D3.6: Final Report on Integrated Development Environment

Revision: V1

Work package	WP 3
Task	Task 3.3
Due date	2026-03-31
Submission date	2026-03-31
Deliverable lead	Carlos Coutinho (CMS)
Version	1.0
Authors	Carlos Coutinho (CMS), Miguel Goulão (NOVA), Miguel Resende (CMS), Carlos Reis (CMS), Miroslav Popovic (UNS), Pavle Vasiljevic (UNS), Alceste Scalas (DTU), Sotiris Spantideas (NKUA), Luís Pisco (EDP), Gonçalo Lacerda (EDP), Manuel Pio Silva (EDP), João Costa Seco (NOVA), Diogo Jesus (NOVA), Cláudia Soares (NOVA), Dimitra Tsigkari (TID), Milos Simic (UNS), Ivan Prokic (UNS), Lidija Fodor (UNS), Ping Hou (OXF)
Reviewers	Lidija Fodor (UNS), Nuno Preguiça (NOVA), Carla Ferreira (NOVA)
Abstract	This document is the third and final iteration of the report which provides a comprehensive assessment of the TaRDIS IDE platform, highlighting its suitability for developing the TaRDIS toolbox. Through detailed analysis, it evaluates the requirements, customisation and integration activities to build the most suitable IDE for supporting the development of swarms using TaRDIS.
Keywords	Integrated Development Environment

Document Revision History

Version	Date	Description of change	List of contributors
V0.1	2025-03-11	Document first draft	Carlos Coutinho (CMS), Miguel Resende (CMS)
V0.2	2026-03-15	First version for internal review	Carlos Coutinho (CMS), Miguel Goulão (NOVA)

Grant Agreement No.: 101093006
Call: HORIZON-CL4-2022-DATA-01

Topic: HORIZON-CL4-2022-DATA-01-03
Type of action: HORIZON- RIA

V1	2026-03-31	Final report	Carlos Coutinho (CMS)

DISCLAIMER

Funded by the European Union

Funded by the European Union (TARDIS, 101093006). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

COPYRIGHT NOTICE

Project funded by the European Commission in the Horizon Europe Programme		
Nature of the deliverable:	**DEM + R**	
Dissemination Level		
PU	*Public, fully open, e.g. web (Deliverables flagged as public will be automatically published in CORDIS project's page)*	✓
SEN	*Sensitive, limited under the conditions of the Grant Agreement*	
Classified R-UE/ EU-R	*EU RESTRICTED under the Commission Decision No2015/ 444*	
Classified C-UE/ EU-C	*EU CONFIDENTIAL under the Commission Decision No2015/ 444*	
Classified S-UE/ EU-S	*EU SECRET under the Commission Decision No2015/ 444*	

* R: Document, report (excluding the periodic and final reports)

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

DATA: Data sets, microdata, etc.

DMP: Data management plan

ETHICS: Deliverables related to ethics issues.

SECURITY: Deliverables related to security issues

OTHER: Software, technical diagram, algorithms, models, etc.

EXECUTIVE SUMMARY

The TaRDIS project builds a distributed programming toolbox to simplify the development of decentralised, heterogeneous swarm applications deployed in diverse settings.

The main contribution of this deliverable is the third revision of the TaRDIS IDE platform, the core tool of the proposed development environment to foster the creation of applications that conform to the TaRDIS programming model. The objective of this tool is to reduce fragmentation across the project toolchain by providing a coherent user experience inside a familiar IDE, enabling project partners and end users to access, configure, and run the tools through consistent workflows rather than through disparate command-line interfaces, standalone GUIs, or ad hoc scripts. The TaRDIS IDE not only provides developers with a source code editor, build automation tools, and debuggers, but also serves as a seamless environment that integrates the TaRDIS tools. It offers centralised access to the TaRDIS toolbox and multiple supporting features to simplify tool configuration and integration into projects that share a common development environment.

A key outcome is an end-to-end workflow that guides users from project setup to execution and inspection of results with minimal context switching. This report details how the integration abstracts heterogeneity across tools (languages, runtimes, dependencies, and execution models) behind a consistent interface, while preserving advanced capabilities for expert users.

This deliverable builds upon D3.2 [1] and D3.4 [2] (first and second versions of the TaRDIS IDE) and documents the developments and improvements since the submission of D3.4.

This document aims to demonstrate the IDE's suitability for developing the TaRDIS toolbox. It reports on the IDE's development using Visual Studio Code (VS Code) and documents its integration with the tools being developed in the project's WP3, WP4, WP5, and WP6. In some cases, it describes completed integration activities, while in others, it focuses on user interface requirements and integration needs. Overall, the environment provides a practical, scalable foundation for consolidating the project's tooling into a single, user-centred workspace, improving usability, repeatability, and adoption across diverse stakeholders.

The document also develops the concept of "developer stories" initiated in D3.4, describing how a developer intending to create a swarm environment would face the challenge of using the TaRDIS toolbox. It includes stories regarding the usage of isolated tools and libraries, and the "use-case" stories of the TaRDIS pilots, describing how the use-cases are being developed considering the usage of the TaRDIS toolbox. This is an essential contribution towards helping new adopters of these paradigms to embrace development strategies, guidelines and other support that may be provided by the project experts on the field.

This deliverable is constituted by a Demonstrator (source code available openly on CodeLab: https://codelab.fct.unl.pt/di/research/tardis/toolkit/ide/vscode/vscode-tardis-extension and it is also available as a standalone package on the Zenodo repository https://doi.org/10.5281/zenodo.19211021), and by a related report to describe the activities performed for the development and customisation of the IDE and the integration activities with the TaRDIS toolbox.

TABLE OF CONTENTS

Funded by the European Union

Funded by the European Union

List of Figures

List of Tables

Funded by the European Union

ABBREVIATIONS

ACT	Actyx
AGV	Automated Guided Vehicle
AI	Artificial Intelligence
API	Application Programming Interface
BDS-3	BeiDou 3rd Generation navigation satellite system
CDF	Cumulative Distribution Function
CFSM	Communicating Finite State Machines
D-Exit	Decentralized Early-Exit of Inference Framework for swarm systems
DCR	Dynamic Condition Relation
DER	Distributed Energy Resources
DL	Deep Learning
DNN	Deep Neural Network
DP	Differential Privacy
DRFL	Deep Reinforcement Federated Learning
DSO	Distribution System Operator
EE	Early Exit
ERP	Enterprise Resource Planning
FAUNO	Federated AI Network Orchestrator
FL	Federated Learning
FLA	FL Algorithm
FLaaS	FL as a Service
G2G	Galileo 2nd Generation of satellites
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IFC	Information Flow Control
IoT	Internet of Things
IP	Internet Protocol
IPFS	InterPlanetary File System
ISL	Inter-Satellite-Link
JAR	Java Archive
JS	JavaScript
LEO	Low Earth Orbit
LSTM	Long Short-Term Memory
MARL	Multi-Agent Reinforcement Learning
MES	Manufacturing Execution System

Funded by the European Union

ML	Machine Learning
MPST	Multiparty Session Types
MPT-FLA	MicroPython Testbed for Federated Learning Algorithms
MSE	Mean-Squared Error
ODTS	Orbit Determination and Time Synchronization
P2P	Peer-to-Peer
PNT	Position, Navigation and Timing
PTB-FLA	Python TestBed for Federated Learning Algorithms
RL	Reinforcement Learning
SDLC	Software Development Life Cycle
SGAM	Smart-Grid Architectural Model
SL	Split Learning
SDLC	Software Development Life Cycle
TCP	Transmission Control Protocol
TID	Telefonica
UDP	User Datagram Protocol
UI	User Interface

1 IDE Analysis

Integrated Development Environments (IDEs) are essential to modern software development because they bring together core activities such as code editing, debugging, version control, build automation, testing, documentation access, and project management within a single workspace. By reducing fragmentation and context switching, IDEs improve productivity, lower error rates, and support faster onboarding for developers working on complex codebases. Beyond basic editing, IDEs typically provide intelligent code completion, syntax highlighting, refactoring support, dependency management, and integration with version control systems such as Git. Modern IDEs are also extensible through plugins and extensions, allowing them to support a wide range of languages, frameworks, testing tools, and specialised workflows.

The choice of IDE has important consequences for both individual developers and teams. An effective IDE must combine functionality, performance, extensibility, stability, and accessibility, while accommodating the heterogeneous nature of current software projects, which often span multiple languages, frameworks, and deployment targets. For TaRDIS, the definition of an IDE for distributed and decentralised swarm applications must consider the requirements of the project pilots, the expected developer experience, and the vision for the tools produced across the project work packages.

