Automated Staging for Built-to-Order Application Systems

Galen S. Swint[1], Gueyoung Jung[1], Calton Pu[1], Akhil Sahai
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2005-161
September 19, 2005*

automation, utility computing, staging, TPC-W, benchmark, e-commerce

The increasing complexity of enterprise and distributed systems accompanying a move to grid and utility computing demands automated design, testing, deployment and monitoring of applications. In this paper, we present the Elba project and Mulini generator. The goal of Elba is creating automated staging and testing of complex enterprise systems before deployment to production. Automating the staging process lowers the cost of testing applications. Feedback from staging, especially when coupled with appropriate resource costs, can be used to ensure correct functionality and provisioning for the application. The Elba project extracts test parameters from production specifications, such as SLAs, and deployment specifications, and via the Mulini generator, creates staging plans for the application. We then demonstrate Mulini on an example application, TPC-W, and show how information from automated staging and monitoring allows us to refine application deployments easily based on performance and cost.

* Internal Accession Date Only
[1]Georgia Institute of Technology, Center for Experiment Research in Computer Systems, 801 Atlantic Ave., Atlanta, GA 30332, USA
Approved for External Publication

Automated Staging for Built-to-Order Application Systems

Galen S. Swint, Gueyoung Jung, Calton Pu
Georgia Institute of Technology,
Center for Experiment Research in Computer Systems
801 Atlantic Ave., Atlanta, GA 30332
swintgs@acm.org,
{helcyon1, calton}@cc.gatech.edu

Akhil Sahai
Hewlett Packard Laboratories
Palo Alto, CA
akhil.sahai@hp.com

http://www.cc.gatech.edu/systems/projects/Elba

***Abstract* – The increasing complexity of enterprise and distributed systems accompanying a move to grid and utility computing demands automated design, testing, deployment and monitoring of applications. In this paper, we present the Elba project and Mulini generator. The goal of Elba is creating automated staging and testing of complex enterprise systems before deployment to production. Automating the staging process lowers the cost of testing applications. Feedback from staging, especially when coupled with appropriate resource costs, can be used to ensure correct functionality and provisioning for the application. The Elba project extracts test parameters from production specifications, such as SLAs, and deployment specifications, and via the Mulini generator, creates staging plans for the application. We then demonstrate Mulini on an example application, TPC-W, and show how information from automated staging and monitoring allows us to refine application deployments easily based on performance and cost.**

I. INTRODUCTION

Managing the growing complexity of large distributed application systems in enterprise data center environments is an increasingly important and increasingly expensive technical challenge. Design, staging, deployment, and in-production activities such as application monitoring, evaluation, and evolution are each complex tasks in themselves. Currently, developers and administrators perform these tasks manually, or they use scripts to achieve limited *ad hoc* automation. In our previous work on automating application deployment, we have demonstrated the advantages of using higher level abstraction deployment languages such as SmartFrog (as compared to scripts) to specify application deployment process [1]. We also have built software tools that automate the deployment process, starting from high level resource requirement specifications [2]. In this paper, we focus on the automation of performance testing and validation (of an automatically generated configuration) during the staging process; this offers a way to anticipate, detect, and prevent serious problems that may arise when new configurations are deployed directly to production.

Staging is a natural approach for data center environments where sufficient resources are available for adequate evaluation. It allows developers and administrators to tune new deployment configuration and production parameters under simulated conditions before the system goes "live". However, traditional staging is usually approached in a manual, complex, and time consuming fashion. In fact, while the value of staging increases with application complexity, the limitations inherent to manual approaches tend to decrease the possibility of effectively staging that same complex application.

Furthermore, increasing adoption of Service-Level Agreements (SLAs) that define requirements and performance also complicates staging for enterprise-critical, complex, and evolving applications; again, the limitations of the manual approach become a serious obstacle. SLAs provide quantitative metrics to gauge adherence to business agreements. For service providers, the staging process allows them to "debug" any performance (or other SLA) problems *before* production and thereby mitigate the risk of non-performance penalties or lost business.

This paper describes the Elba project, the goal of which is to provide a thorough, low-cost, and automated approach to staging that overcomes the limitations of manual approaches and recaptures the potential value of staging. Our main contribution is the Mulini staging code generator which uses formal, machine-readable information from SLAs, production deployment specifications, and a test-plan specification to automate the staging phase of application development. We summarize the design and implementation of Mulini and include an early evaluation of Mulini generated staging code for staging a well-known application, TPC-W [3]. By varying the specifications, we are able to generate and compare several configurations and deployments of TPC-W of varying costs. Note that TPC-W is used as an illustrative complex distributed application for our automated staging tools and process, not necessarily as a performance measure of our hardware/software stack. For this reason, we will refer to TPC-W as an "application" rather than its usual role as a "benchmark."

The rest of the paper is organized as follows. Section II describes the challenges faced in the staging process. Section III summarizes the Elba project and our automated approach to application deployment and staging. Section IV describes the Mulini staging code generator. Section V presents an evaluation of Mulini code generation process and comparison of generated code from the application point of view. Section VI

This project is partially supported by Hewlett-Packard, NSF, and DARPA.

outlines related work and Section VII concludes the paper.

II. Challenges in Staging

A. Requirements in Staging

Staging is the pre-production testing of application configuration with three major goals. First, it verifies functionality, *i.e.*, that the system does what it should. Second, it verifies the satisfaction of performance and other quality of service specifications, *e.g.*, whether the allocated hardware resources are adequate. Third, it should also uncover *over*-provisioned configurations. Large enterprise applications and services are often priced on a resource usage basis. This question involves some trade-off between scalability, unused resources, and cost of evolution (discussed briefly in Section V). Other benefits of staging, beyond the scope of this paper, include the unveiling of other application properties such as its failure modes, rates of failure, degree of administrative attention required, and support for application development and testing in realistic configurations.

These goals lead to some key requirements in the successful staging of an application. First, to verify the correct functionality of deployed software on hardware configuration, the staging environment must reflect the reality of the production environment. Second, to verify performance achievements the workload used in staging must match the service level agreement (SLA) specifications. Third, to uncover potentially wasteful over-provisioning, staging must show the correlation between workload increases and resource utilization level, so an appropriate configuration may be chosen for production use.

These requirements explain the high costs of a manual approach to staging. It is non-trivial to translate application and workload specifications accurately into actual configurations (requirements 1 and 2). Consequently, it is expensive to explore a wide range of configurations and workloads to understand their correlation (requirement 3). Due to cost limitations, manual staging usually simplifies the application and workload and runs a small number of experiments. Unfortunately, these simplifications also reduce the confidence and validity of staging results.

Large mission-critical enterprise applications tend to be highly customized "built-to-order" systems due to their sophistication and complexity. While the traditional manual approach may suffice for small-scale or slow-changing applications, built-to-order enterprise applications typically evolve constantly and carry high penalties for any failures or errors. Consequently, it is very important to achieve high confidence during staging, so the production deployment can avoid the many potential problems stemming from complex interactions among the components and resources. To bypass the difficulties of manual staging, we advocate an automatic approach for creating and running the experiments to fulfill the above requirements.

B. Staging Steps

In automating the staging process, we divide staging into three steps: design, deployment, and the actual test execution. We present short descriptions of our previous work on the first two steps, design and deployment. The automation tools of the first two steps produce automatically generated and deployable application configurations for the staging environment. The third step, execution, is to generate and run an appropriate workload on the deployed configuration and verify the functionality, performance, and appropriateness of the configuration.

In the first step, design, the entire process starts with a machine-readable specification of detailed application design and deployment. Concretely, this has been achieved by Cauldron [4], an application design tool that generates system component specifications and their relationships in the CIM/MOF format (Common Information Model, Managed Object Format). Cauldron uses a constraint satisfaction approach to compute system designs and define a set of workflow dependencies during the application deployment. Readers interested in the design specification step are referred to Cauldron [4] and other similar tools.

