Consistent, Autonomous, Organization-Wide Enforcement of Changing Software Quality Standards using DevOps

Johan Westin

johan.westin@aikospace.com

AIKO

INFINITE WAYS TO AUTONOMY

Abstract

Compliance with spaceflight software quality standards necessitates using various analysis tools and practices. However, ensuring these tools are used consistently across multiple projects requires extensive training and documentation. Introducing new tools requires developers to add them to each project manually. This costs valuable development time and risks inconsistency due to human error. We present an autonomous system which can mitigate these problems. It consists of a highly configurable project synchronization tool deployed in a three-tiered environment.

The core of the system is a synchronization tool which compares a software project to a centrally defined template. If the project does not adhere to the template, adherence issues are reported to the user and automatically fixed, if possible. A project template can specify any number of custom requirements, including specific analysis tools, the structure of DevOps pipelines, and a particular development environment layout. Templates are stored and maintained independently of the synchronization tool, allowing new requirements to be applied to all projects simultaneously, with minimal effort.

The synchronization tool should be used locally by developers, remotely in version control, and globally by administrators. Locally, developers use the tool to identify and resolve template adherence issues. In version control, it is run regularly in Continuous Integration pipelines to highlight issues as they appear. Finally, administrators use the tool to ensure it is being used correctly across projects, and to check that adherence issues are not ignored. This three-tiered system ensures that software quality is maintained organization-wide, even as new requirements are introduced. Streamlining this process reduces development costs, allowing developers to spend more time on significant problem-solving tasks.

The Project Synchronization Tool

The proposed system consists of a project synchronization tool (PST), a piece of software used in three different contexts (see *System Architecture*). The PST has a command line interface, allowing it to be used by developers, continuous integration (CI) pipelines, and by administrators. For illustration, we use `pst` as the program name in example commands. The PST has two core features accessible as subcommands: `pst check` and `pst fix`.

Check

The check feature takes an existing project (i.e., the working directory) as input, and reports to what extent it adheres to a project template. Templates are dynamically generated lists of requirements (see *Templates*), and `pst check` reports which of these are not satisfied by the current project. `pst check` can be seen as a highly configurable static analysis tool.

