Kria as a Service: Design Report

Adam Tozser - 3211770

Mira Gozbenko - 3187152

Jozsef Marton Kakas - 2918064

Adomas Puslys - 3209105

Yi Hu - 3151492

Table of Contents

1 Introduction

In this report, we will start with an introduction outlining foundational information about our project. Then we will discuss the initial system specification as we understood it at the beginning of the project. Afterwards, we will outline our development process and explore the decisions that we made. Then a detailed description of the final design and state of the product, including our testing and security strategies, will be given. To conclude, we'll reflect on what we achieved and what we learned from the project.

The introduction will begin by first establishing the necessary context for understanding our project. Then we will elaborate on the status quo prior to our project, which we wish to improve upon. Finally, we will briefly phrase our overarching goal.

1.1 Context

The hardware used in this project is the AMD Xilinx Kria KV260 single-board computer, referred to throughout the report as the Kria board, or just Kria. A Kria board combines a conventional ARM-based Linux system with FPGA-based programmable logic on the same platform. This means it can function as a regular embedded Linux computer while supporting hardware acceleration for computationally intensive tasks. This dual nature allows users to work very closely with FPGAs from Linux, which is an environment that is familiar for many people in computing. The programmable logic can be configured to implement custom hardware pipelines. This is useful for applications such as real-time machine learning, signal and image processing, and other tasks that require fast and efficient data processing and benefit from hardware level acceleration.

At the University of Twente, Kria boards are used in both educational and research settings. The current setup consists of 32 Kria boards in a server rack. The Kria boards are used in courses involving embedded systems and digital hardware, and are also used by researchers for experiments and development involving FPGA and embedded computing applications.

Since these boards are shared across multiple users, they change hands frequently. This creates an operational challenge: after use, a board may no longer be in a clean, predictable state. Software, settings, and other system-level changes made by one user may affect the next. In this context, provisioning means restoring a board to a clean baseline software image and applying the required configurations so it is ready for the next user. In addition, the Kria boards are frequently soft-bricked by students. The exact cause is unclear, but the issue has been widely reported both by our client and by students who have worked with the platform. For these reasons, resetting and provisioning the boards is a necessary part of maintaining the platform.

1.2 Previous provisioning process

Before the current implementation, provisioning a Kria board was still a largely manual process. When a board failed, became misconfigured, or had to be prepared for the next user,

the operator had to physically remove the microSD card from the board, connect it to another system (typically his laptop), flash the required image, reinsert the card, boot the board, and verify that the system started correctly. In the KaaS setup, ir. D.M. Abeln (Dorus Abeln) was responsible as operator and maintainer of the platform.

The existing provisioning tool was a bash script named Quartermaster. It already automated part of this process, but it still depended on physical SD-card handling. It wrote a base image to the SD card and then modified the filesystem by adding a board-specific configuration and boot files, such as a bootstrap script and a systemd service. Board-specific options, such as hostname, had to be manually specified for every device at every provisioning. After the board booted, this bootstrap process completed the remaining installation and configuration steps on the running system, for example, downloaded libraries and tools often used by students for assignments.

This approach was inconvenient even for a small number of boards, as each reset or provisioning action required manual physical intervention. In a setup with 32 Kria boards, this became especially inefficient. The process was time-consuming, error-prone, and did not scale, particularly because boards are shared between many users and can fail or require resetting regularly. Furthermore, the administrator wishes to have a system extensible beyond the current number of boards, which makes a manual provisioning workflow even less suitable long term.

The preceding provisioning process formed an operational bottleneck. This is the problem that the KaaS (“Kria as a Service”) project sets out to address.

1.3 Goal

The primary purpose of this project was to explore what is possible in the world of automatic Kria provisioning. Building a system that automates reflashing the Kria environment without physical intervention was important, but our client highlighted at our very first meeting that it is not clear how much of the process can be automated. In this report, we hope to show that our project delivers both a body of knowledge on automated Kria provisioning, and automated embedded linux provisioning in general, as well as a high-quality product suitable for future integration with other systems at the UT.

To clarify, for this project, our client and our supervisor were the same person.

2 System specification

In this section we will outline what we understood about the project at a high level in the beginning. We will first go over the initial requirements as they were presented by our client, discussing how they affected our approach to the task. Then we will discuss the milestones we agreed upon with the client and how we planned around them. Finally, we will elaborate on how responsibilities were divided at the start of the project.

2.1 Initial requirements

In this project, due to the exploratory component, it was impossible to perform a textbook initial requirements gathering step where the system would have been thoroughly described. This is due to the fact that the impossibility of a certain implementation path could reveal itself unexpectedly, as the knowledge of the Kria components and bootloader environment is specific to the extent that we started the project with none of the knowledge on the matter. The requirements available to us at the beginning of the project are outlined below

2.1.1 Functional requirements

- The system should have a Preboot Execution Environment (PXE) to enable network-based provisioning of the micro SD cards.
- The system must allow a Kria board to boot from the network and write a new image to its micro SD card without manual intervention.
- The system should have an effective method to manage and store custom micro SD card images for the KaaS platform.

2.1.2 Non-functional requirements

- The system should require minimal manual input from the user
- The system should emphasize correctness and reliability over feature count.
- The system should not interfere with other operations on the University's intranet
- The system should not cause unintentional data loss for the University's Kria users.
- This system should make it easy for administrators to upload new images.

2.1.3 Milestones

Despite what the word "milestone" may imply, these points were not meant to be implemented sequentially, rather, each milestone is a feature that should be developed to a correct and reliable state to be considered complete (as emphasized in the non-functional requirements).

- On-line provisioning through network boot
- The system should enable a better, more modular approach to config management instead of hard-coding it in a Bash script
- The system should enable assignment of provisioning different Krias with different images (without hardcoding the names)
- The system should expose an API for other systems to trigger provisioning

Early on, we determined that the risk of our project was the uncertainty of what is technologically feasible. Thus, we adopted an iterative development strategy that is outlined

in great detail in the “Process” section. Rather than working towards the milestones one by one, we developed the components that our research had determined to be feasible.

2.1.4 Initial model of the system

After our first meeting with the client, we had a mental model of the system, and we met to draw a diagram to make a first high-level overview of the system. Figure 2.1 shows the first high-level system sketch produced after the initial client meeting. We later redrew the sketch in Figma for legibility (Figure 2.2).

Our big picture idea was the following: The bootloader contacts a network resource, which is some server controlled by us, and decides what to do, if requested it pulls a minimal linux image for provisioning, and that image is used to run Quartermaster.

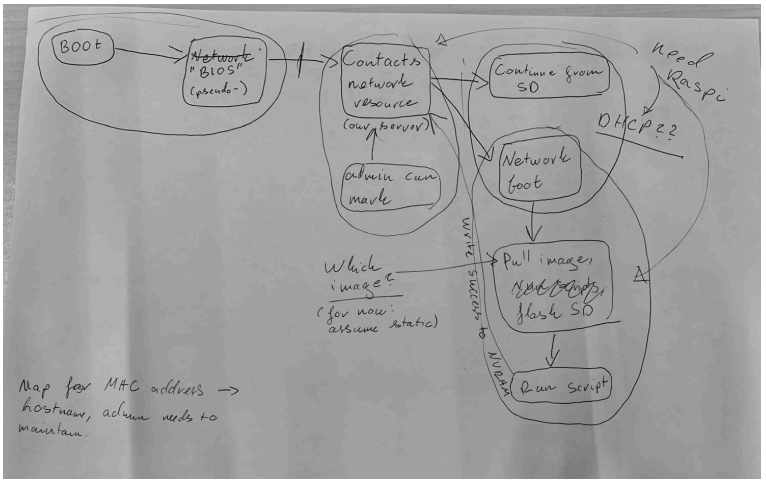


Figure 2.1: Initial hand-drawn system sketch from the first client meeting (photo by authors)

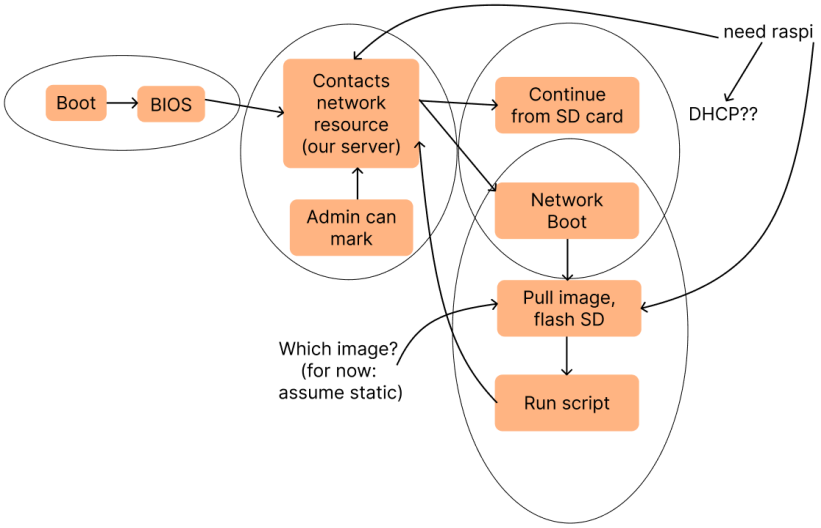


Figure 2.2: Redrawn version of Figure 2.1 created in Figma to improve legibility (authors’ illustration).

