A Software Deployment Risk Assessment Heuristic for Use in a Rapidly-Changing Business-to-Consumer Web Environment

John Comas, Dr. Ali Mostashari, Dr. Mo Mansouri, Dr. Richard Turner

Stevens Institute of Technology, Babbio Center,
Castle Point on Hudson, Hoboken, NJ 07030

jcomas@stevens.edu, Ali.Mostashari@stevens.edu, Mo.Mansouri@stevens.edu,
Richard.Turner@stevens.edu

Abstract

In the 24/7 web industry, business owners can feel a constant need to push the latest and greatest software to the live site in order to remain competitive and ensure growth. However, the bottom line to the business must always be that the site is fully available and functional to consumers. The business cannot make money if a customer is greeted with a maintenance page. Also, the business reputation of the company can be severely damaged if the web site proves to be unavailable to customers or if orders aren't received due to improper fulfillment. The constant rapid updating and changing of software can cause severe site instability to the point where the business can fail. Systems integration engineers are often the last line of defense in preventing this type of catastrophic failure. Due to the need for rapid software integration, a heuristic for quantitatively measuring risk in the system is invaluable to determine what software is riskiest to deploy and where problems are most likely to occur. The software deployment risk assessment heuristic presented in this paper allows systems integration engineers to determine where these risks are most apparent in the system and to explore where best to focus their efforts for ensuring the least disruptive deployment method for live production systems.

Keywords: *Deployment, Risk, Heuristic, Web, Systems Integration, Assessment, Production, E-Commerce, Qualitative, Tier, Components, Code, Software.*

Acronyms: *Systems Integration (SI), Quality Assurance (QA), Information Technology (IT), Development Environment (DEV), Production Environment (PROD).*

1. Introduction

In the internet business, timing is everything. Because the internet business world is so fast-paced, it is imperative for every company to stay on pace with their respective competitors. Each company tries to eclipse its competitor by creating a web site which is more dynamic and more enticing to consumers. "That need to evolve is causing a flurry of activity among the big Internet commerce competitors, as they cross into each others' territories to add services to woo more consumers." [19] This creates a situation where a software infrastructure update to the web site of an internet company may become obsolete the day after it is introduced to the public. A company that releases a brilliant new technology on their web site cannot sit back for a moment; they must immediately introduce a new strategy to keep up their competitive advantage. A company that sits on its laurels may soon find themselves out of business.

Because the nature of the e-commerce business is a 24/7 industry, any site downtime will cause a loss of business to the company. The ultimate goal of course is to be able to deploy the new software into the live production environment quickly and with zero impact to customers. This deployment process must be executed by integration engineers based on information ascertained by the development organization. However, the efforts of systems integration engineers can be seriously handicapped due to the lack of proper documentation. At times, software can be handed over to integration engineers who have little familiarity with regard to its internal intricacies. Vital knowledge of the dependencies between various software components as well as the least disruptive method of deploying the various components to live production systems can be non-existent. "A sound system cannot be established in an environment of constant and unbound transitions." [20] Thus, the risk a particular deployment presents to the overall system is often unknown. Because systems integration engineers are often the last line of defense in finding issues that may lead to production problems, a method of quantitatively measuring the risk to the system is invaluable. The level of risk to the system that the deployment represents affects the level of attention and careful planning which must be performed by the integration team. Unsuccessful deployments to the live-site and loss of revenue due to undiscovered software bugs are all too often the result of a risky deployment believed to be benign.

The problem is that there are no easy methods for quantitatively ascertaining these vital software component dependencies. Since the web site cannot be taken down for the purposes of software upgrades, it is extremely important to know how new components will interact with current components on the live site. New software cannot be instantaneously deployed to all involved server classes throughout the company's enterprise. Because the systems are live, any and all upgrades must be seamless to the user who is conducting business with the company.

The problem to be addressed in this paper is how to determine where the risks are most apparent both to the consumer and to the web site when deploying new software into production and how to effectively determine possible problems that may be incurred during the various transitional states in which the web site will be during the production deployment process. This paper comprises a full literature review of the topic as well as outlining the motivation for its creation and the methodologies utilized. The heuristic is fully analyzed and illustrated by means of a comprehensive example. The results of the heuristic are then finally incorporated into the production system software deployment methodology in order to determine and achieve the lowest risk and most successful production deployment possible.

2. Literature Review

Current research in release management focuses on the importance of release management and the tools necessary for achieving successful release management. As Kajko-Mattsson and Meyer state:

"To make changes to software systems is one thing, to release them in a disciplined and controlled way is another. The changes, if released in an uncontrolled way, may lead to lack of control over the delivered changes and to deteriorated system quality. In today's fast changing business environments, they may strongly affect the survival of many software organizations. Hence, one of the major concerns of any organization should be to have an orderly well thought-out end-to-end release management process. Such a process should help deliver software releases with maximal functionality and quality within the boundaries of the resource constraints." [18]

While the methodologies for release management as well as the various types of testing such as End-to-End integration testing are discussed, there is an obvious gap in current research as to how to ascertain the risks through the systems integration release process involved with deploying new software to a live site without causing a single disruption to a user. Also, current research on methods of successful deployments to live production 24/7 web-based systems is scarcely touched upon. Another serious gap in reviewing the current literature is how to deploy multiple projects into a single systems integration environment and still achieve correct testing results. Rana and Arfi [1] address at length the delta versioning model, the delta effect, and touch on the relationship between different software modules. However, they fall short of explaining how these modules will interact with existing software during the transitional period while the software is being deployed. Fanberg [2] describes the importance of software comparison or 'diffing' tools in verifying compiled software modules to improve the software release management process. While 'diffing' achieves the necessary validation required to assure that the correct files were deployed to production, it only determines that the files themselves are correct and not corrupted. Paul [3] discusses the importance of testing to ensure software quality and how effective end-to-end (E2E) integration testing can ensure proper risk analysis. He uses a thin thread principle to assign a risk to each thin thread and then rank them based on the consequence of failure. For example, threads with a high risk of failure but do not cause catastrophic systems issues may not require the intensity of testing as a thread with a medium risk of failure but could cause catastrophic systems issues. Van Der Hoek, et al. [5, 10, 11, 12] performed extensive research into release management tools and the need for proper documentation. They describe the necessary requirements for software release management tools and how the internet has influenced software development processes. Hong and Kapsu [13] discuss a Software Reuse System which profiles software components in order to readily classify and retrieve them. This classification method can assist the integration engineer in categorizing software components for use in the heuristic.