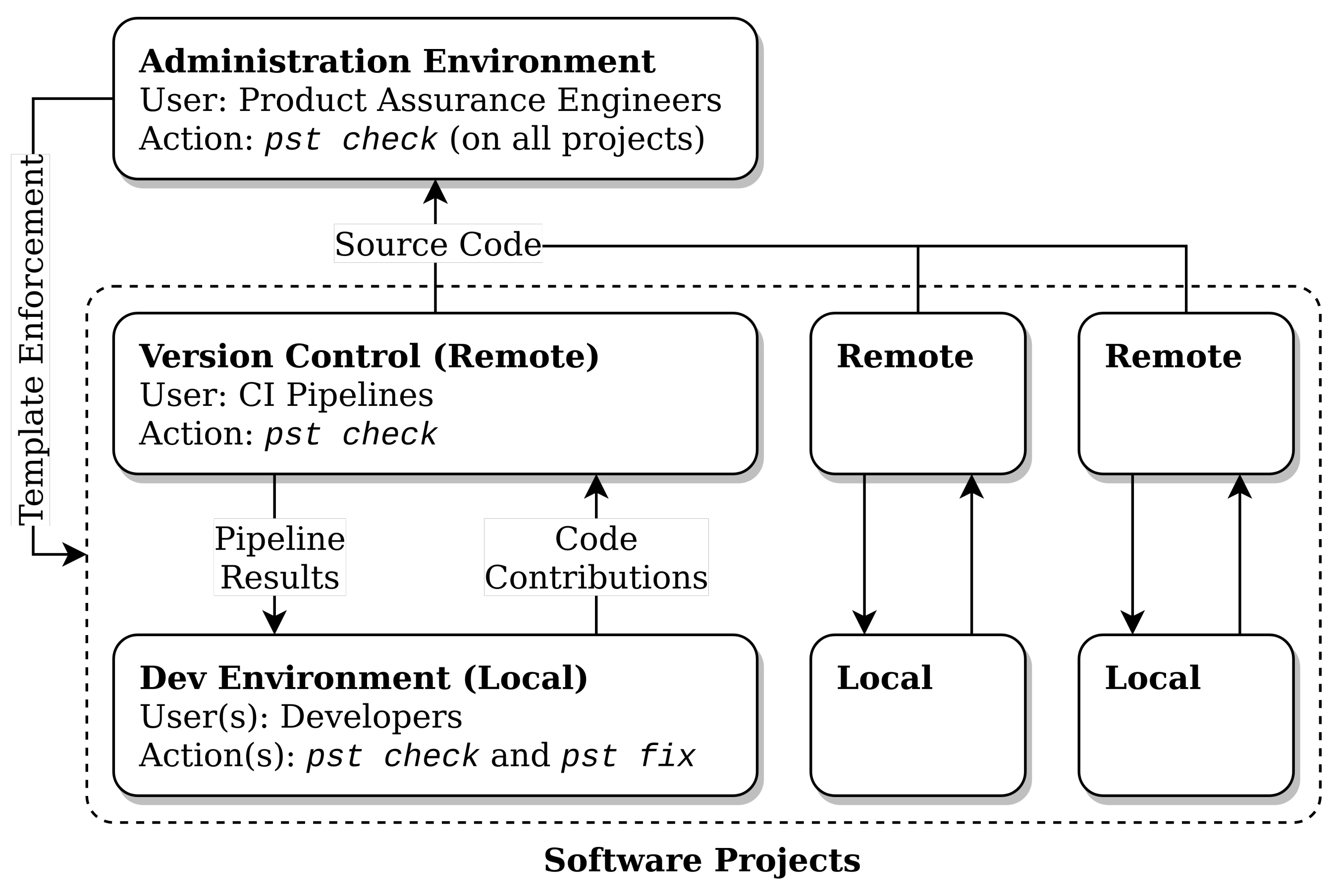
Fix

The fix feature is similar to `pst check`, but additionally attempts to automatically resolve any identified issues. `pst fix` is analogous to a formatter, though its scope is broader than most formatting tools. In fact, both `pst check` and `fix` may run other linters and/or formatters on the project, in addition to custom tools.

Some issues may be unfeasible to fix autonomously, and are instead reported and left for developers to fix manually. For example, if `pst` has never been run in the project before, the user is prompted to specify a project type, name etc. Whenever possible, `pst fix` will run iteratively until all issues are resolved.

System Architecture

To maximize its effectiveness, the project synchronization tool should be used in three different contexts: in the development environment, in version control, and in an administration environment. The following figure gives an overview of this architecture.



Developers use `pst check` locally to verify the quality of new contributions. When adherence issues are found, `pst fix` is used to apply and push prescribed fixes.

Continuous integration pipelines run `pst check` on the version control server. This provides developers and administrators with immediate feedback on template adherence issues, enabling rapid iteration. The CI pipeline also runs other automated actions like building, testing, documenting and releasing the software, allowing developers to focus their efforts on implementing new features.

Finally, the administration environment is used to ensure that the PST is used correctly in all projects. There are two main situations in which enforcement becomes necessary:

- The PST has not yet been set up in a project.
- Developers are not implementing PST fixes when necessary.

To mitigate these two situations, an administrative team (which includes product assurance engineers) periodically runs `pst check` on every software project. This ensures that the PST is integrated into every project, and that neglected fixes are implemented when needed.

When used correctly, the PST can automatically manage a project's configuration within the local and remote contexts. This means administrative enforcement is only necessary in rare cases, allowing many projects to be managed by a relatively small admin team.

Templates

Templates are used by `pst check` and `pst fix` to generate a sequence of requirements. These are generated dynamically based on the project state (including project parameters). It contains a hierarchy of requirements, with e.g. the existence of certain files and parameters being "root" requirements, while things like file contents, formatting and parameter values are derived from these.

Requirements are used by `pst check` to identify issues that need to be fixed. Each requirement has an associated *fix action*, which is what `pst fix` will do if the requirement is not met.

For the PST system concept to function effectively, all templates should require that a CI pipeline is defined for the project. The following table shows a simplified example of what template requirements and fix actions for creating a CI pipeline could look like. This example uses *Gitlab* as the CI framework and *Pylint* as an example third-party linting tool, though any other collection of tools can be used for similar purposes.

