



# **Reducing the Cost and Risk of Managing Multiple Environments**

# Contents

Contents .....	<b>2</b>
Introduction .....	<b>4</b>
The Project.....	<b>5</b>
Application.....	<b>5</b>
SCM Facilities.....	<b>5</b>
Build Facilities.....	<b>5</b>
Development Infrastructure.....	<b>5</b>
Team Structure.....	<b>5</b>
Application Architecture.....	<b>6</b>
Topology.....	<b>6</b>
Network Connectivity .....	<b>6</b>
Physical Environments.....	<b>8</b>
The Problem.....	<b>9</b>
Making A Difference .....	<b>9</b>
The Solution .....	<b>10</b>
Step 1: Define the Environment.....	<b>10</b>
Hostnames .....	<b>10</b>
Network Connectivity .....	<b>11</b>
User Accounts.....	<b>11</b>
File and Directory Structure .....	<b>11</b>
Configuration Information .....	<b>12</b>
Step 2: DNS Virtualisation.....	<b>12</b>
Step 3: Automated Infrastructure Testing.....	<b>12</b>
Step 4: Automated Deployment.....	<b>14</b>
Deploy User Account.....	<b>14</b>
Deploy Scripts.....	<b>14</b>
Post-Deployment Tests.....	<b>14</b>
UAT Tests .....	<b>14</b>

Summary..... **15**

About BuildMonkey ..... **16**

# Introduction

On large projects, it is normal to have more than one environment to manage since each environment is used for a different purpose, e.g.:

- Development
- Testing
- Staging
- Production

If there are multiple development tracks, then there may be more than one instance of each of these environments – different test environments for bug-fix and new releases for example.

The most common problem encountered in this situation is where the deployed code works in one environment but not in another, and the most common question asked – by developers, testers and project managers alike, is “*are these environments the same?*”.

This case study shows how the BuildMonkey tools and methodologies were used by a global OEM on a large distributed development project to mitigate these issues and how a **four-day** deployment was reduced to **eighteen minutes** – a key factor in delivering the project early.

You will also see how the use of BuildMonkey tools and methodologies significantly reduced the risks when handing over to a third-party for production deployment.

# The Project

## Application

The application being developed was a large, mission-critical, distributed J2EE application to run in a Solaris environment.

## SCM Facilities

The SCM facilities were provided by CVS running on a Solaris server and stored all project artefacts, including all project documentation, not just source code.

All of the artefacts had to be considered part of a release.

## Build Facilities

The build technology used was Apache Ant, and it had to be possible for builds to be performed on any of the platforms in use (e.g. so that developers could run sandbox builds on their own workstation).

The build scripts were all created by BuildMonkey resources, after understanding from the architecture team how the application was to be constructed.

## Development Infrastructure

Development infrastructure was a mixture of Solaris and Win32 workstations. Individual developer tools, such as choice of IDE, was a decision left to the individual.

## Team Structure

The project consisted of a number of teams:

- **Infrastructure Team**  
Responsible for commissioning and maintaining the underlying host and network infrastructure, as well as the security of the environment;
- **Development team**  
This was split into a number of sub-teams, with each sub-team responsible for a different component of the application;
- **Testing team**  
Responsible for performing UAT and load-testing of the application prior to it being released;
- **Integration and Release Team**  
Responsible for Configuration and Release Management, and for the creation and maintenance of the application build and deployment.  
This role was fulfilled by BuildMonkey resources.
- **Production Team**  
This was a third-party organisation, hosting the production application on their own secure servers.

# Application Architecture

## Topology

The topology of the application is shown in figure 1, below:

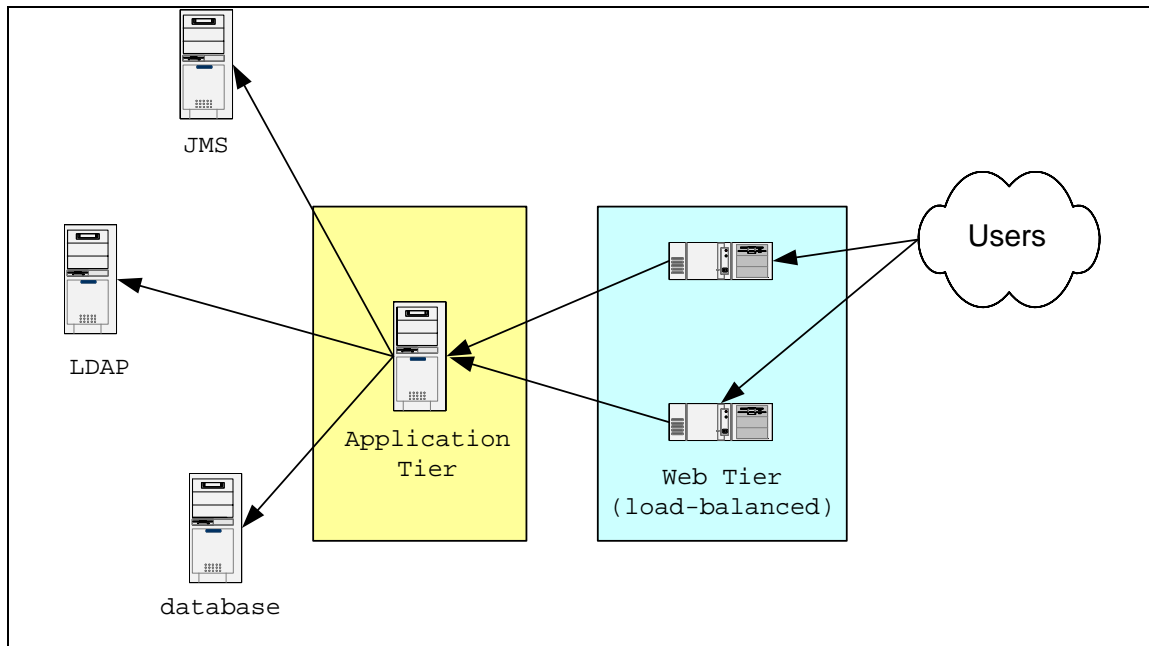


Figure 1: Application Topology

The system was architected to provide horizontal scalability, with the exception of the database and LDAP components, which were intended to be scaled vertically.

This horizontal scaling was to be provided by an Alteon load-balancer, which was only available in the staging and production environments.

The web tier provided SOAP-based web services to clients (where the 'users' were other systems) as well as normal browser-based services.

The application tier housed all of the business logic of the application using a combination of session, entity and message-driven beans.

## Network Connectivity

The network connectivity requirements of the application is shown in figure 2 below. Unless otherwise stated, all port numbers refer to TCP connectivity.