The second step in the automated staging process is the translation of the CIM/MOF specification into a concrete configuration. Concretely, this is achieved by ACCT [2] (Automated Composable Code Translator). In a multi-stage translation process, ACCT transforms the MOF specification through several intermediate representations based on XML and then finally into various Java classes and interfaces and SmartFrog [5], a configuration specification language. The SmartFrog compiler accepts a specification to generate the Java code for actually deploying the application configuration in the staging environment. Readers interested in the automated deployment step are referred to papers on ACCT [2] and the evaluation of SmartFrog [1] as a deployment tool.

Next, we discuss step three, the technical challenges of the workload generation and execution process and the particular focus of this paper and the Elba project.

C. Automated Staging Execution

The third step of staging is the automation of staging execution. Automated execution for staging is assembled from three main components: (1) a description mapping the application to the staging environment, (2) the input to the application – the workload definition, and (3) a set of application functionality and performance goals defined on the workload.

The application description (first component) can be borrowed or computed from the input to the first and second steps in which the application has been formally defined. However, environment dependent parameters may make re-mapping the execution parameters of an application from a deployment to staging environment a non-trivial task. Location sensitive changes include obvious location strings for application components found in the design documents, but non-obvious within the application are references to services the application may require to execute successfully, such as ORB (Object Request Broker) naming services, web URL's, external services, or database locations.

For the staging workload definition (the second component), it is advantageous to reuse the workload of the production environment if available. The use of a similar workload increases the confidence in staging results. Also, by mapping

the deployment workload into the staging environment automatically, the study of the correlation between workload changes and resource utilization in different configurations is facilitated because the low-cost, repeatable experiments encourage the testing of multiple system parameters in fine-grain steps. The repeatability offered by an automated system provides confidence in the behavior of the application to a presented workload as the application evolves during development and testing.

The third component is specification and translation of application functionality and performance goals into a set of performance policies for the application. This is a "management task" and the main information source is the set of Service Level Agreements (SLAs). Typically, SLAs explicitly define performance goals such as "95% of transactions of Type 1 will have response time under one second". These goals, or Service Level Objectives, can serve as sources for deriving the monitoring and instrumentation code used in the staging process to validate the configuration executing the intended workload. Beyond SLAs, there may also be defined performance requirements that derive not from the service customer but from policies of the service provider.

The automated translation processes of each single component and of all three components are significant research challenges. In addition to the typical difficulties of translating between different levels of abstraction, there is also the same question of generality applicable to all application-focused research projects: how to generalize our results and apply our techniques to other applications. While we believe our approach to be general, as shown by this project as well as previous successful experiences [6][7][8][9], we consider the work reported in this paper as an early experiment in automated staging that already reveals as many interesting research questions as answers.

III. ELBA

A. Overall Approach and Requirements

As summarized in Section II.B, we process three major components when automating staging: the application, the workload, and performance requirements. One of the main research challenges is the integrated processing of these different specifications through the automated staging steps. Our approach (described in more detail in Section IV) is to create and define an extensible specification language called TBL (the testbed language) that captures the peculiarities of the components as well as the eventual target staging environment. The incremental development of TBL and associated tools (enabled by the Clearwater architecture [9]) is the cornerstone of the Elba project.

We mention some of the research goals (partially) addressed in this paper, as a motivation for the architecture shown in Figure 1. Research goals for the specification of applications and their execution environments include: auto-

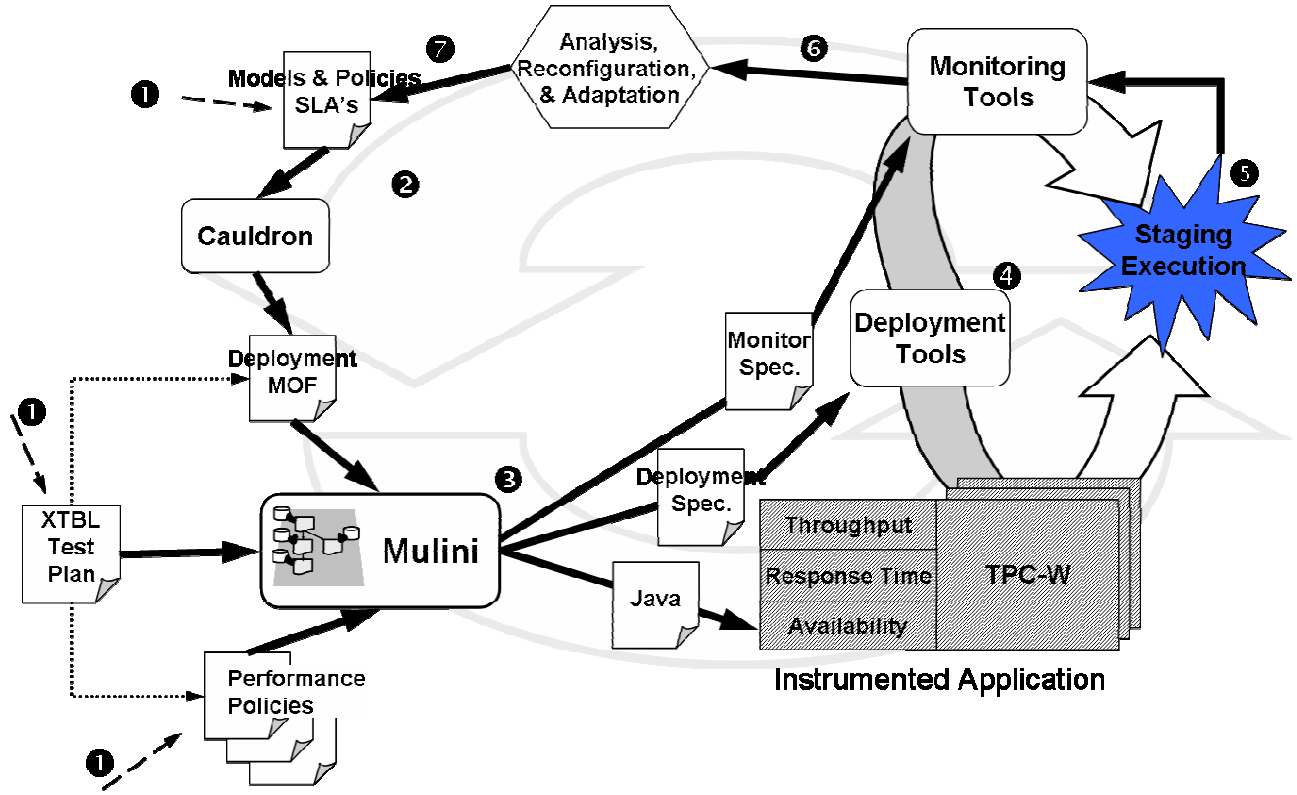

Figure 1. The goal of the Elba is to automate the circular, repetitive process of staging by using data from deployment documents and bringing together automation tools (rounded boxes). The staging cycle for TPC-W is as follows (from the upper-left, counter-clockwise): 1) Developers provide design-level specifications of model and policy documents (as input to Cauldron) and a test plan (XTBL). 2) Cauldron creates a provisioning and deployment plan for the application. 3) Mulini generates staging plan from the input components referred to from XTBL (dashed arrows). 4) Deployment tools deploy the application, monitoring tools to the staging environment. 5) The staging is executed. 6) Data from monitoring tools is gathered for analysis. 7) After analysis, developers adjust deployment specifications or possibly even policies and repeat the process.

mated re-mapping of deployment locations to staging locations; creation of consistent staging results across different trials; extensibility to many environments and applications. Research goals on the evaluation of application quality of service (QoS) include:

1. Appropriate QoS specifications and metrics that capture SLAs as well as other specification methods.
2. Instrumentation for monitoring desired metrics. This automates the staging result analysis.
3. Matching metrics with configuration resource consumption. An SLA defines the customer-observable behavior (*e.g.*, response time of transactions), but typical system resources (*e.g.*, CPU usage of specific nodes) are not.
4. Maintaining a low run-time overhead (*e.g.*, translation and monitoring) during automated staging.
5. Generating reports that summarize staging results, automatically identifying bottlenecks as appropriate.

