Management Document

Software Configuration

Summary

Introduction

This document aims to establish a set of standards and models for software configuration management (SCM), intended for teams responsible for developing Front-end, Back-end and Artificial Intelligence areas. Software configuration management is an essential discipline in software engineering, whose purpose is to ensure the organization, systematic control and traceability of the artifacts generated throughout the life cycle of a software project.

GCS comprises a set of methodological processes aimed at describing, controlling and managing changes in software artifacts, such as source codes, technical documentation, prototypes, data, models and other critical components. Effective management of these artifacts is essential to ensure the integrity, consistency and orderly evolution of software, especially in contexts of collaborative and multidisciplinary development.[1].

In projects involving collaboration between multiple teams and the integration of different technologies, the adoption of SCM practices is essential. Such practices not only facilitate coordination between the various stakeholders involved, but also provide mechanisms for traceability of changes, maintenance of product consistency and the ability to respond to emerging requirements in an agile and controlled manner. Furthermore, SCM helps to mitigate common risks, such as version conflicts, loss of information and code inconsistencies, which can compromise software quality and delivery.

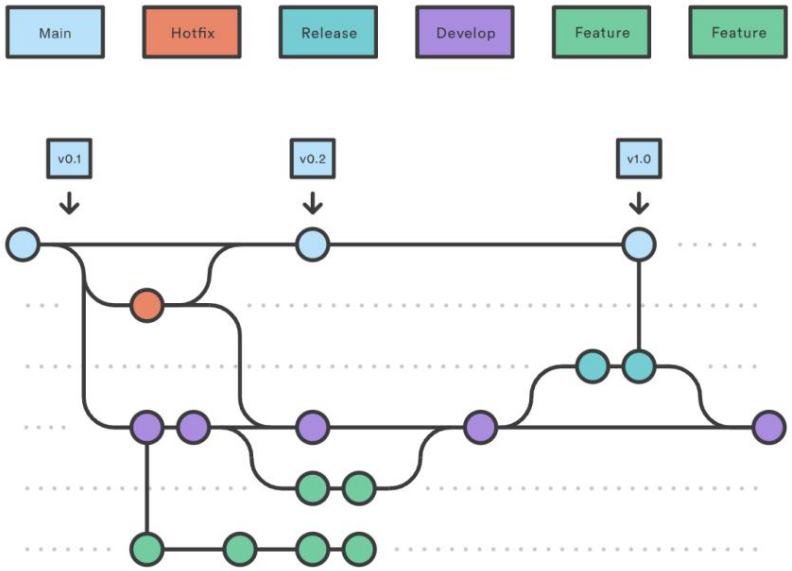
The application of the guidelines presented in this manual aims to promote sustainable collaboration between teams, ensuring transparency, shared responsibility and efficiency in change management. This document was prepared with the intention of serving as a technical and practical reference, aligned with the specific needs of the project in question. By adopting the recommendations outlined here, teams will be able to face the challenges inherent to software development in a structured and coordinated manner, ensuring the quality and consistency of the results delivered.

[1] ***https:// www.atlassian.com/ us/ microservices/ microservices-architecture/ configuration-management***

Configuration Management Software (Front-end/Back-end)

Branching model

Git Flow Template Branches



Source: Git Flow - workflow

Git Flow is a workflow model for software development using the Git version control system. The key principles of Git Flow are:

Branches Each new feature, bug fix, or task is developed on a separate branch. This practice allows multiple people to work on different parts of the project simultaneously without interfering with each other's work.

Frequent commits Developers are encouraged to commit frequently to their local branches as they work on a feature or fix. This helps maintain a detailed history of progress and allows them to return to earlier points in development and allow other team members to see changes in progress.

Pull Requests When a developer completes a feature or fix on their branch, they open a Pull Request (PR) to merge their changes into the main branch of the repository. A PR is a way to start a discussion with other team members, review the code, and provide feedback before merging the changes.

Code reviews	Team members can review code submitted via ***Pull Requests.*** They can provide feedback, request changes, or approve the PR. Code review is an important step in ensuring code quality and sharing knowledge among team members.