2.1.5 Primary Responsibilities

At the very start of the project, when the primary components of the system were already outlined, we distributed the tasks concerning separate parts of the system:

- *Mira & Adam:* U-boot partition responsibility
- *Jozsef:* Network boot path
- *Yi & Adomas:* Server side implementation and management

Tasks were distributed primarily based on which parts of the system could be worked on separately, and also based on individual work preferences.

3 Process

In this section we will recall our development process. First we will explain our collaboration strategy and the reasoning behind it. Then, we will outline the evolution of the project throughout four meetings with our client. In the end, we will enumerate task division in greater detail.

3.1 Collaboration

Due to some extracurricular commitments, our team agreed at the very start to manage our work around not blocking each other; this is sometimes referred to as “async work”. Thus, our collaboration strategy centered around splitting tasks and keeping track of progress between subteams.

The Discord server served as our communication channel. We did not use more complex management techniques like tickets or kanban boards, as the team of five was split into teams of roughly two people, it was not necessary to put so much effort into tracking what had been done, as the sub teams could keep track of themselves just fine.

We met online whenever it was necessary to sync progress, although the Discord was sufficient to keep everyone on the same page. We met in person when it was necessary to do something on a hardware level, or when we needed to discuss with the client, which was about every two weeks. We hosted our code on Gitlab, the repo is available in the attached materials.

3.2 Iterative Design

In total, we met with our client four times. We decided to explain our iterative development process by grouping the project into phases based on the meetings, as the meetings were often both checkpoints and inflection points.

3.2.1 Exploratory phase

This phase began with our first meeting with our client. We established the requirements and split up the work. Our main goal in this phase was to answer some fundamental questions that would determine the course of the rest of the project:

- Is it possible to access a network resource from u-boot?
- Is it possible to run scripts in u-boot with branching logic?
- Is it possible to save custom scripts persistently in the bootloader?
- How can we network boot from the bootloader?
- Is it possible to find or create a linux image that a) has all of the userspace utilities needed by quartermaster and b) is small enough to transfer through TFTP and fits into the Kria's RAM?
- Can we reliably deliver config files during early boot using TFTP?
- What is the minimal information the board needs from the server to make a boot decision?
- Can Quartermaster run inside a RAM-booted minimal Linux environment, and what dependencies does it require?
- Which parts of the old script are hard requirements vs assumptions tied to manual SD handling?
- What board-specific data must be injected offline vs deferred to first boot?
- How do we design safe defaults if config fetch fails to avoid accidental provisioning/wipe of the existing state?

Bootloader

For working with the bootloader, the very first step was to figure out how to access u-boot. Because this is an extremely minimalistic environment, options obvious to us, such as ssh, would not work. In what was a first for most of the team, we figured out how to use serial ports for input and output. In many ways it is simpler than ssh, but in other ways it is less reliable and less flexible.

With access to the bootloader environment, the next step was to ensure that the Kria could see the computer the TFTP server would be hosted on. To this end, a configuration must be performed on the host computer which depends on the operating system. In macOS we encountered an issue that connecting the Kria to the host computer directly via ethernet and setting up the LAN interface would prevent the host computer from accessing the wider Internet; this issue was resolved by indirectly connecting the host computer and the Kria by connecting them to the same router via ethernet. Similar issues did not arise with Linux and Windows systems.

Now it was time to test how transferring files through TFTP works on Krias. We would have preferred HTTP, as TFTP is a slow and unreliable protocol, but by default HTTP is turned off in the bootloader; the assumption behind this is that the bootloader should be as small as possible, and complex protocols take up more memory than simple ones. We made a note to return to this possibility in a later phase.

This was where we encountered something that would become a recurring problem during development: The Krias can be extremely particular about what they will and won't accept, and documentation is scarce. We were able to resolve most of our issues from existing forum answers, but later we will highlight some problems that we never ended up getting a good answer to.

We wrote a very simple TFTP server in python and used it to host a test file with a little bit of text. We had to tinker a little bit to get the Kria to see the TFTP server, but after about roughly one afternoon we successfully transferred a file, answering our question about whether or not network based file transfer was possible in u-boot.

The other big question regarding u-boot was whether or not it is possible to run a script with branching logic from the file that was pulled over the network. This was relatively easy to figure out: u-boot has a simple bash-like scripting language which can load the contents of any file as if they were environmental variables, and those variables could then be used in an if statement.

While we were working, we were considering what are possible alternative solutions if this network based setup doesn't work. The primary alternative we had in mind was connecting each and every single Kria to our server with USB cables (probably with a USB switch of some sort) and then using the serial port to pass information. This would have been superior to the status quo of manual provisioning, but would have been inferior to the network based setup. To name a few potential issues: serial ports are a bit awkward to work with, it would have made scaling to more Krias difficult, and the maximum number of units would be fundamentally limited by the available USB switches.

Up until this point, we were running commands manually, but since the whole point of the project is to automate provisioning we would need to find a way to store our instructions persistently in a way that the bootloader can access and run it. We had several options with various tradeoffs.

First, it is possible to store the entire program as a string in a specific environmental variable. However, this is mostly intended for two or three commands at most, and our program was about 40 lines long, thus, this would have made reading and writing it very difficult. Furthermore, by default, the bootloader is configured to disallow the flashing of environmental variables, so it would not even solve our issue of persistence - it would disappear after powering off the board. We noted that it is possible to reflash the bootloader, and returned to it in a later phase.

The second option was to create a "boot.scr" file that, when placed on a specific partition of the filesystem, would be read and executed by the bootloader. This was much better, since it was persistent, but the issue was that it relies on the integrity of the file system, the presence of which we believed naive to assume, given that the whole point of our system is to recover faulty Krias.

The third and final option was to bake the boot.scr into the bootloader, which would provide a resilient but also maintainable solution to the problem. However, at this point, we had no idea how to create and then reflash a custom bootloader.

In the end, we noted option three as a future nice-to-have, and ended up going with option two as a short term compromise that would let us test other parts of the system without actually changing the execution flow. Overall, we validated that the bootloader part of the project was possible, and had a prototype we could use as a template for further development as well as a list of potential improvements.

We attempted to perform a network boot as well, but ran into some issues that are elaborated upon in the next section.

Network boot

Most crucially we needed to establish if we can boot into a stateless operating system over the network. This step was fundamental to our initial design direction. To that end, we have tried an iterative approach: trying the most minimal solution first to validate the concept. The initial attempt consisted of brute-forcing the bootloader to load a full-sized Ubuntu image over the network, not least because our knowledge of the Kria boards' boot architecture was very limited in the first phase. We could not find a working combination of kernel, device tree, and rootfs, leading us to the question of our initial design direction. Then began a research phase, where instead of experimentation, we studied AMD’s documentation extensively. While reading the official manual, we have come across the intended ways of generating and shipping Linux-based products on Kria KV260 boards: Yocto, the general purpose embedded Linux build platform, and PetaLinux, AMD’s wrapper over Yocto. PetaLinux trades flexibility for a more guided and streamlined developer experience, thus PetaLinux was selected.

While PetaLinux seemed like the easier option, the process was not seamless. For example, the installation process is quite particular. Peta requires specific, and older, versions of Linux on the developer’s machine. Therefore, we have decided to work in virtual machines. Even so, there ended up being software dependencies that had not been listed in AMD’s guides. To simplify the life of any future maintainers, the exact installation steps have been documented for a fresh Ubuntu 22.04 LTS operating system, which we have found out through experimentation. Once installed, the toolkit guided us through the initial configuration process with a TUI. Quartermaster’s software dependencies could be selected in an easy interface. However the next step, the build process, brought its own problems again.

We ran into the issue of PetaLinux’s hardware demands during development. The first issue was when we tried to run it on a mid-range consumer grade laptop, on which the build was estimated to take about a week. We then tried to run it on a high-end desktop computer through WSL, but discovered that building a Linux kernel is extremely disk I/O intensive and WSL has a severe disk I/O bottleneck. The solution that ended up working for us was renting a high performance VPS from Hetzner. Both single core performance and core count are

important, as some parts of the build process are highly parallelized, and some parts are strictly single threaded. We only ever used PetaLinux on SSDs, but HDDs may run into I/O bottlenecks similar to WSL.

However, once the initial *image.ub* file has been compiled, it justified the tedious installation process. On the first attempt, the bundled kernel, device tree, and rootfs have booted without any issues.

To finalize this design direction we needed to establish that the live PetaLinux instance could flash to the SD card in the same slot. This function was verified with a simple *dd* command we ran over the serial console. We could safely proceed now.

Server side provisioning and Quartermaster

On the server side and provisioning side, the main uncertainties in this phase were closely connected. It was not yet clear what kind of network setup would be practical for early boot experiments, how a board should identify itself during that stage, and whether the existing Quartermaster workflow could later be adapted to consume dynamically obtained configuration instead of manually supplied input. In other words, it was not enough to know that a file could be transferred over the network. We also needed to understand whether that information could eventually drive a complete provisioning workflow in a controlled and board specific way.

During these early experiments, the Raspberry Pi became especially useful as a practical development component. It could act as a Linux host that was easy to reconfigure during development and could serve files for the early boot stage. These experiments also helped shape the board identification strategy. Since the system needed to support board specific behavior, the MAC address quickly emerged as the most practical identifier, because it is already available during early boot and can be used to select one configuration file per board without introducing separate bootloader logic for each device.

Quartermaster was still tied to the earlier manual workflow at this point. The main question was therefore whether it could remain part of the solution at all. Before changing it, we first needed to understand which parts of the script were essential and which assumptions only made sense in the old SD card based process.