For the deployment aspect, Hall, et al. [14] discuss at length the idea of a software dock framework so as to employ a standard deployment framework that allows for deployment modeling, automated deployment, and decentralized control. Ganguly, et al. [15] discuss how to reduce the complexity of software deployment by use of the delta configuration. A method of easy deployment to the production environment was developed where pre-configured software images were utilized. Belguidoum and Dagnat [17] discuss the necessity of developing a unified model for production deployment and the importance of identifying the interdependencies between various components. Dependencies are formalized with logical formulas and deployment architectures are created. However, the research gap of how the risks of these various software components will interact with existing software during the production deployment is not addressed in any of the above studies.

3. Motivation

The motivation for this paper is to provide a heuristic that allows systems integration engineers to determine where the highest risk of a deployment related problem in the system is present. This heuristic allows SI engineers to harness their efforts to produce a seamless and non-disruptive production deployment. The heuristic also provides data to determine where to most effectively focus the testing of the various transitional deployment states to mitigate the risk of site downtime. The heuristic can also provide a quantifiable and comparable figure which SI engineers can provide to a particular business unit. For example, with the use of this heuristic, the IT department can confidently state to a business owner that

the risk of deploying this particular software to this particular tier is twice as risky as a similar deployment to another tier or even to the same tier in another system. The business owner can determine that the risk is acceptable and the deployment should proceed or that the risk is too great and the deployment should be indefinitely halted.

Currently, one of the most effective ways to ensure a smooth production deployment is to engage in rigorous release management. Proper release management is a very important aspect of any well-structured professional IT organization. Without proper release management, the entire IT infrastructure may break down as a result of incorrect or unauthorized code being released. [18] Any software written by the IT department's development organization must be first packaged into a particular release, assigned a tag number, and then sent off for QA testing. QA testing must occur in a controlled lab environment whereby the software is tested using a test system which matches the live production system as closely as possible. The QA engineers must test their software in an environment which is extremely accurate. Thus, any software placed into this testing environment must also be properly configured to the required specifications and any necessary dependencies must also be delineated.

One of the drawbacks of a pure release management approach is that the risk of introducing the release into production is not quantified. Risk management plays a very important role in determining what software should be deployed into the live production environment at any given time. In a rapidly-changing web environment, SI engineers may not be able to effectively determine where the greatest risk of where a deployment issue may lie. The e-commerce web site must be available 24/7 even during the production deployment process, and this deployment process must in no way impact the customer. The same quality experience should always be provided to the customer, and the customer must be able to access all features and nuances of the web site at all times. The visual experience to the customer must be consistent and not vacillate between different states. Thus, the software deployment risk scoring heuristic presented in this paper provides the systems integration engineers with a tool to accomplish the aforementioned goals.

4. Methodology

This paper utilizes a qualitative risk-assessment approach and introduces the concept of the Software Deployment Risk Scoring Heuristic. The purpose of this heuristic is to provide systems integration engineers with a working model to determine where the potential problems, complications, etc. are most likely and prevalent for a particular software deployment. While it is beyond the scope of this paper to quantify through data mining the precise risk percentage of site downtime during the deployment process, the goal is to provide a discursive and detailed description of a typical production deployment and interpret the results in a manner which is useful to the participants. In order to accomplish this, a qualitative framework and model must be created using a grassroots approach for the system. The system on which the deployment will be performed is broken down into specific functionality tiers. Each individual tier must then be decomposed into the specific components that will be undergoing modification. The software change must be scrutinized on a line-by-line basis to identify all changes and the requirements necessary to implement these changes. An analysis of each of the modifications will provide the systems integration engineers with a risk guideline to determine where to best focus transitional state testing and identify where problems are most likely to occur.

A typical business-to-consumer web environment may consist of hundreds of computers from the frontend to the back end and have complex non-linear dataflows spanning across the

entire system. The entire web site can be viewed as a system of systems all working together in lockstep. One subsystem cannot be taken down without an impact upon other subsystems. In order to effectively measure the risk a particular deployment may bring to the entire web site architecture, an analysis must be conducted of the individual subsystems within the system to be modified and their interaction with other subsystems. Once all of the individual servers in the system to undergo deployment have been identified, their respective dataflows must be delineated. The goal of this analysis is to create a simple vertical representation of the subsystem upon which the deployment will occur.

The subsystem to undergo deployment is divided into functional tiers and drawn vertically in a conceptual model ordered by way of functional intercommunication. In this model, the backend tier is at the bottom and the customer-facing tier is at the top. Middleware tiers will obviously be located between the backend and customer-facing tiers. A simple example of a three tier system is shown in Figure 1.

Figure 1: Subsystem Conceptual Model

In developing this heuristic, it was assumed that the system which will undergo deployment is live and all functionality will need to be fully available throughout the production deployment process. Further, the deployment process on a particular tier will not create a situation where other tiers above or below that tier will cease to function. Taking this into consideration, a five-question interrogative assessment is performed on each individual tier in the system conceptual in order to begin analyzing the risk in the various transitional states during the deployment process. The purpose of this interrogative assessment is to create an initial risk interrogative factor (R_{IF}) and determine the relative software delta penetration depth for the high-level tiers in the system conceptual model. The five-question interrogative analysis is shown in Table 1:

Table 1: Tier-Level Interrogative Analysis

Software Deployment Risk Scoring Heuristic						
Tier-Level Interrogative Analysis						
Risk Factor Interrogatives	Tier I		Tier II		Tier III	
	Yes or No	Risk Score	Yes or No	Risk Score	Yes or No	Risk Score
1.) Are there changes to existing configuration mappings?	Yes	1	No	0	Yes	1
2.) Are there new communication/configuration paths or mappings?	No	0	No	0	Yes	1
3.) Are there functionality additions or modifications?	No	0	No	0	No	0
4.) Are there changes visible to customers on the front-end?	Yes	1	Yes	1	No	0
5.) Are new system interdependencies introduced?	Yes	1	Yes	1	No	0
Risk Interrogative Factor: (R_{IF})		3		2		2

It is necessary to conduct a preliminary analysis of the high level changes occurring during system deployment in order to determine how many points of failure or risk exist in this particular system deployment. Through experience, the 5 risk factor interrogatives listed in table 1 are the most common failure prone aspects of the overall deployment. All of the factors are weighted equally as the influence of each factor has similar impact on the overall risk analysis.

