2015

Toward Design Decisions to Enable Deployability Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail

Stephany Bellomo
Carnegie Mellon University, sbellomo@sei.cmu.edu

Neil Ernst
Carnegie Mellon University, nernst@sei.cmu.edu

Robert Nord
Carnegie Mellon University, rn@sei.cmu.edu

Rick Kazman
Carnegie Mellon University, rkazman@sei.cmu.edu

Toward Design Decisions to Enable Deployability

Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail

Stephany Bellomo, Neil Ernst, Robert Nord, and Rick Kazman
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
sbellomo, nernst, rn, kazman@sei.cmu.edu

***Abstract*—There is growing interest in continuous delivery practices to enable rapid and reliable deployment. While practices are important, we suggest architectural design decisions are equally important for projects to achieve goals such continuous integration (CI) build, automated testing and reduced deployment-cycle time. Architectural design decisions that conflict with deployability goals can impede the team's ability to achieve the desired state of deployment and may result in substantial technical debt. To explore this assertion, we interviewed three project teams striving to practicing continuous delivery. In this paper, we summarize examples of the deployability goals for each project as well as the architectural decisions that they have made to enable deployability. We present the deployability goals, design decisions, and deployability tactics collected and summarize the design tactics derived from the interviews in the form of an initial draft version hierarchical deployability tactic tree.**

Keywords—deployability; continuous integration; continuous delivery; architecture tactic; test automation

I. INTRODUCTION

There is substantial interest in *practices* for achieving continuous delivery and rapid, robust deployment goals [1], [2], [3]. However, we suggest there is a need to focus on more than practices. The architecture of the target application can also contribute to enabling or impeding deployability goals. If teams fail to make the right *architecture design decisions and tradeoffs* as they build and evolve the system, then critical activities such as continuous build integration, automated test execution, and operational support can become difficult. For example, we have observed cases where a tightly coupled component architecture becameme a barrier to continuous integration (CI) because small changes required a rebuild of the entire system. This limits the number of builds possible in a day and, in some cases, a CI build is not even possible in a single day. Re-architecting to fix problems such as these can require significant work and in some cases these types of problems can become a form of technical debt [4], resulting in high expenditures of time, cost, and effort release after release.

Architecting for deployability benefits software developers who are increasingly called upon to support systems post release, but I also benefits those in operational and release engineering roles responsible for the activities described above (e.g., CI build tools/support, test automation infrastructure, deployment automation). To explore our hypothesis that it is beneficial to consider deployability architectural design implications early in a development effort, we conducted interviews with three project teams practicing continuous integration and delivery. In this paper, we summarize these as case study analysis results. The results we present here are derived from responses to two overarching interview questions consistently asked in each interview:

- IQ1: What are the key goals driving your deployability efforts?
- IQ2: What are some examples of architecture decisions that have enabled these goals?

From the interview data, we summarized examples of deployability goals to gain a better understanding of the desired state that each project aimed to achieve (as described in [5]). In addition, we probed beyond goals to collect examples of architectural design decisions that the project teams felt enabled achievement of their deployability goals.

As we analyzed the results, we found that many of the design decisions the project teams made were instances or variations of previously defined architecture tactics[6]. An *architecture tactic* is a design primitive that an architect can use to satisfy a quality attribute requirement. We claim that deployability can similarly be described and elaborated as a quality attribute. Doing this requires an understanding of stakeholder goals, constraints, requirements (e.g., quality attribute scenarios [6]), and design decisions (e.g., architectural tactics).

The idea of deployability as a quality attribute is not new. A brief description of deployability as a quality attribute is presented in [6]. Adams et al. explore the relationship between rapid deployment and quality [7], [8]. Bass et al. propose use of operational scenarios focusing on cloud and dependability networking to more concretely define operational requirements and identify architecture-related design decisions [9]. Cukier shares a mixture of development and system operations as well as architectural patterns for cloud-based web applications [10]. Spinellis looks at infrastructure as code and design implications of deployable systems [11] as well as tools and techniques for integrating development and operations [12]. Liu et al. suggest a framework for integrating cloud-based application development and production frameworks seamlessly [13]. Schaefer et al. explore automation approaches to promote environment consistency and reduce manual work to enable CI [14]. Gohil et al. experiment with behavior-driven monitoring and how it can be adopted for infrastructure provisioning and deployment [15]. Work in this area is still in the early stages and continues to evolve.

