A Catalog of Continuous Integration Bad Practices

Introduction

The catalog is made up of 79 CI smells properly distributed among 7 different categories related to specific aspects of a CI pipeline management.

For each category, we report:

- A general description of the *Problem*.
- The main *Consequences* of the problem, which are in most cases negatives, but in some cases they can also be positives, therefore requiring a trade-off in the choices to be made (*e.g.,* a balanced set of branches requires to choose between having too few branches and having too many branches).
- The associated *CI Smells* included in it that represent concrete instance problems and/or symptoms of CI misuses.

Bad smells indicated with (∗) have been elicited from interviews only. Bad smells indicated with (∗∗) have been elicited from Stack Overflow only. All others have been elicited from both sources.

Catalog

1 Repository

This category covers smells related to (i) the repository structure, (ii) the branch management, and (iii) the configuration items. More specifically, this first category deals with a sub-optimal organization of projects resources and artifacts as well as poor project decomposition, and the use of Version Control System (VCS) in terms of lacking a proper management of features development branches and missing to properly put all needed resources under version control. All these issues might hinder the CI pipeline efficiency and effectiveness. More in details, it is possible to unnecessarily increase the build duration (*i.e.,* unnecessary rebuild sub-modules not affected by a change), and to have an excessive number of conflicts and build failures as well as going far from the purpose of using a CI pipeline (*i.e.,* deferring the integration later in the overall process).

1.1 Repository Structure

- *The project decomposition in the repository does not follow modularization principles.* Modularization is a key factor in the CI adoption. Using sub-modules can help reducing the overall build duration (*i.e.,* avoiding unnecessary re-builds) and increasing fault-detection effectiveness.
- *Test cases are not organized in folders based on their purposes.*** It can be a good practice to have a folder for each different kind of test to execute within the CI pipeline in order to reduce the manual effort required in order to define/add new test cases. Moreover, if the project misses a clear separation between different type of testing activities, it will unavoidably increase the overall build duration as the size of the production and test code increase since that the CI pipeline will run all of them at each build.
- *Local and remote workspace are misaligned.* In order to avoid a scenario like: "But it works on my machine", it is required to have a local environment that is always aligned with the one used on the CI server in terms of set up (*e.g.,* plugin versions) and folder structure.

1.2 Branch Management

- *Number of branches do not fit the project needs/characteristics.* This CI smell highlights a trade-off. On the one hand, having a huge number of branches will increase the number of conflicts during merging activities due to the deferred integration. On the other hand, instead, increasing the number of branches could be useful in order to have a clear separation of concepts (*i.e.,* having a branch for each different concept).
- *A stable release branch is missing.*** A stable release branch can guarantee the presence of a releasable version (on which developers have executed different kind of tests like regression and acceptance, increasing the overall confidence level on the system) of the system/application.

- *Feature branches are used instead of feature toggles.* Feature toggle allows hiding/revealing features within the same branch, thus facilitating incremental release. The CI smell highlights a trade-off. On the one hand, feature toggles safeguard CI even if developers could have the risk that unfinished features affect production negatively. On the other hand, instead, feature branches help developers dealing with uncompleted features while introducing much effort required for solving conflicts/merges due to the deferred integration.
- *Divergent branches.* This CI smell deals with the presence of aged or unsynched branches with respect to the master branch. More specifically, if some changes are not merged into the release branch, it is possible to have an unstable version of the system/application, which will increase the complexity related to the maintenance of different versions of the system.

1.3 Configuration Items

- *Generated artifacts are versioned, while they should not.* The usage of an ad-hoc Artifact Repository Manager (*e.g.,* Artifactory) is the more suitable solution for tracking artifacts from development to production, helping into guarantee data integrity and simplicity to find any kind of needed artifact. More specifically, on the one hand, using the same versioning system as the one holding the production code can help developers optimize space and bandwidth. On the other hand, instead, this practice might result in an improper integration or deployment (*e.g.,* previously produced artifacts will be used for deployment into production).
- *Blobs (binary large objects, e.g., used to store some data) are unnecessarily checked-in at every build instead of being cached.* Check-in the same huge repository at each build will unavoidably slow down the overall build duration, and, at the same time, could introduce unnecessary build failures and errors in the validation phase.
- *Pipeline related resources (e.g., configuration files, build scripts, test data) are not versioned.*** All resources related to the build process must be put under version control as the production code in order to ensure repeatability, changes' tracking (*e.g.,* changes in the database schema) and avoiding accidental modifications and deletions.

