REST Web Service Maintenance Through API Policy Enforcement

Hiranya Jayathilaka, Chandra Krintz, Rich Wolski
Department of Computer Science
Univ. of California, Santa Barbara, CA
Email: {hiranya,ckrintz,rich}@cs.ucsb.edu

***Abstract*—Web services and cloud computing have revolutionized the way software is developed, deployed, and consumed. As a consequence, there has been a proliferation of web services, which developers make accessible to users via web application programming interfaces (web APIs) and cloud-based deployment technologies. Because this model significantly simplifies and expedites deployment of web APIs, it also poses new software maintenance and evolution challenges. In particular, it becomes difficult to track, control, and compel reuse of web APIs, inadequately tested services can be deployed into production, and API changes can be introduced that break API-user code or that breach security or organizational procedures.**

To address these challenges, we investigate a new approach to API governance – combined policy, implementation, and deployment control of APIs for software and data deployed as web services. Our approach, called EAGER, provides a software architecture that can be easily integrated into cloud platforms as a cloud-native feature, and supports system-wide, deployment-time enforcement of API governance policies. Specifically, EAGER can check for and prevent backward incompatible API changes from being deployed into production, enforces service reuse, and facilitates enforcement of other best practices in software maintenance via policies. We also describe a prototype EAGER implementation that integrates with an open source platform-as-a-service cloud and evaluate its feasibility, efficiency, scalability, and effectiveness for enforcing cloud-based API governance.

I. Introduction

The growth of the World Wide Web (WWW), web services, and cloud computing have significantly influenced the way developers implement software applications. Instead of implementing all the functionality from the scratch, developers increasingly offload (*i.e.* as a "mash-up") as much application functionality as possible to remote, web-accessible services, each of which exports its own application programming interface (API). Thus, a modern application often combines local program logic with calls to APIs that interface to cloud-hosted services. This approach significantly reduces both the programming and the maintenance workload associated with the application. In theory, because the APIs interface to software that is curated by cloud providers, the application leverages greater reliability, scalability, performance, and availability in the implementations it calls upon through these APIs than it would if those implementations were local to the application (e.g. as locally available software libraries). Moreover, by accessing common services, developers avoid "re-inventing the wheel" each time they need a commonly available service and the scale at which clouds operate ensures that these services have access to enough request capacity.

As a result, web-accessible APIs and the software implementations to which they provide access are rapidly proliferating. At the time of this writing, ProgrammableWeb [1], a popular web API index, lists more than $11,000$ publicly available web APIs and a nearly 100% annual growth rate [2]. These APIs increasingly employ the REST (Representational State Transfer) architectural style and many of them target commerce-oriented applications (e.g. advertising, shopping, travel, etc.). However, several non-commercial entities have also recently published web APIs, e.g. IEEE [3], UC Berkeley [4], and the US White House [5].

This proliferation of web APIs demands new techniques and systems that automate the maintenance and evolution of APIs as a first-class software resource. API management in the form of run-time mechanisms to implement access control and performance-based service level agreements (SLAs) is not new, and many good commercial offerings exist today [6], [7], [8]. However, support for *API governance* – consistent, generalized, policy implementation across multiple separate APIs in an administrative domain – is a new area of research made poignant by the emergence of cloud computing.

A lack of API governance can lead to security breaches, denial of service (DoS) attacks, poor code reuse, violation of service-level agreements (SLAs), naming and branding issues, and abuse of digital assets by the API consumers. Unfortunately, most existing cloud platforms within which web APIs are hosted provide only minimal governance support, *e.g.* registration and authorization. These mechanisms (available from various commercial vendors such as 3Scale [6], Apigee [7], Layer7 [8]) are important to policy implementation since governance policies often need to express access control specifications. However, developers are still responsible for implementing governance policies that *combine* features such as API versioning, dependency management, and SLA enforcement as part of their respective applications. Moreover, each application must implement its own governance – there is no system for ensuring that the cloud administrators set and enforce policy to which developers must conform.

In contrast API management solutions (which are typically implemented as stand-alone services that are not integrated with the cloud) do attempt to address some governance concerns beyond access control. However, because they are not

integrated within the cloud provisioning environment, their function is advisory and documentarian. That is, they do not possess the ability to implement full enforcement and, instead, alert operators to potential issues without preventing non-compliant behavior. In addition, many of these systems operate outside the cloud that actually hosts the APIs thereby generating an additional cost that may preclude their use. As such, they can fail independently of the cloud, thereby affecting the scalability and availability of the software that they control. Finally, because they are not integrated with the cloud itself it is difficult for them to implement governance at deployment-time – the phase of the software lifecycle during which an API change or a new API is being put into service. Because of the scale at which clouds operate, deployment-time governance is critical since it permits policy violations to be remediated before the changes are put into production (*i.e.* before run-time).

Thus, our thesis is that governance must be implemented as a built-in, native cloud service to overcome these shortcomings. That is, instead of an API management approach that layers governance features on top of the cloud, we propose to provide API governance as a fundamental component of the cloud platform. Cloud-native governance abstractions

- enable both deployment-time and run-time enforcement of governance policies as part of the cloud hosting functionality,
- avoid inconsistencies and failure modes caused by integration and configuration of governance services that are not end-to-end integrated within the cloud fabric itself,
- leverage already-present cloud functionality such as fault tolerance, high availability, elasticity, and end-to-end security implementation to facilitate governance, and
- unify a vast diversity of API governance features across all stages of the API lifecycle (development, deployment, evolution, deprecation, retirement).

As a native functionality, such an approach also simplifies and automates API governance implementation for the administrators or "DevOps" teams responsible for application software deployment and maintenance thereby allowing the separation of administrative and cloud management concerns from development concerns.

To explore the efficacy of cloud-integrated API governance we have developed an experimental cloud platform that supports governance policy specification and enforcement for the applications it hosts. EAGER – **E**nforced **A**PI **G**overnance **E**ngine for **R**EST – is a model and an architecture that is designed to be integrated within existing cloud platforms in order to facilitate API governance as a cloud-native feature. EAGER enforces proper versioning of APIs and supports dependency management and comprehensive policy enforcement at API deployment-time.

Using EAGER, we investigate the trade-offs between deployment-time policy enforcement and run-time policy enforcement. Deployment-time enforcement is attractive for several reasons. First, if run-time only API governance is implemented, policy violations will go undetected until the offending APIs are used, possibly in a deep stack or call path in an application. As a result, it may be difficult or time consuming to pinpoint the specific API and policy that are being violated (especially in a heavily loaded web scalable web service). In these settings, multiple deployments and rollbacks may occur before a policy violation is triggered making it difficult or impossible to determine the root cause of the violation. By enforcing governance as much as possible at deployment-time EAGER implements "fail fast" in which violations are detected immediately making diagnosis and remediation less complex. Further, from a maintenance perspective, the overall system is prevented from entering a non-compliant state, which aids in the certification of regulatory compliance. In addition, run-time governance typically implies that each API call will be intercepted by a policy-checking engine that uses admission control and an enforcement mechanism creating scalability concerns. Because deployment events occur before the application is executing, traffic need not be intercepted and checked "in flight" improving the scaling properties of governed systems. However, not all governance policies can be implemented strictly at deployment-time. As such, EAGER includes run-time deployment facilities as well. The goal of the research is to identify how to implement enforced API governance most efficiently by combining deployment-time governance where possible and run-time governance where necessary.

EAGER implements policies governing the APIs that are deployed within a single administrative domain. Focusing governance on the APIs simplifies both policy specification and the consistent and automatic implementation of policy enforcement. At the same time, it promotes software maintainability by separating the API lifecycle management from that of the service implementations and the client users. That is, APIs are often longer lived than the individual clients that use them or the implementations of the services that they represent. At the same time they represent the "gateway" between software functionality consumption (API clients and users) and service production (web service implementation). Policy definition and enforcement at the API level permits the service and client implementations to change independently without the loss of governance control.

Enforced API governance further enhances software maintainability by guaranteeing that developers reuse existing APIs when possible to create new software artifacts (to prevent API redundancy and unverified API use). Concurrently, it tracks changes made by developers to deployed web APIs to prevent any backwards-incompatible API changes from being put into production.

