Cloud Platform Support for API Governance

Chandra Krintz, Hiranya Jayathilaka, Stratos Dimopoulos, Alexander Pucher, Rich Wolski, and Tevfik Bultan
Dept. of Computer Science, University of California, Santa Barbara
To Appear: IC2E Future Of PaaS Workshop, March 2014

***Abstract*—As scalable information technology evolves to a more cloud-like model, *digital assets* (code, data and software environments) increasingly require curation as web-accessible services. "Service-izing" digital assets consists of encapsulating assets in software that exposes them to web and mobile applications via well-defined yet flexible, network accessible, application programming interfaces (APIs). In this paper, we postulate that recent advances in cloud computing make cloud platforms as-a-service (PaaS) ideal for deployment, lifecycle management, and policy-based control – i.e. *API governance* – for extant and future digital assets. Toward this end, we overview API governance as a PaaS technology and outline some early results generated by our investigation of a prototype we are developing, called EAGER, for implementing API governance at scale.**

***Index Terms*—API Governance; PaaS; cloud platforms; API similarity;**

I. Introduction

Cloud computing is fostering a model of application development that combines authored code with functionality provided by extant, curated web services. Consumer-targeting applications (particularly those designed for mobile platforms) interact with highly scalable and reliable "back-end" web services that are constantly maintained in well-connected, secure data centers. In addition, enterprise Information Technology (IT) strategies are focusing on deploying both hardware and software infrastructure to host their *digital assets* as web services for controlled access by their employees via internally developed applications or by their customers via software and data "as-a-service".

This rapidly proliferating model of application development and IT operations is designed to scale, both in the number of applications that access the IT-managed services, and in the number of services that must be hosted and curated. Each service exports one or more Application Programming Interfaces (APIs) that must be accessible by users, user applications, and/or other services. Because applications encode their internal logic in terms of remote "calls" to curated services, these APIs define functional boundaries that must be incorporated into the application architecture.

Despite the primacy of APIs in the emerging digital economy, few advances have yet been developed to implement *API governance* – combined policy, implementation, and deployment control of APIs for IT-managed services and digital assets. Some commercial technologies exist that aid implementation of digital assets (including the necessary APIs) and many focus on support for packaging and cataloging APIs [1], [2]. However, technologies for providing stewardship of APIs through all phases of governance are rare.

We believe that recent advances in cloud platform as-a-service (PaaS) technologies make PaaS systems ideally positioned to help IT management with this new yet increasingly important responsibility of API governance. PaaS systems are inherently distributed, scalable, and fault tolerant and have proven successful in automating configuration, deployment, monitoring, and management of a wide variety of web services and their implementations [3], [4], [5], [6]. In addition, PaaS systems provide high-level control over the abstractions they implement and over the software they deploy and manage.

Moreover, some PaaS systems (e.g. Google App Engine and AppScale) take an API-centric approach (verses one based on software stacks) which significantly simplifies implementation of API governance solutions at scale. These PaaS systems decouple the implementation of digital asset access from the technologies that are used to manage and store the assets. That is, while the assets may remain the same, the technologies used to serve and implement them can change, particularly as technological advances reduce implementation costs. This level of indirection can be leveraged to enforce organizational policies that ensure predictable and economic operation of web services and the digital assets they expose.

In the sections that follow, we overview our vision for the next generation of PaaS systems. In particular, we define EAGER – the *E*nforced *A*PI *G*overnance *E*ngine for *R*EST – PaaS support implementing and enforcing API governance. EAGER leverages recent PaaS advances and employs AppScale [3] for its implementation. In this paper, we overview the functionalities and capabilities that we believe are required for API governance for curated digital assets accessed via web services and describe our initial investigations.

II. API Governance for the Next Generation of PaaS Systems

With the advent of cloud computing, digital assets (code, data, and software environments) and not infrastructure are becoming the resources that must capture scientific and research investment and innovation. With high-quality, low-cost compute and storage infrastructure available from public and on-premises clouds, IT management must increase its focus on the maintenance, protection, and lifecycle control of the digital assets – the code, software environments, and data – that comprise the "computational" component of any innovative endeavor.

Moreover, these digital assets are increasingly encapsulated and exposed as services via web application programming interfaces (APIs) for easy access by clients and users via a

network. APIs both define and control what operations can be performed on each asset, by whom, and under what conditions. Moreover, the *lifecycle* of the API follows the lifecycle of its assets and *not* the lifecycle of the surrounding technologies which typically change at a more rapid pace.

