metrics and benchmarks is an important next step.

The current implementation uses a MySQL relational DBMS; while the queries involved are not complex and the data is not inordinately large, scalability will be more difficult with a mySQL database than with a NoSQL database.

The current ontology for describing instances assumes that instances will always have fixed size. However, some IaaS providers allow users to choose custom sizes at service deployment time. Mechanisms for supporting this behavior will be required.

While the current implementation is deployed as a centralized service, could also be deployed to manage private/public decisions, or to index the available cloud services at a multi-organization cloud partnership.

VII. Conclusion

This paper motivated and described the cloud metadata acquisition problem, and proposed a solution in the form of a metadata service. This service is accessible via a RESTful API, is updated semi-automatically, and provides micro-benchmarks for provider-agnostic cloud computing instance comparison. The design and implementation of a prototype service were described, and the various features documented. Two sample applications provided user-friendly access to the API were presented. We believe this cloud metadata service will help enable federated and utility computing, and help address the challenge of choosing cloud resources at design, deployment, and run time.

Acknowledgment

This research was supported by CA Inc., the Natural Sciences and Engineering Research Council of Canada (NSERC), Ontario Centre of Excellence (OCE) and Amazon Web Services (AWS).

References

[1] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Proceedings of The 4th IEEE International Conference on Cloud Computing (CLOUD 2011), 4-9, July, 2011 Washington, D.C., USA.*, July 2011.

[2] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud - protocols and formats for cloud computing interoperability," in *Proceedings of the 2009 Fourth International Conference on Internet and Web Applications and Services*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 328–336.

[3] R. Buyya, R. Ranjan, and R. N. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *ICA3PP (1)*, 2010, pp. 13–31.

[4] P. Pawluk, B. Simmons, M. Smit, M. Litoiu, and S. Mankovski, "Introducing STRATOS: A cloud broker service," in *IEEE International Conference on Cloud Computing (CLOUD)*, 2012, p. To Appear.

[5] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, 2007.

[6] CSMIC, "Service measurement index version 1.0," Carnegie Mellon University Silicon Valley, Tech. Rep., 2011.

[7] A. Hess, J. Holt, J. Jacobson, and K. E. Seamons, "Content-triggered trust negotiation," *ACM Trans. Inf. Syst. Secur.*, vol. 7, pp. 428–456, August 2004.

[8] C. H. Yew and H. Lutfiyya, "A middleware-based approach to supporting trust-based service selection," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*, may 2011, pp. 407 –414.

[9] M. Rambold, H. Kasinger, F. Lautenbacher, and B. Bauer, "Towards autonomic service discovery," *Services Computing, IEEE International Conference on*, vol. 0, pp. 192–201, 2009.

[10] S.-M. Han, M. M. Hassan, C.-W. Yoon, and E.-N. Huh, "Efficient service recommendation system for cloud computing market," in *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, ser. ICIS '09. New York, NY, USA: ACM, 2009, pp. 839–845.

[11] M. Litoiu, C. M. Woodside, J. Wong, J. Ng, and G. Iszlai, "A business driven cloud optimization architecture," in *SAC*, 2010, pp. 380–385.

[12] C. Barna, B. Simmons, M. Litoiu, and G. Iszlai, "Multi-model adaptive cloud environments (MACE)," November 2011, http://www.ceraslabs.com/projects/management-services-for-cloud-computing.

[13] B. Simmons, M. Litoiu, D. Ionescu, and G. Iszlai, "Towards a cloud optimization architecture using strategy-trees," in *Proceedings of The 9th International Information and Telecommunication Technologies Symposium (I2TS 2010), 13-15, December, Rio de Janeiro, Brazil*, December 2010.

[14] K. Kelly, "A cloudbook for the cloud," http://www.kk.org/thetechnium/archives/2007/11/a_cloudbook_for.php, 2007.

[15] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, "Constraint driven web service composition in METEOR-S," in *Proceedings of the 2004 IEEE International Conference on Services Computing (SCC 2004*, 2004, pp. 23–30.

[16] Y. N. Li, K. C. Tan, and M. Xie, "Measuring web-based service quality," *Total Quality Management*, vol. 13, no. 5, pp. 685–700, 2002.

[17] A. E. Saddik, "Performance measurements of web services-based applications," pp. 1599–1605, 2006.

[18] "Rackspace cloud servers versus Amazon EC2: Performance analysis," http://www.thebitsource.com/featured-posts/rackspace-cloud-servers-versus-amazon-ec2-performance-analysis/, 2009.

[19] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: comparing public cloud providers," in *Proceedings of the 10th annual conference on Internet measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 1–14.

[20] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 931–945, 2011.