J. Beningo, *Embedded Software Design*
https://doi.org/10.1007/978-1-4842-8279-3_7

7. Embedded DevOps

Jacob Beningo[1]
(1) Linden, MI, USA

If you've been paying any attention to the software industry over the last several years, you know that DevOps has been a big industry buzzword. DevOps is defined as a software engineering methodology that aims to integrate the work of software development and software operations teams by facilitating a culture of collaboration and shared responsibility.[1] In addition, DevOps seeks to increase the efficiency, speed, and security of the software development and delivery process. In other words, DevOps promises the same dream developers have been chasing for more than half a century: better software, faster. DevOps encompasses a lot of agile philosophy and rolls it up into a few actionable methodologies, although if you read the highlights, it'll give you that warm and fuzzy feeling with few obvious actionable items.

DevOps processes for several years have primarily been implemented in mobile and cloud applications that are "pure applications" with few to any hardware dependencies. Software giants like Google, Facebook, and Amazon have used these philosophies for several years[2] successfully. DevOps success by the tech giants has caused interest in it to filter down to smaller businesses.

In general, embedded software teams have often overlooked DevOps, choosing to use more traditional development approaches due to the hardware dependencies. However, the interest has been growing, and it's not uncommon to hear teams talk about "doing DevOps." From an embedded perspective, delivering embedded software faster is too tempting to ignore. Many teams are beginning to investigate how they can leverage DevOps to improve their development, quality assurance, and deployment processes.

This chapter will look at how embedded software teams can modernize their development and delivery processes. We will explore Embedded DevOps, applying modern DevOps processes to embedded software. In doing so, you'll learn what the Embedded DevOps processes entail and how you might be able to leverage them to develop and deploy a higher-quality software product.

A DevOps Overview

DevOps is all about improving the efficiency, speed, and quality of software that is delivered to the customer. To successfully accomplish such a feat requires the involvement of three key teams: developers, operations, and quality assurance.

Developers are the team that architects, designs, and constructs the software. Developers are responsible for creating the software and creating the first level of tests such as unit tests, integration tests, etc. I suspect that most of you reading this book fall into this category.

Operations is the team that is responsible for managing the effectiveness and efficiency of the business. Where DevOps is concerned, these are the team members that are managing the developers, the QA team, schedules, and even the flow of customer feedback to the appropriate teams. Operations often also encompasses IT, the members who are handling the cloud infrastructure, framework managements, and so forth.

Quality assurance (QA) is the team that is responsible for making sure that the software product meets the quality needs of the customer. They are often involved in testing the system, checking requirements, and identifying bugs. In some instances, they may even be involved in security testing. The QA team feeds information back to the developers.

Traditionally, each team works in their own silo, separate from the other teams. Unfortunately, when having separate teams it's not uncommon for competition to get involved and an "us vs. them" mentality to arise. Mentalities like this are counterproductive and decrease effectiveness and efficiency. The power of DevOps comes from the intersection, the cooperation, and the communication between these three teams. DevOps becomes the glue that binds these teams together in a

collaborative and effective manner as shown in Figure 7-1.

Figure 7-1 The three teams involved in successfully delivering a software product collaborate and communicate through DevOps processes

When teams discuss DevOps, they often immediately jump to conversations about continuous integration and continuous deployment (CI/CD). While CI/CD is an essential tool in DevOps, it does not define what DevOps is on its own, even though it encompasses many DevOps principles and provides one path for collaboration between the various teams. Four principles guide all DevOps processes:

1. Focus on providing incremental value to the users or customers in small and frequent iterations.
2. Improve collaboration and communication between development and operations teams. (These improvements can also be within a single team and raise awareness about the software product itself.)
3. Automate as much of the software development life cycle as possible.
4. Continuously improve the software product.

