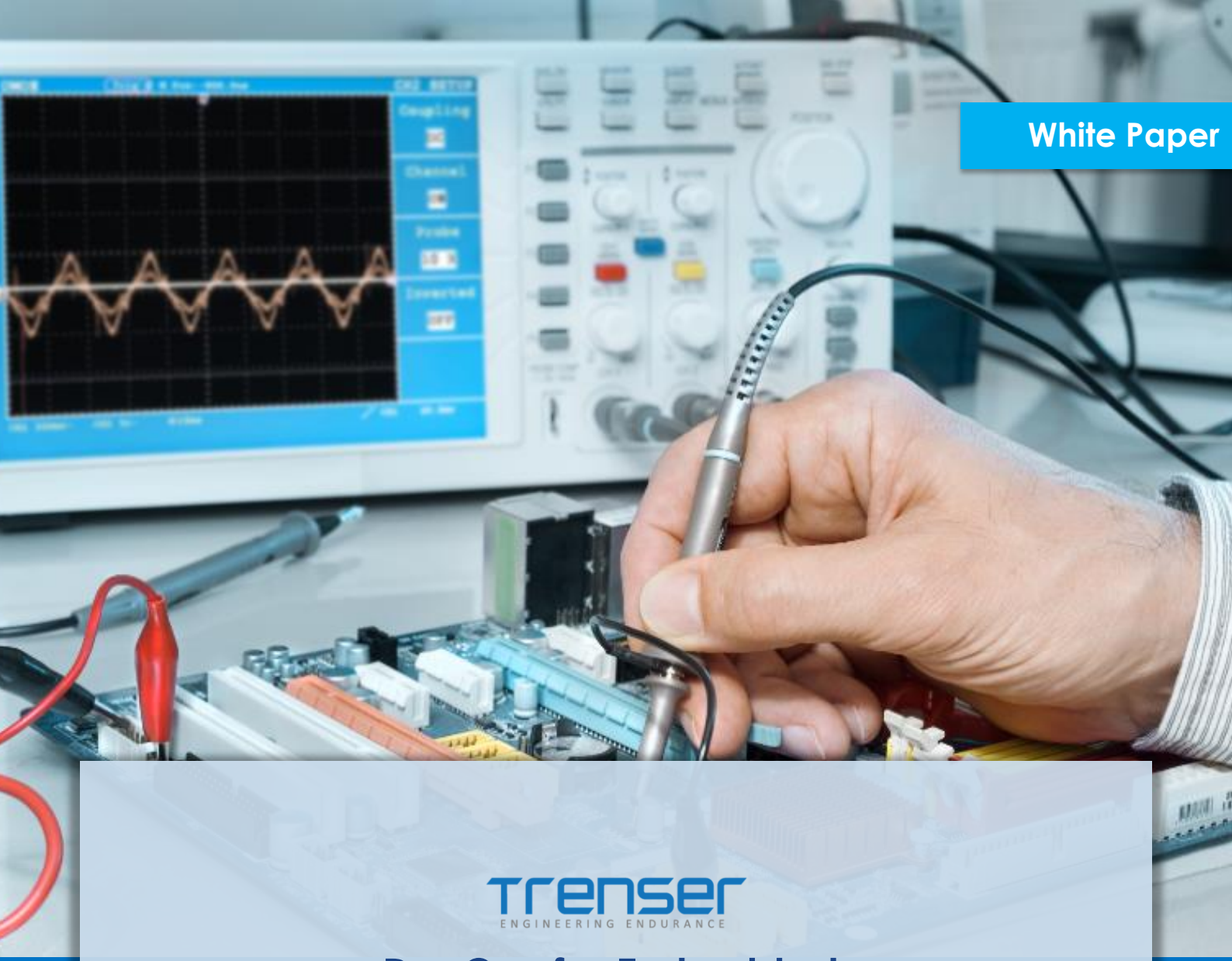

White Paper

trenser
ENGINEERING ENDURANCE

DevOps for Embedded:
Automating Embedded Software Development

By

Hari Menon
Anurag J K
Adhil Xavier

31 August 2021

INTRODUCTION

Traditionally, embedded software is often written for a chosen application-specific hardware platform because of the limited computing capabilities and memory availability. Hence embedded software implementation is mainly dependent on the target hardware platform. The speed of embedded software development is slower than other development domains like web and mobile applications. Though the embedded platforms have become more powerful, the development process has not changed much from the past. In this era of digitization and rapid change where IoT is a critical component, improving the pace of development will be greatly beneficial to reduce the time to market.

