Comparison of Approaches to Service Deployment

Vanish Talwar, Qinyi Wu†, Calton Pu,†
Wenchang Yan†, Gueyoung Jung†, Dejan Milojicic
HP Labs, Georgia Tech†
[vata, dejan]@hpl.hp.com, [qxw, calton, wyan, helcyon1]@cc.gatech.edu,†

Abstract

IT today is driven by the trend of increasing scale and complexity. Utility and Grid computing models, PlanetLab, and traditional data centers, are reaching the scale of thousands of computers. Installed software consists of dozens of interdependent applications and services. As the complexity and scale of these systems continues to grow, it becomes increasingly difficult to administer and manage them. At the same time, the service deployment technologies are still based on scripts and configuration files with minimal ability to express dependencies, to document and to verify configurations. This results in hard-to-use and erroneous system configurations. Language- and model-based tools, such as SmartFrog and Radia, are proposed for addressing these deployment challenges, but it is unclear whether they are beneficial over traditional solutions.

In this paper, we quantitatively compare manual, script-, language-, and model-based deployment solutions as a function of scale, complexity, and susceptibility to change. We also qualitatively compare them in terms of expressiveness and barrier to first use. We demonstrate that script-based solutions are well matched for large scale deployments, language-based for services of large complexity, and model-based for dynamic changes to the design. Finally, we offer a table summarizing rules of thumb regarding which solution to use in which case, subject to deployment needs.

1 Introduction

The scale and complexity of today's IT systems and services makes them increasingly difficult and expensive to administer and deploy. We define a service broadly as a standalone software component that encapsulates and presents useful functionality, is installed in a computing environment, and can be composed into an overall system or application. Services in this broad sense include business services as well as modules such as transaction services or databases. They can be realized as Web or Grid services or even as component services in an operating system. This shift points to a general view of service-oriented computing. By deployment, we mean an action to download, configure, activate, and maintain the life cycle (e.g., react to failures, terminate, and restart) of services.

A system update at a moderately-sized data center may require changes to a thousand machines. In addition, there may be interdependencies among the applications installed on these machines. For example, a typical Web-based e-commerce application consists of a three-tier system, comprising the database, application, and Web server tiers. The application tier further consists of the application server, the application in question, and other services on which the application depends. Large scale data centers in companies such as Yahoo and Google can be significantly larger in size with significantly more complex services.

New computing models, such as Utility Computing [1, 2] and Grid Computing [3] grow even more significantly in scale. Utility computing requires rapid redeployment of software for changing users of computer fabrics. Grid computing poses similar requirements for the Grid services. In practice, the IT service companies (e.g., those handling the outsourced data processing tasks) managing a large number of installations around the world have the same requirements of Utility Computing and Grid Computing. Recent studies show that management of software deployments dominates system administration costs [6], and that configuration is a major source of errors in system deployment [7].

A concrete example of serious challenges in system configuration is the long-lived and evolving nature of large-scale services and applications in these environments, which makes the management of dependency among service components critical. The changes on a service component need to be propagated or contained as appropriate, so the services and applications that use the component may continue to function correctly. In addition to planned changes, the unplanned changes such as failures also need to address dependencies. Specifically, if some services are dependent on a failed service, then these services may also need to be restarted.

Today's deployment tools provide varying levels of automation, classified into manual, script- language-, and model-based (see Table 1 for the deployment tasks executed by these approaches). Automation of service deployment is beneficial for improved correctness (by reducing human errors), speed (parallelizing long-running installations), as well as for improved documentation. However, automation is achieved at an increased cost at the development time and an increase in the

Table 1. ***Comparison of Deployment Approaches.*** *Darker shading means more automated, no shading implies manual steps.*

Deployment phases	***Deployment Approaches***			
	manual	***script-based***	***language-based***	***model-based***
Development	none	1. develop tools (workflow execution scripts, distribution, verification) 2. develop installation and startup script templates (per application)	1. develop configuration language, parser 2. develop tools (engine, distribution) 3. develop configuration, installation specification templates expressed in the language templates (per application)	1. develop schemas for models: package, best practices, SW dependency/inventory, HW resources, interoperability 2. develop tools for: model lifecycle mgmt, dependency analys., distribution, engine 3. create instances of models: package (per app); best practices (per customer) 4. update SW dependency model & create resource models (per node)
Design	none	1. populate application templates with customer-specific attributes 2. construct workflow scripts (per application)	1. populate application templates with customer-specific attributes 2. construct workflow components (per application)	1. based on customer, select package models from best practices model (per app) 2. dependency analysis (app per node), order of (re-)install, activate, terminate
Operational	1. distribute packages to repository 2. login to each target node 3. download binaries, configure SW installation, install apps (per node) 4. configure SW activation and manually activate (per node) 5. check deployment status and application state	1. invoke distribution module 2. invoke installation and ignition workflow scripts 3. invoke verification scripts	1. invoke the distribution module 2. invoke installation and ignition workflow components 3. verify notification events	1. update unified interoperability model 2. invoke distribution module 3. invoke installation and ignition workflow components 4. verify notification events
Change	1. manually detect changes 2. manually repeat any or all of the above phases to adapt to change.	1. discover change (manual, ad-hoc) 2. react to change (ad-hoc process), repeat some/all of above phases	1. discover changes (ad-hoc) 2. react to change, load pre-determined component	1. automatically discover/react to change; reflect on model; activate adaptation & execution modules

learning curve for administrators. This initial cost and overhead may be acceptable if the overall gains are significant and worthwhile. The question facing IT managers today is: which of these deployment approaches should they adopt (and when) for the IT infrastructure and services to achieve overall gains?