Anytime a configuration change is made to a system, there exists the potential for a misconfiguration. Examples of misconfigurations could be as simple as a typographical error in a file, an incorrect database datasource name, username, and/or password, or deploying software with debug logging active.

Functional changes to the system can introduce severe risk during the deployment. For example, let's make the assumption that a database stored procedure and an XML front-end visual change is being deployed to production. When QA performed its functional validation, the SP and the XML change together tested perfectly. However, QA did not validate the intermediate deployment scenario where the stored procedure has been updated in production, but the existing XML is still present awaiting upgrade. The stored procedure proved not to be backward compatible with the existing production software and thus a system failure occurs.

All of these risks may not be present in every deployment conducted. For this reason, it is necessary to distinguish and outline the high level changes occurring in the deployment. Therefore, the initial risk interrogative factor was derived to create a high-level comparison basis for deployments. The risks are additive and therefore multiple 'Yes' answers indicate an increased deployment risk. For example, a deployment with no changes to configurations, mappings, functionality, or interdependencies is at the high level not high risk. Conversely, a deployment with configuration and mapping changes with new interdependencies is much more risk prone.

The risk interrogative factor also serves as a beacon to amplify potential impacts to the business which may occur during the deployment. A customer facing change is always inherently more risky than a non-customer facing change because it affects the customer's ability to interface and interact with the web site. Any loss of ability for the customer to properly or conveniently interact with the web site can be devastating to the business.

The total risk interrogative factor through the interrogative analysis will always fall between $0 \leq R_{IF} \leq 5$ where 0 represents the lowest risk possible and 5 represents the highest risk possible. An answer of yes will give a score of +1 and an answer of no will add nothing. If the answers to all questions were no for instance, the tier would have a risk score of 0 and

would require no transitional deployment testing as there is little risk of an issue occurring during the deployment process. If however, the answer to questions 1 through 4 is yes and the answer to question 5 is no, the risk score will be +4 for that particular tier. A score of 4 would indicate a high degree of probability that a risk of site instability during the deployment process is present.

Upon completion of the initial tier-level interrogative assessment, an intense and thorough analysis must be executed on the individual software components which comprise each tier. This per component scrutinization allows the integration engineer to specifically delineate the state of each individual component and categorize it to conform to one of the six predetermined categories shown in Table 2 below:

Table 2: Component-Level Categorized Analysis

Software Deployment Risk Scoring Heuristic	
Component-Level Categorized Analysis	
Individual Software Component Risk Categorization	**Risk Score per Component (R_C)**
A.) No change to existing code and no additional new code	1
B.) No change to existing code with additional new code	2
C.) Change to existing code and no additional code	3
D.) Change to existing code with additional new code	4
E.) Complete replacement of existing code with no additional code	5
F.) Complete replacement of exiting code with additional new code	6

Through this analysis, a plethora of risk scoring data becomes available for input into the heuristic. In the above table, the component categorizations are listed in order of increasing risk so as to show that R_C=1 has no risk quotient while R_C=6 has the highest risk quotient.

Since software components do not exist in a vacuum, it can be reasonably construed that changing one component will have an affect on another component. An example of this type of dependency is the principal of circular dependency. [7] As a result, individual components may have varying risk factors depending upon their particular functionality and how they interact with other components. It is thus important to break down an individual tier into sublevels which can categorically delineate how interdependent a particular software component may be and how vast the scale of the impact on other components may be. Each tier can be subdivided into 5 primary sublevels (R_{SL}) where changes are likely to occur as shown in Figure 2 below.

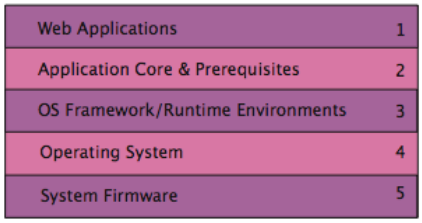

Figure 2: Intra-Tier Sublevel Analysis Model

The Web Applications Sublevel represents the internet software components written by developers for the company that carry out the main tasks the particular tier was designed to perform. The Application Sublevel is supported by the Application Core/Prerequisite

Sublevel which represents the software components that underpin the main applications running in that tier. The Operating System Framework/Runtime Environments Sublevel consists of those extensions, virtual machines, and additional software class libraries to the basic operating system (e.g. J2EE, JRE, Microsoft .NET, etc.) which provide support to the two application and application core sublevels. The Operating System Sublevel represents the software for the full blown basic operating system for the tier (e.g. Linux, Windows 2003 Server, etc). Finally, the System Firmware Sublevel consists of the software components which make up the BIOS or firmware of the device including a low-level machine code components.

Each of the aforementioned levels is dependent upon the correct functionality of all of the corresponding levels below it. For example, the operating system framework/runtime environments sublevel is dependent upon both the operating system sublevel and the firmware sublevel. As a result, it can be construed there is a very high risk that a software change to the system hardware firmware level will greatly impact all of the four other levels above it in that particular tier. Conversely, there is little to no risk that a software component change to the application level will result in a failure of the components in the system hardware firmware level.

As illustrated below in Table 3, each of the sublevels is assigned a risk score $1 \leq R_{SL} \leq 5$ where R_{SL} =1 represents the lowest sublevel interaction/dependency risk and R_{SL} =5 represents the highest sublevel interaction/dependency risk. At R_{SL} =5, all of the intra-tier sublevels are affected by the change and the risk of a deployment related problem is high.