EAGER includes a language for specifying API governance policies. The EAGER language is distinct from existing policy languages like WS-Policy [9], [10] in that it avoids the complexities of XML, and it incorporates a developer-friendly Python programming language syntax for specifying complex policy statements in a simple and intuitive manner. Moreover, we ensure that specifying the required policies is the only additional activity that API providers should perform in order to benefit from EAGER. All other API governance related verification and enforcement work is carried out by the cloud platform automatically.

To evaluate the feasibility and performance of the pro-

posed architecture, we prototype the EAGER concepts in an implementation that extends AppScale [11], an open source cloud platform that emulates Google App Engine. We describe the implementation and integration as an investigation of the generality of the approach. By focusing on deployment actions and run-time message checking, we believe that the integration methodology will translate to other extant cloud platforms.

We further show that EAGER API governance and policy enforcement impose a negligible overhead on the application deployment process, and the overhead is linear in the number of APIs in the applications being validated. Finally, we show that EAGER is able to scale to tens of thousands of deployed web APIs and hundreds of user defined governance policies.

In the sections that follow, we present some background on API governance and overview the design and implementation of EAGER. We then empirically evaluate EAGER using a wide range of APIs and experiments. Finally, we discuss related work, and conclude.

II. Background

The popularization of network computing and the World Wide Web (WWW) has led to the development and adoption of web services [12] as the technology of choice for implementing modern service-oriented architectures (SOA [13]). The interface portion of a web service, which abstracts and modularizes its service implementation details while making the service network-accessible, is commonly referred to as a *web API*. As far as the users and applications that consume a web service are concerned, the web API is the only point of contact and source of functionality for the underlying service implementation.

Software engineering best practices separate the service implementation from API, both during development and maintenance. The service implementation and API are integrated via a "web service stack" that implements functionality common to all web services (message routing, request authentication, etc.). Because the API is visible to external parties (*i.e.* clients of the services), any changes to the API impacts users and applications not under the immediate administrative control of the API provider. For this reason, API features usually undergo long periods of "deprecation" so that independent clients of the services can have ample time to "get ready" for an API change. At the same time, technological innovations often prompt service reimplementation and/or upgrade to achieve greater cost efficiencies, performance levels, etc. Thus, APIs typically have a more slowly evolving and longer lasting lifecycle than the service implementations to which they interface.

Cloud computing is based on the idea of exposing some digital asset or a capability (*e.g.* compute power, database, etc.) as a highly scalable web service. Mobile devices, due to their limited hardware resources often offload much of their processing and storage needs to remote services running in a "cloud" connected to the Internet. Web APIs play a crucial role in both these paradigms.

As a result, modern computing clouds, especially clouds implementing some form of Platform as a Service (PaaS) [14], have accelerated the proliferation of web APIs and their use. Most PaaS clouds [11], [15], [16] include features designed to ease the development and hosting of web APIs for scalable use over the Internet. This phenomenon is making API governance an absolute necessity in the cloud environments.

In particular, API governance promotes code reuse among developers since each API must be treated as a tracked and controlled software entity. It also ensures that software users benefit from change control since the APIs they use change in a controlled and non-disruptive manner. From a maintenance perspective, API governance makes it possible to enforce best-practice coding procedures, naming conventions, and deployment procedures uniformly. API governance is also critical to API lifecycle management – the management of deployed APIs in response to new feature requests, bug fixes, and organizational priorities. API "churn" that results from lifecycle management is a common phenomenon and a growing problem for web-based applications [17]. Without proper governance systems to manage the constant evolution of APIs, API providers run the risk of making their APIs unreliable while potentially breaking downstream applications that depend on the APIs.

Unfortunately, most web frameworks used to develop and host web APIs do not provide API governance facilities. This missing functionality is especially glaring for cloud platforms that are focused on rapid deployment of APIs at scale. Commercial pressures frequently prioritize deployment speed and scale over longer-term maintenance considerations only to generate unanticipated future costs.

As a partial countermeasure, developers of cloud-based web services are frequently given additional tasks associated with implementing custom *ad hoc* governance solutions using either locally developed mechanisms or loosely integrated third-party API management services. These add-on governance approaches often fall short in terms of their consistency and enforcement capabilities since by definition they have to operate outside the cloud (either external to it or as a cloud-hosted application). As such, they do not have the end-to-end access to all the metadata and cloud-internal control mechanisms that are necessary to implement strong governance at scale.

III. Enforcing API Governance in Cloud Settings

API governance policy enforcement, in a cloud setting, is a tradeoff between application intrusiveness and policy expressibility. Detailed, fine-grained policies that penetrate the application provide maximal expressibility. In the logical extreme, building policy into the application to govern each of its internal operations or instructions is maximally expressive. However, this expressivity introduces complexity and performance overhead that may overshadow its benefit. Alternatively, policy enforcement outside the application necessarily limits what can be enforced. For example, ensuring that an application does not connect to a specific network address and/or port requires run-time traffic interception (typically by a firewall that is interposed between the application and the offending network). Thus, the tradeoff is between what can be enforced and when (relative to application execution) it is enforced.

For policy implementation, often the additional complexities introduced by late-binding and intrusiveness outweigh the

benefits. For example, in an application that consists of API calls to services that, in turn, make calls to other services, run-time policy enforcement can make violations difficult to resolve, especially when the interaction between services is non-deterministic. When a specific violation occurs, it may be "buried" in a lattice of API invocations that is difficult to traverse, especially of the application itself is designed to handle large-scale request traffic loads.

Ideally, then, enforcement takes place as non-intrusively as possible before the application begins executing. In this way, a violation can be detected and resolved *before the API is used*, thereby avoiding possible degradations in user-experience that run-time checks and violations may introduce. The drawback of attempting to enforce all governance before the application begin executing is that policies that express restrictions only resolvable at run time cannot be implemented. Thus, for scalable applications that use API calls internally in a cloud setting, an API governance approach should attempt to implement as much as possible no later than deployment time but must also include some form of run-time enforcement.

Note that the most effective approach to implementing a specific policy is not always clear. For example, user authentication is usually implemented as a run-time policy check for web services since users enter and leave the system dynamically. However it is possible to check statically, at deployment time, whether the application is consulting with a specific identity management service (accessed by a versioned API) thereby enabling deployment-time enforcement.

Thus, any efficient API governance solution for clouds must include the following functionalities.

- **Policy Specification Language** – The system must include a way to specify policies that can be implemented either at deployment time (or sooner) or, ultimately at run-time.
- **API Specification Language** – Policies must be able to refer to API functionalities to be able to express governance edicts for specific APIs or classes of APIs.
- **Deployment Control** – The system must be able to check policies no later than the time that an application is deployed.
- **Run-time control** – For policies that cannot be enforced before runtime, the system must be able to intervene dynamically.

In addition, a good solution should automate as much of the implementation of API governance as possible. Automation in a cloud context serves two purposes. First, it enables scale by allowing potentially complex optimizations to be implemented reliably by the system (and not by manual intervention). Secondly, automation improves repeatability and auditability thereby ensuring greater system integrity.

IV. EAGER

To experiment with implementation of API governance, we have developed EAGER – an architecture for implementing governance suitable for integration as a cloud-native feature. In this section, we overview the high-level design of EAGER, describe its main components and the policy language. Our goal in its design is two fold. First, we wished to verify that the integration between policy specification, api specification, deployment control, and run-time control is feasible in a cloud setting. Secondly, we wished to use the design as the basis for a prototype implementation that we could use to evaluate the impact of API governance empirically.

EAGER is designed to be integrated with cloud platforms that provide PaaS services. PaaS platforms accept code (uploaded via a well-defined API) that is then deployed within the platform so that it may make calls to existing services (each via a separate API) supported by the platform. EAGER is designed to intercept all events related to application deployment within the cloud and enforces deployment-time governance checks and logging. When a policy verification check fails, EAGER aborts the deployment of the application and logs the information necessary to perform remediation. EAGER assumes that it is integrated with the cloud and that the cloud initiates[1] in a compliant state (*i.e.* there are no policy violations when the cloud is started before any applications are deployed). It tries to maintain the cloud in a "governed" state at all times. That is, with EAGER active, the cloud is automatically prevented from transitioning out of policy compliance due to a change in the APIs it hosts.

Figure 1 illustrates the main components of EAGER (in blue) and their interactions. Solid arrows represent the interactions that take place during application deployment-time, before an application has been validated for deployment. Short-dashed arrows indicate the interactions that take place during deployment-time, after an application has been successfully validated. Long-dashed arrows indicate interactions at run-time. The diagram also outlines the components of EAGER that are used to provide deployment control and run-time control. Note that some components participate in interactions related to both deployment and run-time control (e.g. Metadata Manager).