3.2.2 Individual components phase

This phase comprises the periods after our second and third meeting with the client. Having established that essential technical components work, we set out to implement the system. The following questions were to be answered:

- Which controller actions must be public versus admin-only?
- How should the controller persist configurations in a way that survives restarts and stays compatible with TFTP filename (MAC)?

- What API identifiers do we expose (MAC and hostname) and why do we need both in practice?
- What is the minimum API surface needed for provisioning flow, and what can stay out of scope?
- How do we package and serve artifacts (images + tarballs) over HTTP in a reproducible deployment (containers/volumes)?

Bootloader

The big question about the bootloader in this phase was whether or not it was worth it to reflash it. We discussed with our client, and he requested that we explore how difficult it is and then produce a cost-benefit analysis.

Luckily for us, it turned out that PetaLinux is capable of building and exporting custom bootloaders. Thus we could reuse our VPS setup for the bootloader as well.

Flashing the bootloader opened up two possibilities for us: switching from TFTP to HTTP, and baking the script into the bootloader. Switching from TFTP for HTTP would have both simplified our server structure (by allowing us to serve everything from one API) and would have sped up the provisioning process, as TFTP is significantly slower than HTTP.

A significant amount of time was spent attempting to get HTTP to work. It is somewhat documented online and there are dedicated flags to set for it in the bootloader, but no matter what we tried the requests simply failed. What we could gather from network diagnostics is that the TCP packets were sent by the Kria, they were received by the server but for some reason the server just did not respond. We thoroughly investigated our networking setup and the server itself, but in the end we were running out of time. Thus we decided to cut our losses and point our fingers at a bug report we found which said that the TCP stack on the Kria is unreliable. In terms of the final product's quality it isn't a big loss, with good ethernet cables TFTP doesn't take more than a few minutes, the loss is more so on the architectural side.

As for baking the script into the bootloader, figuring out how to configure PetaLinux to do that was probably the most difficult part of the bootloader component. This took a lot of reading and guesswork, but in the end we figured it out. See the manual for details.

The next step was to get the built bootloader (in the form of BOOT.BIN) onto the Kria. Several interfaces are exposed for this, but we decided to go with the browser based recovery tool. Given that the process only has to be done once per Kria, and the possibility of catastrophic failures with an automated bootloader flashing script, we decided to keep this step manual. Again, see the manual for details.

Network boot

Without any remaining question marks, the rest of the design process progressed linearly. We discovered PetaLinux uses systemd as its init system, so a systemd service has been designed

to run Quartermaster. At this stage another design decision was made: We decided Quartermaster's script would be dynamically fetched from the Controller API. Since Quartermaster contains instructions unique to the current Ubuntu image, it will likely be modified in the future. However, rebuilding Quartermaster takes a long time, so fetching the provisioning scripts dynamically reduces iteration time significantly. Moreover, if a different script were to be required in the future, for example to accommodate an Arch linux based image alongside the current one, then it could be added easily.

Throughout the development of this component, memory issues were constantly coming up – understandable, given the nature of embedded systems, and dealing with large files in ramdisk. Two notable adjustments were made: The conversion to PetaLinux's initrd mode from initramfs, the latter of which, above a certain size, kept attempting to write to illegal sections of memory, and the modification of the systemd service to stream images instead of pulling it to the local ramdisk. Operating system images can be multiple gigabytes, over the size of the memory of an SBC like the Kria KV260. Therefore, instead of downloading the archive first, the systemd service simply passes the image URL to Quartermaster. Quartermaster then streams it to the SD card; more about this in the appropriate sections about Quartermaster.

Quartermaster

In this phase, the main development task for Quartermaster was to remove assumptions that only made sense in the earlier manual workflow. In its original form, the script expected a board index and a local image file, which fit a provisioning process prepared on another machine but did not fit a workflow in which a temporary Linux environment on the Kria would obtain configuration dynamically.

The first major change was therefore to replace the board index based interface with a hostname based one. This aligned Quartermaster with the rest of the new architecture, where board identity and image selection came from server side configuration rather than direct operator input.

The second major change was support for URL based image input alongside local file input. This was important because the Provisioning OS runs from RAM, while operating system images can be several gigabytes in size. By allowing Quartermaster to consume the image as a URL, the system could stream the image directly to the storage device instead of first downloading it into memory.

At the same time, Quartermaster still had to preserve its core functionality. It still needed to write the image, prepare the target storage, and seed the flashed system with the files required for later stages, including hostname related configuration and bootstrap assets. By the end of this phase, Quartermaster had moved from being a script that automated part of a manual provisioning process to being a component that could be invoked as part of the automated provisioning pipeline.

HTTP Server

After the initial boot decision is made in U-Boot (obtaining IP via DHCP and fetching the per-board config via TFTP), the provisioning environment must retrieve binary files, including multi-GB OS images. TFTP is unfit for this stage: it is UDP-based and intentionally minimal, which makes it suitable for small boot-time files, but inefficient and significantly slower than HTTP for large transfers.

This created two concrete needs on the server side. First, we needed a robust delivery of large provisioning artifacts (OS images and the Quartermaster tarball), which strongly favors HTTP over TFTP. Second, we needed a remote control channel to manage provisioning state and configuration without manual file edits on the server. Specifically, the workflow requires changing boot_flag (normal vs provisioning boot), and updating per-board parameters such as hostname and image_url. These updates are needed both for the provisioning flow itself (the board clears boot_flag back to normal after successful provisioning) and for controller operator actions (trigger provisioning, rename a board, switch a board to another base image).

TFTP cannot serve the control role. It is an unauthenticated file-transfer protocol in which the client requests a file by name and receives its contents. It does not support authentication request/response operations for controlled state changes. For these reasons, we kept TFTP only for the initial per-board configuration fetch from U-Boot, and implemented an HTTP server to handle both large file delivery and a small controller API for configuration/state changes. This also kept the system extensible: other components in the KaaS environment can set a board into provisioning mode when a failure is detected, and the board can clear the flag back to normal once provisioning completes.

We also investigated whether HTTP could be enabled directly inside U-Boot to remove TFTP entirely. In practice, we ran into issues in the U-Boot TCP/HTTP path that we could not resolve within the project timeframe. Since U-Boot only needs to fetch a tiny per-board configuration file, keeping TFTP for the initial step remained an acceptable trade-off, while HTTP was used for all large transfers and control actions.

3.2.3 Integration phase

This phase began after our final meeting with our client, where we confirmed that the individual components are up to standard. With three weeks left, it was time to test and integrate everything.

Until now, we had been working somewhat in silos. Slowly the individual components were validated when we helped to set up our teammates' environments. During this time, issues surfaced that were not present when we were setting up the environments on our own machines, for example, due to differences in operating systems. This phase also allowed us to double check the setup instructions and fill any missing gaps. Additionally, bugs in the system that weren't obvious before integration became apparent. For example, one issue with the bootloader that we realized in this phase was that it didn't have safe defaults: failing to

fetch the configuration would result in trying to boot from the network anyway. This did not come up when it was being tested as an individual component because the emphasis was almost exclusively on the bootloader code and not on a potential failure of the server.

Eventually, with two weeks left of the project, we reached our goal of performing a full “A to Z” test of the entire process, from powering the Kria on all the way to a working Linux machine. The process ran flawlessly, but with one limitation that it was performed on a single Kria board. To address this, and to reinforce our confidence in the system, we have performed another round of complete integration testing, but with two Kria boards.

During our last meeting, the client proposed the possibility of adding error reporting functionality. While it would extend the system in a helpful way, considering the amount of time remaining at this point, we made the strategic decision to omit this feature.

Unfortunately, we did not have the time to perform a full system test with all 32 Krias in a live environment, but we hope that the provided manuals and supporting materials allow this to happen one day. If the client so desires, several members of the team are available to help set up the system outside the confines of the design project. Additionally, as mentioned in an earlier section, the system was built to a specification such that it will be able to interface with other server administration infrastructure at the UT, such as “Surgeon”, which is an automated health checker, and “Scheduler”, where students sign up to receive Krias on an online interface, automating an additional step of the process. Some of these systems do not exist yet, and interactions with others were out of our scope during the implementation of this project, but in the future we would also be happy to help with this as well.