Table 3: Intra-Tier Sublevel Risk Scoring Table

Software Deployment Risk Scoring Heuristic	
Intra-Tier Sublevel Analysis	
Tier Sublevel	**Sublevel Risk Score (R_{SL})**
A.) Web Applications Sublevel	1
B.) Application Core & Prerequisites Sublevel	2
C.) Operating System Framework/Runtime Environments Sublevel	3
D.) Operating System	4
E.) System Hardware Firmware Sublevel	5

In the sublevel component risk categorization, the total number of components in a particular sublevel (N_{CSL}) must be taken into consideration. Formula 1.1 calculates the risk score for the categorized components in the sublevel (T_{RSC}) based upon the relationship of the individually categorized components (C_{SL}) and their respective risk factors (R_{SL}).

$$\frac{\sum C_{SL} R_C}{N_{CSL}} = T_{RSC} \qquad \text{Formula 1.1}$$

The above formula will produce an individualized result for each of the six categories in the sublevel. Now that the risk score is known for each category within the sublevel, a relationship must be developed between the categorized component score and the intra-tier sublevel risk categorization. As shown in Formulas 1.2 & 1.3, the total risk for the sublevel (T_{RSL}) is calculated by multiplying the total categorized risk (T_{RSC}) by the sublevel risk score (R_{SL}).

(T_{RSL}) is calculated by multiplying the total categorized risk (T_{RSC}) by the sublevel risk score (R_{SL}) determined through the intra-tier sublevel component risk categorization analysis.

$$T_{RSC}R_{SL} = T_{RSL} \qquad \text{Formula 1.2}$$

$$\frac{\sum C_{SL}R_C}{N_{CSL}} R_{SL} = T_{RSL} \qquad \text{Formula 1.3}$$

Now that the total risk for each of the five sublevels is known, the total tier component risk (T_{RCT}) must be determined. This figure may be calculated by Formula 1.4 as shown below.

$$\sum T_{RSL} = T_{RCT} \qquad \text{Formula 1.4}$$

In order to calculate the total tier risk (T_{RT}), the total tier component risk (T_{RCT}) must be multiplied by the tier risk interrogative factor (R_{IF}) as shown is Formula 1.5.

$$T_{RCT}R_{IF} = T_{RT} \qquad \text{Formula 1.5}$$

Upon finalization of the aforementioned analyses, the total average risk for overall system conceptual model (S_{AR}) may be determined by dividing summation of the total individual tier risks (T_{RT}) by the total number of tiers in the conceptual system model (n_T) as shown in Formula 1.6.

$$\frac{\sum_{1}^{n_T} T_{RT}}{n_T} = S_{AR} \qquad \text{Formula 1.6}$$

In order to demonstrate the software development risk scoring heuristic, a hypothetical analysis has been performed on the system model shown previously in Figure 1. For the sake of this example, the responses to the risk factor interrogatives have been randomly selected. Also, in order to complete the individual software component risk categorization, Tier I has been determined to have 2300 individual software components, Tier II has 1550 components, and Tier III has 1000 components. Tables 4 - 5 and Tables 6 - 11 demonstrate the heuristic at work:

Table 4: Initial Risk Factor Interrogatives

Software Deployment Risk Scoring Heuristic						
Risk Factor Interrogatives	Tier I		Tier II		Tier III	
	Yes or No	Risk Score	Yes or No	Risk Score	Yes or No	Risk Score
1.) Are there changes to existing configuration mappings?	Yes	1	No	0	Yes	1
2.) Are there new communication/configuration paths or mappings?	Yes	1	No	0	No	0
3.) Are there functionality additions or modifications?	No	0	No	0	Yes	1
4.) Are there changes visible to customers on the front-end?	No	0	Yes	1	Yes	1
5.) Are new system interdependencies introduced?	No	0	Yes	1	Yes	1
Risk Interrogative Factor (R_{IF})		2		2		4
Total Number of Components in Tier (C_T)	1000		1550		2300	

Table 5: Intra-Tier Sublevel Component Risk Categorization

Intra-Tier Sublevel Component Risk Categorization	Number of Components in Category (N_{CSL})	Number of Components in Category (N_{CSL})	Number of Components in Category (N_{CSL})
Web Application Sublevel (x1)	450	650	750
Application Core/ Prerequisites Sublevel (x2)	125	200	100
OS Framework/Runtime Environment Sublevel (x3)	175	400	100
Operating System Sublevel (x4)	150	200	100
Firmware Sublevel (x5)	100	100	1200

Table 6: Intra-Tier Web Application Sublevel Component Risk Categorization

Table 6 **Web Application Sublevel Component Risk Categorization (R_{SL})**	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)
Number of Components in Category (N_{CSL})	450		650		750	
A.) No change to existing code and no additional new code (x1)	60	*60*	100	*100*	400	*400*
B.) No change to existing code with additional new code (x2)	10	*20*	150	*300*	200	*400*
C.) Change to existing code and no additional code (x3)	30	*90*	100	*300*	50	*150*
D.) Change to existing code with added new code (x4)	125	*500*	100	*400*	50	*200*
E.) Complete replacement of existing code with no additional code (x5)	100	*500*	80	*400*	0	*0*
F.) Complete replacement of exiting code with additional new code (x6)	125	*750*	120	*720*	50	*300*
Total Risk for Sublevel Category (T_{RSC})		**4.27**		**3.42**		**1.93**

Table 7: Intra-Tier App Core/Prerequisite Sublevel Component Risk Categorization

Table 7 **App Core/ Prereq Sublevel Component Risk Categorization (R_{SL})**	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)
Number of Components in Category (N_{CSL})	125		200		100	
A.) No change to existing code and no additional new code (x1)	10	*10*	20	*20*	0	*0*
B.) No change to existing code with additional new code (x2)	25	*50*	30	*60*	10	*20*
C.) Change to existing code and no additional code (x3)	35	*105*	45	*135*	5	*15*
D.) Change to existing code with added new code (x4)	20	*80*	50	*200*	5	*20*
E.) Complete replacement of existing code with no additional code (x5)	10	*50*	25	*125*	70	*350*
F.) Complete replacement of exiting code with additional new code (x6)	25	*150*	30	*180*	10	*60*
Total Risk for Sublevel Category (T_{RSC})		**3.56**		**3.6**		**4.65**

Table 8: Intra-Tier OS Framework/Runtime Sublevel Component Risk Categorization

Table 8 **OS Frmwrk / Runtime Env Sublevel Component Risk Categorization (R_{SL})**	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)
Number of Components in Category (N_{CSL})	175		400		100	
A.) No change to existing code and no additional new code (x1)	5	*5*	375	*375*	90	*90*
B.) No change to existing code with additional new code (x2)	10	*20*	0	*0*	0	*0*
C.) Change to existing code and no additional code (x3)	5	*15*	0	*0*	0	*0*
D.) Change to existing code with added new code (x4)	5	*20*	0	*0*	0	*0*
E.) Complete replacement of existing code with no additional code (x5)	100	*500*	0	*0*	10	*50*
F.) Complete replacement of exiting code with additional new code (x6)	50	*300*	25	*150*	0	*0*
Total Risk for Sublevel Category (T_{RSC})		**4.91**		**1.31**		**1.4**