In this paper, we summarize our findings from interviews conducted with three project teams from different

organizations. We summarize deployability goals, design decisions, and tactics that we elicited. The deployability tactics we collected are summarized in a tactic tree. The deployability goals that we collected through interviews form the first level of the tree. The set of deployability tactics that we collected through interviews form the lower branches. The derived tactics tree contains new tactics as well as previously defined tactics that crosscut (or are variations of) existing tactics in quality attribute areas such as modifiability, testability, availability, and performance.

II. Research Method Description

We conducted three interviews with technical leads and architects of projects that have deployability (or continuous delivery) as a major focus. To collect data, we asked each interviewee the questions listed above as IQ1 and IQ2. We probed for "incident descriptions" that allowed us to collect concrete examples from which to derive results. We recorded each interview and extracted raw examples from the recorded transcripts . The results are summarized and presented in Section III.

A. Project Profile

We interviewed technical leads and architects from three projects, which we will refer to as Projects A, B, and C. These teams were from three different organizations working on different types of software projects. The organization for Project A primarily develops federal business systems. The application Project A is currently building enables clients to buy and sell securities. The organization for Project B is an academic institution and the application provides a heavily used virtual training environment with e-learning and virtual lab capability. The organization for Project C a large software contractor building a sales portal for financial transactions. While A and C coincidentally support financial applications, the project teams are from different organizations and the capabilities the applications provide are very different from each other. We provide a high-level profile of some of the project characteristics in Table I.

From Table I we see that all three projects used some variant of an Agile/Scrum project management framework. The size and SLOC varied, but all were large projects. All of the systems have been in operational use for several years. All were releasing internally every two to three weeks for client feedback but were externally deploying only every two or three months. All projects practiced daily CI. More project details, such as brief architecture description, languages, and so forth, are provided in Section III.

TABLE I. Project Profile

Project	Management Approach	Size Metrics	Years Operational	Release Cadence	CI Cadence
A	Agile/Scrum (last 2 years and traditional before that)	1M SLOC	17	Client release available every 2 months (not all accept it)	Daily CI build
B	Water-Scrum-Fall	3M SLOC, team size 6–8, 90,000 users	3+	Internal release every 2–3 weeks, external release as needed	Daily CI build
C	Agile/Scrum	Team size 30	2+	Internal release every 2–3 weeks, customer release every 2–3 months	Daily CI build

III. Research Observations

In this section, we describe example deployability goals from Projects A, B, and C followed by raw design-decision examples and tactics. We conclude the section with a first draft of an architecture tactics tree which summarizes the examples provided in the interview data.

A. Deployability Goals Summary (by Project)

Below we summarize responses to IQ1: What are the key goals driving your deployability efforts? The interviewees were not familiar with the term *deployability goals*, so we interchangeably used the term *continuous delivery goals* during the IQ1 data collection.

Project A Goals:
Goal 1: Shorten feedback cycle time and integrate frequently to avoid integration problems
Goal 2: Enable nightly integration builds with successfully run automated tests
Goal 3: Simplify deployment and minimize deployment time

Project B Goals:
Goal 1: Achieve actual CI (beyond daily integration)
Goal 2: Promote the habit of frequent and automated means of deploying

Project C Goals:
Goal 1: Enable CI and delivery
Goal 2: Enable extensive test automation to include unit, functional, acceptance, and other types of testing
Goal 3: Automate deployment and increase frequency and comfort with deployment
Goal 4: Reduce the gap between environments to keep them consistent and reduce deployment errors and complexity

To synthesize this information in Table II, we affinity grouped the goals into more general goals (as shown in the middle column of Table II). We then mapped these generalized goal names to the specific project goals to show which projects are focused on which goals.

TABLE II. Deployability Goals Summary

	Generalized Name of Deployability Goal	Mapping of Project Goals to Generalized Goal
G1	Enable Build and Continuous Integration	PA-G1, PB-G1, PC-G1
G2	Enable Test Automation	PA-G2, PB-G2, PB-G2
G3	Enable Deployment and Robust Operations	PA-G3, PB-G2, PB-G3
G4	Enable Synchronized and Flexible Environments	PC-G3, PC-G4

B. *Examples of Design Decisions to Support Deployability*

In this section, we provide a summary of the raw-data design decisions that we gathered from Projects A, B, and C in response to IQ2: What are some examples of architecture decisions that have enabled these goals?