2 Infrastructure Choices

This category covers smells related to (i) hardware configuration and (ii) software configuration. More specifically, this category considers the wrong distribution of hardware resources among different projects (*i.e.,* wrong allocation of machines over different tasks), as well as the poor software choices related to the definition of the overall CI pipeline, *e.g.,* the choice of build script plugins with their configuration. The former can negatively impact the overall performance of the CI pipeline in terms of lacking scalability increasing the overall build duration. On the other side, the CI smells dealing with a sub-optimal tool choice along with its configuration may introduce unnecessary build failures, and most important, miss to highlight possible relevant problems. Moreover, all of them negatively impact on the overall build process maintainability.

2.1 Hardware

- *Resources related to the same pipeline stage are distributed over several servers.*** Build jobs that are connected in a chain (*i.e.,* each job performing a specific task in the pipeline) should be run on the same server in order to speed up the build.
- *The CI server hardware is used for different purposes other than running the CI framework.*** It is important to have a dedicated CI server to use in order to run the build process ensuring repeatability, as well as avoiding to overload the developer machine and supplying features for connecting different build jobs.

2.2 Software

- *External tools are used with their default configurations.*** Each tool has to be configured according to (i) the developers' needs, (ii) the final product requirements, and (iii) the policies adopted by the organization. More specifically, each tool should not be used as is in order to avoid its misuse, and most important to avoid that its outcomes are ignored in presence of alerts. As an example, consider the case in which the CI pipeline relies on the default configuration when executing static code analysis tools. In this scenario it may happen that some irrelevant checks are enabled generating useless warnings and unnecessary build failures, and some relevant checks are missed loosing awareness on the overall software quality.
- *Different releases of tools/plugins versions are installed on the same server, causing inconsistencies in the build status.*** Installing different versions of the same tool on the CI server might lead to inconsistencies resulting in lack of repeatability of the overall build process.
- *Different plugins are used to perform the same task in the same build process.*** As an example consider the case in which the CI pipeline use multiple static code analysis tools to perform similar checks. It should be avoided to install different tools in order to perform the same task in order

to not unnecessarily slow down the overall build duration.

- *A task is implemented using an unsuitable tool/plugin.* As an example consider the case in which the CI pipeline uses "surefire" plugin for executing integration testing, while it is well-suited for executing unit testing activities. More specifically, very often it is possible to choose between different available plugins in order to execute the same build task. In the latter case you have to pay attention on the one being selected. Indeed, very often the common mistake is the one related to the re-use of a tool in the pipeline instead of using an adequate one. Another mistake is related to the inclusion of a not-well maintained plugins.
- *Use shell scripts for a task for which there is a suitable plugin available.* When available, it is better to use a proper plugin instead of relying on a combination of different shell scripts. However, for some tasks that have to be performed once, it is not always needed to rely on an external tool, but it is better to rely on fast and easy shell scripts. More specifically, the overblown usage of custom shell scripts to implement the CI pipeline makes it less maintainable and less portable (*i.e.,* some scripts are OS-dependent). On the contrary, there are contexts in which it is more useful to rely on shell scripts. As an example, when a reliable tool for performing a specific task is not available or when projects built with different build tools have to be connected.

3 Build Process Organization

This categories includes the CI smells related with the organization of the overall build process. First of all, there are some CI smells dealing with sub-optimal cleaning up strategies adopted on the CI server. In this context we look at whether all the files generated at build time will be deleted and re-created at each build or whether it is possible to re-use some of them (*i.e.,* incremental build). Moreover, we look at whether the build process checks for verifying the availability of all resources (*e.g.,* tools and libraries) needed to perform a build.

We have another category of CI smells dealing with the build process definition: (i) lack in the definition of cohesive build jobs, defeating the "Fast Feedback" practice, and (ii) build the entire project instead of building each sub-project separately.

Another class of smells is related to problems impacting the build execution. On the one hand, it is possible that the build process does not parallelize builds' execution leading to an increase of the overall build execution time. On the other hand, it is possible to define an incorrect order of build steps, *e.g.,* deploying without a fully integration phase, leading to a CI pipeline that is unable to spot bugs following the "Fast Feedback" principle.

