DRAFT - IT Project Guidance

Development:  
Code Repository Configuration and Management

Version:

0.1

Author:

Sky Sigal, Solution Architecture

## Description

This document provides guidance to developers on how to configure and manage Code Repositories used to develop custom code for ICT Projects.

## Synopsis

The organisation of code repositories improves or impacts the delivery cadence. What is put into code repositories affects security.

## Contents

[Description 1](#_Toc208490323)

[Synopsis 1](#_Toc208490324)

[Contents 2](#_Toc208490325)

[Introduction 3](#_Toc208490326)

[Issues 3](#_Toc208490327)

[Risks 3](#_Toc208490328)

[Resolution 3](#_Toc208490329)

[Types of Code 3](#_Toc208490330)

[Checking out, Staging, Committing, Checking in 5](#_Toc208490331)

[Commit Frequency Pattern 5](#_Toc208490332)

[Commit Message Pattern 6](#_Toc208490333)

[Repository Branching Pattern 6](#_Toc208490334)

[Branch Pull-Requests 6](#_Toc208490335)

[Code Repository Protection 7](#_Toc208490336)

[Verified Code Branch Integration 7](#_Toc208490337)

[Validation by Automated Tests 8](#_Toc208490338)

[Conclusion 9](#_Toc208490339)

[Appendices 10](#_Toc208490340)

[Appendix A - Document Information 10](#_Toc208490341)

[Versions 10](#_Toc208490342)

[Images 10](#_Toc208490343)

[Tables 10](#_Toc208490344)

[References 10](#_Toc208490345)

[Review Distribution 10](#_Toc208490346)

[Terms 10](#_Toc208490347)

# Summary

# Background

In the realm of software development, maintaining and accessing different versions of code is crucial. Historically, developers relied on zip files with consistent naming conventions to manage code versions. However, this method had significant drawbacks, such as the risk of data loss if directories or devices were misplaced, and the inability to easily share code with partners working remotely.

The introduction of versioned code repositories addressed the main challenges associated with earlier methods of code management, such as using zip files. Over time, several tools have been developed to help track and manage changes to code. Initially, many organisations used centralised systems, where all code changes were stored in a single, central location. Examples of these include Microsoft's Team Foundation System (TFS), which is a proprietary solution, as well as Concurrent Version System (CVS) and Subversion (SVN), which are open-source tools. Later, distributed version control systems like Mercurial emerged, allowing every developer to have a full copy of the code repository on their own machine. Each of these systems offered different features, advantages, and drawbacks. Today, the software development industry has mostly settled on using Git, a distributed version control system, as the standard for managing code changes and collaboration among developers.

Git's primary advantage lies in its distributed nature, as opposed to a central repository model. In a distributed system, every developer has their own complete copy of the entire code repository. This contrasts with the central model, where developers only have local subsets of the repository. The distributed approach enhances collaboration and ensures that no single point of failure can compromise the entire codebase. By convention, a "central" repository is often designated to facilitate synchronization among developers.

Another key feature of Git is the low effort required to create branches. Branching allows developers to work on separate features or fixes without affecting the main codebase or the work of others. This flexibility makes it easier to manage parallel development efforts and integrate changes later.

In summary, the evolution from zip files to sophisticated version control systems like Git has significantly improved the efficiency and reliability of code management in software development.

# Issues

However, there were always issues and risks associated with code repositories, whether centralised or distributed. Even before distributed code repositories were common, relying solely on a protected corporate LAN created only an illusion of safety. Central repositories could still be compromised if the network perimeter was breached or if access controls were insufficient. Sensitive information, such as credentials or proprietary code, could be exposed if proper precautions were not taken.

Distributed code repositories introduce additional risks by design. Because every stakeholder can have a complete copy of the repository, it becomes much more difficult to keep track of and control where sensitive code and information reside. Code can be distributed to individuals beyond the secure boundaries of the organisation’s network, making traditional auditing and access controls less effective. This decentralisation means that once sensitive data is committed and pushed, it may be replicated to multiple locations, some of which may not be adequately secured.

# Resolution

Given these inherent risks, it is essential to follow certain principles and best practices to mitigate them:

* Training of developers on the risk of checking in credentials for any environment is a pre-requisite to working on a project’s code.
* Never commit sensitive credentials or secrets: Developers should avoid checking in passwords, API keys, or other sensitive information into repositories. Use environment variables or secure vaults instead.
* Implement access controls and permissions: Restrict repository access to only those who need it, and regularly review permissions.
* Use automated scanning tools: Employ tools that automatically scan commits for sensitive data before they are merged or pushed.
* Educate and train developers: Ensure all team members understand the risks and follow secure coding and repository management practices.
* Monitor and audit repository activity: Regularly review logs and repository activity to detect any unauthorised access or suspicious changes.