For each design decision, we suggest a generalizable architectural tactic that the decision instantiates. The tactics are drawn either from our existing work on architecture tactics [4], a variation of an existing tactic (beyond the instantiation), or new tactics.

Project A is a financial application that has several customized variants of the software deployed to eight client sites. The architecture for Project A is a thick client written in C# with a C++ and Java back end, which is currently being ported to Java. Examples of Project A's design decisions are summarized below:

- **PA-D1: Integrated test framework.** Project A built an integrated test framework to allow the team to simulate the performance of the system under varying conditions. For example, as part of the nightly build, they used the framework to process batches of transactions and monitor performance to see if it falls below an established threshold. The integrated test framework supports testing of distributed message communication (e.g., message queues and backend processes).
  Initial Tactic Assessment: Integrated test framework is an instance or a variation of the Testability tactic Specialized Access Routines/Interfaces.
- **PA-D2: Script-driven process shutdown.** The Caching tactic was used to improve performance. However, this change made it difficult to put the system in an appropriate state to run a clean test (i.e., to clear the cache). As a result, often the team would have to shut down the system several times during the nightly build to obtain a clean test state. To enable automated performance testing using the integrated test framework, the team built "hooks" into the application to allow for stopping and restarting processes. This allowed the team to ensure that all tests begin in a good, known state.
  Initial Tactic Assessment: Stopping and starting of processes for testing is an instance or variation of the Testability tactic Record/Playback. Caching is an instance of the Performance tactic Maintain Multiple Copies.
- **PA-D3: Web service consolidation.** The application has a C# thick client, and updating the software on the client side required configuration of four web services. The customer organization found this to be a time-consuming and error-prone deployment process. So the team consolidated the four web services to a single web service. To make this change, additional interfaces were created in the single web service, and the addresses of the calls in the front end of the application were modified to reflect the new URLs for the existing interfaces. The interfaces in the combined web service had much in common, such as shared data (database tables) and shared classes, so the change resulted in increased semantic cohesion. A problem with the prior implementation was that each web service had its own set of database connection pools, so one service could potentially run out of database connections while another service still had some available. By consolidating to a single web service, they were also able to have a single, larger pool of database connections.
  Initial Tactic Assessment: Combining web services into one web service with several interfaces is an instance of the Modifiability tactic Increase Semantic Cohesion. The connection-pooling decision is an instance of the Modifiability tactic Abstract Common Services as well as the Performance tactic Reduce Overhead.
  We note that there is a trade-off being made in this example (PA-D3). The team is decreasing semantic coherence and increasing coupling, which can reduce modifiability, to reduce deployment time/complexity and minimize performance overhead.
- **PA-D4: Parameterization.** The team made use of parameterization for environment variables such as database and server names. This allowed the client to change these as needed without changing the build.
  Initial Tactic Assessment: Use of Parameterization and Use of Configuration Files are subtactics or variations of the Modifiability tactic Defer Binding Time.
- **PA-D5: Self-monitoring.** The team added alerting to monitor the system during operation. The team employed proactive internally and externally driven logging. For some critical components, they have incorporated the capability for components to do a self-check to detect internal component faults or failures. In other cases they use a polling approach to detect failures that must be checked externally (e.g., to check that the message exchange state is functioning as expected). Based on the output of the detection mechanisms captured in the logs, alerts can be configured to send emails and messages or customized to be integrated with client systems.
  Initial Tactic Assessment: Polling and component self-checks are instances of the Availability tactics Monitor, Self-test, Ping/Echo, or Heartbeat. Notification can be considered part of Availability tactics Exception Detection and Exception Handling.

Project B is a virtual training environment with an e-learning system and virtual lab capability. The architecture for Project B is a cloud platform (software as a service) with virtualization to provide a training sandbox. Languages used on Project B include .NET, C#, Java, HTML, CFS, jQuery, and Javascript. Examples of Project B design decisions gathered in response to IQ2 are summarized below:

- **PB-D1: Adapter container.** Project B leveraged automated deployment scripting, deployment-focused configuration management tools (e.g., Chef), and virtualized environment generation tools (e.g., Vagrant). The team described use of an adapter container that lives within specific environments, or on a specific virtual machine environment, which allows them to run the same Chef scripts in multiple

environments (e.g., development, staging, and production). This promotes environment consistency.
Initial Tactic Assessment: Configuration Files are subtactics or variations of the Modifiability tactic Defer Binding Time. The adapter container has similarity with the Testability tactic Sandbox; however, in this usage context the container enables rapid and consistent deployment. Virtualization is used to create and manage the environments in which the applications run.