Do these principles sound familiar? If you are at all familiar with agile methods, then it should! The four primary values associated with Agile from the Manifesto for Agile Software Development[3] include

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

DevOps is the new buzzword that simply covers the modern-day practices of Agile. I'd go one step further to say that it hyperfocuses on specific values and principles from the agile movement particularly where "automation" and "continuous" are involved. In case you are not familiar with agile principles, I've included them in Appendix B. Note that principles 1, 3, 4, 8, 9, and 12 sound exactly like the principles that DevOps has adopted, just in a different written form.

DevOps Principles in Action

DevOps principles are often summarized using a visualization that shows the interaction between the company and the customer as shown in Figure 7-2. The relationships contain two key points. First, a delivery pipeline is used to deploy software with new features to the customer. Next, a feedback pipeline is used to provide customer feedback to the company which can then in turn be used to create actions that feed the delivery pipeline. In theory, you can end up with an infinite loop of new software and feedback.

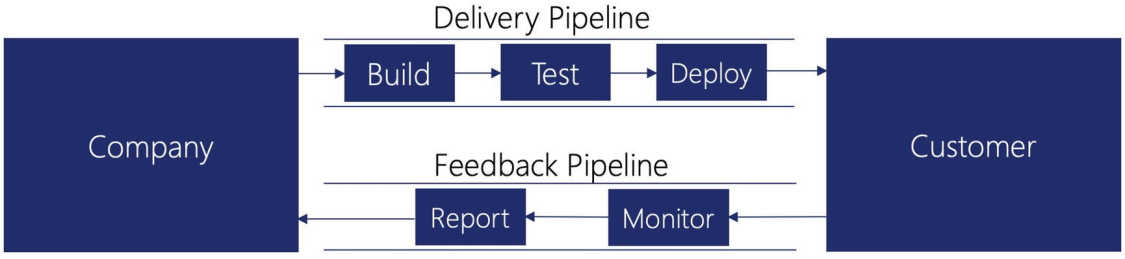

Figure 7-2 A high-level overview of the DevOps process

The delivery pipeline is where most developers focus their attention. The delivery pipeline provides a mechanism for developers to build, test, and deploy their software. The delivery pipeline is also where the QA team can build in their automated testing, perform code reviews, and approve

the software before it is delivered to the customer. When people discuss DevOps, in my experience, most of the focus isn't on customer relations or "facilitating a culture of collaboration and shared responsibility" but on continuous integration and continuous deployment (CI/CD) processes that are used to automate the delivery process. We'll talk more about what this is and entails later in the chapter.

There are at least five parts to every successful delivery pipeline, which include

- Source control management through a tool like Git
- A CI/CD framework for managing pipelines and jobs
- Build automation tools such as Docker containers, compilers, metric analyzers, etc.
- Code testing and analysis frameworks
- Review, approval, and deployment tools

There are certainly several ways to architect a delivery pipeline through CI/CD tools, but the five points I've mentioned tend to be the most common. We'll discuss this later in the chapter as well in the CI/CD section.

So far, I've found that when embedded teams focus on DevOps processes, they jump in and highly focus on the delivery pipeline. However, the feedback pipeline can be just as crucial to a team. The feedback pipeline helps the development team understand how their devices are operating in the field and whether they are meeting the customers' expectations or not. The feedback from the devices and the customers can drive future development efforts, providing a feedback loop that quickly helps the product reach its full value proposition.

While the user feedback from the feedback pipeline can sometimes be automated, it is often a manual process to pull feedback from the user. However, in many cases, the engineering team may be interested in how the device performs if there are any crashes, bugs, memory issues, etc. Getting technical engineering feedback can be automated in the deployed code! Developers can use tools like **Percepio DevAlert**[4] or the tools from **MemFault**[5] to monitor and generate reports about how the device is performing and if it is encountering any bugs.

Note When using device monitoring tools, you must account for your system's extra memory and processor overhead. In general, this is usually less than 1–2%. Well worth the cost!

