**GoCD Features**

**Go or GoCD** is an [open source](https://en.wikipedia.org/wiki/Open_source) tool which is used in software development to achieve [continuous delivery (CD)](https://en.wikipedia.org/wiki/Continuous_delivery) of software. It supports automating the entire build-test-release process from code check-in to deployment. It helps to keep producing valuable software in short cycles and ensure that the software can be reliably released at any time.

Go was originally developed at [ThoughtWorks](https://en.wikipedia.org/wiki/ThoughtWorks" \o "ThoughtWorks) Studios in 2007 and was called Cruise before being renamed GoCD in 2010 and released as open source in 2014.

|  |  |
| --- | --- |
| **Written in** | [Java](https://en.wikipedia.org/wiki/Java_(programming_language)), [Ruby](https://en.wikipedia.org/wiki/Ruby_(programming_language)) |

*ThoughtWorks Studios (the product group of [ThoughtWorks](http://www.thoughtworks.com/)) has built over the last 6 years to support*[*continuous delivery*](https://martinfowler.com/delivery.html)*.*

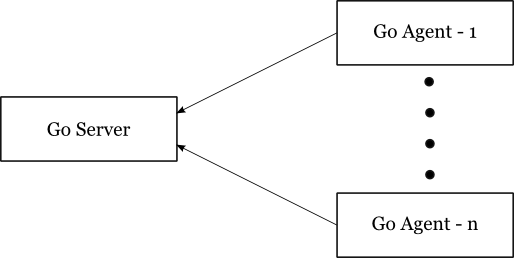
 It’s meant for [**continuous delivery**](https://martinfowler.com/delivery.html) (CD), of which continuous integration is a part.

Go supports easy modeling of sequential and parallel execution, fan-in and fan-out dependency management, and other activities unique to [deployment pipelines](https://martinfowler.com/bliki/DeploymentPipeline.html).

 Almost everything in the ecosystem that Go spans is OSS, from source control and build, to deployment and configuration management

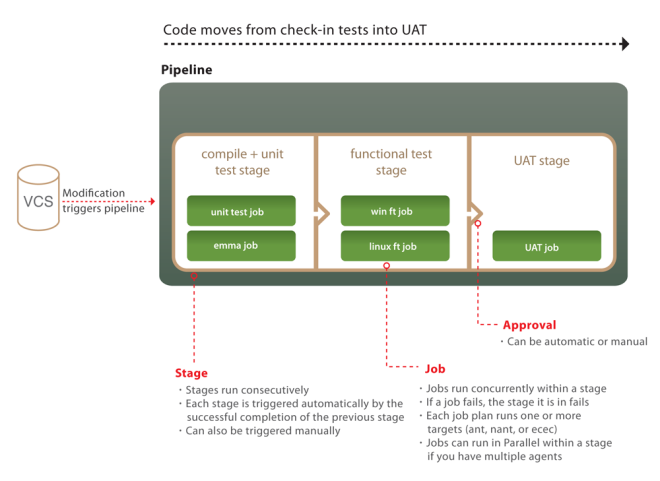
**Architecture**

At the highest level, GoCD consists of two main components, the GoCD Server (referred to as "server") and the GoCD Agent (referred to as "agent").



As soon as the server comes up, it opens two ports (an SSL port and a non-SSL port). The system works on a pull model in the sense that the agents periodically poll the server for work, instead of the server pushing work to the agents. This prevents the agents from having to have listening ports open on their side. The server coordinates everything, making sure that all the builds that need to run get run and all the agents are assigned work when possible.

### **The power of (the right) abstractions**



Although it really is a simplification ([here a more accurate but detail-dense one](https://4.bp.blogspot.com/-UtoYa_o-j9Q/UC4hzS3HneI/AAAAAAAAAPA/txBCydJwamc/s640/Go_abstractions.jpg)), it tries to convey visually 2 very important and often misunderstood/ignored characteristics of GoCD:

1. its 4 built-in powerful abstractions and their relationship: **Tasks** inside **Jobs** inside **Stages** inside **Pipelines**
2. the fact that some are executed in parallel (depending on agents availability) while others sequentially:
   * Multiple Pipelines run in parallel
   * Multiple Stages within a Pipeline run sequentially
   * Multiple Jobs within a Stage run in parallel
   * Multiple Tasks within a Job run sequentially

Without geeking out into Barbara Liskov’s [“The Power of Abstraction”](http://www.infoq.com/presentations/programming-abstraction-liskov)-level of details we can say that a good design is one that finds powerful yet simple abstractions, making a complex problem tractable.