Branches

main	The ***main branch*** is reserved for stable versions of the project. At the end of each ***sprint,*** it is recommended to merge the changes made to it with the ***develop branch.*** The version will be in the production environment.
hotfix	The hotfix ***branch*** is an emergency solution to fix critical issues in the stable ***(main)*** version without having ***to*** wait for the normal development cycle. It allows fixes to be quickly applied to production, while maintaining project stability.
develop	The ***develop*** branch contains a newer development version of the project. Because it is constantly updated with other ***feature branches,*** it may be less stable.
release	The ***release branch*** is created from ***develop*** when the project is about to be finalized for a new stable version. It serves to prepare and test the version before it is officially released, allowing for ***bug*** fixes and final adjustments. After approval, the changes from the ***release branch*** are merged into the ***main*** branch and then into ***develop*** to keep all ***branches*** in sync. The version is then in approval.
feature branches	These are ***branches*** created to implement new features or fix specific problems in the code. These branches allow developers to work on their changes in isolation before integrating them into the main development.

How to use

To make any changes, you need to create a branch from the develop branch, but it is important to check if there is an open branch with the same objective as your activity. In addition, only “hotfix” or “release” branches can be implemented directly on the main branch, for other needs you need to follow the following steps:

1. **Create a new branch:** When adding new functionality to your project, start by creating a new branch from the main development branch. A valid name for your branch would be ***“feature/ 123/ add-header”,*** where ***“feature”*** is the type of ***branch*** you created, “123” is the issue number , and “add-header” is a brief description of the purpose of the branch (more details on branch naming patterns in the “Naming Standards” section).

2. **Implement your changes:** In your new branch, make the necessary changes to the code to perform your activity. It is also important to test your changes to ensure that they work correctly and do not introduce problems into the project. It is recommended to make frequent and descriptive commits, breaking the work into small, logical changes. Remember to use semantic committing, as defined in the 'Managing commit messages' section .
3. **Open a pull request:** When you're ready to submit your changes for review and merge them back into the main branch, open a pull request. A pull request is a request for code review, where you can discuss the changes with other team members, perform code reviews, and resolve any issues that are identified (more details on creating pull requests are provided in the 'Managing Pull Requests' section); 4. **Pull Request Review:** During the pull request process, other team members can review your changes, provide feedback, and suggestions. You can discuss the changes and iterate on them as needed;

5. **Merge the Pull Request:** Once you have received approval and completed all necessary reviews and testing, you can merge the pull request into the development branch. This will incorporate your changes into the main version of the project; 6. **Pull Request Approval:** At least two people on the team must review another member's opened PR for merging into main; 7. **Merge:** Once the PR has two or more approvals, any team member can merge it, including the one who opened the PR.

Standardization of nomenclature

In order to maintain consistent and organized naming, start the branch name with the type of activity the branch is intended for, then inform the ***issue*** that deals with the modification to be made; ending with a brief description of the purpose of the branch.

<branch type>/ <issue number>/ <description>

The branch name must contain only lowercase alphanumeric characters. Words that make up the issue description must be separated by the character "-" (hyphen).

Example: *feature/ 12/ add-header*

Change management

Before performing an experiment or change, an ***issue*** must be opened that allows the identification of the objectives, methods and context of the changes. ***Issues*** must follow a writing pattern that contains information that allows team members to identify the context of the change, what need will be met by it, and what are the possible solutions, or methods used to carry them out, to meet the needs described, within the context in which they arise.

Issue composition

The issue produced may be composed of the following elements:

Title	A short, descriptive title that summarizes the functionality to be implemented. It should be clear and specific, avoiding generic or vague terms.
Context	A description of the current situation and the problem or opportunity that justifies the change. Explain why the change is necessary and what problem it is intended to solve. This may include information about the current impact (e.g., costs, time, errors, usability) and how the proposed change would improve the situation.
Description	A detailed explanation of the change to be made. Describe what the change will do, what data or processes will be affected, and what the expected results are. If it is an improvement or adjustment to an existing functionality, briefly describe the current functionality and the aspects that need to be improved. Include examples or use cases if necessary.
Proposed solution	A high-level description of the possible solutions or methods for implementing the change. Include steps or technical approaches that will be used. If there are multiple alternatives, briefly describe them and justify your choice of the proposed solution.
Criteria of Acceptance	A list of requirements or criteria that must be met for the issue to be considered complete. Especially in the case of changes that impact existing functionality, the criteria should be detailed and clear.
Information Additional	Links, references or observations that may assist in development. Technical documentation, code examples, or links to relevant tools and libraries may be included. If there is no additional information, there is no need to create an issue with this item.