Table 9: Intra-Tier OS Sublevel Component Risk Categorization

Table 9 **OS Sublevel Component Risk Categorization (R_{SL})**	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)
Number of Components in Category (N_{CSL})	150		200		100	
A.) No change to existing code and no additional new code (x1)	100	*100*	200	*200*	0	*0*
B.) No change to existing code with additional new code (x2)	0	*0*	0	*0*	0	*0*
C.) Change to existing code and no additional code (x3)	0	*0*	0	*0*	0	*0*
D.) Change to existing code with added new code (x4)	0	*0*	0	*0*	50	*200*
E.) Complete replacement of existing code with no additional code (x5)	0	*0*	0	*0*	50	*250*
F.) Complete replacement of exiting code with additional new code (x6)	50	*300*	0	*0*	0	*0*
Total Risk for Sublevel Category (T_{RSC})		**2.67**		**1.0**		**4.5**

Table 10: Intra-Tier System Firmware Sublevel Component Risk Categorization

Table 10 **System Firmware Sublevel Component Risk Categorization (R_{SL})**	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)	Number of Components in Sublevel (C_{SL})	Risk Score ($C_{SL}R_C$)
Number of Components in Category (N_{CSL})	100		100		1200	
A.) No change to existing code and no additional new code (x1)	90	*90*	90	*90*	0	*0*
B.) No change to existing code with additional new code (x2)	0	*0*	0	*0*	0	*0*
C.) Change to existing code and no additional code (x3)	0	*0*	0	*0*	0	*0*
D.) Change to existing code with added new code (x4)	10	*40*	10	*40*	0	*0*
E.) Complete replacement of existing code with no additional code (x5)	0	*0*	0	*0*	0	*0*
F.) Complete replacement of exiting code with additional new code (x6)	0	*0*	0	*0*	1200	*7200*
Total Risk for Sublevel Category (T_{RSC})		**1.3**		**1.3**		**6.0**

Table 11: Intra-Tier Total Sublevel Component Risk Categorization

***Table 11* Intra-Tier Sublevel Component Risk Categorization**	Total Risk per Sublevel Categorization (T_{RSC})	Total Risk Sublvl (T_{RSL})	Total Risk per Sublevel Categorization (T_{RSC})	Total Risk Sublvl (T_{RSL})	Total Risk per Sublevel Categorization (T_{RSC})	Total Risk Sublvl (T_{RSL})
Web Application Sublevel (x1)	4.27	*4.27*	3.42	*3.42*	1.93	*1.93*
Application Core/ Prerequisites Sublevel (x2)	3.56	*7.12*	3.6	*7.2*	4.65	*9.3*
OS Framework/Runtime Environment Sublevel (x3)	4.91	*14.73*	1.31	*3.93*	1.4	*4.2*
Operating System Sublevel (x4)	2.67	*10.68*	1.0	*4.0*	4.5	*18*
Firmware Sublevel (x5)	1.3	*6.5*	1.3	*6.5*	1.3	*6.5*
Total Risk for Tier Components (T_{RCT})		***43.3***		***25.05***		***39.93***
Risk Interrogative Factor (R_{IF})		2		2		4
Total Tier Risk (T_{RT})		**86.6**		**50.1**		**159.72**
Total Average System Risk (S_{AR})	**98.81**					

From the heuristic model, it was determined that Tier III (Web Tier as per Figure 1) has the highest total component risk with T_{RT}=159.72 in relation to the other two tiers analyzed. It is thus clear from the heuristic model that Tier III (Web Tier as per Figure 1) has the highest risk of causing a site problem during the deployment process than Tier I or Tier II. The integration engineer can now utilize this information to focus efforts on ensuring the most appropriate deployment approach for this particular tier.

5. The Path to Production

One of the main responsibilities of a systems integration engineer is to systematically coalesce and desegregate an aggregate of subsystems into a working solution for the customer. In an e-commerce company, this is an essential function for ensuring the stability of the production web site. Among the many tasks of systems integration engineers, they must determine how best to host the software, determine the proper types of servers and OS platforms required, and load test the software to establish its resilience. With the need for 24/7 uninterrupted availability of the live production web site, SI engineers must determine all risks that may be present during the production deployment process and how to ensure a transparent deployment to the customer. Armed with the results of the software deployment risk scoring heuristic, SI engineers can carry out this task more effectively and with better certainty of success. In order to demonstrate the power and usefulness of the results obtained from the software deployment risk scoring heuristic, the scores calculated previously were applied to an example web environment that will undergo a deployment. The sample environment to be described represents a standardized and ideal application for which the heuristic would prove highly valuable. While the environment discussed does not represent a particular web environment, it combines elements typical of a web development environment and incorporates the obstacles common in many web-based IT organizations.

The path to production is extremely essential in assuring a smooth deployment, and Figure 3 below illustrates an example of a methodical approach to software testing and analysis.

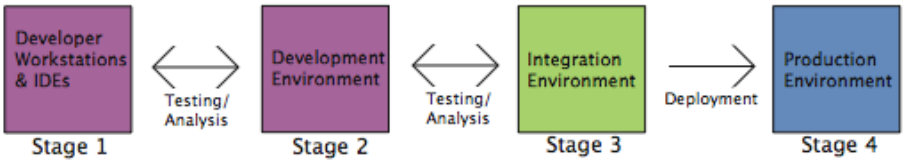

Figure 3: Software Testing & Analysis Path to Production

In Stage 1 of Figure 3, developers code on their own desktops with Integrated Development Environments (IDEs), write software components independently, and perform preliminary tests on those components. In Stage 2, the development organization then copies this software into a development environment which will allow the software to be tested and verified prior to deployment into the integration environment. The development environment, which in this example is run solely by developers, is a vertically-scaled representation of the live production environment. However, in this scenario, software deployment into the development environment is typically accomplished in an ad-hoc manner by developers due to time constraints and the eagerness of developers to test out new functionality with respect to delineated business requirements and see how different components interact. Configurations are often made in a makeshift manner and no effort is made to determine performance or customer impact. Software components that are determined to be malfunctioning or not meeting business requirements are immediately sent back to their respective development group for recoding. When the particular software component has been redeveloped, it is placed back into the development environment for analysis. This process is repeated until a certain level of confidence is achieved by the development organization that the software is stable enough and meets enough of the business requirements to make it ready for integration in Stage 3.