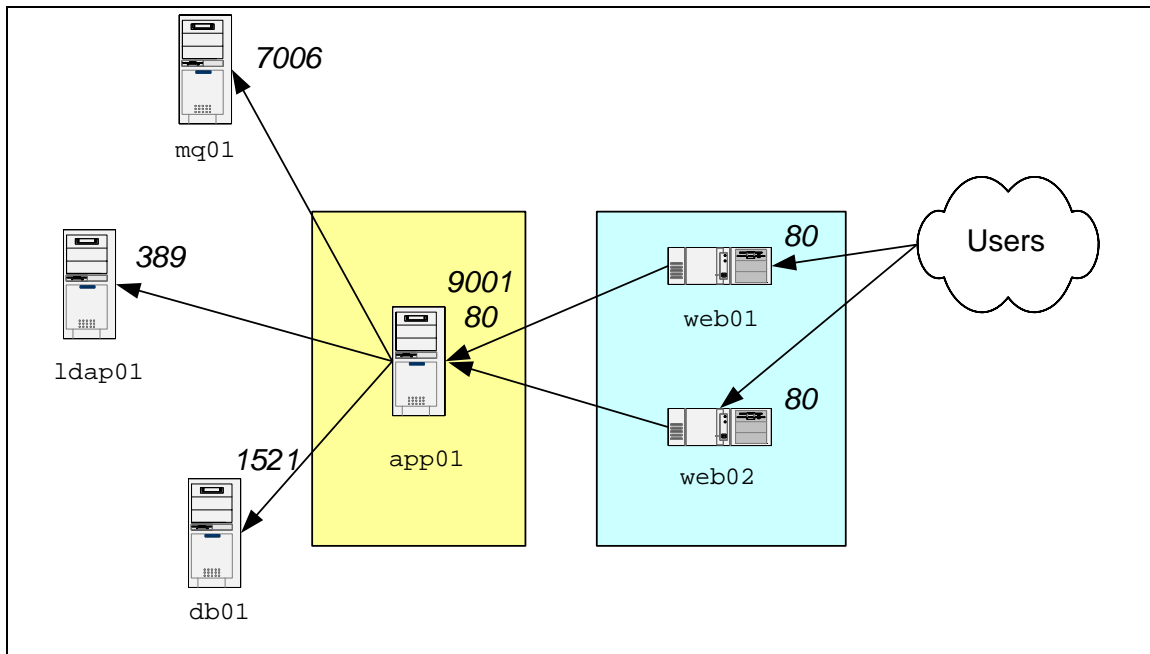


Figure 2: Network Connectivity Requirements

The hostnames will become important when we consider the way in which the DNS was configured later, but the key point to note is that the DNS name is also indicative of the function of the server (e.g. `app01`).

# Physical Environments

There were four physical environments to be considered in the project, each with a distinct function:

- **Development environment**  
As its name suggests, this was essentially a sandbox where the development team could deploy and unit test their code.  
Due to the fact that the contents of this environment could change rapidly (e.g. by a developer dropping and recreating a lot of database tables), the key consideration here was to ensure that it could be rebuilt quickly.
- **Test environment**  
This environment was kept under far tighter control, since it was used by the testing team to perform UAT and load testing.  
The key consideration for this environment was that its contents should be known and stable so that tests could be conducted with confidence.
- **Staging environment**  
This was the on-site replica of the staging environment, used to recreate symptoms discovered in production and to perform full UAT prior to deployment.  
It was absolutely essential that this environment be identical to the production environment for it was here where symptoms were classified (e.g. agreed as bugs).
- **Production environment**  
This was a live environment, hosted by a third party, running the application.

Only the staging and production environments were identical in terms of the number and specification of servers. The other environments were of a much lower specification and servers sometimes had to perform more than one function (e.g. housing the LDAP and database on a single server).

## The Problem

The main problem, as you can probably see, was how to ensure that these physically different environments could be made logically identical to:

- Allow the application to be automatically deployed to multiple environments without requiring manual configuration. Aside from the time taken to make the changes, and the risk of error, the real problem with manual configuration is that if a file is checked out of the SCM repository and then edited, it does not exist in any code baseline.
- Allow automated tests to show – instantly – whether there were differences in the environment that would cause problems with the application;
- Facilitate the use of a single set of test scripts to test application functionality – i.e. the UAT tests could be run against any environment where the application was deployed;
- Reduce management overhead through the environments being consistent – no need to remember individual hostnames or configuration information;
- Reduce the risk of error through manual changes to multiple environments – boiler plate configuration information could be used;
- Make it simple and quick to reliably rebuild environments from scratch

These are common problems in large development projects. Solving them used to depend on the skills of the infrastructure or integration teams, but BuildMonkey removes this key dependency.

## Making A Difference

By following the BuildMonkey solution, a number of benefits became apparent:

- The environments were easier to manage, with the overall management burden reduced;
- The time, and cost, of installing and deploying the application was significantly reduced from **four days** down to only **eighteen minutes**;
- A detailed infrastructure test plan had been created, almost as a side-effect;
- Detailed infrastructure documentation had been created, again almost as a side effect;
- It was possible to know – instantly – whether two environments were the same or not.

Massive amounts of time, and frustration, was saved in the build and deployment processes.

The deployment process, and the third party handover, could be summed up by ***“Know WHAT is wrong, care less about WHO is wrong”***

## The Solution

The first thing is to ensure that the terms “same” and “different” are clearly understood. Whilst the hardware infrastructure is quite obviously different, it needs to be borne in mind that we are concerned only with how the application views the infrastructure.

For example, the application does not care whether the database is running on a massive Sun server or a development laptop – all it cares about is that it can make a JDBC connection to the url specified in the configuration file.

This gives rise to the following definitions:

1. Two environments are the “same” if the application can be deployed to each of them and display identical behaviour – without requiring any configuration or other changes;
2. Two environments are “different” if the application displays different behaviour in each, or if the configuration requires modification in order to have the application behave correctly;

### Step 1: Define the Environment

This is the key step, since it clearly sets out what the application requires of the environment in terms of:

- Hostnames
- Network connectivity
- User accounts
- File and directory structure
- Configuration information

#### Hostnames

The logical host names for the application were defined, ensuring that the configuration information of the application could be the same across all environments. The information in the following table reflects that shown in figure 2 above.