The following example demonstrates filling in the fields with possible values for an ***issue.***

Title	Implement pagination in the process listing
Context	Currently, the process listing is displayed on a single page, which causes slowness and makes navigation difficult when there are a large number of records. Implementing pagination will improve the performance and usability of the interface.

Description The functionality consists of adding pagination to the user listing, allowing data to be displayed in smaller, navigable pages. Pagination should include: - Configurable limit of items per page
(e.g.: 10, 20, 50).
- Navigation between pages (previous, next, first, last).
- Display of the total number of pages and records.

Proposed solution
1. Develop a reusable pagination component.
2. Modify the API endpoint to support pagination (e.g. page and limit parameters).
3. Integrate the pagination component into the existing listing users.
4. Perform tests to ensure functionality and responsiveness.

Criteria of Acceptance The user listing should be displayed in pages, with a configurable limit of items per page. It should be possible to navigate between pages and view the total number of records. The functionality should be responsive and work correctly on mobile devices.

Information Additional
Suggested pagination library documentation: [link]
Example of pagination implementation in Vue.js: [link]
Issue (from the back-end team) related to changing the API of the service used for process listing: [link]

Pull Request Management

The code contained in a ***branch*** will be merged into the target branch ***(develop/ main)*** **if, and only if,** approved by at least **two members,** preferably those who are not directly involved in its development and production.

The title of the ***pull request*** must be the same as the name of the main ***commit*** made.

Commit message management

Format

Commit messages should follow this format: <type>: <subject>

Where

Type A word that describes the nature of the change (e.g. "feat", "fix", "docs", etc.).

Subject A brief description of what was done (e.g. "adds validation of file size").
dataset").

Types

Commit message types must be one of the following:

feat	A new feature added.
fix	Changes that affect the build system or external dependencies, a bug fix.
perf	A code change that improves application performance.
test	Add missing tests or fix existing tests.
refactor	A code change that neither fixes a bug nor adds a feature, but makes the code cleaner, more efficient, or better documented.
docs	Changes made only to the documentation
style	Changes that do not affect the behavior of the code (whitespace, formatting, missing semicolons, etc.),
build	Changes that affect the build system or external dependencies.
ci	Changes to our CI configuration files and scripts (examples of scopes: Travis, Circle, BrowserStack, SauceLabs).
revert	Revert a previous ***commit .***
conflict	Conflict resolution.

Subject

The subject should be a clear and succinct description of what was done. The ***commit*** should be starting with a verb in the present indicative, and all letters must be lowercase.