Now that we have a general, fundamental understanding of what DevOps is supposed to be, let's take a look at what the DevOps delivery pipeline process might look like.

Embedded DevOps Delivery Pipeline

It's critical that as part of Embedded DevOps, we define processes that we can use to achieve the goals and principles that DevOps offer. After all, these principles should provide us with actionable behaviors that can improve the condition of our customers and our team. Before we dive into details about tools and CI/CD, we should have a process in place that shows how we will move our software from the developer to QA and through operations to the customer. Figure 7-3 shows a process that we are going to be discussing in detail and for now consider our Embedded DevOps process.

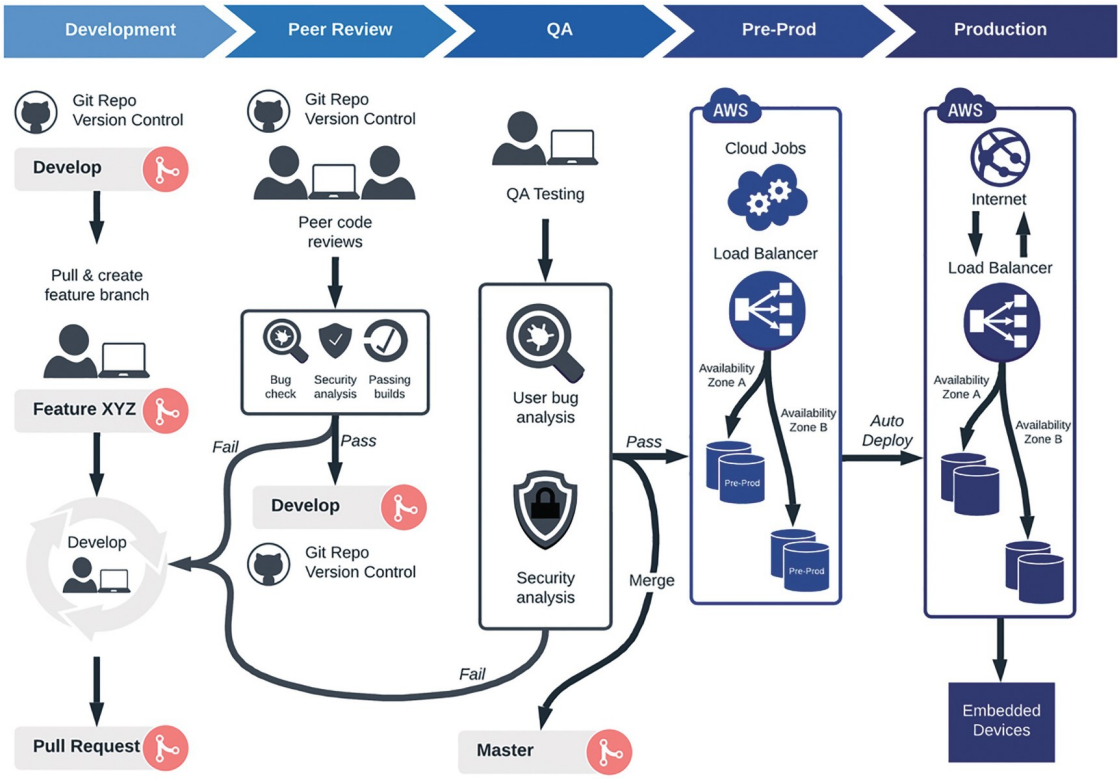

Figure 7-3 An example Embedded DevOps delivery pipeline process[6]

The Embedded DevOps process is broken up into five steps which include

- Development

- Peer review
- QA
- Preproduction
- Production

Each step assists the business in coordinating collaboration between the developers, QA, and operations. The process can also be done in quick intervals that result in reaching production several times per day, or at a slower pace if it makes more sense for the business. Let's discuss this process in a bit more detail.