3.3 Contributions Overview

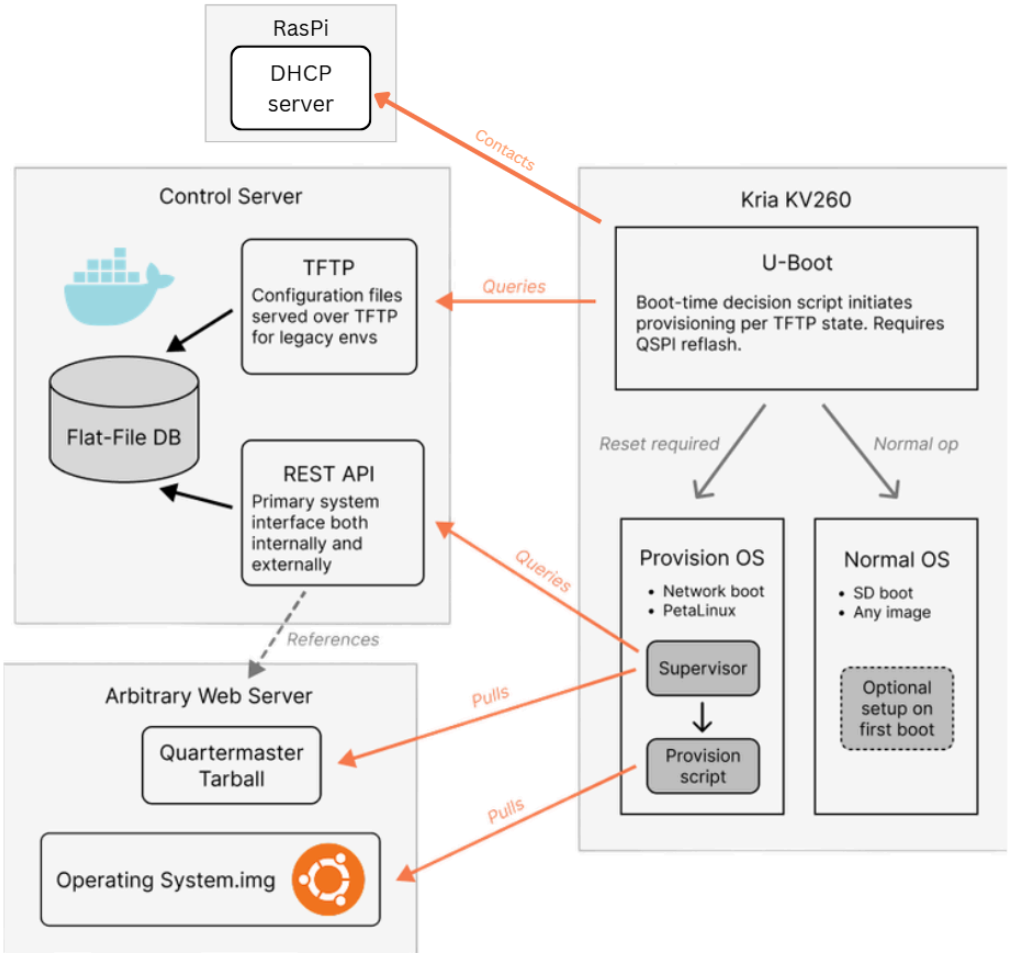
- *Adam:* U-boot work, including bootloader reflashing, custom bootloader compilation, and the boot decision script, final integration, participation in the report
- *Mira:* U-boot work, including custom booting script creation, bootloader reflashing, contact person role, participation in the report, poster creation
- *Jozsef:* Custom provisioning image generation, quartermaster supervisor process, including OS image streaming, final integration, participation in the report, poster creation
- *Yi:* TFTP server setup, DHCP related network integration, coordination with LISA, Quartermaster modification for the automated provisioning flow, participation in the report
- *Adomas:* HTTP server setup, API integration, coordination with LISA, participation in the report, poster creation

4 Design

Now that the overall development process has been recalled, with the various problems that were solved and possibilities that were explored, it is time to concretely describe the final state of our product. We will begin with a detailed description of all of our systems. Then, we will describe our testing strategy and the limitations of testing with our project. Finally, we will analyze the risks associated with our project and explain our security strategy to mitigate said risks.

4.1 Final System Description

4.1.1 High-Level System Overview



The diagram above illustrates the existing components of the system and their interaction

REST API triggers the Kria reset. The provisioning process starts as the Kria enters the U-boot environment. It acquires the IP from our DHCP server, and pulls the config file defining the further process. From here, the process is either directed to the SD card

(functional OS case), or the network boot is initiated by pulling the minimal PETA linux environment. The PETA Linux environment then pulls all the necessary dependencies of the final OS, Quartermaster, the image from the remote server, and installs the final Operating System.

4.1.2 Bootloader

Each Kria is flashed with a custom bootloader that contains a script with our system's logic. As soon as the Kria module powers on and enters the U-Boot environment, the system performs a few necessary setup steps and then runs our custom script. First, the script reaches out to the university DHCP server to get an IP address for the Kria. Once done, it queries a pre-configured TFTP server, whose address is hardcoded into the bootloader's logic, to retrieve its specific device name and configuration using its MAC address.

With its identity established, the branching part of the script can begin. The script checks a flag in the received configuration file to determine what it should do. This creates two distinct operational paths:

- The Standard Boot: If the server confirms the system is in a "valid bootable state," (or the configuration file failed to be retrieved) the bootloader immediately hands over control to the existing OS residing on the SD card. In case of network failure, this path is chosen to avoid accidental erasure of the system.

- The Reflashing Path: If the server marks the unit as "needs reflashing," the bootloader switches into a recovery state. It pulls a minimal Linux image that contains only what is absolutely necessary for provisioning via TFTP. Once the download is complete, the system boots into the minimal image as a ramdisk, and the system begins the next phase of provisioning.

Because the bootloader is built to be universal, the same binary can be deployed across an entire fleet of Kria devices. This is preferable, as the build process for the bootloader takes around thirty minutes, even on high-end consumer hardware. The manual describing the reflashing of the bootloader can be found in the section "Deployment Instructions" of this report in the Appendix, as the user will need a new version containing the IP address of the server they wish to use.

4.1.3 Network Boot Stage

If the Bootloader stage decides the device requires a fresh installation, the Kria will boot into the Provisioning OS. The "Provisioning OS" is our custom-made minimal Linux distribution designed to facilitate the SD card flashing process. The Linux image has been generated with PetaLinux, AMD's wrapper toolkit for *"easier"* Linux product deployment compared to a raw Yocto pipeline. PetaLinux was selected for its flexibility in generating lightweight images, a

necessity with TFTP-based network boot. After adding the dependencies of Quartermaster, the SD card provisioning tool, the compressed image was still below 100MB.

The boot process of the Provisioning OS is similar to other Linux-based operating systems. First, the kernel is initialized which sets up low-level resources like the memory. Then the root filesystem, based on initrd, is unpacked into memory. Importantly, the system uses the older initrd approach, as opposed to initramfs, since the latter has memory limit issues in PetaLinux. Finally, the init system takes over and a systemd service is started with the following steps:

1. Read local MAC address from */sys/class/net/end0/address*
2. Query Controller API for expected state at GET {API_BASE}/api/kria/mac/{MAC}
3. Fetch a packaged Quartermaster instance and extract to a temporary directory
4. Pass the target OS image to Quartermaster as a URL and start the provisioning process *(See Quartermaster section for more details)*
5. Reset boot flag so the bootloader will select normal operation after a reboot
6. Reboot the system (and boot into the freshly provisioned OS)

Polling was implemented in Step 2 to ensure the script only begins after a working network connection has been established. The Kria board obtains its IP from a DHCP server, similarly to earlier stages.

4.1.4 TFTP Server

For the early boot stage, a Raspberry Pi was configured as the TFTP server used by the Kria boards in U-Boot. Its role was to serve the small board specific configuration files required during boot, before the later provisioning stages could begin. This kept the early network stage simple and compatible with the limitations of the bootloader environment.

The TFTP server was not intended to support the entire provisioning workflow. Instead, it was deliberately limited to the initial configuration retrieval step. This matched the strengths of TFTP during early boot while avoiding the protocol limitations that would become more problematic for larger artifacts or richer control operations.

The TFTP server stores one configuration file for each Kria board. The filename of each file matches the MAC address of the corresponding board, which provides a simple and deterministic way to deliver board specific configuration during the earliest stage of boot. This includes information such as the board identity and the flag that determines whether the board should continue with standard boot or enter the provisioning path.

This arrangement allows the bootloader logic itself to remain generic across devices. Instead of maintaining separate boot logic per board, the server side configuration determines how an individual board behaves once the correct configuration file has been retrieved. In practice,

this makes configuration management more flexible and better aligned with the project goal of reducing hardcoded board specific behavior.

4.1.5 HTTP Server & Controller API

The final system includes a FastAPI-based HTTP server with two responsibilities: serving provisioning artifacts over HTTP and exposing a controller API for per-board configuration and provisioning state. The controller maintains a configuration object per board with exactly one hostname, one mac, one boot_flag (default 0), and one image_url. In addition, the API returns a single global provision_tarball_url (configured via environment variable) as part of every Kria object

Configuration persistence is implemented using per-board config files on disk. Each board has exactly one config file stored under the configurations directory. The filename is the board MAC address in colon format (aa:bb:cc:dd:ee:ff) to match the bootloader/TFTP naming convention, while the API uses the MAC without colons (aabbccddeeff) as its identifier in URL paths. This mapping keeps the API stable and URL-safe while remaining compatible with the boot-stage TFTP lookup. Updates are written atomically to avoid partial writes and corrupted configuration files during state changes.

Public controller endpoints provide read access and a restricted reset operation. The public API exposes *GET /api/kria* endpoint, which returns a list of Kria objects, *GET /api/kria/mac/{mac}* and *GET /api/kria/{hostname}* endpoints for looking up Kria objects identified by MAC address or hostname, and reset *PATCH /api/kria/mac/{mac}/reset* and *PATCH /api/kria/hostname/{hostname}/reset* endpoints. Administrative CRUD endpoints live under /api/admin/kria. We support both MAC and hostname lookups because the MAC is the canonical identifier during boot and for TFTP mapping, while hostnames match the labels used for Krias (Kria*XX*), making it a more user-friendly alternative. The public reset endpoint is intentionally limited to clearing provisioning state only by forcing boot_flag=0; public endpoints cannot set boot_flag=1 or modify hostname/image_url. This supports the provisioning workflow where the board (or other components such as Surgeon) can safely return a board to normal boot without being able to force provisioning or change what image is installed.

Administrative endpoints (*POST /api/admin/kria* and *PATCH/DELETE /api/admin/kria/{mac}*) provide full CRUD operations for board configurations and require a bearer token. Administrators can create new Kria objects, update hostname, image_url, and boot_flag, and delete a board configuration. The server enforces uniqueness of hostname and MAC address so each board can be addressed unambiguously. Administrative operations are the only way to set boot_flag=1 (enter provisioning mode) and to change configuration fields beyond the reset operation.