From a different perspective, we have a different category of CI smells dealing with the adoption of poor (*i.e.,* non-fitting with the organization) build triggering policies, *e.g.,* the way the build process is enacted. As a consequence, it may be possible to introduce unnecessary build failures due to issues not handled in private builds.

In terms of build outcome, it may be possible to use wrong policies while assigning the status of a completed build. As an example, let a build succeed even when there are some failed tests or when there are some errors reported in the build log. In this way, developers cannot rely on the build result. Indeed, as an example, a build success may hide a test failure or an interrupted deploy.

As regards the build output, instead, it is necessary to guarantee that is observable and readable. Specifically, if the build reports generated do not have a clear structure, developers might not be able to analyze them, increasing the overall time needed to identify the root cause of a failure. Moreover, when the failures/errors occurred while executing the pipeline are not notified, developers are not aware about the issue.

In defining the overall build process, it is required to adopt a strategy dealing with dependency management. Using a poor strategy may result in a subsequent number of useless or unnecessary build failures. As an example, if you have to build a project that depends on an already built project, you should only include it instead of building it again. In this case, it is possible to avoid the re-build of dependent projects several time, reducing the overall build duration.

Another important aspect to consider when defining the CI pipeline, is related to the build duration (*i.e.,* overall build execution time) mainly impacted by the processed resources being used and time-consuming tasks being executed. Having builds that take too long time increases the feedback time (*i.e.,* the time required to notify the build output to developers), may introduce waste

of resources and, may slow down the developer productivity.

Finally, since the CI pipeline usually run on a specific CI server it is necessary to deal with security issues. More specifically, a common mistake regards the clearly hard-coding of authentication data under version control system.

3.1 Build Initialization

- *Inappropriate build environment clean-up strategy (i.e., whether to (not) use incremental builds).* This CI smell is a trade-off since that, based on the organizations need, and on the type of build to execute, it is possible to use or not incremental builds. On the one hand, the result of the build may be affected by files deleted from the version control system but still present in the project workspace. On the other hand, the CI server does not checkout all the source files and recompile all of them at each build, decreasing the overall build execution time. A particular type of source file is test code. In this case, if the test data is not cleaned up from the previous builds it is possible that tests could pass or fail because of the previous state of the build.
- *Missing package management (e.g., a maven plugin to validate the produced archives).* Without using a package manager, it is hard to know about the availability of the right resources in a proper state (*e.g.,* tools upgraded to the last version) before starting a build, and a package manager might fail the build due to third-party dependencies upgrade or it can slow down the build in case of very huge updates. Specifically, if the CI pipeline does not include a package manager, developers might wrongly assume the presence of resources no longer available or use an outdated version of them. Furthermore, it becomes hard to manage tools included in the pipeline, in terms of installing, upgrading, configuring, and removing them. By means of package management it is possible to specify when to check for updates avoiding to download all dependencies at each build.

3.2 Build Process Definition

- *Wide and incohesive build jobs are used (e.g., a build job mixes up testing and static analysis activities).* This smell deals with the inclusion of several build steps (*e.g.,* compilation, testing, quality check) in the same build job. Using this practice, it becomes hard to isolate the build step where the failure occurs.
- *Monolithic builds (i.e., builds not decomposed into specific, cohesive jobs, when this is appropriate) are used in the pipeline.* If a change affects only one of the sub-projects, it would not be necessary to build the other unchanged sub-projects. However, adopting monolithic build might increase the overall build execution time compared to keep each sub-project build separate. In other words, using separate configuration files for sub-projects belonging to the same root project (and therefore treating them as independent project) and always build them together can increase their

management effort. It might be helpful to at least structure the configuration files as a hierarchy making their build repeatable.