First, development is where we are creating new features and fixing product bugs. The development team leverages a Git repo that manages the software version. Typically, we save the master branch for code that is nearing an official release. Official versions are tagged. We use the Develop branch as our primary integration branch. In development, we create a new feature branch off the Develop branch. Once a feature is completed by the developer, they create a pull request to kick off the peer review process.

The peer review process occurs on the feature branch prior to merging the feature into Develop, the integration branch. Peer reviews should cover several things to ensure that the feature is ready to merge into the Develop branch including

- Verifying the feature meets requirements
- Performing a code review
- Testing and code analysis
- A security analysis (assuming security is important)
- Verifying the feature builds without warnings and errors

Once the team is convinced that the feature is ready and meets the requirements for use and quality, a merge request can be created to merge the new feature into the Develop branch. If any problems are discovered, the development team is notified. The development team will then take the feedback and adjust. Eventually, a pull request is created, and the process repeats until successful.

Once a new feature is ready, a new version might be immediately released, or a release may not take place for an indefinite period of time while additional features are developed. When a new version is going to be released, the QA team will get involved to perform their testing. Typically, the QA team is testing at a system level to ensure that all the desired features work as expected and that the system meets the customer requirements. The QA team will search for bugs and defects in addition to potentially performing additional security testing.

If the QA team discovers any bugs or problems, the discovery is provided to the development team. The development team can then schedule fixes for the bugs and the process resumes with a pull request. If the software passes the QA testing, then the new features will be merged into the master branch, tagged and versioned. The new software then heads to preproduction where the new version is prepared to be pushed out to device fleet.

Keep in mind that the preproduction and production processes will vary dramatically depending on your product. In our process, I've assumed that we are working with an Internet-connected device. If our devices instead were stand-alone devices, we would not have a cloud service provider and jobs that push out the new firmware. Instead, we might have a process that emails updated versions to the customer or just posts it to a website for customers to come and find.

DevOps folks get really excited about automating everything, but just because people get excited about the buzz doesn't mean that it makes business sense for you to jump on board as well. It is perfectly acceptable, and often less complex and costly, to delegate downloading the latest software to the end customer. It's not as sexy, but in all honesty we are going for functional.

The big piece of the puzzle that we are currently missing is how to implement this process that we have been discussing. From a high-level standpoint, this looks great, but what are the mechanisms, tools, and frameworks used to build out the delivery pipeline in Embedded DevOps? To answer this question, we need to discuss continuous integration and con-

tinuous deployment (CI/CD).

CI/CD for Embedded Systems

CI/CD is a process of automating building, testing, analyzing, and deploying software to end users. It's the tool that is used by developers, QA, and operations to improve collaboration between the various teams. CI/CD can provide development teams with a variety of value, such as

- Improving software quality
- Decreasing the time spent debugging
- Decreasing project costs
- Enhancing the ability to meet deadlines
- Simplifying the software deployment process

While CI/CD offers many benefits to embedded developers and teams, like any endeavor, there is an up-front cost in time and budget before the full benefits of CI/CD are achieved. In addition, teams will go through the learning curve to understand what tools and processes work best for them, and there will be growing pains. In my experience, while the extra up-front effort can seem painful or appear to slow things down, the effort is well worth it and has a great return on investment.

Implementing a CI/CD solution requires developers to use a CI/CD framework of one sort or another. There are several tools out there that can be used to facilitate the process. The tools that I most commonly see used are Jenkins and GitLab. No matter which CI/CD tool you pick, there is still a commonality to what is necessary to get them up and running. For example, nearly every CI/CD tool has the following components:

- Git repo integration
- Docker image(s) to host building, analyzing, testing, etc.
- Pipeline/job manager to facilitate building, analyzing, testing, etc.
- A job runner

A high-level overview of the components and their interactions can be seen in Figure 7-4. Let's walk through the CI/CD system to understand better how CI/CD works. First, the team has a Git repository that contains the product's source code. Next, developers branch the repositories' source code to develop a feature or fix a

bug locally on their development machine. In the local environment, the developer may use tools like Docker for a containerized build environment along with testing and debugging tools. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.[7] Containers make it very easy to pass development environments around and ensure that each team member is working with the same libraries and toolchains. Once the feature is ready, the developer will commit changes to their branch in the repository.