Embedded software is slowly becoming more independent from the hardware they reside in, as the computing platforms are becoming powerful. Embedded software development is unique, but the essential principles of DevOps can be applied with the right tools and the right approach. The concept of virtualization can significantly help the developers to extract the available infrastructure for software development. Software for configuration management can be used with automatic verification and validation. Configuration tools that support development and testing will help accelerate the product cycle time. This will boost the confidence level for the initial deployment of feature rollouts from production.

SCOPE

The scope of our study and experiment was to establish how best DevOps practices can be adopted for increasing the pace of embedded software development. The biggest constraint is that often testing must be performed in the target hardware along with the peripheral devices and testing equipment, locally at the developers' office. If the tests could be controlled and supervised over a network, avoiding the need for a physical presence, that would be a leap jump. We found that starting with version control and working towards automated testing, it's possible to continuously improve the embedded software delivery pipeline without the need for large-scale reorganizations.

THE CHALLENGING SCENARIO

DevOps in an IT enterprise will have a generic development and execution environment. Contrary to this, the embedded development environment is often varying. The execution environment is usually different than the one used for development. Hence, after building the software, the deployment to target devices will be specific for each hardware, and the test devices might be at remote locations. The other concerns that need to be addressed are different target hardware cross-compilation, cross-debugging, memory footprints, and security issues.

The DevOps environment is a continuous feedback loop which is easy when the development is on servers that are always online and under the user's control. On the other hand, embedded devices are likely to be remotely distributed, and they may not always be online. So, various such issues come into play when considering traditional DevOps for deployment.

The Ops part of DevOps for embedded is a challenge because, in a traditional DevOps environment for the cloud, the Ops is standard for developers running a website or developing an application with a cloud interface. However, when dealing with embedded, it is about devices in the field. The deployment (OTA) is not a standard process and varies for different manufacturers, and no common approach is available. Hence it needs assistance from the device manufacturers and cannot be a completely "closed-loop" situation, like in the case of a cloud-based enterprise product.

OUR APPROACH

Unlike DevOps in cloud applications, the approach for Embedded needs to be different. The key here is how far we can bring in automation. Most of the "Dev" part in DevOps can be reused by adjusting existing proven solutions. But when it comes to "Ops", things become a bit complicated. There is no ready-to-use solution available. Traditionally manual operation works best with a physical hardware environment. However, those are also not far from automating. The potential approach here is a device that can act as a broker in between the target hardware, which can perform the physical tasks in an automated way.

We are presenting an approach with a custom OS image that can be used to develop, build, and unit test the embedded software, which will be deployed to the target hardware via a sandbox server running in a single-board computer like Raspberry Pi or Beagle Bone. A generic web application in the sandbox server shall be used for communicating with the target hardware. The figure below shows the high-level concept.

