{Coders} Kitchen

# Building the Software Factory

**Niroshan Rajadurai**

**Published on 12/20/2020**

# Table of Contents

# Introduction

One of the biggest challenges with software development today is the unintended propagation of defects or issues through the development cycle of a system. If we consider the simple analogy of our tooth cavities, it is well understood that prevention is better than correction. This holds true for software, where it is 2000%1 more cost-effective to Prevent mistakes than later detect and correct them.

These issues can often be identified very early in the development cycle but are missed because the software is merged without the adequate verification and validation in place.

To address quality issues in software, there are various discussions on the topic of 'Shift Left' (i.e. test earlier in the process). The introduction of Test-Driven Development, Agile, etc. attempt to facilitate the introduction of testing of new functionality at the time of creation, versus being delayed further in the cycle. However, these topics today still focus on enabling the developer but do not address the integration of the pieces of the system, or the updating of features and the side effects they may produce on existing software in the system. Today every aspect of software is growing rapidly; size in lines of code, complexity and the teams that are developing it. The growth is much like Moore's law: exponentially. Improved integration is the secret to this growth; a traditional "develop then test" approach is no longer feasible in today's competitive market.

Moreover, traditional version control tools and strategies were not built to support software development at this scale and did not foresee the level of automated testing that would exist today.

This raises two questions:

1. How can we Shift Left our Quality and Security Assurance (QSA) activities in the Software Development Lifecycle (SDLC)?
2. How can we enable developers to apply Quality and Security Assurance (QSA)-gateways before sharing their code with the rest of the team? (e.g. if developer A introduces technical debt and shares these changes with developer B, the technical debt has now been in-directly inherited by developer B)

# T(x) - Isolate, verify, merge

We can build on an existing concept known as TimeZero (T0) which is used to describe running parts of the QSA process at the developers' desktops (at time zero, when the code is being developed). We can extend this definition and apply it not just to developers but to the Continuous Integration process. The result of T(x) is the ability to Shift Left all possible Quality and Security activities optimally.

This does not, however, answer question 2 about: how can we isolate, verify and then merge developer changes?

The notion of 'isolate, verify, merge' is that when a developer introduces technical debt (e.g. through introducing a failing build, new static analysis issues, failing unit test cases, unapproved open source component) and merges this with a shared development branch, all members working on that development branch will now inherit this technical debt. Similarly, others in the team will likely repeat this and accumulate more technical debt. Now we are in a position where we are working to repay technical debt continuously, and ownership of debt is unclear. What if we could ensure that a developer or a group of developers working on a feature together pay their technical debt before sharing their code with the rest of the team? If we enable developers to isolate their changes and if we enable the verification process to be automated with QSA Gateways at different stages, then we enable an approach to settle our technical debt at the earliest possible stage, allowing the team to focus on features, not repaying debt.

**Example**: Given tool x, the function T(x) will produce a number (typically between 0 and 4) that can communicate with your team at exactly which point in the Continuous Integration process we will run the analysis for that tool. Given that developer A creates a branch for his feature (isolate his change) and has made this branch available on the server repository (making it available to the team and thus our Continuous Integration server) we can now leverage the full power of automation to verify developer A's changes before he is allowed to merge them into shared branches (such as the develop branch, where developer B will inherit code from). This checkpoint in our Continuous Integration process follows T0, so we have called it T1.

---

[1]A short history of the cost per defect metric, Caper Jones, 2012

The real advantage with T1 is that T0 should not be used as a Gateway. Albeit a subjective view, stopping developers from checking in or committing their code is not a viable solution. If a developer is permitted to check-in code with potential technical debt, then we can use T1 to verify this and share and collaborate upon the results with the team. Further to this, we typically run several types of analyses (such as architectural compliance checking, multiple static code analysis tools) that we may not want to burden the developer with running locally on their machine as part of T0.

What we are ultimately creating here is a Continuous Delivery (CD) strategy. CD can be achieved using a development pipeline. A development pipeline breaks down the build of the software into several parts (in general); Build Automation, Continuous Integration, Test Automation, and Deployment Automation.

The idea here is that periodically (ideally on every commit) the full development pipeline is run, with the idea that any issues can be quickly identified. The main challenge is that as the complexity of the system grows, the time taken to run a full development pipeline can grow exponentially until the time taken is so long, that batches of commits need to be validated together. This brings us back to the original issue that when something incorrect is identified, we have to work backwards then to determine which update/commit caused the breakage and invest time to revert this out of the current release branch, so the rest of the commits can progress.