While in this case the integration environment is like the development environment in that it is a vertically-scaled representation of the live site, it is very different in that the integration environment is set up to have a perfectly clean snapshot of the current production site. No superfluous software, debugging code, makeshift configurations, etc. are present in the integration environment. The integration environment in this example is run by a separate group of systems integration (SI) engineers, and the developers do not play a role in its maintenance. Software is moved into the integration environment in the form of documented releases with the use of a source code repository. These releases are handed over to the SI engineers by the developers. All too often little documentation is available for integration engineers to ascertain the background of the software and the complex interdependencies present. Therefore, the path from the development environment to the integration environment is typically the only opportunity for SI engineers to ascertain all of the information required to ensure a smooth production deployment and move on to Stage 4.

It must be determined if the software provided to the SI engineers in this example is backwards compatible with the current production software. For example, if a stored procedure (SP) is updated on a database prior to the front-end web server, the SP update must not cause a service interruption to the customer. The software deployment risk scoring heuristic assumes that the software to be deployed into production is backwards compatible with the legacy software already in use. In order to effectively test backwards compatibility and by extension the various transitional states in which the web site will be during the deployment process, the software releases provided to SI must be applied in sequence and

regression tested. Let's take the scenario of three releases provided to SI: one database release, one middleware tier release, and one web server release and refer to Figure 4 below.

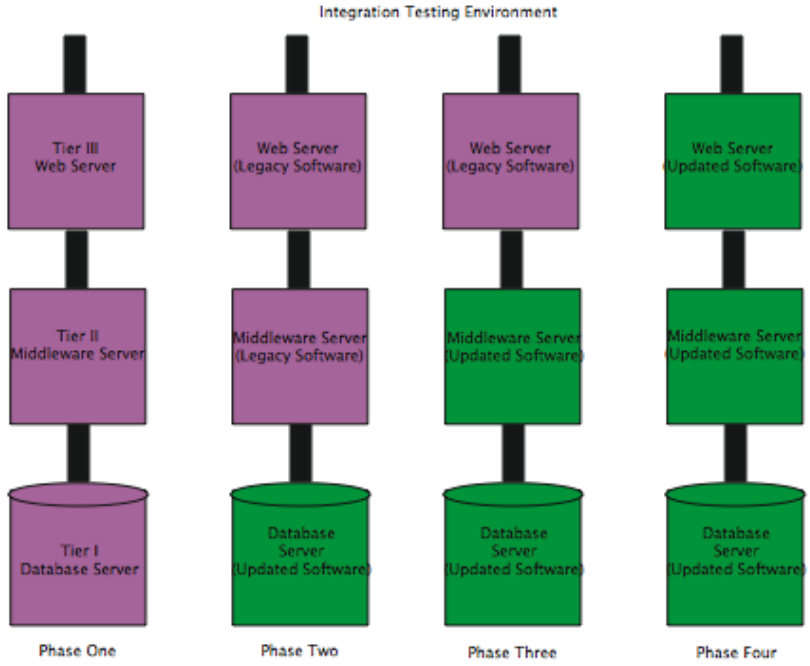

Figure 4: Integration Transitional Testing Phases

At phase one of Figure 4, the example integration environment is a mirror image of the current production environment. A team of testing engineers must first regression test this initial state to ensure that it functions exactly like the live production site. At the completion of this initial regression test, software may be applied to the SI environment with the principle of mirroring the production deployment procedure. It is essential at this phase for the systems integration engineers to analyze all of the configuration mappings and software interdependencies present in the software. When SI engineers reach a level of confidence that all of the information regarding configurations and dependencies has been ascertained, software may be applied to the environment.

At phase two in Figure 4, the integration engineers begin moving software into the environment. During this phase, the DB release is applied to the database server at Tier I. A thorough round of regression testing must be conducted to ensure that the software deployed to the DB server is backwards compatible with the legacy software on the middleware server and web server. During this regression test, it must be ascertained how the customer-facing web server will be impacted by the DB upgrade. If this regression test is successful, the software release for the middleware server Tier II may be applied as shown in phase three. Again, a complete round of regression must be conducted to ensure backwards compatibility and zero customer impact. If this second round of regression testing is successful, the customer-facing web server release is applied. At phase four, all of the software is now deployed into the environment for the final round of regression testing. Assuming all regression tests are successful, the SI engineers may now apply the proposed software deployment risk scoring heuristic to determine the risk at each tier. From the heuristic, the total tier risk (T_{RT}) results for each tier was found to be as follows:

Tier I Database Server Score → T_{RT} = 86.6

Tier II Middleware Server Score → T_{RT} = 48.1

Tier III Web Server Score → T_{RT} = 159.72

With T_{RT} = 159.72, Tier III represents the tier with the most risk. Systems integration engineers know that they must concentrate a significant amount of their efforts on the web server tier, and all functional and unit testing must take place with this risk assessment in mind. When testing is deemed to be successful and determined to be adequate for customer use, the software is considered ready for production deployment. However, because the risk of a deployment related problem at the web server tier is found to be high, it is essential that a carefully crafted deployment approach be utilized.

Let us now refer to Figure 5 below which illustrates a simplified, horizontally-scaled representation of Tier III, the production web server tier. Because of the risk identified through the use of the heuristic, it is obvious that the web server tier deployment must be carefully crafted so that customers do not experience an interruption in service during the deployment process.