Although the initial TaRDIS IDE work was based on Eclipse, owing to the team's existing experience with that platform, the project gradually shifted towards Visual Studio Code (VS Code). While Eclipse remains highly configurable and technically suitable, VS Code became the preferred choice because of its lightweight design, versatility, strong adoption, and extensive extension ecosystem. The transition had already begun in D3.2 [1] and, as experience with the platform matured, the project moved its integration efforts from Eclipse to VS Code. Following reviewer feedback, the team ultimately focused exclusively on VS Code, with the expectation that this would increase the IDE's adoption by the wider development community.

Microsoft Visual Studio Code

Since its release by Microsoft in 2015, Visual Studio Code has become one of the most widely adopted development environments. According to the Stack Overflow Developer Survey [3] (see Figure 1), it has ranked as the most popular development environment since 2018, with adoption above 75% among professional developers. Its success derives from a combination of responsiveness, cross-platform support, extensibility, and a large and active community. Built on Electron, VS Code runs across Windows, macOS, and Linux, while its open-source core has enabled a large marketplace of extensions covering virtually every programming language, framework, and workflow.

Funded by the European Union

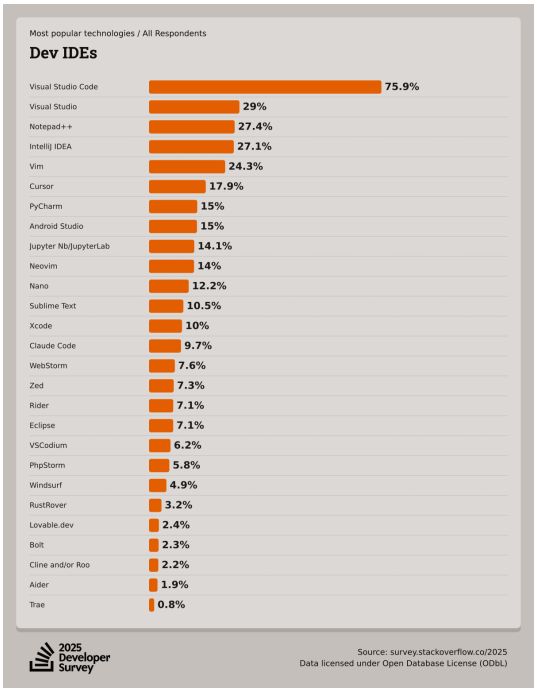

Figure 1: Stack Overflow 2025 Developer Survey on Most Popular Technologies [3].

A major strength of VS Code is that it combines the simplicity of a text editor with IDE-class features delivered through extensions. It supports JavaScript, TypeScript, Python, Java, C++, and many other languages, either natively or through readily available extensions. Features such as IntelliSense, debugging support, integrated terminal access, workspace management, side-by-side editing, and built-in Git integration make it suitable for both small projects and large-scale multi-language development. At the same time, developers can customise the environment through themes, snippets, keymaps, language support, renderers, and other extensions written in JavaScript or TypeScript.

For research and innovation projects such as TaRDIS, VS Code is particularly attractive because its extension API provides a stable and well-documented basis for integrating custom capabilities. Extensions can contribute language support, task runners, debuggers, user interface panels, and other domain-specific interactions while remaining fully embedded in the editor. This makes VS Code especially well suited as a host platform for project-specific tooling. It also supports containerised and remote development patterns, can be aligned with organisational policies through workspace settings and profiles, and provides a practical foundation for consolidating multiple project tools within a single user-friendly environment. These characteristics improve usability, repeatability, uptake of project results, and long-term sustainability by relying on a widely used and actively maintained platform.

VS Code is also free and partially open source, which further strengthens its attractiveness for collaborative and cross-organisational settings. Overall, it provides the right balance of usability, flexibility, and extensibility for integrating the tools developed in TaRDIS and presenting them through a coherent development experience. The present document therefore reports on the work undertaken to adapt VS Code as an integration environment for multiple tools developed within the TaRDIS project.

2 Integration of VS Code with the TaRDIS Toolbox

This chapter describes the work undertaken to integrate VS Code with the TaRDIS toolbox, with the goal of making TaRDIS capabilities accessible through a coherent, IDE-native workflow. While TaRDIS provides the core functionality through its own set of tools and interfaces, users typically experience the toolbox as a sequence of separate steps: installing dependencies, configuring inputs, invoking commands or services, and collecting outputs across different locations.

Positioning VS Code as the primary interaction layer addresses these issues by embedding TaRDIS operations where users already spend most of their time: inside the development environment where code, configurations, and artefacts are edited, executed, and reviewed.

The integration therefore focuses on:

- streamlining onboarding and setup
- providing consistent configuration and execution controls
- exposing TaRDIS outputs: analysis reports, logs, and generated artefacts

The intent is not to replace the TaRDIS components, but to orchestrate them and present them through a user experience that is predictable, discoverable, and scalable as the toolbox evolves.

The remainder of the chapter introduces the integration requirements and assumptions, the chosen architecture and extension mechanisms, the implemented user-facing features, and the validation performed using representative TaRDIS scenarios. It also highlights limitations and integration risks (such as dependency handling and interface evolution) and records recommendations for maintaining the VS Code integration.

Languages support

It can support all languages, either by using an existing extension or by creating new ones.

Customization

Developers can customize the editor with extensions, snippets, themes, language support, keymaps, and notebook renderers, all of which are written in TypeScript or JavaScript.

Prerequisites

- Install Visual Studio Code.
- Install Node.
- Install globally the '**yo**' [8] and '**generator-code**' [9] packages via Node Package Management.
- Run '**yo code**' and choose how to extend the editor.
- Open the created project with the editor and customize it.

Funded by the European Union

2.1 How the Extension Was Built

The **TaRDIS Extension for Visual Studio Code** was developed to streamline the workflow for creating and managing swarms, whether the purpose is to define a full **TaRDIS swarm application**, as described in the TaRDIS deliverable D3.5 [10], or to define the behaviour of individual **managed** or **free-form swarm elements**. This action is performed by the development of one or multiple projects which span the technologies used in the TaRDIS project. The design process focused on integrating seamlessly with the Visual Studio Code ecosystem, automating complex configurations and offering an intuitive user experience. This section outlines the tools, processes and methodologies employed in this development.

2.1.1 Development Objectives

The primary objective behind the development of the TaRDIS extension was to simplify how the developers interact with distributed systems by automating intricate configurations and providing user-friendly workflows. The extension was designed to be versatile and adapt its features to suit their specific project requirements.

To achieve this, the extension supports multiple project types, besides the traditional language development-based (e.g., Java, C++, Python or TypeScript), extended to specific project types related to the TaRDIS project, e.g., Babel-based Java projects, FAUNO and PTB-FLA projects for federated learning, or DCR choreographies for distributed execution. It ensures seamless integration with essential tools such as Maven for Java projects, Python virtual environments for PTB-FLA, and Docker for running DCR-based prosumers. This focus on usability, flexibility and cross-platform compatibility was key to creating an extension that enhances productivity while maintaining robust functionality.

One important functionality that was added for this revision is the integration of an AI/ML generative engine called the TaRDIS Assistant Guide, that allows the user to interact with the TaRDIS chatbot to understand how to perform the development activities, or how to interact with a certain TaRDIS tool, or produce a recipe for the development of a specific challenge (see section 2.5.6).

2.1.2 Tools and Frameworks

Node.js

Node.js was used as the runtime environment to build the extension and execute its operations. It facilitated seamless integration with the Visual Studio Code API while providing an extensive ecosystem for dependency management through `npm` and helped build the structure of the extension with existing `npm` packages like ‘yo’ and ‘`generator code`’.

TypeScript

TypeScript was chosen as the primary development language for its static typing, modern JavaScript features, and strong compatibility with the Visual Studio Code API. This choice ensured the extension’s reliability, maintainability and ease of debugging as the tools provided by the **yo** Node Package Manager facilitated the integration using this language.

Visual Studio Code API

The VS Code API enabled the extension to interact directly with the IDE, allowing for the registration of commands, dynamic file manipulation, and the rendering of interactive WebView's. This API served as the backbone for the extension's core functionality. The API documentation is found in [6].

Webview API

The WebView API played a crucial role in creating visually rich and interactive panels that allowed users to make choices. These panels were used for user inputs, project configuration, protocol documentation and DCR choreography management. This ensured the extension maintained a consistent design while delivering clarity to users.

Maven

For Java-based projects, Maven was employed to manage project dependencies and build configurations. The extension has the ability to automatically update the pom.xml file to include the required Babel libraries and repositories at the press of a button, eliminating any manual configuration steps.

Python Virtual Environment

For PTB-FLA projects, the extension created a dedicated virtual environment (**venv_ptbfla**) with pre-installed dependencies. This environment ensured compatibility with machine learning libraries, providing a ready-to-use workspace for federated learning development.