In this paper, we quantitatively and qualitatively compare the approaches to service deployment in terms of scalability, complexity, and susceptibility to change. We identify cases where each approach is best suited. Our hypotheses are summarized in Figure 1. First, manual deployment is likely to be well suited for simple, small scale deployments because it has small barrier to entrance. Second, for larger scale, yet still simple deployments, script-based deployment should present advantages over the manual approach, even though scripts may require learning the scripting language and then the development and maintenance of scripts. Third, for large scale, complex deployments, a language-based approach such as SmartFrog [5] should be the best. It requires some additional investment in understanding the complexity of the framework and developing the template and configuration descriptions consisting of dependencies. Finally, the environments involving a lot of dynamic, run-time changes are best suited for model-based deployment.

The main technical contribution of the paper is an experimental confirmation of the above hypotheses described in Section 4. In analogy to experimental evaluation of program complexity and quality of service, we define manageability as an ability to manage a system component. We define Quality of Manageability (QoM) as a measure of manageability. QoM has qualitative and quantitative measures. Quantitative QoM includes:

- number of lines of code (loc) written for deployment;
- number of steps involved to deploy;
- loc to express configuration changes; and
- time to develop, deploy, and make a change.

LOC are a relevant metrics because of the maintainability of configuration, which is inversely proportional to the number of LOC. The smaller and more expressive a configuration, the easier it is for a system administrator to install, configure, and maintain. Similarly, number of steps is proportional to the time and cost of engagement of a human operator.

We also use the following Qualitative QoM:

- ability to automate the management process, including adaptability to changes (e.g., failures, load);
- robustness, expressed in terms of misconfigurations;

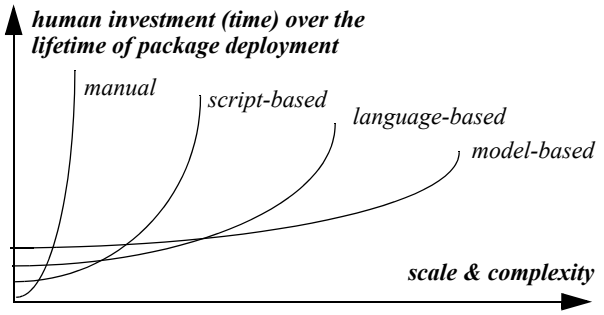

Figure 1. Hypothesis: the level of automation of a tool, pushes the cost earlier in the development cycle in order to benefit from the repeated deployment. Developing the tools, learning them, and creating templates come at an initial cost. The cost will pay off in complex or scaled (repeated) deployments.

- expressiveness of management (e.g., ability to express constraints, dependencies, and models); and
- barrier to first use of the deployment tool.

In order to compare deployment approaches, we leverage the methodologies used for comparing programming languages, domain specific languages, databases, and software engineering in general. The programming language community typically compares programming languages in terms of execution time, ease of use, lines of code, length, amount of commenting, etc. [8]. Our work is also related to domain specific languages, such as the application of compiler extensions to identify errors in systems programming [9, 10]. A comparison between the manageability of the Oracle 9i and Oracle 10g databases motivated us to use number of steps as a metric [11]. Itzfeldt classifies maintainability into modularity and complexity, testability, understandability, and modifiability. He derives the following quality metrics for software management: size, control structures, data structure and flow [12].

A framework for software deployment technologies proposed by Carzaniga et al. [13] characterizes *product*, *site*, and *policy* models. In our classification, the script-based approach supports site model; language-based approach supports product and site model; and model-based approach supports all three models.

Our work differs from the related work in many ways. First, nobody has characterized a spectrum of deployment automation options from manual to model-based. Second, there is no previous quantitative comparison of deployment solutions (Carzaniga et al. offer qualitative comparison [13]). Third, we formulate a set of metrics for comparison of QoM for deployment approaches.

The remainder of the paper is organized in the following manner. Section 2 presents a use case scenario justifying the need for deployment automation. Section 3 provides background information on the deployment of computer systems. In Section 4 and Section 5 we perform quantitative and qualitative comparison of deployment approaches. Section 6 presents a summary of the paper and of future work.

2 Use Case Scenario

There are many scenarios that lend themselves to analysis of deployment tools. In this section, we describe a real-life scenario emphasizing the problems with dependencies, failures, and the need to document changes.

Sarah has installed Java PetStore on a three node Windows-based cluster. She manages it with a remote tool, so she's configured it to be part of a remote domain. It took her a few days to install all the required packages, applications, and tools; because she had specific requirements, she had to make certain changes in several steps of the configuration and deployment. Each part of the installation had its own instructions, so she documented everything in a notebook. Because the application had so many dependencies, she had to manually configure packages with the configuration parameter values from other packages — for example, for node names and IP addresses. She repeatedly had to enter these values in different places, so she occasionally entered them incorrectly. After Sarah used PetStore for several days, an application on the remote system rebooted all her systems because several Windows updates needed to be applied. Unfortunately, this action erroneously reimaged some of her systems, and Sarah had to reinstall everything from scratch.