## Quality software assurance gateways (QSA)

The Quality Software Assurance (QSA) Gateways can be used to identify software that is not fit for purpose at the various states of the Development Pipeline and automatically progress or revert commits based on their assessment of Quality. We are building a tier of pipelines, with gated progression to the next level, until finally being merged into the main release branch.

Let us look at how we bring these pieces together to create a complete CD pipeline workflow.

### Development Branch


Figure 1 Example workflow for a development branch

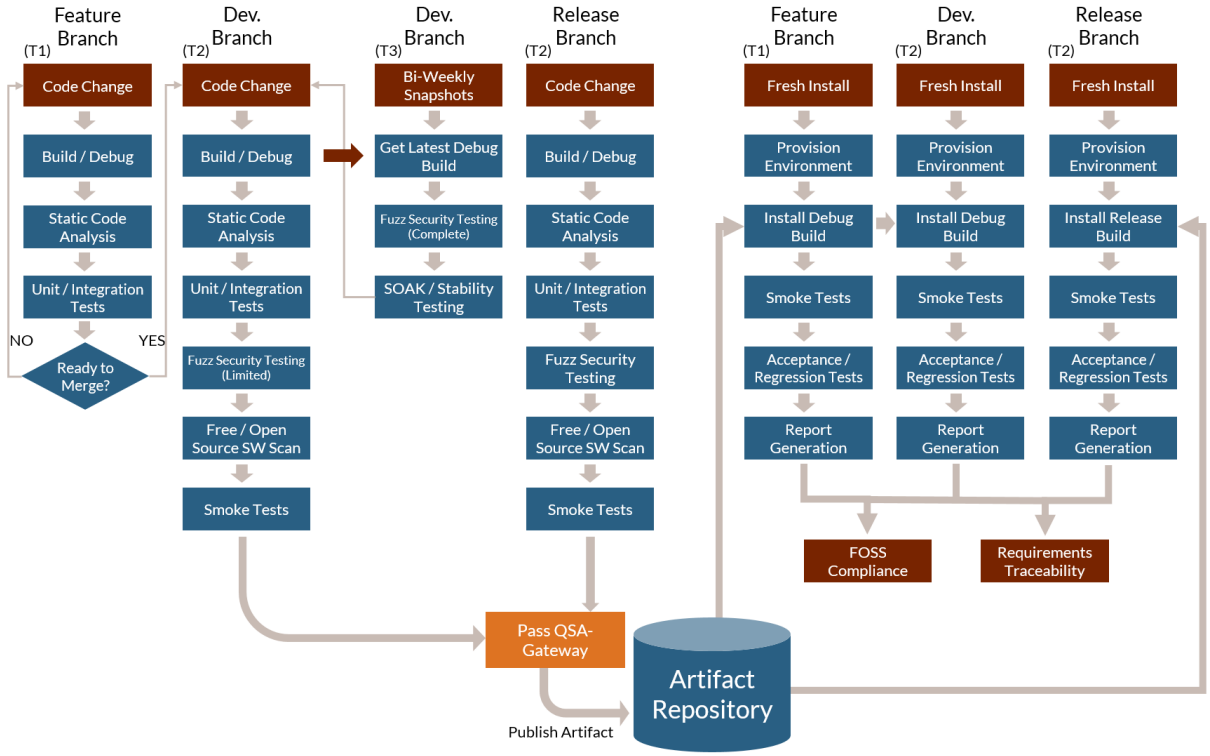


Figure 2 Complete pipeline workflow for a software project

## Static code analysis

**Static code analysis** looks at analyzing the full source code of an application without the need for execution of that program. These tools typically exhibit some of the same behaviours of compilers, but while a compiler produces a binary or object file as its final action, static analysis tools produce a report that is based on deep analysis of the source code. Static Code Analysis can be broken into three types of analysis: Architectural, Programmatic and Security.

**Architectural analysis** tools explore the dependency tree for each element in the software application. Based on this thresholds and specifications can be set for layering componentisation of the software. This provides better guidance for re-engineering of software, and as a result, better maintainability in the longer term.

**Programmatic analysis** tools explore the language semantics to identify potential code that could be ambiguous to the compiler or may result in an undesired side effect because of the way it has been written. Further, they also explore deep data and control flow in the code to ensure that potential issues do not exist as the program might execute.