Docker

Docker was integrated into the extension for running DCR choreographies. The extension automated the creation of Docker containers, networks, and images, ensuring each prosumer instance operated in an isolated yet interconnected environment.

Markdown Renderer

A custom markdown renderer was implemented to display documentation within WebView panels. This allowed users to view formatted protocol specifications, configuration parameters, and API documentation directly within VS Code.

Yeoman

The initial project structure was scaffolded using Yeoman's generator for VS Code extensions, providing a modular foundation for the codebase and ensuring a consistent development workflow.

2.1.3 Implementation Steps

The development of the TaRDIS extension followed a structured approach:

Funded by the European Union

1. Initializing the Extension

The project began with the initialization of the extension using Yeoman's yo code generator. This tool provided a TypeScript-based scaffold, complete with the necessary configuration files and boilerplate code. This foundation enabled the rapid development of core functionalities while maintaining clean code organization.

2. Registering Commands

User commands were defined in the **package.json** file, corresponding to each key feature, such as creating a project, importing protocols, compiling DCR choreographies, and visualizing swarm protocols, as depicted on Figure 2. These commands acted as entry points, allowing users to trigger specific actions through the command palette or the TaRDIS sidebar.

```
"contributes": {
  "commands": [
    {
      "command": "tardis.commands.create-project",
      "title": "TaRDIS: Create Project"
    },
    {
      "command": "tardis.commands.create-generic-class",
      "title": "TaRDIS: Create Generic Class"
    },
    {
      "command": "tardis.commands.import",
      "title": "TaRDIS: Import existing project"
    },
    {
      "command": "tardis.commands.visualize-swarm",
      "title": "TaRDIS: Visualize Swarm Protocol"
    },
    {
      "command": "tardis.commands.compileDcr",
      "title": "TaRDIS: Compile tardis DCR"
    },
```

Figure 2: File package.json defining commands.

3. Creating WebView Panels

The WebView API was used to build interactive panels for project creation, protocol management, and DCR choreography execution. These panels displayed user inputs, protocol documentation, and real-time status updates, providing an intuitive interface for complex workflows, as seen on Figure 3.

```
const panel = vscode.window.createWebviewPanel(
    'protocolDetails',
    `Details for ${importSelected}`,
    vscode.ViewColumn.Beside,
    { enableScripts: true }
);
```

Figure 3: Create WebView panel.

4. Automating Project Setup

To simplify project initialization, the extension automated the creation of folder structures, configuration files, and dependency management. For Babel projects, the pom.xml file was dynamically updated. PTB-FLA projects were equipped with a virtual environment, while DCR projects included Docker setup scripts.

5. Dynamic Protocol Handling

The extension implemented dynamic handling for protocol-specific operations, such as fetching dependencies, displaying configuration parameters, and managing API events. This ensured compatibility with a wide range of Babel-based protocols and DCR choreographies.

6. Integrating Docker for DCR Choreographies

For DCR projects, the extension automated Docker-based execution. It created isolated prosumer instances, each running within its own container, while connecting them via a dedicated Docker network. This approach ensured realistic simulation environments for distributed workflows.

7. Styling and Theming

Custom HTML and CSS were used to style WebView panels, ensuring alignment with Visual Studio Code's dark and light themes. Syntax highlighting for code blocks and a clean layout further contributed to a user-friendly and professional interface.

2.2 Key Features of the Extension

The TaRDIS extension offers a comprehensive set of features designed to enhance the development experience for distributed systems projects.

1. Multi-Project Creation

With the current integration of tools in the IDE, developers are able to create (for the moment) three types of projects, as depicted in Figure 4:

- **Java Babel Projects:** Fully configured Maven-based projects with Babel dependencies, ideal for developing swarm applications.
- **PTB-FLA Projects:** Python-based projects for federated learning algorithms, complete with a virtual environment and predefined templates.
- **DCR Choreography Projects:** Projects designed for distributed execution, allowing users to compile and run dynamic condition-response choreographies.

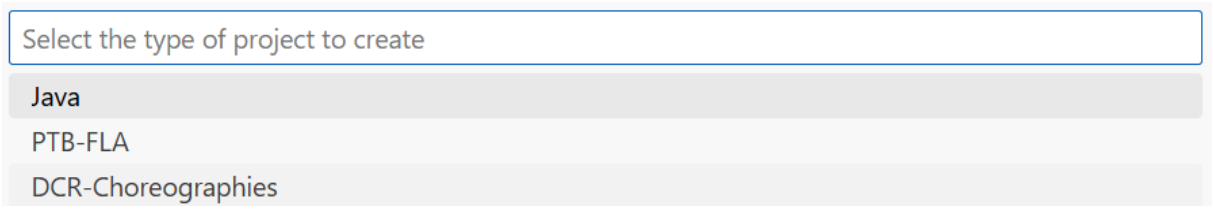

Figure 4: TaRDIS project creation.

2. Documentation Viewer

The extension includes a markdown-based documentation viewer (Figure 5), displaying detailed information for each protocol, including usage instructions, configuration parameters, and API events. This ensures users have all necessary resources to import different protocols, as documentation is always present to inform the programmer of the protocol's functionalities.

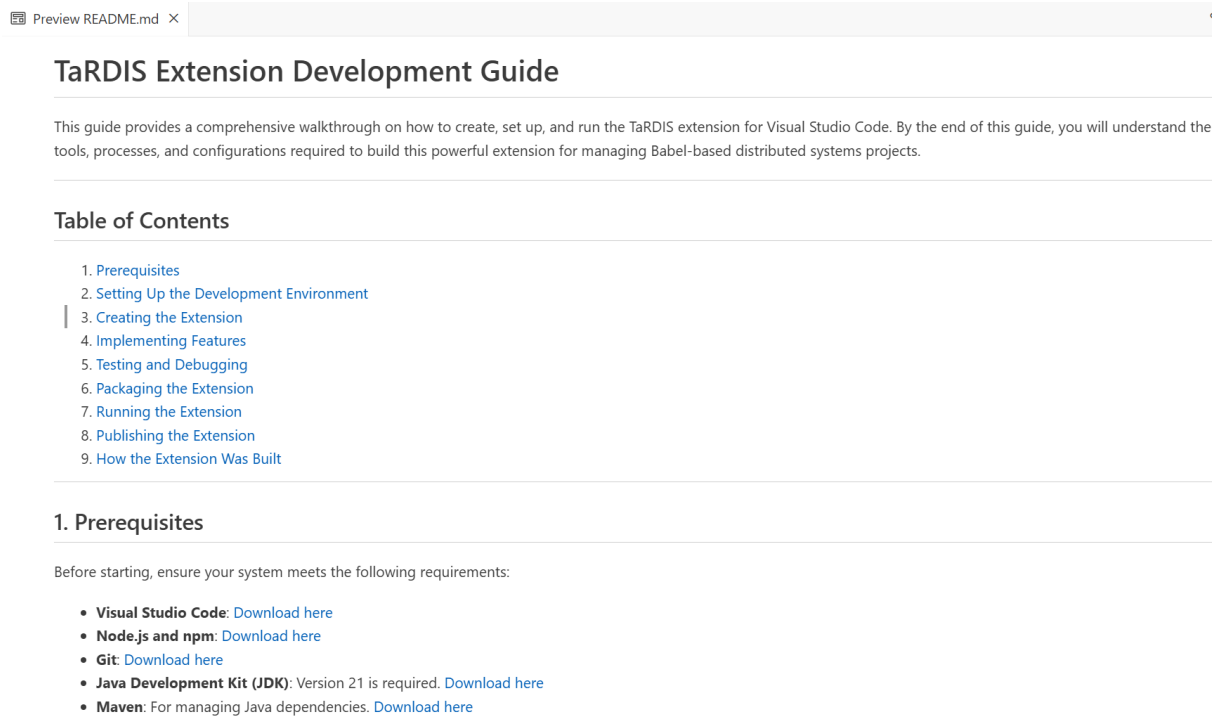

Figure 5: TaRDIS IDE documentation viewer.

3. Dependency Management

Managing dependencies is streamlined through automated functionality. For Java projects, the **pom.xml** file is updated to include both Maven dependencies and those associated with imported protocols. PTB-FLA projects are initialized with all necessary Python packages when a new project is created, while DCR projects ensure Docker and OCaml are properly configured by checking if they are installed before running any commands required for initializing the prosumers.

4. Dynamic Project Setup

The extension offers guided workflows for project setup, providing pre-configured environments for Java, Python, and DCR projects. Users can easily switch between project

types without manual setup, by selecting the type of project to be added to the workspace. All projects created are added to one common workspace in order to be managed and viewed all in the same place for better development interaction.