In this scenario, a more sophisticated deployment tool would have benefited Sarah in many ways. First, she wouldn't have had to do the installation manually each time. A carefully drafted template describing the steps and tools would have helped during the first deployment and even more so during subsequent deployments. Next, the dependencies between certain components could have been instantiated in one spot with variable names used at other areas, reducing the need to make changes and the likelihood of making errors. Moreover, the deployment tool could have evaluated many values at the time the components were started, eliminating the need for manual initiation altogether. The remote configuration tool that caused the incident could also have been part of the configuration, which would have automated changes in interdependent systems. Finally, the system's documentation would have been very coherent and consistent, reduced to a single configuration file, documenting an absolute minimum number of parameters and making subsequent changes easy.

3 Background

In this section we describe Nixes, SmartFrog, and Radia, as examples for the script-, language, and model-based deployment approaches respectively. In Figure 2(a-d), we illustrate deployment steps for each of approaches. There are many other deployment tools, such as those described in [14-17] and a number of other tools surveyed by Anderson et al. [18]. Their description and comparison is beyond the scope of this paper.

Nixes is a tool used to install, maintain, control, and monitor applications on PlanetLab [19]. Nixes consists of a set of bash scripts, a configuration file, and a Web repository, and can automatically resolve the dependen-

cies among RPM packages. For small-scale systems, Nixes is easy to use: users simply create the configuration file for each application and modify scripts to deploy on the target nodes. But for large and complicated systems, Nixes is not as effective, because it does not provide any automated workflow mechanism.

SmartFrog (SF) is a framework for service configuration, description, deployment, and life-cycle management [5, 20, 21]. SF consists of a declarative language, the engines that run on remote nodes and execute templates written in the SF language, and a component model comprising a wrapper around the deployed services. The SF language supports encapsulation a la classes in Python as well as inheritance and composition to allow configurations to be customized and combined. It enables static and dynamic bindings between components to support different ways of connecting components at deployment time. The SF component model enforces life-cycle management by transitioning components through five states: installed, initiated, started, terminated, and failed. This allows the SF engine to automatically redeploy components in case of failure.

Radia [4], a change-and-configuration management tool, employs a model-based approach. For each managed device, administrators define a desired state, which is maintained as a model in a central repository. Clients on the managed devices synchronize to this desired state, which triggers deployment actions. We also consider a **hypothetical model-based deployment solution** that uses the following six models: package (configuration, installation, registry entries, binaries, and such); best practices (matching the needs of specific customers); software dependency (deployment relationship with other software components, operating systems, and hardware); infrastructure (servers, storage, and network elements); a software inventory (currently installed software); and interoperability among management services models.

4 Quantitative Evaluation

We present in this section a quantitative comparison of the deployment approaches introduced in Section 3. We conducted three sets of experiments.

The first set of experiments is for the deployment of n-tier testbeds. We chose a three-tier testbed for the experiment that can exemplify the system administrator's work. The testbed consists of a web server, an application server, and a database. The system is complex enough to have numerous dependencies among various components across the different tiers. The system can

1. Propagate packages to web repository
1. Login to web server host
2. unpackage Apache
3. build & install Apache
4. edit httpd.conf
5. start Apache
6. Inspect Apache process list
7. Login out of web server host

Figure 2(a) Steps for installing Apache manually.

```
WEB_SERVER=poseidon.cc.gatech.edu #Web
repository
APACHE_ARCHIVE=httpd-2.0.49.tar.gz #Binary
Archives
DIR=/usr/local #Installation Directory
cd $DIR
if [[ ! -d $APACHE_HOME ]]; then
wget
$WEB_SERVER:$WEB_PORT/$WEB_DIR/$APACHE_ARCHIV
E
tar -xzf $APACHE_ARCHIVE
cd $APACHE_INSTALL_HOME
./configure
make > /dev/null
make install > /dev/null
fi
```

Figure 2(b) Illustration of a bash script for installing Apache.

```
ApacheInstaller extends GenericInstaller
webServerHost;
tarLocation "<location>"; // eg. /apache.tar
installLocation "<location>"; // eg. /
file "<file>"; // eg. apache.tar
name "<name>"; // eg. apache
installScript extends vector
— ATTRIB cdApacheHome;
— ATTRIB configureScript;
— ATTRIB makeScript;
cdApacheHome extends concat
— "cd ";
— ATTRIB home;
configureScript "<script>"; // eg.
./configure
makeScript "<script>"; // eg. make install
sfConfig extends ApacheInstaller;
```

Figure 2(c) Illustration of SF language for installing Apache.

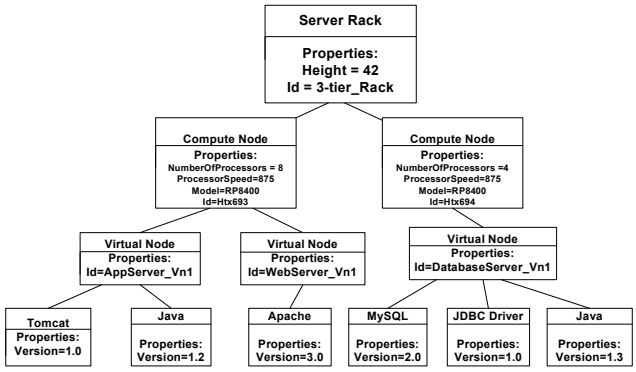

Figure 2(d) Visual form of the system model.

also be configured in different scales. For example, one could introduce multiple instances of the web server or multiple instances of the application server. We compare bash script- with SmartFrog language-based approach.