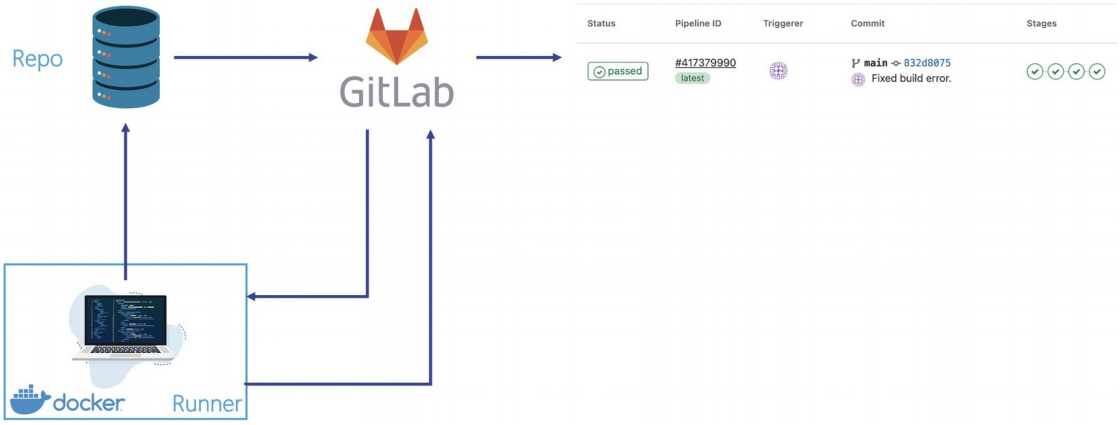

Figure 7-4 An overview of the components necessary in a CI/CD system for embedded systems (Source: Beningo Embedded Group)

The Git repository is linked to a CI/CD framework like Jenkins or GitLab. There are quite a few CI/CD frameworks available. I like Jenkins because it is the most widely used open source tool. GitLab is a commercial tool but offers a lot of great integrations and is free for small teams up to a certain number of "server minutes" per month. I've found that commercial tools are often easier to get up and running, so if you are new to CI/CD, I recommend using something like GitLab to get your feet wet.

The CI/CD framework will have a defined pipeline containing a series of automated steps performed whenever code is committed. Usually, teams will configure their CI/CD framework to run when any code on any branch is committed. The process is not just run on the master branch. Allowing the CI/CD process to run on any branch helps developers get feedback on their commits and features before merging them into an established and functional branch.

> **Best Practice**In 2010, Vincent Driessen wrote a blog post entitled "**A successful Git branching model**." The model has become very popular and

is worth checking out if you need an excellent Git process.

The job that runs in the pipeline first will ultimately depend upon how the pipeline was designed and configured. However, building the application image is usually the first job. Building the application image requires having all the tools necessary to build the application successfully. In general, CI/CD tools don't have the capacity to support building. This is where Docker comes in!

Docker is an open source platform for building, deploying, and managing containerized applications.[8] Docker allows a development team to install all their development tools and code into a controlled container that acts like a mini virtual machine. Docker manages resources efficiently, is lighter weight than virtual machines, and can be used to scale applications and/or build CI/CD pipelines.

The CI/CD tool can use a Docker image with the build toolchain already set up to verify that the code can build successfully. To build successfully, the CI/CD tool often needs a runner set up that can dispatch commands to build and run the Docker image. The runner is itself a containerized application that can communicate with the CI/CD framework. The runner is responsible for building the CI/CD pipeline images and running the various jobs associated with it. For small teams, the runner may exist on a single developer's laptop, but it's highly recommended that the runner be installed on a server either on-premises or in the cloud. This removes the dependency of the developer's laptop being on when someone needs to commit code.

