Risk and Configuration management

Configuration management is the process of identifying system components, tracking relationships between components, and communicating the status and flux in those components. Typically, this consists of storing detailed information about versions and characteristics for each component, as well as keeping the records current and complete. It is optimal to also track all changes to any components, so that a full history of the environment at any point can be reconstructed. Reporting the state of components is another important aspect, as any interested parties can then review the information.

The first step is to define the components that will be tracked. Traditionally, configuration management was often applied only to tracking source code revisions, but in recent years it has expanded to cover the complete set of components that go into a system. This includes the source code, third party binaries, loose files, build tools, user environments, machine configurations, and any other items that are consumed by the system.

Once the types of components are identified, a baseline should be established that details the components as of that moment. While there is no perfect rule on what specific data should be recorded, there should be sufficient information that if there were a catastrophic loss, the system could be reconstructed from scratch. As a result, it is common to have bootstrapping procedures documented and included in the record.

As time passes and the system evolves, it is critical to keep the records up to date. While it is possible to do this by updating the documentation as necessary, it is better to have a change control process so that any modifications was clearly identified. Instead of having to remember to update system documentation when, for example, a new JDK is added, the process of updating the JDK should trigger an explicit documentation review. This could be done by having a component upgrade procedure that is followed whenever a component is modified, added, or removed. This is particularly important when resources are short and changes are urgently needed, when it would be easy to forget to update documentation.

Periodic audits should be performed to ensure the record is accurate and complete. This is less necessary when using a mature change control process, but can still be beneficial to assess new risks and put new components under proper management.

The final step in configuration management is proper communication of the system components to interested groups. The primary use of this is for the maintainers of the system to be able to restore or duplicate environments. However, internal customers such as development and quality assurance can also benefit as they are able to see the complete system and verify that they have compatible environments.

As well as the benefits to meeting direct build engineering goals, proper configuration management can also lead to many advantages for the rest of the software development organization, including developers, quality assurance, documentation, and support.

Developers can configure their own build environments to match the well-documented build engineering environment, and use the same tools. This reduces the chance of bad builds due to the environment or tools in the developer's local path. Properly insulated build environments make it easy for developers to support multiple codelines.

Since every changed component going into the build system is tracked, developers will only be contacted when they've committed into a newly broken build. Rather than a widespread email, buildmasters can contact the specific developers who made modifications, so developers who don't commit into that build aren't distracted from their work. Change control also benefits developers when the changes are communicated to the team. Developers should be able to review the builds and see exactly what source changes made it into each build, so that they can see that the build containing their changes was successful.

Furthermore, a well-designed set of build tools and appropriate infrastructure can allow people to work in locations independent of network domains, geographical locations, and machine architectures. If the behavior of the build tools is uniform across platforms, it becomes cheaper to develop and support the system, and to allow it to scale as the team grows.

For quality assurance, having a more reliable build process means they can get access to the latest source changes as soon as possible, instead of spending effort testing an older build. They also benefit from seeing what source changes went into a build, as they can focus their testing on those areas, instead of having to start a full new round of testing. This allows QA to work more efficiently. Because the QA team can trust the build system to be repeatable, they are able to drastically reduce testing time and focus on specifics areas of the product that have been modified since the previous build.

Documentation teams can see similar benefits from being able to track changes and more easily identify what documents need to be updated. Support teams can track bugs and see when they are fixed, and then access a specific build to verify the fix.

Each of these teams will benefit directly when the build engineering team applies configuration management techniques to improve their system.

**Specific Improvements to Reduce the Risk**

|  |
| --- |
| A Risky System  *Figure 1: A Risky System* |

While there are many theoretical benefits from applying configuration management techniques, it's also worth looking at specific examples of risks that can be identified and mitigated using the process. Each of the examples below has been seen in a live production build system, and includes a suggested improvement. In Figure 1, we see a risky system, where the build system picks up loose files and binaries from unknown sources. Each of them is outside the control of the build engineering team and could easily change without notice. Similarly, the build host is not clearly identified and would be hard to replace if there was a system failure.

**Source Code**

Source code is the most common component that configuration management is applied to. However, some teams have done development work in local source trees, and share code among developers by manually copying files. Not only does this not provide any mechanism to track changes, but it is much more likely to have one developer's changes overwritten by another and break the build.

All product source code should be stored in a centralized source code repository. At a bare minimum, the repository must track revisions. Ideally, the system would also be able to produce a change log of all modifications since a certain time, support multiple branches, and be accessible for users in multiple physical locations working on multiple operating systems. For example, if the system relies on NFS to access the source, such as when using a TeamWare workspace, it means that users on non-UNIX systems will need special configurations to access the source code. (However, if NFS access is a central theme of a build system, then it can be managed as a primary requirement, and the risk of using it is reduced.) If the source control system has a multi-platform client/server architecture, users can work on any operating system and work with the source.

Tracking changes is particularly important, as the most common failures of well-designed build systems are due to source code changes. It is imperative that build engineers be able to see what source code changes have been put into the system since the last good build in order to focus their troubleshooting efforts on the appropriate developer. Each build should report on the list of changed components since the previous build and identify the person who made the change. This allows the build engineering team to contact the developers who committed troublesome source code, and also allows developers to verify that their source code changes are complete and picked up by a specific build.