- **PB-D2: Single-responsibility principle and distributed service architecture.** The team described use of the single-responsibility principle, which is an instance of the Increase Semantic Coherence tactic, to support unit testing and rapid deployment. They designed methods and classes as isolated services with very small responsibilities and well-defined interfaces. This allows the team to test individual units independently and to write (mocks of) the inputs and outputs of each interface. It also allows them to test the interfaces in isolation without having to interact with the entire system. They also use services to communicate independently (e.g., Web server APIs or RPCs), so they can have the individual, fine-grained permissions to access the database and specific services. They added that modularizing features, lower coupling, and increased cohesion enable deployment and continuous delivery, explaining, "otherwise you may have to push the whole three million lines of application every time a change is made and if you have to do that you are in a world of hurt."
  Initial Tactic Assessment: The single-responsibility principle is another way of describing the Modifiability tactic Increase Semantic Coherence. Writing small, encapsulated unit tests is an example of the Modifiability tactic Encapsulation and Maintain Existing Interface.
- **PB-D3: Managing and reproducing state.** Project B suggested that it is helpful to design the system such that it is possible to inject the state easily for automated testing (e.g., a database or cache rather than RAM). This required making changes to the architecture to reproduce the state. They suggested that management of state to support testing is a design consideration that must be considered early because of implications over scope of control for development artifacts such as containers, application, and data. An approach Project B says they have observed in practice, but that they do not subscribe to, is pushing shared services logic into the container (e.g., Java authentication and authorization). This is not preferred because the teams says their developers have less control to debug and run automated tests when they can't work within isolated environments.
  Initial Tactic Assessment: State injection to support automated testing is a variant of the Testability tactic Record/Playback. The changes made to support state injection are an instance of the Testability tactic Specialized Access Routines/Interfaces.
- **PB-D4: Self-initiating version update (supports Version Control.** Project B described challenges with the client site database version getting out of sync with the application. In response, they wrote an application version-checking harness that checks database version upon user login and automatically runs scripts to update the database (if needed) to align with the current version of the application. Application changes to enable this capability were made to the presentation layer and business/data layers, such as the ability to detect database state upon login. This self-initiating update approach also supports availability because the upgrade approach did not require taking down the application server.
  Initial Tactic Assessment: Login-initiated version monitoring is similar to the Availability tactic Condition Monitoring; however, the usage focuses on updating an element in the application (the database) rather than handling fault or failure. The ability to bring the database to the correct version is a variation of the Availability tactic State Resynchronization.
- **PB-D5: Monitoring and auto-scaling.** Load balancing and monitoring capabilities were used to monitor and manage average load cycles. To support scalability, they added capability for worker components to grab jobs as they become free or start more workers if needed. This approach supports the ability to scale up and down in an automated manner.
  Initial Tactic Assessment: Load balancing is an example of the Performance tactics Maintain Multiple Copies and Increase Available Resources.

Project C is a financial application sales portal. The architecture is a distributed set of services with a front end built using Java Server Faces and a backend Microsoft SQL server database. Project C was originally built with a middle tier leveraging web services but was later re-architected to use Enterprise Java Beans instead of web services. Examples of Project C design decisions gathered in response to IQ2 are summarized below:

- **PC-D1: Removing web services and collapsing the middle tier.** Project C struggled with version synchronization between application and web services, latency issues, and an overly complex deployment configuration. They made the decision to remove web services used for data access from architecture and rewrite these as Enterprise JavaBeans—essentially, the collapsing of the entire middle tier. This change immediately improved performance, eliminating marshaling and un-marshaling of XML and middle-tier transformation. With this change they could deploy the entire application as one file (one version) to the primary and replicated server environments. They also eliminated a configuration in which the application servers had to be updated and managed separately from the middle-tier servers. After the change, the servers were exactly the same, which eliminated synchronization of the web-tier and the middle-tier deployment. Finally, this change also made testing easier, because developers could test without having to set up and utilize web services.

