Resolutions to Technical Challenges Regarding the Distributed Development and Deployment of a Node.js Web Application for Cloud Solution Design

Scott Trent†1 Takayuki Kushida†1 Hamid R. Motahari-Nezhad†2 Taiga Nakamura†2 Peifeng Yin†2
Gil Shurek†3 Karen Yorov†3 Cristian Petrut Pertrache†4 Juan Cappi†2 Uma Subramanian†5

Abstract: Our team, a distributed team of over 10 researchers and developers in six countries, has encountered multiple technical issues in the development and deployment of a Node.js based web application to enhance the automation of the design of Cloud Solutions. This paper provides an overview of the goals and uses of our Cloud Solution Design Tool known as COOL, and presents the six major categories of technical challenges we encountered and how we overcame them. These areas include: application reliability, serviceability/maintainability, functionality verification, development coordination, and usage reporting. It is our belief that this information will be useful to other development teams.

Keywords: Node.js, web application, reliability, performance, serviceability, verification, development, reporting.

1. Introduction

This paper describes many of the technical challenges and solutions that our team has encountered in the design and development of a Node.js based web application that provides services to ease the creation of cloud solution design. We developed specific solutions to the various issues surrounding reliability, serviceability, verification, coordination, and reporting as our team grew in number and our application grew in complexity.

The layout of this paper is as follows. After an introduction of the tool, we describe each of five major technical challenges and how we overcame them in this project. We conclude the paper with a summary of the lessons we have learned.

2. Overview of the Cloud Solution Design Tool (COOL)

2.1 High Level Overview

The Cloud Solution Design Tool (hereafter abbreviated as COOL) that our team is developing has been previously described in papers by Kushida [1, 2, 3] and Motahari-Nezhad [4]. Given the increasing need for and adoption of cloud based infrastructure deployment, there is also an increased need for low-overhead, easy-to-use tools to design such solutions. The design tool that our team has developed enables a solution architect to enter by hand as well as import information regarding a cloud deal from an existing sales databases and from spreadsheet files. This information includes overview information, business requirement questions, and detailed IT Requirements regarding the existing installation (if any) and the desired environment. The initial release of the design tool focuses on supporting the design of comprehensive SAP installations with multiple servers in multiple SAP landscapes across multiple data centers. The tool requests information from the solution architect in order to generate an architectural solution including memory, storage, and processing capacity, number, disaster recovery, high availability, required hardware, software, and version information, etc. This information is then verified against our extensible model for completeness and suitability. This is used to create a draft solution complete with the required feature codes (i.e., purchasable components which make up a sales offering). This solution is then created in a data format which can be used in the next step in the sales process. The intent of this tool is not only to make it easier for a solution architect to design a cloud solution, but to decrease the end-to-end time required for a cloud sales engagement. The architectural overview of this tool is graphically illustrated in Figure 1.

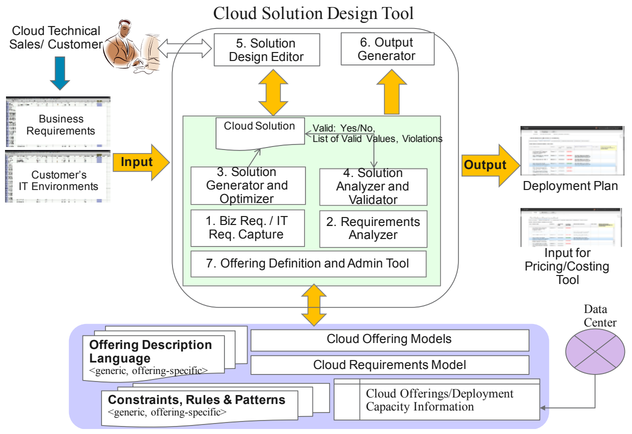
Cloud Solution Design Tool
Cloud Technical Sales/ Customer
Business Requirements
Customer's IT Environments
Input
5. Solution Design Editor
6. Output Generator
Cloud Solution
Valid: Yes/No, List of Valid Values, Violations
3. Solution Generator and Optimizer
4. Solution Analyzer and Validator
1. Biz Req. / IT Req. Capture
2. Requirements Analyzer
7. Offering Definition and Admin Tool
Output
Deployment Plan
Input for Pricing/Costing Tool
Data Center
Offering Description Language <generic, offering-specific>
Cloud Offering Models
Cloud Requirements Model
Constraints, Rules & Patterns <generic, offering-specific>
Cloud Offerings/Deployment Capacity Information