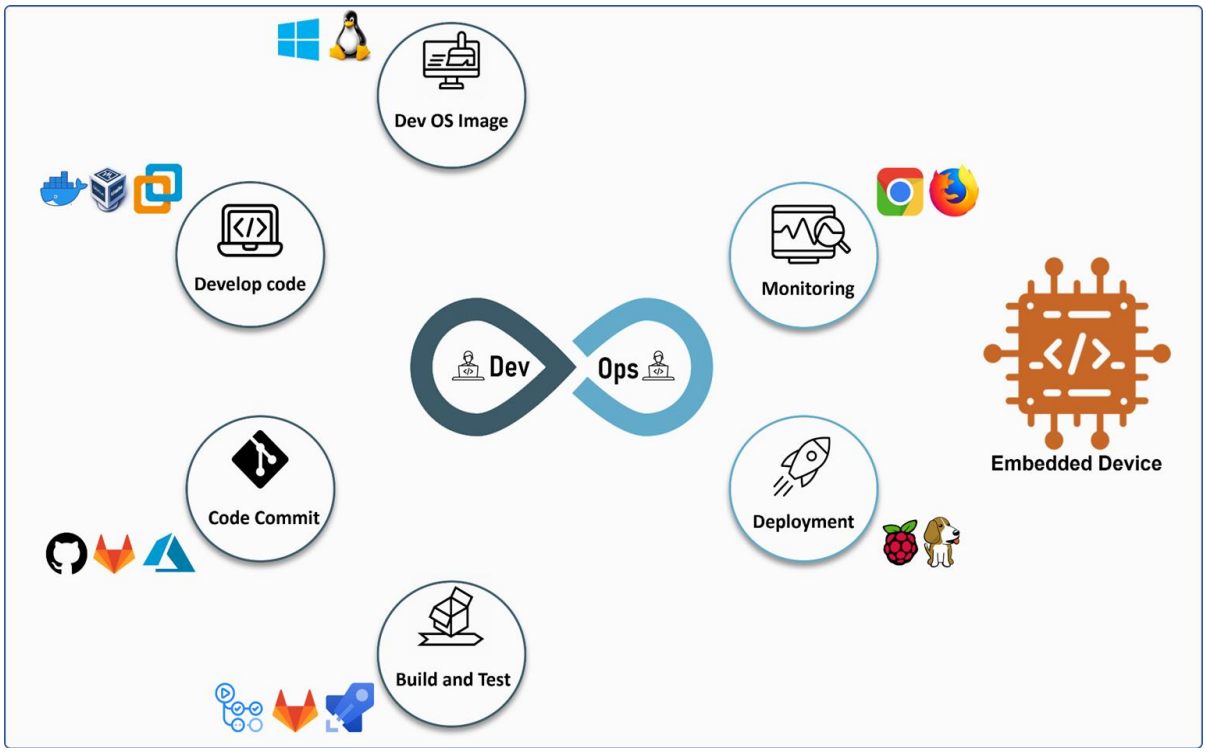

Figure I: Overview

The technologies can vary based on the target platform to the skill set of the developers. The best suited mix are:

Development Phase	Automation Approach/ Tools
Dev	Customized OS Image in Docker or as VM in Virtual box or VMWare.
Source Code Repository	GitHub, Gitlab or Azure
CI/CD Pipeline	GitHub Actions, Gitlab CI or Azure Pipelines
Deployment	Raspberry Pi or BeagleBone
Monitoring	NodeJS/ Python based web app viewed from Chrome or Firefox.

SOLUTION DETAILS

The figure below shows the step-by-step workflow of the proposed approach taken by us in the specific case we experimented.

