Contents

[Development 2](#_Toc152665724)

[Guidelines 3](#_Toc152665725)

[Versioning Strategy 3](#_Toc152665726)

[Versioning Product Suites 4](#_Toc152665727)

[Versioning Feature Flags 4](#_Toc152665728)

[Source Code Branching Strategy 5](#_Toc152665729)

[Main or Code Flow Branches 5](#_Toc152665730)

[Supporting or Temporary Branches 6](#_Toc152665731)

[Code Review & Merge Process 8](#_Toc152665732)

[Code Quality Guideline 10](#_Toc152665733)

[Release Management Guideline 12](#_Toc152665734)

[Secure Development 15](#_Toc152665735)

[Templates 17](#_Toc152665736)

# Development

This is the central hub or the repository for the overall engineering quality related information and the engineering team should always refer to this for any engineering quality related queries or updates. And hence the engineering team should always make sure that the information shared is correct and up to date and any updates or changes should be communicated and shared in a timely manner within the organization. The information shared here should be kept as confidential and must not share with any external parties or publish in any public forum.

## Guidelines

All the defined guidelines and the strategies are listed here.

## Versioning Strategy

All the software projects must adhere to these guidelines and must refer to this document for any queries regarding the software versioning.

The version management must be done using the [Semantic Version Specification](https://semver.org/). Normal version number must take the form of X.Y.Z .

X, Y & Z are non-negative integers and must not contain leading zeros.

X – Major version. Must increment when incompatible / breaking changes are introduced.

Y – Minor Version. Must increment when adding backward compatible features.

Z – Patch version. Must increment when making backward compatible bugfixes.

Major version zero (0.Y.Z) is for initial development releases. Hence the initial development release must be started with 0.1.0 and the minor (Y) and the patch (Z) versions must be incremented accordingly for all the subsequent development releases prior to the first production release.

Version 1.0.0 must be used as the initial public release.

Patch version Z must be incremented with the backward compatible bug fixes.

Minor version (Y) must be incremented with the backward compatible functionality or feature additions or implementations of change requests (CRs). Further when batch of bugfixes are incorporated and if the change is not trivial, then the minor (Y) version must be incremented instead of patch version. When incrementing major version (Y), the patch version (Z) must be reset to zero.

Major version (X) must be incremented when backward incompatible or breaking changes are introduced with the new release. Here both the minor (Y) and the patch (Z) versions must be reset to zero.

A pre-release version may be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers must comprise only ASCII alphanumeric and hyphens [0-9A-Za-z-]. Identifiers must not be empty. Numeric identifiers must not include leading zeroes. Examples: 1.0.0-alpha.1, 1.0.0-0.3.7

 Build metadata may be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch or pre-release version. Identifiers must comprise only ASCII alphanumeric and hyphens [0-9A-Za-z-]. Identifiers must not be empty. Build metadata must be ignored when determining version precedence. Examples: 1.0.0-alpha+001, 1.0.0-beta+exp.sha.5114f85

### Versioning Product Suites

When versioning product suites which includes one or more subcomponents or products, the main or the product suites must have a separate version number adhering to this version guideline and rest of the subcomponents can have their own version numbers and they can be evolved independently. But should not share or mentioned them in the product suite production releases. Only the suite’s version information must be included instead.

Examples: Let’s consider a real-world example from one of Company ongoing client project here. The MGT project, we have a product suite which includes four subcomponents.

MGT Product Suite

 MGT Connector API

 MGT Heartbeat Application

 MGT Floating Icon Application

 MGT Connector Host Controller Application

In this case initially we will release the initial production release of the product suite as 1.0.0. And the rest of the subcomponents can have their own production versions. But we will not share the subcomponent versions to the public, instead should use that information internally when tracking source code branches, bugs or features and internal releases.

#### Mobile Application Display Version

The mobile application releases must have a separate display version number which is not compatible with the Semantic versioning schema. But it must be used only for the display purpose only and always the releases must have the version numbers compatible with Semantic versioning schema. In mobile application releases, this is common practice or guideline used, when the [CodePush](https://github.com/microsoft/react-native-code-push) approach is used to update the application without introducing a new release. This approach must be used when CodePush is possible for any hot fixe for a production release. In this case, the release version remains the same, but since the users get the new update via CodePush that must be reflected in the display version.