Figure 1. The architecture of the Cloud Solution Design Tool (COOL) [4]

2.2 Technical Architecture of Cloud Solution Design Tool (COOL)

To enable rapid prototyping and development of a web based application, The Cloud Solution Design Tool is an Express web application largely written in Javascript served by Node.js [5]. Some of the primary modules used include Express as the web

†1 IBM Research – Tokyo
†2 IBM Research – Almaden
†3 IBM Research – Haifa
†4 IBM Romania
†5 IBM Cloud Business Unit

application framework, csv-parse and xlsx for importing and exporting spreadsheet data, multer for file upload, passport and passport-ldapauth for ldap authentication, pug as a template language, and Jasmine for unit testing. MongoDB is used for database storage. In many cases, due to flexibility and built-in availability in Javascript, JSON strings are used for data interchange, permanent storage, configuration and customization. Certain operations including constraint solving are performed with external calls to separate applications written in C or Java. Although, by carefully avoiding the use of our reference platform in development, our development team has shown the flexibility of this tool by running it on a wide variety of platforms including multiple flavors of Linux, MacOS, Windows, as well as our reference platform, Red Hat Enterprise Linux version 6.

3. Technical Challenges and Solutions Regarding Project Development and Deployment

Although the project included many facets including business requirement definition, model creation, constraint solving, content creation, etc., this paper will focus on the development and deployment issues surrounding this project.

3.1 Ensuring Application Reliability

The first topic we will touch on is the issue of reliability or availability. We define this to mean that the application is available for use by an end user, providing the current developed level of functionality. Essentially, this section describes the techniques used to keep the tool up and running. The importance of this topic is clear, since service outages could endanger the continuation of this kind of project.

As with most software development projects, we start by logging and analyzing errors and warnings and other debugging activities. We quickly learned that it was all too easy to crash our application with minor Javascript programming errors. Initial approaches included creating and logging known exceptions as shown in Figure 2. Next, we utilized the cluster [6] module using cluster.on('exit', …) to fork() a new thread to immediately replace each thread that died/crashed. This gave us the additional advantage of utilizing multiple threads of execution to assist with performance scaling with our multiple CPU server. Additionally, we used the forever [7] tool to ensure that the entire application is restarted in the event of a high level application crash. We wrote additional monitoring scripts which run periodically as Unix cron jobs to restart any processes or services which are not running. This technique is useful for recovering from unexpected system crashes or reboots. Each approach also generates log output for future debugging and problem determination.

We also found it useful to create an off-system heartbeat monitoring application which verifies that end users can access the tool over the network. Failures here are considered serious enough to immediately send diagnostic email messages to team members for analysis and resolution. This has enabled us to quickly identify issues with external dependencies including

```
process.on('ECONNRESET', function (err) {
  console.error("app.js:" + (new
Date()).toUTCString() + " otherwise uncaught
connection reset exception");
  console.error("app.js: err = " +
JSON.stringify(err,null,4));
});
```

Figure 2. Sample Execution Trapping

network routing, name resolution, and authentication service availability. It has also enabled the team to quickly resolve environmental problems with the server that prevent the application from running, for example, disk full conditions, or lack of other resources that prevent the application from fully running.

The concept of software rejuvenation [8] also has a place in our repertoire, as we also restart the application periodically during times of low use to reduce certain classes of errors.

3.2 Enabling Painless Serviceability and maintainability

As the project progresses, bugs are identified and fixed, and new features are created, it is necessary to update and service an application. Our team uses two primary approaches to enable painless serviceability and maintainability. Our first is to use a documented development process that embraces the git branching philosophy, and our second is to use extensive automation and monitoring in our development and infrastructure.