5. Extensibility

Built with a modular design, the extension allows for easy expansion. Developers can add new protocols, classes, and features as project requirements evolve. This includes the possibility to add buttons to create classes (to add new predefined classes and protocols to facilitate the process for the programmer), or to import and visualize swarms in a workflow editor.

6. Integrated Docker Execution

For the DCR projects, the extension automates Docker-based execution. It creates isolated containers for each prosumer, connects them to a shared network managed by docker, and executes the choreography, providing real-time logs in the terminal.

2.3 Tools Required for the Operation

To ensure seamless operation, the following tools are required:

- **Java Development Kit (JDK):** Java 21 is essential for Babel-based projects, providing the runtime environment for Java applications [11].
- **Node.js:** Serves as the runtime environment for the extension itself, managing dependencies and executing commands [12].
- **Maven:** Used for dependency management and build automation in Java projects [13].
- **Python:** Required for PTB-FLA projects, with virtual environments ensuring package isolation [14].
- **Docker:** Facilitates containerized execution of DCR choreographies [15].
- **Visual Studio Code:** Serves as the primary development environment [5].
- **Git:** Enables version control, code sharing, and collaborative development [7].

Optional tools, such as ESLint [16] for code quality and the VS Code Debugger for real-time testing, further enhance the development experience. Together, these tools create a robust environment for building and managing distributed systems using the TaRDIS extension.

By combining project management, protocol integration, federated learning support, and distributed execution capabilities, the TaRDIS extension provides a comprehensive solution for developing modern distributed systems within Visual Studio Code. It not only simplifies complex workflows but also ensures users can focus on development without being burdened by intricate configurations.

Funded by the European Union

2.4 Instruction Manual for the TaRDIS Extension

2.4.1 Overview

The TaRDIS Visual Studio Code extension is a comprehensive toolbox designed to streamline the development and management of distributed systems and projects. The extension provides a range of functionalities, including project creation, protocol importing, and class generation, to help developers quickly and efficiently set up and manage their projects. This manual outlines how to use the TaRDIS extension and its current features.

2.4.2 Installation

Currently the TaRDIS IDE extension is not yet published into the Microsoft marketplace, but it will be made available soon. Nevertheless, it is possible to download it through the TaRDIS public Git repository – named CodeLab[1] – by following step 3 of the installation process described below.

1. **From the Visual Studio Code Marketplace:**
   - Open the Extensions view in VS Code (Ctrl+Shift+X)
   - Search for “TaRDIS”
   - Click “Install”
2. **Manually via VSIX:**
   - Obtain the VSIX file for the TaRDIS extension
   - Open the Extensions view in VS Code (Ctrl+Shift+X)
   - Click on the “...” in the top right corner of the Extensions view
   - Select “Install from VSIX...” and locate the VSIX file
3. **From debugging mode in VS Code:**
   - Obtain the code present in CodeLab in the link:

     https://codelab.fct.unl.pt/di/research/tardis/toolkit/ide/vscode/vscode-tardis-extension
   - Open the code on VS Code
   - Run the extension by choosing Run->Start Debugging or simply press the F5 button to load the extension in debug mode
   - A new instance of VS Code will pop up in a new window, with the extension already installed.
   - Keep in mind that in order to run the extension, there are prerequisites and technologies mentioned above in order to create and run Java and PTB-FLA projects or open the Workflow editor (install the extension available on the VS Code marketplace, on link https://marketplace.visualstudio.com/items?itemName=CarolineFallesen.visual-swarm-protocol-editing), as well as OCaml (https://ocaml.org) and Dune (https://github.com/tarides/dune-release) for DCR Choreographies compilation to work.

[1] https://codelab.fct.unl.pt/di/research/tardis

2.4.3 Getting Started

Once installed, the TaRDIS VS Code extension can be accessed directly from the Activity Bar under the “TaRDIS” view, serving as the central hub for all its features. This interface provides an intuitive way to create new projects, import protocols, generate classes, and manage various development tasks. Users are presented with a set of clearly labelled buttons, each corresponding to a specific functionality, making it easy to navigate and utilize the extension’s capabilities, as depicted in Figure 6.

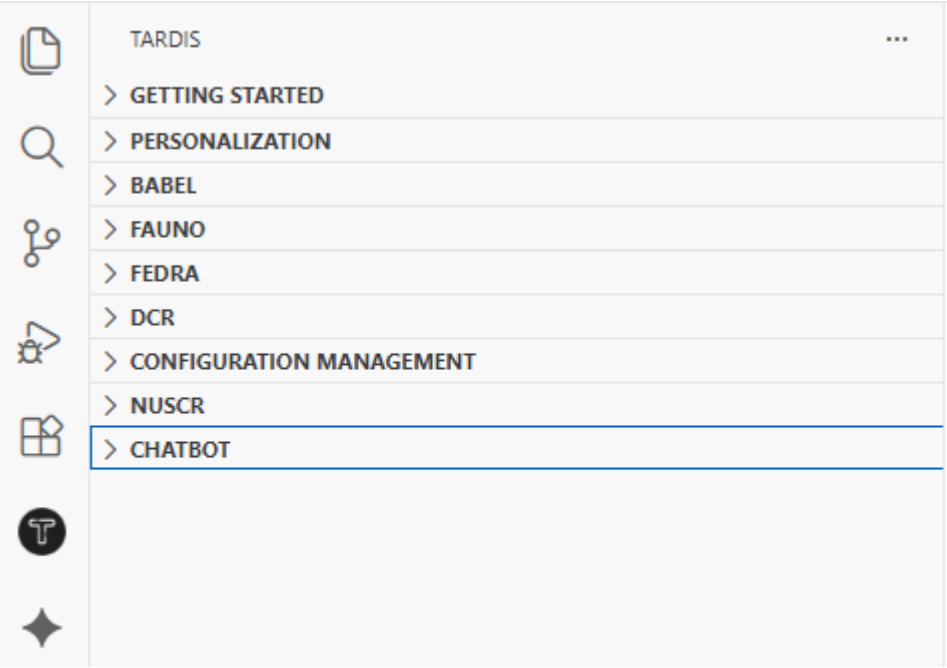

Figure 6: The TaRDIS VS Code extension

The TaRDIS extension allows users to create structured projects with predefined setups and configurations. By selecting the **Create Project** option, developers can initiate a new project tailored to their needs. During the setup process, they can choose between different project types, including Java-based projects using Babel libraries, PTB-FLA projects designed for federated learning applications in Python, or DCR Choreographies for orchestrating distributed prosumers. After specifying a project name and selecting a destination folder, TaRDIS automatically configures the project with the necessary dependencies and structure. For Java projects, this means integrating Babel and its required configurations. PTB-FLA projects, on the other hand, include the creation of a virtual environment (**venv_ptbfla**), automatic installation of the PTB-FLA package, and generation of a dedicated source folder for implementations. DCR Choreography projects establish an execution-ready structure, enabling users to compile and run choreographies with customizable prosumer counts. Once created, the project workspace seamlessly opens in VS Code, allowing users to start coding immediately.

For developers working with external codebases or collaborating on shared repositories, TaRDIS provides a Clone Repository feature. By selecting this option, users can enter a repository URL and specify a destination folder. The extension then clones the repository, making the project available in the local workspace. This feature simplifies the integration of

external projects into the development environment, ensuring that users can quickly access and modify the cloned codebase without additional setup.

TaRDIS also includes a functionality for opening existing projects stored locally. Using the “Open Project” option, users can browse their file system, select a project folder, and have it loaded into the workspace. This allows developers to seamlessly switch between projects without needing to manually configure the workspace each time. By automating these processes, TaRDIS enhances productivity and provides a streamlined experience for managing multiple development workflows within VS Code.

In addition to project creation and management, TaRDIS simplifies the development of Babel-based applications by providing built-in support for generating essential components. With the Create Class feature, developers can quickly define fundamental building blocks such as messages, protocols, timers, notifications, replies, and requests, as shown in Figure 7. After selecting a class type, the user is prompted to enter a name, and the extension generates a corresponding template file within the appropriate package. This eliminates repetitive boilerplate coding and ensures consistency in the project's structure.