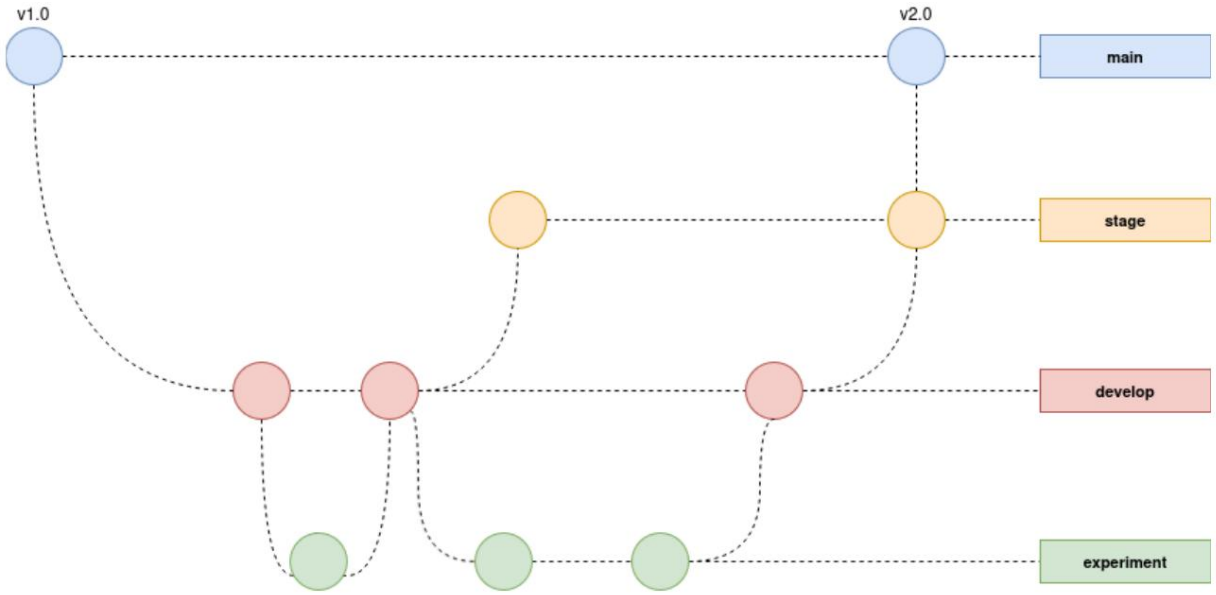
Commit example

feat: adds the application header

Configuration Management Software (AI)

Branching model

Branches for versioning AI pipelines



Source: Authors

The branching model developed is based on the workflow ***model***
Git Flow, which specifies branching patterns for software development using the Git version control system. The key principles of the model are:

Branches	Each new experiment, feature, ***bug*** fix , or task is developed on a separate ***branch .*** This practice allows multiple people to work on different parts of the project simultaneously without interfering with each other's work.
Commits frequent	Developers are encouraged to ***commit*** frequently to their local ***branches*** as they work on a feature, fix, or experiment. This helps maintain a detailed history of progress and allows them to return to earlier points in development and allow other team members to see changes in progress.
Pull Requests	When a developer completes a feature, fix, or experiment in their branch, they open a ***Pull Request*** (PR) to integrate the changes into the repository's development branch . Likewise, when the version maintained in the development branch meets the established criteria for the model to be orchestrated in the staging environment, a PR is opened to merge the code into the staging ***branch .*** When the configuration of ***the***

model artifacts are suitable for integration into the main branch and orchestrated in the staging environment, PR should also be used prior to merging. PR serves as a mechanism to promote discussions among team members, review code, and provide feedback prior to integration.

After approval of the PR, any team member can complete the merge.

Change reviews Team members should review changes and experiment results submitted via ***Pull Requests.*** They should provide technical feedback, request changes, or approve the PR. Reviewing changes is an important step in ensuring ***pipeline*** quality and sharing knowledge among team members.

Branches

main Stores stable, consolidated versions of the model and surrounding artifacts

Its pipeline artifacts ensure the production of adequate and functional models, validated for use in production environments. The ***main*** branch should serve as the main reference for ***releases*** and integrations, ensuring reliability and consistency at all stages of the project life cycle.

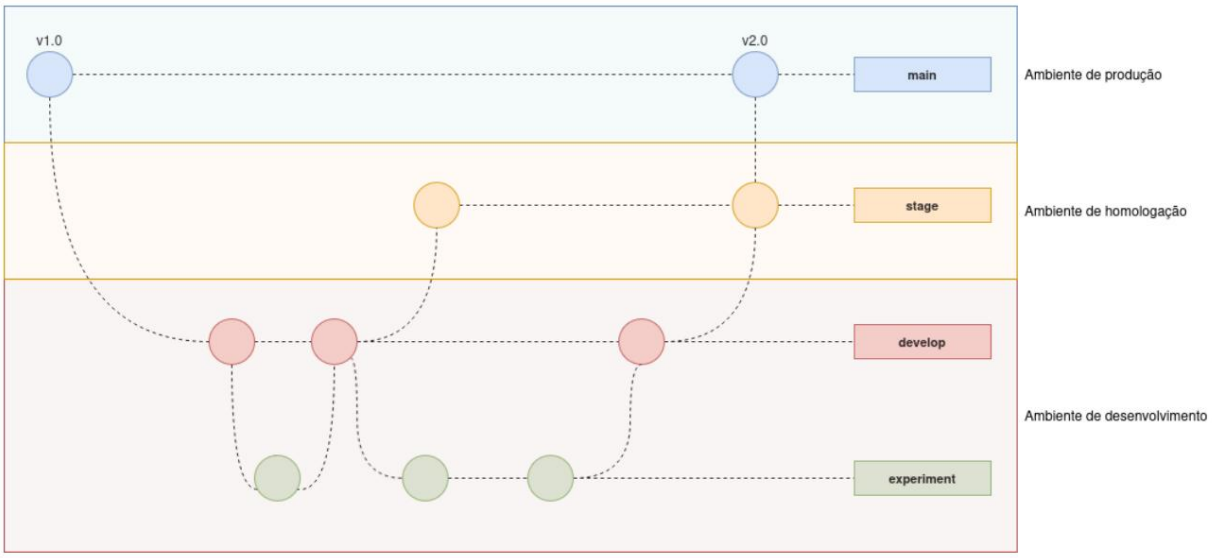
stage Its purpose is to centralize configurations that are suitable for orchestration in a homologation environment. Before being merged into the main branch , modifications must pass through the stage ***branch .*** The pipeline artifacts contained in it ensure the production of suitable and functional models for use in the development environment of other teams, and in the homologation environment.