When the runner has completed the pipeline stage, it will report the results to the CI/CD tool. The CI/CD tool may dispatch further commands with multiple pipeline stages. Once completed, the CI/CD tool will report the results of running the pipeline. If everything goes well, then the status will be set to pass. If something did not complete successfully, it would be set to fail. If the pipeline failed, developers could go in and review which job in the pipeline did not complete successfully. They are then responsible for fixing it.

Designing a CI/CD Pipeline

Designing a CI/CD pipeline often requires careful coordination between the developers, QA, and operations. A CI/CD pipeline is just a collection of executed jobs. However, these jobs specify essential aspects of the software application build, analysis, and deployment process. Each team is often responsible for managing a piece of the CI/CD pipeline. For example, developers manage the build jobs, unit tests, and basic analysis. QA manages the detailed system-level testing and analysis. Operations manages the server, frameworks, tools, and the deployment process.

The end goal is to verify and test the software in an automated fashion before releasing the software into the wild. The jobs executed in an Embedded DevOps process are completely product and company specific. Teams looking to get started with CI/CD often get overwhelmed because they look at the end goal, which can often look quite complicated. The best way to get started with CI/CD pipelines is to start simple and build additional capabilities into your process only as the previous capabilities have been mastered.

A simple, getting started pipeline will often perform the following jobs:

- Build the image(s)
- Statically analyze the source code
- Perform unit tests
- Deploy the application

When implemented in a tool like GitLab, the pipeline dashboard will look like Figure 7-5. You can see that the pipeline is broken up into three main pieces: a build stage, a test stage, and a deploy stage. I recommend not concerning yourself with a deployment job during a first deployment. Instead, start simple and then build additional capabilities into your pipeline.

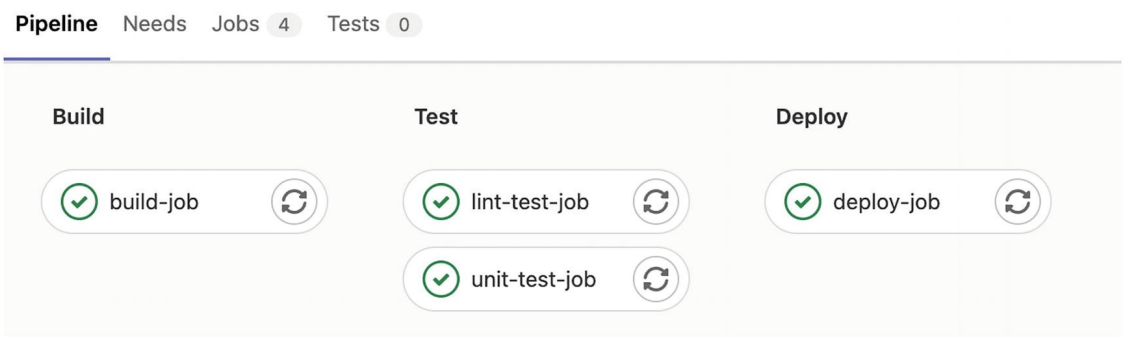

Figure 7-5 A CI/CD pipeline for first-time DevOps developers will contain four jobs for building, testing, and deploying the application

In general, before you start to implement a pipeline for your product, you will want to design what your pipeline looks like. While the four-job getting started pipeline is excellent, you'll most likely want to design what your final pipeline would look like before getting started. The goal would be to implement one job at a time as needed until the entire pipeline is built.

For embedded software developers, I often recommend a four-stage pipeline that includes

- Build
- Analyze
- Test
- Deploy

Each stage covers an essential aspect of the software development life cycle and can be seen in Figure 7-6.