**Non-Source File Dependencies**

A common threat to a reliable build system is the use of files in the build that are not considered part of the product source, yet are important to a successful overall build. Common examples of these are documents that are handed off manually, or license files that are taken from another user's home directory. Files that are stored in locations outside the control of build engineering are much more likely to be moved, deleted, or renamed and cause a build failure.

Any file that is consumed by the build in any way should be stored on a build engineering-owned machine. Ideally those files would be stored along with the product source code in a source code repository. Regardless of the exact location of the files, they should be under the control of build engineering. Changes to the files should be managed similar to product source code, where all changes are communicated, and it's possible to access older revisions of the files.

**Tools (Binaries and Automation)**

Care should be taken to avoid using build tools that are not under the control of build engineering. For example, relying on executables that are made accessible on a wide-scale within the company mean that if those executables are changed or moved, the build could fail. Since finding a specific version of a tool can be difficult, once a good version is found, it should be stored under the control of the build engineering team, and distributed to the development teams.

Like other components that go into the build, the tools should be identified and tracked. Ideally, a build engineering group can provide a distribution of tools that are produced from source code or stored binaries in a repeatable fashion. Reliance on system versions of tools like “ls” or “ps” can cause compatibility problems if a different shell environment or operating system type is used. The tools distribution should also contain “third party” binaries like the Java SDK and other build-related binaries like Ant, make, or compilers.

If the build engineering team provides a set of build tools, then both the build engineering team and the developers can use the same set of tools. Uniform behavior across different machine operating systems means that test builds are more reliable. It's also cheaper for build engineering to maintain complex build scripts, as they can rely on a consistent base of low-level binaries as the foundation of their tools.

**Machine Configuration**

It's fairly common for build systems to run on top of machines that are inadequately documented. This covers both the current state of the system, as well as a procedure for configuring the machine. Common machine elements that can affect the build include the operating system version and patch level and hardware configuration. When a machine's state is unknown, there could be changes to the system that would cause a problem for a build. If there is a catastrophic data loss and the build machine needs to be replaced, the lack of a machine inventory and configuration documentation makes it much harder for the build engineer to set up a replacement machine in a timely manner.

|  |
| --- |
| A Safer System  *Figure 2: A Safer System* |

To reduce the risk of problems due to the machine environment, any host that is used by build engineering must be clearly described. Easily-accessible documentation should contain a list of the hardware specifications, operating system version, patch level, and additional software that has been installed. It should also include a configuration procedure sufficient to rebuild the machine in the case of a catastrophic failure. The best method for testing the documentation is to have someone unfamiliar with the system follow the procedure and then clean up any confusing or incorrect sections.

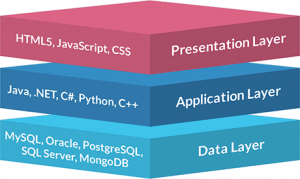
In many cases these procedures can be made available to downstream customers such as development to be used in setting up their own systems. This makes it easier for developers to work in an environment that is consistent with the build engineering environment, which then reduces the likelihood of compatibility issues.

Architecture

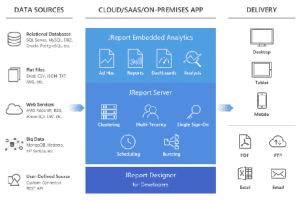
A 3-tier architecture is a type of software architecture which is composed of three “tiers” or “layers” of logical computing. They are often used in applications as a specific type of client-server system. 3-tier architectures provide many benefits for production and development environments by modularizing the user interface, business logic, and data storage layers. Doing so gives greater flexibility to development teams by allowing them to update a specific part of an application independently of the other parts. This added flexibility can improve overall time-to-market and decrease development cycle times by giving development teams the ability to replace or upgrade independent tiers without affecting the other parts of the system.

For example, the user interface of a web application could be redeveloped or modernized without affecting the underlying functional business and data access logic underneath. This architectural system is often ideal for embedding and integrating 3rd party software into an existing application. This integration flexibility also makes it ideal for embedding analytics software into pre-existing applications and is often used by embedded analytics vendors for this reason. 3-tier architectures are often used in cloud or on-premises based applications as well as in software-as-a-service (SaaS) applications.

## **What Do the 3 Tiers Mean?**



* **Presentation Tier-**The presentation tier is the front end layer in the 3-tier system and consists of the user interface. This user interface is often a graphical one accessible through a web browser or web-based application and which displays content and information useful to an end user. This tier is often built on web technologies such as HTML5, JavaScript, CSS, or through other popular web development frameworks, and communicates with others layers through API calls.
* **Application Tier-**The application tier contains the functional business logic which drives an application’s core capabilities. It’s often written in Java, .NET, C#, Python, C++, etc.
* **Data Tier-**The data tier comprises of the database/data storage system and data access layer. Examples of such systems are MySQL, Oracle, PostgreSQL, Microsoft SQL Server, MongoDB, etc. Data is accessed by the application layer via API calls.

[](https://www.jinfonet.com/wp-content/uploads/2017/12/3-tier-architecture-jreport-1-1024x690.png)

Example of a 3-tier architecture: JReport.

The typical structure for a 3-tier architecture deployment would have the presentation tier deployed to a desktop, laptop, tablet or mobile device either via a web browser or a web-based application utilizing a web server. The underlying application tier is usually hosted on one or more application servers, but can also be hosted in the cloud, or on a dedicated workstation depending on the complexity and processing power needed by the application. And the data layer would normally comprise of one or more relational databases, big data sources, or other types of database systems hosted either on-premises or in the cloud.