The second set of experiments is for MySQL configuration parameters. We chose an application with an inter-

esting set of configuration parameters that need to be tuned for different system setups. MySQL is a well-known open source database software that has a set of tunable configuration parameters for setting up the database under small, medium, large, and very large setups. Here we compare native MySQL configuration files (corresponding to script-based approach) and SmartFrog.

The third set of experiments compare the installation time when deploying a 2-tier testbed using Nixes (a script-based deployment tool), and SmartFrog. We describe these experiments below.

4.1 N-Tier Testbed Deployment

Experimental Setup. Experiments for the deployment of n-tier testbeds were conducted using SmartFrog 3.0, a web server (Apache 2.0.49), an application server (Tomcat 5.0.19), a database server (MySQL 4.0.18), and the PetStore (iBATIS JPetStore 4) & Guest Book applications. The testbed was setup on a Linux environment. The components of the n-tier testbed were deployed on separate nodes. Each of the components had native configuration files: httpd.conf for Apache; server.xml for Tomcat; web.xml for web applications using Tomcat; and my-*.cnf for MySQL. We wrote bash scripts and SmartFrog components for the installation, and ignition phases of the components in the system.

Experiment Description. The goal of the experiments is to quantitatively compare the quality of manageability for different deployment approaches. The metrics we choose for this measurement are *number of steps,* and *number of lines of code*. The measurements are done by varying the scale and complexity of the n-tier testbed. The scale is measured by the number of nodes. The complexity is defined as a function of the number of deployment dependencies, and the number of software components. The deployment dependencies we consider for our experiments are installation dependencies, configuration dependencies, and ignition dependencies. We define various levels of complexity of the testbed (see Figure 3). The scale of the system is varied through horizontal scaling of the tiers. In our experiments, we performed simultaneous horizontal scaling of the web server and application server tiers. The ratio of the Apache web server to Tomcat application servers for the purpose of horizontal scaling are 1:2, 2:4, and 4:8**.**

Test Workload. We identified a *test workload* consisting of a set of *tasks* to be performed by an end-administrator for the deployment of the testbed. The test

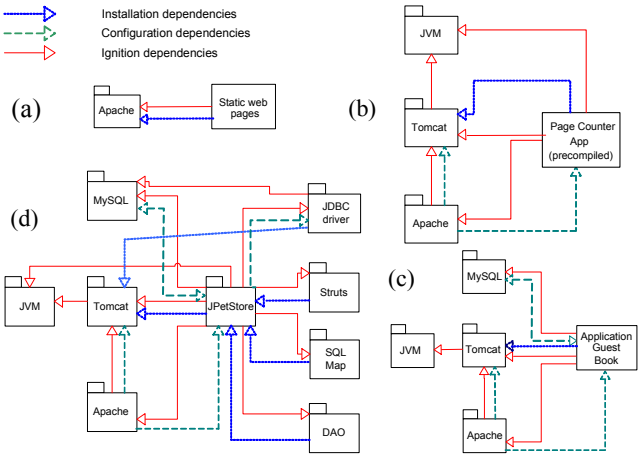

Figure 3. Complexity definitions (a) Simple: 1-tier testbed, (b) Medium: 2-tier testbed with simple application, (c) Complex: 3-tier testbed with Guestbook application, (d) Very complex: 3-tier testbed with PetStore application.

workload covers the installation and ignition of the testbed. The following tasks are included:

1. Create the specifications for configuration, installation, and ignition of the software by filling in appropriate values in the specification templates.
2. Create workflow descriptions for the deployment tasks of installation and ignition of the applications.
3. Distribute the binaries, specifications, and workflow descriptions to all of the nodes.
4. Execute the installation workflow descriptions to install the testbed.
5. Execute the ignition workflow descriptions to activate the testbed.
6. Verify if the installation and ignition completed successfully.

Results.The results obtained represent the deployment effort for an *end-administrator*, and reflect the cost of deployment incurred beyond initial development.

Figure 4 shows the comparison of the number of steps that an end-administrator performs as a function of scale and complexity. The graph shows that as the complexity of the system increases, the difference in the number of steps to be performed by an end-administrator widens for a manual approach in comparison to either a script-based approach or a SmartFrog-based approach. Comparing results for SmartFrog with the script based approach, one sees a constant difference in the number of steps to be performed. For the manual approach, the number of steps is linear to the number of nodes because the administrator needs to repeat the same steps for each node. However, the steps for the script and SmartFrog cases remain the same with increasing scale. The reason

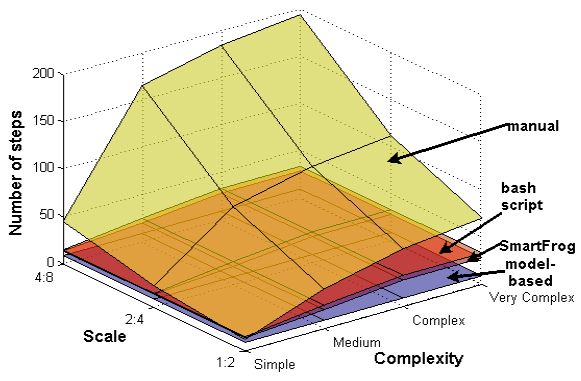

Figure 4. ***Number of steps as a function of scale and complexity.*** *The results shown for bash and SmartFrog are from real systems. The results for the model-based approach are estimated based on our definition of the hypothetical model-based solution. It is represented as a flat plane in the graph.*

