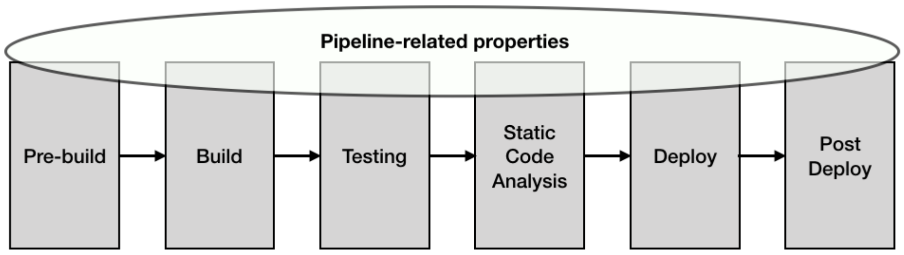

Figure 1 Various stages of a pipeline.

4.1 Desirable Properties of Pipelines and How to Verify Them

Bram Adams (Polytechnique Montreal, CA), Foutse Khomh (Polytechnique Montreal, CA), Philipp Leitner (Chalmers University of Technology – Göteborg, SE), Shane McIntosh (McGill University – Montreal, CA), Sarah Nadi (University of Alberta – Edmonton, CA), Andrew Neitsch (Cisco Systems Canada Co. – Toronto, CA), Christopher J. Parnin (North Carolina State University – Raleigh, US), Gerald Schermann (University of Zurich, CH), Weiyi (Ian) Shang (Concordia University – Montreal, CA), and Hui Song (SINTEF ICT – Oslo, NO)

Release/deployment/delivery pipelines, what we call "pipelines", are becoming fundamental artifacts in modern software organizations. These pipelines automate different stages of a deployment process and have the ability to filter out poor quality commits and target specific delivery endpoints (such as a staging or production environment). Figure 1 shows an overview of such a pipeline. Broadly speaking, a pipeline includes:

1. *Pre-build*: Making decisions about which configurations/platforms/variants to check throughout the pipeline.
2. *Build*: Transforming the sources into deliverable format.
3. *Testing*: Execution of various types of automated tests (e.g., unit, integration, performance, regression).
4. *Static Code Analysis*: Automated analysis of source code for common problematic patterns.
5. *Deploy*: Shipping newly built deliverables to staging and production environments.
6. *Post Deploy*: Making sure that the released application performs/behaves as expected (e.g., through monitoring and production testing).

The recent shift towards continuous deployment has resulted in the need to deploy changes into production environments at an ever faster pace. With continuous deployment, the elapsed time for a change made by a developer to reach a customer can now be measured in days or even hours. Such ultra-fast and automatic changes to production means that testing and verifying the design and implementation of the pipelines is increasingly important. The high-level idea is that pipeline artifacts are also susceptible to the same sorts of problems that we encounter when writing code (e.g., defects, anti-patterns); however, pipeline quality assurance practices are rarely applied (if at all).

So how can we apply quality assurance practices to pipelines? In this blog post, we begin by defining properties that stakeholders would consider desirable (or undesirable) in the pipeline and each phase within it. We provide examples of what could go wrong if a property is violated and some potential avenues for verifying these properties in practice. Finally, future research and tool smiths can leverage these properties in the design of better pipeline engines and tools.

These observations are based on brief discussions between DevOps researchers and practitioners. Our goal is not to advocate for the completeness of this list, but rather to start a discussion within the community around these properties.

4.1.1 Overarching Pipeline Properties

- *Scalability/Cost*: A pipeline needs to be able to deal with a high load of commits coming in
- *Repeatability*: Treating a pipeline as an artifact itself, installing the same pipeline under the same configuration should lead to the exact same results/outcome when fed with the same data
- *Simplicity*: It should be easy to understand of what phases/stages a pipeline consists of and how changes flow through the pipeline.
- *Trustworthiness*: Trusting tools or bots that have privileges to modify the configuration or properties of the pipeline.
- *Security*: Making sure that all phases/steps along the pipeline are reasonably secured (e.g., ports only open when needed)
- *Robustness/Availability*: Involves the different phases/steps that need to be up and running (e.g., staging environments).
- *Velocity/Speed*: The time needed to run through all the phases of the pipeline.
- *Traceability*: At which specific stage/phase of a pipeline is a (code) change currently? Is the pipeline able to route each change along the right sequence of tasks and to detect each problem as soon as possible?

4.1.2 Do I really need to care about the pipeline?

At this point, you may be wondering how violations of the above properties manifest themselves as problems in practice. My code works well, do I really need to care about the pipeline and this long list of properties you provide? The answer is yes! Before diving into the details of each of the above properties and how they are related to each stage of the pipeline, let us talk about some real-world examples first.