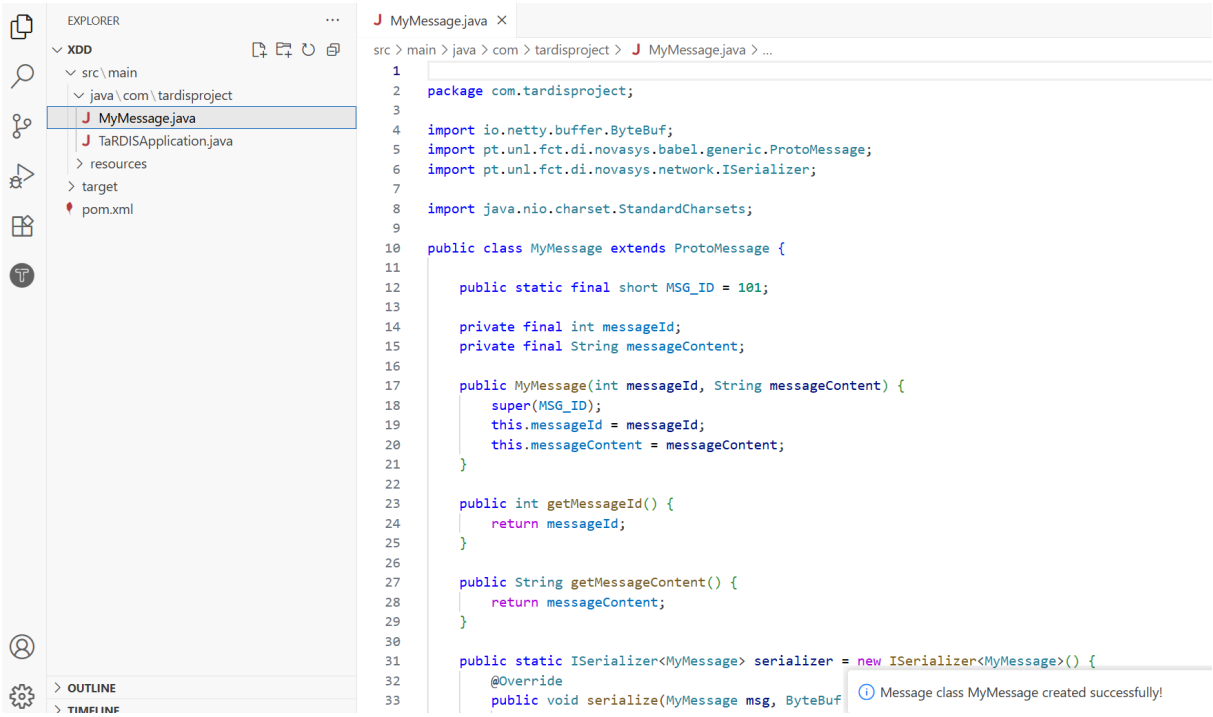

Figure 7: Babel feature for Create Class.

For users who need to extend their projects with predefined communication patterns, the Import Protocol functionality allows for seamless integration of Babel protocols. By selecting this option, developers can browse and import communication protocols categorized by type, such as Eager Gossip Broadcast.

Once a protocol is chosen, TaRDIS automatically handles dependency management, adding the required entries to the project's configuration files and injecting initialization code into the main application. This streamlined workflow accelerates development while ensuring that imported protocols are correctly integrated and fully functional.

This process can be seen below: Figure 8 shows the `init` function before the import action of a chosen protocol. Figure 9 shows the after-import state, with Eager Gossip protocol imported and properly initiated in the function, ready to be used.

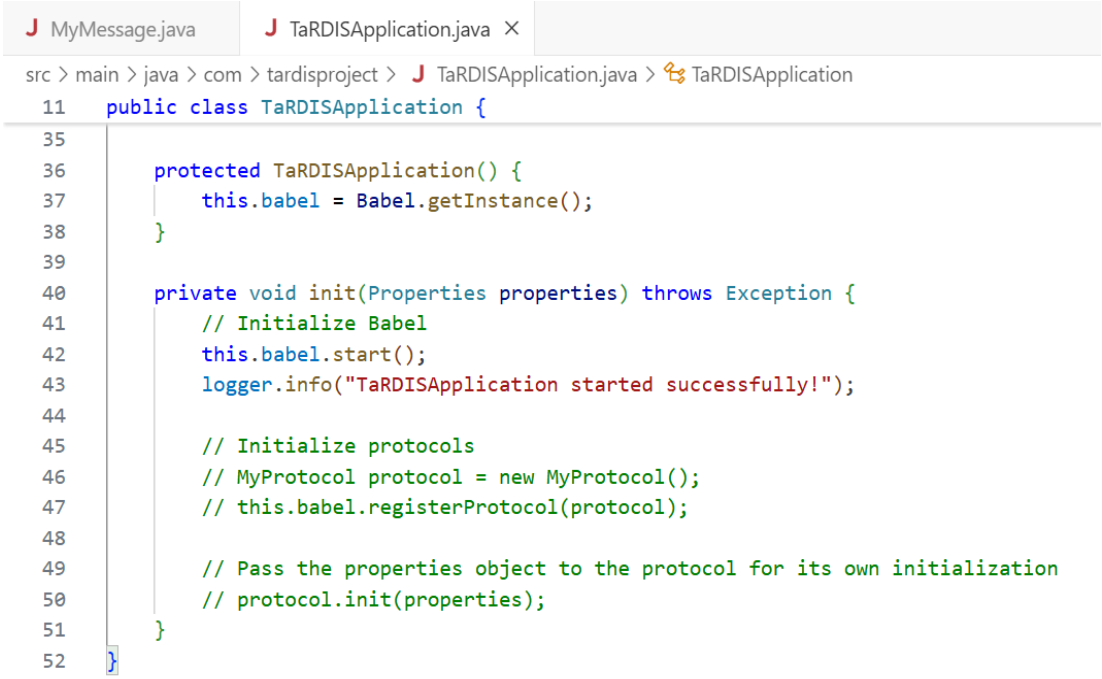

Figure 8: Babel importing of a protocol – before the importing.

Figure 9: Babel importing of a protocol – after the importing.

Another powerful feature within TaRDIS is the ability to Visualize Swarm Protocols, which integrates with the “Visual Swarm Protocol Editing” extension. If a valid swarm protocol file is detected in the workspace, users can launch a graphical representation of the protocol directly

within VS Code. This visualization provides insights into the communication flow and interactions between entities, aiding in debugging and validation of distributed applications.

For projects based on DCR Choreographies, TaRDIS includes a dedicated Compile & Run DCR feature, allowing users to build and execute distributed systems based on dynamic condition response logic. By selecting a `.tardisdcr` file, users can configure the execution environment, specify the number of prosumers, and initiate the build process. The extension ensures that Docker is running, compiles and packages the project, builds a Docker image, and sets up a dedicated network for communication. Once the setup is complete, multiple prosumer instances are launched in separate Docker containers, each executing a part of the choreography. The integrated terminal in VS Code provides real-time logs, allowing users to monitor execution and analyse interactions between components. This feature facilitates the development and testing of distributed applications, making it easier to validate choreography behaviour in a controlled environment.

By combining project management, class generation, protocol importation, visualization tools, and execution capabilities, TaRDIS provides a comprehensive development environment tailored for decentralized and distributed systems. Whether users are working on Java-based Babel projects, federated learning algorithms, or DCR choreographies, the extension ensures a smooth and efficient workflow, integrating seamlessly with VS Code to enhance productivity and streamline development.

2.5 Specific TaRDIS Tools Installation Guidelines

2.5.1 FAuNO Guide

Create FAuNO Configuration

To create a FAuNO configuration file:

- Open the FAuNO extension and click “Create FAuNO Configuration”
- A webview will open where you can define your configuration
- After filling in the fields, click “Create Configuration”
- You will be prompted to choose where to save the file on your system
- Once saved, the configuration will automatically open in your editor.

Edit FAuNO Configuration

To edit an existing FAuNO configuration file:

- Open the FAuNO extension and click “Edit FAuNO Configuration”
- Select a configuration file (YAML, YML, or JSON) from your workspace
- The configuration will load into a webview for editing
- After making changes, click “Save Configuration”
- Changes will be written back to the original file.

Run FAuNO Tests

To run tests with FAuNO:

- Open the FAuNO extension and click “Run FAuNO Test”
- Select a dataset file (CSV or JSON)
- The path to the selected file will be updated in file “local_config.json”
- Then select the FAuNO configuration file you want to use
- A terminal named 'Run FAuNO' will start the test using the selected configuration.

2.5.2 FEDRA Guide

Setup D-Exit Project

- Press the “Setup D-Exit Project” command in the command palette
- Select a folder where you want to clone the D-Exit repository
- If the folder “dexit/” already exists in the selected directory, it will skip cloning
- If not, the extension will:
  - Clone the D-Exit repo from GitHub
  - Add it to your current workspace
  - Open a terminal and run:

```
pip install -r requirements.txt
```

You will find the cloned repo under the “dexit/” folder.

Run D-Exit

- Press the “Run Dexit” command from the command palette
- This command looks for the file “dexit/dexit.py” in the current workspace
- If found, it will:
  - Open a terminal named Dexit Project
  - Navigate to the `dexit/` folder
  - Execute:

```
python dexit.py
```

You should see D-Exit running in the terminal.

