Förbättring av CI/CT/CD-processer för firmwareutveckling och testning av IoT-sensornoder

Examensarbete inom högskoleingenjörsprogrammet i datateknik

Zebastian Björkqvist

INSTITUTIONEN FÖR Data- och informationsteknik

CHALMERS TEKNISKA HÖGSKOLA
Göteborg 2025
www.chalmers.se

Examensarbete 2025

Förbättring av CI/CT/CD-processer för firmwareutveckling och testning av IoT-sensornoder

Zebastian Björkqvist

CHALMERS

Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
Göteborg 2025

Förbättring av CI/CT/CD-processer för firmwareutveckling och testning av IoT-sensornoder
Zebastian Björkqvist

Handledare:
Fareed Qararyah, *Chalmers University of Technology*
Muhammad Ibrahim, *ZessAB*

Examinator:
Nicholas Smallbone , *Chalmers University of Technology*

Examensarbete 2025
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Skriven i LaTeX
Göteborg 2025

Förbättring av CI/CT/CD-processer för firmwareutveckling och testning av IoT-sensornoder
Zebastian Björkqvist
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola

Sammanfattning

I dagens snabbt växande IoT-landskap är effektiv firmwareutveckling och bra distribution avgörande för att upprätthålla produktkvalitet och möta behovet av snabba iterationer. Manuella processer för firmwarehantering leder ofta till ineffektivitet, kvalitetsproblem och ökad risk för fel vid uppdateringar. Detta examensarbete beskriver design och implementation av en automatiserad Continuous Integration (CI), Continuous Testing (CT) och Continuous Deployment (CD) process för firmwareutveckling och testning av IoT-sensornoder hos ZESS AB. Arbetet fokuserar främst på företagets Itirub-plattform, som finns i sex olika hårdvaruvarianter där de olika kommunikationsgränssnitt ethernet+, WiFi och LoRa kombineras med sensorgränssnittet analogt eller digitalt. Den implementerade lösningen ersätter en tidigare manuell bygg-, test- och driftsättningsprocess med en automatiserad pipeline som inkluderar variantspecifik konfigurationshantering, automatiserad byggning, artefakthantering och testautomatisering. Systemet utvecklades med Python som primära programmeringsspråk, Jenkins användes som plattform för testning, testerna är skrivna i språket Robot Framework, Git som versionshanteringssystem och JFrog Artifactory som platform för att lagra artifakter. Detta skapade en sammanhängande CI/CT/CD-pipeline som förbättrar utvecklingseffektiviteten och firmwarekvaliteten. Resultaten visar betydande tidsbesparingar i utvecklingsprocessen genom automatisering, förbättrad spårbarhet genom Git-integration och utökad testtäckning över alla hårdvaruvarianter. Arbetet visar hur rätt automatisering kan transformera utvecklingsprocesser för inbyggda system samtidigt som rigorösa kvalitetsstandarder för industriella IoT-applikationer upprätthålls.

Nyckelord: Continuous Integration, Continuous Testing, Continuous Deployment, CICD, Inbyggda System, Firmware Over-The-Air, IoT, Testautomatisering, DevOps

Akronymer

Nedan är en lista på akronymer som används genom projektrapporten

ADC	Analog-to-Digital Converter
CI	Continuous Integration
CT	Continuous Testing
CD	Continuous Deployment/Delivery
FOTA	Firmware Over-The-Air
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
OTA	Over-The-Air
SNTP	Simple Network Time Protocol
YAML	YAML Ain't Markup Language

Innehåll

Figurer

Listings

1

Inledning

Detta kapitel presenterar bakgrunden till projektet, dess syfte och mål samt de avgränsningar som gjorts. Kapitlet ger läsaren en förståelse för varför det är viktigt att automatisera firmwareutvecklingsprocessen för IoT-sensornoder och vilka specifika problem som detta examensarbete adresserar.

1.1 Bakgrund

I alla utvecklingsprocesser är det av stor vikt att testningen är gedigen, effektiv och tillförlitlig. Detta är av särskild vikt i system med flera hårdvaruvarianter där olika kodversioner måste hantera olika konfigureringar och kommunikationsprotokoll. ZESS AB är ett perfekt exempel på detta då ZESS AB utvecklar firmware för Itirub-sensornoder i sex olika hårdvaruvarianter, antingen digitala eller analoga ingångar med kommunikationsmetoderna LoRa, WiFi eller ethernet. Itirubs sensorer är designade för industrimiljöer och finns i flera varianter för att kunna anpassas till kundens specifika behov. Till exempel är LoRa en lämplig lösning i stora utomhusområden med gles sensorplacering, medan ethernet erbjuder hög stabilitet men kan vara opraktiskt i större anläggningar där kabeldragning är utmanande.

För närvarande kräver varje kodändring att utvecklare manuellt bygger firmwarebinärfiler för samtliga varianter och testar dem individuellt med en rad befintliga tester för de olika varianterna. Denna process är ineffektiv, tidskrävande och ökar risken för fel vid uppdateringar. För att ta itu med dessa utmaningar behöver en automatiserad continuous integration(CI)/continuous distribution(CD)/continuous testing(CT) process implementeras som säkerställer en smidig och pålitlig firmwareutvecklingsprocess.

En viktig del av den nya distributionsprocessen är integration med företagets befintliga Over-The-Air (OTA)-uppdateringssystem, vilket möjliggör fjärrinstallation av ny firmware på testsensorerna i labbet. I det nuvarande arbetsflödet kräver varje uppdatering flera manuella steg där utvecklare först måste bygga firmwaren, sedan konvertera den till FOTA-format och därefter manuellt initiera överföringen till testenheterna. Detta är både tidskrävande och öppnar för fel vid hanteringen. Med den automatiserade processen hanteras hela denna kedja från byggnation till distribution utan manuella ingrepp, vilket siktar på att förkorta testcykeln och säkerställa att rätt firmware installeras på respektive testenhetsvariant

1.2 Syfte

Syftet med detta examensarbete är att jämföra hur en CI/CT/CD-process förbättrar utvecklingsprocessen genom att designa och implementera en automatiserad CI/CT/CD-pipeline för firmwareutveckling och testning av Itirub-sensornoder. Denna pipeline möjliggör automatisk byggnation, distribution och testning av firmware för samtliga hårdvaruvarianter. Genom att minska manuella insatser och potentiella fel leder det till en mer effektiv utvecklingsprocess och ökad pålitlighet i firmwaredistributionen.

1.3 Mål

Målet med projektet är att skapa en automatiserad CI/CT/CD-process som innefattar följande punkter:

- Automatiskt bygga firmware-binärfiler för alla hårdvaruvarianter vid kodändringar.
- Implementera en automatiserad mekanism för att identifiera rätt hårdvaruvariant och säkerställa att binärfiler installeras korrekt.
- Implementera och köra variantspecifika tester genom en CI/CT-process.
- Skapa en robust återställningsmekanism där main-firmware automatiskt återställs vid test- och firmwarefel.
- Automatiserar firmwaredistributionen via Over-The-Air (OTA) uppdateringar.

1.4 Avgränsningar

För att säkerställa ett genomförbart och fokuserat examensarbete har följande avgränsningar gjorts:

- Projektet fokuserar enbart på de sex befintliga Itirub-varianterna och tar inte hänsyn till framtida varianter som kan tillkomma.
- Arbetet omfattar inte optimering av själva firmware-koden eller testskripten, utan fokuserar på automatiseringen av byggprocessen och testutförandet.
- Implementeringen använder företagets befintliga infrastruktur för OTA-uppdateringar och testning, med modifikationer endast där det krävs för automationsprocessen.
- Automatiseringen implementeras i en CI/CD-plattform, i detta arbete användes Jenkins, som redan fanns delvis implementerad.
- Testning begränsas till befintliga testfall.

2
Metod

Detta kapitel beskriver de förberedande stegen som genomfördes innan implementeringen av processen.

2.1 Utvecklingsmetodik

Implementationen av CI/CT/CD-processen för Itirub-sensornoder genomfördes enligt en iterativ metod. Detta tillvägagångssätt valdes för att möjliggöra kontinuerlig utvärdering och anpassning under projektets gång. Metodologin omfattade fyra huvudfaser där varje fas förfinade och byggde vidare på resultat från föregående steg.

1. **Analysfas** där befintliga manuella processer kartlades och krav på automationslösningen definierades.
2. **Designfas** där systemarkitektur och komponentval fastställdes.
3. **Implementationsfas** där komponenter utvecklades, integrerades och validerades.
4. **Validering** där alla komponenter samt helhetens funktionalitet validerades

I Implementationsfasen organiserades utvecklingsarbetet kring ett modifierat agilt arbetssätt med flera iterationer. Den första iterationen fokuserade på grundläggande byggprocessfunktionalitet och FOTA-konvertering. I den andra iterationen implementerades integration med Jenkins för testautomatisering. Den tredje iterationen omfattade installation och konfiguration av Artifactory inom företagets miljö, en fas som visade sig vara mer komplex än initialt förväntat. Den fjärde och avslutande iterationen fokuserade på integrering av samtliga komponenter till en sammanhängande pipeline och slutförande av projektleveransen.
Denna inkrementella metod möjliggjorde tidigt värdeskapande genom att systemets centrala funktionalitet blev tillgänglig för testning redan i projektets tidiga skeden. Det faciliterade också hantering av oförutsedda tekniska utmaningar, särskilt relaterade till tredjepartsintegrationer, genom att isolera dessa till specifika iterationer.

2.2 Systemarkitektur

Den planerade CI/CT/CD-lösningen byggdes med en modulär arkitektur med Git-repository som källa för kodincheckningar och startpunkt för CI-processen. Ett centralt Python-skript agerade som orkestrerare för att hantera den dynamiska konfigurationen och byggprocessen för alla sex Itirub-varianter. Artifactory användes för versionshanterad lagring av firmware, och Jenkins konfigurerades för automatiserad testexekvering på dedikerade testenheter.

2.3 Val av programmeringsspråk

För implementationen av CI/CT/CD-systemet valdes Python som huvudsakligt programmeringsspråk av flera skäl. Pythons plattformsoberoende natur och omfattande standardbibliotek gjorde det idealt för automationsuppgifter med filhantering och processexekvering. Språkets väletablerade integrationer med Jenkins API, JFrog Artifactory och Git genom bibliotek som requests och andra plattformsspecifika bibliotek förenklade kommunikationen med dessa tredjepartssystem som var del av CI/CD-processen.

2.4 Konfigurationshantering

En central utmaning i projektet var hanteringen av sex olika hårdvaruvarianter av Itirub. För att adressera detta användes en "single source of truth"-strategi, alltså att konfigurationen ska endas finnas på en plats och alla som behöver konfigurationen hämtar från den platsen, med en central YAML-konfigurationsfil som definierade alla variantspecifika parametrar. Denna fil innehöll kompileringsparametrar för olika hårdvaruvarianter, testflaggor för variantspecifika tester, och mappning mellan varianter och fysiska testnoder.
Detta tillvägagångssätt förväntades förenkla underhåll och säkerställde konsekvent konfiguration genom hela CI/CT/CD-flödet. Konfigurationsfiländringar versionshanterades, vilket gav spårbarhet vid ändringar. Samt att Python-skriptet behövde inte modifieras när fler tester eller varianter tillkom.

2.5 Testautomation

För automatiserad testning användes Robot Framework, som erbjuder en nyckelordsstyrd syntax lämplig för testautomation. Testerna exekverades på dedikerade fysiska Itirub-enheter.
Processen omfattade automatisk nedladdning av firmware från Artifactory med Over The Air-uppdatering (OTA) av testenheter via Message Queuing Telemetry Transport (MQTT), variantspecifik testexekvering baserat på konfigurationen, samt rapportgenerering och resultatlagring.
För att förenkla underhåll strukturerades testfallen så att gemensam funktionalitet delades mellan varianter, medan variantspecifika tester isolerades. Jenkins konfigurerades med parameteriserade jobb där projektet avsåg att parametrar valdes dynamiskt baserat på firmware-variant.

