EAGER: Deployment-time API Governance for Modern PaaS Clouds

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

***Abstract*—To track, control, and compel reuse of web APIs, we investigate a new approach to API governance – combined policy, implementation, and deployment control of web APIs. Our approach, called EAGER, provides a software architecture that integrates into PaaS platforms to support systemwide, deployment-time enforcement of governance policies. Specifically, EAGER checks for and prevents backward incompatible API changes from being deployed into production PaaS clouds, enforces service reuse, and facilitates enforcement of other best practices in software maintenance via policies. Our experiments with an EAGER prototype show that enforcing API governance at deployment-time in PaaS clouds is efficient and scalable to thousands of APIs and policies.**

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 as much application functionality as possible to remote, web-accessible application programming interfaces (web APIs) hosted "in the cloud". As a result, web APIs are rapidly proliferating. At the time of this writing, ProgrammableWeb [1], a popular web API index, lists over $12,000$ web APIs and a nearly 100% annual growth rate.

This proliferation of web APIs demands new techniques that control and govern the evolution of APIs as a first-class software resource (i.e. API governance) [2]. A lack of API governance can lead to security breaches, denial of service (DoS) attacks, poor code reuse and violation of service-level agreements (SLAs). Unfortunately, most existing cloud platforms within which web APIs are hosted provide only minimal governance support. API governance for cloud platforms consists of:

- *deployment-time governance* in which governance checks are performed when the APIs are deployed to the cloud, and
- *runtime governance* in which governance checks are carried out when the APIs are invoked by clients.

Of these two, deployment-time enforcement (heretofore unexplored) is attractive for several reasons. First, if runtime only API governance is implemented, policy violations will go undetected until the offending APIs are used (at which point it is too late and expensive to take corrective measures). By enforcing governance at deployment-time, cloud platforms can support "fail fast" in which violations are detected immediately, before they become accessible to clients. Further, APIs served from a cloud platform are invoked many more times than they are deployed and redeployed. Therefore, deployment-time API governance helps reduce the overall API governance overhead by eliminating many checks that would otherwise be repeated unnecessarily if executed as runtime checks.

In order to explore the feasibility and the performance traits of deployment-time API governance in cloud platforms, we are developing EAGER (**E**nforced **A**PI **G**overnance **E**ngine for **R**EST) [2], a model and an architecture that augments existing Platform as a Service (PaaS) clouds 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, when APIs are deployed to a PaaS.

EAGER enhances software maintainability by guaranteeing that developers reuse existing APIs when possible to create new software artifacts. Concurrently, it tracks changes made by developers to deployed web APIs to prevent any backwards-incompatible API changes from being put into production. EAGER also includes a language for specifying API governance policies. It incorporates a developer-friendly Python programming language syntax for specifying complex policy statements in a simple and intuitive manner. Moreover, our approach ensures that specifying the required policies is the only additional activity that API providers perform to benefit from EAGER.

To evaluate the usefulness and the performance of deployment-time API governance, we implement EAGER as an extension to AppScale [3], an open source cloud platform that emulates Google App Engine. Through this prototype we show that the EAGER architecture and hence deployment-time API governance can be easily implemented in extant clouds with minimal changes to the underlying platform technology. Further, our performance test results show that EAGER scales well to handle thousands of APIs, policies and API inter-dependencies, thus bringing the cloud computing community several steps closer to supporting comprehensive low-overhead API governance as a cloud-native feature.

II. EAGER

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 (an application may export zero or more web APIs). Short-dashed

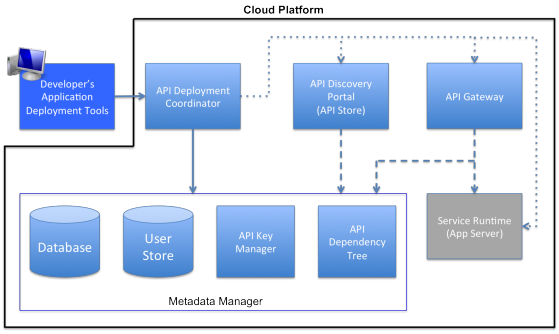

Fig. 1. EAGER Architecture

arrows represent deployment-time interactions that follow application validation. Long-dashed arrows represent runtime interactions.