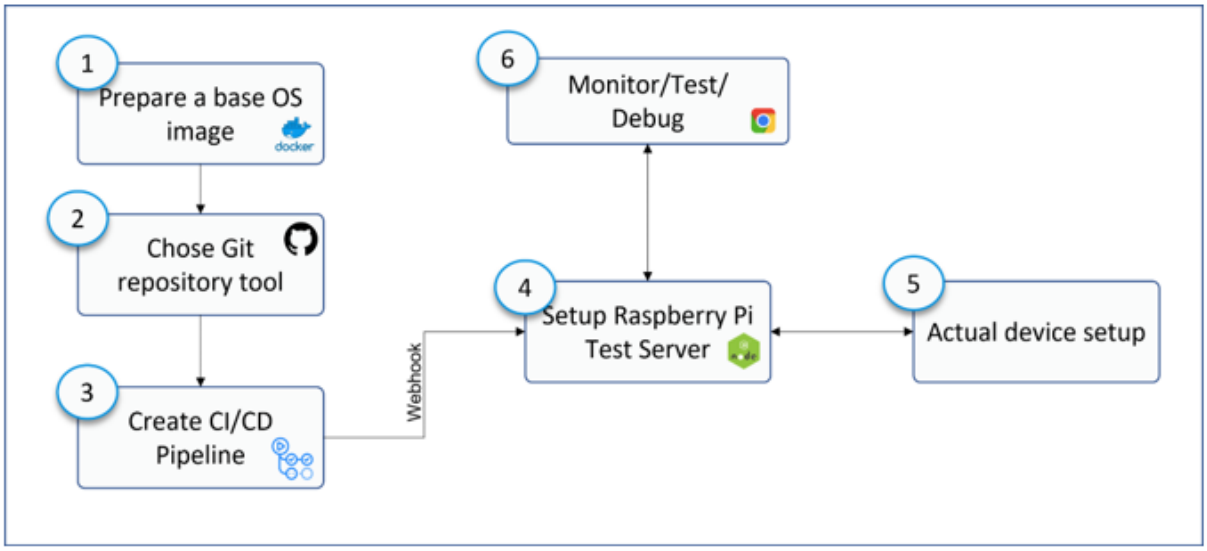

Figure II: Steps involved in the solution

1. Prepare Base Image:

First things, first. For every project, the initial challenge lies in setting up a working environment that includes a test setup and a development environment. When it comes to setting up a development environment for embedded, you need to set up the toolchain, compiler, linker, etc., before starting. The ideal solution to avoid a delay in starting is to prepare a common development environment that includes the base host image installed with dependent libraries, toolchains, compiler, etc. This base image can be converted to a virtual machine image or as a docker container. Here we chose Docker containers for ease of development.

2. Chose Git Repository Management Tool:

There is no confusion on choosing Git as the source code repository. But when selecting a management tool, we have options like GitHub, Gitlab, Azure, etc. All of them have advantages and disadvantages. To get started quickly, we think GitHub is the best choice.

3. Create CI/CD Pipeline:

Based on the repository management tool selected, the pipeline also differs. Here we will make use of GitHub Actions as our CI/CD pipeline. The pipeline shall contain the workflow jobs to pull off the development image, clone the repository, build and execute unit tests. The result of the pipeline shall be the executables for the target hardware.

4. Setup Webhook and Deployment:

In most cases, the real hardware may not have direct access to the internet by default. So, a broker device that will be running preferably on a Single Board

Computer like Raspberry Pi/ Arduino can be used as an interlink, which gets the build image from the pipeline and transfers the same locally to the target hardware. Considering the features and community support, we propose Raspberry Pi here. The Raspberry Pi and the pipeline must be connected through a webhook notification event. Whenever a new build is available at the cloud server, a notification will be received in the web application running in Raspberry Pi, and further deployment can be initiated.

5. Device Setup

Peripheral devices are interfaced to the sandbox via a communication bus (say, via I^2C bus) and various hardware interfaces required by the target hardware and test equipment involved. The target device also will need special software hooks to interact with the sandbox server to act and respond to the hardware level interactions via communication bus and various peripheral devices and test equipment. These *hooks* will also include redirecting the debug logs and hardware responses back to the sandbox server via the communication interface.

6. Monitor/ Test/ Debug:

A NodeJS or Python web server will act as a "bridge" for the user who wants to perform some actions on the actual hardware. The web application shall have the option to select the interfaces to which inputs to be provided, and once the input is processed, the output can be monitored as well. This application can be enhanced for real-time debugging and performing functional tests by inputting more sophisticated scripts with improved functionality.

A TYPICAL WORKING SETUP

In developing embedded systems, testing on the target device is mandatory to capture bugs that may show up only in the real hardware. Figure 3 shows a typical working setup. For functional verification, manual testing is done by developers when new features are implemented. This can be time-consuming and error-prone.

There should be a test branch that is properly named and maintained always to have the latest source code (main branch). After implementing a feature, the developer will push changes to the same branch. The compiling, building, and unit testing can be done as part of CI/CD pipeline. Unit testing is also performed at this stage. Developers need to update the scripts to perform unit tests. If the unit test fails, the developer will get notifications regarding the status and will stop other processes involved.