3

Teknisk Bakgrund

Detta kapitel presenterar den tekniska kontexten för projektet genom att redogöra för de underliggande koncept och teknologier som implementationen bygger på. Kapitlet behandlar principer för CI/CT/CD, Firmware Over The Air-uppdateringar (FOTA) och IoT-sensornoder i industriella miljöer.

3.1 CI/CT/CD-principer

Kontinuerlig Integration (CI), Kontinuerlig Testning (CT) och Kontinuerlig Driftsättning (CD) utgör grundläggande praktiker inom modern mjukvaruutveckling som syftar till att automatisera utvecklingsprocessen från kod till produktion [1]. Dessa koncept är centrala komponenter i DevOps-metodiken [2] och har särskild relevans för firmware-utveckling där testning på fysisk hårdvara är nödvändig.

3.1.1 Kontinuerlig Integration (CI)

Kontinuerlig Integration definieras som processen att regelbundet integrera kodändringar i en gemensam repository [3]. Genom att frekvent integrera mindre ändringar och automatiskt verifiera dem reduceras risken för integrationsproblem som kan uppstå när flera utvecklare arbetar parallellt.

3.1.2 Kontinuerlig Testning (CT)

Kontinuerlig Testning utvidgar CI-konceptet genom automatiserade tester som exekveras vid varje kodändring [4]. För firmware-utveckling innefattar detta testning på fysiska enheter i verklig miljö, validering under olika driftsförhållanden, kompatibilitetskontroller med externa system och mätning av icke-funktionella egenskaper som energiförbrukning och prestanda.

3.1.3 Kontinuerlig Driftsättning (CD)

Kontinuerlig Driftsättning representerar den avslutande fasen i utvecklingsflödet, där verifierad kod automatiskt distribueras till produktionsmiljön [5]. För IoT-enheter innebär detta oftast trådlösa (Over-The-Air) uppdateringar till enheter i drift [6].

3.2 Företagsspecifika teknologier

På Zess AB och i Itirub så används tekonoligerna FOTA, tre olika kommunikationsteknologier och två olika sensorinterface

3.2.1 FOTA-systemarkitektur

FOTA-system består av flera samverkande komponenter som möjliggör säker och tillförlitlig uppdatering av firmware på distans. [6] FOTA-systemet i Itirub-plattformen bygger på en konverteringsprocess där de kompilerade binärfilerna transformeras till ett särskilt FOTA-format. Detta format består av en krypterad textfil som innehåller firmware-koden. Konverteringen hanteras av ett specialanpassat verktyg (ItirubFotaCreator) som säkerställer korrekt formatering och kryptering av data. Distributionsprocessen använder MQTT-protokollet för att överföra FOTA-filer till målenheter, där dessa textfiler tas emot av en dedikerad OTA-klient implementerad i firmware. Klienten avkrypterar innehållet och installerar den nya firmwaren.

3.2.2 Kommunikationsteknologier

Itirub-plattformen implementerar tre kommunikationsmetoder med olika karakteristik [7]:

- **Ethernet:** Erbjuder bandbredd på 1000 Mbps, latens under 1 ms, och fast anslutning lämplig för dataintensiva tillämpningar [8].
- **WiFi (IEEE 802.11n):** Tillhandahåller trådlös kommunikation med bandbredd upp till 600 Mbps och en räckvidd på 12-70 meter inomhus [9].
- **LoRa:** Möjliggör kommunikation över långa avstånd runt 10 km med en bandbredd upp till 50 Kbps beroende på avstånd och konfiguration [10].

3.2.3 Sensorinterface

De två sensorinterface som implementeras i Itirub-plattformen är konfigurerade för olika typer av datainsamling [7]:

- **Analogt interface (ADC):** Baserat på ADS1115 16-bitars analog-till-digital-omvandlare med fyra kanaler och programmerbara förstärkning up till 16 gånger. Samplingsfrekvens upp till 860 prover per sekund. [11]
- **Digitalt interface (I/O-expander):** Implementerar digitala in- och utgångar för signaler.

Varje sensorkonfiguration kräver specifik firmware-parametrisering, vilket ökar komplexiteten i byggnations- och testprocessen eftersom varje kommunikationsmetod kan kombineras med varje sensortyp, vilket resulterar i sex distinkta varianter.

3.3 Andra tekniker

3.3.1 Python

Python är ett högninvåspråk med tydlig syntax och omfattande biblioteksekosystem som gör det idealiskt för automationsuppgifter [12]. Dess plattformsoberoende natur och kraftfulla funktioner för textmanipulation och API-kommunikation är särskilt värdefulla för byggprocessautomation i detta projekt.

3.3.2 Jenkins

Jenkins är en öppen källkods-automatiseringsserver för kontinuerlig integration [13]. För Itirub-utvecklingen utnyttjas särskilt dess förmåga att hantera parametriserade jobb för de olika hårdvaruvarianterna och automatisk schemaläggning av byggen och tester.

3.3.3 JFrog Artifactory

Artifactory fungerar som centralt artefaktlager med robust versionsspårning [14]. I projektet används det för att organisera och distribuera firmware-binärer med spårbarhet mellan källkod och installerad programvara via dess REST API.

3.3.4 Robot Framework

Robot Framework är ett testautomatiseringsramverk med nyckelordsstyrd syntax [15]. För Itirub-enheterna möjliggör det automatiserade funktionstester över olika kommunikationsprotokoll och sensorgränssnitt med tydlig resultatrapportering.

4
Genomförande

Detta kapitel beskriver implementationen av CI/CT/CD-systemet för Itirubplattformen. Fokus ligger på den systematiska utvecklingen av automationsprocessen, från nulägesanalys till färdig implementation. Kapitlet redogör för designbeslut, tekniska utmaningar och lösningsstrategier som tillämpats under projektets gång.

4.1 Analys av utgångsläget

Inledningsvis genomfördes en analys av företagets existerande firmwareutvecklingsprocess för Itirub. Analysen identifierade flera ineffektiviteter i den manuella processen. Den befintliga utvecklingscykeln för Itirub-firmware präglades av manuella processer som krävde betydande tidsåtgång och teknisk expertis. Varje kodändring nödvändiggjorde manuell konfiguration för specifika hårdvaruvarianter, vilket innebar modifiering av kompileringsparametrar i `conf_board.h` för varje variant. Denna process var inte bara tidskrävande utan även benägen att introducera fel genom mänskliga misstag vid konfigurationsändringar.
Byggprocessen utfördes genom manuellt exekverande av kompileringskommandon för varje variant, följt av manuell hantering och lagring av resulterande binärfiler. Avsaknaden av standardiserade lagringsrutiner leder till opålitlig hantering och svårigheter att spåra specifika firmwareversioner tillbaka till motsvarande källkod.
Testningen utfördes på individuell basis, där utvecklare manuellt konfigurerade testmiljön och exekverade testfall, ofta endast för ett begränsat antal varianter på grund av tidsbegränsningar. Detta medförde risk för att fel i specifika varianter förblev oupptäckta, vilket kunde leda till kvalitetsproblem i produktionsmiljö.
Distribution av firmware till testutrustning och produktionsenheter genomfördes manuellt via fysisk anslutning eller genom att manuellt lägga in firmwarefiler i FOTA-format i test-repositoryn (Den repository som innehåller alla testfall) samt att ändra i Robot Framework testet för att skicka till testenheter. Detta begränsade möjligheten till systematisk uppdatering av enheter.
Utöver processanalys genomfördes en teknisk granskning av källkodsstruktur och byggsystem. Versionshanteringen av byggda artefakter var undermålig, vilket försvårade spårbarhet mellan distribuerad firmware och källkod.

4.2 Design av systemarkitektur

Baserat på analysen av nuläget utformades en systemarkitektur med fokus på modularitet, skalbarhet och tillförlitlighet. Arkitekturen konstruerades för att lösa de

specifika problemområden som identifierats i den befintliga processen, samtidigt som den skulle vara anpassningsbar för framtida behov.

4.2.1 Arkitekturella principer

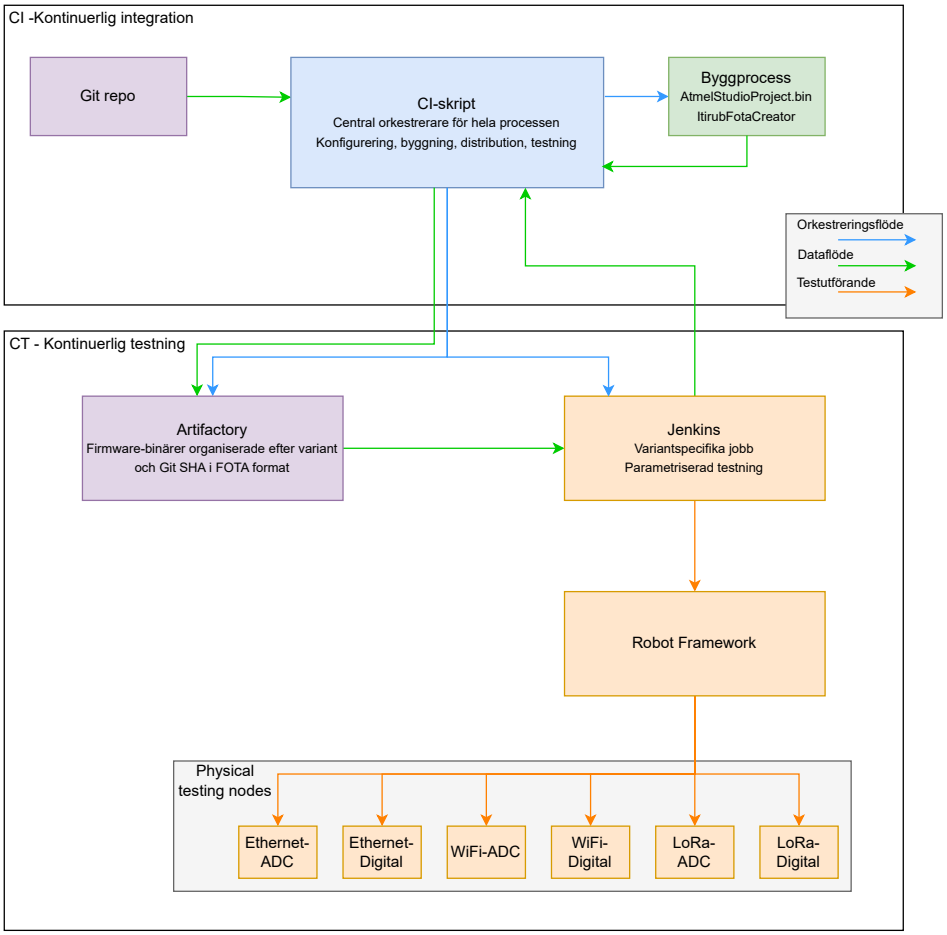
Automatisering prioriterades för alla repetitiva och manuella processer, med särskilt fokus på konfigurationshantering, byggprocess och testexekvering. Tanken var att minimera behovet av manuella ingrepp utan att minska på kvaliteten. Spårbarhet designades in i systemet från början, med automatisk inkludering av Git SHA i alla artefakter för att möjliggöra spårning från källkod till installerad firmware. Principen om *single source of truth* tillämpades genom designen av konfigurationen, där en central YAML-konfigurationsfil skapades som primära källa för all variantspecifik konfiguration. Detta eliminerade risken för olika konfigurationer och minskade behovet av manuella ändringar.