Examples: Release version 1.0.0, Display version 1.0.0.00, once the hot fix is published via CodePush the to the users’ devices, release version remains as 1.0.0, but the display version is changed to version like 1.0.0.001 / 1.0.0.002. Here users are able to see that they are having small updates on top of a given production release by looking at the display version.

### Versioning Feature Flags

When using [feature flags](https://www.atlassian.com/continuous-delivery/principles/feature-flags), to define the version of a given feature, it is not a must to consider Sematic version schema as it is used internally. Hence version format like explained in the display version section shall be implemented.

Examples: Feature A might have multiple versions like 1.0.0.005, 1.0.0.006, here version 1.0.0.005 can be targeted for user segment S1 while version 1.0.0.006 is targeting the user segment S2.

## Source Code Branching Strategy

There are two types of branch categories.

 *Main or code flow branches* - These are the branches that follow the code flow of the code changes from the development to production. They are permanently available on the repository and have infinite lifetime.

 *Supporting or temporary branches* - These branches are created with a specific goal and have a limited lifetime. Hence, they are destroyed when they are no longer needed.

### Main or Code Flow Branches

As mentioned above following four main or code flow branches should be permanently available throughout the lifetime of the repository and further, each branch directory mapped to a specific deployment environment as they follow the code flow from initial development to the production release. Direct commits to these branches are not allowed and all the work should be done by creating separate supporting branch. They should be incorporated or merged to the given code flow branch via a pull request (PR) or merge request (MR). Hence appropriate branch protection rules should be applied to each code flow branches. For test codes, only the develop and the main code flow branches should be created, and all the branch specific guideline and rules should be considered unless explicitly specified any exceptions here.

*develop* - All the development work should be branched off and brough into this branch. All the new features developments, bug fixes and any other work not included to the next planned release should be branched off from this and create a separate supporting branch accordingly.

*qa* - This branch contains all the codes ready for quality assurance (qa) testing. For any qa reported issues, new supporting branch should be created by branching off from the *qa* branch and merged once the fix is completed. Finally, these fixes should be merged to *develop* branch.

*staging* - This branch contains all the tested features and bugfixes ready for demo, user acceptance testing (UAT) or the approval for stakeholders prior to the next planned production release. For any issues reported at this stage, new supporting branch should be created by branching off the *staging* branch and merged once the fix is completed. Finally, these fixes should be merged to *develop* branch.

 *main / master* - This branch is for the production and hence this should only contain the latest production code. For any issues reported from the production, new supporting branch (hotfix branch) should be created from specific version or the tag of the production branch and merged once it is ready to release. Finally, these fixes should be merged to *develop* branch. Further, merging to *qa* or *staging* branch should be done if there is an active release is in progress. This branch should be protected by locking (read-only) and should only be allowed to update when ready for a production release.

### Supporting or Temporary Branches

These branches should be created by branching off from one of the code flow branches (develop, qa, staging & main) mentioned above. Only the specific set of supporting or temporary branches are allowed in certain code flow branches.

#### Naming Guideline

All the supporting branches except build, release and merge supporting branches should be created with their standard prefixes and the format given below.

*[prefix]/[ticket system prefix]-[ticket id]\_[work item description]*

The hyphen character should be used to separate words in the work item description. The work item description should be short and self- explanatory. Note that the ticket information and the work item description must be separated by the underscore character. And should use lower case letters for words.

For build and merging supporting branches, following format should be used.

*[prefix]/[work item description]*

For release supporting branches, following format should be used.

*[prefix]/[version number]-[prerelease suffix][pre release number with leading zero]*

The version number should be minor, major and the patch component defined in the [versioning strategy guideline](https://inivos.atlassian.net/wiki/spaces/T/pages/128221250). Allowed prerelease suffixes are alpha (for alpha testing), beta (for beta testing) and rc (for release candidate testing) while the prerelease version number should be versioned sequentially with leading zero.