**Security analysis** looks to perform deep control and data flow analysis but focused on known security exploits or potential newly identified exploits. Typically, if a new vulnerability is found in the code, it is possible that there might be other variants of the same vulnerability in the software. These tools offer the capability to explore the code for other variants of a known vulnerability. They also perform data taint analysis, which provides potential ways the program might be exploited through un-sanitised inputs into the system.

## Unit testing

Unit testing looks to break the software into small modules that are easy to test in isolation. These modules can be single source file, or a class, or even a small collection of these, but still sufficiently small enough to be able to test quickly and easily. Unit Testing can be broken into three levels of quality.

**Level 1 – Input Values Only Test Cases -** test cases serve to dynamically exercise the software and ensure all code is exercised (i.e. 100% code coverage). In many cases, these can be derived directly from the original code (e.g. Basis Path Test Cases)

Using the path analysis and the input parameter analysis for the code under test, we can also construct robustness test cases, which can iterate of the range of the types used for the input parameters to ensure that the function handles all valid and invalid values correctly (.e.g. <MIN>, <MID> and <MAX>).

**Level 2 – Input and Expected Value Test Cases -** test cases not only ensure all code is exercised, but also the behaviour of the software is defined. This means future changes or refactoring activities can detect if existing behaviour has been broken.

**Level 3 – Input, Expected & Traceability to Requirements or Design Test Cases -** extend on Level 2 test cases by having traceability links to the Low-Level Design or Low-Level Requirements. These are the highest value test cases as they not only prove the correctness of existing behaviour, but they also demonstrate the need for the verified software. These types of test cases also benefit in the future when requirements or design changes are proposed, and impact analysis needs to be completed.

## Fuzz testing

Fuzz testing is an automated software testing technique where the program under test is called with random or invalid data values. The program is then monitored to check its behaviour, specifically to see if it crashes, hangs, or exhibits some other form of errant behaviour. In a robust application, it should handle the fuzzed data gracefully and either notify the detection of the incorrect data or disregard it and continue to the next correct data set.

Where the data sets are more complicated, it is possible to use tools to harvest real input data into the program to be tested, and then have the tool mutate the data for the purpose of fuzzing.

Failures due to fuzzing may also indicate potential security vulnerabilities and should be compared to results from Security Analysis to see if any overlap exists.

## Smoke testing

Smoke testing is used to provide quick feedback on the status of the system prior to handing over to the next phase in the software development life cycle (SDLC), typically System Integration or Functional Testing. The test used for Smoke Testing would typically be a smaller set of tests that are used by either the System Integration team or the Functional Test team. These tests would look for simple scenarios like, Can the program be:

- Initialized to a steady-state
- Transitioned through its various commonly used states
- Handle basic error conditions

It is also critical that these smoke tests can be automated so they can be integrated into the pipeline and automatically run every time the software is updated.

## Open source software scan

Open-source security demands a different sort of solution since it does not follow a standard logic like that of code that is written in-house by a single organisation. While an open-source project may have a manager that oversees the code, it often has many contributors who make their own additions and fixes. Moreover, the primary threat to software that makes use of open source components is that of known vulnerabilities that have been published to a range of security databases and advisories, as well as issue trackers where developers go to flag problems that they come across in the code.

Even as this publicised information can be a boon for developers and security teams that are using the code for their products, it is also a valuable resource for hackers who can receive free information on which open source components are vulnerable, and even on how to carry out the exploitation.

Therefore, it essential that a security tool aimed at protecting software that uses open source components be capable of identifying which open source components an organisation is using and matching them with their associated known vulnerabilities to help keep teams ahead of the hackers.

The only technology capable of providing an answer to these concerns is Software Composition Analysis (SCA) tools. In order to be truly useful, though, the SCA solution must be able to run continuously at all points throughout the software development lifecycle, giving it the ability to detect dangerous open source components from the moment that it enters the development environment to post-deployment when newly discovered vulnerabilities or changes in licenses can leave products exposed.

## Version control systems

Typically creating feature or task-related branches in Centralised Version Control Systems is costly, as it means creating extra directories and updating build scripts and configurations to use that branch. The cost of deleting branches is also considered high.

With a Decentralised Version Control System, branching is typically cheaper because branches can be created and deleted without affecting the entire team and without creating additional directories to store these branches. This inherently simplifies the build process for specific branches.

Hence, it is strongly recommended to consider using a Decentralised Version Control System if you want to reap the full benefits of the "Isolate, Verify, Merge" approach.