**make your complex and often overcomplicated path from check-in to production tractable**

Model complex workflows

With its parallel and sequential execution, GoCD can easily configure dependencies for fast feedback and on-demand deployment. Its fan-in/fan-out dependency management always does the "right thing," avoiding spurious builds.

**Fan-out**

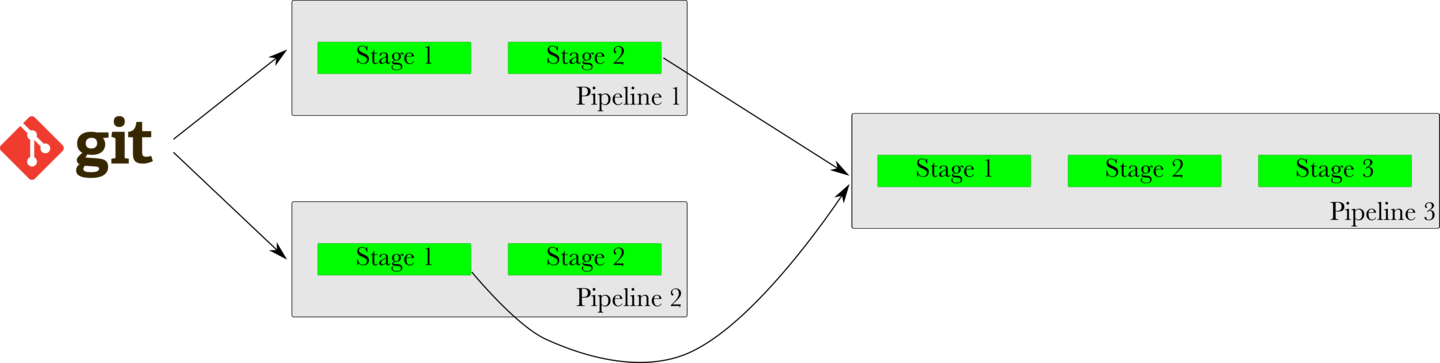
A material is said to "fan-out" to downstream pipelines, when the material's completion causes multiple downstream pipelines to trigger



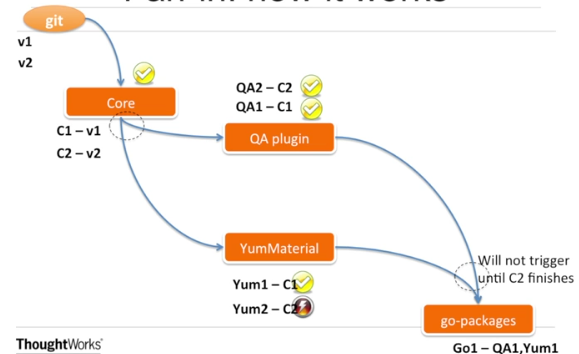
**Fan-in**

A "fan-in" is when multiple upstream materials are needed to trigger a downstream pipeline.

Go will ensure that the revisions of upstream pipelines are consistent, before triggering a downstream pipeline.



If Stage 2 of Pipeline 1 is slow and Stage 1 of Pipeline 2 is quick, Pipeline 3 will wait for Pipeline 1 to finish before triggering. It will not trigger with an inconsistent or old revision of Pipeline 1, just because Pipeline 2 finished quickly.