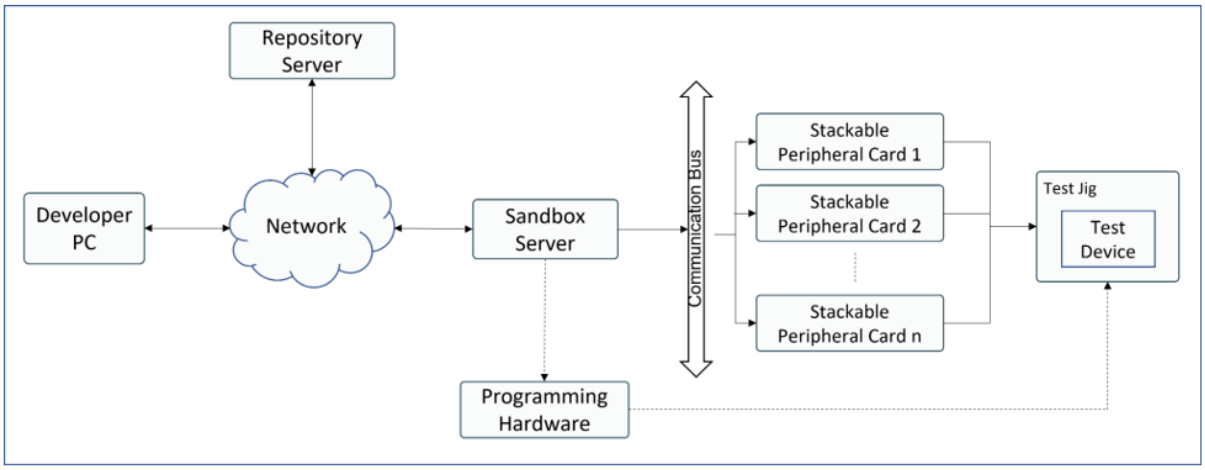

Figure III: Communication flow diagram

Sandbox server will get a webhook notification event whenever a new build is available from the pipeline task. Upon receiving notification, the Sandbox server will fetch the latest binary and make use of the programming hardware for flashing the binary to the test device or directly installing it on hardware.

Developers need to create a Flash script that contains the commands for flashing or transferring the executable to the test device. The programming hardware may be JTAG, serial, or any other interface. Sandbox server must be installed with proper drivers and software required for the programming hardware. In many cases, the programming hardware might not be necessary, with the availability of in-circuit programming capability of the platform, and hence a simple SSH file transfer will be all that is needed.

Once the binary file is flashed successfully, the total functionality can be tested. Functionality testing is done using an F-test script. This script will utilize special commands for interacting with the peripheral devices. Necessary drivers for supporting the peripherals shall be readily installed in the Sandbox server. Peripheral devices are connected to sandbox via a communication bus (say, via I2C bus) and can be used for (but not limited to) the following:

- GPIOs / Relays
- UART
- SPI
- I^2C
- CAN

If any *F-test* script or *flash script* fails, an email will be sent with required details, and corresponding logs of the failed script will help determine the cause of failure. The test status will be sent to the concerned personnel upon completion of the test as well.

Figure IV: Accessing the Raspberry Pi terminal through which the user can get access to the device if required.