EAGER must be invoked by the cloud whenever a user attempts to deploy an application in the cloud. The cloud's application deployment mechanisms must be altered so that each deployment request is intercepted by EAGER, which then performs the required governance checks. If a governance check fails, EAGER will preempt the application deployment, log relevant data pertaining to the event for later analysis, and return an error. Otherwise, it proceeds with the application deployment by activating the deployment mechanisms on the user's behalf.

Architecturally, the deployment action requires three inputs: the policy specification governing the deployment, the code to be deployed, and a specification of the APIs that the code exercises and exports. EAGER assumes that cloud administrators have developed and installed policies (stored in the Metadata Manager) that are to be checked against all deployments. API specifications for the application must also be available to the governance engine. Because the API specifications are to be derived from the code (and are, thus, under developer

[1] We use the term "initiates" to differentiate the first clean installation of the cloud, from a cloud restart. EAGER must be able to maintain compliance across restarts, but it assumes that when the cloud is installed and suitably tested, it is in a policy compliant state.

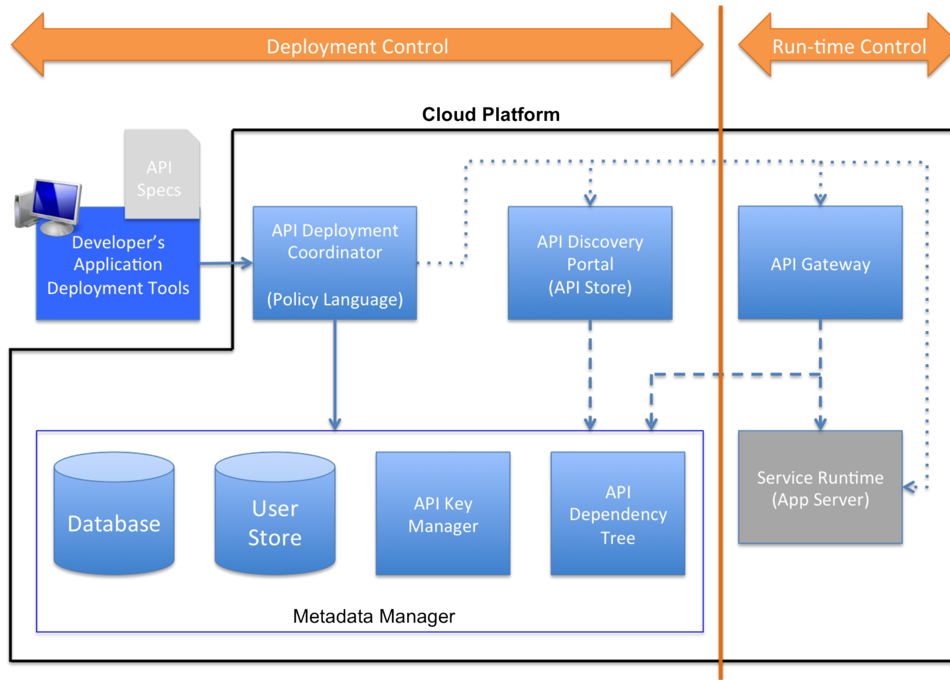


Fig. 1: EAGER Architecture

control and not administrator control) our design assumes that automated tools are available to perform analysis on the application, and generate API specifications in a suitable API specification language. These specifications must be present when the deployment request is considered by the platform. In the prototype implementation described in Section V, the API specifications are generated as part of the application development process (*e.g.* by the build system). They may also be offered as a trusted service hosted in the cloud. In this case, developers will submit their source code to this service, which will generate the necessary API specifications in the cloud and trigger the application deployment process via EAGER.

The proposed architecture is designed not to require major changes to the existing components of the cloud since its deployment mechanisms are likely to be web service based. However, EAGER does require integration at the service level (*e.g.* it must be a trusted service component in a PaaS cloud).

A. Metadata Manager

The Metadata Manager stores all the API metadata in EAGER. This metadata includes policy specifications, API names, versions, specifications and dependencies. It uses the dependency information to compute the dependency tree among all deployed APIs and applications. Additionally, the Metadata Manager also keeps track of developers, their subscriptions to various APIs and the access credentials (API keys) issued to them. For these purposes, the Metadata Manager must logically include both a database and an identity management system.

The Metadata Manager is exposed to other components through a well defined web service interface. This interface allows querying existing API metadata and updating them. In the proposed model, the stored metadata is updated occasionally (only when a new application is deployed or when a developer subscribes to a published API). Therefore the Metadata Manager does not need to support a very high write throughput. This performance characteristic allows the Metadata Manager to be implemented with strong transactional semantics, which reduces the development overhead of other components that rely on Metadata Manager. Availability can be improved via simple replication methods.

B. API Deployment Coordinator

The API Deployment Coordinator (ADC) intercepts all application deployment requests and determines whether they are suitable for deployment, based on a set of policies specified by the cloud administrators. It receives application deployment requests via a web service interface. At a high-level, ADC is the most important entity in the EAGER deployment control strategy.

An application deployment request contains the name of the application, version number, names and versions of the APIs exported by the application, detailed API specifications and other API dependencies as declared by the developer. Application developers only need to specify explicitly the name and version of the application and the list of dependencies (i.e. APIs consumed by the application). All other metadata can be computed automatically by performing introspection on the application source code.

The API specifications used to describe the web APIs should specify the operations and the schema of their inputs and

outputs. Any standard API description language can be used for this purpose, as long as it clearly describes the schema of the requests and responses. For describing REST interfaces, we can use Web Application Description Language (WADL) [18], Swagger [19], RESTful API Modeling Language (RAML) or any other language that provides similar functionality.

When a new deployment request is received, the ADC checks whether the application declares any API dependencies. If so, it queries the Metadata Manager to make sure that all the declared dependencies are already available in the cloud. Then it inspects the enclosed application metadata to see if the current application exports any web APIs. If the application exports at least one API, the ADC makes another call to the Metadata Manager and pulls any existing metadata related to that API. If the Metadata Manager cannot locate any data related to the API in question, ADC assumes it to be a brand new API (i.e. no previous version of that API has been deployed in the cloud), and proceeds to the next step of the governance check, which is policy validation. However, if any metadata regarding the API is found, then the ADC is dealing with an API update. In this case, the ADC compares the old API specifications with the latest ones provided in the application deployment request to see if they are compatible.

To perform this API compatibility verification, the ADC checks to see whether the latest specification of an API contains all the operations available in the old specification. API specifications are generated at the application developer's end and submitted to EAGER along with the application deployment request. If the latest API specification is missing at least one operation that it had previously, the ADC reports this to the user and aborts the deployment. If all the past operations are present in the latest specification, the ADC performs a type check to make sure that all past and present operations are type compatible. This is done by performing recursive introspection on the input and output types declared in the API specifications. EAGER looks for type compatibility based on the following rules inspired by Hoare logic [20] and the rules of type inheritance from object oriented programming:

- New version of an input type is compatible with the old version of an input type, if the new version contains either all or less attributes than the old version, and any new attributes that are unique to the new version are optional.
- New version of an output type is compatible with the old version of an output type, if the new version contains either all or more attributes than the old version.

In addition to the type checks, ADC may also compare other parameters declared in the API specifications such as HTTP methods, mime types and URL patterns. Once the API specifications have been successfully compared without error, the ADC initiates policy validation.

C. EAGER Policy Language and Examples

Policies are specified by cloud or organizational administrators using a subset of an object oriented language (we chose Python for the prototype). We restrict the language to prevent state from being preserved across policy validations. In particular, the EAGER policy interpreter disables file and network operations, third party library calls, and intrinsics that allow state to persist across invocations. In addition, EAGER processes each policy independently of others (i.e. each policy must be self-contained and access no external state). All other language constructs and language features can be used to specify policies in EAGER.

To accommodate built-in APIs that the administrators trust by *fiat*, all module and function restrictions of the EAGER policy language are enforced through a configurable white-list. The policy engine evaluates each module and function reference against this white-list to determine whether they are allowed in the context of EAGER. Cloud administrators have the freedom and flexibility to expand the set of allowed built-in and third party modules by making changes to this white-list.

