KDAB's Software Development Best Practices

CI and CD

Jan Marker | Senior Software Engineer, KDAB

KDAB

The software engineering community has learned hard lessons over time about the best way to build software. Following tried-and-true practices provides a huge number of benefits: increased software resilience, faster release schedules, higher product quality, more effective teamwork, and happier developers.

In a modern software development process, continuous integration (CI) and continuous deployment (CD) plays a vital role. However, implementing CI/CD within an organization can be challenging. Due to the need for customization to align with specific work products and workflows, it often requires some trial and error. To help you minimize needless dead-ends, we'll share some of our hard-won advice on using and configuring CI/CD systems, as well as explain why you should be using CI/CD if you're not already.

1. Basics

1.1. Why CI/CD?

Code reviews are already an integral part of your process, and their significant contribution to improving code quality is widely recognized. Now, imagine **the CI process as an additional, automated reviewer** for your code. Every time a change is pushed into the repository, your CI "reviewer" ensures that the software successfully builds across all platforms and passes all tests before moving forward. In addition to catching compiler warnings and validating unit tests, the CI build can also include static analysis tools (like Clang or Clazy for C++) and other linters and sanitizers to identify other potential issues. This automated computer-blessed code review helps maintain quality and stability across all platforms. This alone justifies the time investment to set up CI.

Another aspect is code consistency. If your developers' IDE settings are individually configured to format line breaks, bracket/brace placement, spacing, comments, and identifiers, it's easy to introduce

multiple competitive styles into a large code base. Such competing styles do not promote code readability or consistency. The CI build can use tools to verify that consistent formatting styles are applied across all source files. This is especially helpful when working with large and distributed teams. Additionally, these tools can check for spelling errors in variable names and comments. By performing these checks within the CI system, all contributors will automatically adhere to the same standards, eliminating the need for time-consuming debates on formatting issues during human reviews.

Finally, while developers should run unit tests themselves, for expediency, they often focus on the tests immediately impacted by their changes. So, CI does two great things for the developer. Firstly, because the CI system runs all tests and not just a restricted set, it can uncover unforeseen issues that the developer may have missed. Secondly, it saves valuable developer time. Developers can confidently limit themselves to a small set of local tests to ensure the code passes an initial sanity check. Because the CI testing is done asynchronously on dedicated CI systems, they know comprehensive testing will still be done.

1.2. What do you need to make it work?

At a minimum, you need a shared repository that all developers are working on. In a later section, we'll cover how and why you'll want to implement your CI in an on-prem or cloud solution, but if you're implementing an on-prem solution, **you need a dedicated CI server** and, depending on the size and complexity of your project, several CI worker machines.

Whether you need a dedicated CI person or not depends on the CI system that you choose, which we'll discuss later. However, it's best if you **have a dedicated CI person**. If that's not possible, you can manage with two or three people that have CI as their part-time role to start, although clearly their work will be often interrupted. Essentially,

some systems distribute CI knowledge and responsibility among all the engineering staff, while others concentrate it in a dedicated CI team.

Depending on which CI tool chain you use, it can be pretty straightforward to set up – in simple cases, within a day or two. For example, GitHub Actions (and similar cloud-based services) offer a pre-paid service that is already configured and working. You need to understand how to customize it for your build, but you don't need to acquire and set up build machines.

On the other hand, a solution like Jenkins or Buildbot will require setting up servers, installing software, setting up worker machines, and keeping the OS updated, and customizing it for your project. Since user-centric OSes like Windows or Mac are not really intended to be run as servers, you'll probably need to include automation systems like Chef or Ansible along with special setups for unattended operation so that reboot logins or application crashes don't halt the CI process. The benefit, however, is that Jenkins or Buildbot can more easily give you full control of your CI environment, which is often preferable and may be necessary.

To determine the right approach for you, read through the rest of this document and focus on the sections about cloud versus on-prem and the individual tools.

1.2. Is CI of value without unit tests?

Yes! Especially at the beginning of a project, you really don't need a lot of tests for CI to be effective. As mentioned before, you still get a lot of value from compiling on other platforms, performing static analysis, and running format checkers. But you can easily start with a small number of effective tests that validate important items. They won't be too hard to write, and they'll probably fail more often than you expect. **Start your test framework simple** and accumulate new tests as you find and fix bugs.

2. Comon methodologies

2.1. Splitting your CI

A well-working CI system becomes an automatic part of the development workflow, greatly aiding overall quality. However, one of the most important attributes of your CI system is speed. The CI builds need to be fast – once a commit is pushed to the repository, developers should get feedback within 10 minutes, not 10 hours. The longer the feedback loop, the less that developers will rely on it and the more they'll be tempted to ignore it. But big, complex projects can be very challenging to build quickly, no matter how many Gigahertz you throw at it. That's why we recommend that you **have at least two distinct CI build configurations**, one for quick per-commit builds and one for thorough nightly builds.