develop It centralizes all development actions, consolidating code changes resulting from experiments carried out in the experiment branches . When the configuration of the ***develop branch*** reaches a more stable state — that is, the model meets a set of pre-defined assessments and goals adequately — it can be integrated into the homologation branch ***(stage).***

experiment Used to develop new features or experiments in isolation, without impacting the development branch. This approach allows you to safely test and iterate on changes. After completion and validation, the branch can be merged into the development branch.

Experiment branches can have experiment sub-branches if necessary.

Relationship between environments and model branches



Source: Authors

How to use

To make any changes, you need to create a branch from the ***develop branch,*** but it is important to check if there is an open branch with the same purpose as your activity. In addition, only the ***stage*** branch can be merged directly into the main branch, for other needs you need to follow the following steps:

1. **Create a new branch:** When performing a new set of experiments, or features, a new branch should be created from the develop branch . A valid way to name your branch would be “experiment/123/add-lightgbm”, where “experiment” is the type of branch created, “123” is the issue number, and “add-lightgbm” is a brief description of the purpose of the branch (more details on branch naming patterns in the “Naming Standards” section).

2. **Implement the changes:** in your new branch, make the necessary changes to the code to perform your activity. In addition, it is important to test your changes to ensure that they work correctly and do not introduce problems into the project. If there are changes to the ***datasets*** in the develop branch , you can either close the experimentation branch (if it is no longer necessary or feasible with the new data set or format), or you should update the ***datasets*** so that experiments can be developed based on the same data set. It is recommended to make frequent and descriptive commits, dividing the work into small, logical changes. Remember to use semantic commit, as defined in the 'Commit message management' section;

3. **Open a pull request:** When you're ready to submit your changes for review and merge them back into the development branch , open a pull request. A pull request is a request for code review, where you can discuss the changes with other team members, perform code reviews, and resolve any issues that are identified (more details on pull requests can be found in the following sections).

creating pull requests are presented in the 'Pull Management' section.
Requests');

4. **Review the Pull Request:** During the pull request process, other team members should review your changes, provide feedback, and suggestions. You should discuss the changes and iterate on them as needed; 5. **Merge the Pull Request:** After you receive approval and complete all

necessary reviews and testing, you can merge the pull request into the development branch;

6. **Pull Request Approval:** At least two people from the team must review another member's open PR for merging into main (more details on the criteria for approving changes are presented in the 'Approval Criteria' section);

7. **Merge:** when the PR has two or more approvals, any team member can merge it, including the one who opened the PR.

Standardization of nomenclature

In order to maintain consistent and organized naming, start the branch name with the type of activity the branch is intended for, then inform the ***issue*** that deals with the modification to be made; ending with a brief description of the purpose of the branch.

<branch type>/ <issue number>/ <description>

The branch name must contain only lowercase alphanumeric characters. Words that make up the issue description must be separated by the character “-” (hyphen).

Example: experiment/12/add-lightgbm

Change management

Before performing an experiment or change, an ***issue*** must be opened that allows the identification of the objectives, methods and context of the changes. ***Issues*** must follow a writing pattern that contains information that allows team members to identify the context of the change, what need will be met by it, and what are the possible solutions, or methods used to carry them out, to meet the needs described, within the context in which they arise.