As part of policy language, EAGER defines a set of assertions that policy writers can use to specify various checks to perform on the applications. Currently, this assertion list includes:

```
assert_true(condition, optional_error_msg)
assert_false(condition, optional_error_msg)
assert_app_dependency(app, d_name, d_version)
assert_not_app_dependency(app, d_name, d_version)
assert_app_dependency_in_range(app, name,\
  lower, upper, exclude_lower, exclude_upper)
```

In addition to these assertions, EAGER adds a function called "compare_versions" to the list of available built-in functions. Policy writers can use this function to compare version number strings associated with applications and APIs.

In the remainder of this section we illustrate the use of the policy language through examples. The first example policy mandates that any application or mash-up that uses both Geo and Direction APIs must adhere to certain versioning rules. More specifically, if the application uses Geo 3.0 or higher, it must use Direction 4.0 or higher. Note that the version numbers are compared using the "compare_versions" functions described earlier.

```
g = filter(lambda dep: dep.name == 'Geo', \
        app.dependencies)
d = filter(lambda dep: dep.name == 'Direction',\
        app.dependencies)
if g and d:
  g_api, d_api = g[0], d[0]
  if compare_versions(g_api.version, '3.0') >= 0:
    assert_true(compare_versions(d_api.version,
        '4.0') >= 0)
```

In the above example, *app* is a special immutable logical variable available to all policy files. This variable allows policies to access information pertaining to the current application deployment request. The assert_true and assert_false functions allow testing for arbitrary conditions, thus greatly improving the expressive power and flexibility of the policy language.

The next example shows a policy file that mandates that all applications deployed by the "admin@test.com" user must have role-based authentication enabled, so that only users in

the “manager” role can access them. To carry out this check the policy accesses the security configuration specified in the application descriptor (e.g. the web.xml for a Java application).

```
if app.owner == ‘admin@test.com’:
  roles = app.web_xml[‘security-role’]
  constraints = app.web_xml[‘security-constraint’]
  assert_true(roles and constraints)
  assert_true(len(roles) == 1)
  assert_true(‘manager’ == roles[0][‘role-name’])
```

Next, we present an example policy, which mandates that all deployed APIs must explicitly declare an operation which is accessible through the HTTP OPTIONS method. This policy further ensures that these operations return a description of the API in the Swagger [19] machine-readable API description language.

```
options = filter(lambda op : op.method == ‘OPTIONS’,
  api.operations)
assert_true(options, ‘API does not support OPTIONS’)
assert_true(options[0].type == ‘swagger.API’,
  ‘Does not return a Swagger description’)
```

Returning machine-readable API descriptions from web APIs makes it easier to automate the API discovery and consumption processes. Several other research efforts confirm the need for such descriptions [21], [22]. A policy such as this can help enforce such practices, thus resulting in a high-quality API ecosystem in the target cloud.

The policy above also shows the use of the second and optional string argument to the assert_true function (the same is supported by assert_false as well). This argument can be used to specify a custom error message that will be returned to the application developer, if his/her application violates the assertion in question.

The next example is a relatively simple policy file that prevents developers from introducing dependencies on deprecated web APIs. Deprecated APIs are the those that have been flagged by their respective authors for removal in the near future. Therefore introducing dependencies on such APIs is not recommended. The following policy will enforce this condition in the cloud.

```
deprecated = filter(
  lambda dep : dep.status == 'DEPRECATED',
  app.dependencies)
assert_false(deprecated,
  'Must not use a deprecated dependency')
```

Our next example presents a policy that enforces governance rules in a user-aware (i.e. tenant-aware) manner. Assume a multi-tenant private PaaS cloud that is being used by members of the development team and the sales team of a company. The primary goal in this case is to ensure that applications deployed by both teams log their activities using a set of preexisting logging APIs. However, we further want to ensure that applications deployed by the sales team log their activities using a special analytics API. A policy such as the one that follows can enforce these conditions.

```
if app.owner.endswith(‘@engineering.test.com’):
  assert_app_dependency(app, ‘Log’, ‘1.0’)
elif app.owner.endswith(‘@sales.test.com’):
  assert_app_dependency(app, ‘AnalyticsLog’, ‘1.0’)
else:
  assert_app_dependency(app, ‘GenericLog’, ‘1.0’)
```

The example below shows a policy that mandates that all HTTP GET operations exposed by APIs must support paging. APIs that do so define two input parameters named “start” and “count” to the GET call.

```
for api in app.api_list:
  get = filter(lambda op : op.method == ‘GET’,
    api.operations)
  for op in get:
    param_names = map(lambda p : p.name,
      op.parameters)
    assert_true(‘start’ in param_names and
      ‘count’ in param_names)
```

This policy accesses the metadata of API operations that is available in the API descriptions. Since API descriptions can be auto-generated from the source code of the APIs, this policy indirectly references information pertaining to the actual API implementations.

Finally, we present an example for the POST method. The policy below mandates that all POST operations exposed by an API are secured with OAuth version 2.0.

```
for api in app.api_list:
  post = filter(lambda op : op.method == ‘POST’,
        api.operations)
  for op in post:
    assert_true(op.authorizations.get(‘oauth2’))
```

EAGER places no restrictions on how many policy files are specified by administrators. Applications are validated against each policy file. Failure of any assertion in any policy file will cause the ADC to abort application deployment. Once an application has been checked against all applicable policies, ADC persists the latest application and API metadata into the Metadata Manager. At this point, the ADC may report success to the user and proceed with application deployment. In a PaaS setting this deployment activity typically involves three steps:

1) Deploy the application in the cloud application run-time (application server).
2) Publish the APIs enclosed in the application and their specifications to the API Discovery Portal or catalog.
3) Publish the APIs enclosed in the application to an API Gateway server.

Step 1 is required to complete the application deployment in the cloud even without EAGER. We explain the significance of steps 2 and 3 in the following subsections.

D. API Discovery Portal

The API Discovery Portal (ADP) is an online catalog where developers can browse available web APIs. Whenever the ADC

approves and deploys a new application, it registers all the APIs exported by the application in ADP. EAGER mandates that any developer interested in using an API, first subscribe to that API and obtain the proper credentials (API keys) from the ADP. The API keys issued by the ADP can consist of an OAuthaccess token (as is typical of many commercial REST-based web services) or a similar authorization credential, which can be used to identify the developer/application that is invoking the API. This credential management identification process is used for auditing and run-time governance in EAGER.

The API keys issued by the ADP are stored in the Metadata Manager. When a programmer develops a new application using one or more API dependencies, we can require the developer to declare its dependencies along with the API keys obtained from the ADP. The ADC verifies this information against the Metadata Manager as a part of its dependency check and ensures that the declared dependencies are correct and the specified API keys are valid.

Deployment-time governance policies may further incentivize the declaration of API dependencies explicitly by making it impossible to call an API without first declaring it as a dependency along with the proper API keys. These types of policies can be implemented with minor changes to the application run-time in the cloud so that it loads the API credentials from the dependency declaration provided by the application developer.

In addition to API discovery, the ADP also provides a user interface for API authors to select their own APIs and deprecate them or retire them. Deprecated APIs will be removed from the API search results of the portal, and application developers will no longer be able to subscribe to them. However, already existing subscriptions and API keys will continue to work until the API is eventually retired. The deprecation is considered a courtesy notice for application developers who have developed applications using the API, to migrate their code to a newer and active version if the API. Once retired, any applications that have not still been migrated to the latest version of the API will cease to operate.

E. API Gateway

Run-time governance of web services by systems such as Synapse [23] make use of an API "proxy" or gateway. The EAGER API Gateway does so to intercept API calls and validate the API keys contained within them. EAGER intercepts requests by blocking direct access to the APIs in the application run-time (app servers), and publishing the API Gateway address as the API endpoint in the ADP. We do so via firewall rules or router configuration that prevents the cloud app servers from receiving any API traffic from a source other than the API Gateway. Once the API Gateway validates an API call, it routes the message to the application server in the cloud platform that hosts the API.

The API Gateway can be implemented via one or more (load-balanced) servers. In addition to API key validation, the API Gateway performs other functions such as monitoring, throttling (rate limiting), SLA enforcement, and run-time policy validation.

V. Prototype Implementation

We implemented a prototype of EAGER by extending AppScale [11], an open source PaaS cloud that is functionally equivalent to Google App Engine (GAE). AppScale supports web applications written in Python, Java, Go and PHP. Our prototype implements governance for all applications and APIs hosted in an AppScale cloud.