# Dashboards

A dashboard is used to provide a bird's eye view of the health of the code. It aggregates data from various activities in the SDLC and then combines them to provide a high-level perspective. This data can then be used to prioritise activities in the continued development of the software. A good dashboard solution should offer the following types of information:

- The overall health of the project
- Help highlight hot spots
- Visualize the history of a project
- Drill down into specific details of a project

Some useful metrics[2] to consider when monitoring the health of your software are:

1. Process Health Metrics - This category assesses day-to-day delivery team activities and evaluates process changes.
   a) Cumulative Flow Diagrams
   b) Control Charts
   c) Per cent Complete and Accurate
   d) Flow Efficiency
   e) Time Blocked per Work Item
   f) Blocker Clustering
2. Release Metrics - This group directs attention to identifying impediments to continuous delivery.
   a) Escaped Defects
   b) Escaped Defect Resolution Time
   c) Release Success Rate
   d) Release Time
   e) Time Since the Last Release
   f) Cost Per Release
   g) Release Net Promoter Score
   h) Release Adoption / Install Rate
3. Product Development Metrics - These help to measure the alignment of product features to user needs.
   a) Customer / Business Value Delivered
   b) Risk Burndown
   c) Push / Pull
   d) Product Forecast
   e) Product Net Promoter Score
   f) User Analytics
4. Technical / Code Metrics - The following help determine the quality of implementation and architecture.
   a) Test Coverage
   b) Build Time
   c) Defect Density
   d) Code Churn
   e) Code Ownership
   f) Code Complexity
   g) Coding Standards Adherence
   h) Crash Rate
5. People/Team: The Human Elements - This group of metrics reveals issues that impact a team's sustainable place and level of engagement.
   a) Team Happiness / Morale
   b) Learning Log
   c) Team Tenure
   d) Phone-a-Friend Stats
   e) Whole Team Contribution
   f) Transparency (access to data, access to customers, sharing of learning, successes, and failures via sprint reviews)

[2] 30+ Metrics for Agile Software Development Teams, Andy Cleff, 4th November 2016

## Automated validation profiles

By automating the analysis of the quality of software being developed, we can introduce different quality profiles. The quality profiles could be; prototype, technology demonstrator, R&D for future commercialisation, field deployment no compliance, and formal certification.

Depending on the profile, different elements in the software quality gate can be enabled or disabled. This permits much greater agility within a business and also a clear organisational wide understanding of the quality of a given software component, and the efforts required to deploy it in a different quality profile.

## How do I deploy this?

Let us now consider how to deploy a solution like this within an organization. A typical deployment within an organization can take several months, while the pieces are brought together. For this reason, it is best to take a three-stage approach to deploy a complete Continuous Delivery Solution. The stages are

1. Establish the foundation
2. Integrate the various tools into the CD process
3. Integrate the CD pipeline into the SDLC

## Establish the foundation of continuous delivery

To establish the foundation of our CD deployment, the first thing we need to do is define the architecture and technology specification to be used for Continuous Integration pipeline:

- Infrastructure as code for scalable, maintainable, reproducible tooling
- Virtualization vs Containerization for individual tools
- Virtualization vs Containerization for the build environment

Once these elements have been identified, the plan should be documented and reviewed with relevant stakeholders.

Now we are ready to implement the proposed architecture specification for the initial CD pipeline. We need to consider the following pieces of the foundation:

- Continuous Integration Pipeline Manager
- A tool for local artefact management and retrieval
- A Source Configuration Management System

Depending on the tools being deployed in the foundation, various strategies and features can be taken advantage of. Finally, we need to configure the CD infrastructure to perform our Automated Build. To do this, we need to:

- Define CD process with a branching strategy
- Configure automated change-based build according to branching strategy
- Configure initial quality gateway for source code changes based on successful CI build

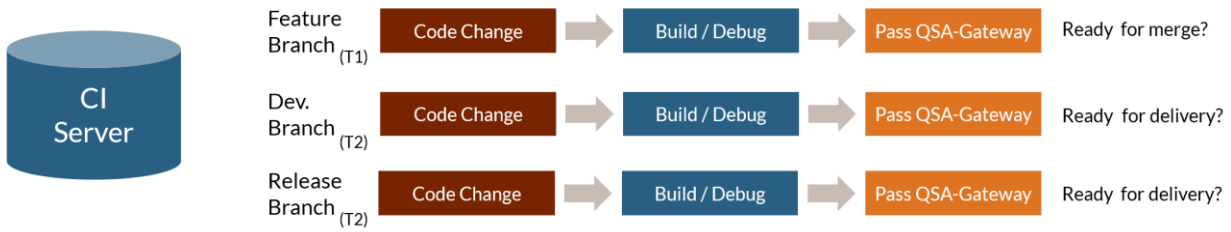


Figure 3 - CD Foundation Setup

## Integrate the various tools into the CD process

Once the foundation has been set up, the next phase is to extend the CD deployment. We can now introduce our static and dynamic source code analysis tools.