Starting with the importance of the scalability of the pipeline: Major projects like Openstack receive so many commits per hour and have thousands of tests to be executed, such that they are not able to trigger the full CI build for each individual commit. Instead, OpenStack's testing automation team[1] came up with custom algorithms to run the CI build on a group of commits, while not losing the ability of singling out the specific commit responsible for failing the (group) build.

Another property to think about is security. If my product code is secure, as well as that of the data servers it connects to etc., what can possibly go wrong? Well, the server that you eventually deploy to, e.g., for others to download your application from, may itself be insecure. A recent example is Eltima's Media Player[2], where a trojan was inserted in the Mac app after it was already deployed to the download server. An older example is a backdoor code change that was added into the Linux kernel CVS mirror of the official BitKeeper repo[3], without any trace where it came from (i.e., it seemingly escaped code review, and likely was

[1] https://archive.fosdem.org/2014/schedule/event/openstack_testing_automation/
[2] https://www.macrumors.com/2017/10/20/eltima-software-infected-with-malware/
[3] https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003/

Table 1 Overview of pipeline properties and in which phases they should be enforced.

	Pipeline	Pre-build	Build	Testing	Static Code Analysis	Deploy	Post Deploy
Scalability	✓			✓			
Repeatability	✓		✓	✓		✓	✓
Simplicity/Understandability	✓						
Trustworthiness	✓						
Security	✓						
Robustness	✓						
Availability	✓					✓	✓
Velocity/Speed	✓		✓	✓	✓	✓	
Cost	✓						
Coverage		✓		✓			
Incrementality			✓		✓	✓	
Correctness			✓		✓	✓	
Regularity							✓
Traceability	✓						✓
Independence			✓				

hacked into the repository). Luckily, it did not end up in kernel releases, since the commit was just inserted into a mirror, but who knows what would have happened if this was not the mirror server!

4.1.3 What are examples of what I can do to avoid this?

In terms of scalability, one solution is that instead of letting pipelines process and/or run every type of file (e.g., shell script) and third-party library as part of a code change that is checked in, the pipeline should verify the type of changes that are allowed to trigger the pipeline, as well as specify explicitly who is allowed to check in what (humans, bots, ...).

To ensure that all servers you use in your pipeline have the same properties (e.g., security, installed software etc.), we can use the concept of Pipeline-as-Code (e.g., https://jenkins.io/blog/2017/09/25/declarative-1/), which allows specifying the pipeline infrastructure in textual form and to automatically instantiate the pipeline, similar to infrastructure-as-code. This enables instantiating identical pipeline instances across different machines, to track changes in the specification (Git), etc. Another option is to implement the concept of *flow testing*. *Flow testing* can be performed in the style of integration testing by validating that a given test commits indeed performs as expected. Did the pipeline hit the major pipeline steps that it should hit (build, test, deployment etc.). For a commit that causes a performance problem, was this problem caught at the right stage and flagged? Each kind of problem requires a "barrage" point behind which we do not want the problem to pass. Finally, specific steps can be taken to ensure the security of all pipeline servers by using checksums and reproducible builds (i.e., a build process that always generates the same binary, bitwise, for a given input), which can further avoid intermediate tampering with build artifacts.

Now that we have given you a big picture of things, let's get down to the details a bit. Table 1 summarizes the different stages of the pipeline and the properties we believe each stage should enforce. In the following sections, we discuss these properties in more detail and provide examples of what can be done to enforce these properties to avoid problems such as those previously described.

Pre-build

Desirable properties:
- *Cost*: The minimal set of configurations/variants/branches should be used within the subsequent phases (Tradeoff with Coverage)
- *Coverage*: The largest set of configurations/variants/branches should be used within subsequent phases (Tradeoff with Cost)
- *Understandability*: How could the engineers understand the configurations as specified in the pipeline

Example problem: It is very easy to waste resources on running test cases on multiple configurations.

Example verification: Through different traceability links, the pipeline can assess if a test case touches certain configuration points and plan accordingly by reducing the number of representative configurations.

Build

Desirable properties:
- *Correctness*: Dependencies are fully expressed.
- *Velocity/Speed*: Builds are completed in a reasonable amount of time (reasonable varies from project to project).
- *Incrementality*: Builds (re)execute the minimal subset of necessary commands to update deliverables without missing any necessary commands.
- *Repeatability*: Given the same input, builds perform the same commands (i.e., deterministic builds). Moreover, it should be possible to reproduce a past build in the future (e.g., if a package service goes down or is no longer available).
- *Independence*: Builds should be isolated from each other.