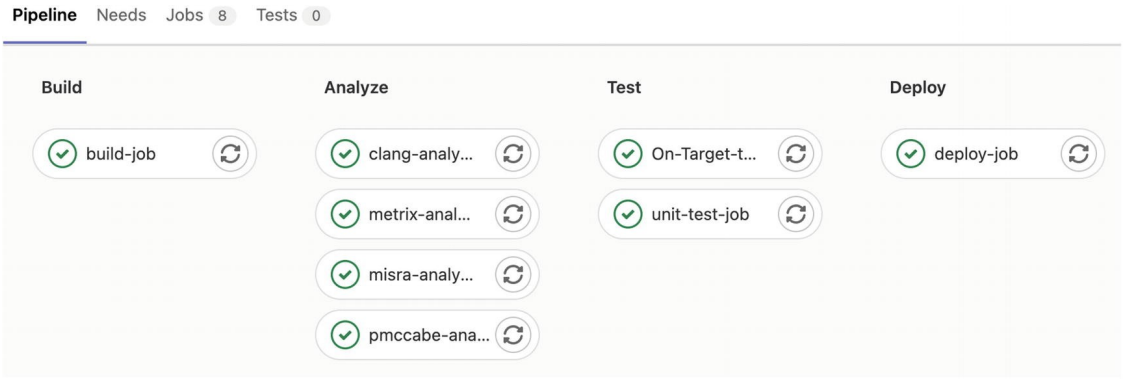

Figure 7-6 A more advanced embedded software pipeline adds code analysis and on-target testing

The first stage builds the application image(s). I like to have the first stage

build the software because if the code won't build, there is no need to run the other jobs. Developers first need to commit code that fully compiles without errors and warnings. A highly recommended industry best practice is to only commit code when it builds successfully with zero warnings. If there are warnings, then it means there is something in the code that the compiler is making an assumption about and the resulting code may not behave as we expect it to. I can't tell you how often I see code compile with warnings and developers just ignore it! A successful build is zero errors *and* zero warnings. When this is done successfully, then we can move forward to analyzing the code base. If the build completes with errors, the pipeline will halt.

> **Tip**Add the compiler option `-Werror` to turn all warnings into errors. You'll be less tempted to commit code with a warning.

The second stage analyzes the application code. It's common knowledge that whatever gets measured gets managed. The analysis phase is the perfect place to perform static code analysis through tools like clang, gather metrics through tools like Metrix++, and verify that the code meets the desired coding standards. The CI/CD pipeline can be configured to generate the results in a report that developers can then use to verify the code meets their quality expectations.

The third stage is the testing stage. There are at least two types of testing that we want to automate. First, we want to rerun our unit tests and ensure they all pass. Running unit tests will often involve running a tool like CppUTest along with gcov to check our test coverage. The second type of testing that we want to run is on-target testing. On-target testing can involve running unit and integration tests on the target, or it could be a system-level test where the latest build is deployed to the target and external tools are used to put the product through its paces.

Finally, the last stage deploys the embedded software to the customers. The deploy stage can be fully automated so that when the testing completes successfully, it is automagically pushed to the customer or such that it requires manual approval. How the final deployment is performed will dramatically vary based on the team, product, and comfort level with au-

tomated deployment.

Usually, the CI/CD pipeline is executed on code branches without the ability to deploy the software. When a branch successfully passes the tests, the code will go through a final inspection. If everything goes well, then a request will be made to merge the feature into the mainline. When the merge occurs, the CI/CD process will run again except this time the deploy job can be enabled to allow the new version to be pushed to the customer.

CI/CD provides a lot of value to embedded software teams and holds a lot of potential for the future. At first, the process can seem complicated, especially since embedded systems often require custom hardware. However, once the process is set up, it can help identify issues early, which saves time and money in the long run. In addition, the process can help keep developers accountable for the code they write, especially the level of quality that they write. A good CI/CD process starts with designing what capabilities your pipeline needs and then incrementally adding those capabilities to the pipeline.