4.2.2 Komponentöversikt

Arkitekturen strukturerades kring följande huvudkomponenter.
CI-skript Ett centralt Python-baserat verktyg utvecklades för att orkestrera hela byggprocessen. Skriptet utformades för att läsa konfiguration från YAML-filen och hantera konfigurering, kompilering, konvertering och distribution för samtliga varianter. Detta tillvägagångssätt centraliserade bygglogiken och eliminerade behovet av manuell konfiguration.
Artifactory JFrog Artifactory implementerades som centralt artefakthanteringssystem för strukturerad lagring av byggda firmware-binärer i FOTA-format. Lagringen konfigurerades med en hierarkisk struktur baserad på variant och Git SHA, vilket möjliggjorde enkel navigering och hämtning av specifika firmware-versioner.
Jenkins Jenkins användes som plattform för kontinuerlig integration och testautomatisering. Jenkins konfigurerades med parametriserade jobb som automatiskt kunde exekvera relevanta tester för respektive variant, baserat på information från CI-skriptet.
Robot Framework Det befintliga Robot Framework utökades för automatiserad funktionell testning. Ramverket kompletterades med nya nyckelord och funktioner för att möjliggöra automatisering av firmwareinstallation.
FOTA-system Den existerande infrastrukturen för Firmware Over-The-Air-uppdatering integrerades med CI/CD-pipelinen för att möjliggöra trådlös distribution av firmware till testnoder. Komponenternas relationer och dataflöde illustreras i figur 4.1.

4.3 Implementation av CI-skript

En central del i den implementerade lösningen var utvecklingen av CI-skriptet, vilket fungerar som orkestrerings- och automationslager för hela systemet. Skriptet implementerades i Python på grund av språkets flexibilitet, omfattande standardbibliotek och goda samverkan med externa system.



Figur 4.1: Systemarkitektur för CI/CT/CD-lösningen

En särskild utmaning i implementationen var att skapa den dynamiska konfigurationshanteringen. Det tog mycket tid för att utveckla en robust metod för att manipulera konfigurationsfiler på ett konsekvent och pålitligt sätt. Implementationen av interaktionen med externa system som Jenkins och Artifactory var också mer komplex än från början förväntat, främst på grund av de specifika API-krav och autentiseringsbegränsningar som dessa system ställer.
CI-skriptet utformades med flera distinktiva funktionalitetsområden:

4.3.1 Konfigurationshantering

För att lösa utmaningarna med variantspecifik konfiguration implementerades en robust mekanism för dynamisk modifiering av konfigurationsfiler. Funktionaliteten centrerades kring YAML-konfigurationsfilen som definierar parametrar för varje variant. Som visas i listing 4.1 finns ett exempel på hur en variant är definierad i YAML-filen. På rad 2 och 3 definieras vilken board och app define just denna variant ska ha. På raderna 5-10 definieras vilka tester som ska köras eller inte för denna variant.

```
rpi-itirub-eth-adc:
  BOARD_DEFINE: BOARD_WITH_ADS1115_ADC
  APP_DEFINE: BUILD_ETHERNET_APP
  tests:
    RUN_ANALOG_SENSOR_TEST: true
    RUN_PARTNUMBER_TEST: true
    RUN_FOTA_TEST: true
    RUN_WIFI_TEST: false
    RUN_MQTT_SUB_TEST: true
    RUN_INCOMING_DATA_TEST: true
```

Listing 4.1: YAML variant konfigurations exempel

Skriptet implementerar en `update_conf_board`-funktion som dynamiskt manipulerar `conf_board.h`-filen baserat på den valda varianten. Funktionen använder regex uttryck för att identifiera och selektivt kommentera/avkommentera relevanta definitioner. Detta eliminerar behovet av manuell ändring och säkerställer konsekvent konfiguration över alla varianter.
En särskild utmaning var att utveckla tillräckligt exakta regex uttryck som korrekt kunde identifiera rätt kodrad även i närvaro av olika kodningsformat och kommentarstilar. Testning med olika konfigurationsfilversioner var nödvändig för att validera robustheten i denna lösning.

4.3.2 Byggnadsprocess

För kompilering av firmware implementerades en automatiserad byggprocess som består av automatisk identifiering och extraktion av Git SHA för inkludering i byggda artefakter, exekvering av `BuildAtmelStudioProject.bat`, och till sist konvertering av binärfil till FOTA-format genom exekvering av `ItirubFotaCreator`-verktyget med binärfilen som parameter. `BuildAtmelStudioProject.bat` batch-skriptet bygger källkoden för Itirub i dess egna repository och genererar en .bin fil som är firmwarebinären.

4.3.3 Artefakthantering

För strukturerad hantering av byggda artefakter implementerades integration med Artifactory via dess API. Implementationen omfattar följande:

- Konstruktion av standardiserad lagringshierarki baserad på variant och Git SHA.
- Uppladdning av artefakter med relevant metadata inkluderad både i filsökväg och som HTTP-headers för utökad sökbarhet.
- Robust felhantering med återförsökslogik för att hantera tillfälliga nätverksproblem.

Integrationen med Artifactory visade sig vara en av de mer utmanande aspekterna av implementationen. Databasen som Artifactory använder i bakgrunden var svårare att konfigurera än väntat. Det var blandad dokumentation kring hur det skulle göras och vad som behövdes för open source versionen av Artifactory.

4.3.4 Testorkestrering

För att möjliggöra automatisk testning implementerades integration med Jenkins via dess API genom Python biblioteket python-jenkins. Funktionaliteten omfattar dynamisk generering av testparametrar baserat på variantens konfiguration och testegenskaper, triggering av Jenkins-jobb med korrekt nodnamnparameter för att säkerställa att tester exekveras på rätt fysisk testnod, samt inkludering av parametrar såsom firmware-URL och Git SHA för att möjliggöra automatisk nedladdning och verifiering av rätt firmware. Jenkins-integrationen krävde djupare förståelse av Jenkins API och jobbparametrisering. Särskilt utmanande var att säkerställa hur python-jenkins biblioteket användes för att starta specifika testnoder med rätt parametrar.

4.3.5 Interaktionsgränssnitt

Ett interaktivt kommandoradsgränssnitt med guidade instruktioner, vilket underlättar för utvecklare som är mindre bekanta med systemet.
Implementationen inkluderar omfattande loggningsfunktionalitet för att möjliggöra diagnostik och felsökning vid eventuella problem.

4.4 Implementation av artefakthantering

För att etablera strukturerad lagring av firmware-artefakter installerades en JFrog Artifactory-instans med anpassad konfiguration. Installationen och konfigurationen av Artifactory i företagets miljö var en mer omfattande utmaning än initialt förväntat. Den krävde integration med företagets befintliga infrastruktur och hantering av databasinegration.
Implementationen fokuserade på de aspekter som beskrivs i kapitel 4.4.1 och 4.4.2

4.4.1 Repositorystruktur

Artifactory konfigurerades med en dedikerad repositorystruktur för Itirub firmware: Ett `itirub-fota` repository etablerades som primär lagringsplats för alla FOTA-formaterade firmwarefiler.
Inom detta repository implementerades en hierarkisk struktur där artefakter organiseras enligt mönstret `/{variant}/{git-sha}/{filnamn}`, exempelvis enligt det som visas i listing 4.2

```
1    /itirub-fota/rpi-itirub-eth-adc/2c07cca01dd5/
   Itirub_Ethernet_Digital_2c07cca01dd5.txt
```

Listing 4.2: FOTA filename example

Denna struktur möjliggör enkel navigering och filtrering baserat på variant och/eller Git-commit, vilket är värdefullt vid specifika sökningar eller vid behov av att identifiera alla varianter byggda från en specifik commit.

4.4.2 Säkerhet och åtkomstkontroll

För att säkerställa säker hantering av artefakter implementerades en strukturerad säkerhets- och åtkomstkontroll i systemet. Användare och grupper konfigurerades med distinkta åtkomsträttigheter baserat på roll och ansvar. En writers-grupp etablerades med behörighet att både läsa och skriva artifakter i itirub-fota repository, medan en readers-grupp skapades med begränsad behörighet som endast tillåter läsning av artifakter från samma repository.

4.5 Modifiering av Jenkins-konfiguration

För att möjliggöra automatiserad testning modifierades den befintliga Jenkins-infrastrukturen för integration med övriga komponenter i CI/CD-systemet. Modifieringarna omfattar det som beskrivs i kapitel 4.5.1 och 4.5.2

4.5.1 Jobbkonfiguration

Det redan implementerade Jenkins-jobbet `itirub_nightly` modifierades för att stödja automatiserad testning. Jobbet innehöll innan detta projekt en rad parametrar för att starta olika test. Jobbet modifierades genom att parameterisering utökades med nya parametrar såsom `FIRMWARE_URL` och `FOTA_FILENAME` för att möjliggöra automatisk nedladdning av specifika firmware-versioner från Artifactory. Testflaggor som tidigare var manuellt satta (t.ex. `FOTA_TEST` som visas i kod C.1 i appendix) sätts nu dynamiskt av CI-skriptet, vilket möjliggör dynamisk aktivering av variantspecifika tester. Ett nytt steg för firmware-nedladdning lades till i jobbets exekveringspipeline, vilket automatiskt hämtar firmwaren som specificerats av CI-skriptet från Artifactory i början av testprocessen.

4.5.2 Testintegration

För att säkerställa korrekt integration mellan Jenkins och testsystemet implementerades ytterligare funktionalitet i `FOTA_TEST` som visas i C.1 i appendix. Loggningsmekanismer förbättrades för att inkludera tydlig information om vilken firmware som testas med Git SHA. Testresultatarkivering implementerades i sista steget av Jenkins filen, som visas i B.1 i appendix, för att bevara testdata över tid, möjliggöra trendanalys och spårning av kvalitetsutveckling.

4.6 Utökning av Robot Framework-testfiler

För att möjliggöra automatiserad firmware-installation och testning utökades de befintliga Robot Framework-testfilerna med ny funktionalitet:

4.6.1 Firmware-hantering

Robot Framework-testfilen för FOTA-testning utökades med ett nytt nyckelord för automatiserad hantering av firmware. Nyckelord `Use Workspace Firmware` imple-

menterades för att automatiskt lokalisera och konfigurera tester för nedladdade firmware-filer. Det tillagda keyword visas i listing 4.3

```
| Use Workspace Firmware
| | [Documentation] | Use firmware from Jenkins workspace
| | # Look for firmware in workspace directory
| | ${firmware_dir}= | Set Variable | ${workspace_path}/firmware
| | ${files}= | List Files In Directory | ${firmware_dir} | *.txt
| | ${file_count}= | Get Length | ${files}
| |
| | Should Be True | ${file_count} > 0 | No firmware files found in
    workspace
| |
| | ${latest_file}= | Set Variable | ${files}[0]
| | ${firmware_path}= | Set Variable | ${firmware_dir}/${
   latest_file}
| | Log | Using firmware file from workspace: ${firmware_path}
| | Set Suite Variable | ${fmw_to_send} | ${firmware_path}
```

Listing 4.3: Keyword added in FOTA_TEST robot file

Detta nyckelord söker igenom Jenkins arbetskatalog efter nedladdade firmware-filer, identifierar den senaste filen och konfigurerar testet att använda denna firmware för FOTA-uppdatering.

4.6.2 Testflödesanpassning

Testflödet modifierades för att stödja både manuell och automatiserad exekvering:
Villkorlig logik implementerades för att avgöra om firmware ska hämtas från arbetskatalogen eller från en standardplats.
Verifieringslogik förbättrades för att validera att rätt firmware installerats, genom att jämföra Git SHA i firmware-filnamnet med information som rapporteras från enheten efter installation.
Testresultatrapportering utökades för att inkludera information om den testade firmwareversionen, vilket underlättar korrelation mellan testresultat och specifika kodändringar.