Logical Name	Details	Dev	Test	Staging	Production
web01	Web server	ccdev1	cctest1	ccstg1	ccprod1
web02	Web server	ccdev2	cctest2	ccstg2	ccprod2
app01	Application server	ccdev3	cctest3	ccstg3	ccprod3
bb01	Database server	ccdev4	cctest4	ccstg4	ccprod4
ldap01	Directory server	ccdev4	cctest4	ccstg5	ccprod5
mq01	Web server	ccdev5	cctest5	ccstg6	ccprod6

*Table 1: Logical hostnames mapped to physical hostnames*

The problem being solved here is immediately apparent. In order to configure the database connection in the Application Server, it would be necessary to connect to a different host in every environment. By using logical names, the hostname is the same every time.

The information is also easier to remember, particularly for people who are unfamiliar with the environment, since it is not related to the physical machines. This significantly reduces the learning curve for new staff, as well as reducing the overall management overhead.

Since all IP-based network communication that uses hostnames relies on the name-resolution facilities of the platform on which it is running, there is no issue to using logical – rather than physical – hostnames.

## Network Connectivity

Next, the connectivity requirements of the application were considered – again reflecting the contents of figure 2. Notice how, in the following matrix, it is the logical hostnames that are used:

	web01	web02	app01	db01	ldap01	mq01
web01	-	-	80	-	-	-
web02	-	-	80	-	-	-
app01	-	-	9001 <sup>1</sup>	1521	389	7006
db01	-	-	-	-	-	-
ldap01	-	-	-	-	-	-
mq01	-	-	-	-	-	-

Table 2: Connectivity Matrix

This matrix, as well as being useful information in its own right, is now starting to look like an Infrastructure Test Plan.

## User Accounts

The user account information for each of the servers in the environment was then defined. This was of particular importance since the security considerations required that the applications – where possible – were not running as root.

Host	Application	Users	Groups
web01	Web server	web deploy	web
web02	Web server	web deploy	web
app01	Application server	iuser deploy	igroup
db01	Database server	oracle deploy	dba
ldap01	Directory server	iuser deploy	igroup
mq01	Message queue	iuser deploy	igroup

Table 3: User and Group Accounts

This not only allowed the infrastructure team to know which accounts had to be created on which box, it also increased the relevance of the Infrastructure Test Plan.

## File and Directory Structure

The file and directory paths required by each of the applications were then defined. This provided a number of benefits:

- All paths being the same reduces the management overhead;
- Installation and Deployment could be fully automated since it was consistent;
- Disk layout requirements were easier to specify for the infrastructure team;

<sup>1</sup> This is an IIOP listener, so that the application server could make callbacks to its own beans

The results are shown in the table below:

Host	Software	Path
web01	Web server	/apps/webserver /var/apps/webserver/log
web02	Web server	/apps/webserver /var/apps/webserver/log
app01	Application server	/apps/appserver /apps/appserver/domains /var/apps/appserver/log
db01	Database server	/apps/db/product /apps/db/data /var/apps/db/log
ldap01	Directory server	/apps/directory /var/apps/directory/log
mq01	Message queue	/apps/mq/product /apps/mq/queue /var/apps/mq/log

*Table 4: File and Directory Structure*

The test plan for the environment now covered files and directories as well as network connectivity, allowing even more rigorous environmental testing to be performed as part of the pre-deployment checks.

### Configuration Information

The configuration for the application could now be defined, since the underlying infrastructure was known.

The details of the information is not included in this case study, but included:

- Data and log file paths
- Connectivity URLs (e.g. for `iiop://` and `jdbc://` connections)
- User information (e.g. which processes run as which user)

A complete infrastructure test plan was now available.

## Step 2: DNS Virtualisation

Since all of the environments were using DNS for hostname resolution, the next step was to add aliases for all of the logical hostnames to the DNS zone files.

The aliases were added to the DNS zone files for each environment as CNAME records for the physical hosts. These zone files, being an integral part of the environment, were stored in the SCM repository.