Examples: feature/jira-123\_login-module, bugfix/jira-001\_home-page-layout-fix, merge/combine-user-login-backend-frontend, build/code- coverage-test-run, release/1.2.0, release/1.3.0-alpha01, release/2.3.0-beta02, release/5.0.0-rc01

#### Feature Branches

Any new feature, module or development should be done on a new feature branch and as the development is based on *develop* code flow branch, new feature branch should be branched off from the it. The prefix ‘feature’ should be used when creating a feature branch and they can only be created by branching off the *develop* code flow branch*.* They are not allowed to create with other code flow branches (*qa, staging & main / master*). Hence branch naming rules should be enforced to make sure that correct naming standards are followed by the teams.

Examples: feature/jira-500\_user\_management, feature/jira-400\_user-profile-module

#### Bugfix Branches

Any issues or bugs reported should be fixed in separate supporting branch by branching off from a given code flow branch. The prefix ‘bugfix’ should be used when creating a bugfix supporting branch and can only be branch off from one of the code flow branches except main or master (*develop*, *qa*, *staging)* or any release branch targeting for a release. Further, any bugfixes done during a test or UAT stage should be merged into the *develop* branch.

Examples: bugfix/jira-100\_bulk-update-performance-fix, bugfix/jira-002\_login-error-typo-fix

#### Hotfix Branches

When the production issues reported, hot fix branches are created by branching off the specific version or the tag. The prefix ‘hotfix’ should be used when creating this type of supporting branch and can only be branch offed from the *main / master* (the production) code flow branch. The hot fixes should be merged into the *develop* branch and then into qa or staging branches if any release is in progress.

Examples: hotfix/jira-888\_broken-user-profile-fix, hotfix/jira-700\_invalid-configuration-fix

#### Experimental Branches

Any research & development (R&D) work or proof of concept (POC) which are not part of a release should be created as an experimental supporting branch. The prefix ‘experimental’ should be used when creating this type of supporting branch and can only be branch offed from the *develop* code flow branch.

Examples: experimental/jira-005\_web-socket-poc, experimental/jira-009\_reaseach-on-open-source-img-processing-libs

#### Build Branches

These are the supporting branches for creating specific build artifacts or for doing code coverage runs. The prefix ‘build’ should be used when creating this type supporting branches.

Examples: build/demo-configuration, build/code-coverage-test-run

#### Release Branches

These types of branches are created for specific release of component or system. The prefix 'release' should be used when creating this type of supporting branches and can only be branch offed from the *develop* code flow branch.

Examples: release/1.2.0, release/1.0.0, release/1.3.0-alpha01, release/2.3.0-beta02, release/5.0.0-rc01

#### Merging Branches

The merging supporting branches are used for resolving merge conflicts. The prefix ‘merge’ should be used when creating this type of supporting branches and can only be branch offed from the *develop* code flow branch.

Examples: merge/combine-user-login-backend-frontend, merge/combine-user-management-sub-modules

## Code Review & Merge Process

Any work that is targeting for a release should always be reviewed and approved by the one or more members who are either component or code owners before releasing. For any test or test automation related work also should govern under these same policies and hence they should be reviewed and approved always even though they are not deployed to the production systems. It is mandatory to implement branch protection rules in each source code repository to enforce these policies and guidelines.

New feature or component development should provide a feature design document details along with the code review request so that the reviewers can validate the implementation with the design. For performance and security features or issue fixes, authors should provide performance test execution reports and the security test execution reports information to the reviewers when submitting the code review requests. If any of those mentioned above is not available with code review requests, the reviewers should request the required information from the authors and should not approve the changes until the enough information is available before approving. Further, it is a must to review and approve any work regardless of the size and the complexity of the work. The lead of the development team is responsible for code reviews and he or she can add any other senior members within the team or outside the team to the code review team.