For deployment, the server is containerized and can be started reproducibly with Docker Compose. The API container mounts the configuration directory read-write and mounts the artifacts directory read-only (so served images/tarballs are not modified by the service). This

setup allows for easy redeployment of controller without losing state, as configurations are persisted on the host volume.

Finally, the HTTP server exposes a static file route to serve provisioning artifacts (OS images and provisioning tarballs) from a configured directory. This is used by the provisioning OS to download multi-GB images reliably over TCP. A health endpoint is also exposed to allow operators to quickly verify the service is set up

4.1.6 Quartermaster

Quartermaster originally supported a more manual provisioning workflow. In the earlier version, it expected a target device, a local image path, and a board index. This made it suitable for provisioning from a separate environment, but less suitable for integration into an automated network based pipeline running directly on the Kria board.

To make Quartermaster fit the final architecture, it was modified to accept a hostname directly instead of a board index. In addition, the newer version supports either a local image path or an image URL. This made it possible to invoke Quartermaster using provisioning information obtained dynamically during the automated workflow, instead of relying on fixed local assumptions known in advance.

This change was important for the provisioning flow. After the intermediate Linux environment determines the correct hostname and image source for a board, Quartermaster can now be executed directly with those values. As a result, it became much easier to integrate into the end to end automated provisioning pipeline than the original version.

4.1.7 Custom DHCP Server

One of the main design questions in the network boot stage was how the Kria boards would obtain the information needed to continue booting in a controlled and board specific way. A fully project managed DHCP setup was considered during development, since this would have given complete control over early network bootstrapping. However, after coordination with LISA, this was not possible in the university environment, especially in combination with an externally managed DHCP server.

The final solution therefore integrated with the existing university network instead of replacing it. LISA indicated that additional DHCP options could be configured for selected MAC addresses. This made it possible to keep the university DHCP infrastructure in place while still directing the Kria boards to the project’s own Raspberry Pi based TFTP server. In this way, the final design remained compatible with the actual deployment environment while still supporting automated board specific boot behavior.

4.2 Testing

This section describes how we validated the system. We first define the test approach and scope, then list the executed tests and outcomes per component, and finally state what we were not able to validate within the project constraints.

4.2.1 Component testing

Network boot

Test	**Expectation**	**Result**
Provisioning OS boot health	The Kria board is able to boot into the Provisioning OS over TFTP, and a functioning shell is presented over serial.	**Successful:** Board booted and a working bash prompt appear.
Systemd build recipe	The systemd service is included in the Provisioning OS. provision-startup is acknowledged by systemd, and the shell script is included in /usr/bin.	**Successful:** */usr/bin/provision-startup.sh* and */lib/systemd/system/provision-startup.service* are both valid files.
Controller API is reachable	Using the system shell of the Provisioning OS, the controller API's HTTP resources are available.	**Successful:** GET /api/kria/mac/<mac> returns a well formed request.
DHCP lease is obtained	When put in a network with a well-functioning DHCP server, the Provisioning OS obtains a valid IP address.	**Successful:** *ip address* (command) returns an IP on the expected subnet.
Quartermaster runs in isolation	When manually provided the preconditions (such as controller API state, Quartermaster assets), Quartermaster successfully flashes the SD card in the Kria board	**Successful:** Quartermaster exits cleanly (Quartermaster's testing is out of scope for this section)

Boot flag is reset on clean termination	Once Quartermaster has exited, the provision startup script contacts the Controller API to reset the boot_flag to 0.	**Successful:** TFTP config file confirms boot_flag is reset.
Provision startup script reboots	The system is rebooted once Quartermaster exited and the boot flag has been reset.	**Successful:** Logs corresponding to a successful system reboot are observed over the serial console.
Systemd service starts automatically on startup	Without any actions by an administrator, the provision-startup systemd service should start automatically.	**Successful:** State is active as reported by *systemctl status provision-startup*

HTTP Server

Test	**Expectation**	**Result**
OS images and provisioning artifacts are downloadable over HTTP	GET /files/<name> returns the correct artifact and completes transfer reliably for large files	**Successful:** Multi-GB OS image download succeeded; tarball download succeeded
Path safety, protection for file routes	Only files inside the configured files directly can be served.	**Successful:** Filename validation prevents invalid paths.
Health endpoint GET /health	Returns {“ok”: true}	**Successful:** Returned {“ok”: true}
Static file delivery endpoints GET /files/<image>.img.xz GET /files/quartermaster-main.tar.gz	Downloads succeed; usable by provisioning stage	**Successful:** downloads succeed; used as inputs for provisioning flow.
Public Kria read endpoints GET /api/kria GET /api/kria/mac/<mac> GET /api/kria/hostname/<hostname>	Returns Kria objects derived from /srv/tftp/configs/<mac> with provision_tarball_url injected from env	**Successful:** Returned correct objects and values for tested entries

Public reset endpoints PATCH /api/kria/mac/<mac>/reset PATCH /api/kria/hostname/<hostname>/reset	Only clears provisioning state by forcing boot_flag=0.	**Successful:** config file updated and boot_flag set to 0.
Admin CRUD endpoints POST /api/admin/kria PATCH /api/admin/kria/<mac> DELETE /api/admin/kria/<mac>	Create/update/delete per-board config files; reject hostname duplicates; validate hostname format	**Successful:** worked as expected in the test setup
Auth enforcement Call admin endpoints without/with wrong bearer token	Reject with 401/403	**Successful:** Rejected correctly
Persistance After PATCH/reset/admin update, inspect /srv/tftp/configs/<mac> contents	Changes persist on disk and after subsequent reads	**Successful:** persisted correctly (verified by reading config files after calls)

Bootloader

Test	**Expectation**	**Result**
Standard boot path (boot_flag = 0)	The bootloader retrieves the config with boot_flag 0, interprets the flag, and boots from the SD card.	**Successful:** Boots from SD card
Provisioning boot path (boot_flag = 1)	The bootloader retrieves the config with boot_flag 1, the bootloader fetches the minimal provisioning image via TFTP, loads it into a ramdisk, and boots.	**Successful:** Boots into provisioning image
Safe default on network failure	While the Kria is disconnected from ethernet, the bootloader must fall back to standard boot	**Successful:** Fell back to SD card.

	from the SD card rather than attempting provisioning.	
Safe default on missing or unreadable config	When the Kria cannot fetch a config file for whatever reason, the bootloader must fall back to standard boot rather than entering the provisioning path.	**Successful:** Fell back to SD card.
Universal binary with MAC-based config routing	The same BOOT.BIN flashed to two different Kria boards should result in identical behavior, but they should retrieve their own config and only set their own boot_flag.	**Successful:** Identical behavior except for expected differences.

TFTP server

Test	**Expectation**	**Result**
TFTP server reachability from U-Boot	Kria can contact the Raspberry Pi based TFTP server during early boot	**Successful:** Kria Board could reach the TFTP server during U-Boot
Board specific config retrieval	Kria retrieves the correct configuration file for its identity	**Successful:** Board specific config file could be retrieved
MAC address based mapping	Configuration file selection matches the board MAC address	**Successful:** MAC based mapping worked as intended for tested board entries
Config import into bootloader environment	Retrieved config can be imported and used by bootloader logic	**Successful:** Retrieved values were usable for boot decision logic

Failure behavior on missing config	System should not enter destructive provisioning by mistake	**Successful:** Fallback behavior remained standard boot in the tested logic path

Quartermaster

Test	**Expectation**	**Result**
Modified interface validation	Quartermaster accepts --hostname and exactly one of --image or --image-url, and no longer depends on a board index	**Successful:** Modified interface behaved as intended
Local image provisioning path	Quartermaster can provision correctly when given a local image file	**Successful:** Local image input worked correctly
URL based provisioning path	Quartermaster can provision correctly when given an image URL and stream the image directly to the target device	**Successful:** URL based image input worked correctly
Invocation from the PetaLinux provisioning environment	Quartermaster can be fetched and started from the ramdisk based provisioning environment used in the automated workflow	**Successful:** Quartermaster could be invoked correctly from the PetaLinux provisioning environment
Post write storage preparation	After writing the image, Quartermaster correctly performs the expected storage preparation steps needed for the flashed system	**Successful:** Provisioning continued correctly and the resulting prepared system state matched the expected workflow
System seeding on the flashed image	Quartermaster writes the configuration and bootstrap	**Successful:** Bootstrap files, SSH related files

	assets needed by later stages of the system	and other required assets were written to the prepared system

4.2.2 Integration testing

A-Z Provisioning test

1. Set boot_flag=1 for a board (admin)
2. Reboot board, U-Boot fetches config, enters provisioning, boots minimal Linux
3. Provisioning environment downloads OS image via HTTP
4. Quartermaster flashes SD, injects config, reboots
5. New OS boots, bootstrap runs once, provisioning completes
6. Board clears flag back to normal (boot_flag=0) via controller API

4.2.3 Testing Limitations

- Limited hardware coverage: we tested on a small subset of boards, not across the full 32-boards
- We did not run repeated provisioning cycles over days/weeks to measure failure rate or there are cases when human intervention is needed.
- No full end-to-end deployment: we validated the workflow in our project setup, but we didn’t complete a full university-network integration (final DHCP option setup, full rack integration, testing with the operator)
- Security testing limitations: we used Bearer-token protection and least-privilege endpoints, but we did not perform formal pen-testing, traffic interception tests, or TLS/HTTP deployment tests.
- Some outcomes were confirmed by manual checks (boot success, correct hostname, service enabled), rather than automated tests.