The **quick builds** will be run throughout the day as developers commit their changes. They should take advantage of pre-built dependencies and cache as much as possible. The static checkers and sanitizers that do get run in a quick build should be with settings that are the most effective, not necessarily most comprehensive. This will give developers feedback on the validity of their changes while requiring the least amount of work, delivering results quickly.

The **nightly builds** should do everything from scratch. Start with just source and empty binary directories. Turn on all the static checks that make sense for your code. Include all the necessary installers and packagers. If all your developers are in the same time zone, the nightly build can use the same build machines as the quick build, although with round-the-clock development, you might need a dedicated machine that does your thorough once-a-day build.

In cases where your development team needs to periodically

share updates outside the organization, such as when customers are involved in the testing cycle, you may need one more build configuration: a **packaging build**. This build is more complete than quick builds as it's done from scratch, but because it can rely on the nightly build to do the full gamut of checks, it can be a bit lighter weight. This build can be run several times a day as developers interact with the off-site testing team. Alternatively, if the packaging step is sufficiently fast, it can be incorporated into the quick builds.

2.2 . Overriding CI gating

Developer tools (GitHub, Gerrit, Bitbucket, etc) in a CI workflow usually have an option for a CI-gated check-in. This step requires that the CI system has validated any developer changes before commits are merged back into the main branch. It ensures that every commit properly passes compiler and unit tests.

However, it's recommended to **avoid making CI-gated check-ins** optional in the development workflow. If developers get in the habit of skipping the CI build or bypassing the CI results, the value provided by CI can rapidly diminish. Because if your CI system isn't working right, developers may override it.

For example, if the build takes too long because the CI system experiences a network problem, developers may choose to bypass the CI check. Since it's easier to bypass checks instead of addressing underlying CI build issues, the team may eventually stop using CI altogether.

This emphasizes that the CI system must be reliable, quick, and accurate. If you intend to enforce CI validation for each commit, it is essential to ensure that the CI system receives continuous maintenance and consistently works properly.

2.3 . CI for non-web applications

. While CI techniques have broad applicability, the implementation of continuous deployment (CD) differs depending on the type of development. It's worth noting that the quality of CI/CD tooling often excels in the realm of web development. The process for CD in web development is relatively simple since there is a single platform and a controlled number of distribution endpoints. Once a source code commit is deemed valid and verified, changes can be immediately published.

CD for embedded and desktop development has a couple of extra steps. It must first create a downloadable version of the application – in other words, an installer and/or over-the-air packages for all supported platforms. Then, it needs to publish a metadata file that points to those package file URLs along with their versions. Finally, either manually initiated by the user or kicked off by an automated process, the target machines will consult that publicly accessible metadata file to find the installer or update, check if it's newer than the current software, and download and install it.

The big difference between these models for the CI/CD system is that in general with **desktop and embedded system development, CD should not be done off the main branch**. (Note that there might be exceptions if the target is a single very stable hardware platform, but planning ahead doesn't hurt.)

There are a few possible branches you will need:

- Normal work is done on the main development branch, and this is the branch that the CI system runs off of.
- The user-facing CD process needs more control, so it runs on a separate release branch that's used for generating testable packages. Once a release manager says the software is ready to go, it then gets packaged as an official versioned release off this branch.

2.4. Cloud versus on-prem CI

Should you use a cloud-based CI system or build your own on-premises system?

The simplicity of cloud-based CI makes it highly attractive. Someone else has done all the work – you don't need to acquire hardware through your purchasing department, work with your IT organization, order huge hard disks or beefy processors, or install and configure all the many bits of software. **You can start working right away with a cloud CI system.**

The same is true if you think your CI needs are going to change quickly, scaling either up or down. It's far easier to grow or shrink your cloud assets as needed than to automatically configure new VMs or to acquire new machines for your CI server room. **If your CI needs to ramp up quickly or scale dynamically then go with a cloud-based system.**

Additionally, **if you're already using a cloud hosting platform** like GitHub or GitLab, **incorporating a related cloud-based CI solution** like GitHub Actions becomes significantly easier.

While cloud can be a very easy way to get into CI, there are two big factors to consider. The first is cost. You'll be racking up cloud processing and storage fees when your CI system runs, and it will be generating several cross-platform builds many times per day. That can turn out to be a rather expensive investment with your cloud provider and, in fact, **cost is probably the biggest reason to go on-prem**.