By adhering to these principles, organisations can significantly reduce the risks associated with both centralised and distributed code repositories, protecting their codebase and sensitive information from potential threats.

The selected text outlines essential principles and best practices for managing the risks associated with both centralised and distributed code repositories. These practices are highly relevant and appropriate given the context, which emphasises the challenges of protecting sensitive information in modern version control systems, especially with the widespread use of distributed repositories like Git.

* Training developers: Mandating training on the risks of committing credentials is crucial. Developers must understand the security implications of version control and the potential consequences of mishandling sensitive data.
* Avoid committing secrets: The recommendation to never commit passwords, API keys, or other sensitive credentials is aligned with industry standards. Using environment variables or dedicated secret management solutions ensures that sensitive information is not exposed in repositories.
* Access controls and permissions: Implementing strict access controls and regularly reviewing permissions helps limit exposure and reduces the risk of unauthorised access, especially in distributed environments where copies of the repository may exist outside organisational boundaries.
* Automated scanning tools: Employing automated tools to scan for sensitive data before code is merged or pushed is a proactive measure to catch inadvertent exposures early.
* Education and ongoing training: Continuous education ensures that all team members remain aware of evolving risks and best practices in secure coding and repository management.
* Monitoring and auditing: Regularly reviewing logs and repository activity is vital for early detection of suspicious changes or unauthorised access, enabling timely response to potential threats.

These practices, when consistently applied, directly address the risks highlighted in the context—such as accidental exposure of credentials, the challenges of distributed repository management, and the need for secure and auditable codebase handling. They are considered industry best practices and are fundamental to maintaining the security and integrity of software development projects.

In summary, the principles listed in the selected text are valid, comprehensive, and well-aligned with both the historical challenges and the current realities of code repository management described in the surrounding context. Adhering to them will significantly enhance an organisation’s ability to safeguard its code and sensitive information.

# Types of Code

Code Repositories manage the persistence of versions of committed code. A risk is that stakeholders incorrectly imagining the code that is required to be managed this way is that which is needed to develop a custom system.

When discussing code, we include all the following as well:

* Workstation Configuration: tools to get download, install, etc. tools for developers and others. E.g., PowerShell scripts that in turn invoke Chocolate, etc.
* System Code: the custom code to develop a system from scratch.
* Customisation Code: custom code used to develop extensions to a subscribed or purchased platform (e.g.: a custom app running on Salesforce).
* Automated System Unit Tests:   
  Run by the development pipeline after compilation of system code before packaging & deploying the system[[1]](#footnote-2).
* Infrastructure schema definitions:  
  the code used to instruct the building of secure target environments developed on cloud infrastructure. Preferably idempotently.
* Data storage schema definitions:  
  SQL code used to define the schema of target databases used by the logic system[[2]](#footnote-3).

**Important:**The code MUST be idempotent to not drop catalogues or attributes in an unplanned manner when run, damaging production data.

* **Client-side check-in hooks:**code to protect the central repository from insecure commits.
* **Pipeline development code:**the code that instructs the pipeline how to download dependencies, compile, static test, package, deploy, seed, import data, dynamic test, report on.
* **Configuration & Integration Code:**executed post deployment, pre-runtime to define integrations to dependency services (database, cache, email, search, ip-geo translation, etc.)
* **System Settings Code:**executed post initiation, to invoke the system’s APIs to change, idempotently, system settings (branding, links, etc.)
* **Reference Data & Code sets:**instructions to seed the system with agreed reference data
* **System Data Migration code:**ETL code run post deployment to import seed and/or legacy data from other systems,
* **System Data Restoration code:**Code to instruct the code environment to automate the regular backing up of system transactional and reporting data.
* **System Data backup code:**Code to automate the restoration of the backed data when required,
* **Automated Behavioural Testing scripts:** code executed post compilation, packaging, and deployment to target environments,
* Project Dashboard Automation:  
  code to automate the summarisation of work item completion, velocity, test results, qualities, etc.

# Checking out, Staging, Committing, Checking in

This document does not summarise Git, bar describing the following:

**Staging** is the Git process of selecting files to be Committed. Usually these are the files that have been added or changed locally.

**Committing** is the Git process of safeguarding Stage selected files to the local database, with a commit message summarising the current state. Note that at this stage, the state is only recorded locally and has not been shared with others.

**Rollback** is the Git process that can be performed when changes made after the last Commit are deemed disposable, and one rolls back to the last acceptable state by unpacking the files that were Committed.

Note:  
this is only available for developers have the good habit of Committing in regularly.

**Pushing** the Git process of synchronising the local instance of a Git database with a remote instance of a Git database (e.g., a project’s Git repository on GitHub or ADO or other).