3.3 Build Execution

- *Independent build jobs are not executed in parallel.*** When build jobs are independent, they can be launched at the same time. Furthermore, the build jobs related to the same project should be structured as independent jobs (when possible) in order to decrease the overall build duration of the project. In this way it is possible to fully use the available hardware for decreasing the build duration.
- *Only the last commit is built, aborting obsolete and queued builds.* Developers should always build the last committed changes. Aborting the previous builds still in progress, triggered by previous (*i.e.,* outdated) changes may lead to an unstable state. Moreover, in presence of a failure it becomes complex to isolate the change responsible of the error thrown.
- *Build steps are not properly ordered (e.g., static analysis could precede testing to avoid failing it very late).* Some build steps should be always performed before others like integration testing should be scheduled before deployment in the production environment. Only doing this, it is possible to spot bugs as soon as they are introduced.
- *Private builds are not used.* It is a good practice to perform a "private build" before checking the change in the pipeline because it is easier to spot and locate the bugs locally avoiding unnecessary build failures.
- *Pipeline steps/stages are skipped arbitrarily.* Developers should not skip pipeline steps since this practice makes untrustworthy the released version. For instance, it is a bad practice to skip some tests in order to avoid failures in the build process but also to speed up the build in the commit stage. Adopting this approach, developers will only delay the discovery of potential problems in the development code. Furthermore, when you have build failures due to warnings/errors raised by static analysis tools, instead of focusing on the problem resolution you could simply skip their execution in order to make the build passed.
- *Tasks are not properly distributed among different build stages (e.g., load testing is done in the commit stage).* Developers must properly define the tasks to execute in each stage of the CI pipeline and, at the same time, provide different stages. As an example, running load testing in the commit stage could slow down the overall build process uselessly.

3.4 Build Triggering

- *Incremental builds are used while never building the whole project from scratch.* This CI smell can be seen as a trade-off. Indeed, on the one hand, developers can adopt incremental builds (*i.e.,* checkout and build only what is changed), speeding up the build but possibly being affected by source control update issues and by the usual issues due to starting

a build in a non-cleaned environment. On the other hand, developers can always perform a full checkout and rebuild, that is much slower and requires a higher load on the CI server but is definitely more robust. A possible trade-off is to perform an incremental checkout (*i.e.,* only pull changes from source control) followed by a full rebuild, making the process somewhat more robust.

- *Poor build triggering strategy (e.g., build each single commit or, on the contrary, build less than once a day).* As the previous CI smell, also in this case the antipattern highlights a trade-off. Indeed, developers might launch the build at every push or perform one build per day, only including modules that have been changed. At the same time, developers may decide to filter commits that have to trigger a build (*e.g.,* release commits).
- *Some pipeline's tasks are started manually.* It is always suggested to avoid to include manual steps in the CI process but in special cases (*e.g.,* release on a production environment) the organization needs to start a build manually. Missing full automation will make the overall build process non-repeatable.
- *The project only uses nightly builds when having multiple builds per day is feasible.*** The use of nightly builds is highlighted as a trade-off. The main principle is that the build should be relatively fast. To avoid slow builds, long time-consuming tasks (*e.g.,* the full test suite) should be run out of the working hours (*i.e.,* during the night). Instead, short tasks (*e.g.,* a smaller test suite) could be part of regular builds. Furthermore, if developers schedule all kind of tasks (*e.g.,* the entire test suite) during regular builds, nightly builds would be redundant. If developers think that scheduling time-consuming tasks during regular builds make them not able to complete in a reasonable amount of time, they should consider running those tasks over the night. If you not rely on nightly builds it is possible to slow down the build execution time, while relying only on their usage may compromise the early discovery of issues/problems.
- *Inactive projects are being polled.** Developers should not forget to remove the obsolete (*i.e.,* no longer maintained) projects among the projects to which the CI server pools for changes because it leads to wasting resources.

3.5 Build Outcome

- *A build is succeeded when a task is failed or an error is thrown.*** When a build step wraps in a failure, the entire build must end in a failed status. Assigning a wrong status to a completed build implies that developers cannot rely on the overall build result.

3.6 Build Dependencies Management

- *Dependency management is not used (e.g., the same project is rebuilt more than once).* When developers have to deal with several projects developed in parallel, it is crucial to define the right dependencies among these

projects. As an example, developers might need to build a project A that requires the build of another project B from whom A depends on. Sometimes in case of "circular builds" the build of one project in the "cycle" leads to several (and unnecessary) rebuilds of another project.
- *Including unneeded dependencies.*** It is important to delete dependencies to projects no longer maintained or not needed in order to speed up the whole build process.