EAGER is invoked whenever a developer attempts to deploy an application, using the developer tools available on his/her workstation. In some cloud implementations these tools may be available as an online service accessed via a web browser. In either case, the application deployment request is intercepted by EAGER API Deployment Coordinator (ADC), which then performs the required governance checks based on the metadata stored in the Metadata Manager. The Metadata Manager stores application names, versions, dependencies, API specifications, user profiles and API keys. Governance checks are driven by a set of administrator-specified policies that are stored along with the ADC. These policy files are written in Python, and make use of the following assertion functions:

```
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 order to keep all policy executions simple and stateless, EAGER prevents policy code from accessing the file system, network and most third-party Python libraries. Moreover, the policy language prohibits storoage of global state.

If a governance check fails (i.e. assertion failure), EAGER preempts the deployment process and returns an error. Otherwise it proceeds with the application deployment by activating the deployment mechanisms on the developer's or administrator's behalf. Additionally, all application metadata is saved to the Metadata Manager, and if the application contains any web APIs, they will be published to the API Discovery Portal (ADP) and the API Gateway components. The API Discovery Portal is a web GUI that enables application developers to browse and discover available APIs, and obtain the necessary API keys. The API Gateway intercepts API calls at runtime and performs security, rate-limiting, and runtime policy checks.

In addition to the governance policy validations, EAGER also performs a set of built-in sanity checks on all applications and APIs deployed in the PaaS cloud. One of these checks is the backwards compatibility check. If EAGER detects that an API which is already deployed in the cloud is being redeployed, it performs a comparison between the old and latest specifications of the API to make sure that the developer is not introducing a backward incompatible change to the API. This comparison is based on our past and ongoing work related to syntactic and semantic similarity of web APIs [4].

We implemented a prototype of EAGER using the AppScale open source PaaS cloud. Prototype was mostly implemented in Python, while using MySQL as the underlying datastore of the Metadata Manager. The prototype implementation did not require any code changes at the AppScale core. However, we made configuration changes so that AppScale treats EAGER components as built-in elements of the PaaS cloud. As such, we are able to leverage the existing reliability and high availability mechanisms of AppScale to make EAGER components more reliable and high available. We also made some minor code modifications to the AppScale developer tools, so that when a developer attempts to deploy an application into AppScale, the deployment request is routed to the EAGER ADC.

III. 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 (Metadata Manager) with a set of APIs, and examine the overheads associated with governing a set of sample applications for varying degrees of policy specifications and dependencies.

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. Also, our experiments have shown that the absolute overhead introduced by EAGER is very small compared to the overall time taken by AppScale to deploy an application (100's of milliseconds versus to 10's of seconds). Therefore, for clarity and ease of comparison, all the figures presented in this section only show the absolute overhead of EAGER.

Figure 2 shows that EAGER overhead grows linearly with the number of APIs exported by a deployed 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 of 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. In the unlikely case that an application exports 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. Then we deploy an application on EAGER which exports a single API and declares artificial dependencies on the synthesized APIs in the Metadata Manager. We vary the number of declared dependencies and observe the EAGER overhead.

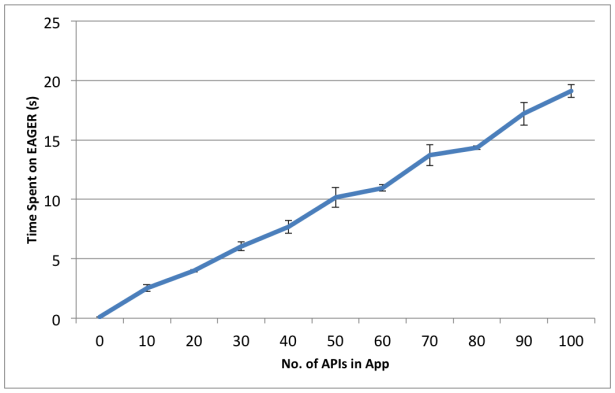

Fig. 2. Average EAGER overhead vs. number of APIs exported by the application.

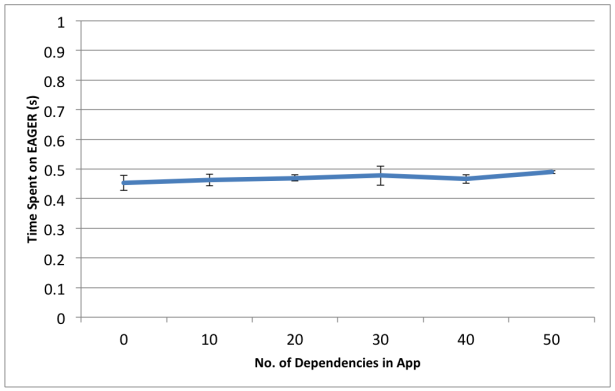

Fig. 3. Average EAGER Overhead vs. number of dependencies declared in the application.

Figure 3 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.

Thus far we have conducted our experiments without active governance policies in the system. We next investigate policy validation. For this study, we consider three simple web apps: guestbook.py which exports no web APIs, and simple-jaxrs-app and dep-jaxrs-app which export two and one APIs, respectively.

The overhead of policy validation depends on policy content. 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.

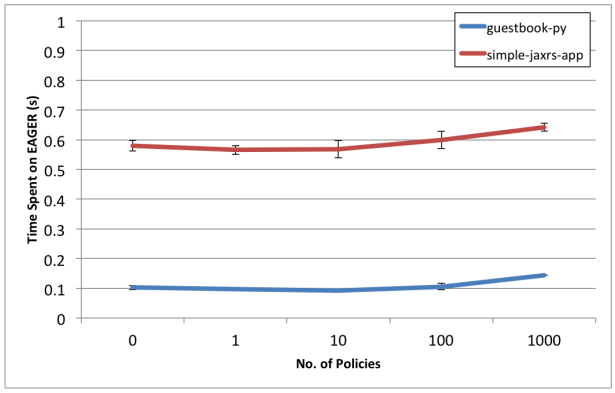

Fig. 4. Average EAGER overhead vs. number of policies.Note that some of the error bars for guestbook-py are smaller than the graph features at this scale and are thus obscured.