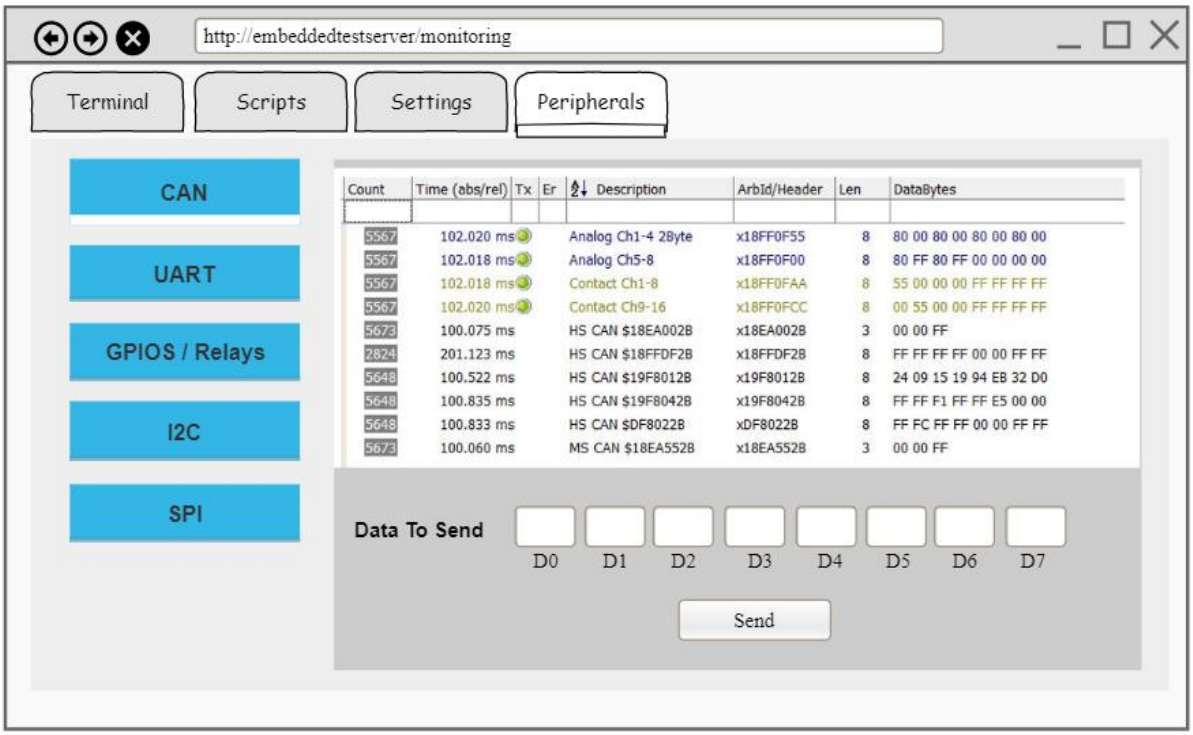

Figure V: A CAN peripheral page where user can view and send CAN messages in device

A web application will allow developers to opt for the required setup to perform the test. It will provide an interface where the developers can interact with the real hardware. They could even manage their scripts used for building (Build script), flashing (flash script), or even for the total functionality test (F-test script). The developers are given a chance to customize the test by opting for the peripherals required to perform the module-level testing.

With consistency in the integration process in place, teams are likely to commit code changes more frequently, which leads to better collaboration and software quality. Continuous Delivery picks up where continuous integration ends. Continuous Delivery is established by automating quality gates and building confidence in the code to move from development to production. Continuous Deployment starts with manual approvals and identifies how to establish and then automate the quality gates.

CONCLUSION

The practices and tools described in this paper can be effectively utilized to implement a DevOps-like system in embedded software development. Bare metal systems are, however, not considered. This approach can be adopted for moderately powerful computing platforms, especially platforms that run any RTOS or Embedded flavors of Linux. Our approach features development, testing, and deployment (until the product testing stage) in the Embedded platform. Creation of the base image shall be a one-time process during the start, and whenever a new developer joins the team, he can be productive from day one; thus, the developer can adapt to a new project with minimum effort and time.

As we are using an automated system for continuous testing, which is a time-consuming and repetitive task, developers will be relieved to work on more productive and fruitful tasks. Identifying, tracking, and documenting bugs now becomes seamless as proper logs and reports are generated.

The automated deployment process can be further extended to move your software from testing and production stages to field deployments by automating device-specific OTA processes. However, this needs several considerations of the installed environments and application criticality, as indicated in the challenges section. If implemented, it creates a repeatable deployment process across the entire software delivery cycle. This helps you release new features and applications more quickly and frequently, reducing the human interventions in the deployment.