The combination of recent advances in cloud platforms, the exploding proliferation of "service-ized" digital assets, and the separation of concerns that both PaaS systems and APIs facilitate, exposes new opportunities for advancing the state-of-the-art in IT management of digital assets. PaaS systems today simplify and automate distributed deployment, elasticity, and fault tolerance of web applications (apps), services, and their runtime ecosystems. Moreover, they decouple digital asset implementation from the technologies that manage and store them. APIs provide a common entry point through which digital assets are accessed that can be monitored and controlled. By leveraging both, we can investigate and develop the next generation in PaaS technologies to facilitate API governance, i.e., the combined policy, implementation, and deployment control of APIs for IT-managed digital asset services.

To facilitate API governance, extant approaches to PaaS must be extended with new functionality (in additional to extant/typical API management features such as cataloging, search, deployment support, etc.) that provides

- *Change Control* – When API changes are necessary, the extent of the effects of the change must be predictable and implemented in a uniform, consistent way. If changes need to be rolled back, the return to previous functionality is likewise consistent, complete, and managed. This requires development of efficient automated change-impact analysis techniques that can determine the potential effects of a proposed change.
- *Policy Specification and Analysis* – Since APIs are gateways to digital assets, they should allow only the authorized clients access these resources. API governance requires development of mechanisms for specification of access control policies, and analysis and runtime enforcement of these policies.
- *Consistent Policy Implementation* – Policies governing the use of digital assets and/or their APIs are implemented consistently across all assets regardless of the constituent technologies that are used to implement the assets themselves.
- *Implementation Portability* – API implementation is decoupled from the implementation of the digital assets. As technologies evolve or, more problematically, devolve when they sunset, API integrity must be maintained across different implementations.
- *Monitoring and Auditing* – API governance must include a unified approach to monitoring and auditing API activity. A unified approach is particularly important when digital assets make heavy use of open source as many research and educational environments do today. Using runtime monitoring, erroneous or malicious behaviors can be identified and resolved by appropriate exception handling mechanisms. This is necessary since clients that interact with an API are typically not controlled by the organization that governs that API.

Toward this end, we propose to leverage PaaS technologies and to extend them with these key functionalities to provide a unifying framework for implementing and enforcing API governance called EAGER – the *E*nforced *A*PI *G*overnance *E*ngine for *R*EST. Governance policies in EAGER specify the conditions that must be met before an API is exposed to users and while it is in service. The policies themselves cover IT management functions such as change control, versioning and dependency management, auditing, and access control. In a dynamic and scalable cloud setting, such IT governance functions must be implemented in a reliable, auditable, and automated way to preserve the scaling benefits that cloud computing makes possible.

EAGER also extends PaaS with support for the specification and analysis of governance policies for APIs and their application to cloud-hosted web services statically, during deployment, and at runtime. EAGER targets RESTful web services since REST is the predominant service architecture used for web and cloud APIs and REST governance solutions are incomplete and ad hoc. In summary, EAGER supports and provides automation for API policy specification, static and dynamic policy verification, and scalable load-time and runtime policy enforcement throughout the lifecycles of APIs.

III. Initial Investigations into EAGER

To implement EAGER, we employ the AppScale PaaS as our research vehicle and software infrastructure. AppScale is a distributed, scalable, and elastic runtime system that we have developed in prior work to facilitate research into and development of the next-generation of cloud programming systems. AppScale is open source, is easy to use and deploy over on-premise or public IaaS systems, and modularly integrates, via a unifying set of APIs, a wide range of cloud services, technologies, and research innovations that can be compared, contrasted, extended, and empirically evaluated. We extend AppScale to provide the capabilities required to support EAGER (scalable API governance) within an open source distributed cloud platform. In particular, we extend AppScale with

- Policy language support that unifies policy specification across different API policy types (e.g., business, maintenance, security and access, and performance) that is amenable to fast verification and analysis,
- Policy verification and translation techniques for statically checking the correctness of policies and for automatically generating test cases, deployment, and runtime checks,
- Deployment techniques that manage the distributed environment into which APIs and policies are added, updated, and removed, and
- Runtime techniques that enforce runtime policies, synchronize test/dev and production deployments, and provide API deployment rollback.

Our early investigations into EAGER have two foci: deployment-time and runtime API policy enforcement, and tools for API analysis that facilitate change control. To enable the former, we integrate API management (i.e. the WSO2 API Manager [7]) into AppScale. For end-to-end policy enforcement for APIs, we decouple the API manager from the extensive web services technologies to which it is linked and extend it with new features that integrate it into the cloud platform and interoperate with our EAGER language tools, deployment service, and runtime mechanisms. The EAGER API manager intercepts API deployment and access requests in the PaaS and enforces policies at deployment and run time.