# Commit Frequency Pattern

Best practice is for developers to Commit frequently to their local instance of a Git database – generally when a system compiles and/or passes another unit test.

The practice provides valuable outcomes:

* This permits developers who make mistakes to roll back to a working state without losing lots of work.
* When viewed in a list the incremental Commit Messages visually describe in a coherent manner what the developer was thinking, and how they went about solving the problem the current work item

Note:  
this depends on the quality of the Commit Messages (see below).

# Commit Message Pattern

Commit messages are messages summarising what is being Committed to a local instance of a Git database.

They should be:

* Start with the Work Item (User Story) Identifier (e.g.: “PRJ-473:”) the work is addressing
* Succinctly describe what is being checked in (e.g.: “PRJ-473: Add Foo endpoint to BarService”)
* Remain overall short, so that they fit within online lists (e.g.: < 64 characters)

# Repository Branching Pattern

When working collaboratively it is important that one’s work does not negatively impact the work of others.

Git addresses this by making Branching cheap (i.e., computationally inexpensive, therefore easy to use often).

Branching allows a developer to cut themselves a clone of the current state, (“a branch”) so that work within it is independent of both the combined work of the whole team (e.g.: the “trunk” branch) as well as the work of all other developers, if they are all following best practice for branching.

Best practise is that for each work item, developers create a new branch.

The branch should be given a name that reflects the work item identifier (e.g.: “PRJ-292”).

# Branch Pull-Requests

The Agile Manifesto was put together by top-notch developers. Even for them – and certainly for everyone else -- review of code submissions is a valuable process to protect the code base.

Pull-Requests for Review of changes must be issued before branches are merged into target branches.

Mature code integration pipelines can be configured to automate reviews of code formatting, code complexity, unit testing, functional testing, etc – basically taking care of the minutia -- before the submitted code is reviewed by a human team member for opportunities to optimise the code.

**Note:**   
if the continuous pipeline has been configured to automate the review of format, complexity, unit testing, and functional testing, review by a human can often be skipped – most apps are not required to be race cars (they really just need to be functionally correct, while meeting quality expectations).

# Code Repository Protection

As outlined earlier, developers can compromise system security by committing code that contain confidential information, such as integration identification (URLs & endpoint names) and required credentials (service account name and associated secret password).

To mitigate this risk:

* Developers should undergo training to best understand why it is a risk, and to be instructed on what to do instead. This generally implies the following:
* Using “password-less” service accounts where available (e.g., Microsoft Secure Identities (MSI) in Azure, or equivalent on other cloud providers),
* Persisting credentials in credential stores, retrieving them when needed, rather than persisting them in code (e.g.: config files in the root of services).
* Encrypting the settings if there is no work around to including them in system configuration files.
* Developing and Installing server-side (e.g., in ADO, or GitHub) repository validation protection logic to scan code being committed or pushed for patterns that appear to be credentials.

Note:   
as the IT industry moves away from passwords of a certain length and complexity towards passphrases, pattern matching becomes less reliable for catching insecure commits, and newer solutions are required.

* Potentially doing the same on the client side using commit “hooks”.

# Verified Code Branch Integration

An important responsibility of a version control system is to verify submitted code feature branches before integrating the code with the protected `master` branch.

This upfront effort to configure the continuous integration pipeline to protect the code base from getting polluting, protects other developers from downloading poor code and working around it – only to have to remove the work arounds when the original poor code is fixed.

Tip:  
This is no different than the kinds of real-world safeguards that would be put in place if we were talking about a communal water reservoir, with everyone tasked with filling it up, and everyone drinking from it. Unless catching typhoid was an objective.

If the Version Control Service rejects the code due to it failing tests -- or a `Pull Request` reviewer (see elsewhere in this View) has manually rejected the submission -- the developer is required to modify their code and re committing before the Version Control Service will allow the submitted feature branch to be integrated with the protected `master` or ‘trunk’ branch:

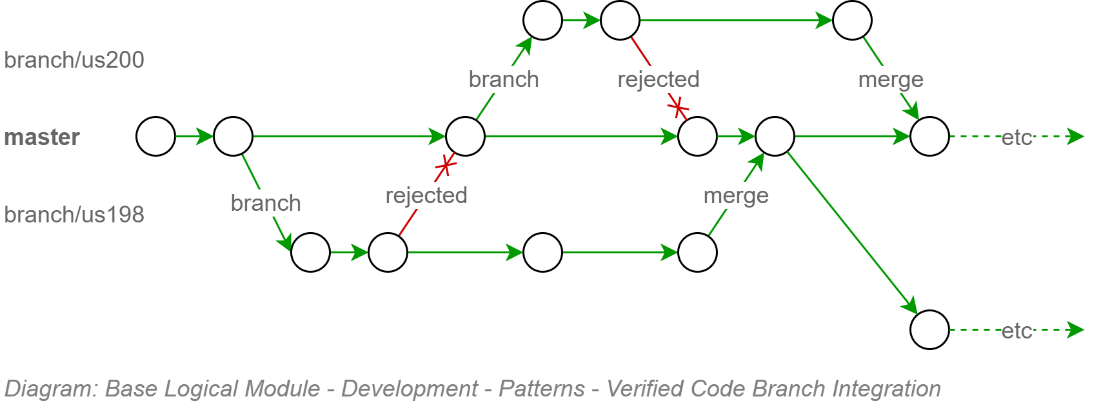


Figure 1: Validated Branch Merging

# Validation by Automated Tests

Validation of the code for containing security credentials is but one of the tests that should be run over the committed code.

While it is unreasonable to expect all forms of validation be implemented in a pipeline by day one, it is reasonable to expect the pipeline’s logic to evolve over time, taking over more and more the responsibilities of validating the code before committing it to the common pool that all developers take new branches from.

The following categories are recommended for testing by automation.

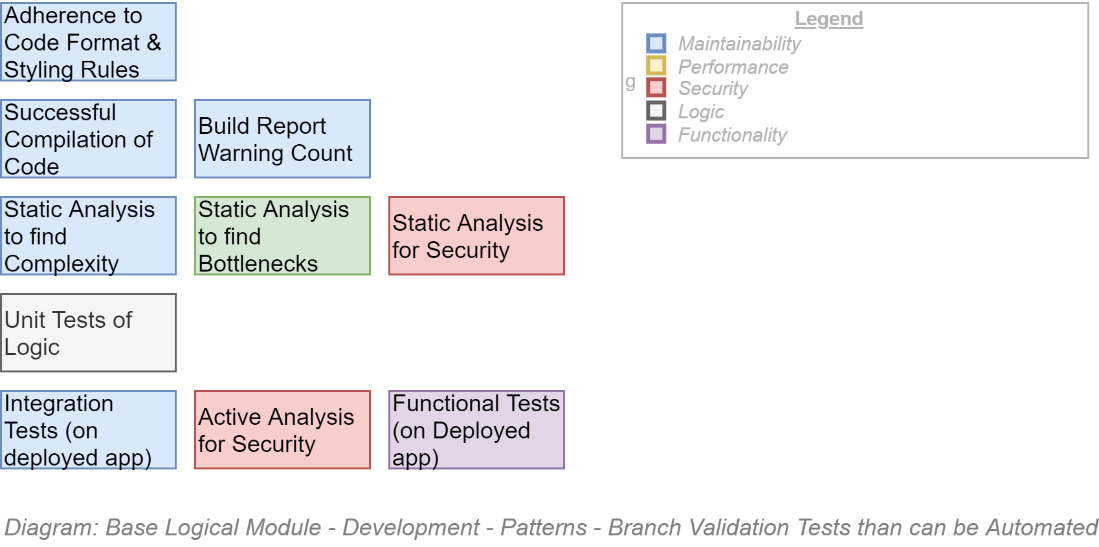


Figure 2: Branch Validation Aspects

The project is not expected to automate every test possible before beginning to develop the information system – but the project manager and dev team lead are expected to put aside resource time to implement in an ongoing continuous manner till they are completed.

**Note:**  
The importance of these automated tests are two-fold:   
- ensuring the quality of code during the main upfront development phase,  
- putting in place before handover to different (support) developers the automated safeguards required to ensure the initial quality of the product does not degrade over service’s whole service lifecycle duration.

# Conclusion

This document summarises expectations on code repository setup and automation of maintenance to decrease the risk of it becoming a liability in terms of security and maintainability, instead of remaining a valuable asset over the service’s full service lifespan.

Appendices

Appendix A - Document Information

### Versions

* 1. Initial Version

### Images

**No table of figures entries found.**

### Tables

**No table of figures entries found.**

### References

**There are no sources in the current document.**

### Review Distribution

The document was distributed for review as below:

|  |  |
| --- | --- |
| Identity | Notes |
| Sandy Britain, Enterprise Architect |  |
| Matt Duguid, DevOps Specialist |  |
| Amy Orr, Data Domain Architect |  |

### Terms

Refer to the project’s Glossary.

##### IT

: acronym for Information, using Technology to automate and facilitate its management.

##### ICT

: acronym for Information & Communication Technology, the domain of defining Information elements and using technology to automate their communication between entities. IT is a subset of ICT.

1. Usually reliant on Moqing or other system to permit testing the code independently of dependencies. [↑](#footnote-ref-2)
2. If using an ORM, use the ORM to develop the schema description to ensure both tiers are in agreement. [↑](#footnote-ref-3)