Although our team is not huge, we feel that having a documented development process helps keep processes and output consistent. This is also useful when onboarding new team members. We follow the git branching model eloquently described by Driessen [9]. This involves separate source branches for development (development branch) and production (master branch), the use of function branches to update development release branches, and the use of hotfix branches to update staging and production branches. Our development process also covers issues including source control, coding conventions, work item/project management, unit testing, and standards compliance.

Regarding the second approach, we actively utilize automation for steps that must be reliably repeated. For example, we use scripts which automatically update and deploy our application to team servers. This is used to ensure that each of the servers that our team uses is running a known current version of the application. Specifically, we automatically update and deploy our development branch to our development server every 60 minutes, similarly, we update the application running on our staging server every 60 minutes with the staging branch. This enables developers to observe their code on a controlled reference platform in a timely fashion. And since we use automated error and availability monitoring, we can rapidly identify problems as soon as code is deployed to one of our servers. We are more conservative with our User Acceptance Testing (UAT) and production servers. These servers must be more stable than a development environment, so we limit our modifications to these branches, and we update the servers less frequently. However, update, deployment, and monitoring on these servers is also fully automated with the same tools we use in development.

3.3 Verification of Application Functionality

The ability to easily verify the correct functionality of software is critical to maintain the trustworthiness of that software. This problem becomes increasingly serious as teams grow, members are replaced, and is even more challenging when members are located in multiple locations.

```
describe("A suite", function() {
  it("contains spec with an expectation",
function() {
    expect(true).toBe(true);
  });
});
```

Figure 3. Sample Jasmine spec from Jasmine Documentation [10]

Initially our developers perform ad-hoc and/or manual unit testing on the functionality that they add to our source control system. Next while creating new functionality developers also write Jasmine unit test cases. The Jasmine test framework enables straight forward Javascript code that can test for expected conditions. Figure 3 demonstrates how a simple test case can be written. These Jasmine test cases then form the foundation of a test suite that can be run by developers before they deliver modified source code, and can also be run as an automated process to identify potential problems immediately and without manual intervention.

Our development process requires separate functional verification at certain points. Namely, before we release a new version we prepare a User Acceptance Test (UAT) server with our release candidate code. The staging server hardware and software is configured to be as similar as possible to the production server. The UAT server is then used by the UAT team to ensure that all required functionality is included, and that in fact the application behaves as it should. The next functional verification point is when we update our production server with hot fixes. Before modifying the production server with non-functional but important bug fixes, we make our changes to the staging branch, which is then deployed to the staging server. As with the UAT server, the staging server hardware and software is configured to be as similar as possible to the production server. When the hotfix has been applied to the staging server, both standard functionality and the absence of the bug are verified before allowing the hotfix to be applied to the production server via the master branch.

As mentioned in a previous section, we also run tools which frequently monitor the end-user accessibility of our application on each of the servers it runs on, including development, demonstration, staging, UAT, and production. We also have automation that notifies our team when error and log files are found to contain certain “bad” patterns such as “thread died” that requires human intervention or problem determination.

3.4 Coordination of Development

As hinted at in earlier sections, there are obvious coordination issues involving a growing, changing team with members in multiple time zones throughout the world. Our development process is not explicitly either “agile” or “extreme”, however, our activities are certainly intense as demonstrated by a high level of user and sponsor expectation, a short release cycle, and frequent detailed coordination for development and planning, (including close collaboration regarding business requirements). The techniques, processes, and tools we have found useful fall into four categories: direct communication, task/project management process/tools, artifact management tools, and shared environments.

Direct communication may seem the most obvious. Standing all-hands telephone conference calls are both unavoidable and necessary, though we have found that smaller 2-3 person calls focused on a single topic are quite productive and efficient. Screen sharing can make these working sessions almost as effective as a face-to-face meeting with a whiteboard and a projector. It goes without saying that email and instant messaging are also irreplaceable tools. Even though our team has members residing in six countries we have found occasional face-to-face meetings both productive and invaluable for laying the groundwork for intense remote coordination afterwards.