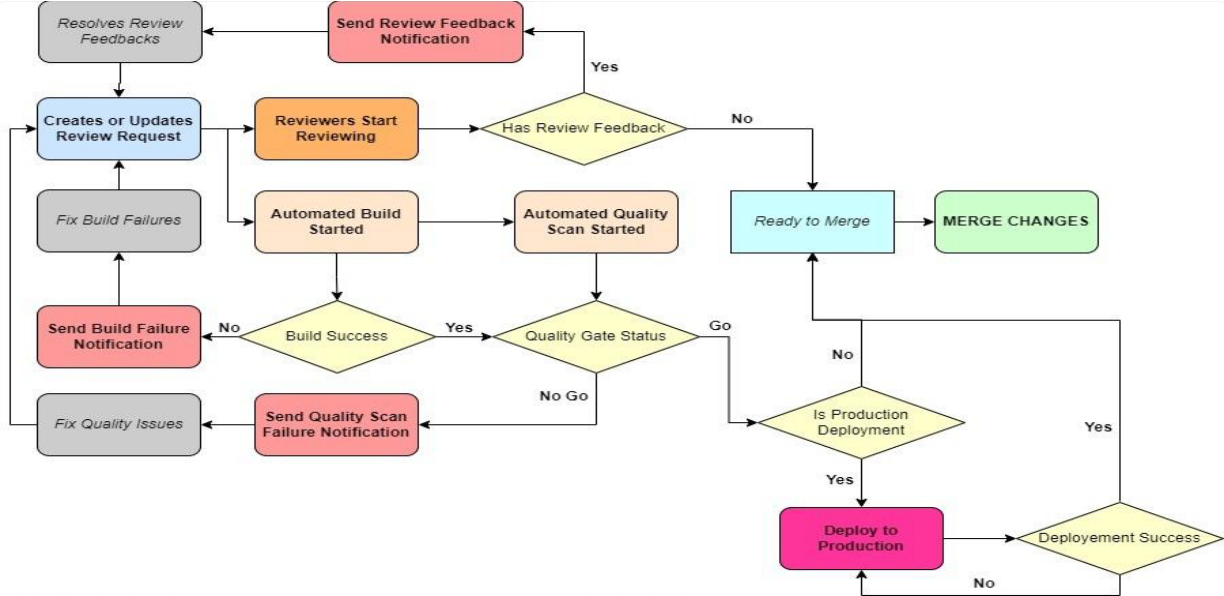
All the teams should collect or keep code review statistics as they are using to evaluate the quality aspect of the deliverables of each individual. Hence it is a must to have at least once a month code review meeting for entire team to discuss the review findings of members of the team, possible improvements and plan trainings if required. Hence team leads or technical leads should spend some quality time on these tasks and should consider as high priority lead task.

Apart from above mentioned manual checks or verification it is a must following automated checks, green build, quality gate pass, test execution and the code coverage under the continuous integration pipeline execution while successful deployment is a must when merging release branch to the production branch (main / master). Based on the verification results, merging the pull request should be automatically allowed or blocked.

Note: For more information about release management and the code quality refer to the [release management guideline](https://inivos.atlassian.net/wiki/spaces/T/pages/128221415) and [code quality](https://inivos.atlassian.net/wiki/spaces/T/pages/128221361) [guideline](https://inivos.atlassian.net/wiki/spaces/T/pages/128221361) respectively.

#### Process Flow

Following diagram illustrates the code review process flow. The branch naming validation checks are not included to the diagram for simplicity.