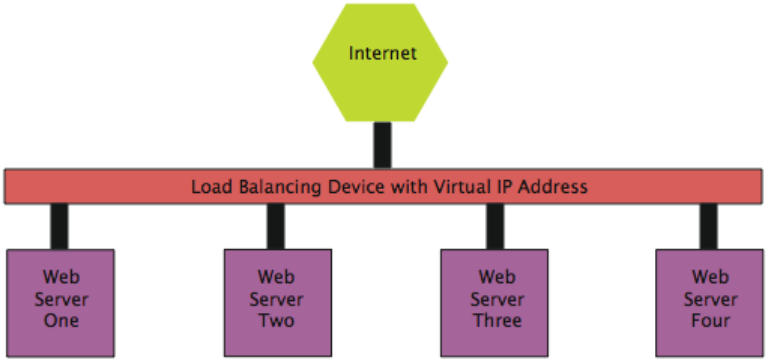

Figure 5: Simplified Example of Typical Production Web Server Arrangement

In this example, there are four identical production web servers linked to a network load-balancing device. A customer making a request from the internet to a web server is directed to a particular web server bound to the device to keep the load on each server as even as possible. Let's assume that in order to perform the upgrade on the web server, the server must be taken out of service. Let's also assume that it is not possible to perform a swap due to performance issues (i.e. take half of the servers OOS, upgrade these, and simultaneously take the other half OOS while putting the upgraded half back IS). Since it is not possible to shut down all of these web servers at once to perform the software upgrade, a server-by-server approach must be taken.

When initially performing the deployment, one of the web servers will be taken out of service so that customers are no longer utilizing it. The server is then upgraded, placed back in service, and the next server to be upgraded is taken out of service. This process is repeated until all servers are upgraded. Let us now refer to Figure 6 below. This diagram represents the transitional state of the live production web site during the production deployment process.

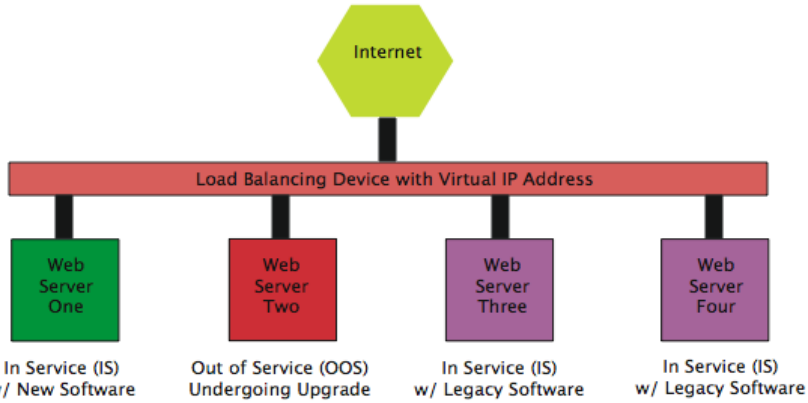

Figure 6: Live Production Web Site Transitional State

It is clear from this diagram that customers will be simultaneously utilizing web servers both with the legacy software and the new software when the production web site is in this state. If, for example, the software update incorporates a visual change to the web site, it is undesirable for customers to have this form of inconsistent experience. In this case, a type of 'on-switch' must be implemented via either a database flag, date condition, etc. This switch would be turned 'on' only at such time that all web servers have been upgraded and individually validated.

6. Conclusion

By using the software deployment risk scoring heuristic, the risk of a deployment related problem at a particular tier in a system can be quantified. An analysis of this risk score provides engineers with the data necessary to determine where to focus testing and how best to deploy the software in each tier. A tier with a high risk factor will almost certainly require a more lengthy and drawn out deployment process. Obviously, a longer deployment process means that customers will not see the changes on the web site as quickly as otherwise possible which could result in a loss of revenue. Also, a more lengthy deployment process will be more costly to the company as additional IT staffing time would be required to carry out the deployment. This heuristic has great value in illustrating where the efforts of integration engineers would be most valuable and where best to focus the majority of the deployment time interval in order to facilitate the greatest cost savings to the company. If the organization finds that the deployment on a particular tier is too risky and by extension a deployment too costly, the software to be deployed can be scaled back so that less dramatic changes are introduced. Once the software changes are scaled back, the system can be easily reanalyzed using the heuristic to quantify how much the changes have reduced the risk. Conversely, if by use of the heuristic, the organization finds that the risk to the system is very low, more software changes may be introduced and the risk assessment may again be calculated. It may be concluded that the use of the software deployment risk scoring heuristic is thus an essential element of the systems integration process.

7. Future Work

Future work on expanding upon this paper would include perfecting the heuristic to include more precise measuring tools for determining the severity of the changes to individual software components. For example, the complexity of the heuristic may be augmented to account for a plethora of tiers and numerous non-linear data flows. One of the limitations of

this heuristic is that it assumes all of the individual software components are equal in terms of impact and importance. In practice, some of the many software components considered may be inconsequential and thus have little impact on the system upon modification. Conversely, the integrity of one or more of the software components may be extremely imperative and the slightest modification may prove to have a catastrophic impact. In order to address this potential oversight, research may be performed to determine how the heuristic could be augmented to include score rankings for each particular software component. Also, the heuristic interrogatives may be enhanced to include more precise scoring techniques to account for interrogatives which may carry more weight than others for a particular application.

The heuristic may also be expanded to account for instances where one tier in the environment is undergoing a deployment for decommissioning. The heuristic in this instance must be able to determine the respective impact the elimination of this tier will have upon all other tiers in the system. Additionally, the heuristic may be tuned to be more flexible to account for many types of interdependencies and other complications. An interesting analysis of the heuristic would also be how it would perform in a scenario where backwards compatibility is not certain. Finally, future research on the heuristic ideology would be beneficial to determine how this methodology could be applied to a cloud computing environment. In a cloud computing environment, all of the technology is accessed from the internet (cloud) and provided as a service. Because in this situation individual software components are indistinguishable, the heuristic would require modification to use individual services as a metric instead. A method of identifying these services and incorporating them into the heuristic would require further investigation.

References

[1] Rana, A.I.; Arfi, M.W. 2005. *Software Release Methodology: A Case Study.* Pages 1-10. Digital Object Identifier 10.1109/SCONEST.2005.4382893

[2] Fanberg, V. 2001. *Use of Binary File Comparison Tools in Software Release Management.* Pages 436-444. Digital Object Identifier 10.1109/APAQS.2001.990049

[3] Paul, R. 2001. *End-to-Rnd Integration Testing.* Pages: 211-220 Digital Object Identifier 10.1109/APAQS.2001.990022

[4] Davis, Jake. 2005. *One-Click Release Management.* June 2005. Linux Journal, Volume 2005. Issue 134. Publisher: Specialized Systems Consultants, Inc.