The DNS client configuration on all of the hosts was set to append the correct domain name for the environment (e.g. `dev`, `test` or `stg`). One of the advantages of using DNS is that the domain to look up can be automatically appended to the hostname (e.g. `web01` will automatically become `web01.dev` or `web01.test` depending on the environment).

This ensured that specifying the unqualified hostname (e.g. `web01`) would result in the correct host being resolved (e.g. `web01.dev`).

## Step 3: Automated Infrastructure Testing

Armed with all of the information in the tables above, an Infrastructure Test Plan was created.

These infrastructure tests proved invaluable when transitioning from development to deployment in the third-party infrastructure. The successful completion of this plan became the gating entry criteria for the deployment – saving massive amounts of time spent arguing over whether or not symptoms were problems in the application or the infrastructure by de-personalising the whole deployment process.

The BuildMonkey EVT software was installed on all of the hosts. This is a small footprint application, written in C, that runs on a host and performs infrastructure tests as defined in a configuration file:

- Hostname resolution
- File / directory checking
- Network connectivity
- Web page retrieval
- Process checking

It posts the results of these tests to the BuildMonkey dashboard, allowing the status of all tests – from all hosts – to be visible in a single location:

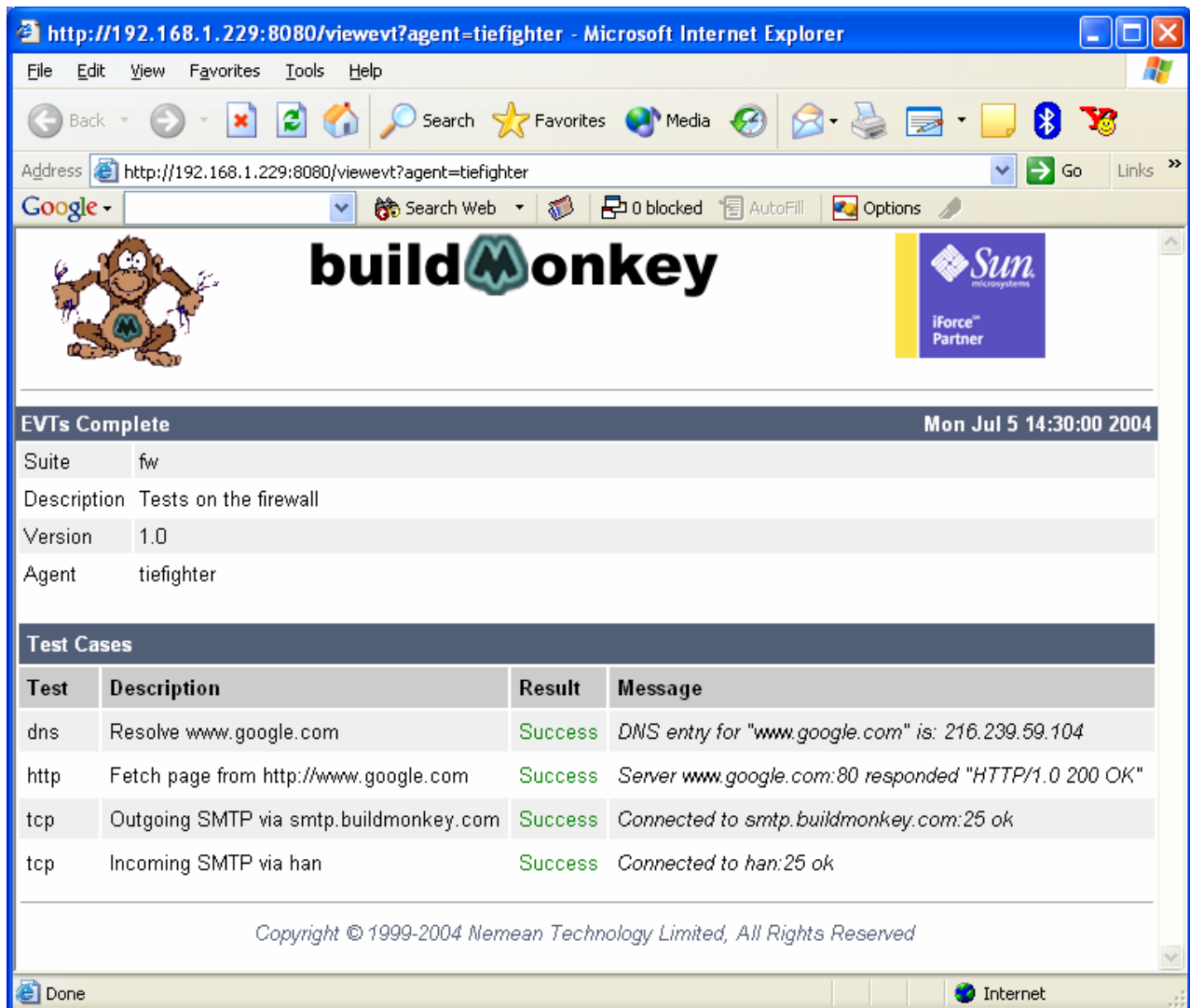


Figure 3: BuildMonkey EVT Tests

The configuration files for these tests were stored in the SCM repository, and the tests were scheduled to automatically run every 10 minutes.

## Step 4: Automated Deployment

The final step was to ensure that every single host, in all of the environments, could be rebuilt to a known state in as short a space of time as possible.

The reasons for this requirement in development are fairly obvious – since this environment is likely to be contaminated with unexpected changes as developers try out various things to further their development activities. It is important to be able to refresh this environment to a known state without human intervention.

In a testing environment, it is essential to have a baseline and to know exactly what is being tested. By being able to build the test environment quickly and accurately as part of the automated suite, it is possible to significantly reduce the length of the test cycle and to have more confidence in the results.

In the staging environment, like the testing environment, it is crucial to know that the baseline is identical to production. By having automated deployment, and not relying on manual configuration, it is possible to recreate the production baseline quickly and with confidence.

The automated deployment implemented by BuildMonkey allowed an environment to be built, from bare operating system, in less than twenty minutes and perform a complete UAT cycle on it after deployment.