Another reason might be data privacy. Privacy and security regulations that constrain code and data storage by physical location, security methodology, or data handling rules can make it difficult to use the cloud. You'll probably need to go **on-prem if you need to follow strict data privacy or security rules** either as a result of your region (for example, complying to GDPR in

Europe), or industry (adhering to military or medical data handling requirements), or your own company's IT and cybersecurity policies.

3. Exploring solutions

At the current point in time, there are three CI solutions that are taking up most of the discourse. There are many others in use, but a team setting up a new CI/CD system will likely encounter one of these three main options. Their main distinctions are whether CI work is concentrated in a dedicated team or spread throughout all developers in the project, and whether the CI system is based in the cloud or not.

3.1. Buildbot

Buildbot is an on-prem solution that maintains all CI information in a single place. With it, you can specify what projects to build and how to build them. Lots of critical projects use Buildbot such as Python, Webkit, LLVM, Mecurial, Blender, GDB, Gentoo Linux, and Yocto.

Buildbot needs a system administrator to keep CI software up to date. It assumes the use of VMs for both the build master and the workers. It's necessary to provide your own hardware and set up this environment accordingly. With Buildbot, it's possible to construct either a pet or cattle environment for the workers, and it supports both load balancing across workers and pinning projects to specific workers.

Buildbot's design offers pros and cons.

Pros:

- **Single point:** Maintainers can quickly make changes that impact all projects, like adding a new static analysis tool or validating against a new set of checks.

- ·**Scaleable:** Scaling with Buildbot is easier for multiple projects that use the same tool chain.
- ·**Flexible:** The system is very configurable and can be made to align with your team's preferred workflow. It provides exceptional flexibility since it uses Python code, but it's also simple to set up for easy use cases.

Cons:

- ·**Maintainer:** Buildbot lacks a user-friendly web portal for easily controlling its settings. Instead, it's necessary to modify code and configuration files, which means you pretty much need a dedicated Buildbot person or team to effectively maintain the system.
- ·**Complexity:** With flexibility comes with increased complexity. As the number of projects grows, the configuration complexity can escalate quickly, making the set up technically demanding.
- ·**Non-SaaS-able:** Despite its capability to create distinct servers, Buildbot lacks support for crucial concepts needed to service multiple external clients, like separating projects and users into distinct access groups.

Overall, **Buildbot** is the better tool when dealing with **many similar projects** (in terms of language, framework, and dependencies) and a **need to scale easily**. It's is also better for companies who need to highly customize their CI workflow. Buildbot is a good choice if you have many **projects that are consistent in workflow but relatively unique in their dependencies**. However in this case, Jenkins is also a viable alternative, which we'll discuss next.

3.2. Jenkins

Jenkins is another on-prem solution. Unlike Buildbot's centralized

approach to CI configuration, Jenkins offers the flexibility to either centralize or distribute CI information throughout the source repository as a set of configuration files that drive the build. Jenkins is also widely used with companies such as FaceBook, Netflix, Udemy, and Twitch.

Jenkins is similar to Buildbot in several ways. Like Buildbot, Jenkins needs an administrator to manage the system (for all but the simplest use cases). It uses VMs for both the build master and the workers, and you are responsible for providing the necessary hardware and configuring the environment, similar to the efforts involved in setting up Buildbot. Additionally, it supports both pet and cattle workers and can load balance tasks across workers and pin tasks to specific workers.

Differences between the two environments start entering into the picture when considering flexibility and ease of use.

Pros:

- **Non-centralized**: While you can use a centralized CI configuration, you don't need to. If you use a non-centralized structure, Jenkins lets the CI configuration be easily changed on a per-project basis by developers. •
- **CI configuration** within a project is also easier if you have many projects with a diverse set of tools, languages, libraries, and platforms.
- **Easy on-prem**: Jenkins tries to provide a plug-and-play setup and uses a flexible plug-in architecture, making it easy to use for more common use cases. Its simplicity is particularly evident when used by someone with deep Jenkins knowledge. Because it uses a clean web interface to administer projects and build configs, it may not even require a dedicated person for simple use cases.

- **SaaS-able**: Jenkins offers robust features related to user management, security, and visibility, making it possible to create a CI/CD environment that can support projects involving multiple clients or customers.

Cons:

- **Change**: Because the CI configuration is (often) scattered throughout the source tree, it can be harder to implement global changes consistently.
- ·**Scale**: Similarly, Jenkins can be more difficult to scale when adding or changing multiple projects.
- ·**Adaptability**: Not every configuration is provided out of the box, and plug-ins don't cover all use cases. If what you need is outside existing workflows, the customization or modification of plug-ins becomes necessary. This involves using Groovy, a scripting language specific to Jenkins. While Groovy is flexible, it does not offer the same power as the Python code used in Buildbot.