Initial Tactic Assessment: The removal of web services in this example is an instance of the Performance tactic Reduce Overhead.
It is also important to note the trade-off being made in this example. This removal of the web services layer promotes performance and reduces deployment time and complexity, which may influence modifiability.

- **PC-D2: Parameterization.** Parameterization is used to allow for changing content such as branding or messages and marketing text. Project C also suggested avoiding use of static variables for ease of testing.
  Initial Tactic Assessment: Use of Parameterization and use of Configuration Files are subtactics/variations of the Modifiability tactic Defer Binding Time.
- **PC-D3: Load balancer.** Project C uses a load balancer to provide active and passive redundancy. This also allowed them to roll out a change by switching from one application server to another (described as "blue-green switching" in *Continuous Delivery* [1]). If there are problems with the release, the other version is still running and users can be redirected to the other server.
  Initial Tactic Assessment: The load balancing capability leverages the Performance tactic Maintain Multiple Copies and the Availability tactic Active Redundancy.
- **PC-D4: Bundle and rollback feature and data layer change.** The team used a tool called RedGate to deploy database SQL changes with a feature. With this approach, the database scripts are checked in and the SQL is automatically run against the database by the build tool. This approach allows application feature and database changes to be rolled back simultaneously.
  Initial Tactic Assessment: The bundling and rollback capability leverages the Availability tactic Rollback.

C. *Deployability Tactics Strawman Tree*

Fig. 1 is a straw-man deployability tactics tree derived from raw data in Section III.B intended to summarize the tactics collected from the three case study interviews. Most of the tactics captured in the interview data crosscut existing tactic trees. It is premature to try to say with absolutely certainty whether these are existing "as is", new or variations of existing tactics. However, based on the interview data we have collected to date we felt comfortable making an initial assessment which we hope to validate through follow on work. In the Fig 1 we also point out that the modifiability-related tactics shown under Enable Continuous Integration also enable Test Automation. The crosscutting tactics shown in Fig 1 are primarily from the performance, modifiability, testability, and availability tactics trees.

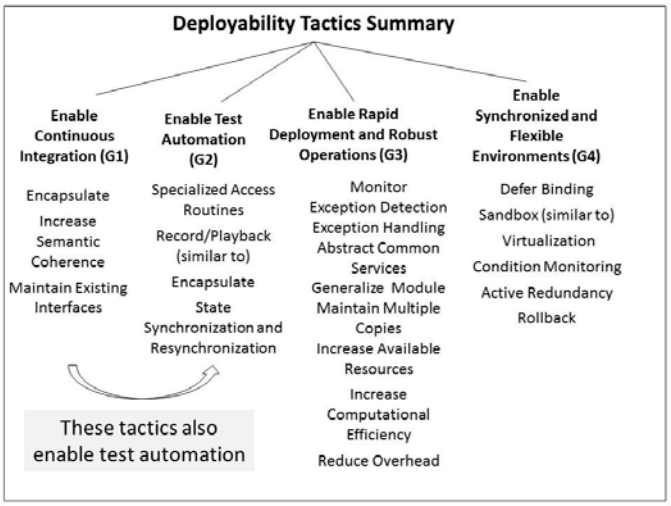

FIG. 1. DEPLOYABILITY TACTICS TREE

IV. DISCUSSION

In addition to employing software application-related architectural decisions and tactics, all three projects described using several other enablers that were critical to their success. These were generally a mix of practices and tool/environment support. For example, Project A created an automated database script to minimize update time during database release. Project B leveraged automated deployment scripting, deployment-focused configuration management tools (e.g., Chef), and virtualized environment generation tools (e.g., Vagrant). Project C used a tool called RedGate so they could bundle and deploy database SQL changes with a feature. On Project B, while software architecture was not impacted by adopting these approaches and tools, other related architectural elements, such as network and deployment architecture, were impacted. For example, automated deployment systems had to be reconfigured to communicate with staging and production networks, and all necessary automation client tools had to be installed on virtual machines housing the application.