5 Risk Analysis & Security

5.1 Bootloader

Under some circumstances (typically when using specific browsers on specific operating systems), it is possible for the bootloader to get corrupted when overwriting it. The flashing tool can reliably detect this failure with checksums, however, the corrupted bootloader must still be either replaced with a working one or the other working version must be selected.

If the system cannot acquire the configuration file, or the configuration file is corrupted, it defaults to standard boot to avoid wiping systems unintentionally.

Krias have memory for two different bootloaders, with one being marked as the one to use for boot. This allows recovery in case either something is wrong with the new bootloader or because something went wrong during flashing.

5.2 TFTP

5.2.1 Dependence on Network Configuration

One practical risk of the TFTP stage is its dependence on correct surrounding network configuration. In the final setup, the project does not use an independently managed DHCP server. Instead, the TFTP stage depends on the correct DHCP options being configured through the university infrastructure and on correct MAC address based mapping for the Kria boards. If this configuration is incorrect, a board may fail to retrieve the correct configuration file and may therefore fail before the later provisioning stages are reached.

5.2.2 Protocol Limitations

TFTP also has limitations at the protocol level. Since it operates over UDP and follows a simple request and acknowledgement model, it is generally less efficient and less robust for larger transfers than TCP based protocols such as HTTP. Packet loss or network instability can lead to retransmissions, delays, or failed transfers more easily than in later stages that rely on HTTP. In addition, TFTP provides no built in authentication or encryption, which makes it unsuitable for sensitive control operations or for use outside a trusted internal environment.

5.3 HTTP Server

The HTTP server is deployed inside the University of Twente IoT network and is not intended to be reachable from the public internet. This strongly reduces the attack surface compared to an internet-facing service. As a result, the risk analysis is mainly concerned with misuse from within the local network, accidental misconfiguration, and unauthorized modification of provisioning state, rather than with large-scale external attacks.

The HTTP server has two main responsibilities: serving provisioning artifacts such as images and tarballs, and exposing an API for reading and modifying per-board provisioning state. These two responsibilities have different risk profiles. Public file delivery is relatively low risk, whereas unauthorized state changes are more important because they directly affect board behavior.

5.3.1 Public file delivery risk

The file server exposes provisioning artifacts such as operating system images and provisioning tarballs over HTTP. Within the UT IoT network, the main risk here is unauthorized downloading of these files by another device on the same network. In this project such risk is limited. The files do not contain private user data and are intended for use by the Kria provisioning workflow. Downloading an image without authorization would not by itself let an attacker modify board state or take control of the system. For that reason, public read access to provisioning files is acceptable in this environment. The main impact would be bandwidth usage, not compromise of the controller state, which is why protecting file downloads was treated as lower priority than protecting state-changing API operations.

5.3.2 API state-change risks

The more important risk is unauthorized modification of Kria configuration. If an attacker can change values such as boot_flag, hostname, or image_url, they can directly affect how a board boots and which image it provisions. This could disrupt normal operation, trigger unnecessary reprovisioning, or cause a board to boot into an incorrect environment. For this reason, state-changing administrative operations are protected with a Bearer token.

5.3.3 Public reset endpoint

One exception in the current design is the public reset operation. As implemented, the reset endpoint does not require authentication and only allows setting boot_flag = 0, which returns a board to normal boot. This operation is part of the normal provisioning workflow and may need to be called by the board itself or by other components (i.e. monitoring Surgeon service) in the system. Its impact is limited, since it cannot force provisioning mode, change the image URL, rename a board, or delete configuration. The main possible effect is that a client on the local network could clear the provisioning flag at the wrong moment, causing the board to return to normal boot instead of continuing provisioning. In the restricted UT IoT network, the limited risk was considered acceptable.

5.3.4 Need for HTTPS

A remaining risk in the current design is that plain HTTP does not protect traffic in transit. If a device on the same network were able to observe or interfere with traffic, it could potentially read administrative requests, including the Bearer token, or modify requests and responses before they reach the server. Using HTTPS would reduce this risk by providing

encryption, integrity protection, and server authentication. However, HTTPS was not implemented in this project because the service is only intended to run inside the restricted UT IoT network, where the likelihood of such interception is significantly lower than on a public network. In addition, deploying HTTPS would require extra infrastructure, such as certificate management, trusted hostnames, and reverse-proxy or TLS configuration, which was outside the scope of the project. For the current deployment, Bearer-token protection on administrative endpoints was considered a sufficient trade-off. For future versions of the system, adding HTTPS would be a good improvement.

5.3.5 Overall assessment

The main risk for the HTTP server is not public file download, but unauthorized state changes through the controller API. The risk is reduced by requiring authentication for administrative endpoints. The main remaining trade-off is the unauthenticated public reset endpoint, which has a very limited impact.

6 Conclusion

6.1 Completeness of Final Product

Most importantly, the KaaS system fulfills its primary goal: On-demand autonomous online provisioning via network boot. The end-to-end flow described in the Design section all works together: The Controller API dictates whether a reset is necessary, the U-Boot script reads its output and hands off to the SD card or fetches the provisioning OS over the network, and Quartermaster then flashes the target image, with no physical intervention required. The initial requirements are met alongside this: Configurations are now stored in a database, a considerably cleaner, more modular approach to config management compared to the original hardcoded Bash. The controller API itself is a RESTful service with a documented specification, so other services can build on top, and the dynamic OS image and Quartermaster references are ready to support multiple images, including ones of different operating systems.

A handful of planned minor features did not make it in the final product. The decision was made to abandon HTTP inside U-Boot after we ran into unsolvable issues with the bootloader’s TCP stack. This was an acceptable compromise, as the Provisioning OS is small, and the TFTP server is very lightweight. Together with our client, we determined HTTPS on the Controller API to be out of scope, and the Bearer-token authentication on state-modifying endpoint covers the threat model. Additionally, a more detailed state reporting and logging storage system was mentioned by the client but not implemented, as we prioritised getting the core provisioning pipeline ready. Due to time management issues, a full 32-board rack deployment ended up out-of-reach for the project. The system has also been designed with a future Surgeon and scheduler system integration in mind. Lastly, the one time bootloader flashing step is knowingly unavoidable, but a small price to pay for what is otherwise

complete automation. Overall, both the exploratory and implementation goals of the project have been met.

6.2 Takeaways

Overall, the project was a fresh and challenging experience. We started out with lots of question marks; we had to pick a design direction for ourselves, while adhering to the requirements of our client. We learned a lot about embedded environments: how to work with U-Boot; what goes into assembling a custom Linux distribution; and that even though it's a relic from the past, the serial interface is an exceptionally useful tool when working with hardware.

We also came to appreciate, sometimes the hard way, that working on a complicated system spanning software, hardware, and networking has its own peculiarities. Our asynchronous way of working allowed for a smooth and independent workflow, but had notable downsides during the integration phase. Bugs nobody saw in isolation showed up the moment the components were put together. We also underestimated unfamiliar platforms; build times, toolchain quirks, and undocumented dependencies all ate more of the schedule than planned. Moreover, we had not established a definite end for the project; we did not consider what came after the final integration testing and did not end up having time for a wider live deployment with the production Kria boards.

Ultimately, we set out to explore what was possible in automated Kria provisioning and we left with a working system. We architected a solution spanning multiple domains, from embedded firmware, through operating systems and networking, to web servers. The learned lessons have been documented, and the remaining work is large scale validation, and not more exploration. We hope the code, the manuals, and this report can make life easier for whoever ends up deploying KaaS in the wild.

Appendix

Appendix A: Link to source code

In this repository, you can find our project’s source code. Furthermore, the manuals are available in the repository as well. However, for your convenience, they are included in the appendix as well.

https://gitlab.utwente.nl/m11_kaas

Appendix B: Manuals

The manuals are included here for completeness, but they were originally written in markdown, and they are available in the best visual quality in the Gitlab, linked above.

B.1 Bootloader: Compiling & Reflashing

When setting up the system, you will need to create your own version of the bootloader that has the IP address of your server. Assuming that this address does not change, you only need to do this once. This process also uses PetaLinux, but version 2025.1 instead of 2025.2. Thankfully, PetaLinux supports multiple different versions being installed on the same machine.

Requirements

PetaLinux installation instructions can be found in the kria_provisioning_os repository's README.

Install PetaLinux SDK 2025.1 from AMD:

```
mkdir -p /opt/petalinux/2025.1
# Installer needs to run as a non-root user
yes | ./petalinux-v2025.1-*-installer.run -d /opt/petalinux/2025.1
```

Allow 300+ GB disk space and plenty of time for the installation.

In addition to a different version of the PetaLinux SDK, you will also need a template for the bootloader

Building for the first time

First, we need to create a project from the .bsp with

```
petalinux-create project --source
./xilinx-kv260-starterkit-v2025.1-05221048.bsp --name
"custom_loader"
```

This can take a few seconds. Enter the directory that was created by this command. Then run `petalinux-build` This will build the project for the first time. It is necessary that you do this before the later configuration steps.

Baking a custom script

In order to bake a custom script, you have two options.

The primary (and recommended way) is to use generate-boot-bbappend.sh from this repository. It will take a server ip as a parameter and perform all necessary configurations.