Issue composition

The issue produced may be composed of the following elements:

Title	A short, descriptive title that summarizes the functionality to be implemented. It should be clear and specific, avoiding generic or vague terms.

Context	A description of the current situation and the problem or opportunity that justifies the new functionality or experiment. Explain why the change is necessary and what problem it is intended to solve. This may include information about the current impact (e.g., costs, time, accuracy, errors) and how the proposed change could improve the situation.
Description	A detailed explanation of the functionality or experiment to be performed. If the experiment is to develop a new functionality, describe what it will do, what data or processes will be affected, and what the expected results are. If the experiment is to modify some aspect of an existing functionality, briefly describe the functionality and the aspects that need to be improved or adapted (e.g. accuracy, response time). If necessary, include examples or use cases.
Proposed solution	A high-level description of the possible solutions or methods for implementing or modifying the functionality. Include any steps or technical approaches that should be used. If there are multiple alternatives, it is important that these are briefly described and the choice of the proposed solution justified.
Criteria of Acceptance	A list of requirements, or general criteria, that must be met for the ***issue*** to be resolved. Especially in the case of experiments that impact existing functionality, the definition of acceptance criteria must be detailed and clear.
Information Additional	Links, references or observations that may assist in development. Technical documentation, code examples, or links to relevant tools and libraries may be included. If there is no additional information, there is no need to create an issue with this item.

The following example demonstrates filling in the fields with possible values for an ***issue.***

Title	Implement extraction of the 'Applicant's Name' field from “Initial Petition” and “NatJus Opinion”.
Context	Currently, the ‘Applicant Name’ field is manually extracted from procedural documents, which is time-consuming and prone to human error. Automating this task will reduce processing time and increase the accuracy of the extracted information.
Description	The functionality consists of developing an AI module to automatically extract the 'Applicant Name' field from procedural documents. The module should identify the field in different text formats and return the extracted name in a structured format.

Proposed solution	1. Use regular expressions (regex) to identify common patterns in the text - extracting chunks from regions most likely to contain the information. 2. Train an NLP model with an annotated dataset to extract information from text chunks. 3. Implement a function that returns the extracted name in structured format.
Criteria of Acceptance	The module must extract the 'Requester Name' field with a minimum accuracy of 90%; The extracted name must be returned in a structured format (string).
Information Additional	SpaCy can be useful for developing the module. Here is the documentation: [link]; Examples of regex for extracting chunks: [link]; We did something similar to extract the judge's name. We discussed it in the following issue: [link] (pay special attention to the ***fix*** commits for using ***"spacy.load").***

Pull Request Management

The code contained in a ***branch*** will be merged into the target branch ***(develop/ main)*** **if, and only if,** approved by at least **two members,** preferably those who are not directly involved in the development and production of the same. The evaluators **must perform technical evaluations** explicitly in the ***Pull request,*** detailing in discussion the reason for approving or not the merge request.
After discussion and approval by two members, anyone can merge the changes, including developers and reviewers.

The title of the ***pull request*** should match the name of the parent ***commit*** made, or represent the nature of the changes.

Approval criteria

When evaluating a change merge request—whether from an experiment branch ***to*** a develop branch , from a development branch to a ***stage branch,*** or from a stage branch to main **—** reviewers should apply a set of criteria that allow them to verify that the changes are suitable for integration into the target branch.

These criteria should be documented and shared with the team, ensuring that evaluations are consistent and manageable throughout the project lifecycle. Discussions held during Pull Request approvals should be based on the predefined criteria; explicitly stating the considerations used to accept or reject the merge request.

Criteria standards may include checks related to metrics such as accuracy, precision, F1, or SHAP (Shapley Additive Explanations). It is also important to consider

factors such as ***overfitting*** of the model generated by the ***pipeline*** and the cost-benefit in relation to other versions or experiments.