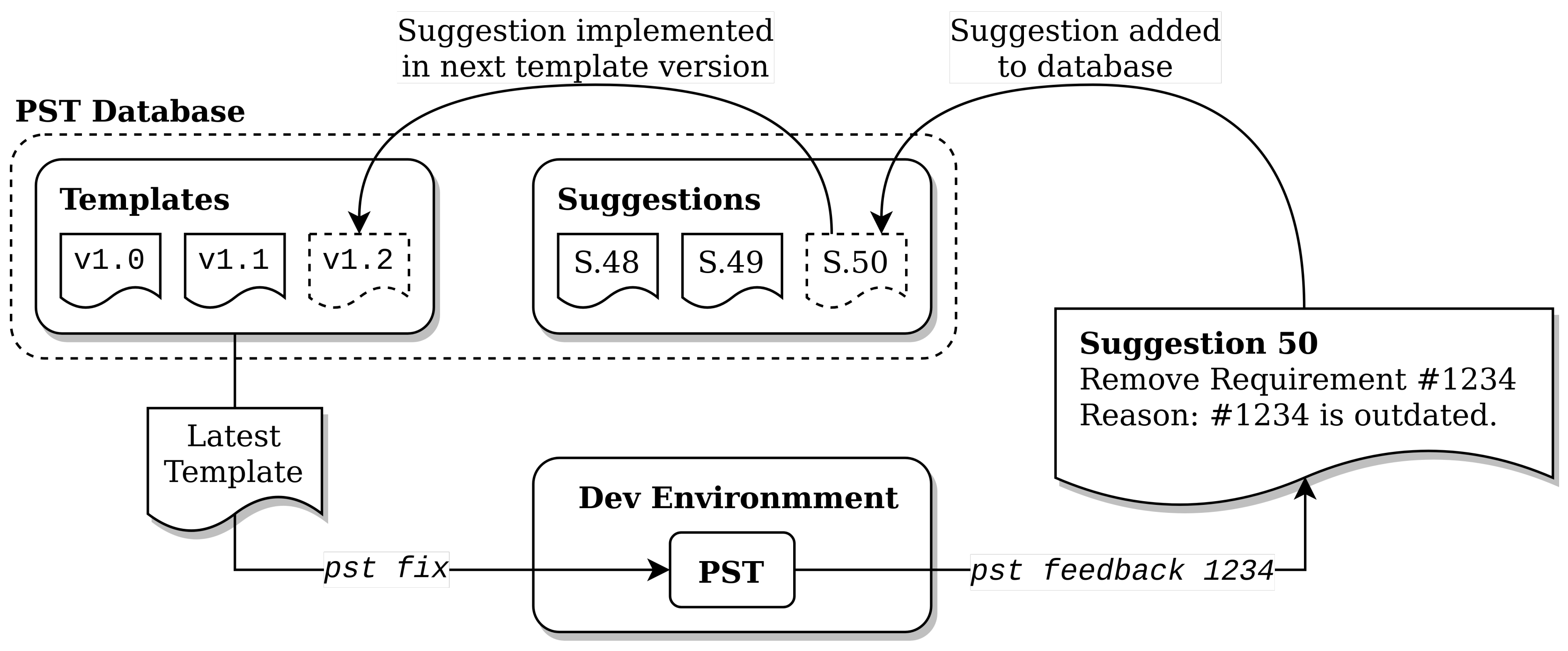
Requirement	Fix Action
The project shall contain a `.gitlab-ci.yml` file.	Create an empty `.gitlab-ci.yml` file.
`.gitlab-ci.yml` shall contain a job template.	Create a `.job: ...` section in the file.
The job template shall not be allowed to fail.	Set `allow_failure: false` in `.job`.
The job template shall be interruptible.	Set `interruptible: true` in `.job`.
…	
`.gitlab-ci.yml` shall contain a *Pylint* job.	Create a `pylint: ...` section.
`pylint` shall inherit from `.job`.	Set `extends: .job` in `pylint`.
`pylint` shall run *Pylint* on all Python files.	Set `script: tools/run-pylint.sh`.
`tools/run-pylint.sh` shall exist.	Create `run-pylint.sh` from a template.
…	
`.gitlab-ci.yml` shall contain a project check job.	Create a `project-check: ...` section.
`project-check` shall run `pst check`.	Set `script: pst check`
…	
`.gitlab-ci.yml` shall contain a build job	Create a `build-library: ...` section.
…	
`.gitlab-ci.yml` shall contain a unit testing job	Create a `tests: ...` section.
`tests` shall use the built library.	Set `dependencies: [build-library]`.
…	
`.gitlab-ci.yml` shall contain a deployment job.	Create a `deploy: ...` section.
`deploy` shall only run if all other jobs pass	Set `when: on_success`.
…	

This granular approach is advantageous in terms of code re-use and abstraction. Templates can be specified using data formats like JSON or Yaml. Requirement types are specified by name (e.g. `"file_shall_exist"`) and correspond with a specific behavior (e.g. creating the specified file). Furthermore, groups of requirements and actions can be collected together and re-used to construct large templates with a relatively short definition.

Template Maintenance

The set of requirements and fix actions prescribed by a template will differ between project types and organizations. Product assurance engineers are tasked with ensuring that templates correspond with best practices, and that they accurately reflect the needs and standards of the organization. A user feedback mechanism is used to gather and implement ideas for template changes.

In addition to a fix action, each template requirement has an associated *description* and *ID*. Descriptions summarize what the requirement does and why it is part of the template. The description and ID are automatically shown when a requirement fails during `pst check` or `fix`. This allows the user to send feedback (using `pst feedback <ID>`) suggesting changes to the template. The product assurance team may then incorporate these suggestions into future template versions.



This approach to template maintenance is used to streamline the integration of new tools and techniques. Developers and product assurance engineers can independently research and test new tools for inclusion in future PST templates.

By encouraging regular feedback on templates, developers can globally automate repetitive tasks. If a developer notices that they often perform the same action e.g. during project setup, a requirement which automates the action for every project eliminates this time investment for the entire team.

Finally, this maintenance approach allows the PST software and the templates it uses to be kept separate. Since templates can be defined purely as data, updates to the core PST software is only necessary when a new type of fix action is needed.

Summary

The system presented here consists of a project synchronization tool (PST) used in three context by developers, CI pipelines and product assurance engineers. The PST automatically checks and fixes projects against a dynamically generated template. It is designed to be run frequently in many different contexts to provide continuous feedback and quality improvements.

The aim of the PST system is to improve the efficiency of product assurance, configuration management and development. Autonomous fixes are implemented whenever possible, and the effort of applying these fixes is distributed between team members. This reduces the time spent on repetitive tasks, improves overall project quality and helps keep an entire organization closer to the cutting edge.