3.7 Build Output

- *Some tasks are executed without clearly reporting their results in the build output.*** Make sure that the result of each task is clearly reported in the build report. Moreover, some issues causing a build failure are not shown in the log file. For example, in case of test failures more detailed information about the reason for a failed test case is stored in a different file. In the latter case, developers might not be aware about the issue.
- *The output of different build tasks are mixed in the build output.*** The build reports mix the results of different build activities decreasing their readability.
- *Failures notifications are only sent to teams/developers that explicitly subscribed.*** On the one hand, everyone needs to be notified, otherwise they might not be aware of critical issues they could help to solve. On the other hand, sending notification in broadcast might annoy and decrease the productivity of those developers that do not have the right expertise.
- *Missing a build notification mechanism (e.g., email to interested developers).*** The notification system should be fast and should provide a good overview about what it is going on in the pipeline.
- *Build reports contain verbose, irrelevant information.*** Some build reports are too verbose even if they have to never provide truncated information.

3.8 Build Duration

- *Time-out is not properly configured.*** Specifically, timeout is not set considering a realistic time to complete certain tasks, *e.g.,* timeout too short for an expensive load testing. Developers should choose a strategy for setting the time-out, especially for testing activities. More specifically, they can either select an absolute time (if developers know exactly, in advance, for how much time the build should not exceed) or an elastic time based on the average execution of the last n builds.
- *Unneeded tasks are scheduled in the build process.*** Some tasks such as testing or code analysis could be sped up avoiding to perform them on resources that are not important for the specific task (*e.g.,* static analysis on generated classes).
- *Build time for the "commit stage" overcomes the 10-minutes rule.* The build should last maximum 10-minutes at the commit stage for not de-

crease the developer productivity but also to ensure fast feedbacks.

- *Unnecessary re-build steps are performed.*** An incorrect tasks partition among the build jobs might cause unnecessary re-execution of some build steps. A typical scenario is trying to separate compilation and unit testing. Due to building tool constraints, splitting those two steps makes a new compilation needed when trying to run unit tests.

3.9 Security

- *Authentication data is hardcoded (in clear) under VCS.* It has to be avoided to hard-code credentials. The use of environment variables could mitigate without solving the problem because they might need a restart of the server to be working. The use of shared and protected files might be the solution. The same happen for the build configuration parameters that have to be stored in a separate file. In this way, the same configuration might be run in different environments relying on environment variables when some modifications are needed.

4 Build Maintainability

This category of CI smells concerns the maintainability of build scripts used. More specifically it accounts for (i) build scripts highly tightened to a specific environment and (ii) presence of duplicate configurations. As a consequence it is possible to have an excessive number of build failures, as well as, a limited build maintainability and portability.

- *Absolute/machine-dependent paths are used.* Hardcoding paths is always a bad habit and referring to local paths make the build working on local machine, not on remote CI server. It might be a solution to use PROPERTIES FILES, BUILD PARAMETERS and paths relative to a CI ROOT.
- *Build scripts are highly dependent upon the IDE.*** The build environment should not be tied to the local workspace. It causes errors like "employ local accessible resources while building on the CI server".
- *Environment variables are not used at all.* Empowering the proper set of environment variables makes the build easily portable on a different machine.
- *Build configurations are cloned in different environments.*** When developers need to run the same project in different environments, it can be re-used one single build job with proper parameters depending on the different environments. In other words, when several builds share the same configuration, it should be avoided to clone it and put it in several build jobs. Moreover, it is better to follow the principle of "one point configuration", where configuring all common configurations *e.g.,* by means of environment variables.
- *Build jobs are not parametrized.* In order to avoid duplication in the configuration files, it is better to use parametrized jobs. Moreover, using them it is possible to solve problems related to the manual changes onto configuration files in order to start a new build (*i.e.,* the build is not replicable). Of course, in order to obtain advantages from this solution it is important that you have properly configured and set the environment variables.
- *Lengthy build scripts.*** It makes the entire build process difficult to understand and maintain. Each build script should include one specific task, increasing its re-usability.
- *Missing smoke test (tests to verify the testability of the build).* The entire pipeline needs to be tested to be sure that it is able to release new software according to the established work-flow. Recurring and trivial issues (*e.g.,* incorrectly include files or missing resources) can be easily spotted adopting smoke tests (*e.g.,* unit test that fails because basic files are missing).
- *Missing/Poor strict naming convention for build jobs.* It is important to define clear naming conventions for the build jobs. As a result, it becomes easy to prevent duplicate jobs (*e.g.,* same purpose and same project).