To facilitate change control, we provide a tool that developers, IT staff, and business stakeholders can use to reason about the differences between related APIs, so that they may ultimately control the impact of API churn on their API consumers via EAGER governance policies. API "churn" refers to the rapid evolution (versioning and replacement) that APIs experience as a result of consumer demands, competition, and market pressure. Churn has become increasingly frequent for many popular APIs and is particularly disruptive when API changes require client-side code modification. For example, there have been 86 releases of the Amazon EC2 service from August 2006 to August 2013 [8]. Twitter released the version 1.1 of their web API on September 2012, and pulled the version 1.0 of out of production on May 2013 [9]. eBay has released 13 versions of their trading API during the first two quarters of year 2013 alone [10]. The licensing terms of the Amazon Product Advertising API have changed twice between the years of 2011 and 2013 [11]. This churn has impacted vast numbers of individuals and client-side technologies.

Our approach employs a formal and automated mechanism for quantifying the similarity of two APIs or two API versions. Our tool uses this information to estimate client-side "porting effort", i.e. the work that is needed to modify code that consumes and accesses the API so that it can consume and access a new or different version of the API. Key to our approach is that we estimate porting effort (via API similarity) without requiring client-side code analysis (requiring such is not scalable and is infeasible in many cases).

API similarity can be analyzed from two perspectives:

- *Syntactic similarity* – Similarity of the inputs and outputs of web APIs, and
- *Semantic similarity* – Functional and behavioral similarity of web APIs.

Of the two, checking for syntactic similarity is a solved problem. Given a machine-readable description of the inputs and outputs of the APIs, static analysis methods can be used to verify whether two web services APIs are syntactically compatible. However, checking the semantic compatibility of web APIs is significantly more challenging. Existing semantic matchmaking methods solve this problem using semantic ontologies, process models or state machine models (e.g. [12], [13], [14], [15], [16]), all of which are complex, laborious and potentially computationally intensive. We address this limitation via a significantly more efficient technique that is simple and scalable to implement.

Our approach uses axiomatic semantics to describe the functionality and behavior of web APIs. We document the semantics in a machine-readable manner using a Python subset specifically designed to capture web service API characteristics. We keep our approach simple by disallowing complex programming constructs (e.g. loops, functions etc.), and restricting the subset to a side-effect-free programming model when documenting API semantics. Then we derive abstract syntax tree (AST) representations [17] of semantic predicates expressed in our language to compare and reason about the semantic similarity of different web APIs. We use a Dice coefficient [18] based AST similarity algorithm and Hoare's consequence rule [19] to compute a *porting effort score* for any two given web APIs.

IV. Initial Findings

To assess our approach for estimating porting effort and API similarity, we have implemented a EAGER prototype. Using this prototype, we have empirically evaluated its efficacy using randomly generated APIs with which we can control API similarity *a priori*. We have also analyzed the prototype using a number of web APIs from popular e-commerce and social networking venues. Our experimental results indicate that our mechanism is efficient, and delivers accurate results under most circumstances. We have further tested the validity of our approach by comparing the results computed by our formal mechanism with those provided by some human developers when manually analyzing several web APIs.

Unsurprisingly, the initial experiments performed using randomly generated web API specifications show that the porting effort between APIs tends to increase with the number of semantic predicates. As the number of semantic predicates increases, the API consumer (*e.g.* the developer of the application consuming the API) is put under more and more restrictions. Therefore, when porting among different web APIs, the developer has to take more constraints into account, and do more patchwork to reconcile the differences among these restrictions. These activities increase porting effort and the experimental results suggest that our porting effort evaluation mechanism captures this phenomenon accurately.

When considering real world web APIs, we observe a fairly large proportion of the API pairs within a set of APIs ostensibly serving the same function, have a low porting effort score. For example, we looked at three API populations: social media, airline e-commerce, and video search. In each of them, 50% of the pairs have a low porting effort score, a modal characteristic not present in the data obtained from the randomly generated APIs. Again, this result is confirmational with respect to intuition. In API populations that serve similar web services most APIs have a lot in common with each other. For example, most social media login APIs have similar constraints on username and password. Most airline APIs have similar requirements with respect to specifying departure and arrival cities, travel dates and the number of passengers. Most

video search APIs also have some constraints in common, in the sense most APIs at least accept simple text queries to perform keyword-based search. These similarities explain how current mobile and desktop applications currently manage multiple APIs in e-commerce and social computing settings.

Finally, we asked student developers who were in the process of building applications to access some of these venues for their respective scoring of the difficulty associated with porting from on API to another. We then applied statistical clustering to both the scoring results generated by our methodology and to the results given to us by the developers. The clustering results show a strong mapping between our methodology and human perception of porting effort with respect to categorizing a particular port as "hard" or "easy."