Creating Your First CI/CD Pipeline

Talking about all these topics is great, but learning the high-level theory and what the potential benefits can be is one thing. Actually creating a pipeline successfully is something totally different. I've put together a simple tutorial in Appendix C that you can use to gain some hands-on experience creating a CI/CD pipeline for an STM32 microcontroller using GitLab. The tutorial covers a few basics to get you started. You may find though that you need additional assistance to design and build a CI/CD system for your team. If you run into issues, feel free to reach out to me and we can discuss how best to get you and your team up and running.

Is DevOps Right for You?

If you ask anyone who has bought into the DevOps buzz, the answer will be that you can't create a quality software product without DevOps. It's a resounding YES! Like most things in life though, the real answer may be

more in the gray.

Embedded DevOps, at least parts of the processes and principles, can benefit a lot of embedded teams both big and small. However, you do need to perform an analysis on the return on investment and the risk. Investing in DevOps does not mean you are going to achieve all the benefits. In fact, I've heard stories through the grape vine of teams that tried to implement DevOps only to fail miserably.

I have had success internal to Beningo Embedded Group and with several of my clients implementing Embedded DevOps. With some clients, we only implemented processes within the development team to improve their quality and visibility into the code for the rest of the team. In others, we integrated with the QA team to carefully coordinate testing. In others, it made sense to implement the whole set of processes and tools.

In some situations, it can make little sense to continuously deploy. Do you really want to manage complex firmware updates for entire fleets of devices? Does your customer who just bought your microwave product really want new features added daily? Honestly, that's a little bit terrifying from a user's perspective.

Is DevOps right for you? It's a loaded question and only one that you or a professional (like me) can really answer. In my experience, everyone can benefit from adopting some of the principles and processes, but only a few companies benefit from going all in.

Final Thoughts

Embedded DevOps has become a major buzzword in software circles. There are critical processes and principles included in Embedded DevOps that every team can benefit from. The trick is to not get caught up in the buzz. Selectively and carefully develop your Embedded DevOps processes. If you do it right, you can do it piecemeal and manage your investment and your return on investment to maximize benefits to both you and your customers. We've only scratched the surface of what can be done, but we will look to build on these concepts as we explore additional

processes that are crucial to the success of every embedded software team.

Action Items

To put this chapter's concepts into action, here are a few activities the reader can perform to start improving their Embedded DevOps:

- What can you do to improve how your developers, QA, and operations teams interact with each other?
- How might you modify the Embedded DevOps process that we examined in Figure 7-3? What would it take to implement a similar process in your own team?
- Schedule time to walk through Appendix C step by step if you haven't already done so. How easy or difficult is it to get a build pipeline up and running? What challenges did you encounter that you weren't expecting?
- What CI/CD tool is right for you? Jenkins? GitLab? Something else? Investigate the capabilities of the various CI/CD tools and select which tool best fits you and your team.
- What is your dream CI/CD pipeline? (Yes, who thought you'd ever be asked that question!) Then, spend 30 minutes designing the stages and the jobs in your dream pipeline.
  - What changes/improvements do you need to make to your DevOps processes over the next three months to move toward it? Then, map out a plan and start executing it.
- Common tools that are used for communication, like Slack, can be integrated into the CI/CD process. Take some time to research how to connect your CI/CD pipeline so that the team can be notified about successful and failed commits. This can help keep you as a developer accountable.
- What can you do to further develop your feedback pipeline? While we focused mostly on the delivery pipeline, what could you do to get good feedback about your customers' experience?

Footnotes

1 https://about.gitlab.com/topics/devops/

2 https://bit.ly/3aZJkXW

3 https://agilemanifesto.org/

4 https://percepio.com/devalert/

5 https://memfault.com/

6 The process was adapted from an existing Lucidchart template from lucid-chart.com.

7 www.docker.com/resources/what-container/

8 www.ibm.com/ae-en/cloud/learn/docker