4.7 Integration och validering

Efter implementation av de individuella komponenterna genomfördes integrationsarbete för att säkerställa samverkan mellan alla delar av systemet. Den iterativa utvecklingsmetoden möjliggjorde inkrementell integration, där varje iteration byggde på och expanderade funktionaliteten från föregående steg.
Den första iterationen fokuserade på grundläggande byggprocessfunktionalitet och FOTA-konvertering, vilket etablerade den tekniska grunden för systemet. I den andra iterationen adderades Jenkins-integration, vilket möjliggjorde automatiserad testning. Den tredje iterationen omfattade installation och konfiguration av Artifactory, vilket var mer utmanande än förväntat men kritiskt för strukturerad artefakthantering som beskrivs i 4.4.1. Den fjärde och avslutande iterationen fokuserade på att

integrera alla komponenter till en sammanhängande pipeline och finslipa användargränssnittet.

4.7.1 Systemvalidering

För att verifiera systemets korrekta funktion genomfördes omfattande validering av hela processen. End-to-end-testning genomfördes för att validera hela flödet från första exekvering av skriptet till testrapportering, med särskilt fokus på korrekt hantering av variantspecifik konfiguration och testexekvering. Under arbetet och i de iterationer som beskrivs i 4.7 så testades varje funktion och steg efter dess implementering.
Prestandamätningar genomfördes för att kvantifiera förbättringar jämfört med den tidigare manuella processen, med fokus på tidsåtgång, manuell arbetsinsats och kvalitetsmetrik. Resultaten från valideringen visade att systemet uppfyllde de ställda kraven för automatisering, pålitlighet och användarvänlighet. Specifika mätresultat presenteras i kapitel 5.

5
Resultat

Detta kapitel presenterar utfallet av den implementerade CI/CD/CT-processen för Itirub-systemet. Fokus ligger på systemets funktionalitet samt kvantifierbara förbättringar jämfört med den tidigare manuella processen.
Resultaten relaterar direkt till projektmålen från kapitel 1 och bygger på implementationen beskriven i metodkapitlet.
Kapitlet redogör först för systemets huvudfunktioner illustrerade i figur 5.1, följt av uppmätta tids- och kvalitetsförbättringar. Avslutningsvis diskuteras implementationsutmaningar och återstående utvecklingsmöjligheter.

5.1 Systemets funktionalitet

5.1.1 Automatiserad byggprocess och CI-flöde

Det utvecklade CI-skriptet automatiserar byggprocessen för alla sex Itirub-varianter, vilket eliminerar behovet av manuell konfiguration och byggning. Flödesschemat i Figur 5.1 visar den automatiserade processen, vilken består av fyra huvudfaser: integrationsfasen, byggfasen, artefaktfasen och testfasen.
Integrationsfasen initieras när en utvecklare kör CI-skriptet, vilket representeras i den övre delen av flödesschemat. Skriptet extraherar först Git-SHA som senare kommer att inkluderas i de byggda binärfilerna för spårbarhet. Därefter utförs en kontrollpunkt, illustrerad som en romb i flödesschemat, där systemet avgör om samtliga varianter ska byggas parallellt eller om endast en specifik variant ska konfigureras. Detta val påverkar det efterföljande processflödet, då ett svar av alla leder till exekvering av alla sex byggprocesser, medan ett svar för en specifik nod resulterar i konfiguration och byggning av endast den valda varianten. Detta alternativ mellan att starta processen för alla sex eller för en specifik var det som efterfrågades av företaget.
I byggfasen, visad i den andra delen av flödesschemat, sker själva kompileringen av källkoden för alla sex Itirub-varianter. Varje variant representerar en unik kombination av kommunikationsprotokoll, Ethernet, WiFi eller LoRa och sensortyp analog eller digital. För varje variant konfigureras `conf_board.h`-filen automatiskt med rätt parametrar från YAML-konfigurationsfilen, vilket eliminerar behovet av manuella konfigurationsändringar som tidigare kunde ta 15-20 minuter per variant. Flödesschemat visar tydligt hur systemet separerar byggflödena för att möjliggöra parallell byggning av alla varianter, vilket avsevärt reducerar den totala byggtiden.
Artefaktfasen, som illustreras i den tredje delen av flödesschemat, hanterar pro-

cesserna efter kompilering. När byggningen är färdigställd passerar de resulterande binärfilerna genom FOTA-konverteringssteget, där de formateras för trådlös distribution och berikas med Git-metadata för spårbarhet. Efter konverteringen laddas FOTA-filerna automatiskt upp till JFrog Artifactory, representerat som en blå romb i flödesschemat. Artifactory fungerar som ett centralt lager med en hierarkisk struktur för alla byggda firmwareversioner.
Den avslutande testfasen, visad i den nedre delen av flödesschemat, visar hur systemet automatiskt triggar Jenkins-jobb för att testa den nybyggda firmwaren. När Jenkins-jobben har initierats, körs Robot Framework-tester på fysiska testenheter som motsvarar de specifika varianterna. Testautomationen använder ett intelligent testflöde som automatiskt anpassar tester baserat på hårdvarukompatibilitet, vilket säkerställer att endast relevanta tester körs för varje variant.
Efter att CI-skriptet har startat ett testflöde så inväntar den och bevakar Jenkins för testets identifieringsnummer och skapar en URL-länk till det testflödet. Detta underlättar för utvecklare att följa de tester som startas och skapar en robust process när flera utvecklare eller testare startar tester samtidigt.
Det automatiserade systemet tillhandahåller följande funktionalitet. Automatisk konfigurering av variantspecifika definitioner i `conf_board.h`, vilket säkerställer konsekvent konfiguration för varje variant utan mänskliga fel. Systemet använder YAML-konfigurationen som single source of truth för alla variantspecifika inställningar. Dessvärre fanns det inget sätt att i Jenkins konfigurera jobbparametrar dynamiskt utefter YAML-filen Korrekt kompilering med variantspecifika flaggor och inställningar, styrd av konfigurationsfilen och implementerad genom skriptets byggprocess. Varje variant kompileras med exakt de inställningar som krävs för dess specifika hårdvara och kommunikationsprotokoll. Konvertering av binärer till FOTA-format med Git-metadata för spårbarhet, vilket möjliggör identifiering av exakt vilken källkodsversion som motsvarar en specifik binär. Detta är kritiskt för felsökning och versionshantering i distribuerade system. Möjlighet att bygga en specifik variant eller alla varianter automatiskt, vilket drastiskt reducerar den totala byggtiden jämfört med sekventiell byggning. Systemet eliminerar helt behovet av manuella konfigurationsändringar, vilket avsevärt minskar risken för mänskliga fel i processen.

5.1.2 Strukturerad artefakthantering

Integrationen med JFrog Artifactory, som visas i artefaktfasen i flödesschemat, har skapat en infrastruktur för hantering av byggda firmware-binärer. Artifactory, representerat som en blå romb i diagrammet, fungerar som ett centralt lager för byggda binärer, med organisation efter variant, Git-commit och versionsnummer.
Systemet tillhandahåller strukturerad lagring i hierarkiskt organiserade repositories, vilket möjliggör organisation efter olika kriterier som variant, version och byggdatum. Denna struktur förenklar sökning och återhämtning av specifika firmwareversioner.
Tydlig versionsspårning genom Git-SHA och versionsinformation skapar en direkt koppling mellan binärer och källkod. För varje binär lagras metadata som Git SHA, version, byggdatum och ansvarig utvecklare.
Automatisk uppladdning direkt från byggprocessen utan manuella steg eliminerar

risken för fel i artefakthanteringen och säkerställer att alla byggda binärer lagras konsekvent. Sömlös integration med testmiljön genom standardiserade URLs och nedladdningsmekanismer möjliggör automatisk hämtning av rätt firmware för test av varje variant.
Den implementerade arkitekturen i Artifactory möjliggör avancerade sökfunktioner som att hitta alla byggen för en specifik variant, alla byggen från en specifik commit, eller alla byggen med en viss version. Detta är värdefullt både i utvecklings- och felsökningssyfte, särskilt när problem behöver korreleras med specifika kodändringar.
Ett exempel på en typisk artefaktsökväg i systemet är:

```
1 /itirub-fota/rpi-itirub-eth-adc/2
    c07cca01dd5725920d8df4951d3ffff626fa593/
    Itirub_Ethernet_Digital_2c07cca01dd5.txt
```

Denna struktur innehåller all nödvändig information för att identifiera exakt vilken kod och konfiguration som användes för att skapa binären, vilket är kritiskt för spårbarhet i industriella system.

5.1.3 Effektiv FOTA-distribution

Systemet stödjer nu en fullständigt automatiserad FOTA-distributionsprocess, vilket är resultatet av integrationen mellan byggsystemet, Artifactory och testmiljön som visas i flödesschemat. FOTA-konverteringssteget i artefaktfasen säkerställer att binärer formateras korrekt för trådlös distribution med all nödvändig metadata.
Systemet tillhandahåller automatisk konvertering av binärer till FOTA-format med korrekt metadata för versionsidentifiering och säkerhetsverifiering, vilket säkerställer säker distribution av firmware. Strukturerad lagring i Artifactory med metadata för versionshantering möjliggör enkel identifiering och distribution av specifika firmwareversioner baserat på olika kriterier.
Testkontrollerad distribution till fysiska enheter via MQTT och andra kommunikationsprotokoll stöds för alla Itirub-varianter oavsett kommunikationsmetod. Validering av korrekt installation genom del-nummer-verifiering säkerställer att rätt firmware har installerats på rätt enhet och fungerar korrekt.
Denna infrastruktur kan direkt återanvändas för produktion, vilket möjliggör säker och effektiv uppdatering av enheter i fält. Systemet stödjer olika distributionstrategier som gradvisa utrullningar, A/B-testning eller fullständiga uppdateringar, beroende på behov och miljö. FOTA-processen har också optimerats för låg bandbredd, särskilt för LoRa-varianter där dataöverföring är begränsad, genom komprimering och effektiv kodning av firmware-data.

5.1.4 Förbättrad testautomation

Genom modifieringen av Jenkins-konfigurationen och utökningen av Robot Framework-testfilerna har testprocessen förbättrats avsevärt, vilket illustreras i testfasen av flödesschemat. Jenkins-konfigurationen har modifierats för att inkludera nya parametrar som FIRMWARE_URL och FOTA_FILENAME, vilket möjliggör direkt nedladdning av byggda firmware-filer från Artifactory utan manuell intervention.

Systemet tillhandahåller automatisk FOTA-uppdatering och verifiering på korrekt fysisk hårdvara, vilket säkerställer att varje firmware testas på exakt den hårdvaruvariant den är avsedd för. Variantspecifik testaktivering baserad på hårdvarukompatibilitet säkerställer att endast relevanta tester körs för varje variant och förhindrar falska testfel på grund av inkompatibla tester.
Testflödet med automatisk nedladdning, installation och verifiering hanterar hela processen från hämtning av firmware till verifiering av korrekt installation och funktion. Detta har reducerat testtiden avsevärt samtidigt som kvaliteten på testningen har förbättrats genom konsekvent och omfattande verifiering av alla varianter.
Tillagda parametrar i Jenkins-konfigurationen etablerar en direkt koppling mellan CI-processen och testautomationen, vilket möjliggör sömlös validering av nybyggda binärer. Det nya "Use Workspace Firmware-nyckelordet i Robot Framework-testfilen möjliggör automatiserad hämtning och installation utan manuell intervention, vilket är särskilt värdefullt för nattliga byggen och kontinuerlig testning.
Testprocessen har också förbättrats med avseende på rapportering och visualisering, med tydlig koppling mellan testresultat och den specifika firmwareversion som testades, inklusive Git-information och byggdatum.