B. Summary of Tools

Figure 1 shows the cyclical information and control flow in Elba to support the automated staging process, using TPC-W as an illustrative application. On the left upper corner is the first step, implemented by the Cauldron design and provisioning tool, a constraint-based solver that interprets CIM application descriptions scenarios to create application deployment scenarios [1]. This allows application developers to leverage the inherent parallelism of the distributed environment while maintaining correctness guarantees for startup. For example, the database data files are deployed before the database is started and the image files are deployed before the application server is started.

Future Cauldron development as part of Elba will extend it to incorporate SLA information in the provisioning process. This move will allow Cauldron's formal constraint engines to verify the SLAs themselves and incorporate SLA constraints into the provisioning process. These SLAs can then be converted into XML-based performance policy documents for Mulini, as we have illustrated in Figure 1.

The second step and third steps have been merged into the Mulini generator in Figure 1. The automated deployment generator tools such as ACCT generate the application configuration from the CIM/MOF specification generated by Cauldron. ACCT output (specifications for automated deployment) is executed by deployment tools such as SmartFrog for actual deployment. The new component described in this paper is the Mulini code generator for staging process. Mulini provides the implementation framework that realizes the creation of staging deployments from the TBL language. Elba's approach to the entire staging phase of an application's lifecycle iterates (the circular arrows) as feedback from staging execution is re-incorporated in the design phase, where a new design may be generated and tested again in the staging environment.

IV. MULINI

A. Overview

Mulini maps a high-level TBL description of the staging process to low-level tools that implement the staging process. TBL is an in-progress language, and its current incarnation is an XML format called XTBL. Eventually, one or more human-friendly, non-XML formats such as GUI tools or script-like languages will be formulated, and subsequently XTBL will be created automatically from those representations.

The use of XML as a syntax vehicle for the code generator stems from our experiences building code generators around the Clearwater code generation approach [9]. If using traditional code generation techniques that require grammar specification and parser creation, a domain specific language might require a great deal of maintenance with each language change. Our experience with Clearwater generators for problems in distributed information flow [8] and in automatic, constrained deployment of applications [2] has shown these generators to be very flexible with respect to changing input languages and very extensible in their support of new features at the specification level and at the implementation level.

We describe the Clearwater approach to creating processors for domain-specific languages, next, and follow that discussion with the Clearwater-based Mulini in particular.

B. The Clearwater Code Generation Approach

The Clearwater approach to code generation is to use an XML-based input and intermediate representation and then perform code generation from XSLT templates. The use of XML allows for flexible and extensible input formats since defining a formal grammar may be deferred to later in the language development process. During the generation process, XML documents are used as the intermediate representations of the domain language code and XML is used to contain generated code fragments; they are stored in-memory as DOM (the Document Object Model) trees; the DOM interface is a W3C standard for working with XML documents [10]. XSLT allows template driven generation of the target code; invocations of XSLT templates can be one of two ways: either as explicit calls to a specific XSLT templates or as a pattern match triggered by the input specification. Since XSLT is compiled at runtime, extending such generators to new targets is easy – one simply adds a new XSLT template and inclusion reference. Such extensions may take advantage of specific features in the target platform, or extend the generator to entirely different target platforms.

Adding support for new features found in the domain level languages that serve as input is also straightforward. First, new tags codifying the new domain-expertise are added to the specification document; then, XSLT is written to pattern-match against the new tags to generate code in the proper target language. Because XSLT makes use of XPath, it supports structure-shy operations on the specification tree; importantly, these additional tags do not break program generation (or only require minimal changes) for the original template code. This low barrier to change encourages and supports language evolution which is particularly valuable when developing a new domain specific language.

One of the biggest advantages of the Clearwater approach is its ability to support multiple implementation (or target) platforms and platforms that require heterogeneous language output. As an example of heterogeneous target support, The

Infopipes Stub Generator supports simultaneous generation and source-level weaving of C and C++. C and C++ each supported more than one communication library with, again, the ability to generate to multiple communication layers during a single invocation. The ACCT generator, which is re-used as a component in Mulini, supports Java source code and SmartFrog specifications for SmartFrog deployment.

Furthermore, XSLT's support for on-demand parsing of XML documents allows auxiliary specifications to be consulted very easily during the code generation process [6]. As our staging generator evolves, this will allow conversion information to be stored as XML documents. For instance, one document might describe how or which deployment machines that host databases should be mapped onto machines available for staging. In this instance, an IT department or facility might define a single such document thus avoiding the need to include it with each Mulini invocation while also avoiding the direct inclusion mutable data within the generator or generator templates.

C. Code Generation in Mulini

As mentioned earlier, the staging phase for an application requires three separate steps: design, deployment, and execution. Again, requirements for automated design are fulfilled by Quartermaster/Cauldron and deployment is fulfilled by ACCT. Mulini's design wraps the third step, execution, with deployment to provide an automated approach to staging.

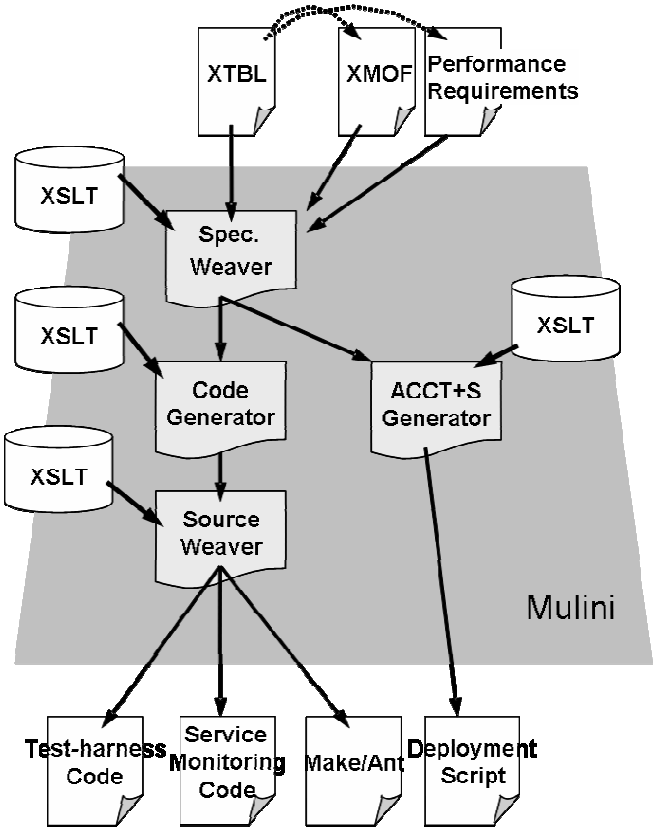

Figure 2. The grey box outlines components of the Mulini code generator. Initial input is an XTBL document, from which it retrieves references to the performance requirements and the XMOF files and then loads and weaves those three filetypes to create the XTBL+ document. XTBL+ is the document from which information is directly retrieved during generation.

Mulini has four distinct phases of code generation: specification integration, code generation, code weaving, and output. In the current, early version, these stages are at varying levels of feature-completeness. Because of this, we will describe all features to be included in near term releases, and then at the end of this section we will briefly describe our current implementation status. Figure 2 illustrates the generator and relationships between its components. The design of these components will be described in the remainder of this section.

In specification integration, Mulini accepts as input an XML document that contains the basic descriptors of the staging parameters. This document, the XTBL variant of TBL introduced to earlier, contains three types of information: the target staging environment deployment information should be re-mapped to, a reference to a deployment document containing process dependencies, and references to performance policy documents containing performance goals.

Mulini automatically integrates these three documents into a single XTBL+ document. The XTBL+ is organized with the same structure as the XTBL, but leverages XML's extensibility to include the deployment information and performance requirements information as new elements. The process to weave these documents:

1. Load, then perform XML parsing, and construct a DOM tree of the XTBL document. This DOM tree is copied to become the core for the new XTBL+ document.
2. Retrieve all references to deployment documents (XMOF documents) from the XTBL document. There may be more than one deployment document since deployment of resource monitors may be specified separately from deployment of.
3. Load any referenced deployment documents and incorporate their code onto the XTBL+ document. At this point, the deployment directives are re-targeted to the staging environment from the deployment environment, *e.g.*, machine names and URLs must be remapped. In the XTBL document, each deployable unit is described with an XMOF fragment; each of the targeted deployment hardware is also described with an XMOF fragment.
4. Load the performance requirements documents. Mulini maps each performance requirement mapped onto its corresponding XMOF deployment component(s). This yields an integrated XTBL+ specification.