Alternatively, if you don't like scripts for whatever reason, you can directly copy-paste this command into your shell, making sure to adjust the server ip:

```
cat >
/home/admin/xilinx-kv260-starterkit-2025.1/project-spec/meta-user/
recipes-bsp/u-boot/u-boot-xlnx_%.bbappend << 'BBEOF'
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI += "file://custom-env.cfg"

do_configure:append() {
   cat > ${S}/include/configs/xilinx_zynqmp_custom.h <<
'HEADER_EOF'
#ifndef __XILINX_ZYNQMP_CUSTOM_H
#define __XILINX_ZYNQMP_CUSTOM_H

#define CUSTOM_TFTP_ENV \
   "ipaddr=192.168.0.50\0" \
   "serverip=192.168.0.100\0" \
   "tftp_boot=" \
     "echo Starting Network Configuration...;" \
     "echo Fetching file for MAC: ${ethaddr};" \
     "if tftpboot ${scriptaddr} configs/${ethaddr}; then " \
          "echo File loaded successfully. Importing
variables...;" \
          "env import -t ${fileaddr} ${filesize};" \
          "if itest ${boot_flag} == 1; then " \
               "echo Boot flag is 1, booting from network.;" \
          "else " \
               "echo Boot flag is 0, booting from SD card.;" \
               "setenv boot_targets mmc1;" \
               "run distro_bootcmd;" \
          "fi;" \
          "md.b ${scriptaddr} ${filesize};" \
     "else " \
          "echo TFTP Error: Could not fetch file from
${serverip};" \
          "echo Falling back to SD card boot...;" \
          "setenv boot_targets mmc1;" \
          "run distro_bootcmd;" \
     "fi\0" \
   "tftp_kernel=" \
     "if tftpboot 0x10000000 boot/image.ub; then " \
          "echo Got linux kernel image;" \
     "else " \
          "echo Failed getting linux kernel image;" \
```

```
           "echo Falling back to SD card boot...;" \
           "setenv boot_targets mmc1;" \
           "run distro_bootcmd;" \
      "fi\0"

#endif
HEADER_EOF

   sed -i '/#ifndef __XILINX_ZYNQMP_H/a #include
"xilinx_zynqmp_custom.h"' \
       ${S}/include/configs/xilinx_zynqmp.h

   sed -i 's/ENV_MEM_LAYOUT_SETTINGS \\/ENV_MEM_LAYOUT_SETTINGS
\\\n\tCUSTOM_TFTP_ENV \\/' \
       ${S}/include/configs/xilinx_zynqmp.h

   sed -i 's|^bootcmd=.*|bootcmd=run tftp_boot; run tftp_kernel;
bootm 0x10000000|' \
       ${S}/board/xilinx/zynqmp/zynqmp_kria.env
}
BBEOF
```

Regardless of which path you've chosen, you're now ready to move on to the next step. If you want to change something in the script that gets baked in other than the server ip, in both methods you will need to edit the section indented under

```
#define CUSTOM_TFTP_ENV
```

Build

Building the bootloader specifically takes a few more commands than building an entire image.

Run the following:

```
petalinux-build -c u-boot -x cleansstate
petalinux-build -c u-boot
petalinux-build
petalinux-package --boot --u-boot --force
```

Depending on your hardware, after anywhere between half an hour and several days the process will complete.

Flashing with the recovery web-ui

On the computer where you ran the build, in the project folder, under images/linux, you will find BOOT.BIN.

Copy this file to a computer you have access to with a desktop environment. This may be the same computer you ran the build on.

To this computer, connect your Kria through ethernet. You have quite a bit of freedom in how you precisely set this up, but the most important thing is that it is on a network where the routers don't assign the address 192.168.0.111, or any address in the range 192.168.0.50 - 192.168.0.100. We recommend connecting the Kria to the computer with the desktop environment directly. With this set up, you give your machine a static IP within the previously mentioned interval. Then, you hold down the button on the Kria that says FWUEN. While holding it, you power it on. If you set up the serial output it will show a message when it has entered recovery mode, but if you do not want to bother with that it is sufficient to wait 10 seconds before you let go of FWUEN.

Once you have started the Kria in recovery mode, open the Chrome web browser on your computer. We recommend Google Chrome for the flashing process, as this is the only browser (to our knowledge) that has no issues with corrupting the uploaded bootloader on any operating system. The interface is fairly straight forward, you want to upload your BOOT.BIN into one of the two slots, and make sure that "requested image" is set to that slot. It will upload and perform a CRC check. If everything clears without error messages, your Kria should have a shiny new bootloader the next time it starts.

B.2 Flag API

The following repository contains the software running on the central server of the Kria-as-a-Service (KaaS) system:

- A TFTP server providing the boot image, and communicating the boot state to the Kria boards.
- A RESTful API for external systems, and Kria boards (provisioning OS state) to interact with KaaS and each other.

Requirements

The network and the host should be configured in a way they are accessible from both the Kria boards, and the external systems. This involves setting up firewall rules for the web server (usually TCP `8000`), and the TFTP server (UDP `69`). TFTP may require a stateful firewall or enabling a TFTP specific profile on the firewall.

The production deployment is wrapped inside of a Docker container. The essential requirements to deploy are:

- Docker
- Docker Compose

For development, `Python3` with `Pip` is necessary.

Deployment

1. Point the `DATA_DIR` environment variable where application data (configs, and images) will live. (default: `./data`)
2. Set `API_KEY` to a secure random value. This Bearer token will be used to authenticate external services and the system administrator.
3. Set `PROVISION_TARBALL_URL` to a URL pointing to a packaged Quartermaster or other provisioning script. (recommended: `/files/quartermaster.tar.gz` - relative URLs are resolved to the API's host)
4. Deploy the software stack with `docker compose up -d`. Docker will automatically create directories for the web and TFTP servers.
5. Obtain a packaged Quartermaster instance (`quartermaster.tar.gz`) and place it in `DATA_DIR/https/quartermaster.tar.gz`. This file may be hosted on other web servers as well, in which case `PROVISION_TARBALL_URL` needs to point to an external URL i.e. `http://example.org/static/quartermaster.tar.gz`. *See the Quartermaster repository for more details.*
6. Obtain a provisioning OS image and place it in `DATA_DIR/tftp/boot/image.ub`. *See the Provisioning OS repository for more details.*
7. Place your OS images of choice in `DATA_DIR/https/*` or host them in another location of your choice. Relative URLs are resolved, so `/files/ubuntu.img.xz` will point to `ubuntu.img.xz` in the API's static directory.

That's it; the Kria board objects can now be added via the API. See detailed API spec in the last section.

Example file structure:

data/

├── https/

│ ├── iot-limerick-kria-classic-desktop-2204-20240304-165.img.xz

│ └── quartermaster.tar.gz

└── tftp/

 ├── boot/

```
│   └── image.ub
└── configs/
    ├── 00:0a:35:0f:25:74
    └── 00:0a:35:14:b6:7a
```

Note: Compose supports `.env` files out-of-the-box.

API Development

For local development outside of Docker.

Set environment variables (for example via `.env` in the project root):

Variable	**Purpose**
`API_KEY`	Bearer token for admin Kria endpoints (required).
`TFTP_CONFIG_DIR`	Directory of per-board config files, e.g. `/srv/tftp/configs`.
`FILES_ROOT`	Directory exposed at `GET /files/{filename}`, e.g. `/srv/http/files`.
`PROVISION_TARBALL_URL`	Full URL returned as `provision_tarball_url` on Kria objects (global default; not stored per board).

The app verifies that `TFTP_CONFIG_DIR` and `FILES_ROOT` exist and are directories at startup.

```
pip install -r requirements.txt
uvicorn app.main:app --host 0.0.0.0 --port 8000
```

API

Health

`GET /health`

```
curl "http://<PI_IP>:8000/health"
```

Returns `{"ok": true}`.

File hosting (public)

`GET /files/{filename}` — streams a file from `FILES_ROOT`.

```
curl -LO "http://<PI_IP>:8000/files/quartermaster-main.tar.gz"
curl -LO "http://<PI_IP>:8000/files/image.ub"
curl -LO
"http://<PI_IP>:8000/files/iot-limerick-kria-classic-desktop-2204-
20240304-165.img.xz"
```

Kria objects (public)

Three simple rules

Where	**What you use**
Config file on disk (under `TFTP_CONFIG_DIR`)	MAC **with** colons, e.g. `00:0a:35:0f:25:74` — same name your bootloader expects.
HTTP URL (query by MAC)	Same address **without** colons: `000a350f2574` — one path segment, no `%3A`.
JSON `id`	Same as the URL: `000a350f2574`. **`mac`** in JSON is the pretty form with colons.

The API turns `000a350f2574` into the filename `00:0a:35:0f:25:74` internally. You do **not** put colons in the URL.

Typical keys in the file:

- `hostname=...`
- `boot_flag=0|1`
- `image_url=http://.../files/....img.xz`

`provision_tarball_url` in JSON is **not** read from these files; it always comes from `PROVISION_TARBALL_URL`.

List all boards

`GET /api/kria`

```
curl "http://<PI_IP>:8000/api/kria"
```

Routes **`/mac/...`** (12 hex, no colons) and **`/hostname/...`** do not overlap. If you paste a MAC with colons into `/mac/...`, it still works; prefer **no colons**.

Get one by MAC

`GET /api/kria/mac/{mac}` — `{mac}` = **12 hex digits, no colons** (same as `id`).

```
curl "http://<PI_IP>:8000/api/kria/mac/000a350f2574"
```

Get one by hostname

`GET /api/kria/hostname/{hostname}`

```
curl "http://<PI_IP>:8000/api/kria/hostname/kria26"
```

Public reset (only clears provisioning)

Sets `boot_flag=0` only (use **`PATCH /api/admin/kria/{mac}`** with a Bearer token to set `0` or `1`). Path `{mac}` is **12 hex without colons**, same as `id`.