Setup EarlyExit Project

- Press the “Setup EarlyExit Project” command in the command palette
- Choose a folder where you want to clone the EarlyExit repo.
- If the directory “earlyExit/” already exists in the folder, it will skip cloning.

- If not, it will:
  - Clone the EarlyExit repo
  - Add it to your workspace
  - Open a terminal and run:

```
pip install -r requirements.txt
```

The project will appear under “earlyExit/”

Setup FEDRA Project

- Press the “Setup Fedra” command from the command palette.
- This command will let you choose a destination folder for Fedra git repository:
  - If this folder exists, it will pull the latest change
  - If not, it will Clone the repository
- Install the dependencies
- Open a WebView prompting the user to either view the documentation and learn more about Fedra, or open the project folder:
  - If the user clicks “Yes, show me more”, it will install the dependencies for an interactive documentation, followed by opening it inside a vscode webview

Setup FLaaS Project

- Under Fedra tab, on the extension menu, click the “Set up FlaaS” button. This will install the project locally
- After the project installation, the user will be shown a webview inside VS Code prompting to choose the setup options:
  - “Basic Setup” - Orchestrates the installation of core project dependencies and generates a lightweight directory structure and essential configuration files
  - “Full Django Setup” - Does everything the “Basic Setup” does, with an additional installation of Django-specific libraries and database drivers, and an automated setup of the local database instance and management tools.
- The extension will programmatically spawn a VS Code Terminal to guide the user through the `createsuperuser` process (Prompting for Username, Email, and Password). Upon successful terminal exit, a dedicated Django Dashboard Webview will launch to confirm the local server status.

2.5.3 DCR Project Guide (DCR Tools)

DCR Editor

- Under the DCR tab, on the TaRDIS extension menu, click the button “Open DCR Editor”

- After clicking the button, the repository will be cloned, or if the repository already exists, it will pull the latest version of DCR Editor and install all the project dependencies if not installed already. Subsequently, it will open a webview inside the VS Code window with a live DCR Editor

DCR Dashboard

- Under the DCR tab, on the TaRDIS extension menu, click the button “Open Dashboard”
- After clicking the button, the repository will be cloned, or if the repository already exists, it will pull the latest version of DCR Dashboard and install all the project dependencies if not installed already. Subsequently, it will open a webview inside the VS Code window with a live DCR Dashboard

Note: This extension supports multiple webviews inside VS Code. Therefore, it is possible to open both DCR Editor and DCR Dashboard simultaneously, allowing the user to work with both tools side by side.

Check DCR Project Presence

- The “Run DCR Choreography” button will only appear if you have opened a folder that contains a valid DCR project. Open the DCR folder (in Documents\tardis-vscode\assets) on the workspace to show the button
- A valid DCR project must contain a “pom.xml” file inside “tardis/babel-backend/”
- If this file is not found, the command will be hidden from the interface

Open the DCR Choreography Panel

- Click the “Run DCR Choreography” button in the side menu. This opens a webview where you can configure and launch DCR setups

Select a Choreography File

- Click “Choose File”
- Select a “`.tardisdcr`” file from your workspace
- The selected file name will appear below the button

Configure the Number of Prosumers

- Enter how many prosumers you want to run in the input box
- Default value is 3

Compile & Setup

- Click “Compile & Setup”
- The terminal will execute the following:
  - Navigate to the scripts directory

- Run `./build.sh`
- Navigate to “tardis/babel-backend”
- Run `mvn clean package`
- Build Docker image
- Create Docker network

This prepares the environment to launch the prosumers.

2.5.4 Configuration Management Guide

- Click on the “Show Configuration Menu” button in the TaRDIS view to start the configuration interface
- Select the folder where the configuration-management project is located when prompted. This should contain a “cockpit” folder inside
- After selecting the folder, a terminal will open, and the configuration UI will appear in a new tab
- Use the “Register” button to create a new user if needed. Fill in the required details.
- Click on the “Login” button to authenticate. Enter your username, then type your password in the terminal
- Once login is successful, additional configuration command categories will become visible.
- Expand any section to see available commands. Click on a button to execute the respective operation using the Cockpit CLI

2.5.5 Scribble Editor (nuScr) Guide

- Click on the “Clone nuScr Editor” button to clone and set up the nuScr project
- After successful cloning, a button to reload the VS Code window will be displayed. After reloading VS Code, the user will see all the buttons featuring the Scribble Editor extension
- In order to use the nuScr features, a **`README.md`** file is available with all the prerequisites
- The nuScr project also includes some demonstration files

2.5.6 TaRDIS Assistant Guide

- Click on the “Open Chatbot” button to launch the TaRDIS Assistant
- The assistant can list all available commands, explain what each command does, or even execute commands on your behalf
- Type your message in the input box, click **Send**, and wait for the chatbot’s response

The TaRDIS Assistant is designed to make interaction with the system more intuitive; it helps you discover features, understand commands, and perform tasks directly through natural language conversation.

3 TaRDIS Toolbox Integration

The development of the TaRDIS Toolbox is being progressively accompanied by the integration of its tools into the TaRDIS IDE. This section outlines the ongoing evolution of these integration activities.

3.1 T-WP3-01 WorkflowEditor

Describe Tool

The WorkflowEditor is a graphical editor which allows its users to edit either the textual specification or the graphical representation of a swarm protocol and update the other accordingly. It is used to design, analyse, and implement workflows between actors based on the Actyx tooling.

Details

See Caroline Bjørch Fallesen, “A Visual Studio Code Extension for Editing Swarm Protocols.” MSc thesis, Technical University of Denmark, 2025.

Available at: https://findit.dtu.dk/en/catalog/679d746fabd7f915d2a6809d

Integration needs

Visual Studio Code extension, with preliminary release and source code available at: https://marketplace.visualstudio.com/items?itemName=CarolineFallesen.visual-swarm-protocol-editing

The WorkflowEditor is a graphical editor for swarm workflows, offering a bidirectional link where the final user can edit either the text or the graphical representation of a swarm protocol, and update the other accordingly. It is used to design, analyse, and implement workflows between actors hosted on swarm devices.

The description of the tool was already stated in the TaRDIS deliverable D3.4 [2], showing that the tool has a self-contained functionality, and listing the foreseen interactions with the IDE.

This tool was already integrated in the IDE, it is offered as an VS Code extension that is invoked by the IDE in case of the creation of an Actyx project, as shown in Figure 10. The IDE instantiates an empty workflow and invokes the editor. All edits of the workflow definition then occur via the graphical editor. The resulting workflow and corresponding declarative persistency statements are then stored in the environment workspace, which can then be used to define the swarm activities, using e.g., the Actyx machine-runner library (T-WP4-01) and the Actyx machine-check library (T-WP4-02).

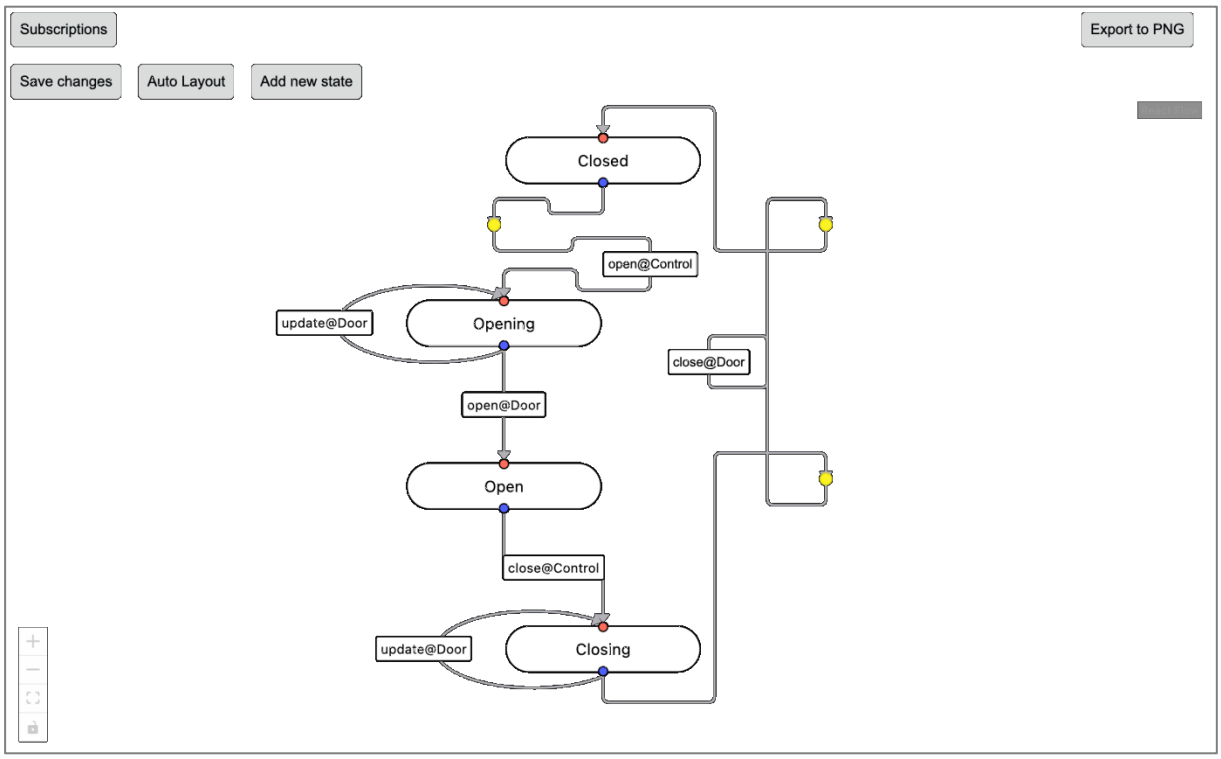

Figure 10: Screenshot by Caroline Bjørch Fallesen - from the MSc thesis "A Visual Studio Code Extension for Editing Swarm Protocols" (DTU, 2025). Available at: https://findit.dtu.dk/en/catalog/679d746fabd7f915d2a6809d