Example problem: A build phase that does not have the independence property may suffer from builds that interfere with each other. In turn, the build phase may become non-deterministic. For example, builds that are running in parallel or sequentially may access resources from each others' environments by mistake.

Example verification: A possible step towards checking this property could be to apply a form of taint analysis, i.e., track all outputs of a build and check who reads those outputs. Taint analysis has been effectively applied to the analysis of the surface area that is exposed to security issues (e.g., SQL injections). The same concepts may apply to the leakage of state within the scope of builds.

Testing

Desirable properties:

- *Scalability/Cost*: The testing stage needs to be "smart" about its decisions. Depending on the size of the test suite and types of tests available, not every test needs to be run for each commit. Only tests affected by updated artifacts should be run. Good traceability links between code and the test suites, as well as test prioritization can be used to make the testing stage more scalable.
- *Repeatability*: Running the same test suite on the same commit should always produce the same result in terms of passed and failed tests. Flaky tests[4] are especially problematic for repeatability. One solution is to identify/flag flaky tests in order to have special handling for them.
- *Velocity/Speed*: Execution time of test suites is a major bottleneck in the overall velocity of the pipeline. In this phase, velocity/speed is related to the scalability/cost property since smarter test selection will probably eventually lead to better speed of the testing phase.
- *Coverage*: As many possible variants/configurations of a product need to be tested, without sacrificing speed.

Example problem: If no test prioritization/selection strategy is used, large amounts of testing resources can be wasted on not impacted artifacts, delaying the delivery.

Example verification: With a predefined set of mappings between code and tests, given the code changes, the pipeline should trigger and only trigger those tests.

Static Code Analysis

Desired properties:

- *Incrementality*: Only the needed analysis is applied in the release pipeline. For example, the static analysis is only applied on the code that is impacted by the code change. (may remove: Capture the intended properties of the analysis)
- *Correctness*: A static analysis should yield a low rate of false positives, since false positives reduce the trustworthiness of results and lead of gain adoption from the practitioners.
- *Performance*: The static code analysis should be able to finish within a reasonable time, since a long duration of the analysis will affect the deliverable of the product into the next step in the pipeline (e.g., testing or deployment).

Example problem: A typical off-the-shelf static analysis tool often report a large number of issues, while not all of them are of interest to impact the release pipeline. Reporting all the issues, or having all the issues to determine the next step in the pipeline is problematic in practice.

Example verification: A dataset with the issues that may be detected by static code analysis needs to be labeled into whether they are of interest of the practitioners to impact the release pipeline. To test the correctness of the static analysis, a randomly generated sample from the dataset is the test input of this phase and the precision and recall with threshold can be used to assert whether the output is satisfactory.

[4] https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

Deploy

Desired properties:

- *Repeatability*: The deployment results of the software should not be impacted by the deployment environment.
- *Availability*: It is desirable that the service is continuously available during deployment, instead of having to choose between either working nights/weekends to release, or bringing down the service during peak times.
- *Velocity*: A regular release rhythm and/or immediate releases that let developers say that their feature is truly "done done" since it is released to production, can aid development velocity.
- *Incrementality*: An incremental release that initially sends only a fraction of traffic to the new version, and is ramped up until the new version is handling all traffic, can limit the "blast radius" of a problematic release.
- *Correctness*: Deploying software can be risky. A correct process that will not leave the service in a broken state on error is highly desirable.

Example problem: The new version of the service requires configuration for a new feature, but the new configuration has not been applied to the target environment. After deploy, the new version crashes on startup with an error message about invalid configuration.

Example verification: Recently added and changed acceptance tests are run against the newly-deployed service before it is exposed to external clients. Automatic rollback is triggered if the service crashes, or the tests fail.

Post Deploy

Desirable properties:

- *Availability*: Are the deployed artifacts available (e.g., appropriate ports open, heartbeat)
- *Regularity*: Configuration and code changes perform within nominal operational metrics.
- *Traceability*: Data is "collectable" on a continuous basis from various parts of the system, and configuration changes should be auditable and mapable to code changes.
- *Repeatability*: To what extent is the infrastructure resilient to changes in external dependencies, versions, and tools.

Example problem: Inadvertently reusing a feature flag's name can make dead code active again, as happened with the Knight Capital bankruptcy.

Example verification: Turning on an old feature flag could violate two properties: a) *Traceability*, code associated with a feature flag may not have been recently changed, which could set off a warning. b) *Regularity*, the performance of the code with the wrong feature flag may generate metrics that are not consistent with recent range of metrics.