Figure 12 of Appendix B illustrates the three source documents and the woven XTBL+ result.

Following the specification weaving, Mulini generates various types of source code from the XTBL+ specification. To do so, it uses two sets of code generators. The first of these code generators is the ACCT generator which has been used to generate SmartFrog deployments of applications [2]. We have extended ACCT to support script-based deployments and so will refer to it here as ACCT+S. ACCT+S accepts the XTBL+ document, extracts relevant deployment information, generates the deployment scripts, and writes them into files.

The second code generator generates staging-phase application code, which for TPC-W is Java servlet code, shell scripts for executing monitoring tools, and Makefiles. The

TPC-W code includes test clients that generate synthetic workloads, application servlets, and any other server-side code which may be instrumented. If source code is available, it can be added to Mulini's generation capabilities easily. We provide a description of the process to import a TPC-W servlet into Mulini for instrumentation later in this section. Mulini, in this phase, also generates a master script encapsulating the entire staging execution step, compilation, deployment of binaries and data files, and execution commands, which allows the staging to be executed by a single command.

We mentioned that staging may require the generation of instrumentation for the application and system being tested. Mulini can generate this instrumentation: it can either generate tools that are external to and monitor each process through at the system level (*e.g.*, through the Linux `/proc` file system), or it may generate new source code in the application directly.

The source weaver stage of Mulini accomplishes the direct instrumentation of source code by weaving in new Java code that performs fine grain instrumentation on the base code. The base code may be either generic staging code generated from the specification inputs, or it may be application-specific code that has been XSLT-encapsulated for Mulini weaving. To achieve source weaving, we use an XML-weaving approach similar to that of the AXpect weaver [8]. This weaving method consists of three major parts: inserting XML semantic tags on the template code, using XSLT to identify these tags and insert new code, and adding instructions to the specification that direct which aspects are to be woven into the generated code.

Practically, of course, this means that the application code must be included in the generation stream. Fortunately, the use of XML enables this to be done quite easily. Source code can be directly dropped into XSLT documents and escaping can be automatically added for XML characters with special meaning, such as ampersand and '<'. Again, at aspect weaving time, this code can be directed to be emitted, and semantic tags enable the weaver to augment or parameterize the application code.

As mentioned earlier, the instrumentation code may derive from SLAs that govern the service expectations of the deployed application. These service level documents contain several parts. First, they name the parties participating in the SLA as well as its dates of enforcement. Then, they provide a series of service level objectives (SLO's). Each SLO defines a metric to be monitored, location at which it is to be measured, conditions for monitoring (start and stop), and any subcomponents that comprise that metric. For instance, the "ResponseTime" metric comprises response time measurements for each type of interaction in the TPC-W application.

At this time, most of the Mulini functionality has been implemented. This includes generation of scripts, modification of ACCT into ACCT+S, source-level weaving, and the creation of instrumentation aspects for monitoring applications. Specification weaving of XTBL, a Web Service Management Language (WSML) document of performance policies and an XMOF document is currently partially implemented.

Near-term plans are to add to Mulini code to generate scripts that collect data from monitoring tools. This data will then be automatically placed in files and data analyzers generated. The analyzers will provide automatic assessment of whether the performance policies (and by extension the SLAs) have been met as well as how the system responds to changing workloads. Also, we will extend specification weaving to allow multiple performance policy and deployment documents.

V. EVALUATION

A. *Application Scenario: TPC-W*

As an early experiment, we have chosen a well known application, the TPC-W benchmark, a transactional web e-commerce benchmark from the Transaction Processing Performance Council – TPC, as an exemplar for our automated staging approach. As mentioned before, we use TPC-W as an illustrative mission-critical enterprise application, not for performance comparison of different platforms. We include a short summary of the application to make the paper self-contained.

The TPC-W bookstore application was conceived by the TPC to emulate the significant features present in e-commerce applications [3][11]. TPC-W is intended to evaluate simultaneously a vendor's packaged hardware and software solution – a complete, e-commerce system. Normally, benchmarks provide a preliminary estimate for vendors to compare system performance, but the TPC-W application also suggests measuring the resource utilization of subcomponents of the system under test, a concept which matches our goals in staging to uncover system behavior.

The TPC-W application comprises two halves: the workload, which is generated by emulating users, and the system-under-test (SUT), which is the hardware and application software being tested. Our system re-uses and extends the software provided as part of the PHARM benchmark [12] and studies of performance bottlenecks and tuning in e-commerce systems [1][13]. For the system we test, we employ a combined Java web/application server, using Apache Tomcat, and a database server, in this case MySQL.

In the TPC-W scenario, customers' navigation of the web pages of an online bookstore is simulated by remote emulated browsers (EB's). Each emulated browser begins at the bookstore's home page where it "thinks" for a random time interval after which the EB randomly chooses a link to follow. These link choices are from a transition table in which each entry

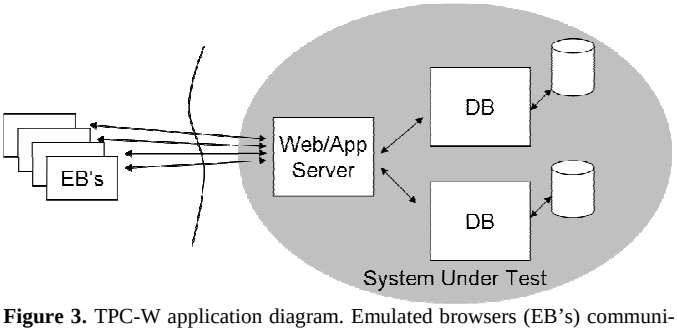

Figure 3. TPC-W application diagram. Emulated browsers (EB's) communicate via the network with the web server and application server tier. The application servers in turn are backed up by a database system. This is a simplified diagram, and commercial implementations for benchmark reporting may contain several machines in each tier of the system under test as well as complex caching arrangements.

represents the probability *p* of following a link or going offline. The specification provides three different models (that is, three different transition tables) each of which emulates a different prevailing customer behavior.

For the application, there are two primary metrics of concern: requests served per second, which is application throughput; and response time, which is the elapsed time from just before submitting a URL GET request to the system until after receiving the last byte of return data. The result of a TPC-W run is a measure called WIPS – Web Interactions per Second. There are several interaction types, each corresponding to a type of web page. For instance, a "Best Seller" interaction is any one of several links from the home page that leads to a best seller list such as best sellers overall, best selling biographies, or the best selling novels.

One test parameter is to select the number of concurrent clients (number of EB's). Varying the number of concurrent clients is the primary way of adjusting the number of web interactions per second submitted to the system. The most influential parameter is the number of items in the electronic bookstore offered for sale. This particular variation is also called the *scale* of the experiment and may be 1000, 10,000, or 100,000 items. Since many of the web pages are generated views of items from the database, normal browsing behavior can, excluding caching, slow the performance of the site. As an application parameter, scale level has the greatest impact on the performance of the system under test [11].

B. TPC-W Implementation

We chose TPC-W v1.8 due to its widespread use in research and the availability of a reference implementation over the newer 2.0 benchmark which has even yet to be fully ratified by TPC. Our evaluations utilized only the "shopping" browsing model. This model represents the "middle ground" in terms of interaction mix when compared to the "order" and "browsing" models.

The implementation of the TPC-W bookstore is as Java servlets using the Apache project's Jakarta Tomcat 4.1 framework, communicating with a backend MySQL 4.0 database both running on Linux machines using a 2.4-series kernel. For all of our evaluations, the database, servlets, and images are hosted on local drives as opposed to an NFS or storage-server approach. As in other TPC-W studies, to speed performance, additional indexes are defined on data fields that participate in multiple queries. Connection pooling is also utilized to re-use database connections between the application server and the database.