As described in subsection IV-C, our prototype policy specification language is based on Python. Using Python allows EAGER to leverage existing programming tools to edit and debug policy files. It also allows the deployment control module (also written in Python) to execute the policies (using a modified Python interpreter to implement the restrictions previously discussed) directly.

The prototype relies on a separate tool chain (*i.e.* one not hosted as a service in the cloud) to automatically generate API specifications and other metadata (*c.f.* Section IV-B), which currently supports only the Java language. Developers must document the APIs manually for web services implemented in languages other than Java.

Like most PaaS technologies, AppScale includes an application deployment service that distributes, launches and exports an application as a web-accessible service. EAGER controls this deployment process according to the policies that the platform administrator specifies.

A. Autogeneration of API Specifications

To autogenerate API specifications, the build process for an application must include an analysis phase that generates specifications from the code. The prototype includes two stand-alone tools for implementing this "build-and-analyze" function.

1) An Apache Maven archetype that is used to initialize a Java web application project, and
2) A Java doclet that is used to auto-generate API specifications from web APIs implemented in Java

Developers can invoke the Maven archetype from the command-line to initialize a new Java web application project. Our archetype sets up projects with the required AppScale (GAE) libraries, Java JAX-RS [24] (Java API for RESTful Web Services) libraries and a build configuration.

Once a developer creates a new project using the archetype s/he can develop web APIs using the popular JAX-RS library. Once code is developed, it can be built using our auto-generated Maven build configuration, which introspects the project source code to generate specifications for all enclosed web APIs using the Swagger [25] API description language. It then packages the compiled code, required libraries, generated API specifications, and the dependency declaration file into a single, deployable artifact.

Finally, the developer submits the generated artifact for deployment to the cloud platform (which in our prototype is done via AppScale developer tools). To enable this, we modified the tools so that they send the application deployment request to the EAGER ADC and delegate the application

EAGER Component	Implementation Technology
Metadata Manager	MySQL
API Deployment Coordinator	Native Python implementation
API Discovery Portal	WSO2 API Manager [26]
API Gateway	WSO2 API Manager

TABLE I: Implementation Technologies used to Implement the EAGER Prototype

deployment process to EAGER. This change required just under 50 additional lines of code in AppScale.

B. Implementing the Prototype

Table I lists the key technologies that we use to implement various EAGER functionalities described in Section IV as services within AppScale itself. For example, AppScale controls the lifecycle of the MySQL database as it would any of its other constituent services. EAGER incorporates the WSO2 API Manager [27] for use as an API discovery mechanism and to implement any run-time policy enforcement. In the prototype, the API Gateway does not share policies expressed in the policy language with the ADC (although this integration is planned).

Also, according to the architecture of EAGER, Metadata Manager is the most suitable location for storing all policy files. The ADC may retrieve the policies from the Metadata Manager through its web service interface. However, for simplicity, our current prototype stores the policy files in a file system, that the ADC can directly read from. In a more sophisticated future implementation of EAGER, we will move all policy files to the Metadata Manager where they can be better managed, while providing easy access to other distributed components of the cloud.

VI. Experimental Results

In this section, we describe our empirical evaluation of the EAGER prototype and evaluate its overhead and scaling characteristics. To do so, we populate the EAGER database with a set of APIs and then examine the overheads associated with governing a set of sample applications (which execute in AppScale) for varying degrees of policy specifications and dependencies (shown in Table II). In the first set of results we use randomly generated APIs so that we may vary different parameters that may affect performance. We then follow with a similar analysis using a large set of API specifications “scraped” from the ProgrammableWeb [1] public API registry.

Note that all the figures included in this section present the average values calculated over three sample runs. The error bars cover an interval of two standard deviations centered at the calculated sample average.

We start by presenting the time required for AppScale application deployment without EAGER as it is this process on which we piggyback EAGER support. These measurements are conservative in that they are taken from a single VM deployment of AppScale (it is designed to run at scale) so that logging and timing information is easier to gather. AppScale uses an Ubuntu 12.04 Linux image hosted via VirtualBox on a 2.7 GHz x86 CPU with 4 GB of memory. Table II lists a number of App Engine applications that we consider, their artifact size, and their average deployment times across three runs, on AppScale without EAGER. We also identify the number of APIs and dependencies for each application in the `Description` column. These applications represent a wide range of programming languages, application sizes, and business domains.

Application	Description	Size (MB)	Deployment Time (s)
guestbook-py	A simple Python web application that allows users to post comments and view them	0.16	22.13
guestbook-java	A Java clone of the guestbook-python app	52	24.18
appinventor	A popular open source web application that enables creating mobile apps	198	111.47
coursebuilder	A popular open source web application used to facilitate teaching online courses	37	23.75
hawkeye	A sample Java application used to test AppScale	35	23.37
simple-jaxrs-app	A sample JAXRS app that exports 2 web APIs	34	23.45
dep-jaxrs-app	A sample JAXRS app that exports a web API and has one dependency	34	23.72
dep-jaxrs-app-v2	A sample JAXRS app that exports 2 web APIs and has one dependency	34	23.95

TABLE II: Sample AppScale Applications

On average, deployment without EAGER takes 34.5 seconds and this time is correlated with application artifact size. The total time consists of network transfer time of the application to the cloud (which in this case is via localhost networking) and disk copy to the application servers. For actual deployments, both components are likely to increase due to network latency, available bandwidth, contention, and large numbers of distributed application servers.

A. Baseline EAGER Overhead by Application

Figure 2 shows the average time in seconds taken by EAGER to validate and verify each application. We record these results on an EAGER deployment without any policies deployed, and without any prior metadata recorded in the API Metadata Manager (that is, an unpopulated database of policies). We present the values as absolute measurements (here and henceforth) because of the significant difference between them and deployment times on AppScale without EAGER (100’s of milliseconds compared to 10’s of seconds). We can alternatively observe this overhead as a percentage of AppScale deployment time by dividing these times by those shown in Table II.

Note that some applications do not export any web APIs; for these EAGER overhead is negligibly small (approximately 0.1s). This result indicates that EAGER does not impact deployment time of applications that do not require API governance. For applications that do export web APIs, the

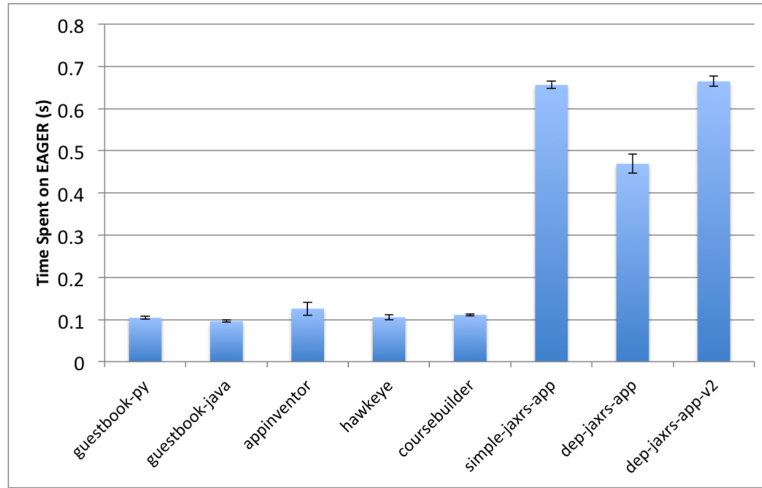


Fig. 2: Average EAGER absolute overhead in seconds by application. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds.

recorded overhead measurements include the time to retrieve old API specifications from the Metadata Manager, the time to compare the new API specifications with the old ones, the time to update the API specifications and other metadata in the Metadata Manager, and the time to publish the updated APIs to the cloud. The worst case observed overhead for governed APIs (simple-jaxrs-app in the Figure 2) is 2.8%.

B. Impact of Number of APIs and Dependencies

Figure 3 shows that EAGER overhead grows linearly with the number of APIs exported by an application. This scaling occurs because the current prototype implementation iterates through the APIs in the application sequentially and records the API metadata in the Metadata Manager. Then EAGER publishes each API to the ADP and API Gateway. This sequencing individual EAGER events, each of which generates a separate web service call, represents an optimization opportunity via parallelization in future implementations.