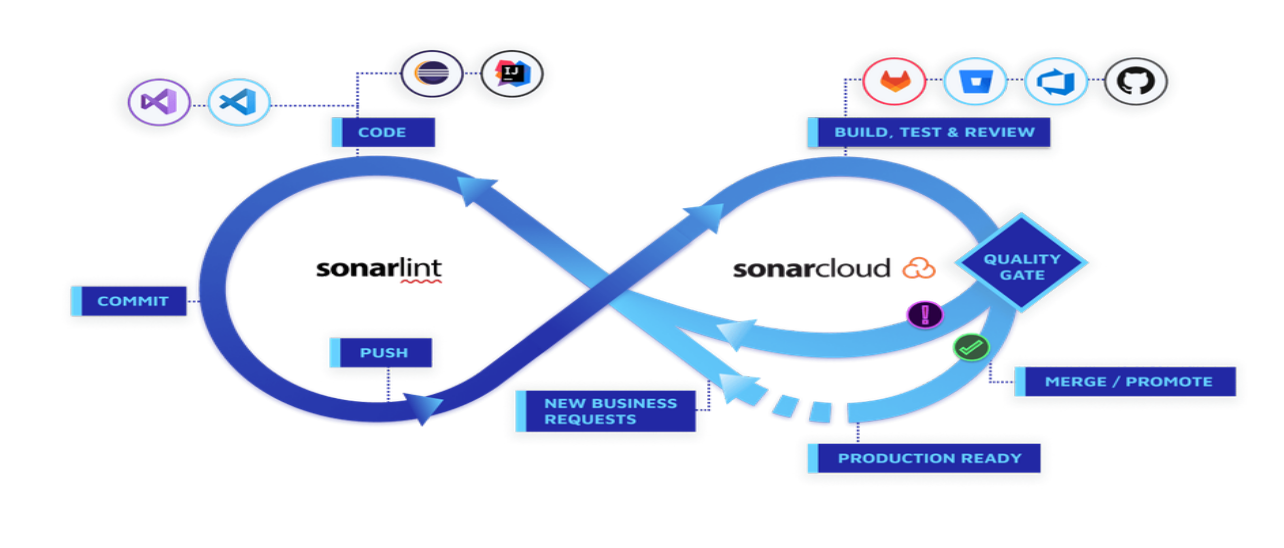


## Code Quality Guideline

#### Guidelines

The [SonarLint](https://www.sonarsource.com/products/sonarlint/) static code analysis tool extension for the integrated development environments (IDEs) or editors should be installed and configure so that the first line of defense will resolve most of the issues or problems. Further the [connected mode](https://www.sonarsource.com/products/sonarlint/features/connected-mode/?gads_campaign=SL-Mroi-Brand&gads_ad_group=SonarLint&gads_keyword=sonarlint&gclid=EAIaIQobChMIh9igjfeEgAMV2ml9Ch1rIgEeEAAYASABEgIdEPD_BwE) should be configured if supported for the IDE or editor where the rules and the analysis settings are automatically, and real time synchronized from [SonarQube](https://www.sonarsource.com/products/sonarqube/) or [SonarCloud](https://www.sonarsource.com/products/sonarcloud/) to SonarLint which will make sure that the development teams have the exact rules and settings in their local development setup.

All the code repositories should be integrated with the [SonarCloud](https://www.sonarsource.com/products/sonarcloud/) solution, and the quality scan results should be used to implement the quality gates for go or no-go decision for code review requests (pull requests or merge requests). Hence continuous integration pipelines should integrate with the SonarCloud so that every change is scanned and going through the quality gate regardless of the size or the complexity of the change.



Sonar Ecosystem in Code Quality Process

For more details on SonarCloud refer to the official [documentation](https://docs.sonarcloud.io/).

The code coverage should be greater than or equal to 80% and for any projects not meeting this should request special request to change the threshold for a certain time period with the plan on how to increase the code coverage of the projects. No one in the team except the lead or technical lead should allow to or given permission to disable or off the quality scan or the quality gate checks. Hence all the code

review requests (pull requests or merge requests) and the builds should be validated make sure that the quality of the releases or deliverables are up to the expected level and consistent. Hence these checks should be implemented and enforced in the continuous integration pipelines.

Finally, the single source of truth for the code quality will be the SonarCloud dashboard projects summary view and hence the integration from SonarLint to SonarCloud is a must for all the ongoing and future projects.

Develpment teams should have at least once a month code review meeting for their teams and should discuss the code review findings (manual code review and static code analysis details) for each individual and have a plan on improving the code quality of the team members as quality of the deliverables is one of the key performance indicators (KPIs) of each individual as well as the team. The lead of the team is responsible for improving the code quality of the team and hence should spend some considerable lead time on above mentioned tasks.

## Release Management Guideline

#### Guidelines

##### General

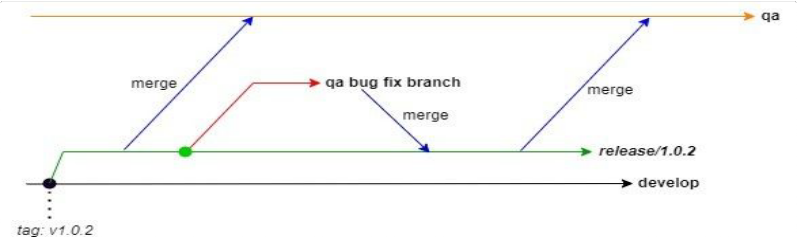
This guideline explains how a given release of component or subsystem or system should be managed and guides with the relevant instructions and standards that should be followed by the teams.