3.2 T-WP3-02 Scribble Editor

The Scribble editor (nuScr) serves as an extensible toolchain for MPST-based multiparty protocols. This toolchain converts multiparty protocols into global types within the MPST theory. These global types are then projected into local types and further transformed into corresponding communicating finite state machines (CFSMs). Additionally, nuScr generates APIs from these CFSMs to implement endpoints in the protocol. The design of nuScr supports language-independent code generation, enabling APIs to be generated in various programming languages. This tool is packed closely with the Scribble extensible toolchain for MPST (T-WP4-05).

This tool was integrated into TaRDIS IDE as a sub-extension, providing a UI for all the commands of the original Scribble Editor. Initially, a button to clone the repository of nuScr Editor is displayed in the IDE button menu, as shown in Figure 11. If the repository URL changes, we allow the user to change the URL settings via our extensions settings as shown in Figure 12.

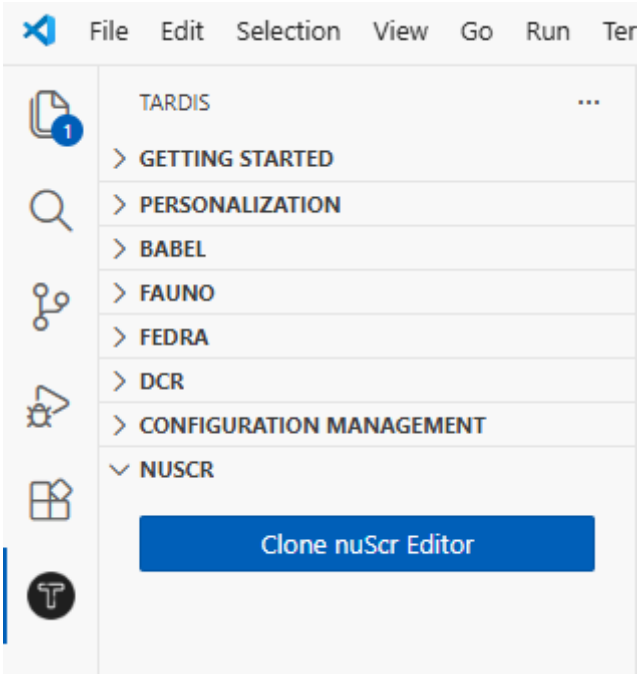

Figure 11: nuScr initial menu view in the TaRDIS IDE.

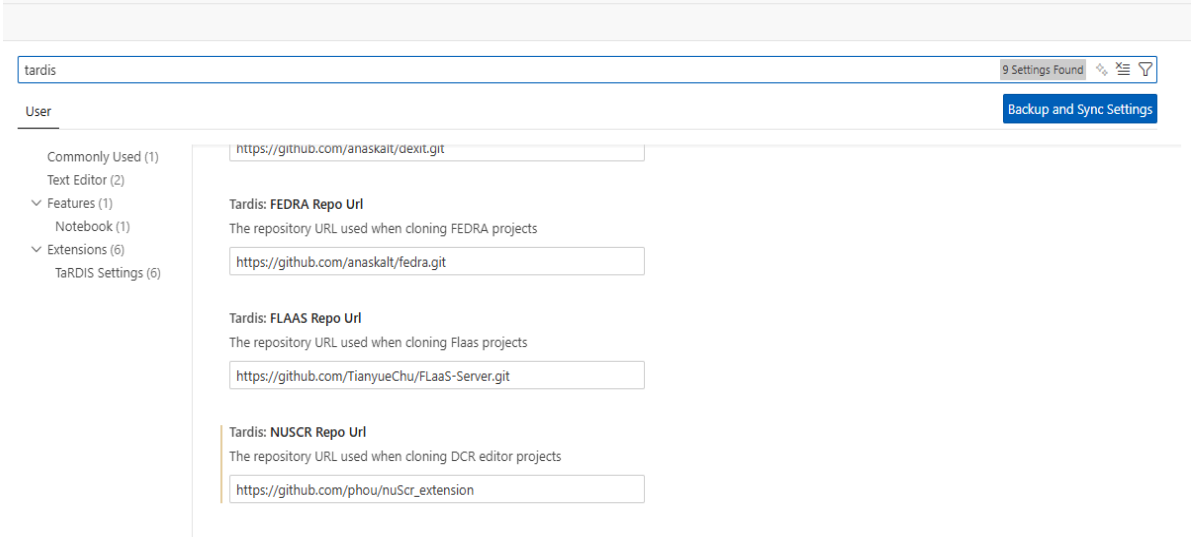

Figure 12: Scribble editor repository URL settings in the TaRDIS IDE.

After completing the clone of the nuScr Editor, behind the scenes all the commands from nuScr will be merged into our main extension commands and added to the NUSCR menu. For the changes to take effect, a prompt with a confirmation button will show on the bottom right of the editor asking to reload the VS Code window as shown in Figure 13.

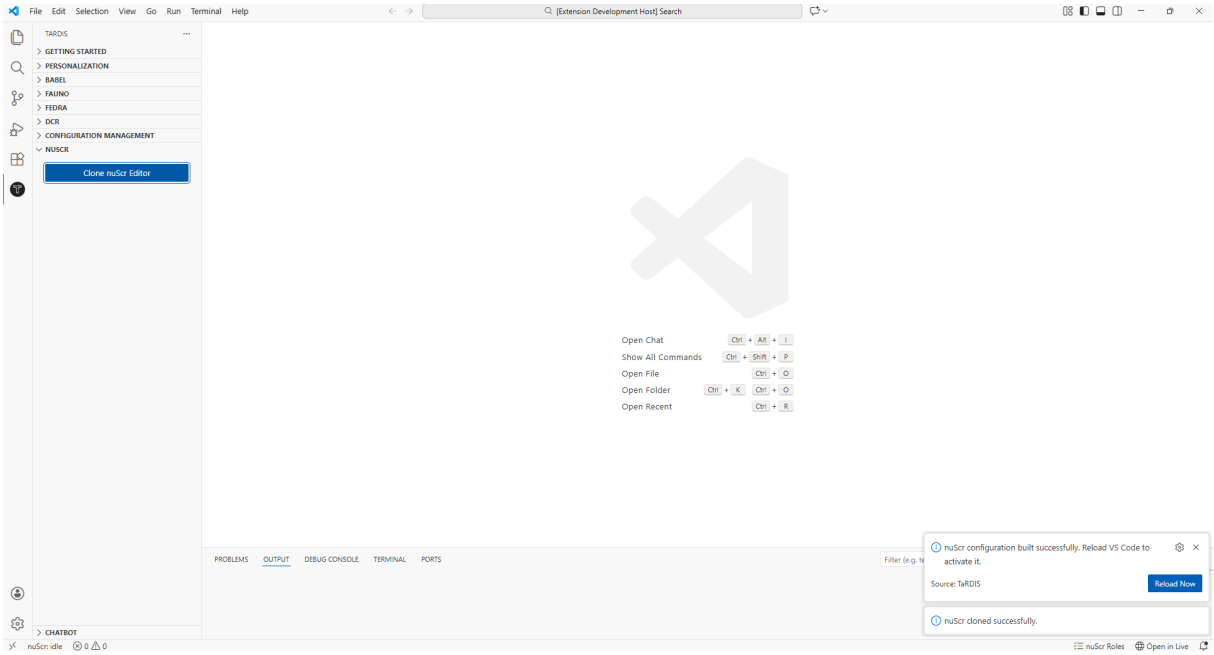

Figure 13: Reload VS Code after successfully merging scribble editor into TaRDIS IDE.