V. Future Work

EAGER's API similarity toolset provides a way for developers and business stakeholders to reason about how clients/consumers of their APIs will be impacted by potential changes. We plan to extend this toolset to automate analysis syntactic of syntactic compatibility as part of future work. Most web APIs use data types such as XML and JSON for receiving inputs and sending outputs. This uniformity enables modeling the web service inputs and outputs using a structured data model consisting of a simple type system. Once we formulate such a data model, it is straightforward to automate the process.

We also believe that our approach to semantic specification of APIs in machine readable form can be extended to specify access control policies. Using a restricted form of an existing programming language (such as Python) provides two benefits: 1) It does not require the users to learn a new language for policy specification, 2) It allows us to appropriately restrict the language to enable automated analysis. Using this approach, we plan to develop automated techniques for policy analysis, such as checking if a policy is stronger or weaker than another policy, and change-impact analyses that identify the effects of a policy change. We plan to implement these analyses by translating the analysis questions to satisfiability queries in decidable theories and then using automated decision procedures (such as SAT-Solvers or SMT-Solvers). Focusing on API usage and using a restricted policy language will enable us to develop a scalable static analysis framework for API policies.

Finally, we are investigating other aspects of API governance within the EAGER framework. In particular, we are developing methodologies for automating policy-based change control and auditing capabilities to manage APIs in large-scale deployments. Cloud computing, as both a research and commercial discipline, has developed a number of new approaches for managing "soft" resources in highly scalable settings. We plan to leverage many of these developments to develop API governance methods for scalable systems.

VI. Conclusion

APIs have emerged as a key component of the modern digital economy and we believe the scientific community will want to leverage the technological developments that this primacy is engendering. However, even though APIs are the longest-lived and most expensive software artifacts, little research has yet focused on what is necessary to implement good IT governance of them.

Our work is taking an initial step towards the development of a system for implementing API governance by leveraging recent advances in cloud platform as-a-service technologies. This paper describes our approach and platform (called EAGER), and overviews our initial investigations into tools that help developers and business stakeholders reason about API similarity and change control as part of the API governance process and that facilitates EAGER enforcement of policy-based change control. Our initial results are promising, leading us to conclude that API governance is worthy of on-going investigation and that it will play a key role in the next generation of PaaS systems.

VII. Acknowledgements

We thank the reviewers for their valuable feedback. This work was funded in part by NSF (CNS-0905237 and CNS-1218808) and NIH (1R01EB014877-01).

References

[1] "Mashery," "http://www.mashery.com" [Online; acc 27-Sept-2013].
[2] "Layer7," "http://www.layer7tech.com" [Online; acc 27-Sept-2013].
[3] C. Krintz, "The appscale cloud platform: Enabling portable, scalable web application deployment," in *Internet Computing*, 2013.
[4] "Google app engine," "http://code.google.com/appengine/" [Online; acc 27-Sept-2013].
[5] "Microsoft windows azure," "http://http://www.windowsazure.com/en-us/" [Online; acc 27-Sept-2013].
[6] "Amazon Web Services," "http://aws.amazon.com" [Online; acc 27-Sept-2013].
[7] "WSO2 API Manager," http://wso2.com/products/api-manager/, 2013, [Online; acc 27-Sept-2013].
[8] "Release Notes: Amazon Web Services," http://aws.amazon.com/releasenotes/Amazon-EC2, 2013, [Online; acc 02-Sept-2013].
[9] "Twitter API v1 Retirement: Final Dates," "https://dev.twitter.com/blog/api-v1-retirement-final-dates" [Online; acc 27-Sept-2013].
[10] "eBay Trading Web Services: Release Notes," "http://developer.ebay.com/DevZone/XML/docs/ReleaseNotes.html" [Online; acc 27-Sept-2013].
[11] "Product Advertising API," "https://affiliate-program.amazon.com/gp/advertising/api/detail/agreement-changes.html" [Online; acc 27-Sept-2013].
[12] G. Salaun, L. Bordeaux, and M. Schaerf, "Describing and reasoning on web services using process algebra," in *IEEE Web Services*, 2004, pp. 43–50.
[13] Z. Shen and J. Su, "Web service discovery based on behavior signatures," in *Services Computing*, 2005, pp. 279–286.
[14] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire, "Runtime verification of web service interface contracts," *IEEE Computer*, vol. 43, no. 3, pp. 59–66, Mar. 2010.
[15] H. S. Kim, S. I. Kim, and W. Jung, "Ontology modeling for rest open apis and web service mash-up method," in *Information Networking (ICOIN)*, 2013, pp. 523–528.
[16] M. Melchiori, "Hybrid techniques for web apis recommendation," in *International Workshop on Linked Web Data Management*, 2011, pp. 17–23.
[17] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance*, 1998, pp. 368–377.
[18] W. B. Frakes, "Stemming algorithms." 1992.
[19] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.