In the case of merging evaluations of a branch of experiments, for example, reviewers should analyze classic metrics, such as accuracy rate, F1 or SHAP, to determine whether there was a significant improvement in the pipeline compared to other existing configurations. If there are multiple experiments seeking to improve a specific aspect of the model, the defined metrics should be used to select the most promising experiment and integrate it into the development branch, promoting the evolution of the model. It is important that the application of the evaluations, when comparing different pipeline configurations, seeks to avoid biases. A relevant practice for this is to use the same set of ***datasets*** to evaluate the experiments; and that, if possible, they are new ***datasets ,*** not previously used in the development operations of the experiments. Similar criteria should be applied in merges between sub-branches of experiments, if necessary. Conclusions about the criteria defined in the ***issue*** to measure the success of the task, as well as other general criteria defined by the team to evaluate merge requests to the ***develop branch,*** such as criteria about coding standards and ***drawback assessment,*** in relation to the current version of the model ***pipeline*** (in ***develop),*** must be presented through technical comments in ***Pull Requests.*** In order to highlight the reasons for accepting or not a modification to the model ***pipeline ,*** carried out through the experiment in question.

When evaluating merge requests for the stage ***branch ,*** it is necessary to consider whether the set of experiments carried out since the last integration between the ***branches*** is sufficient to compose a new increment or improvement. Considering the execution of a ***sprint,*** for example, it is necessary to evaluate whether the modifications represent an increment that adheres to the established criteria so that the modifications can be tested, or shared with teams from other areas of the project. In addition, it is necessary to verify whether the model has adequate compatibility, stability, and performance to be orchestrated in a homologation environment, offering advantages over previous versions of the pipeline. Therefore, the decision to merge the modifications under development by the AI team to the ***stage*** branch should not only consider the concepts and metrics of the AI team members. It is also necessary to consider the objectives and criteria established by the entire team, in order to meet the general objectives of the ***sprint.*** Preferably, one of the reviewers should be a professor, leader, or senior professional on the team. If the change being evaluated generates a MAJOR modification (more details in Semantic Versioning), the merge will also depend on discussion with other teams. Affected teams must be aware of the merge and prepared to adapt to the new version in order to avoid incompatibilities in the application. Conclusions about the general criteria defined by the team to evaluate merge requests for the ***stage branch,*** as well as reports of communication with teams affected by a MAJOR modification, must be documented through technical comments in the ***Pull Request.***

Finally, when evaluating PRs from the homologation branch to the main branch, the criteria may include the analysis of possible inconsistencies, irregular behaviors or ***overfitting*** identified during validations in the homologation environment. It is also important to consider whether the results, although accurate, aroused negative evaluations by the ***stakeholders*** responsible for homologation, or if there were instabilities

or incompatibilities in modules that have the model as a dependency (developed by other teams) when using the staged pipeline version . ***It is*** also necessary to evaluate whether the model pipeline adds secondary effects (such as higher latency in communication with dependent systems, or user response) when tested in an integrated manner with the system. A set of criteria and tests must be applied to determine whether the secondary effects, or other ***drawbacks*** of the model are “acceptable”, in relation to the benefits it offers — as in the case where a pipeline update resulted in much higher accuracy than other configurations, but resulted in a large ***overhead*** in the system, and a considerably longer ***delay ,*** to provide an ***output*** to users. These factors must be carefully evaluated before approving the merge to the main ***branch .*** It is important that the process of accepting the merge of the ***stage*** branch to the main branch involves discussion with members of the other teams, considering the objectives and criteria of the current iteration of the development process, as well as the evaluation of ***stakeholders*** on the overall results of the system. Ideally, merges from the model ***pipeline*** to the main branch should be performed in sync with other teams — which may not happen in exceptional circumstances, such as updates that seek to make specific corrections or are of an emergency nature. However, if the change made generates a MAJOR modification (more details in Semantic Versioning) in relation to the current version of the ***main branch,*** the merge must be performed in sync with the modules of the other teams, in order to avoid inconsistencies in the production environment. Preferably, the two reviewers of the ***Pull Request*** should be professors, leaders, or senior professionals from the team. Conclusions about the general criteria defined by the team to evaluate merge requests to the ***main*** branch should be documented through technical comments in the Pull Request, by the reviewers.

Commit message management

Commits represent the completion and "submission" of a set of changes intended to accomplish a specific goal. The messages used to describe the type and purpose of a ***commit*** are critical to understanding the change history of ***software artifacts.*** Without clear, descriptive messages, it would be necessary to manually inspect each ***commit*** individually—a task that would quickly become unfeasible and would compromise effective management of changes made throughout the project.