During our testing, we employ two classes of hardware. Machines labeled "low-end," or "L," are dual-processor P-III 800 MHz with 512 MB of memory, and we assign them an approximate value of $500 based on straight-line depreciation from a "new" price of $2000 four years ago. "High-end" or "H" machines are dual 2.8GHz Xeon blade servers with hyperthreading enabled and 4GB of RAM, and we have assigned them an approximate value of $3500 each based on current replacement cost. Assigning cost values to each server is a convenient proxy for the cost of a deployment configuration. For instance, we can assign a "2H/L" configuration of two high-end servers and one low-end server an approximate value of 2 × $3500 + $500 or $7500.

C. Automatic Staging for TPC-W

The first step in preparing TPC-W for automatic staging is to create the clients and select service-side tools for monitoring resource usage. We wrote new clients and then encapsulated them in XSLT templates to support generation of them by Mulini and therefore parameterization through TBL. We then created the deployment documents and TBL specification that describe TPC-W staging.

We created a MOF file containing the CIM description of the hardware and software needed to support the TPC-W application. Once given the MOF description, Cauldron uses the MOF to map the software onto hardware and produce a workflow for deployment of the applications. This MOF file is translated into an XML document using a MOF-to-XML compiler resulting in XMOF as described in [2].

Next, we created the SLAs for the TPC-W performance requirements. While future incarnations of Mulini will rely on documents *derived* from the SLAs, we currently re-use the SLAs as a convenient specification format, WSML, for encoding performance requirements pertaining both to customer performance data and to prescribe metrics for monitoring as performance policies pertaining only to the system under test. There are 14 types of customer interaction in the TPC-W application. Each of these interactions has its own performance goal to be met which as detailed in Table 2 of Appendix A. For instance, to meet the SLA for search performance, 90% of search requests must complete the downloading of search results and associated images in 3 seconds.

Finally, the third specification document is the TBL specification of the staging process. TBL is directly convertible by hand to XTBL, an XML based format, suited as input for Mulini. TBL includes information that relates the staging test to performance guarantees and the specific deployment workflow. Example excerpts from each of the three specifications documents can be seen in Figure 12 of Appendix B.

D. Overhead Evaluation

Our first evaluation is designed to show that there is reasonable overhead when executing staging tests that are generated. This is important because too high overhead could reduce the relevance of staging results. To evaluate the generated benchmark, we run the original reference implementation (non-Mulini generated) first to provide a baseline for performance. Since we can not glean an accurate understanding of overhead when the system runs at capacity, we use much lower numbers of concurrent users. These two tests, shown in Figure 4 and Figure 5, are executed first on low-end hardware with 40 concurrent users and then on the high-end hardware with 100 concurrent users. We use `sar` and `top` to gather performance data during the execution of the application of both the Mulini generated variant and the reference implementation. From this evaluative test, we see that the Mulini generated code imposes very little performance overhead in terms of resource usage or response time on application servers or database servers in both the L/L and H/H cases.

However, our target performance level for the TPC-W application is 150 concurrent users. Since we have now established that the generative techniques employed impose little overhead (<5%), we proceed to measurements based on the application's formal design parameters.

E. Tuning TPC-W: Mulini in Use

We generate Mulini variants of TPC-W to illustrate the utility of generated staging as compared to an "out-of-the-box" TPC-W. In fact, we wrap the performance monitoring tools of the primary experiment for re-use in our generated scenarios. This way, they can become part of the automatic deployment of the TPC-W staging test. We also begin recording "average query times" for the best seller query. Being one of the more complex queries, it was shown in our initial test also to be longer running than most of the other queries.

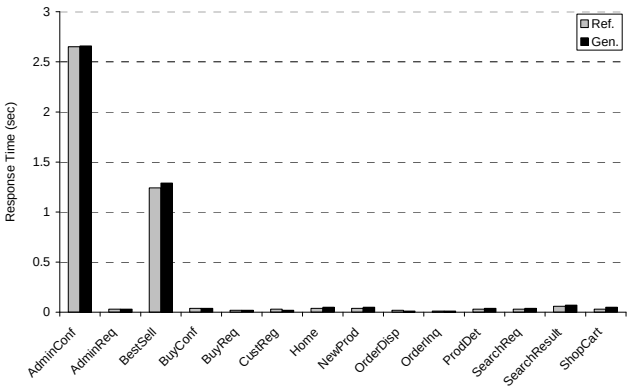

Figure 4. L/L reference comparison to gauge overhead imposed by Mulini with 40 concurrent users. In all interactions, the generated version imposes less than 5% overhead.

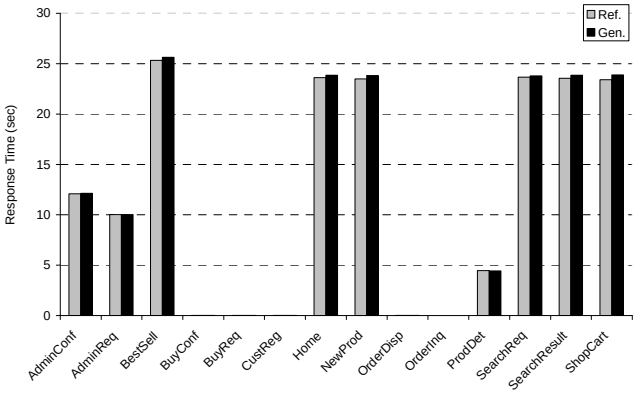

Figure 5. H/H reference comparison to gauge overhead imposed by Mulini with 100 concurrent users. For all interactions, the generated version imposes less than 5% overhead.

Wrapping the performance tools we use for monitoring performance is reasonably straightforward for a Clearwater-style generator. First, we construct command-line scripts that execute the tools and then wrap these scripts in XSLT. This process that consists of adding file naming information and escaping special characters; we add these XSLT templates to the main body of generator code. Once this is completed, we can easily parameterize the templates by replacing text in the scripts with XSLT statements that retrieve the relevant information from the XTBL+ document.

We perform this same technique to escalate database servlet code into the generator templates for direct instrumentation of their source. This is followed by adding an XML marker to denote a joinpoint in the code around the database query execution that we wish to monitor. We write an XSLT aspect template with XPath that selects the marker and inserts timing code that implements measurement of the query.

Once aspect writing and template extension is complete, we can begin executing our application staging and tuning experiments. Figure 6 shows the level of QoS satisfaction as specified by SLAs. Most of the high-end configurations perform well, while the low-end configurations have some problems. The raw data for this graph is available in Table 2 and Table 3 in Appendix A.

We focus on the BestSeller transaction to illustrate the differences among the configurations. Figure 7 zooms into Figure 6's third column from the left, showing very poor performance of the L/L configuration, very good performance of 2H/L and 2H/H (more than 90% satisfaction), with the H/L and H/H configurations in between (above 60% satisfaction) but still failing to meet the SLA.

To explain the differences in performance shown in Figure 7, we studied the response time and throughput of the configurations via the direct instrumentation of the database servlet. The average response time is shown in Figure 8, where we see

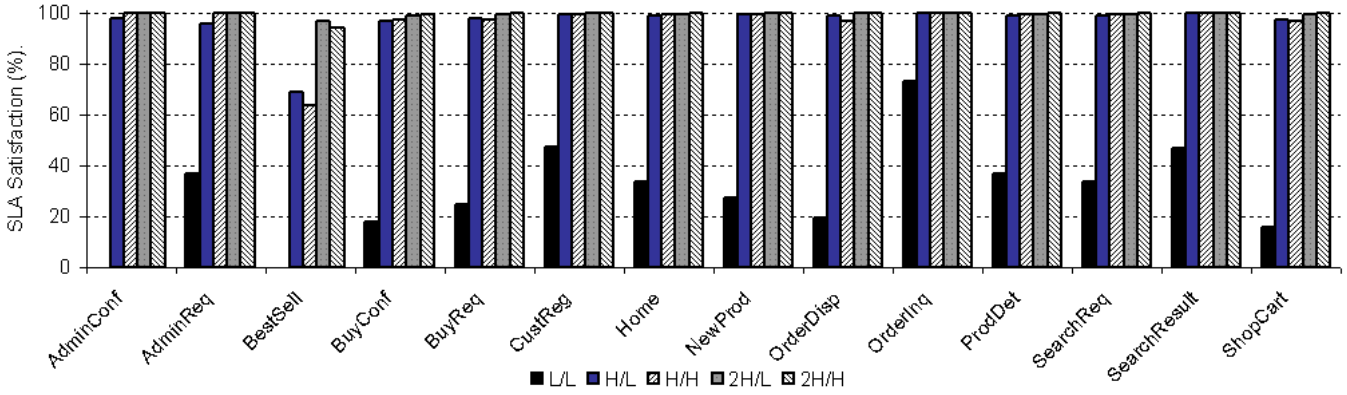

Figure 6. Summary of SLA satisfaction.