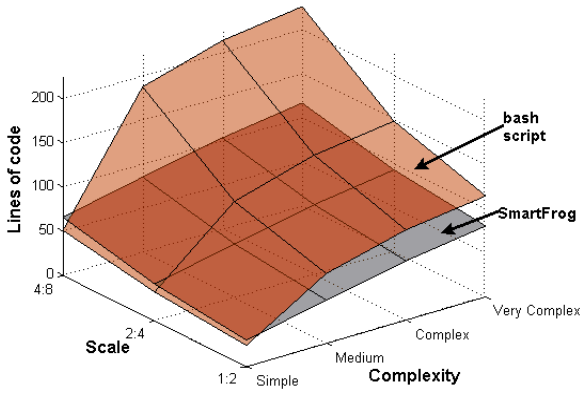

Figure 5. ***Lines of code as a function of scale and complexity.*** *We present only results for bash and SmartFrog; lines of code make no sense for manual approach and for model-based we do not have an accurate estimate.*

is that both can manage the scale by reusing the previous code. The administrator only needs to add extra lines of code at each step. In comparison to the manual, script, and SmartFrog based approaches, the (hypothetical) model-based approach provides the advantage of a constant number of steps to be performed by the end-administrator with varying scale and complexity.

Figure 5 shows the comparison of the number of lines of code that an end-administrator writes as a function of scale and complexity. The graph shows that as the complexity of the system increases, the difference in the number of lines of code to be written by an end-administrator widens for a script-based approach in comparison to a SmartFrog based approach. When the system scales on Apache and Tomcat, the dependencies between different components increase in proportion to the number of Apache and Tomcat servers. The new lines of code to be written with a script-based approach is $O(n \cdot m)$ where n is the number of Apache servers and m is the number of Tomcat servers. By comparison, SmartFrog only needs to specify the configuration value for Apache and Tomcat servers once; one can use its link reference facility to capture these dependencies. As a result, SmartFrog can capture the dependencies linear to the number of Tomcat and Apache servers. Furthermore, SmartFrog saves lines of code for software distribution and verification of successful installation and activation of testbed. This is result of the underlying transport built into the SmartFrog engine and the provision of the SmartFrog event framework. These features could have also been built into a script-based solution but it would involve much more development effort and it has not been done in our experiments.

SmartFrog benefits in terms of reduction in number of steps and lines of code through *automation*, *workflow*s, and the ability to handle added dependencies through the *link reference* feature of SmartFrog language. (See Table 1 for details on the automation of language- and script-based deployment solutions.) Furthermore, the SmartFrog language provides rich facilities to specify the sequencing relationships between different software components through workflows. One can also create workflows composed of sub-workflows. For example the workflow in the 3-tier application shown in Figure 6 can be easily captured by Sequence and Parallel components in SmartFrog.

4.2 MySQL Deployment

Experimental Setup. The experiments are conducted with the configuration file for MySQL version 4.0.18. The configuration files used in the experiment are my-[small, medium, large, huge].conf. We represented the configuration information using the SmartFrog language and applied the inheritance and link reference features of SmartFrog.

Experiment Description. The goal of this experiment is to quantitatively compare the quality of manageability for maintaining the configuration of MySQL using the various deployment approaches. The metrics we choose for this measurement are *lines of configuration code to maintain*, and *number of configuration values to be edited in response to changes in the system*. The configuration values under consideration are the performance tuning parameters for MySQL. These parameters can be broadly classified under port number parameters, key and sort buffer sizes, read and write buffer sizes. The evaluation in the experiment is to vary the complexity of

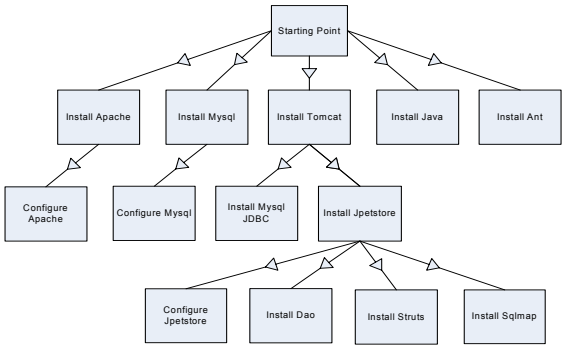

Figure 6. Workflow of the installation of the 3-tier testbed with PetStore application.

the MySQL setup from small to medium to large to very large. Small refers to a setup serving a small number of clients (10-100), being used infrequently and consuming few resources. Medium refers to a setup that serves more clients (100-1000), and is used together with other programs (like a web server). Large refers to a MySQL setup serving an even greater number of clients (1000-5000), with the system running mainly MySQL. Very Large refers to a MySQL setup serving a huge number of clients (>5000), the system running mainly MySQL, and having a large available memory space (>10GB).

The deployment approaches considered are a language-based approach to represent configuration information, specifically, SmartFrog language, and a non-language based approach, specifically, the MySQL default configuration files. Figure 7 provides examples of code representing MySQL configuration information using the two approaches.