5 Quality Assurance

This category includes CI smells related to (i) testing and (ii) code analysis phases. More specifically, we have issues related to the usage of poor practices while creating, organizing, and executing test suites for different kinds of testing activities (*e.g.,* unit, integration, performance, acceptance testing). Moreover, we also have issues related to a poor definition of checks performed by static analysis tools, as well as of builds' quality gates.

First of all, among the CI smells dealing with testing practices we have some of them related to a bad environment representativeness (*i.e.,* miss to test the whole system in a production-like environment using properly configured simulated environments). We also have some issues accounting for the absence of a clear separation and balance between different kinds of testing activities. As regards the test execution we have issues dealing with an improper settings of code coverage thresholds that make the build fail, as well as the inadequate test suites used within the CI pipeline (*e.g.,* individual branches do not have their specific test suites or a test suite effectiveness/coverage degrades as the system evolves and the tests are not updated). In terms of poor practices while creating test cases the catalog accounts for (i) inappropriate storage of test data, (ii) the presence of non-automated tests, and (iii) the presence of flaky tests. Finally, there are also CI smells related to the test execution strategy, *i.e.,* retesting everything, running specific tests only on the CI server or arbitrarily skipping tests.

In terms of code analysis, instead we have included CI smells related to a poor definition of quality gates (*i.e.,* criteria to make a build fail) used by the organization, as well as bad skip practices adopted such as running static code analysis tools only on changed code without consider the whole system/application. The above CI smells leads to a poor fault-detection effectiveness since developers cannot completely trust the results of static analyzers.

5.1 Testing

- *Lack of testing in a production-like environment.* If developers miss testing in a production-like environment, it will delay the discovery of integration bugs which might lead to seriously damage in production environment, thus, requiring a recovery action. In the worst case, the integration bugs will be identified only when the software is released in production. In other words, this CI smell is summarized in: "But it works on my machine".
- *Code coverage tools are run only while performing testing different from unit and integration (e.g., performance testing).*** The code coverage tools must be executed in conjunction with low-level tests. Indeed, if a developer runs a performance test to evaluate the time needed in order to complete a functionality, it is supposed that a great amount of the code will be executed resulting in an incorrect coverage level.
- *Coverage thresholds are fixed on what reached in previous builds.*** If developers do not define a code coverage threshold, assuming that the test

coverage will never be worst than the one obtained in the previous build, they might misunderstand the build results, loosing the control or visibility on the amount of both production and test code added and/or deleted.

- *Coverage thresholds are too high.* Increasing too much the code coverage threshold will raise the number of subsequent build failures decreasing the developer productivity since developers will spend time on fix the un-needed failures.
- *Missing tests on feature branches.* When developers have several branches they must run at least unit and integration tests on each of them before to merge each branch in the latest stage of the CI pipeline (*i.e.,* the release stage). In this way, developers will not demand the discovery of bugs at the latest stage of the pipeline.
- *All permutations of feature toggles are tested.*** Feature toggles allow showing/hiding features in a code branch, facilitating incremental development. Testing all permutations might become too expensive or even produces unnecessary failures. More specifically, feature toggles are used as a means to have a stable release even if developers have half-implemented feature on the releases. Try to test all combinations of those feature toggles will lead to a combinatorial explosion of the test cases, increasing the overall build duration. It is a good practice to run only the combinations of those toggles that will appear in the next release.
- *Production resources are used for testing purposes.*** The best practice is to use mock or fake database data in unit tests and reserve the real DB only for integration tests in the latest stage of the CI pipeline.
- *Testing is not fully automated leading to a non-reproducible build.*** Consider as an example those cases in which input test data is supplied by the developer during the build process. Moreover, mixing automated and manual testing will allow the testes to access the server staged for automated tests. The latter could produce false negatives tests' results both in the automated and the manual tests. Furthermore, manual tests will make the build unique (non-repeatable).
- *Test suite contains flaky tests (test cases having a non-deterministic behaviour).*** A common principle is that "tests are supposed to just work". The presence of flaky tests may lead to the belief that the test suite is no longer useful for catching bugs and developers could start to waste time investigating problems that do not actually exist.
- *Bad choice on the subset of test cases to run on the CI server.*** Developers can run the whole test suite inevitably increasing the build duration or, in the commit stage, run only those test cases impacting the files in the last change set. The latter will speed-up the build process but could delay the discovery of regression bugs.
- *Failed tests are re-executed in the same build.*** Make sure that each test case is executed only once for each build in order to avoid the generation of incorrect reports but mainly avoid the randomness of the results.