Therefore, the standards detailed below aim to facilitate collaboration, understanding and change management. In order to, in addition to organizing the change history, promote clear and efficient communication between team members.

Format

Commit messages should follow this format: <type>: <subject>

Where

Type A word that describes the nature of the change (e.g. "feat", "fix", "docs", etc.).

Subject A brief description of what was done (e.g. "adds validation of file size"). ***dataset").***

Types

Commit message types must be one of the following:

feat A new feature added.

fix Changes that affect the build system or external dependencies, a bug fix.

in Addition, or alteration, of input artifacts (such as data, files, etc.), which serve as input to the system.

tune A change in hyperparameters in order to perform fine tuning on operation of model(s).

perf A code change that improves application performance.

test Add missing tests or fix existing tests.

refactor A code change that neither fixes a bug nor adds a feature, but makes the code cleaner, more efficient, or better documented.

docs Changes made only to the documentation

style Changes that do not affect the behavior of the code (whitespace, formatting, missing semicolons, etc.),

build Changes that affect the build system or external dependencies.

ci Changes to our CI configuration files and scripts (examples of scopes: Travis, Circle, BrowserStack, SauceLabs).

revert Revert a previous ***commit .***

conflict Conflict resolution.

Subject

The subject should be a clear and succinct description of what was done. The ***commit*** should begin with a verb in the present tense, and all letters should be lowercase.

Reference to Issues

A ***commit*** can reference or resolve an ***issue,*** through an optional ***footer*** at the end of the commit message , informing the number of the ***issue*** addressed.

Commit example

feat: add creation date extraction from mat file
References: #1221

Tags

Tags should be used to classify specific configurations (pipeline versions of the developed model). ***Tags*** should be used to categorize ***pipeline*** versions into stage and ***main branches .*** This way, ***releases*** can be easily explored, switched, and orchestrated — either manually ***or*** automatically — based on the information and categorizations defined by ***the tags.***

A practical example of using ***tags*** is the classification of ***pipeline*** releases ***based*** on the classifiers used in their development. For example, it would be possible to identify which ***pipeline*** versions used classifier ***X*** and which used classifier Y. This approach facilitates exploration, analysis, and deployment processes , allowing for more efficient and organized management of the different ***pipeline versions.***

Example Tags

classifier-X-v1.2.3: Identifies the classifier used and the pipeline version.
dataset-2023-10: Identifies the version of the dataset used for training.

Semantic versioning

Semantic versioning is a convention for assigning version numbers to software projects, with the aim of clearly communicating the changes introduced in each release. This standard is essential to ensure compatibility between different versions of a software and to facilitate dependency management.[2].

Basic Format

Semantic versioning follows the format MAJOR.MINOR.PATCH, where:

MAJOR: Incremented when there are incompatible changes to the public API.

MINOR: Increased when new features are added in a compatible way.

[2] Semantic Versioning <https://semver.org/lang/pt-BR/>

PATCH: Incremented when ***bug*** fixes are made that maintain compatibility.

Examples

1.0.0: First stable release.

1.1.0: Added new supported functionality.

1.1.1: Fixed a ***bug*** without changing the API.

Semantic Versioning Rules

MAJOR (Xyz)

It should be incremented when incompatible changes are introduced to the public API.
This may include removing functionality or making significant changes to existing behavior.
When the MAJOR version is incremented, the MINOR and PATCH versions should be reset to 0.

MINOR (xYz)

It should be incremented when new functionality is added in a way that is compatible with the existing API. ***Bug*** fixes may also be included in this version. The PATCH version should be reset to 0 when the MINOR version is incremented.

PATCH (xyZ)

It should only be incremented for ***bug*** fixes that do not affect the public API.
These fixes should be backwards compatible.

Pre-releases and Metadata

Pre-release versions can be identified by adding a hyphen and alphanumeric identifiers (e.g. 1.0.0-alpha). Build metadata can be added with a plus sign (e.g. 1.0.0+001).

Example If

a package depends on a library at version ^1.2.0, it can accept updates to any version 1.xy where x >= 2, but not to 2.0.0, which may contain incompatible changes.