### Deploy User Account

A special account for deployment, called `deploy`, was created on all of the hosts – including the SCM server. SSH trust relationships were established to allow automated ssh access from the SCM server to any host in the environment<sup>2</sup>.

### Deploy Scripts

Deployment scripts, using both Ant and shell scripts<sup>3</sup>, were created. These scripts, when invoked, would:

- Remove the instance of an application from a target server;
- Install the instance of an application from a target server;
- Configure the application on the target server, using the required configuration versions from the SCM repository;
- Deploy any bespoke code to the target host;
- Start the application

These were completely automated, requiring no human interaction. A master script would invoke all sub-scripts and could therefore recreate an entire environment from base OS installation in less than twenty minutes.

### Post-Deployment Tests

The EVT<sup>2</sup>s on the target hosts were then run, to ensure that the application was running correctly in the environment, specifically that all required services were started.

### UAT Tests

The UAT tests, as implemented by the testing team, were then invoked to test the running application.

---

<sup>2</sup> This was disabled in the production and staging environments unless it was explicitly required, providing a level of security.

<sup>3</sup> These would have been written in Perl, for maximum portability, but Perl was not permitted in the environment as part of the security requirements

## Summary

Managing multiple environments on a development project can be a costly and complex exercise if not done correctly.

By looking at the environments in a new way – i.e. from the application perspective – it is possible to have two environments with widely different hardware infrastructure that are logically the same.

By considering the requirements of the infrastructure, from a project perspective, it becomes possible to ensure that the infrastructure meets the needs of the project plan and not the other way round.

Massive savings in time, cost and complexity can be realised by combining the creation of logical environments with automated EVT testing, and an early-warning system is available for infrastructure issues.

Massive benefits in accuracy and reliability are realised through the implementation of automated deployments and testing, reducing a four day manual task to only eighteen minutes.

There can be utter confidence in the baseline in any environment, and problems can be identified – and resolved – quickly through the use of appropriate tools and methodologies.

## About BuildMonkey

BuildMonkey are the market leaders in Build, SCM and Deployment.

Formed in 1999, and with many Fortune 500 and FTSE 100 blue-chip clients, we are the original and the best.

All of the concepts described in this paper have been encapsulated in a suite of off-the-shelf tools and associated processes to facilitate rapid implementation of the Best Practices set out in this paper.

We are passionate about solving the problems which plague software development. We know that, with very little effort, it is possible for software to be delivered on-time, on-budget and free of defects.