At present we expect most applications deployed in cloud to have a small to moderate number of APIs (10 or fewer). With this API density EAGER's current scaling is adequate. Even in the unlikely case that a single application exports as many as 100 APIs, the average total time for EAGER is under 20 seconds.

Next, we analyze EAGER overhead as the number of dependencies declared in an application grows. For this experiment, we first populate the EAGER Metadata Manager with metadata for 100 randomly generated APIs. [2] Then we deploy an application on EAGER which exports a single API and declares artificial dependencies on the set of fictitious APIs that are already stored in the Metadata Manager. We vary the number of declared dependencies and observe the EAGER overhead.

[2]To generate random APIs we use the API specification autogeneration tool chain to generate fictitious APIs with randomly varying numbers of parameters.

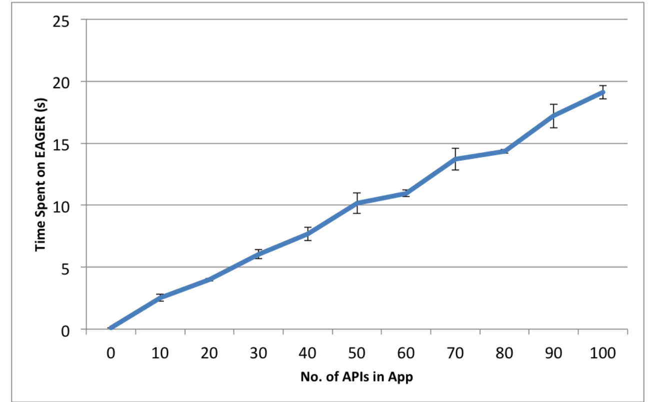


Fig. 3: Average EAGER overhead vs. number of APIs exported by the application. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds.

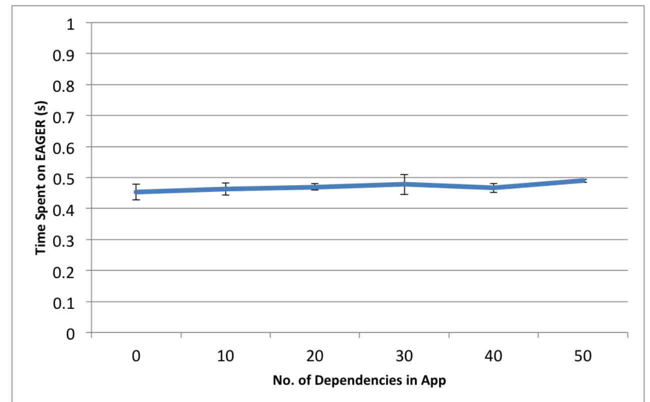


Fig. 4: Average EAGER Overhead vs. number of dependencies declared in the application. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds.

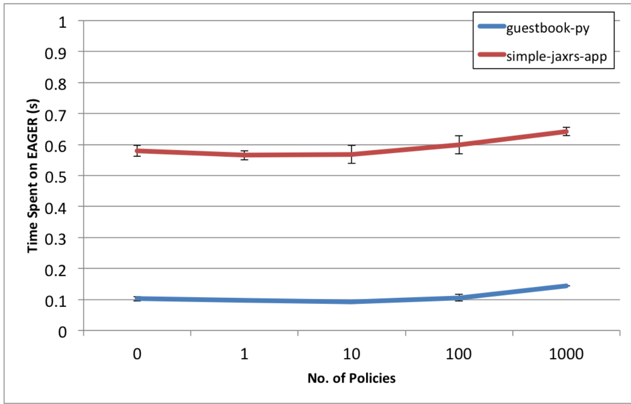


Fig. 5: Average EAGER overhead vs. number of policies. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds. Note that some of the error bars for guestbook-py are smaller than the graph features at this scale and are thus obscured.

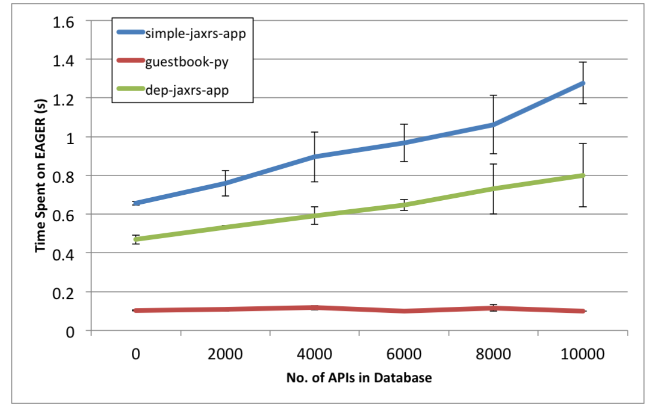


Fig. 6: Average EAGER overhead vs. number of APIs in Metadata Manager. Each data point averages three executions, the error bars are two standard deviations, and the units are seconds. Note that some of the error bars for guestbook-py are smaller than the graph features at this scale and are thus obscured.

Figure 4 shows the results of these experiments. EAGER overhead does not appear to be significantly influenced by the number of dependencies declared in an application. In this case, the EAGER implementation processes all dependency-related information via batch operations. As a result, the number of web service calls and database queries that originate due to varying number of dependencies is fairly constant.

C. Impact of Number of Policies

So far we have conducted all our experiments without any active governance policies in the system. In this section, we report how EAGER overhead is impacted by the number of policies.

The overhead of policy validation is largely dependent on the actual policy content (implemented as Python code). Since users may include any Python code (as long as it falls in the accepted subset) in a policy file, evaluating a given policy can take an arbitrary amount of time. Therefore, in this experiment, our goal is to evaluate the overhead incurred by simply having many policy files to execute. We keep the content of the policies small and trivial. We create a policy file that runs following assertions:

1) Application name must start with an upper case letter
2) Application must be owned by a specific user
3) All API names must start with upper case letters

We create many copies of this initial policy file to vary the number of policies deployed. Then we evaluate the overhead of policy validation on two of our sample applications (guestbook-py and simple-jaxrs-app).

Figure 5 shows how the number of active policies impact EAGER overhead. Interestingly, even large numbers of policies do not impact EAGER overhead significantly. It is only when the active policy count approaches 1000 that we can see a small increase in the overhead. Even then, the increase in deployment time is under 0.1 seconds.

This result due to the fact that EAGER loads policy content into memory at system startup (or when a new policy is deployed), and executes them from memory each time an application is deployed. Since policy files are typically small (at most a few kilobytes), this is a viable option. The overhead of validating the simple-jaxrs-app is higher than that of the guestbook-py because, simple-jaxrs-app exports web APIs. This means the third assertion in the policy set is executed for this app and not for guestbook-py.

Our results indicate that EAGER scales well to hundreds of policies. That is, there is no significant overhead associated with simply having a large number of policy files. However, as mentioned earlier, the content of a policy may influence the overhead of policy validation and will be specific to the policy and application EAGER analyzes.

D. Scalability

Next, we evaluate how EAGER scales when a large number of APIs are deployed in the cloud. In this experiment, we populate the EAGER Metadata Manager with a varying number of random APIs. We then attempt to deploy various sample applications. We also create random dependencies among the APIs recorded in the Metadata Manager to make the experimental setting more realistic.

Figure 6 shows that the deployment overhead of the guestbook-py application is not impacted by the growth of metadata in the cloud. Recall that guestbook-py does not export any APIs nor does it declare any dependencies. Therefore the deployment process of the guestbook-py application has minimal interactions with the Metadata Manager. Based on this result we conclude that applications that do not export web APIs are not significantly affected by the accumulation of metadata in EAGER.

Both simple-jaxrs-app and dep-jaxrs-app are affected by the volume of data stored in Metadata Manager. Since these applications export web APIs that must be recorded and

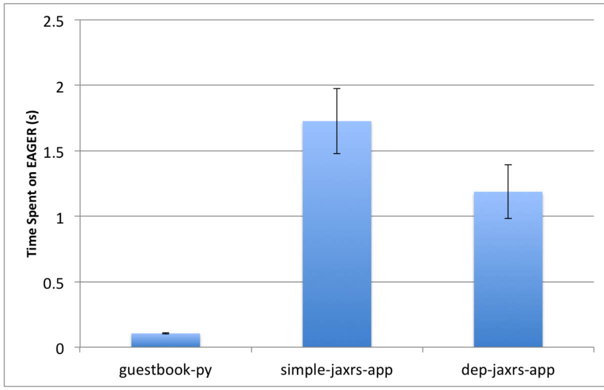


Fig. 7: Average EAGER overhead over three experiments when deploying on ProgrammableWeb Dataset. The the error bars are two standard deviations and the units are seconds.