Results. Figure 8 shows the comparison of the cumulative number of lines of code that a MySQL administrator has to maintain as we increase the complexity of the system from simple to very large. A MySQL administrator maintaining configurations for only *small* setups maintains the same lines of code with a language and non-language based approaches. When the administrator is asked to maintain a *medium* setup, the cumulative number of lines of code that need to be maintained dou-

```
[client]port= 3306
[client]socket= /tmp/mysql.sock
[server]port= 3306
[server]socket= /tmp/
                ....
key_buffer = 16K
.               ....
[isamchk]key_buffer = 8M
[isamchk]sort_buffer_size = 8M
[myisamchk]key_buffer = 8M
[myisamchk]sort_buffer_size = 8M
```

```
MySQL-small extends Prim {
   client:port "3306";
   client:socket "/tmp/mysql.sock";
   server:port ATTRIB client:port;
   server:socket ATTRIB client:socket;
                ....
   key_buffer "16K";
                ....
   isamchk:key_buffer "8M";
   isamchk:sort_buffer_size ATTRIB
        isamchk:key_buffer;
   myisamchk:key_buffer ATTRIB
        isamchk:key_buffer
   myisamchk:sort_buffer_size AT-
TRIB
```

Figure 7. Snippet of MySQL original configuration and SF one.

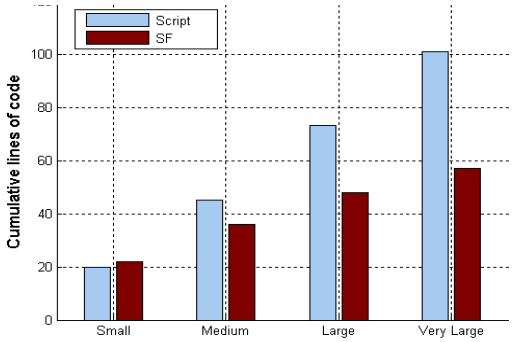

Figure 8. Number of cumulative lines of code to maintain as a function of complexity of deployed service.

bles with a non-language based approach. With a language based approach (e.g., SmartFrog), we need to introduce fewer lines of code that are either specific to a *medium* setup, or for tuning parameters whose values for a medium setup are different than those for a small setup. The rest of the configuration can be inherited from the small setup. Similar reasoning holds true as we move to introducing large and very large systems. As seen in the figure, the difference in the cumulative lines of code for maintaining up to a very large system using a language- and a non-language-based approach is quite significant.

Figure 9 shows the comparison of the number of configuration values to be edited in response to changes in the system. The changes introduced are port number changes, key/sort buffer size changes, read/write buffer size changes. In this comparison, we assume that the MySQL administrator is maintaining configurations for all the four kind of setups—simple, medium, large, and very large. The changes introduced therefore cause the administrator to edit configuration values across all four kind of setups. Without a language-based approach, the number of values to be edited would be equal to the number of places where the changed configuration parameter appears in the configuration files. With a language based approach, we use link references to link together related parameters. This feature of link references together with the ability to inherit values across the setups helps us to reduce the actual number of configuration values to be edited by the MySQL administrator in response to changes in the system. Since there are fewer values to modify, there are fewer opportunities for errors to be introduced into the system.

MySQL is simple compared to more complicated software systems, such as a supply chain. We believe that latter systems even more favor deployments at the higher level of abstraction (language- and model-based approaches).

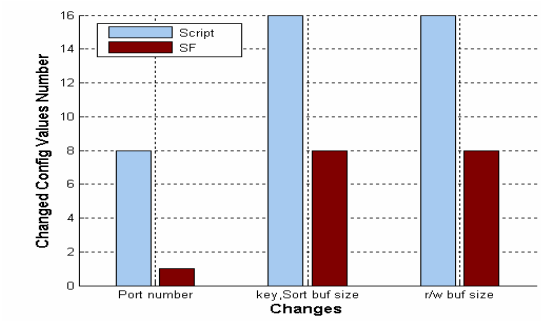

Figure 9. Number of changed configuration variables for changes in three different types of parameters.

4.3 Deployment and Installation Time

Figure 10(a) compares the results of overall time to deploy a 2-tier testbed using a language-based solution (SmartFrog 3.0) and a script-based solution (Nixes 0.3). The testbed consists of a Guestbook web application using JBoss 3.2.3 and PostgreSQL 7.4.1. The overall time can be decomposed into two parts. The first part is the environment setup time. For the script-based case, the environment setup time was the time required to install a JVM; for the language-based scenario, it also includes the SmartFrog daemon setup time. The second part, Figure 10(b), compares the application installation and ignition time. In this experiment we scaled the ratio of database to application servers from 1:1 up to 2:8.

In Figure 10(a), SmartFrog took more time compared to to Nixes tool. We attribute this to the additional setup time associated with starting the SmartFrog daemon. The daemon implements additional functionality needed to support a language-based solution, which is not needed for a script-based solution. This is a trade-off between increasing the level of automation which reduces administrator effort, and a more complex implementation leading to initial performance penalties. However, if we measured the time for application installation and ignition only, SF turned out to take less time than script. SmartFrog scales better than Nixes tool. The reason is that SmartFrog is able to exploit the maximum concurrency between different workflows. By comparison, Nixes does not provide an automatic workflow mechanism like SmartFrog. Users must manually control the order of installing and controlling applications.

5 Qualitative Evaluation

Table 2 provides a qualitative comparison between manual, script-, language-, and model-based deployment solutions in terms of automation, expressiveness, self-management, and barrier to first use.

Automation (self-management). Scripts are the first level that introduces automation of deployment through their ability to repeat a set of steps specified in a file and to form closed-closed control loops through events. Language-based solutions extend this by introducing lifecycle management through the use of dependencies (e.g., ordering of deployment and redeployment upon failures). Model-based solutions extend automation to the design-time, by enabling the use of models for the creation of deployment instantiations.