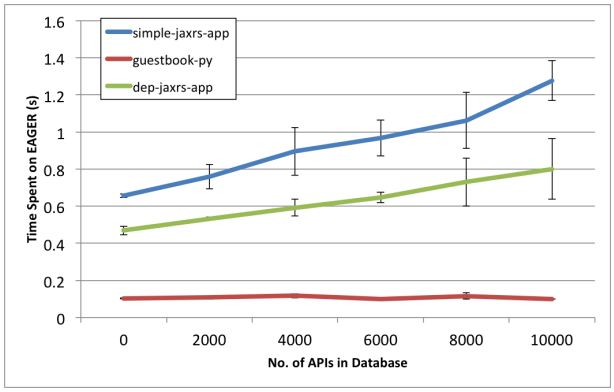

Fig. 5. Average EAGER overhead vs. number of APIs in Metadata Manager. Note that some of the error bars for guestbook-py are smaller than the graph features at this scale and are thus obscured.

Figure 4 shows 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 is 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 it exports two APIs.

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 deploy various sample applications. We create random dependencies among the APIs recorded in the Metadata Manager to make the experimental setting more realistic.

Figure 5 shows that the deployment overhead of the guestbook-py application is not impacted by the growth of metadata in the PaaS. 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 API metadata in EAGER.

Both simple-jaxrs-app and dep-jaxrs-app are affected by the volume of data stored in Metadata Manager, since they export web APIs which need to be recorded and validated by the Metadata Manager. 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. Also, even after deploying 10000 APIs, the overhead on simple-jaxrs-app increases only by 0.5 seconds.

In summary, the current EAGER prototype scales well to thousands of APIs. If further scalability is required, we can employ parallelization and data storage optimizations. 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 overhead. Based on this analysis we conclude that enforced deployment-time API governance can be implemented in modern PaaS clouds with negligible overhead and high scalability. Further, deployment-time API governance can be made an intrinsic component of the PaaS cloud itself thus alleviating the need for poorly integrated third-party API management solutions.

IV. Related Work

Our research builds upon advances in the areas of SOA governance and service management. Guan et al introduced FASWSM [5] a web service management framework for application servers. Wu et al introduced DART-Man [6] a web service management system based on semantic web concepts. Our work is different from these past approaches and from recent API management solutions [7], [8] in that EAGER targets policy *enforcement* and we focus on doing so by extending extant cloud platforms 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 [9]. However, this system only supports run-time service management and provides no support for deployment-time governance. Kikuchi and Aoki [10] proposed a technique based on model checking to evaluate the operational vulnerabilities and fault propagation patterns in cloud services. But this system provides no active monitoring or enforcement functionality.

Other researchers have shown that policies can be used to perform a wide range of governance tasks for SOA such as access control [11], fault diagnosis [12], and management [13]. We build upon these past efforts and use policies to govern RESTful web APIs deployed in cloud settings. 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 [14]. 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 comprehensive manner. Moreover, EAGER's governance features "fail fast" to detect violations immediately.

V. Conclusions

In this paper, we describe EAGER, a model and a software architecture that facilitates deployment-time API governance as a cloud-native feature. EAGER supports comprehensive policy enforcement, dependency management, and a variety of other deployment-time API governance features. Our empirical results show that EAGER adds negligible overhead to the cloud application deployment process, and the overhead grows linearly with the number of APIs deployed. Our future work considers static and dynamic analysis that automates detection of API specifications and dependencies.

References

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

[2] C. Krintz, H. Jayathilaka, S. Dimopoulos, A. Pucher, R. Wolski, and T. Bultan, "Cloud platform support for api governance," in *Intl. Conference on Cloud Engineering*, March 2014, pp. 615–618.

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

[4] H. Jayathilaka, C. Krintz, and R. Wolski, "Towards automatically estimating porting effort between web service apis," in *Services Computing (SCC), 2014 IEEE International Conference on*, June 2014, pp. 774–781.

[5] 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.–.

[6] 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.

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

[8] "Enterprise API Management & API Strategy – http://apigee.com/."

[9] 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.

[10] 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.

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

[12] 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.

[13] B. Suleiman and V. Tosic, "Integration of uml modeling and policy-driven management of web service systems," in *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, ser. PESOS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 75–82. [Online]. Available: http://dx.doi.org/10.1109/PESOS.2009.5068823

[14] 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.