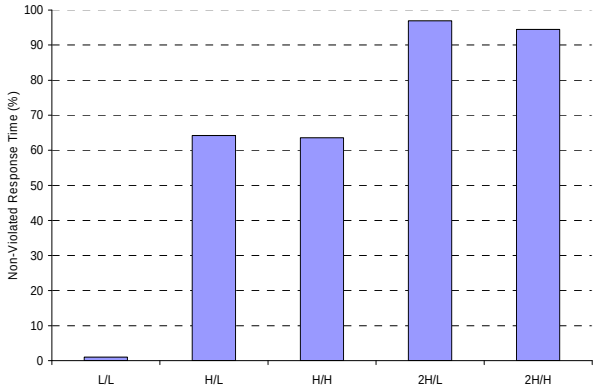

Figure 7. SLA Satisfaction of BestSeller

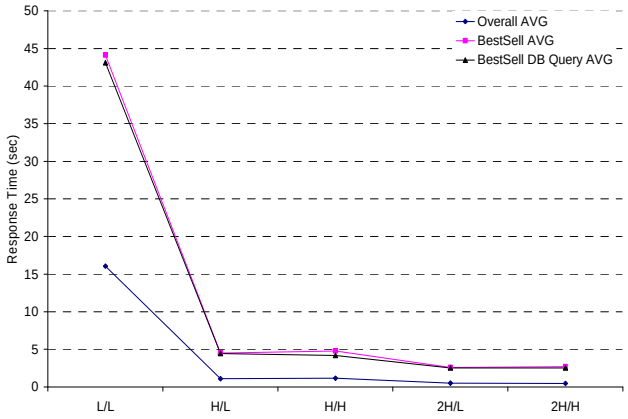

Figure 8. BestSeller average response time.

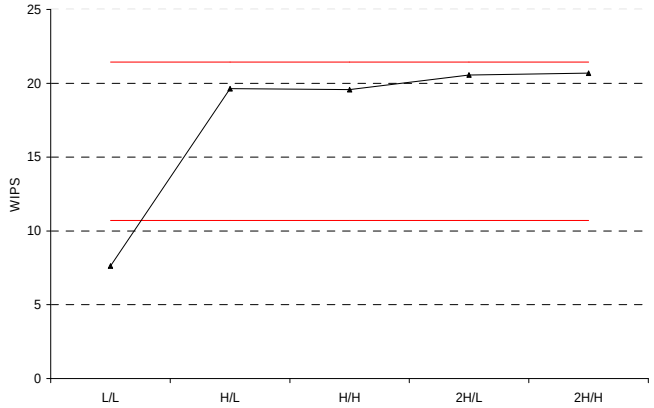

Figure 9. System throughput (WIPS). Lines above and below the curve demarcate bounds on the range of acceptable operating throughput based on ideal TPC-W performance as a function of the number of EB's.

a clear bottleneck for L/L configuration. In addition, we measured the response time of a critical component of BestSeller interaction, the BestSeller database query. Figure 8 shows that the response time of the BestSeller transaction is almost entirely due to the BestSeller Database Query, demonstrating the database to be the bottleneck. This finding is confirmed by Figure 9, which shows a marked increase in WIPS throughput when the database is moved to more powerful (and more expensive) hardware.

To migrate to more powerful hardware, we simply re-mapped the deployment to a high-end machine and re-deployed the staging and monitoring code automatically.

From this data, we can observe that average response time of the query from the servlet to the database remains fairly long, indicating that even though the application server on low-end hardware is strained in terms of memory usage, the database remains the bottleneck even in cases of high-end hardware. Fortunately, MySQL allows database replication out-of-the-box. While this does not allow all database interactions to be distributed, it does allow "select" queries to be distributed between two machines, and these queries constitute the bulk of the TPC-W application. A straightforward re-write of the database connection code expands TPC-W to take into account multiple databases in the application servlet; this is followed by adding and modifying deployment to recognize the replicated database server. Using this method to allow the 2H/L and 2H/H cases, we were able to create a system within our performance specification. To understand the operating differences between deployments, it is instructive to examine the resource utilization reported by our monitoring tools.

First, for the database server we note that while it uses only about 60% of the CPU, the *system* memory utilization consistently gets close to 100% due to filesystem caching as shown in Figure 10. This was ascertained by generating a script that measured actual process memory usage and comparing this data with the overall system memory usage reported by the kernel. As the daemon process for the servlets remained constant in size, it was apparent that application activity was exerting pressure through the operating system's management of memory. The memory and CPU utilization of the database server is plotted below in, showing the memory bottleneck in addition to the CPU bottleneck of the database server in the L/L configuration. Note that we have included the approximate asset cost for each deployment

We record resource usage for the application server, too, in Figure 11 again including approximate asset cost for the deployments. The figure shows consistent CPU and memory utilizations for high-end and low-end configurations. At around 20% CPU utilization and 15% memory utilization, our results indicate the low-end hardware is a viable application server choice for the target workload of 150 concurrent users, since the high-end configuration uses less than 5% of CPU resources with very little memory pressure (evidenced by system memory utilization being below 80%, a number which in our experience is not atypical for a Linux system under only light load).

At this point, our automated generation has allowed rapid testing that begins to provide enough information on which system administrators may base deployment decisions. Referring back to the previous two figures, the TPC-W "application provider" now has a clearer picture of the cost of deploying his service; while technically two configurations do meet the SLAs, we note that there is a choice for the final configuration. The administrator can either choose between a deployment at lower cost (2H/L) with less growth possibility, higher cost with ample resource overhead (2H/H), or request another round of staging (automatically) to find a better mix of the

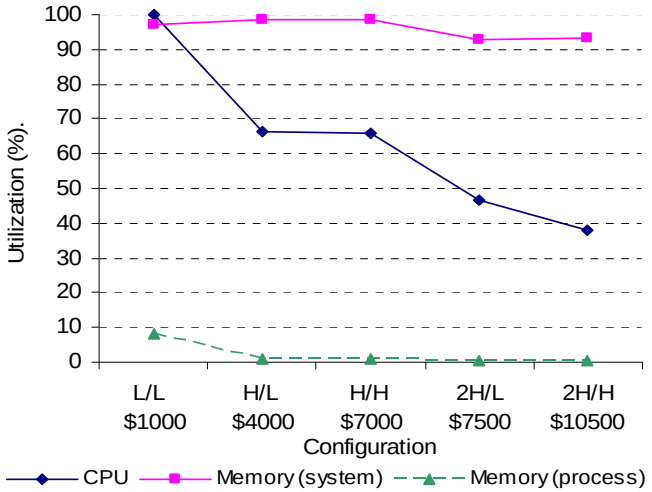

Figure 10. Database server resource utilization. The kernel's file system caching creates the spread between system memory utilization and the database process's memory utilization.

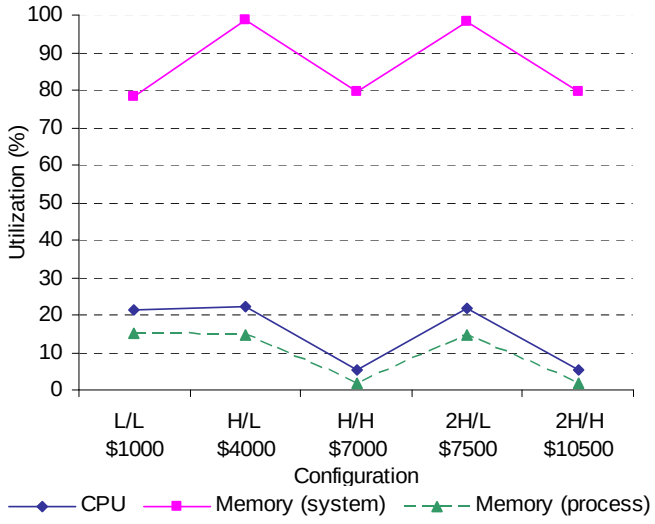

Figure 11. Application server resource utilization.

three machines that fulfill the SLAs.