5.2 Prestandaförbättringar

5.2.1 Tidsbesparingar

Jämfört med den ursprungliga manuella processen har implementationen av den automatiserade CI-processen som visas i flödesschemat resulterat i betydande tidsbesparingar. Den totala tid som krävs för en komplett build-test-cykel har reducerats från cirka fyra timmar till under två timmar, vilket ger en tidsvinst på över 50%.
Systemet eliminerar cirka 10 minuters manuell konfigurering per variant, vilket innebär en besparing på upp till 1 timme vid byggning av samtliga varianter. Den automatiserade konfigurationsprocessen tar nu endast några sekunder per variant.
Möjligheten att bygga samtliga sex varianter automatiserat, som tydligt illustreras i byggfasen av flödesschemat, kan spara cirka 2 timmar vid fullständig byggning jämfört med manuell byggning.
Automatiserad testning, representerad i testfasen av flödesschemat, reducerar tidsåtgången med ca 50%, från ca 2 timmar till under 1 timme. Detta beror på eliminering av flera manuella steg.
Automatisk nedladdning av firmware från Artifactory sparar uppskattningsvis 5-10 minuter per testtillfälle genom att eliminera manuell filöverföring och konfiguration. Ännu viktigare är att den aktiva tiden för utvecklare har minskats från flera timmar till några få minuter för konfiguration och initiering, eftersom resten av processen är helt automatiserad.

5.2.2 Kvalitetsförbättringar

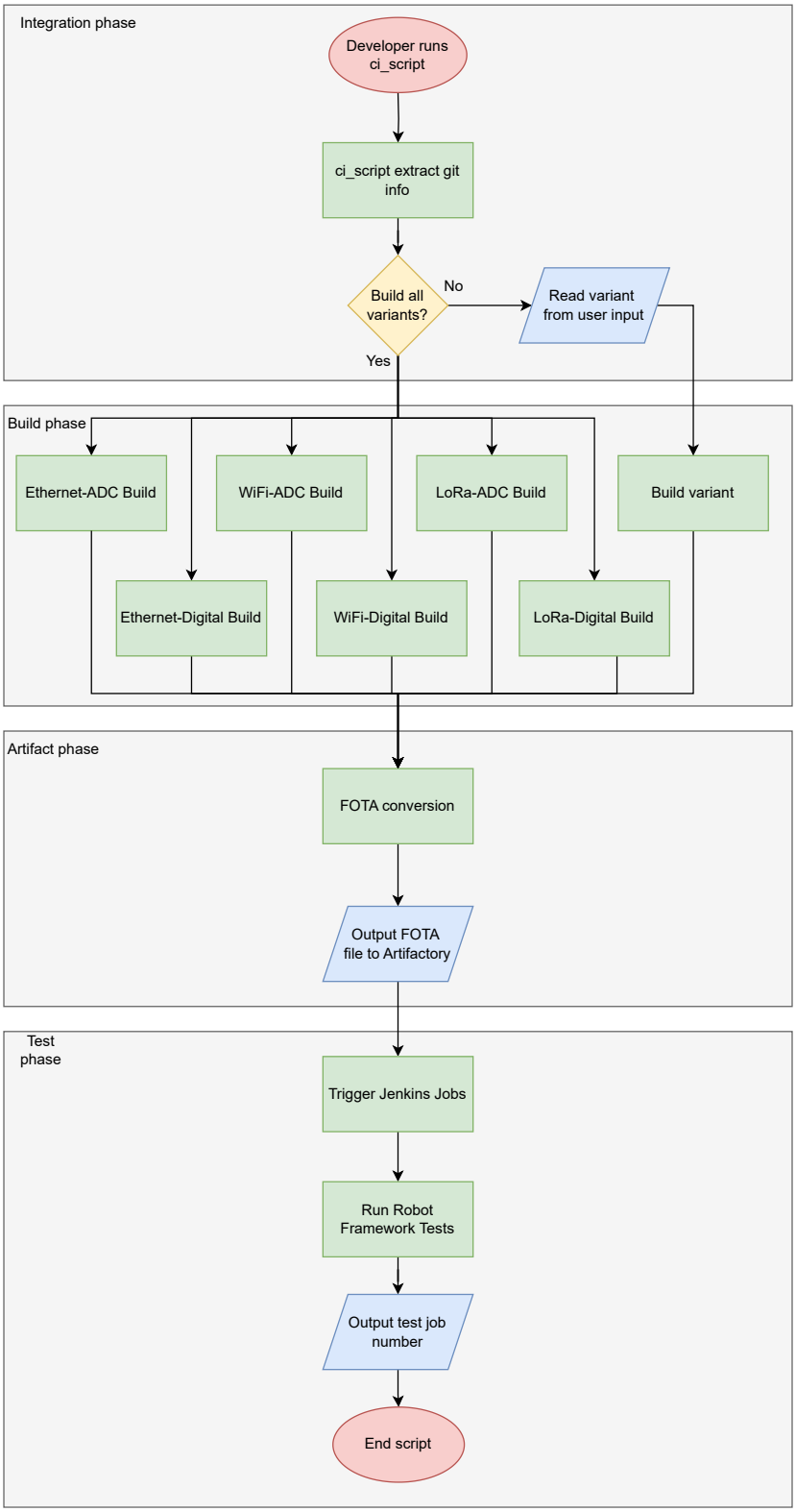
Automation har resulterat i förbättrad kvalitetssäkring, vilket är en konsekvens av den strukturerade processen som visas i flödesschemat. Den automatiserade byggprocessen och standardiserade konfigurationen från en enda källa till sanning har

minskat manuella konfigurationsfel.
Konsekvent och reproducerbar byggmiljö för alla varianter genom standardiserade byggskript och konfigurationer säkerställer att samma källkod alltid resulterar i identiska binärer oavsett vilken utvecklare som initierar bygget. Automatiserad verifiering av samtliga varianter vid kodändringar säkerställer att ändringar testas på alla berörda hårdvarukonfigurationer och förhindrar oavsiktliga regressioner på vissa varianter. Tidigare testades ofta endast en eller två varianter på grund av tidsbegränsningar.
Spårbarhet från kodbas till specifik binär via Git-SHA, implementerad i artefaktfasen av flödesschemat, möjliggör exakt identifiering av vilken kodversion som finns i en specifik firmware. Detta är kritiskt vid felsökning av problem i distribuerade enheter. Standardiserad testprocess med konsekvent resultatrapportering ger tydligare insyn i testresultat och möjliggör trendanalys över tid för att identifiera mönster i testfel.
Ett särskilt viktigt resultat är att förändringar nu kan valideras på samtliga hårdvaruvarianter, vilket kraftigt reducerar risken för oväntade fel i produktionen. Automatisk validering av installerad firmware genom modifieringarna i Robot Frameworktestfilen säkerställer också att korrekt binär har installerats, vilket eliminerar en vanlig felkälla i den tidigare processen där fel firmware ibland testades på grund av manuell förväxling.

5.3 Uppnådda projektmål och systemets robusthet

Implementationen har lyckats lösa samtliga identifierade problemområden från projektets inledning. Den ursprungliga frågeställningen om hur en automatiserad CI/CT/CD-process kan implementeras för Itirub-plattformen har besvarats genom utveckling av ett fullständigt funktionellt system som eliminerar manuella processteg och förbättrar både effektivitet och kvalitet.
Genomförandet resulterade i en robust lösning genom systematisk utveckling i fyra iterationer, där varje iteration byggde vidare på tidigare funktionalitet. Detta iterativa tillvägagångssätt möjliggjorde både tidig värdeleverans och effektiv hantering av tekniska utmaningar, särskilt relaterade till tredjepartsintegrationer som Artifactory och Jenkins.
Systemets tekniska arkitektur byggdes på genomtänkta designval som optimerades för långsiktig underhållbarhet. Python valdes som implementationsspråk för dess plattformsoberoende natur och lättlästa syntax, vilket gör systemet underhållbart av flera utvecklare. YAML valdes som konfigurationsformat för dess mänskligt läsbara syntax och hierarkiska struktur, vilket förenklar framtida expansion med nya varianter.
Kvalitetssäkringen implementerades genom flera lager av validering. Kodgranskning genomfördes för alla kritiska komponenter, med fokus på felhantering och komponentintegration. Manuell validering av hela flödet genomfördes efter varje iteration för att säkerställa korrekt integration mellan faser.
Systemets modulära arkitektur möjliggör framtida utveckling och skalning. Den

centraliserade konfigurationsmodellen förenklar addition av nya varianter genom enkel uppdatering av YAML-filen utan kodändringar. Abstraktion av variantspecifika tester möjliggör enkel addition av nya testtyper, medan standardiserade gränssnitt mellan komponenter underlättar framtida integration med ytterligare externa system.
De särskilda tekniska utmaningar som framgångsrikt löstes inkluderade dynamisk konfigurationshantering för robust manipulation av konfigurationsfiler, integration med externa system där API-krav och autentisering var mer komplicerat än förväntat, samt installation och konfiguration av Artifactory i företagsmiljön.
Resultatet är en transformation av firmwareutvecklingen från en manuell, tidskrävande och felbenägen process till ett automatiserat, standardiserat och pålitligt flöde som säkerställer hög kvalitet och full spårbarhet mellan källkod och distribuerad firmware.



Figur 5.1: A flowchart of the CI/CT/CD process

6
Slutsats

Detta kapitel sammanfattar projektets resultat, diskuterar hur de ursprungliga målen har uppfyllts och pekar på framtida utvecklingsmöjligheter. Här reflekteras också över projektets bidrag till området för firmwareutveckling och IoT-sensorsystem.

6.1 Resumé av arbetet

Detta examensarbete har presenterat design och implementation av en automatiserad CI/CT/CD-lösning för firmwareutveckling och testning av IoT-sensornoder vid ZESS AB. Projektet har framgångsrikt adresserat de utmaningar som identifierades i den befintliga utvecklingsprocessen genom utveckling av ett automationssystem som integrerar alla aspekter av firmwareutveckling från kodändring till verifierad distribution.

6.1.1 Uppfyllelse av projektmål

Projektet har uppnått fyra av de fem uppsatta målen. De största effekterna har varit förbättrad automation, bättre spårbarhet och möjligheten att verifiera firmware på ett mer systematiskt sätt genom testfall.
Följande mål har uppnåtts:

- Firmware byggs automatiskt för samtliga hårdvaruvarianter vid ändringar i koden.
- En automatiserad mekanism för att identifiera rätt hårdvaruvariant och säkerställa korrekt installation av binärfiler har implementerats.
- Variantspecifika tester körs automatiskt via CI/CT-processen.
- Firmware distribueras automatiskt via OTA-uppdateringar.

Däremot har målet att skapa en robust återställningsmekanism där main-firmware automatiskt återställs vid test- och firmwarefel inte uppnåtts. Funktionen visade sig vara för komplex att implementera inom projektets tidsram. Återställningsmekanismen krävde ytterligare integration mellan flera komponenter, inklusive testlogik, firmwareöverföring och byggstatushantering, vilket i slutändan prioriterades bort till förmån för övriga, mer realistiska mål.

6.1.2 Effekter på utvecklingsprocessen

Införandet av CI-flödet har lett till flera positiva förändringar. Tidigare manuella moment har automatiserats, till exempel:

- Konfiguration av firmware sker automatiskt via ändringar i en gemensam header-fil.
- Firmware byggs automatiskt beroende på vald variant.
- Binärfiler konverteras till OTA-format och laddas upp till Artifactory utan manuell inblandning.
- Systemtestfall körs automatiskt, och resultaten rapporteras tillbaka till utvecklingsteamet.

Detta har lett till minskade ledtider och färre manuella fel. CI-processen har blivit mer förutsägbar och skalbar, även om vissa delar – som manuell initiering av Jenkinsjobbet – fortfarande kvarstår.
Projektet har också påverkat tidplanen. Komplexiteten i testautomatisering och miljöintegration visade sig vara högre än förväntat, vilket ledde till att funktioner som återställningsmekanismen fick prioriteras bort i senare iterationer.