5.2 Quality checks/gates

- *Quality gates are defined without developers considering only what dictated by the customer.** The developer is the "hearth" in the definition of the quality gates since she is not the only one that has a full and clear knowledge about the code. The developer should be involved in the definition of the quality gates for the product she is developing. Indeed, customers have not enough knowledge in software development so even if it is important that the product will satisfy their needs also in terms of quality, developers cannot only rely on those rules for verifying the overall quality of the code (they can delay too much the discovery of bugs/problems in the system that are simple to monitor by means of appropriate checks).
- *Use quality gates (e.g., lines of code checked-in) in order to monitor the activity of specific developers without using them for measuring the overall software quality.* If quality gates are defined in order to monitor the activity of each developers instead of focusing on the global quality of the system/application, it is possible that the overall team spirit will be negatively impacted.
- *Unnecessary static analysis checks are included in the build process (e.g., checks for databases or for mobile applications when they do not pertain to the application domain).*** The inclusion of unnecessary and/or duplicated checks will slow down the overall build duration but also will decrease developers' productivity since they will waste their time trying to fix violated checks that do not completely fit the need of the organization.

6 Delivery Process

This category concerns the storage of artifacts related to a project's repository as well as to wrong or suboptimal choices in executing and setting deployment tasks. Moreover, the same category also covers problems due to a missing tagging policy for artifacts related to a specific release.

When this category of problems is present in a CI pipeline (i) it becomes unclear the circumstances under which the project can be shipped, (ii) it limits the pipeline roll-back capability and complexity, (iii) it leads to unnecessary rebuild, and (iv) developers might not be aware of some missing files that are expected to be in the package resulted from the build.

- *Artifacts locally generated (i.e., on the developers' machine rather than on the CI machine) are deployed.* Never deploy artifacts that are not built by through the CI server. They are not reliable because the build server in the pipeline has to be the single point of truth (*i.e.,* local machine might produce wrong artifacts which should not be deployed).
- *Missing artifacts' repository (e.g., platform like Artifactory to handle releases).* It is a good practice to schedule an intermediate step between "Build" and "Deploy" to actually store the built artifacts that have to be deployed. It allows to avoid building again projects as soon as developers need to deploy them several times, thus it decouples the two stages.
- *Missing roll-back strategy (for artifacts deployed on an Artifact repository).* A good roll-back strategy should concern not only the versioned code (*e.g.,* the source code in VCS) but also the other resources involved in the releasing process (*e.g.,* database used during testing, build configuration files as static analysis rules). It makes possible to completely restore a previous state (*i.e.,* version) of the release.
- *Release tag strategy is missing.* Developers should have a clear tag strategy. It might be useful to tag each new version released in production to facilitate the roll-back in case of unsuccessful deploy or critical bugs.
- *Missing check for deliverables (e.g., check whether all files are in the delivery).* It is important to check for the presence of all awaited files after each build.

7 Culture

This category includes CI smells related to human factors in the adoption of the CI pipeline. More specifically, it accounts for (i) the lack of proper guidelines to follow pushing/merging into the main branch, (ii) keeping developers and operators roles separate — *i.e.,* do not adopt the DevOps model — and (iii) lack of a prioritization between the development of new features and fixing failed builds.

Having this category of CI smells within the pipeline will increase the communication overhead, will corrupt the local environment pulling changes from an unstable main branch and will increase the effort needed to fix a build failure. Furthermore, on one side, giving low priority on fixing build failures will not allow to always have a stable version of the system/application. On the other side, instead, focusing too much on addressing build failures will unavoidably decrease the developer productivity.