Since it's easy to understand, set up, and use, **Jenkins** is the best tool **if you're getting started in CI and need an on-prem solution**. Jenkins is a good choice if you have many projects that are **consistent in workflow but relatively unique in their dependencies**. Because Jenkins is more "opinionated" as to how it works, it's not as easily used by teams with a lot of projects that have distinct workflow requirements. While you might be able to get away without a dedicated CI person for a Jenkins setup, it is similar to Buildbot in that it would still greatly benefit from having a dedicated person (or team).

3.3. GitHub Actions

GitHub Actions (GHA) is a cloud-based CI solution that is an option

for GitHub users and offered by GitHub as a freemium option (free for open source, paid for private repos). Like Jenkins, it uses project-contained configuration files.

Pros:

- **Ease of use**. GHA is straightforward to use and very friendly for the uninitiated with lots of pre-existing prepared environments and scripts to handle common problems, making it the simplest to get started.
- **Public projects**. GHA makes perfect sense and is a great fit for anyone who's contributing to open-source projects and needs a CI system.
- **Maintenance**. You don't need a dedicated CI person, since maintenance work is being handled via the cloud service.

Cons:

- **Platform**. GHA is tied to the GitHub ecosystem, so you can't use GHA if you need an on-prem solution, you'd prefer another system besides GitHub, or you need flexibility in your repository platform.
- **Cost**. Whereas Jenkins and Buildbot are open-source and freely available, GHA uses a model where you have to pay to use it with private repositories.

GHA offers the simplest way to get started in CI. If you're working on a project that has external code contributions, you may want to consider using **GHA** since it **works great for open-source projects**. (That's also because CI systems can run user-submitted code as part of the build process; you want any potentially unsafe contributions contained in a server environment!) However, GHA has some limitations that make it harder for private companies to use. And notably, while we've only

discussed GHA for GitHub, there are similar solutions with similar pros and cons for many other popular repository platforms, such as GitLab CI/CD for GitLab or Bitbucket Pipelines for Bitbucket.

or other optional features, requiring a plug-in based architecture. But to make plug-ins effective, you have to ensure they have access to all the bits that make them work properly. That often requires continually expanding the scope of the plug-in interface. In turn, this drives a desire to make nearly everything in the program a plug-in since that approach ensures the plug-in APIs are all being properly exercised and are complete. However, if you've ever built an application with plug-ins, you know that they introduce a number of problems into development. Plug-ins can prevent cross-application compiler optimizations, and because plug-ins run as modules loaded by the application, they can be very difficult to test and debug.

If you need to incorporate plug-ins, make sure that you **design plug-ins to add features that are truly optional**, and bring all other functionality into the main development branch. This helps you keep the plug-in scope from taking over the entire application and turning development into an awkward and slow process.

3.4. CI Helpers

There are a couple of noteworthy tools for desktop and embedded development that, although not explicitly designed as CI solutions, significantly contribute to streamlining the CI process.

CMake Presets enhances the functionality of the CMake build tool by enabling the creation of multiple different build configurations. CMake defines the structure of a build with environment variables, dependency locations setting which files or modules need to be built. With CMake Presets, you can organize different parts of the build process into different preset configurations. This allows you to create a build configuration specific to CI, and use it alongside

the standard developer build process. This feature is especially useful for larger projects, where much of the environment is shared between multiple types of build.

pre-commit is a framework for managing pre-commit hooks in Git that allowing certain processes to run before files are submitted for code review. This can be used to set up static code checkers and code formatting tools, running those features on the developer's computer for a speedy turnaround. This approach can take a lot of load off the CI system.

What is KDAB's Software Development Best Practice series?

This series of whitepapers captures some of the hard-won experience that our senior engineering staff has developed over many years and projects. Offered up as a grab bag of techniques and approaches, we believe that these tips have helped us improve the overall development experience and quality of the resulting software. We hope they can offer the same benefits to you.

View all three parts of this whitepaper series online at: ***www.kdab.com/publications/bestpractices/***

About the KDAB Group

The KDAB Group is the world's leading software consultancy for architecture, development and design of Qt, C++ and OpenGL applications across desktop, embedded and mobile platforms. KDAB is the biggest independent contributor to Qt and is the world's first ISO 9001 certified Qt consulting and development company. Our experts build run-times, mix native and web technologies, solve hardware stack performance issues and porting problems for hundreds of customers, many among the Fortune 500. KDAB's tools and extensive experience in creating, debugging, profiling and porting complex applications help developers worldwide to deliver successful projects. KDAB's trainers, all full-time developers, provide market leading, hands-on, training for Qt, OpenGL and modern C++ in multiple languages.

www.kdab.com