The examples in the previous paragraph raise an interesting topic that our team spent some time discussing. Traditionally, there has been a fairly strict line drawn between the application and supporting infrastructure and tool environment (e.g., testing tools, configuration management tools, deployment scripts, and other components). The later have been considered external to the system. Examples such as those described by Project B, where decisions related to deployment support significantly impact the infrastructure or network on which the applications run (in some cases, even production environments), beg the question of whether these components and tools traditionally considered external to the application should continue to be thought of as external or whether we are entering an era when the lines are blurred and, perhaps these should be reconsidered as a combined ecosystem.

V. CONCLUSION

This work is intended to explore the hypothesis that architecture contributes to achieving continuous delivery and deployability goals. The concrete tactics captured in this paper provide a start toward validating our hypothesis. Our hope is to continue to investigate this research area through additional empirical research activities. Takeaways from our exploration include the following:

- As we analyzed the data collected in our interviews, we found that many of the key decisions made by the projects were architectural.

- In the examples that we collected in response to IQ1 and IQ2, we found an initial set of embedded goals that we can use to form the top layer of a tactics tree.
- We found that several tactics were employed, and while there are some new variants of tactics, the Deployability tactics crosscut many of the existing Quality Attribute tactics described in [6].
- We found examples of important trade-offs the teams made to reduce deployment time, complexity, and overhead. We suggest that, just like with other quality attributes, it is important to have a clear understanding of stakeholder priorities and visibility of tradeoffs to make the right decisions for short and long term deployability.

ACKNOWLEDGMENTS

We acknowledge Salient Federal Solutions for their technical contribution to this paper.

 This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This material has been approved for public release and unlimited distribution. DM-0001103

REFERENCES

[1] J. Humble and D. Farley, Continuous Delivery. Boston, MA: Addison Wesley, 2010.

[2] D. Feitelson, E. Frachtenberg, and K. Beck, "Development and deployment at Facebook," IEEE Internet Computing, vol. 17, no. 4, pp. 8-17, 2013.

[3] S. Bang, S. Chung, Y. Choh, and M. Dupuis, "A grounded theory analysis of modern web applications: knowledge, skills, and abilities for DevOps," Proc. 2nd Annual Conf. Research in Information Technology, pp. 61-62, 2013.

[4] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: from metaphor to theory and practice," IEEE Software, vol. 29, no. 6, pp. 18-21, 2012.

[5] F. Bachmann, R. L. Nord, and I. Ozkaya, "Architectural tactics to support rapid and agile stability," CrossTalk: The Journal of Defense Software Engineering, vol. 25, no. 3, pp. 20-25, 2012.

[6] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed. Boston, MA: Addison-Wesley, 2012.

[7] M. Mäntyla, F. Khomh, B. Adams, E. Engström, and K. Petersen, "On rapid releases and software testing," Proc. 29th IEEE Intl. Conf. Software Maintenance (ICSM), pp. 20-22, 2013.

[8] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? An empirical case study of Mozilla Firefox," Proc. Working Conf. Mining Software Repositories (MSR), pp. 179-188, 2012.

[9] L. Bass, R. Jeffery, H. Wada, I. Weber, and L. Zhu, "Eliciting operations requirements for applications." Presented at the International Workshop on Release Engineering, 2013.

[10] D. Cukier, "DevOps patterns to scale web applications using cloud services," Proc. 2013 Companion Publication for Conf. Systems, Programming, and Applications: Software for Humanity, pp. 143-152, 2013.

[11] D. Spinellis, "Don't install software by hand," IEEE Software, vol. 29, no. 4, pp. 86-87, 2012.

[12] D. Spinellis, "Package management systems," IEEE Software, vol. 29, no. 2, pp. 84-86, 2012.

[13] S. Hosono, J. He, X. Liu, L. Li, H. Huang, and S. Yoshino, "Fast development platforms and methods for cloud applications," Proc. 2011 IEEE Asia-Pacific Services Computing Conf., APSCC 2011, pp. 94-101, 2011.

[14] A. Schaefer, M. Reichenbach, and D. Fey, "Continuous integration and automation for DevOps," Lecture Notes in Electrical Engineering, vol. 170 LNEE, pp. 345-358, 2013.

[15] K. Gohil, N. Alapati, and S. Joglekar, "Towards behavior driven operations (BDOps)," IET Seminar Digest, Proc. 3rd Intl. Conf. Advances in Recent Technologies in Communication and Computing, pp. 262-264, 2011.