6.2 Kritisk diskussion

6.2.1 Projektets styrkor

Den implementerade lösningen uppvisar flera betydande styrkor som bidrar till dess framgång och funktionalitet. Den modulära arkitekturen möjliggör flexibel utveckling och enkel underhållning, vilket är kritiskt för långsiktig hållbarhet. Valet av Python som implementationsspråk och YAML som konfigurationsformat har visat sig vara väl avvägda beslut som förbättrar både läsbarhet och underhållbarhet.
Den systematiska iterativa utvecklingsprocessen möjliggjorde kontinuerlig värdeleverans och effektiv hantering av oförutsedda tekniska utmaningar. Detta tillvägagångssätt visade sig särskilt värdefullt vid integrering med externa system som Artifactory och Jenkins, där komplexiteten var större än initialt förväntat. Installationen av Artifactory tog en betydande mängd tid då dokumentationen av open source versionen var otillräcklig. Hur databasen som ligger under Artifactory konfigureras var otydligt och svårare än väntat men när konfigurationen gjordes korrekt så startade systemet upp. Den iterativa processen tillät resten av CI-skriptet fungera och utvecklas vidare medans lösningen för artifactory arbetades på.
Systemets robusthet visas genom den omfattande kvalitetssäkringen med kodgranskning och manuell validering. Den centrala konfigurationsfilen som “single source of truth” har effektivt formaliserat tidigare implicit kunskap och eliminerat risken för inkonsekvent varianthantering.

6.2.2 Identifierade svagheter och utmaningar

En begränsning i den aktuella lösningen är att vissa delar fortfarande är beroende av manuell inblandning. Till exempel krävs det att en utvecklare manuellt initierar körningen av CI-skriptet. Detta är ett avsteg från ett fullständigt CI-flöde och gör systemet mindre effektivt än det annars hade kunnat vara.
Denna svaghet är dock främst en konsekvens av företagets befintliga struktur, där firmware och testfall hanteras i separata repositories. Denna uppdelning försvårar automatisk sammankoppling av bygg- och testflöden utan fler utomstående system.

Ytterligare en svaghet är att om något går fel under en bygg-testprocess för en variant så finns det inget sätt att återanvända de delar som lyckades. Processen måste då alltså startas om helt från början för den variant som misslyckades.

6.2.3 Återstående tekniska begränsningar

Flera tekniska begränsningar har identifierats som kräver framtida åtgärder. Hantering av nätverksfel vid Artifactory-access har grundläggande återförsökningslogik, men mer sofistikerade algoritmer skulle förbättra systemets robusthet vid längre nätverksstörningar.
Återhämtning vid misslyckade byggen kan förbättras genom bättre isolering och diagnostisering av specifika byggproblem. Nuvarande implementation hanterar hela byggcykeln som en enhet, vilket kan vara ineffektivt när endast en variant misslyckas.
Testrapporteringsformatet, även om funktionellt, ger begränsad insikt vid komplexa felsituationer eller när flera tester misslyckas samtidigt. Detta begränsar förmågan att snabbt identifiera och åtgärda systematiska problem.

6.3 Utvärdering av metod och planering

6.3.1 Metodval och genomförande

Den valda iterativa utvecklingsmetoden visade sig vara väl lämpad för projektets karaktär. Möjligheten att leverera funktionalitet stegvis och anpassa sig till tekniska utmaningar var kritisk för projektets framgång. Den fyra-iterations-strukturen möjliggjorde systematisk progression från grundläggande automation till avancerad integration.
Valet att prioritera funktionalitet före optimering i de tidiga iterationerna var effektivt och möjliggjorde tidig validering av systemkonceptet. Detta tillvägagångssätt identifierade potentiella integrationsproblem tidigt och möjliggjorde anpassning av senare iterationer.
Designbeslutet att implementera ett modulärt system med tydliga gränssnitt mellan komponenter visade sig vara värdefullt när oförutsedda tekniska utmaningar uppstod, särskilt vid tredjepartsintegrationer.

6.3.2 Avvikelser från ursprunglig planering

Flera betydande avvikelser från den ursprungliga planeringen uppstod under projektets gång. Artifactory-installation och konfiguration krävde betydligt mer tid än förväntat på grund av företagets infrastrukturkrav och säkerhetsrestriktioner. Detta påverkade tidsplanen för senare iterationer och krävde omprioritering av funktionalitet.
Jenkins-integrationen visade sig vara mer komplex än initialt bedömt, särskilt gällande parameterisering och bakåtkompatibilitet. Den ursprungliga planen underestimerade komplexiteten i att modifiera befintliga testsystem utan att störa pågående utvecklingsarbete.

Utvecklingen av variantspecifik konfigurationshantering krävde mer iterationer än planerat för att hantera alla specialfall och säkerställa robust filmodifiering. Komplexiteten i att automatisera tidigare manuella processer var större än förväntat.

6.3.3 Generaliserbara lärdomar för liknande projekt

Projektet har genererat flera viktiga lärdomar som är tillämpliga för liknande automationsprojekt inom firmware-utveckling. Betydelsen av tidig validering av tredjepartsintegrationer kan inte överskattas - komplexiteten i externa system är ofta större än dokumentationen indikerar och kräver dedikerad tid för utforskning och testning.

- Implementation av automation i befintliga utvecklingsmiljöer kräver noggrann hänsyn till bakåtkompatibilitet och gradvis övergång. Förändringar medför risker för att störa kritiska utvecklingsprocesser, medan evolutionära förändringar möjliggör smidigare adoption.
- Centraliserad konfiguration är kritisk för att hantera komplexitet i system med många varianter, men designen av konfigurationsstrukturen kräver noggrann planering för att balansera läsbarhet, flexibilitet och underhållbarhet.
- Kvalitetssäkring av automationskod är lika viktig som kvalitetssäkring av produktionskod, särskilt när automationen blir kritisk för utvecklingsprocessen. Kodgranskning och systematisk validering är nödvändiga för att säkerställa tillförlitlighet.

6.4 Framtida utveckling och forskning

6.4.1 Vidareutveckling av nuvarande system

Systemet har lagt en solid grund för framtida förbättringar som skulle ytterligare öka dess värde och tillämpbarhet. Följande punkter är förslag på framtida utveckling av systemet.

- Utökat stöd för kontinuerlig distribution till fältenheter, baserat på samma infrastruktur som används för testnoder, skulle möjliggöra säker och automatiserad uppdatering av enheter i produktion.
- Integration med ytterligare testverktyg för mer omfattande testning, inklusive automatiserad prestandaanalys och säkerhetstestning, skulle förbättra kvalitetssäkringen och identifiera potentiella problem som inte täcks av nuvarande funktionella tester.
- Förbättrad visualisering och analys av testdata för att identifiera trender och mönster i testresultat över tid skulle möjliggöra förebyggande kvalitetsförbättring och identifiering av systematiska problem innan de påverkar produktionen.
- Konfigurera Jenkins att använda YAML-filen som input för vilka parametrar det går att välja i start av test.

6.4.2 Tekniska förbättringsområden

Flera specifika tekniska förbättringar har identifierats som skulle öka systemets robusthet och användbarhet.

- Implementation av avancerade återförsöksstrategier med exponentiell backoff och jitter för Artifactory-access, tillsammans med caching-mekanismer för att minska beroendet av kontinuerlig nätverksåtkomst.
- Förbättrad isolering av byggprocesser och mer detaljerad felrapportering skulle möjliggöra bättre diagnostisering av byggproblem och stöd för selektiv ombyggnad av endast misslyckade varianter.
- Standardisering av filnamns- och URL-format med omfattande validering och förbättrad kodning för att hantera specialfall skulle öka systemets robusthet och minska risken för fel vid komplexa scenarier.
- Lagring av testresultat tillsammans med firmware binärer i Artifactory för att ha mer detaljer om testningen av den mjukvaran.

6.4.3 Forskningsområden och bredare tillämpning

Projektet öppnar för flera intressanta forskningsområden inom automation av firmwareutveckling.

- Utveckling av intelligenta teststrategier som automatiskt anpassar testtäckning baserat på kodändringar och historisk testdata skulle kunna förbättra både effektivitet och kvalitet.
- Undersökning av maskininlärningsbaserade metoder för förutsägelse av byggfel och optimering av byggordning baserat på historiska data och kodändringar representerar ett spännande forskningsområde.
- Integration av säkerhetsanalys och sårbarhetsscanning direkt i CI/CD-pipelinen för firmware-utveckling är ett växande behov inom IoT-utveckling som skulle kunna dra nytta av fortsatt forskning.
- Utökning av automationsprinciperna till andra aspekter av inbyggd systemutveckling, såsom automatiserad hårdvarutestning och validering av elektromagnetisk kompatibilitet, representerar potentiella framtida forskningsriktningar.

6.5 Avslutande reflektion

Den utvecklade CI/CT/CD-lösningen demonstrerar hur automation kan förbättra produktiviteten och möjliggöra effektivare testning inom industriella IoT-ekosystem. Metoderna och lösningarna som implementerats i detta projekt kan användas som modell för liknande automatiseringsprojekt inom firmware-utveckling för embedded system.

Projektet visar också värdet av DevOps-principer tillämpade på traditionellt hårdvarunära utveckling, där automatisering historiskt sett har varit mer begränsad än i ren mjukvaruutveckling. Genom att överbrygga klyftan mellan mjukvaru- och hårdvaruutveckling möjliggör denna typ av automation snabbare innovation och högre kvalitet i alltmer komplexa inbyggda system.

Den balanserade utvärderingen av projektets styrkor och svagheter understryker vikten av systematisk planering, iterativ utveckling och kontinuerlig kvalitetssäkring i automationsprojekt. Samtidigt belyser de identifierade utmaningarna och begränsningarna områden där framtida forskning och utveckling kan bidra till ännu mer robusta och effektiva lösningar inom firmware-utveckling.

Litteratur

[1] E. Soares, G. Sizilio, J. Santos, D. A. da Costa och U. Kulesza, "The effects of continuous integration on software development: a systematic literature review," *Empirical Software Engineering*, årg. 27, nr 3, maj 2022, ISSN: 15737616. DOI: 10.1007/s10664-021-10114-1.

[2] E. Naresh, S. V. Murthy, N. Sreenivasa, S. Merikapudi och C. R. Rakhi Krishna, "Continuous Integration, Testing Deployment and Delivery in Devops," i *2024 International Conference on Knowledge Engineering and Communication Systems, ICKECS 2024*, Institute of Electrical och Electronics Engineers Inc., 2024, ISBN: 9798350359688. DOI: 10.1109/ICKECS61492.2024.10616918.

[3] Martin Fowler, *Continuous Integration*, jan. 2024.

[4] Adam Auerbach, *Part of the Pipeline: Why Continuous Testing Is Essential*, aug. 2015.

[5] S. Nandgaonkar och V. Khatavkar, *CI-CD Pipeline For Content Releases*, okt. 2022. DOI: 10.1109/GCAT55367.2022.9972129. URL: https://research.ebsco.com/linkprocessor/plink?id=9d98e95b-1cc3-306a-9bc2-0432efb7d3e4.

[6] M. Kubascik, I. A. Tupy, J. Sumsky och T. Baca, "OTA firmware updates on ESP32 based microcontrolers," i *2024 IEEE 17th International Scientific Conference on Informatics, INFORMATICS 2024 - Proceedings*, Institute of Electrical och Electronics Engineers Inc., 2024, s. 185–189, ISBN: 9798350387674. DOI: 10.1109/Informatics62280.2024.10900824.

[7] ZessAB, *Industrial IoT Sensor Nodes*, maj 2025.

[8] *IEEE Standard for Ethernet*, Piscataway, NJ, USA, maj 2022. DOI: 10.1109/IEEESTD.2022.9844436.

[9] LitePoint, "IEEE 802.11ac: What Does it Mean for Test?" Tekn. rapport, 2013. URL: https://web.archive.org/web/20140816132722/http://litepoint.com/whitepaper/80211ac_Whitepaper.pdf.

[10] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui och T. Watteyne, "Understanding the Limits of LoRaWAN," *IEEE Communications Magazine*, årg. 55, nr 9, s. 34–40, 2017, ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1600613.