Self-healing. As an extension of self-management, self-healing enables a system to react to failures. Scripting has some ability to react to events and trigger handlers. Language based-solutions build on this by exploring dependencies (see the discussion on expressiveness) and can handle redeployment in a more sophisticated way. Finally, model-based solutions can change the deployed system design as a reaction to the failure.

Expressiveness (and ease of use)**.** Language-based approaches introduce benefits of inheritance, scopes of naming, and lazy evaluation for easier (re)configuration. This is of particular interest in deploying large scale, complex systems. Model-based approach contributes additional aspects of (meta-)model- and policy-based

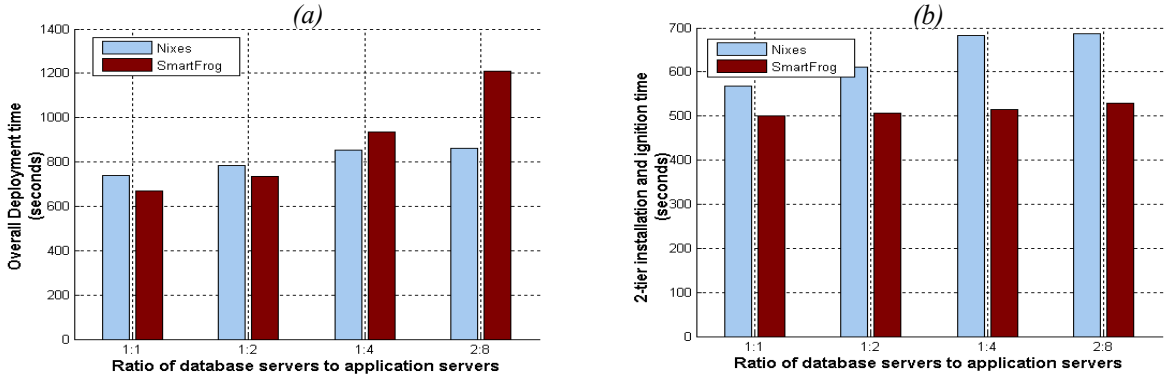

Figure 10. Deployment (a) and installation time (b) as a function of the scale (different ratio of DB and Application servers).

Table 2. High Level Comparison of Deployment Approaches

characteristics	*Deployment Approaches*			
	manual	***script-based***	***language-based***	***model-based***
solution based on	human language	configuration. files, scripts	declarative language	models & policies
automation	no	event-based closed-loops	+life cycle mgmt	+automated design
self-healing	no	minimal	redeployment + dependencies	+change design
expressiveness (see also Table 3)	no	partial	significant	complete
barrier to first use	none	low	high	very high

support that better captures run-time state and best practices. As a result, from manual to model-based, there is increasing level of re-use, correctness, and maintenance. See Table 3 for more details.

Barrier to First Use (time to learn, comprehend, etc.). The system administrator is the main user of deployment tools. Manual deployment usually requires no or minimal a priori knowledge. Scripts are relatively straightforward and require relatively low effort to start using them, although some of the script programs can become quite sophisticated. Language-based approaches requires a certain amount of education before one can use them. Developing encoding wrappers for the lifecycle management of services adds additional challenges. Finally, development of models introduces the largest barrier to the model-based approach. Front-end tools can somewhat alleviate the problem.

6 Summary and Future Work

We have compared manual, script-, language-, and model-based deployment solutions in a qualitative and a quantitative manner. Our results indicate that the number of steps and the number of lines of code are reduced with the introduction of more sophisticated deployment tools. Maintainability and documentability are proportional to the number of lines of code. Manageability is proportional to the number of the lines of code modified and number of steps added in response to changes in system configuration.

Table 4 illustrates the advantages and disadvantages of the spectrum of deployment approaches. The shaded regions denote the advantages. Ultimately, no universally optimal solution exists - the best approach is the one that closest matches the deployment need. When the number of deployed systems is small or systems' configurations rarely change, a manual solution is the most reasonable approach. For services with more comprehensive configuration changes, a script-based deployment strategy offers several benefits. In larger environments in which changes involve dependencies, language-based solutions are likely a better choice. If the changes also involve significant perturbations to the underlying service's design, the model-based approach is probably ideal. From the perspective of documentability, manual deployment offers poor support; scripts offer minimal support for the deploy-time changes; language-based approaches support incremental documentability based on inheritance and composition; and model-based approaches add runtime documenting by virtue of capturing all the changes in the deployed service's lifetime.

We would like to point out that there is no clear winner among the approaches to deployment—a best approach is the one that matches the deployment need closest. The four approaches are different in their nature, yet synergisestic. The manual approach is imperative; the script-based is automated imperative; the language based is declarative; the model-based is goal-driven. All four approaches are synergistic in the way that they increase the level of abstraction from manual to model-driven, yet they can each be accomplished in terms of predecessors. Any language-based feature can be accomplished by script-based solutions and any model can also be expressed in terms of languages. It is the ease of use and barrier to first use that determines the optimal choice. Complexity, dependencies, configuration space, and requirements for performance, availability, and scalability of services are dimensions that have a bearing on our results. The simpler the service configuration the more it lends itself to manual or script-based deployment. More complex services, with more requirements, are better suited for model-based deployment.