As mentioned in the [branching strategy guideline](https://inivos.atlassian.net/wiki/spaces/T/pages/128221283) a new release should be created from the *develop* code flow branch. when creating a particular release, this checkpoint must be tagged in the develop branch and the tag format should be [v]X.Y.Z-[prerelease suffix][prerelease number with leading zero] and the X, Y, & Z represents the minor, major and patch components mentioned in the [versioning strategy](https://inivos.atlassian.net/wiki/spaces/T/pages/128221250) [guideline](https://inivos.atlassian.net/wiki/spaces/T/pages/128221250). For prerelease suffixes and number refer to the [branching strategy guideline](https://inivos.atlassian.net/wiki/spaces/T/pages/128221283). Examples: v1.0.0, v1.2.0-alpha01, v5.0.0-rc02



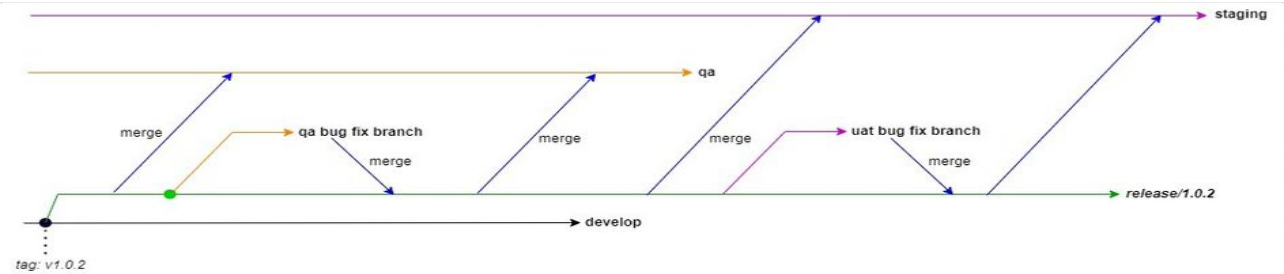
Release Branch Creation

Once a given release is created, then that release branch should be merged into the *qa* code flow branch once it is ready for quality assurance testing. The merge operation should still be adhered to the code review and merge process guidelines. Then if any qa bugs reported, then again, they should be fixed by creating separate bugfix branches from the release branch. Once the issues are fixed, they should be merged to the same release branch and redeploy the fixes for the further testing.



Fixing QA Bugs

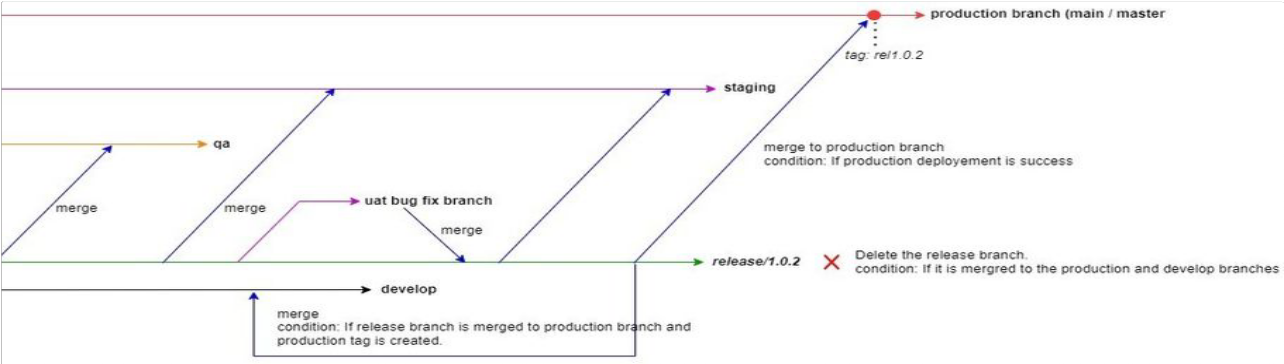
Once the quality assurance test is completed and passed, the same release branch should be merged to the staging code flow branch for user acceptance testing (UAT). If any UAT issues reported, again those bugs should be fixed in separate bug fix branches by branching off from the same release branch and redeploy them to the staging environment for further testing.