VI. Related Work

Other projects address the monitoring of running applications. For instance, Dubusman, Schmid, and Kroeger instrument CIM-specified enterprise Java beans using the JMX framework [16]. Their instrumentation then provides feedback during the execution of the application for comparing run-time application performance to the SLA guarantees. In the Elba project, our primary concern is the process, staging, and follow-on data analysis that allows the application provider to confirm *before deployment* that the application will fulfill SLAs. Furthermore, this automated staging process allows the application provider to explore the performance space and resource usage of the application on available hardware.

Several other papers have examined the performance characteristics and attempted to characterize the bottlenecks of the TPC-W application. These studies generally focused on the effects of tuning various TPC-W parameters [11], or on the bottleneck detection process itself [13][15]. The paper takes TPC-W, not as the benchmark, however, but as a representative application that allows us to illustrate the advantages of tuning applications through an automated process with feedback.

The ActiveHarmony project also addressed the automated tuning of TPC-W as an example cluster-based web application [14]. While tuning is an important part of Elba, the Elba project stresses automation including design and deployment to the staging area by reusing top-level design documents.

Finally, there are also projects such as SoftArch/MTE and Argo/MTE that automatically benchmark various pieces of software [17][18]. Our emphasis, however, is that this benchmarking information can be derived from deployment documents, specifically the SLAs, and then other deployment documents can be used to automate the test and staging process to reduce the overhead of staging applications.

VII. Conclusion and Future Work

The Elba project is our vision for automating the staging and testing process for enterprise application deployment. This paper presents the initial efforts of the Elba project in developing the Mulini code generation tool for staging. These efforts concentrate on mapping high-level staging descriptions to low-level staging experiments, dependency analysis to ensure proper component composition, and creating robust, adaptable designs for applications. Ultimately, long term efforts in Elba will be directed at closing the feedback loop from design to staging where knowledge from staging results can be utilized at the design level to create successively better designs.

The early results for Mulini reported here have shown promise in several areas. First, they show that Mulini's generative and language-based technique can successfully build on existing design (Cauldron) and deployment (ACCT) tools for staging. Second, our experiences show that automatic deployment during the staging process is feasible, and furthermore, that instrumentation of application code is feasible when using a Clearwater-based generator.

Ongoing research is addressing questions raised by our experiences and the limitations of the initial efforts. For example, we are exploring the translation of SLAs into performance policies, which are translated into monitoring parameters to validate staging results. Another important question is the extension of TBL to support new applications. A related issue is the separation of application-dependent knowledge from application-independent abstractions in TBL and Mulini. A third question is the migration of staging tools and parameter settings (e.g., monitoring) to production use, so problems can be detected and adaptation actions triggered automatically during application execution.

Acknowledgment

We would like to thank Sharad Singhal of HP Labs for his valuable insight and comments during the development of this paper.

REFERENCES

[1] Talwar, Vanish, Dejan Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung. "Comparison of Approaches to Service Deployment." ICDCS 2005.

[2] Sahai, Akhil, Calton Pu, Gueyoung Jung, Qinyi Wu, Wenchang Yan, Galen Swint. "Towards Automated Deployment of Built-to-Order Systems." 16th IFIP/IEEE Distributed Systems: Operations and Management (DSOM). 2005.

[3] Menascé, Daniel A. "TPC-W: A Benchmark for E-Commerce." *IEEE Internet Computing.* May-June 2002.

[4] Sahai, Akhil, Sharad Singhal, Rajeev Joshi, Vijay Machiraju. "Automated Generation of Resource Configurations through Policies." IEEE Policy, 2004.

[5] Goldsack, Patrick, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, Peter Toft.: SmartFrog: Configuration and Automatic Ignition of Distributed Applications. HP Openview University Association conference, 2003.

[6] Pu, Calton, Galen Swint. "DSL Weaving for Distributed Information Flow Systems." (Invited Keynote.) Proceedings of the 2005 Asia Pacific Web Conference. (APWeb05). Springer-Verlag LNCS. March 29 - April 1, 2005. Shanghai, China.

[7] Galen S. Swint, Gueyoung Jung, Calton Pu. "Event-based QoS for Distributed Continual Query Systems." IEEE 2005 International Conference on Information Reuse and Integration (IRI-2005), August 2005.

[8] Swint, Galen S., Calton Pu. "Code Generation for WSLAs Using AXpect." Proceedings of 2004 IEEE International Conference on Web Services (ICWS 2004). July 6-9, 2004. San Diego, California.

[9] Galen Swint, Calton Pu, Charles Consel, Gueyoung Jung, Akhil Sahai, Wenchang Yan, Younggyun Koh, Qinyi Wu. "Clearwater - Extensible, Flexible, Modular Code Generation." To appear in the Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005). November 7-11, 2005. Long Beach, California.

[10] W3C Document Object Model. http://www.w3.org/DOM/

[11] García, Daniel, Javier García. "TPC-W E-Commerce Benchmark Evaluation." *IEEE Computer*, February 2003.

[12] PHARM benchmark http://www.ece.wisc.edu/~pharm/tpcw.shtml

[13] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Alan Cox, Romer Gil, Julie Marguerite, Karthick Rajamani and Willy Zwaenepoel. "Bottleneck Characterization of Dynamic Web Site Benchmarks." Rice University Technical Report TR02-388.

[14] Chung, I-Hsin, Jeffrey K. Hollingsworth. "Automated Cluster Based Web Service Performance Tuning." HPDC 2004. Honolulu, Hawaii.

[15] Zhang, Qi, Alma Riska, Erik Riedel, and Evgenia Smirni. "Bottlenecks and Their Performance Implications in E-commerce Systems." WCW 2004. pp. 273-282. 2004

[16] Debusman, M., M. Schmid, and R. Kroeger. "Generic Performance Instrumentation of EJB Applications for Service-level Management." NOMS 2002.

[17] Grundy, J., Cai, Y., Liu, A. SoftArch/MTE: Generating Distributed System Test-beds from High-level Software Architecture Descriptions. *Automated Software Engineering, 12, 1*.

[18] Cai, Y., Grundy, J., and Hosking, J. Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool. ASE 2004.

APPENDIX A: EVALUATION DATA

Table 1. Resource utilization. "L" is a low end, "H" a high-end machine (see text for description). Percentages are for the system. "M/S" is "Master/Slave" replicated database

	DB host		APP server host	
	cpu(%)	mem (%)	cpu(%)	mem (%)
L/L	99.8	96.9	11.3	78.3
H/L	66.3	98.4	22.2	98.5
H/H	66.0	98.72	5.48	79.7
2H(M/S)/L	36.6/46.9	96.2/89.5	21.9	98.2
2H(M/S)/H	47.3/38.0	96.6/90.0	5.2	79.5

Table 2. Average response times. 90% WIRT is the web interaction response time within which 90% of all requests for that interaction type must be completed (including downloading of ancillary documents such as pictures). Each number is the average over three test runs. As we can see, even though some entries have an *average* response time that may be less than that in the SLA, the deployment may still not meet the SLAs 90% stipulation (e.g. "H/H" case for "Best Seller")