The first tool to introduce is a Static Code Quality and Security Analysis tool. As part of setting this up, it is essential to consider which static analysis rules will be enforced. Many standards exist for various languages. Not all might be applicable for your project. It is crucial to consider aspects such as portability and maintainability when selecting these rules.

Next, to consider a tool to manage architectural erosion and define a remediation strategy. To do this, you will need to:

- Future-proof software maintainability through architectural compliance
- Automate architectural analysis as part of Continuous Integration pipeline
- Implement architectural rules detailing compliance criteria according to the intended/desired architectural model (e.g. Enterprise Architect model, design documents)

Finally, we need to consider the dynamic analysis of the software, more specifically, unit-level Verification and Test Automation. We need to define the testing strategy and identify the correct level of requirements or design artefacts we will use for defining the test cases. The idea is that we want to do automated regression testing as part of the Continuous Delivery pipeline.

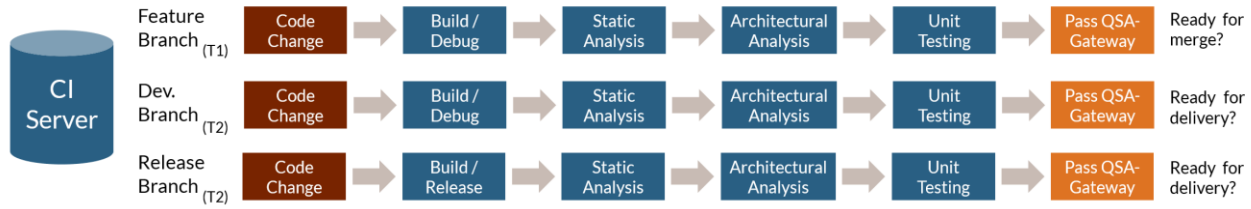


Figure 4 - Tool stack integrated into Continuous Delivery Pipeline

## Continuous delivery as part of the SDLC

Let us revisit our Static Code Quality and Security Analysis. We want to enable these tools at the developers' desk (i.e. T(0) so that we can provide the fastest feedback possible. It is ok if we compromise on a full and complete scan if we can provide quick feedback, as the next stage of the pipeline will do this for us anyway.

After this, we should configure our Dashboard to integrate all the results from the various tools we have in the pipeline.

Finally, we are ready to enforce our Automated Quality Gateway. We can use the aggregation of metrics from the various tools to decide a GO/NOGO on a branch/commit. We can use it to remediate build-up of technical debt, ensuring that code cannot enter the main development branch unless it meets the criteria for software quality and security

## What benefits or KPIs should be observed?

There are several benefits that should come as a result of deploying a Continuous Delivery environment. The improvements should be visible immediately in the Dashboard, with the trend lines showing progress over time as the workflow is integrated and embraced by more and more members of the team.

1. Most errors are prevented, thus lessening the need for latter detection/correction
2. Automated builds, identify broken builds immediately
3. Automated Static Code Analysis
   a) Ensure software does not regress
   b) Comply with coding guidelines (build maintainability, modifiability, and readability into development)
   a) Bug detection – dramatically reduce the number of bugs discovered later in development or in field
   b) Build security testing and defensive coding into the development process and make it part of developers' everyday work
   c) Architectural compliance to future-proof software maintainability and modifiability

4. Automated Unit testing
   a) Automatically integrate pre-verified software
   b) Bug detection – shift left
5. Software Quality Gates
   a) Automatically verify incremental software changes before sharing with team members
   b) "Always releasable" approach from mainline branch
   c) Optimal shift left approach for bug detection at the earliest possible stage
   d) Enable developers to focus on features
6. Virtualisation of environments
   a) Scalable, maintainable, upgradeable, services to minimise downtime
   b) Virtualized, immutable, reproducible build environments that we can really trust – if the build is broken, it is really broken
   c) Scalable build infrastructure
   d) If building against every change, this can scale to 100s of developers working in parallel
7. Other optimizations
   a) Reduced build times and optimised version control usage

# Imprint

**Vector Informatik GmbH**
Ingersheimer Str. 24
70499 Stuttgart
Germany
Phone: +49 711-80670-0
E-mail: info@vector.com

**www.coderskitchen.com**