Table 3. Expressiveness of Deployment Approaches

characteristics	*Deployment Approaches*			
	manual	***script-***	***language-***	***model-based***
dependencies	no	yes	workflow-component	(meta-)models- & policy-based
constraints	no	yes	language	
inheritance	no	no	static	+dynamic
lazy evaluation	no	no	yes	arbitrary binding
re-use	no	some	significant	+best practices
correctness	poor	some	pre-deploy	+run-time
maintenance	poor	some	good	excellent

Table 4. Applicability of Deployment Approaches. Shaded area represent preferable choice

IT infrastructure characteristics	*Deployment Approaches*			
	manual	***script-***	***language-***	***model-based***
change	simple	config	dependency	design
repeat/scale	rare/small	many/large	many/large	many/large
complexity	simple	simple	complex	complex
documentability	none	deploy-time	+incremental	+run-time

We have not presented a rigorous evaluation of the model-based approach (prototyping or experimentation) as we have for the other deployment approaches. However, our qualitative comparison is useful, because it is based on our practical experience in using models. Similarly, we have omitted some other aspects of deployment, such as exceptions, without losing the generality of our conclusions. Exceptions are an important topic for the deployment, but addressing exceptions is either orthogonal to the comparison we make or well aligned with our results. For example, moving to the higher levels of abstraction may make it harder to understand some of the errors that have occurred at the lower levels of abstraction. However, similarly to the handling of network protocol stacks, failures should be analyzed at an appropriate level of abstraction and dealt with accordingly.

There is an opportunity to develop more elaborate quantitative comparison, potentially based on software metrics, such as those in software engineering [12]. We plan to pursue some of these approaches in the future. Although such techniques will increase our evaluation's precision, we expect the general conclusions to remain unchanged. We intend to continue using models to extensively automate the deployment and adaptation within complex systems and to increase scalability and ease of use. Integration with development tools, such as Eclipse will improve the ease of use, and decrease the barrier to first use. We also plan further examination of deployment in different underlying environments, such as PlanetLab, Grid, and Enterprise.

Acknowledgments

We thank the ICDCS reviewers and track vice-chair, Munindar P. Singh, for their insightful comments. We are also indebted to Paul Anderson, Martin Arlitt, Jamie Beckett, Patrick Goldsack, Michael Huhns, Steve Loughran, Jim Rowson, and Carol Thompson for reviewing the paper. Their comments improved contents and presentation. The scenario was offered by Julie Symons.

References

[1] Wilkes, J., Mogul, J., Suermondt, J., "Utilification," Proc. ACM European SIGOPS Workshop, September 2004.

[2] "Utility Computing," IBM Systems Journal special issue 43(1), 2004.

[3] Foster, I. et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration."

[4] www.novadigm.com.

[5] www.smartfrog.org, Open source directory.

[6] Humphreys, J. et al., "Service-Centric Computing: An Infrastructure Perspective, Outlook and Analysis," IDC #28934, March 2003.

[7] The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project, roc.cs.berkeley.edu.

[8] Prechelt, L., "An Empirical Comparison of Seven Programming Languages," IEEE Computer, vol 33, no 10, October 2000, pp 23-29.

[9] Engler, D., et al., "Checking System Rules Using System-Specific Programmer-Written Compiler Extensions," pp 1-16, Proc 4th USENIX OSDI, Oct 2000, San Diego, CA.

[10] Mèrillon, F., et al., "Devil: An IDL for Hardware Programming," Proc. 4th USENIX OSDI, pp 17-30, Oct 2000, San Diego, CA.

[11] Oracle Database 10g and Oracle 9i Database Manageability Comparison. Oracle Technic al Report. http://www.oracle.com/technology/products/manageability/database/pdf/twp03/oracle10g-oracle9i_manageability_comparison.pdf.

[12] Itzfeldt, W.D., "Quality Metrics for Software Management and Engineering," in Mitchell, R.J. (editor), "Managing Complexity in Software Engineering," IEE Computing Series 17, 1990, Short Run Press, Ltd., Exiter.

[13] Carzaniga, A., et al., "A Characterization Framework for Software Deployment Technologies," TR CU-CS-857-98, University of Colorado, Boulder, April 1998.

[14] Oppenheim, K., and MCormick, P., "Deployme: Tellme's Package Management and Deployment System," Proceedings of the Usenix IVth LISA Conference, December 2000, New Orleans, pp187-196.

[15] www.cfengine.org.

[16] Sapuntzakis, C., et al., Virtual Appliances for Deploying and Maintaining Software,", Proc. USENIX LISA'03 Conference, pp 181-1194, October 2003, San Diego, CA.

[17] Wang, Y.M., et al., "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," Proc. of the USENIX LISA'03, pp 159-172, October 2003, San Diego, CA. .

[18] Anderson, et al., "Technologies for Large-Scale Configuration Management," GridWeaver Technical Report, http://www.gridweaver.org/WP1/report1.pdf.

[19] http://www.aqualab.cs.northwestern.edu/nixes.html

[20] Goldsack, P., et al., "Configuration and Automatic Ignition of Distributed Applications", 2003 HP Openview University Association conference.

[21] Anderson, P., et al., "SmartFrog Meets LCFG: Autonomous Reconfiguration with Central Policy Control," Proc. USENIX LISA'03 pp 173-180, Oct 2003, San Diego, CA.

[22] Peterson, L, et al., "A Blueprint for Introducing Disruptive Technology, "PlanetLab Tech Note, PDN-02-001, July 2002.

[23] CDDLM Charter Document, https://forge.gridforum.org/projects/cddlm-wg