[5] van der Hoek, Andre, Hall, Richard S., Heimbigner, Dennis, Wolf, Alexander L. November 1997. *Software Release Management.* ACM SIGSOFT Software Engineering Notes, Volume 22 Issue 6. Publisher: ACM

[6] ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Publisher: Springer-Verlag, NY, Inc.

[7] de Jonge, M. July 2005. *Build Level Components.* IEEE Transactions on Software Engineering. Volume 31, Issue 7. Pages 588-600. ISSN: 0098-5589. INSPEC Accession Number 8568787.

[8] Krishnan, Mayuram S. Oct 1994. *Software Release Management: a Business Perspective.*

[9] CASCON '94: Proceedings of the 1994 Conference of the Centre for Advanced. Studies on Collaborative Research. Publisher: IBM Press.

[10] van der Hoek, A., Heimbigner, D., and Wolk, A.L. Wolf. March 1996. *A Generic, Peer-to-Peer Repository for Distributed Configuration Management.* Proceedings of the 18th International Conference on Software Engineering. IEEE Computer Society.

[11] van der Hoek, A. 2001. *Integrating Configuration Management and Software Deployment.* Proceedings of the Working Conference on Complex and Dynamic Systems Architecture.

[12] van der Hoek, A. 2002. *A Testbed for Configuration Management Policy Programming.* IEEE Transactions on Software Engineering.

[13] Hong, S.B., Kim, Kapsu. September 1997. *Classifying and Retrieving Software Components Based on Profiles.* Proceedings of 1997 International Conference on Information, Communications and Signal Processing. ICICS. Volume 3. Pages 1756-1760.

[14] Hall, R.S., Heimbigner, D., Wolf, A.L. 1999. *A Cooperative Approach to Support Software Deployment Using the Software Dock.* Proceedings of the 1999 International Conference on 16-22. May 1999. Pages 174-183.

[15] Ganguly, A.; Yin, J.; Shaikh, H.; Chess, D.; Eilem, T.; Figueiredo, R.; Hansom, J.; Mohindra, A.; Pacifici, G. 2007. *Reducing Complexity of Software Deployment with Delta Configuration.* Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on May 21 2007-Yearly 25 2007 Pages: 729-732 Digital Object Identifier 10.1109/INM.2007.374699

[16] Alan. 2007. *Software Deployment, Past, Present and Future.* Future of Software Engineering, 2007. FOSE '07 23-25 May 2007 Pages: 269-284. Digital Object Identifier 10.1109/FOSE.2007.20

[17] Belguidoum, M., Dagnat, F. 2007. *Dependability in Software Component Deployment.* Dependability of Computer Systems, 2007. DepCoS-RELCOMEX '07. 2nd International Conference on 14-16 June 2007 Pages: 223-230 Digital Object Identifier 10.1109/DEPCOS-RELCOMEX.2007.16

[18] Kajko-Mattsson, M., Meyer, P. 2005. *Evaluating the Acceptor Side of EM3: Release Management at SAS.* Empirical Software Engineering. 2005 International Symposium on 17-18 Nov 2005 Digital Object Identifier 10.1109/ISESE.2005.1541840

[19] Walker, Leslie. Saturday, June 25, 2005. *E-Commerce's Growing Pains: Competition Intensifies as Industry Turns 10 Years Old.* Washington Post Staff Writer. http://www.washingtonpost.com/wpdyn/content/article/2005/06/24/AR2005062401905.html

[20] Keefer, Gerold. Feb. 6, 2002. *Extreme Programming Considered Harmful for Reliable Software Development.* AVOCA GmbH. Pages: 1-14.

Authors

John Comas is a doctoral student at the School of Systems & Enterprises at the Stevens Institute of Technology. He received his Bachelor of Engineering and Master of Engineering degrees in Computer Engineering in 2003 from Stevens Institute of Technology. John Comas is conducting research on systems integration in the information technology field focusing on agile software development, risk management, quality assurance, and process flow modeling. In addition, he works in New York, NY as a senior integration engineer in the software development and systems engineering group at barnesandnoble.com.

Dr. Richard Turner is a Distinguished Service Professor at Stevens Institute, a Visiting Scientist at the Software Engineering Institute of Carnegie Mellon University and a respected researcher and consultant with thirty years of international experience in systems, software and acquisition engineering. Dr. Turner has supported defense, intelligence and civil government agencies. A charter member of the author team for CMMI, he has led process improvement initiatives across a broad range of disciplines. Dr. Turner is co-author of three books: Balancing Agility and Discipline: A Guide for the Perplexed, CMMI Distilled, and CMMI Survival Guide: Just Enough Process Improvement.

Dr. Ali Mostashari is the Director of the Complex Adaptive Sociotechnological Systems (COMPASS) Research Center and an Associate Professor of Systems Engineering at the School of Systems and Enterprises at Stevens Institute of Technology. His research focuses on the application of complexity science to systems engineering, and to sociotechnological systems analysis and design. He currently serves as a senior strategy consultant for the United Nations Development Programme's Knolwedge Management 2.0 strategy. Dr. Mostashari was selected as a Asia 21 Young Leader, and was nominated by the UNDP Assistant Secretary General for Africa for the World Economic Forum's Young Global Leaders 2008 award. In 2004, he was selected as a top finalist of UNDP's Leadership Development Programme from over 7000 applicants from 78 countries worldwide. Between 2004 and 2007

he served as a LEAD Project Manager and Strategoic Resource Manager, with over $2 billion in development project portfolio in sub-Saharan Africa. Dr. Mostashari holds a Ph.D. in Engineering Systems from MIT, a Master's in Civil Engineering from MIT, a Master's in Technology and Policy from MIT, a Master's in Chemical Engineering from the University of Nebraska and a Bachelor's in Chemical Engineering from Sharif University of Technology.

Dr. Mo Mansouri is a Postdoctoral Associate at the Center for Adaptive Sociotechnological Systems (COMPASS) of the School of Systems and Enterprises (SSE) at Stevens Institute of Technology. He is conducting research on resilience of ports, maritime transportation and infrastructure systems, and enterprises. Dr. Mansouri received his Doctor of Science in Engineering Management from The George Washington University and for several years served as a management consultant for The World Bank, HAND Foundation, NIAC and other for-profit and non-profit organizations on issues of development planning, strategic management, and decision-making.