## Promote trusted artifacts

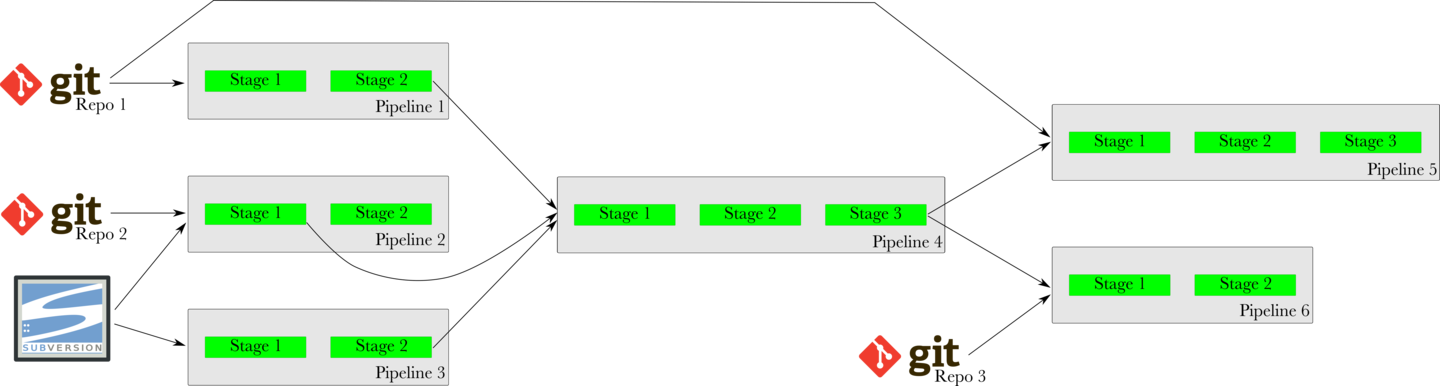
Every pipeline instance is anchored to a particular changeset. GoCD makes it easy to pass once-built binaries between stages so you know exactly what's being deployed and that the binary has been tested.

## Visibility-See how your workflow really works

GoCD's real power is in the visibility it provides over your end-to-end workflow. GoCD's Value Stream Map lets you track a change from commit to deploy at a glance. And when things go wrong, it's easy to see both the upstream cause and the downstream effects.

The Value Stream Map (VSM) is an end-to-end view of a pipeline, its upstream dependencies and the downstream pipelines it triggers. When deciding which pipelines to trigger, Go's fan-in and fan-out resolution will take care of all the dependencies consistently.

For instance, in the image below, when a new commit is found in Repo 1 (git), Go will not trigger Pipeline 5 immediately. It will wait for Pipeline 1 to trigger and finish successfully, then it will wait for Pipeline 4 to trigger and finish successfully. Finally, it will trigger Pipeline 5 with the same revision of Repo 1 that was used with Pipeline 1.



## Deploy any version, any time

GoCD's manual triggers allow you to deploy any known good version of your application to wherever you like. This increases reliability of pushing to production, and empowers QA teams with self-service environments. And, if necessary, it's securable and auditable.

## Run and grok your tests

Verification is a key piece of any deployment pipeline. GoCD will execute tests written in most languages or frameworks. GoCD's agent grid provides parallel and cross-platform execution. GoCD's test reporting will tell you in exactly which changeset and on which platform a test started breaking, which comes in extremely handy when fixing a complex broken build.

## Compare builds

GoCD's compare builds feature can provide a simple bill of materials for any deployment. Perhaps more powerful is its ability to compare the content - both files and commit messages - across any two arbitrary builds. This is invaluable when troubleshooting a broken pipeline.

## Eliminate bottlenecks

GoCD's agent grid eliminates bottlenecks, providing trivial parallel execution across pipelines, platforms, versions, branches, etc.

## Keep configuration tidy

Easily reuse pipeline configurations via GoCD's template system. This makes managing pipelines for versions and branches easy as pie.

## Trust your team; be responsible

While most enterprise applications go overboard with their permission model, GoCD seeks to provide just enough. In particular, GoCD supports auditable deployment and can delegate the configuration of pipelines to users without full-blown admin privileges.

## Plugins

GoCD has extension points for which plugins can be created. There are [numerous plugins](https://www.gocd.io/plugins/) already available, or if you don't see what you need, you can [write your own](https://developer.gocd.io/current/writing_go_plugins/go_plugins_basics.html).