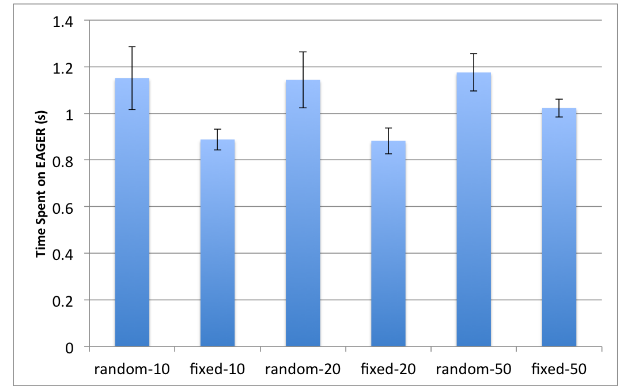


Fig. 8: EAGER Overhead When Deploying on ProgrammableWeb Dataset with Dependencies. The suffix value indicates the number of dependencies; the prefix indicates if these dependencies are randomized (or not) upon redeployment. Each data point averages three executions, the error bars that are two standard deviations, and the units are seconds.

validated by the Metadata Manager, the growth of metadata has an increasingly higher impact on them. The degradation of performance as a function of the number of APIs in the Metadata Manager database is due to the slowing of query performance of the RDBMS engine (MySQL) as the database size grows. Note that the simple-jaxrs-app is affected more by this performance drop, because it exports two APIs compared to the single API exported by dep-jaxrs-app. However, the growth in overhead is linear to the number of APIs deployed in the cloud (presumably indicating linear scaling factor in the installation of MySQL that EAGER used in these experiments). Also, even after deploying 10000 APIs, the overhead on simple-jaxrs-app is only been increased by about 0.5 seconds.

Another interesting characteristic in Figure 6 is the increase in overhead variance as the number of APIs in the cloud grows. We believe this is due to the increasing variability of database query performance and the data transfer performance as the size of the database increases.

In summary, the current EAGER prototype scales well to 1000's of APIs. If further scalability is required, we can employ parallelization and database query optimization.

E. Experimental Results with a Real-World Dataset

Finally, we explore how EAGER operates with a real-world dataset with API metadata and dependency information. For this, we crawl the ProgrammableWeb registry and extract metadata regarding all registered APIs and mash-ups. At the time of the experiment, we collected 11095 APIs and 7227 mash-ups (each mash-up depends on one or more APIs).

We autogenerated API specifications for each and then populated the EAGER Metadata Manager with these specifications. We then used the mashup-API dependency information detected by EAGER to register dependencies among the APIs in EAGER. This resulted in a total dependency graph of 18322 APIs with 33615 dependencies. We then deploy a subset of our applications and measure EAGER overhead.

Figure 7 shows the results for three applications. The guestbook-py app (without any web APIs) is not significantly impacted by the large dependency database. Applications that export web APIs show a slightly higher deployment overhead due to the database scaling properties previously discussed. However, the highest overhead observed is under 2 seconds for simple-jaxrs-app, which is an acceptably small percentage of the 23.45 second deployment time (as shown in Table II).

The applications in this experiment do not declare dependencies on any of the APIs in the ProgrammableWeb dataset. The dep-jaxrs-app does declare a dependency, but that is on an API exported by simple-jaxrs-app. To see how the deployment time is impacted when applications become dependent on other APIs registered in EAGER, we deploy a test application that declares random fictitious dependencies on APIs from the ProgrammableWeb corpus registered in EAGER. We consider 10, 20, and 50 declared dependencies and deploy each application three times. We present the results in Figure 8. For the "random" datasets, we run a deployment script that randomly modifies the declared dependencies at each redeployment. For the "fixed" datasets the declared dependencies remains the same across redeployments.

Interestingly, dependency count does not have a significant impact on the overhead. The largest overhead observed is under 1.2 seconds for 50 randomly varied dependencies. In addition, when the dependency declaration is fixed, the overhead is slightly smaller. This is because our prototype caches the edges of its internally generated dependency tree, which expedites redeployments.

In summary, EAGER adds a very small overhead to the application deployment process, and this overhead increases linearly with the number of APIs exported by the applications and the number of APIs deployed in the cloud. Interestingly, the number of deployed policies and declared dependencies have little impact on the EAGER governance overhead. Finally, our results indicate that EAGER scales well to 1000's of APIs and adds no more than 2 seconds with over $18,000$ "real-world" deployed APIs in its database.

VII. Related Work

Our research builds upon advances in the areas of SOA governance and service management. Guan et al introduced FASWSM [28] a web service management framework for application servers. FASWSM uses an adaptation technique that wraps web services in a way so they can be managed by the underlying application server platform. Wu et al introduced DART-Man [29] a web service management system based on semantic web concepts. Zhu and Wang proposed a model that uses Hadoop and HBase to store web service metadata and process them to implement a variety of management functions [30]. Our work is different from these past approaches in that EAGER targets policy *enforcement* and we focus on doing so by extending extant cloud platforms (e.g. PaaS) to provide an integrated and scalable governance solution.

Lin et al proposed a service management system for clouds that monitors all service interactions via special "hooks" that are connected to the cloud-hosted services [31]. These hooks monitor and record service invocations, and also provide an interface so that the individual service artifacts can be managed remotely. However, this system only supports run-time service management and provides no support for deployment-time policy checking and enforcement. Kikuchi and Aoki [32] proposed a technique based on model checking to evaluate the operational vulnerabilities and fault propagation patterns in cloud services. However, this system provides no active monitoring or enforcement functionality. Sun et al proposed a reference architecture for monitoring and managing cloud services [33]. This too lacks deployment-time governance, policy validation support, and the ability to intercept and act upon API calls which limits its use as a comprehensive governance solution for clouds.

Other researchers have shown that policies can be used to perform a wide range of governance tasks for SOA such as access control [34], [35], fault diagnosis [36], customization [37], composition [38], [39] and management [40], [41], [42]. We build upon the foundation of these past efforts and use policies to govern RESTful web APIs deployed in cloud settings. Our work is also different in that it defines an executable policy language (implemented as a subset of Python in the EAGER prototype) which is capable of capturing a wide range of governance requirements.

Peng, Lui and Chen showed that the major concerns associated with SOA governance involve retaining the high reliability of services, recording how many services are available on the platform to serve, and making sure all the available services are operating within an acceptable service level [43]. EAGER attempts to satisfy similar requirements for modern RESTful web APIs deployed in cloud environments. However, EAGER's Metadata Manager and ADP record and keep track of all deployed APIs in a simple, extensible, and comprehensive manner. Moreover, EAGER's policy validation, dependency management, and API change management features "fail fast" to detect violations immediately making diagnosis and remediation less complex, and prevent the system from ever entering a non-compliant state.

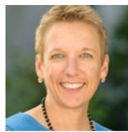
API management has been a popular topic in the industry over the last few years, resulting in many commercial and open source API management solutions [27], [7], [8], [44]. These products facilitate API lifecycle management, traffic shaping, access control, monitoring and a variety of other important API-related functionality. However, these tools do not support deep integration with cloud environments in which many web applications and APIs are deployed today. EAGER is also different in that it combined deployment time and run-time enforcement. Previous systems either work exclusively at run-time or do not include an enforcement capability (i.e. they are advisory).

VIII. Conclusions and Future Work

In this paper, we describe EAGER, a model and a software architecture that facilitates API governance as a cloud-native feature. EAGER supports comprehensive policy enforcement, dependency management, and a variety of other deployment-time API governance features. It promotes many software development and maintenance best practices including versioning, code reuse, and retaining API backwards compatibility. EAGER also includes a language based on Python that enables creating, debugging, and maintaining API policies in a simple and intuitive manner. EAGER can be integrated into cloud platforms that are used to host APIs to automate governance tasks that otherwise require custom code or developer intervention.

Our empirical results (gathered using a prototype of EAGER developed for AppScale) show that EAGER adds negligibly small overhead to the cloud application deployment process, and the overhead grows linearly with the number of APIs deployed. We also show that EAGER scales well to handle tens of thousands of APIs and hundreds of policies.