- *Changes are pulled before fixing a previous build failure.** When developers pull changes from the main branch they must take care of the previous build status since in presence of a failure it will corrupt the local workspace as well.
- *Team meeting/discussion is performed just before pushing on the master branch.** It there is a lack of a proper definition of a branching strategy, it will require a meeting, between different development teams, each time is needed to push changes on the main branch, This will unavoidably increase the communication overhead.
- *Developers and Operators are kept as separate roles.** CI is based on the adoption of DevOps so it is unacceptable to have developers separated from operators.
- *Developers do not have a complete control of the environment.** Relying on environments handled by different organizations is a clear symptom of not adoption of the DevOps model. The last is a key element for a right adoption of the CI pipeline.
- *Build failures are not fixed immediately giving priority to other changes.** Build failures must be considered equally important and in presence of a failure, developers must guarantee that the whole effort will be spent on its resolution.
- *Issue notifications are ignored.** The main purpose of the feedback mechanism is to quickly alert developers in presence of build failures so, in this case, their priority must become the one associate to the fix of the issues raised.

Table 1: Mapping between Duvall's patterns (and their antipatterns) and CI smells.

Duvall Pattern	CI Bad Smell	Rel.
Configurable Third-Party Sw.	A task is implemented using an unsuitable tool/plugin	↓
Configuration Catalog	✗	
Mainline	Number of branches do not fit the project needs/characteristics	↓
Merge Daily	Divergent branches	↑
Protected Configuration	Authentication data is hardcoded under VCS	↑↑
Repository	Pipeline related resources are not versioned	↑↑
Repository	Generated artifacts are versioned, while they should not.	↓
Short-Lived Branches	Divergent branches	↑
Single Command Environment	Some pipeline's tasks are started manually	–
Single Path to Production	Pipeline related resources are not versioned	↑↑
Build Threshold	A build is succeeded when a task is failed or an error is thrown	↑↑
Build Threshold	Coverage thresholds are too high	↓
Build Threshold	Coverage thresholds are fixed on what reached in previous builds	↓
Commit Often	✗	
Continuous Feedback	Missing notification mechanism	↑↑
Continuous Feedback	Failures notif. only sent to teams/devel. that explicitly subscribed	↓
Continuous Feedback	Issue notifications are ignored	↑↑
Continuous Integration	Use of nightly builds	↑
Continuous Integration	Poor build triggering strategy	↑
Continuous Integration	Only the last commit is built, aborting absolete and queued builds.	↓↓
Stop The Line	Build failures are not fixed immediately giving priority to other changes	↑↑
Independent Build	Build scripts are highly dependent upon the IDE	↑↑
Visible Dashboards	Failures notif. only sent to teams/devel. that explicitly subscribed	↓
Automate Tests	Testing is not fully automated	↑↑
Isolate Test Data	Production resources are used for testing purposes	↑↑
Parallel Tests	Independent build jobs are not executed in parallel.	↓↓
Stub Systems	Production resources are used for testing purposes	↑↑
Deployment Pipeline	Some pipelines' tasks are started manually	–
Value-Stream Map	✗	
Dependency Management	Dependency management is not used	↑
Common Language	✗	
Externalize Configuration	Environment variables are not used at all.	↓
Externalize Configuration	Build configurations are cloned in different environments.	↓
Externalize Configuration	Authentication data is hardcoded (in clear) under VCS	↑↑
Fail Fast	Wide and incohesive jobs are used	↓
Fast Builds	Build time for the "commit stage" overcomes the 10-minutes rule	↑↑
Fast Builds	Tasks are not properly distributed among different build stages	↓↓
Scripted Deployment	Some pipelines' tasks are started manually	–
Unified Deployment	Build configurations are cloned in different environments	↓
Binary Integrity	Missing artifacts' repository	↑
Canary Release	✗	
Blue-Green Deployments	✗	
Dark Launching	✗	
Rollback Release	Missing rollback strategy	↑↑
Self-Service Deployment	Developers and operators are kept as separate roles	↑
Automate Provisioning	Developers do not have a complete control of the environment.	↑
Behavior-Driven Monitoring	Missing smoke test, set of tests to verify the testability of the build	↑
Immune Systems	✗	
Lockdown Environments	Developers do not have a complete control of the environment	↑
Production-Like Environments	Lack of testing in a production-like environment	↑↑
Transient Environments	✗	
Database Sandbox	Lack of testing in a production-like environment	↑↑
Database Sandbox	Inappropriate build environment clean-up strategy	↑
Decouple Database	✗	