[11] Adafruit, *ADS1115 16-Bit ADC - 4 Channel with Programmable Gain Amplifier - STEMMA QT / Qwiic*, jan. 2023.

[12] Python Software Foundation, *Python About*, maj 2025.

[13] Jenkins, *Jenkins User Documentation*, maj 2025.

[14] JFrog, *What is Artifactory*, maj 2025.

[15] Robot Framework Foundation, *Robot Framework documentation*, maj 2025.

A

Appendix 1: Komplett CI-skript

```python
import os
import logging
import subprocess
import re
import time
import dotenv
import jenkins
import requests
import requests.auth
import yaml

# Load environment variables
dotenv.load_dotenv()

# The following are settings as enviromentable variables. They have a default value but can be overridden in a .env file
# Configuration constants from a single source of truth
CONFIG_FILE = os.getenv("CONFIG_FILE", "./itirub_config.yaml")
# Configuration from environment variables
ARTIFACTORY_URL = os.getenv("ARTIFACTORY_URL", "http://192.168.254.145:8082/artifactory")
ARTIFACTORY_REPO = os.getenv("ARTIFACTORY_REPO", "itirub-fota")
ARTIFACTORY_USER = os.getenv("ARTIFACTORY_USER")        # Users username in artifactory
ARTIFACTORY_TOKEN = os.getenv("ARTIFACTORY_TOKEN")      # Token created in artifactory
JENKINS_URL = os.getenv("JENKINS_URL", "http://192.168.254.145:8080")
JENKINS_USER = os.getenv("JENKINS_USER")                # Users username in jenkins
JENKINS_TOKEN = os.getenv("JENKINS_TOKEN")              # Token created in jenkins
JENKINS_JOB_NAME = os.getenv("JENKINS_JOB_NAME", "itirub_nightly")
# Tool paths (will be combined with project root)
BUILD_SCRIPT_REL_PATH = os.getenv("BUILD_SCRIPT_PATH", "AtmelStudioProject\BuildAtmelStudioProject.bat")
ITIRUB_FOTA_CREATOR_REL_PATH = os.getenv("ITIRUB_FOTA_CREATOR_PATH", "tools\ItirubFotaCreator")
# Logging configuration
LOG_LEVEL = os.getenv("LOG_LEVEL", "INFO")

def setup_logging(level=LOG_LEVEL):
    """Configure logging with user-specified level"""
    numeric_level = getattr(logging, level.upper(), None)
```

```
    if not isinstance(numeric_level, int):
        raise ValueError(f"Invalid log level: {level}")
    logging.basicConfig(
        level=numeric_level,
        format='%(asctime)s - %(levelname)s - %(message)s'
    )
    return logging.getLogger(__name__)

# Initialize logger
logger = setup_logging()

def load_config():
    """Load configuration from YAML file - single source of truth
    """
    try:
        with open(CONFIG_FILE, 'r') as file:
            config = yaml.safe_load(file)
            logger.info(f"Configuration loaded from {CONFIG_FILE}")
            return config
    except FileNotFoundError:
        logger.erorr(f"Configuration file {CONFIG_FILE} not found."
    )
        raise

# Load configuration
CONFIG = load_config()

def get_variant_defines(variant):
    """Get board and app defines for a specific variant from
    configuration file"""
    variant_config = CONFIG["variants"].get(variant, {})
    return {
        "BOARD_DEFINE": variant_config.get("BOARD_DEFINE"),
        "APP_DEFINE": variant_config.get("APP_DEFINE")
    }

def apply_test_flags(variant):
    """Get test flags for a specific variant from configuration
    file"""
    test_flags = CONFIG["default_tests"].copy()
    variant_tests = CONFIG["variants"].get(variant, {}).get("tests"
    , {})
    test_flags.update(variant_tests)
    logger.info(f"Test flags for {variant}: {test_flags}")
    return test_flags

def get_conf_board_path(project_root):
    """Return full path to conf_board.h from project root"""
    return os.path.join(project_root, "AtmelStudioProject", "src",
    "config", "conf_board.h")

def update_conf_board(project_root, variant):
    """Update configuration based on variant"""
    conf_board_path = get_conf_board_path(project_root)
    defines = get_variant_defines(variant)

```

```
86      if not defines or not defines["BOARD_DEFINE"] or not defines["
    APP_DEFINE"]:
87          raise ValueError(f"Unknown variant: {variant}")
88
89      board_define = defines["BOARD_DEFINE"]
90      app_define = defines["APP_DEFINE"]
91
92      logger.info(f"Updating conf_board.h for {variant} with {
    board_define} and {app_define}")
93
94      with open(conf_board_path, 'r') as file:
95          lines = file.readlines()
96
97      # Comment out all board and app defines
98      new_lines = []
99      for line in lines:
100         if re.search(r'#define\s+BOARD_WITH_(IOEXPANDER|ADS1115_ADC
    )', line) or re.search(r'#define\s+BUILD_(WIFI|LORA|ETHERNET)
    _APP', line):
101             if not line.strip().startswith('//'):
102                 new_lines.append(f"//{line}")
103             else:
104                 new_lines.append(line)
105         else:
106             new_lines.append(line)
107
108     # Uncomment or add the needed defines
109     for i, line in enumerate(new_lines):
110             if "#define " + board_define in line:
111                 new_lines[i] = f"#define {board_define}\n"
112             if "#define " + app_define in line:
113                 new_lines[i] = f"#define {app_define}\n"
114
115     # Write the updated content back
116     with open(conf_board_path, 'w') as file:
117         file.writelines(new_lines)
118
119     logger.info(f"Updated {conf_board_path}")
120
121 def get_git_info():
122     """Get Git SHA and version information"""
123     try:
124         git_sha = subprocess.check_output(["git", "rev-parse", "
    HEAD"], text=True).strip()
125         try:
126             git_version = subprocess.check_output(["git", "describe
    ", "--tags"], text=True).strip()
127         except subprocess.CalledProcessError:
128             # If no tags available
129             git_version = git_sha[:8]
130     except subprocess.CalledProcessError:
131         logger.warning("Unable to get git information. Using
    placeholders.")
132         git_sha = "unknown"
133         git_version = "unknown"
134
```

```
135     logger.info(f"Git SHA: {git_sha}, Version: {git_version}")
136     return git_sha, git_version
137 
138 def build_firmware(project_root, variant):
139     """Build firmware for the specified variant"""
140     build_script = os.path.join(project_root, BUILD_SCRIPT_REL_PATH
    )
141     logger.info(f"Building firmware for {variant} using {
    build_script}")
142 
143     try:
144         # Use shell=True for batch files on Windows
145         result = subprocess.run(f'"{build_script}"', shell=True,
    capture_output=False, text=True, cwd="C:\Projects\itirub\
    AtmelStudioProject")
146         if result.returncode != 0:
147             logger.error(f"Build failed: {result.stderr}")
148             raise RuntimeError("Build failed")
149 
150         logger.debug(f"Build output: {result.stdout}")
151         logger.info(f"Build successful for {variant}")
152 
153         # Find the binary file path from the build output
154         binary_path = None
155         for root, dirs, files in os.walk(project_root+"\
    AtmelStudioProject\Debug"):
156             if ".bin" in files:
157                 binary_path = os.path.join(root, files)
158 
159         if not binary_path:
160             raise FileNotFoundError("Binary file not found in build
     output")
161 
162         logger.info(f"Binary file found: {binary_path}")
163         return binary_path
164     except Exception as e:
165         logger.error(f"Error during build: {e}")
166         raise
167 
168 def convert_to_fota(project_root, binary_path, variant, git_sha,
    git_version):
169     """Convert binary firmware to FOTA text file format"""
170     fota_creator = os.path.join(project_root,
    ITIRUB_FOTA_CREATOR_REL_PATH)
171     logger.info(f"Converting firmware to FOTA format for {variant}"
    )
172 
173     try:
174         # Run the FOTA creator tool with git info
175         cmd = [
176             fota_creator, binary_path,
177         ]
178 
179         result = subprocess.run(cmd, capture_output=True, text=True
    )
180         if result.returncode != 0:
```

```
            logger.error(f"FOTA conversion failed: {result.stderr}")
            raise RuntimeError("FOTA conversion failed")

        logger.debug(f"FOTA conversion output: {result.stdout}")

        # Extract filename from output
        fota_filename = None
        for line in result.stdout.splitlines():
            if "Output file:" in line or "Firmware written to" in line:
                fota_filename = line.split(":")[-1].strip()
                break

        if not fota_filename:
            raise FileNotFoundError("FOTA file path not found in conversion output")

        logger.info(f"FOTA conversion successful. File: {fota_filename}")
        return fota_filename
    except Exception as e:
        logger.error(f"Error during FOTA conversion: {e}")
        raise

def upload_to_artifactory(fota_file_path, variant, git_sha):
    """Upload FOTA file to Artifactory"""
    fota_filename = os.path.basename(fota_file_path)
    # Include git SHA in path for traceability
    artifact_path = f"{variant}/{git_sha}/{fota_filename}"
    upload_url = f"{ARTIFACTORY_URL}/{ARTIFACTORY_REPO}/{artifact_path}"

    logger.info(f"Uploading {fota_file_path} to Artifactory: {upload_url}")

    try:
        with open(fota_file_path, 'rb') as f:
            response = requests.put(
                upload_url,
                data=f,
                auth=requests.auth.HTTPBasicAuth(ARTIFACTORY_USER, ARTIFACTORY_TOKEN),
                headers={
                    "X-Git-SHA": git_sha,  # Add SHA as header for reference
                    "X-Variant": variant    # Add variant as header for reference
                }
            )

        if response.status_code in (200, 201):
            logger.info(f"Uploaded to Artifactory: {upload_url}")
            return upload_url, fota_filename
        else:
            logger.error(f"Upload failed: {response.status_code} -
```

```
    {response.text}")
228             return None, None
229     except Exception as e:
230         logger.error(f"Error uploading to Artifactory: {e}")
231         raise
232
233 def trigger_jenkins_fota_test(variant, fota_url, fota_filename,
    git_sha):
234     """Trigger Jenkins job to run FOTA test"""
235     test_flags = apply_test_flags(variant)
236
237     parameters = {
238         "NODE_LABEL": variant,
239         "FIRMWARE_URL": fota_url,
240         "FOTA_FILENAME": fota_filename,
241         "GIT_SHA": git_sha,
242         "BRANCH": git_sha,
243         **test_flags
244     }
245
246     logger.info(f"Triggering Jenkins FOTA test for {variant} with
    parameters: {parameters}")
247
248     try:
249         server = jenkins.Jenkins(
250             JENKINS_URL,
251             username=JENKINS_USER,
252             password=JENKINS_TOKEN
253         )
254
255         queue_id = server.build_job(JENKINS_JOB_NAME, parameters=
    parameters)
256
257         logger.info(f"Jenkins FOTA test job triggered for {variant}
     (Queue ID: {queue_id})")
258
259         timeout = 0
260         while timeout < 10:
261             timeout += 1
262             time.sleep(1)
263             try:
264                 build_id = requests.get(
265                     f"{JENKINS_URL}/queue/item/{queue_id}/api/json"
    ,
266                     auth=requests.auth.HTTPBasicAuth(JENKINS_USER,
    JENKINS_TOKEN)
267                     ).json()["executable"]["number"]
268             except:
269                 continue
270             if build_id != None:
271                 logger.info(f"URL to build: {JENKINS_URL}/job/
    itirub_nightly/{build_id}")
272                 return True
273
274         logger.info(f"Couldnt get build ID. (Queue ID: {queue_id})"
    )
```

```
275         return True
276     except Exception as e:
277         logger.error(f"Error triggering Jenkins FOTA test job: {e}"
    )
278         raise
279 
280 def prompt_for_variant():
281     """Prompt user to select variant"""
282     print("Available sensor variants:")
283     for variant in CONFIG["variants"]:
284         print(f" - {variant}")
285     print(" - all")
286     while True:
287         user_input = input("\nEnter sensor variant to build (or '
    all'): ").strip()
288         if user_input == "all" or user_input in CONFIG["variants"]:
289             return user_input
290         print("Invalid variant. Please try again.")
291 
292 def main():
293     """Main function for CI process"""
294     # Set logging level
295     global logger
296     logger = setup_logging()
297 
298     # Get project root path
299     project_root = input("Enter path to itirub project root: ").
    strip()
300     if not os.path.exists(project_root):
301         logger.error(f"Project root path does not exist: {
    project_root}")
302         return False
303 
304     # Check if configuration file exists
305     if not os.path.exists(CONFIG_FILE):
306         print(f"Configuration file {CONFIG_FILE} not found. Create
    an example?")
307 
308     # Get git information
309     git_sha, git_version = get_git_info()
310 
311     # Select variant
312     variant_input = prompt_for_variant()
313     variants = list(CONFIG["variants"].keys()) if variant_input ==
    "all" else [variant_input]
314 
315     results = {}
316     for variant in variants:
317         logger.info(f"\nStarting CI process for variant: {variant}"
    )
318         try:
319             # Update configuration
320             update_conf_board(project_root, variant)
321 
322             # Build firmware
323             binary_path = build_firmware(project_root, variant)
```

```
324
325             # Convert to FOTA format with git information
326             fota_file = convert_to_fota(project_root, binary_path,
    variant, git_sha, git_version)
327
328             # Upload to Artifactory with SHA in path
329             fota_url, fota_filename = upload_to_artifactory(
    fota_file, variant, git_sha)
330
331             # Trigger Jenkins job for FOTA testing with SHA
    verification
332             trigger_jenkins_fota_test(variant, fota_url,
    fota_filename, git_sha)
333
334             results[variant] = "Success"
335         except Exception as e:
336             logger.error(f"CI process failed for {variant}: {e}")
337             results[variant] = f"Failed: {e}"
338
339     # Print summary
340     print("\nCI Summary:")
341     for v, r in results.items():
342         print(f" - {v}: {r}")
343
344     # TODO start monitoring jenkins to get build ID
345
346 if __name__ == "__main__":
347     main()
```