Hiranya Jayathilaka is a PhD student in the Computer Science Department at UC Santa Barbara (UCSB). His research interests include distributed systems and web/cloud services. Prior to joining UCSB, Hiranya was a Senior Technical Lead at WSO2 Inc. and received a BS degree in Engineering from the Univ. of Moratuwa, Sri Lanka.



Dr. Chandra Krintz is a Professor in the Computer Science Department at UCSB. Her research interests include cloud platforms and programming systems, and she is the progenitor of the open source cloud platform-as-a-service, AppScale. She holds MS and PhD degrees from UC San Diego.

Dr. Rich Wolski is a Professor in the Computer Science Department at UCSB. His research interests include cloud infrastructures and scientific computing, and he is the progenitor of the open source cloud infrastructure-as-a-service, Eucalyptus. He holds MS and PhD degrees from UC Davis.

This work is funded in part by NSF (CNS-0905237 and CNS-1218808) and NIH (1R01EB014877-01).

REFERENCES

[1] "ProgrammableWeb – http://www.programmableweb.com."

[2] "ProgrammableWeb Blog – http://blog.programmableweb.com/2013/04/30/9000-apis-mobile-gets-serious/."

[3] "IEEE Xplore Search Gateway – http://ieeexplore.ieee.org/gateway/."

[4] "Berkeley API Central – https://developer.berkeley.edu."

[5] "Agency Application Programming Interfaces – http://www.whitehouse.gov/digitalgov/apis."

[6] "Free and Enterprise API Management Platform and Infrastructure by 3scale – http://www.3scale.net."

[7] "Enterprise API Management and API Strategy – http://apigee.com/about/."

[8] "Enterprise API Management - Layer 7 Technologies – http://www.layer7tech.com."

[9] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and U. Yalcinalp, "Web services policy framework (wspolicy)," September 2007. [Online]. Available: http://www.w3.org/TR/ws-policy

[10] "SOA Governance Technical Standard – http://www.opengroup.org/soa/source-book/gov/intro.htm."

[11] C. Krintz, "The AppScale Cloud Platform: Enabling Portable, Scalable Web Application Deployment," *IEEE Internet Computing*, vol. Mar/Apr, 2013.

[12] F. Belqasmi, R. Glitho, and C. Fu, "Restful web services for service provisioning in next-generation networks: a survey," *Communications Magazine, IEEE*, vol. 49, no. 12, pp. 66–73, December 2011.

[13] M. N. Haines and M. A. Rothenberger, "How a service-oriented architecture may change the software development process," *Commun. ACM*, vol. 53, no. 8, pp. 135–140, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1787234.1787269

[14] G. Lawton, "Developing software online with platform-as-a-service technology," *Computer*, vol. 41, no. 6, pp. 13–15, June 2008.

[15] "Platform as a Service - Pivotal CF," http://www.gopivotal.com/platform-as-a-service/pivotal-cf.

[16] "OpenShift by RedHat," https://www.openshift.com.

[17] H. Jayathilaka, C. Krintz, and R. Wolski, "Towards automatically estimating porting effort between web service apis," in *To Appear: Services Computing, 2014. SCC 2014. IEEE International Conference on*. IEEE, 2014.

[18] "Web Application Description Language," http://www.w3.org/Submission/wadl/, 2013, [Online; accessed 27-September-2013].

[19] "Swagger: A simple, open standard for describing REST APIs with JSON," https://developers.helloreverb.com/swagger/, [Online; accessed 05-August-2013].

[20] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969. [Online]. Available: http://doi.acm.org/10.1145/363235.363259

[21] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle, "Functional descriptions as the bridge between hypermedia APIs and the Semantic Web," in *International Workshop on RESTful Design*, 2012.

[22] T. Steiner and J. Algermissen, "Fulfilling the hypermedia constraint via http options, the http vocabulary in rdf, and link headers," in *Proceedings of the Second International Workshop on RESTful Design*, ser. WS-REST '11. New York, NY, USA: ACM, 2011, pp. 11–14. [Online]. Available: http://doi.acm.org/10.1145/1967428.1967433

[23] "Apache Synapse," https://synapse.apache.org/, [Online; accessed 25-March-2014].

[24] "JSR311 - The Java API for RESTful Web Services – https://jcp.org/aboutJava/communityprocess/final/jsr311/."

[25] "Swagger - A simple, open standard for describing REST APIs with JSON – https://helloreverb.com/developers/swagger."

[26] "WSO2 API Manager," http://wso2.com/products/api-manager/, 2013, [Online; accessed 27-September-2013].

[27] "WSO2 API Manager – http://wso2.com/products/api-manager/."

[28] H. Guan, B. Jin, J. Wei, W. Xu, and N. Chen, "A framework for application server based web services management," in *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, Dec 2005, pp. 8 pp.–.

[29] J. Wu and Z. Wu, "Dart-man: a management platform for web services based on semantic web technologies," in *Computer Supported Cooperative Work in Design, 2005. Proceedings of the Ninth International Conference on*, vol. 2, May 2005, pp. 1199–1204 Vol. 2.

[30] X. Zhu and B. Wang, "Web service management based on hadoop," in *Service Systems and Service Management (ICSSSM), 2011 8th International Conference on*, June 2011, pp. 1–6.

[31] C.-F. Lin, R.-S. Wu, S.-M. Yuan, and C.-T. Tsai, "A web services status monitoring technology for distributed system management in the cloud," in *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2010 International Conference on*, Oct 2010, pp. 502–505.

[32] S. Kikuchi and T. Aoki, "Evaluation of operational vulnerability in cloud service management using model checking," in *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, March 2013, pp. 37–48.

[33] Y. Sun, Z. Xiao, D. Bao, and J. Zhao, "An architecture model of management and monitoring on cloud services resources," in *Advanced Computer Theory and Engineering (ICACTE)*, vol. 3, Aug 2010, pp. V3–207–V3–211.

[34] R. Bhatti, D. Sanz, E. Bertino, and A. Ghafoor, "A policy-based authorization framework for web services: Integrating xgtrbac and ws-policy," in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, July 2007, pp. 447–454.

[35] S.-C. Chou and J.-Y. Jhu, "Access control policy embedded composition algorithm for web services," in *Advanced Information Management and Service (IMS), 2010 6th International Conference on*, Nov 2010, pp. 54–59.

[36] L. Li, K. Xiaohui, L. Yuanling, X. Fei, Z. Tao, and C. YiMin, "Policy-based fault diagnosis technology for web service," in *Instrumentation, Measurement, Computer, Communication and Control, 2011 First International Conference on*, Oct 2011, pp. 827–831.

[37] H. Liang, W. Sun, X. Zhang, and Z. Jiang, "A policy framework for collaborative web service customization," in *Service-Oriented System Engineering, 2006. SOSE '06. Second IEEE International Workshop*, Oct 2006, pp. 197–204.

[38] A. Erradi, P. Maheshwari, and S. Padmanabhuni, "Towards a policy-driven framework for adaptive web services composition," in *Next Generation Web Services Practices, 2005. NWeSP 2005. International Conference on*, Aug 2005, pp. 6 pp.–.

[39] A. Erradi, P. Maheshwari, and V. Tosic, "Policy-driven middleware for self-adaptation of web services compositions," in *International Conference on Middleware*, 2006.

[40] B. Suleiman and V. Tosic, "Integration of uml modeling and policy-driven management of web service systems," in *ICSE Workshop on Principles of Engineering Service Oriented Systems*, 2009.

[41] M. Thirumaran, D. Ponnurangam, K. Rajakumari, and G. Nandhini, "Evaluation model for web service change management based on business policy enforcement," in *Cloud and Services Computing (ISCOS), 2012 International Symposium on*, Dec 2012, pp. 63–69.

[42] F. Zhang, J. Gao, and B.-S. Liao, "Policy-driven model for autonomic management of web services using mas," in *Machine Learning and Cybernetics, 2006 International Conference on*, Aug 2006, pp. 34–39.

[43] K.-Y. Peng, S.-C. Lui, and M.-T. Chen, "A study of design and implementation on soa governance: A service oriented monitoring and alarming perspective," in *Service-Oriented System Engineering, 2008. SOSE '08. IEEE International Symposium on*, Dec 2008, pp. 215–220.

[44] "Mashery – http://www.mashery.com."