Once a project grows to a certain size, some form of task management tool becomes irreplaceable for both project management purposes and also to track and document deliverables including bug fixes, new function development, and progress towards larger goals such as release delivery. Our team has found the flexibility of Rational Team Concert^{TM} to be powerful and flexible for these purposes.

Artifact management is a broad and vague concept. Initially, a source management system is invaluable for storing source code, maintaining multiple versions, understanding historical changes, and sharing a common code base amongst team members. It is hard to conceive of any software development project with more than several people without a source control system. Our team has found a community version of GitLab to suite our purposes well. Other artifact management systems include shared wiki type collaboration systems for sharing and collaborating on text type documents, and shared file storage systems for sharing larger binary files such as spreadsheets and presentation files. Our team has found all of these tools useful.

Finally, shared computer environments are similar to speaking the same language. Although we have an official reference platform, since our application is built on open software components, it can run on nearly any environment that can support Node.js. Thus we find it necessary to maintain shared reference platforms that team members can use to verify functionality and compatibility.

3.5 Enablement of Usage Data Reporting

As we have developed an application for business which is used by hundreds of people, it is clearly necessary to provide consistent management reports and insights on the data stored in this system. Initially, we gathered and emailed simple daily usage reports through an automated reporting program that directly accessed the backend database. However, as the amount of data grew, and our requests for analysis became more complicated and frequent, we have experimented with various tools to access and run queries on MongoDB databases, as well as writing numerous ad-hoc MongoDB queries in Node.js Javascript to answer specific questions. As of this point, we are developing a more flexible framework for predefined and custom reporting of important and consistent queries, which will be accessible by a defined set of administrative users. Additionally from a more technical point of view we have several tools which create email notifications and reports when abnormal conditions such as crashed threads or inaccessible services are detected.

4. Conclusions

This project is still ongoing and expanding. As the number of users we have continue to grow, new solutions will need to be found for certain areas. However, we are confident that the various approaches we use to accomplish the following are common to many development projects: (a) ensure an available web application through rejuvenation, redundancy, monitoring, and automation, (b) manage serviceability and maintainability through the use of a common development process, branching philosophy, and extensive automation, (c) verify correct application functionality through unit testing, automated testing, staging and UAT development processes, and automated monitoring, (d) coordinate development through shared source repository, work flow system, communication tools, and various shared artifacts and servers, and (e) create and distribute consistent timely usage reports. It is our hope that these tools and techniques can be considered best practices and benefit other development teams.

5. Bibliography

[1] T. Kushida and S. Trent, "The Provision of Cloud Solution Design Service," in *Information Processing Society of Japan*, Tokyo, Japan, 2015.

[2] T. Kushida, "Cloud Solutioning," IPSJ SIG-DPS, Tokyo, Japan, 2015.

[3] T. Kushida and T. Mishina, "Cloud Solutioning - Self-service infrastructure design," IPSJ SIG-DPS technical Report (March), Tokyo, Japan, 2015.

[4] H. R. Motahari-Nezhad and et. al., "COOL: A Model-Driven and Automated System for Guided and Verifiable Cloud Solution Design," in *Submitted to ICSOC*, Banff, Alberta, 2016.

[5] Node.js Foundation, "About Node.js(r)," [Online]. Available: https://nodejs.org/en/about/. [Accessed July 2016].

[6] Node.js Foundation, "Cluster Node.js Manual & Documentation," [Online]. Available: https://nodejs.org/api/cluster.html. [Accessed July 2016].

[7] C. Robbins, "foreverjs/forever: A simple CLI tool for ensuring that a given scrip runs continuously," [Online]. Available: https://github.com/foreverjs/forever. [Accessed July 2016].

[8] R. Hanmer, "Software rejuvination," in *Proceedings of the 17th Conference on Pattern Languages of Programs*, Reno, Nevada, 2010.

[9] V. Driessen, "A successful Git branching model," [Online]. Available: http://nvie.com/posts/a-successful-git-branching-model/. [Accessed July 2016].

[10] Pivotal Labs, "Jasmine: Introduction," [Online]. Available: http://jasmine.github.io/2.0/introduction.html. [Accessed July 2016].