Listing A.1: Komplett CI-skript för automatiserad byggprocess och testning

B

Appendix 2: Jenkins Pipeline-konfiguration

```
1
2  pipeline {
3      environment{
4          FIRMWARE_PATH = "${env.WORKSPACE}/firmware"
5      }
6      agent {
7          // Placeholder to select the node based on user input
8          label "${params.NODE_LABEL}"
9      }
10     options {
11         lock(label: "${params.NODE_LABEL}", quantity: 1, resource:
       null)
12     }
13
14     // Define initial parameters that will be overridden in the
       initialization stage
15     parameters {
16         // Choice parameter to select the node
17         choice(
18             name: 'NODE_LABEL',
19             choices: ['rpi-itirub-eth-adc', 'rpi-itirub-eth-digital
       ', 'rpi-itirub-wifi-adc', 'rpi-itirub-wifi-digital', 'rpi-itirub
       -lora-adc', 'rpi-itirub-lora-digital'],
20             description: 'Select the node where the pipeline should
        run'
21         )
22         booleanParam(name: 'RUN_FOTA_TEST', defaultValue: false,
       description: 'Enable to run fota.robot')
23         booleanParam(name: 'RUN_MQTT_SUB_TEST', defaultValue: false
       , description: 'Enable to run mqtt_sub_test.robot')
24         booleanParam(name: 'RUN_SNTP_TEST', defaultValue: false,
       description: 'Enable to run sntp_test.robot')
25         booleanParam(name: 'RUN_PARTNUMBER_TEST', defaultValue:
       false, description: 'Enable to run partnumber_test.robot')
26         booleanParam(name: 'RUN_INCOMING_DATA_TEST', defaultValue:
       false, description: 'Enable to run incoming_data_test.robot')
27         booleanParam(name: 'RUN_UNSENT_DATA_STORAGE_TEST',
       defaultValue: false, description: 'Enable to run
       data_storage_test.robot')
28         booleanParam(name: 'RUN_WIFI_TEST', defaultValue: false,
       description: 'Enable to run wifi_test.robot')
```

```
29          booleanParam(name: 'RUN_DIGITAL_TEST', defaultValue: false,
     description: 'Enable to run digital.robot')
30          booleanParam(name: 'RUN_ANALOG_SENSOR_TEST', defaultValue:
    false, description: 'Enable to run analog_sensor_test.robot')
31          booleanParam(name: 'RUN_DIGITAL_SENSOR_TEST', defaultValue:
     false, description: 'Enable to run digital_sensor_test.robot')
32          booleanParam(name: 'RUN_ITIRUB_RELAY_TEST', defaultValue:
    false, description: 'Enable to run itirub_relay_test.robot')
33          string(name: 'FIRMWARE_URL', defaultValue: '', description:
     'URL to the FOTA firmware in Artifactory')
34          string(name: 'GIT_SHA', defaultValue: '', description: 'Git
     SHA of the commit that produced the firmware')
35      }
36      stages {
37          stage('Download Firmware') {
38              when {
39                  expression { return params.FIRMWARE_URL != '' }
40              }
41              steps {
42                  script {
43                      // Create firmware directory in workspace
44                      sh "mkdir -p ${env.FIRMWARE_PATH}"
45
46                      // Extract filename from URL if not provided
47                      def filename = params.FIRMWARE_URL.tokenize('/'
    )[-1]
48                      echo "Using filename from URL: ${filename}"
49
50                      // Download firmware file from Artifactory
51                      def downloadPath = "${env.FIRMWARE_PATH}/${
    filename}"
52
53                      sh "curl -u jenkins:
    cmVmdGtuOjAxOjE3Nzk2MjA2NTc6bVlZZUk2THdUSnl2amVNUHBPb3pEQjN0VDlv
     -o ${downloadPath} ${params.FIRMWARE_URL}"
54                  }
55              }
56          }
57
58          Test stages...
59
60          stage('Publish Robot Tests Results') {
61              steps {
62                  script {
63                      // Archive the downloaded firmware file
64                      def fileExists = sh(
65                          script: "ls ${env.FIRMWARE_PATH}/*.txt 2>/
    dev/null | wc -l",
66                          returnStdout: true
67                      ).trim() as Integer
68
69                      if (fileExists > 0) {
70                          echo "Found firmware at ${env.FIRMWARE_PATH
    }/*.txt"
71                          archiveArtifacts artifacts: "firmware/*.txt
    ", allowEmptyArchive: true
```

```
72
73                  } else {
74                      echo "No firmware found at ${env.FIRMWARE_PATH
    }/*.txt"
75                  }
76
77                      step(
78                          [
79                          $class              : 'RobotPublisher',
80                          outputPath          : 'robot_results',
81                          outputFileName      : 'output.xml',
82                          reportFileName      : 'report.html',
83                          logFileName         : 'log.html',
84                          disableArchiveOutput: false,
85                          //passThreshold       : "${env.
    ROBOT_PASS_THRESHOLD}" as double,
86                          //unstableThreshold   : "${env.
    ROBOT_UNSTABLE_THRESHOLD}" as double,
87                          otherFiles          : "**/*.png,**/*.jpg",
88                          ]
89                      )
90                  }
91              }
92          }
93      }
94  }
```

Listing B.1: Jenkins Pipeline-konfiguration för automatiserad testning endast nytt steg samt alla parametrar

C

Appendix 3: Robot Framework-testfil för FOTA_TEST

```
*** Keywords ***
| Use Workspace Firmware
| | [Documentation] | Use firmware from Jenkins workspace
| | # Look for firmware in workspace directory
| | ${firmware_dir}=  | Set Variable  | ${workspace_path}/firmware
| | ${files}=  | List Files In Directory  | ${firmware_dir}  | *.txt
| | ${file_count}=  | Get Length  | ${files}
| |
| | Should Be True  | ${file_count} > 0  | No firmware files found in workspace directory: ${firmware_dir}
| |
| | ${latest_file}=  | Set Variable  | ${files}[0]
| | ${firmware_path}=  | Set Variable  | ${firmware_dir}/${latest_file}
| | Log  | Using firmware file from workspace: ${firmware_path}
| | Set Suite Variable  | ${fmw_to_send}  | ${firmware_path}
| |
| | # Verify the firmware file exists
| | File Should Exist  | ${fmw_to_send}
| | Log  | FOTA test will use firmware: ${fmw_to_send}
```

Listing C.1: Nytt keyword in robot Framework-testfil för FOTA-funktionalitet

D

Appendix 4: Konfigurationsfil i YAML

```yaml
# Itirub Configuration
# Single source of truth for Itirub variants and tests

variants:
  rpi-itirub-eth-adc:
    BOARD_DEFINE: BOARD_WITH_ADS1115_ADC
    APP_DEFINE: BUILD_ETHERNET_APP
    tests:
      RUN_ANALOG_SENSOR_TEST: true
      RUN_PARTNUMBER_TEST: true
      RUN_FOTA_TEST: true
      RUN_WIFI_TEST: false
      RUN_MQTT_SUB_TEST: true
      RUN_INCOMING_DATA_TEST: true

  rpi-itirub-eth-digital:
    BOARD_DEFINE: BOARD_WITH_IOEXPANDER
    APP_DEFINE: BUILD_ETHERNET_APP
    tests:
      RUN_DIGITAL_SENSOR_TEST: true
      RUN_PARTNUMBER_TEST: true
      RUN_FOTA_TEST: true
      RUN_WIFI_TEST: false
      RUN_DIGITAL_TEST: true
      RUN_ITIRUB_RELAY_TEST: true

  rpi-itirub-wifi-adc:
    BOARD_DEFINE: BOARD_WITH_ADS1115_ADC
    APP_DEFINE: BUILD_WIFI_APP
    tests:
      RUN_ANALOG_SENSOR_TEST: true
      RUN_PARTNUMBER_TEST: true
      RUN_FOTA_TEST: true
      RUN_WIFI_TEST: true
      RUN_MQTT_SUB_TEST: true

  rpi-itirub-wifi-digital:
    BOARD_DEFINE: BOARD_WITH_IOEXPANDER
    APP_DEFINE: BUILD_WIFI_APP
    tests:
      RUN_DIGITAL_SENSOR_TEST: true
```

```
      RUN_PARTNUMBER_TEST: true
      RUN_FOTA_TEST: true
      RUN_WIFI_TEST: true
      RUN_DIGITAL_TEST: true

  rpi-itirub-lora-adc:
    BOARD_DEFINE: BOARD_WITH_ADS1115_ADC
    APP_DEFINE: BUILD_LORA_APP
    tests:
      RUN_ANALOG_SENSOR_TEST: true
      RUN_PARTNUMBER_TEST: true
      RUN_FOTA_TEST: true
      RUN_WIFI_TEST: false

  rpi-itirub-lora-digital:
    BOARD_DEFINE: BOARD_WITH_IOEXPANDER
    APP_DEFINE: BUILD_LORA_APP
    tests:
      RUN_DIGITAL_SENSOR_TEST: true
      RUN_PARTNUMBER_TEST: true
      RUN_FOTA_TEST: true
      RUN_WIFI_TEST: false

default_tests:
  RUN_FOTA_TEST: true
  RUN_MQTT_SUB_TEST: false
  RUN_SNTP_TEST: false
  RUN_PARTNUMBER_TEST: false
  RUN_INCOMING_DATA_TEST: false
  RUN_UNSENT_DATA_STORAGE_TEST: false
  RUN_WIFI_TEST: false
  RUN_DIGITAL_TEST: false
  RUN_ANALOG_SENSOR_TEST: false
  RUN_DIGITAL_SENSOR_TEST: false
  RUN_ITIRUB_RELAY_TEST: false
```

Listing D.1: Konfigurationsfil i YAML

INSTITUTIONEN FÖR Data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige
www.chalmers.se

CHALMERS