Fixing UAT Bugs

Once the UAT testing is completed and approved for the production release by the stakeholders, first the release branch should be deployed to the production environment and verify it to make sure that the new production deployment is working as expected. Now the release branch should be merged into the production branch (master or main) and the new production release should be tagged in the production branch. The tagging format is same as for tagging, but with no prerelease information and the release prefix should be ‘rel’. Examples: rel1.0.2, rel1.0.0, rel3.0.1

Note: Now the release branch can be deleted as it no longer needed.



Production Release

##### Hotfix Process

The hotfix releases are not preplanned are made to address the critical bugs or issues in the current production release. A hotfix is a quick fix for a significant bug that affects the majority of the users of the system for which there is no reasonable workaround. Hence should consider followings questions or criteria when deciding to go for a hotfix.

 Is the bug significant?

 Does it produce incorrect results?  Is existing feature broken?

 Is the bug affecting to the majority of users?

 Is it a common or major functionality failure?

 Is it an edge or corner case?

 Will the average user encounter the bug during normal usage?

 Is there a reasonable workaround?

 Can a potential workaround be communicated effectively to the users?  Will the average user understand the workaround?

 How many steps the workaround takes?

 When the bug is fixed, will the workaround continue to work?

 Is it possible to include this fix to next release?

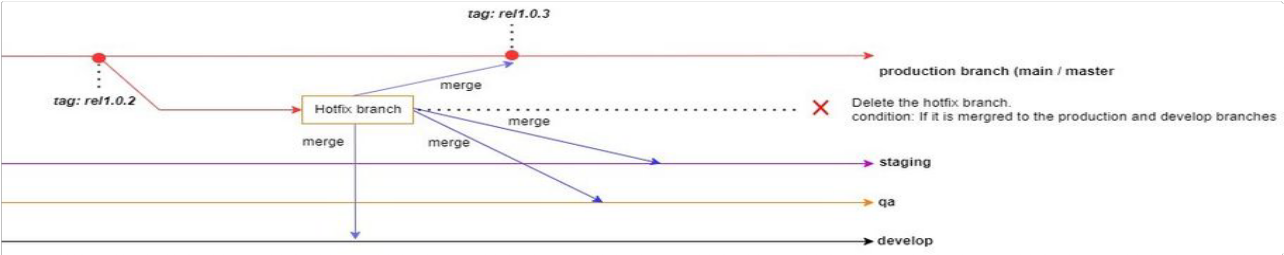
 Is the next release within a few days?

 Does the bug affect the entire user experience?

Checks the above criteria when considering a hotfix and the fix for these types of bugs should not be the proper fix as it is critical to fix them immediately, hence the developers should do a quick fix with minimal code changes within a limited time period. But still should update the relevant documentation and run unit tests. Further if the fix is not the proper fix, separate bug should be created to track it.

It is not mandatory to go through the standard testing cycles (QA and UAT) here as the it is critical to deploy the fix the production as quickly as possible. The hotfix branch should be branch offed from the correct version or the production release tag and once the fix is completed and verified the fix can be deployed to production and the hotfix branch should be merge to the production branch (*main or master*) and the *develop* branches. The production branch should be tagged with the new release version information and finally the hotfix branch can be deleted as it is no longer required.

Normally the patch component of the previous or issue reported production version is incremented for a hot fix release. But it is not a must even the minor version can be incremented if the change is significant. For more information about versioning refer to the [versioning](https://inivos.atlassian.net/wiki/spaces/T/pages/128221250) [strategy guideline](https://inivos.atlassian.net/wiki/spaces/T/pages/128221250).



Hotfix Release

## Secure Development

All the versioned templates under different areas are listed here. All new documents should be created with the latest version of the templates while existing documents shall continue with the older versions. It is recommended to migrate the documents to the latest version of the template, and it is a must when the older versions are discontinued.

## Templates