Interaction Type / **Hardware Provision**	Admin Conirm	Admin Request	Best Seller	Buy Confirm	Buy Request	Custormer Registeration	Homepage	New Product	Order Display	Order Inquiry	Product Detail	Search Request	Search Result	Shopping Cart
90% WIRT	20	3	5	5	3	3	3	5	3	3	3	3	10	3
L/L	54.6	9.3	44.1	18.9	11.1	6.9	9.0	12.6	11.6	3.10	8.04	8.6	12.6	14.6
H/L	**7.2**	**0.4**	**4.5**	**0.7**	**0.3**	**0.2**	**0.3**	**0.4**	**0.2**	**0.04**	**0.2**	**0.3**	**0.42**	**0.39**
H/H	**7.8**	**0.2**	**4.8**	**0.7**	**0.4**	**0.2**	**0.2**	**0.4**	**0.3**	**0.01**	**0.2**	**0.2**	**0.39**	**0.4**
2H/L	**2.9**	**0.04**	**2.6**	**0.7**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.03**	**0.1**	**0.1**	**0.2**	**0.1**
2H/H	**2.8**	**0.04**	**2.7**	**0.4**	**0.1**	**0.1**	**0.1**	**0.1**	**0.1**	**0.02**	**0.1**	**0.1**	**0.1**	**0.1**

Table 3. Percentage of requests that meet response time requirements. 90% must meet response time requirements to fulfill the SLA

L/L	0	36.8	0	17.5	24.6	47.1	33.7	27.3	19.3	73.4	36.9	33.7	46.6	15.5
H/L	**97.8**	**95.7**	68.8	**97.1**	**97.7**	**99.5**	**99**	**99.3**	**99.1**	**100**	**99.2**	**99**	**99.9**	**97.2**
H/H	**100**	**100**	63.6	**97.2**	**97.2**	**99.6**	**99.4**	**99.4**	**96.9**	**100**	**99.3**	**99.3**	**99.9**	**97**
2H/L	**100**	**100**	**96.9**	**99.1**	**99.7**	**100**	**99.6**	**100**	**100**	**99.8**	**99.7**	**99.5**	**100**	**99.5**
2H/H	**100**	**100**	**94.5**	**99.7**	**99.8**	**100**	**100**	**100**	**100**	**100**	**100**	**99.9**	**100**	**100**

APPENDIX B: XTBL, XMOF AND PERFORMANCE POLICY WEAVING

a) XTBL

```
<xtbl version="0.1">
  <DeploymentInfo>
    <input name="TPCW-XMOF" type="meta-model">
      <version>0.1</version>
      <description>XML based CIMMOF meta-model describing
                   resources...</description>
      <location>./meta-model/TPCW-XMOF.xml</location>
    </input>
    <input name="TPCW-WSML" type="sla">
      <version>0.1</version>
      <description>XML based SLA describing performance
                   metrics...</description>
      <location>./sla/TPCW-WSML.xml</location>
    </input>
    <input name="TPCWShopping" type="flow-model">
      <version>0.1</version>
      <description>TPCW benchmark shopping
                   model</description>
      <location>./flow-model/TPCWShopping.xml</location>
    </input>
  </DeploymentInfo>

  <AppUnits>
    ...
    <AppUnit name="EmulatedBrowser" dependency="ClientWorkloadGenerator">
      <params>
              <param name="target"/>
              <param name="maxInteractions"/>
      </params>
    </AppUnit>
      ...
</xtbl>
```

b) XMOF

```
<CIM CIMVERSION="2.0" DTDVERSION="2.0">
...
<INSTANCE CLASSNAME="QM_ComputeServer">
  <PROPERTY PROPAGATED="true" NAME="Id" TYPE="string">
    <VALUE>awing14</VALUE></PROPERTY>
  <PROPERTY PROPAGATED="true" NAME="ProcessorsSpeed" TYPE="uint64">
    <VALUE>2.8GHz</VALUE></PROPERTY>
  ...
</INSTANCE>
...
<INSTANCE CLASSNAME="QM_LogicalServer">
  <PROPERTY PROPAGATED="true" NAME="Id" TYPE="string">
    <VALUE>WorkloadGenerator_LS1</VALUE></PROPERTY>
  ...
</INSTANCE>
...
<INSTANCE CLASSNAME="QM_Activity">
  <PROPERTY PROPAGATED="true" NAME="Id" TYPE="string">
    <VALUE>WorkloadGenerator-1_Install</VALUE></PROPERTY>
</INSTANCE>
<INSTANCE CLASSNAME="QM_Activity">
  <PROPERTY PROPAGATED="true" NAME="Id" TYPE="string">
    <VALUE>WorkloadGenerator-1_Ignition</VALUE></PROPERTY>
  <PROPERTY PROPAGATED="true" NAME="ActivityHandler" TYPE="string">
    <VALUE>awing14.cc.gatech.edu//start_client</VALUE>
  </PROPERTY>
</INSTANCE>
...
<INSTANCE CLASSNAME="QM_ActivityPredecessorActivity">
  <PROPERTY PROPAGATED="true" NAME="DependencyType" TYPE="string">
    <VALUE>Finish-Start</VALUE></PROPERTY>
  <PROPERTY PROPAGATED="true" NAME="AntecedentActivity" TYPE="string">
    <VALUE>QM_Activity.Id=WorkloadGenerator-1_Install</VALUE></PROPERTY>
  <PROPERTY PROPAGATED="true" NAME="DependentActivity" TYPE="string">
    <VALUE>QM_Activity.Id=WorkloadGenerator-1_Configuration</VALUE></PROPERTY>
</INSTANCE>
...
```

c) Performance Requirements (as WSML)

```
<SLA>
...
<provider>http://hera.cc.gatech.edu:8000/tpcw</provider>
<consumer>WorkloadGenerator</consumer>
<SLO id="TPCWCommon">
  <clause>
    <evalOn id="TPCWConcurrentUsers" recordtype="input">
      <expression>
        <container name="numBrowsers" type="integer">
        <value>100</value></container>
      </expression>
    </evalOn>
    <evalOn id="TPCWMaxInteractions" recordtype="input">
      <expression>
        <container name="maxInteractions" type="integer">
        <value>100000</value></container>
      </expression>
    </evalOn>
    <evalOn id="TPCWMaxErrors" recordtype="input">
      <expression>
        <container name="maxErrors" type="integer">
        <value>30</value></container>
      </expression>
    </evalOn>
  </clause>
</SLO>
<SLO id="TPCWThroughput">
  <metric id="Throughput"></metric>
  <clause>
    <measuredAt id="TPCWClient">WorkloadGenerator</measuredAt>
      ...
    </evalOn>
    <evalFunc id="TPCWThroughputGT12" recordtype="static">
      ...
</SLA>
```

d) XTBL+

```
<xtbl name="TPCWBenchmark" version="0.1">
  <AppUnits>
    <AppUnit name="WorkloadGenerator" template="WorkloadGenerator.xsl" lang="JAVA">
      <params>
        <param name="numThreads" type="integer" value=100/>
        <param name="thread" type="Object" value="EmulatedBrowser"/>
          ...
            <param name="target" type="string"
                   value="http://hera.cc.gatech.edu:8000/tpcw"/>
            <param name="maxInteractions" type="integer" value=100000/>
          </params>
          ...
          <aspect name="Stat" template="Stat.xsl>
            <params>
              <param name="numThreads" type="integer" value=100/>
              <param name="maxErrors" type="integer" value=30/>
            </params>
            <docs>
             ...
            <aspect name="Throughput" template="Throughput.xsl">
              <docs>
                <doc name="TPCW-WSML" type="xml"/>
              ...
    </AppUnit>
    ...
    <instances>
      ...
      <instance type="install">
        <src>WorkloadGenerator/*.class</src>
        <target>awing14.cc.gatech.edu//tmp/tpcw/genCode</target>
      </instance>
      <instance type="ignition">
        <src>awing13.cc.gatech.edu//tmp/tpcw/mysql/mysql-standard-4.0.18-pc-linux-i686/start_mysql</src>
      </instance>

    </instances>
      ...
</xtbl>
```

Figure 12. Example XTBL, XMOF, Performance requirements, and XTBL+. XTBL+ is computed from developer-provided specifications. XTBL is the primary source document and the primary input to Mulini. In the `input` tag, the XTBL contains references to deployment information, the XMOF, shown in b) in reverse-print. It also refers to a WSML document, c) with underlined text. The WSML encapsulates performance policy information applicable to staging. Mulini's first stage then incorporates the three documents into a single, interwoven XTBL+ document in d). Reverse-print and underlining correspond to the data retrieved from the XMOF and WSML, respectively.