`PATCH /api/kria/mac/{mac}/reset`

```
curl -X PATCH
"http://<PI_IP>:8000/api/kria/mac/000a350f2574/reset"
```

`PATCH /api/kria/hostname/{hostname}/reset` (same behavior without the MAC in the URL)

```
curl -X PATCH "http://<PI_IP>:8000/api/kria/hostname/kria26/reset"
```

Returned Kria object shape

```
{
  "id": "000a350f2574",
  "mac": "00:0a:35:0f:25:74",
  "hostname": "kria26",
  "boot_flag": 0,
  "image_url": "http://<server>/files/<image>.img.xz",
  "provision_tarball_url":
"http://<server>/files/quartermaster-main.tar.gz"
}
```

Kria admin CRUD (admin)

All routes require **`Authorization: Bearer <API_KEY>`**.

Create

`POST /api/admin/kria`

- **Body:** `{ "mac", "hostname", "image_url", "boot_flag" }`
- Creates on-disk file `TFTP_CONFIG_DIR/<aa:bb:cc:dd:ee:ff>` (colon MAC for the bootloader). Response **`id`** is **12 hex without colons**.
- Rejects duplicate hostnames across existing board files.
- **`hostname`** must be `1-64` characters: letters, digits, and hyphens only.

```
curl -X POST "http://<PI_IP>:8000/api/admin/kria" \
  -H "Authorization: Bearer <API_KEY>" \
  -H "Content-Type: application/json" \
  -d '{
    "mac":"00:0a:35:0f:25:74",
    "hostname":"kria26",

"image_url":"http://<PI_IP>:8000/files/iot-limerick-kria-classic-d
esktop-2204-20240304-165.img.xz",
    "boot_flag":0
  }'
```

Update

`PATCH /api/admin/kria/{mac}` (path: **12 hex without colons**, same as public `/api/kria/mac/...`; colons in the path are still accepted)

- **Body:** any of `hostname`, `image_url`, `boot_flag` (omit fields you do not change).
- **`hostname`** must match `1-64` characters: letters, digits, and hyphens only.

Use this endpoint (with auth) to set **`boot_flag` to `0` or `1`**; there is no separate `/v1/flags` route.

```
curl -X PATCH "http://<PI_IP>:8000/api/admin/kria/000a350f2574" \
  -H "Authorization: Bearer <API_KEY>" \
  -H "Content-Type: application/json" \
  -d '{"boot_flag":1}'
```

Delete

`DELETE /api/admin/kria/{mac}`

- Removes `TFTP_CONFIG_DIR/<aa:bb:cc:dd:ee:ff>` from disk (`unlink`). Path `{mac}` is **12 hex without colons** (same as `id`).

```
curl -X DELETE "http://<PI_IP>:8000/api/admin/kria/000a350f2574" \
  -H "Authorization: Bearer <API_KEY>"
```

B.3 Provisioning OS: Building and Flashing

The following is a quick guide intended for team members and future maintainers to get up and running with the provisioning OS setup.

The provisioning OS is a minimal ramdisk Linux for remotely provisioning Kria KV260 SD cards. It boots into RAM, obtains a DHCP lease, pulls the Quartermaster provisioning tool, and flashes the connected SD card with the intended operating system.

The provisioning OS image is based on PetaLinux, AMD's wrapper toolkit for an *"easier"* Linux based product compared to a raw Yocto pipeline.

The output of this process is an `image.ub` file containing the kernel, device tree, and rootfs required for network booting. The file needs to be placed inside of the appropriate config directory of the controller API.

Note: This process only needs to be performed when the `image.ub` file is missing, or i.e. when the network environment changes.

Requirements

Spin up a fresh Ubuntu 22.04 Server LTS VM and run the following commands exactly as they are provided. AMD's official documentation has missing prerequisites, so the provided instructions are recommended. The exact dependencies and commands are a result of experimentation.

Nonetheless, the following link may still be a helpful resource in case of installation failure.

Build dependencies

```
apt-get update && apt-get install -y \
  gawk python3 python3-pip python3-pexpect python3-git
python3-jinja2 \
  xz-utils texinfo gcc-multilib libsdl1.2-dev libglib2.0-dev \
  build-essential autoconf automake libtool libncurses5-dev \
  zlib1g-dev libssl-dev net-tools xterm locales \
  cpio rsync bc u-boot-tools diffstat chrpath socat \
  wget git unzip tar gzip bzip2 lz4 zstd iproute2 \
  libtinfo5 libncurses5 screen tftpd-hpa

# Yocto requires bash as /bin/sh (not dash)
dpkg-reconfigure dash  # select "No"

# Yocto requires UTF-8
locale-gen en_US.UTF-8 && update-locale LANG=en_US.UTF-8
```

Install PetaLinux SDK 2025.2 from AMD:

```
mkdir -p /opt/petalinux/2025.2
# Installer needs to run as a non-root user
yes | ./petalinux-v2025.2-*-installer.run -d /opt/petalinux/2025.2
```

Allow 300+ GB disk space and plenty of time for the installation.

Build

This is the actual build process. Run the following to apply any configuration changes and to generate a `image.ub` file.

```
# Activate PetaLinux installation
source /opt/petalinux/2025.2/settings.sh

# Necessary on a freshly cloned git repository
```

```
# Generates files ignored by git
petalinux-config --silentconfig

# The actual build instruction
petalinux-build
```

Output: `images/linux/image.ub` (~75 MB FIT image containing kernel + device tree + rootfs ramdisk).

Development

To change the general system configuration, such as bundled packages, use the interactive `petalinux-config` menu.

To change the API base URL or the target block device, edit the `API_BASE` and `DEVICE` variables at the top of `project-spec/meta-user/recipes-apps/provision-startup/files/provision-startup.sh`.

Boot

Serve via TFTP, or bundle with `petalinux-package` and flash to an SD card (development):

image.ub

Serial console: (usually) `ttyPS1` at 115200 baud. Login: automatic over serial.

B.4 Quartermaster

Quartermaster is an automated provisioning tool for the Kria KV260 Vision AI Starter Kit used at the University of Twente.

This version of Quartermaster is designed to run automatically from an intermediate Linux environment (Petalinux) directly on the Kria device itself to overwrite the device's internal storage without manual intervention.

How it Works

Quartermaster operates in a **two-stage** architecture:

1. **Offline Stage** Runs in Petalinux on the Kria. It flashes the final OS image to the SD card/eMMC, expands the partition, and writes seed configuration files (hostname,

SSH config, SSH keys, GnuPG proxy, and the bootstrap service) directly into the filesystem.

2. **Online Stage** When the newly formulated OS image boots up, `systemd` automatically runs the `quartermaster-bootstrap.service`. This service downloads the necessary PYNQ environment setup, generates and applies a secure Jupyter password, and then self-disables so it never runs again.

Intermediate Environment (Petalinux) Requirements

For Quartermaster to successfully run inside the intermediate Petalinux environment, the environment **must provide the following**:

1. System Tools & Image File

Ensure standard Linux utilities (like `partprobe`, `e2fsck`, `lsblk`, `dd`, `mount`) are present in the Petalinux environment.

The corresponding OS image must be available in the Petalinux environment. If the mapping server indicates a "normal" base image, the corresponding image is: `https://people.canonical.com/~platform/images/xilinx/kria-ubuntu-22.04/iot-limerick-kria-classic-desktop-2204-20240304-165.img.xz`

2. Available Resources & Files

Quartermaster requires its assets to be localized nearby. A typical Petalinux layout before execution should look like this:

```
/
├── tmp/
│   └── ubuntu-base-image.img.xz        <-- 1. Downloaded OS image
└── opt/quartermaster/
    ├── quartermaster                   <-- 2. The main executable script
    └── data/                           <-- 3. The data directory
        ├── authorized_keys
        ├── 10-quartermaster.conf
        ├── quartermaster-bootstrap.service
        └── quartermaster-bootstrap.sh
```

Note: The script relies on the `data/` folder being in the exact same directory as the `quartermaster` script itself.

3. Dynamic Configuration via HTTP

The Petalinux environment must resolve the specific device data (like hostname and the correct OS image version) before invoking Quartermaster, by communicating with an HTTP mapping server mapping MAC addresses to configurations.

Usage

Since Quartermaster runs destructive block-level commands, it requires **root / superuser privileges**.

Command Syntax

sudo ./quartermaster --hostname <hostname> --image <image> --device <device> [-y]

- `--hostname` (-h): The desired hostname for the new OS (e.g. `kria05`).
- `--image` (-i): Path to the image (filename matters)
- `--device` (-d): Target device to flash
- `--yes` (-y): Skip all safety prompts and immediately overwrite the device.

Example Automation Flow

When Petalinux boots up, the script should:

```
# 1. Ask mapping server who we are and what image we need
HOSTNAME="kria05"
IMAGE_URL="http://your-server/images/ubuntu.img.xz"

# 2. Download the OS into Petalinux temporary memory
wget $IMAGE_URL -O /tmp/os.img.xz

# 3. Flash it to the SD Card silently (-y)
sudo ./quartermaster --hostname "$HOSTNAME" --image /tmp/os.img.xz
--device /dev/mmcblk1 -y

# 4. Restart Kria into the newly flashed permanent OS
reboot
```

Statement on the use of Artificial Intelligence

No artificial intelligence tools were used during development or the writing of this report.