After reloading the VS Code window all the commands (note: this command list may change over time since it depends on the development of the original repository) from the scribble editor will be displayed under the menu NUSCR as shown in Figure 14.

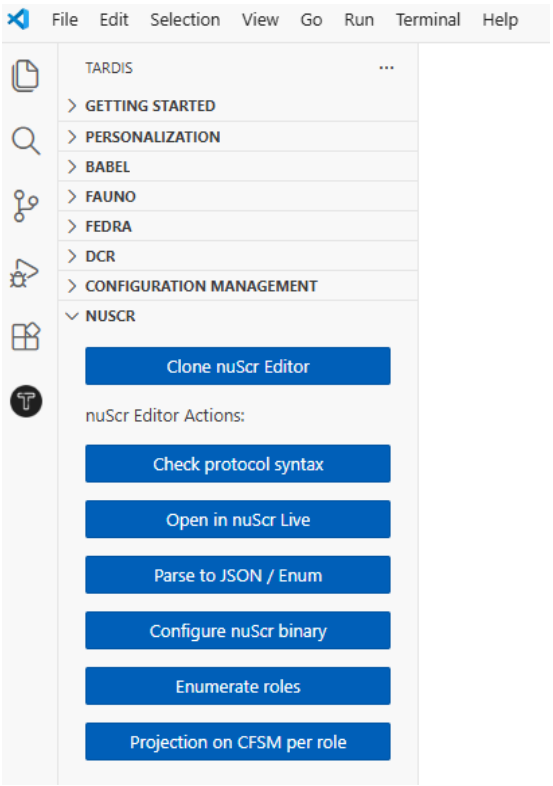

Figure 14: nuScr action list inside NUSCR menu in the TaRDIS IDE.

In order to use nuScr a **`README.md`** file with all the requirements is located inside the extension's main directory: `src\features\nuScr\extension\README.md`

3.3 T-WP3-03 DCR Choreography Editor

TaRDIS provides an editor and compiler for files whose source language is formatted as a DCR choreography. Its output is the projected behaviour in the form of Java code to be integrated in a fully functional communication framework.

A DCR choreography specifies the messages exchanged between participants as well as constraints on control flow that define causality between events and messages in the system's logic. Such choreographies go beyond the traditional sequential choreography specifications (c.f. sessions). It allows for a more flexible and extendable programming framework for swarms.

Clicking **"Open DCR Editor"** lets you choose a folder where the TaRDIS DCR Editor will be set up. The extension automatically clones the Git repository (or updates it if it already exists), installs all dependencies, and opens the editor within a VS Code WebView, as shown in Figure 15.

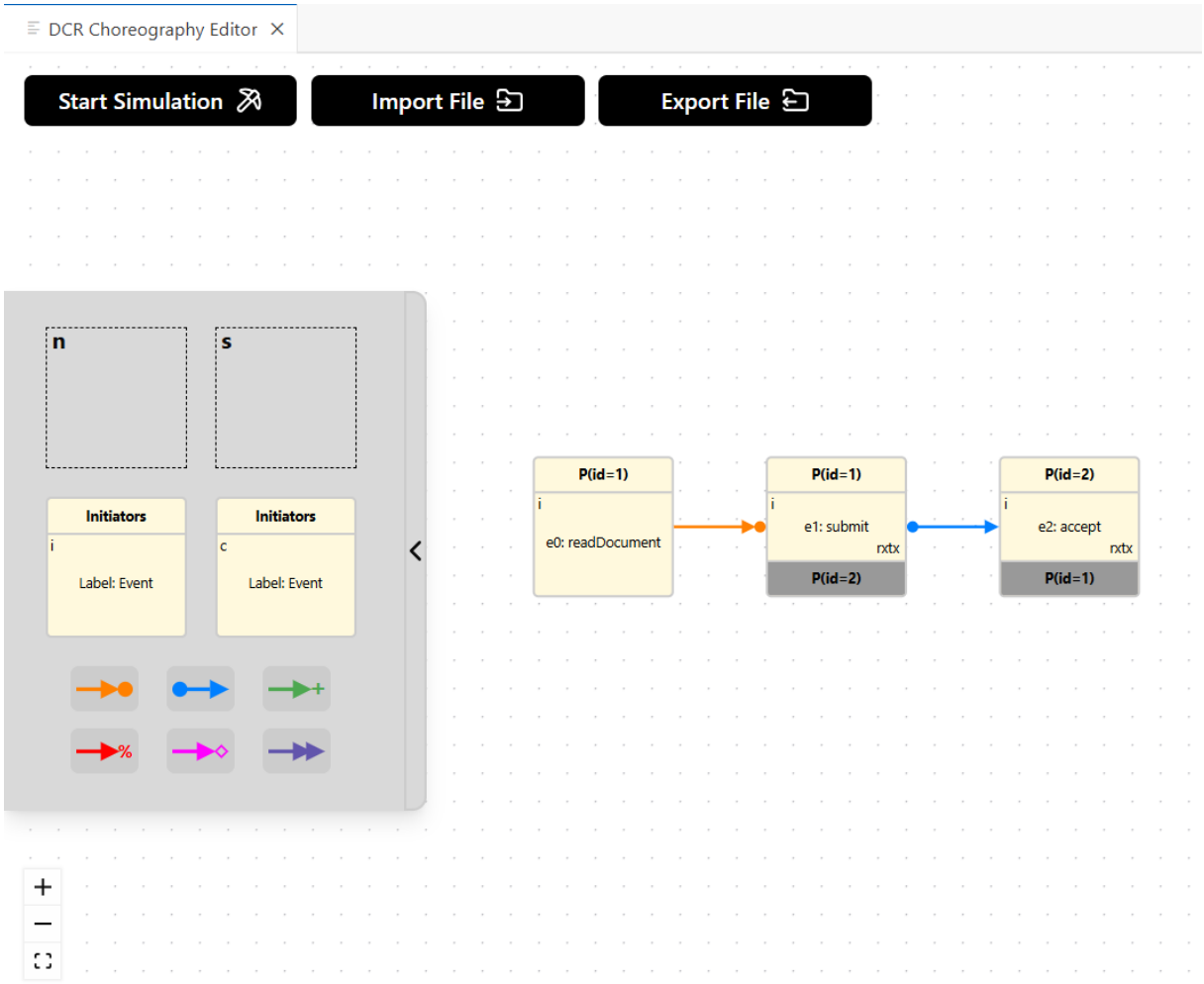

Figure 15: VS Code WebView for DCR editor.

Clicking **"Open Dashboard"** lets the user choose a folder where the TaRDIS DCR Dashboard will be set up. The extension automatically clones the Git repository (or updates it if it already

exists), installs all dependencies, and opens the editor within a VS Code WebView, as shown in Figure 16.

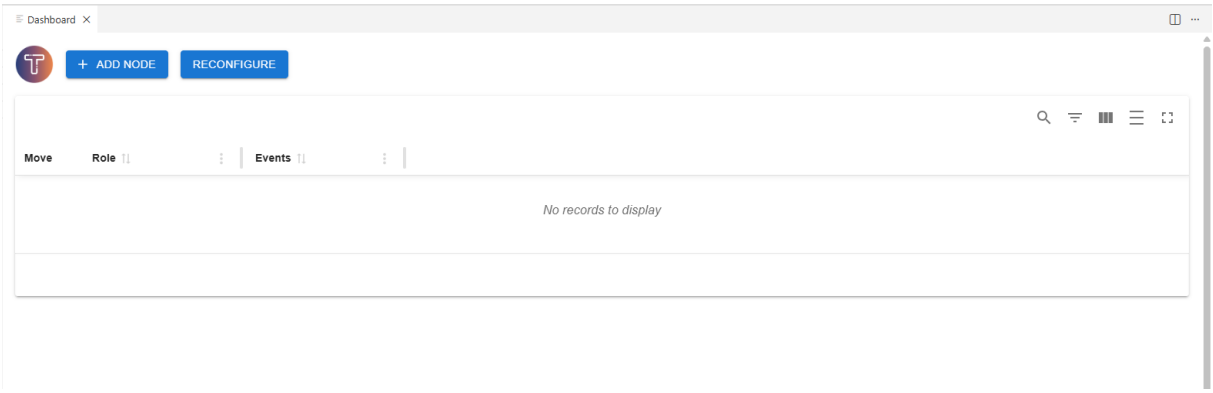

Figure 16: VS Code WebView for DCR dashboard.

The integration of this tool into the TaRDIS IDE streamlines the entire process, from project creation to compilation and execution, by providing an intuitive interface for managing DCR Choreography projects, as seen in Figure 17.

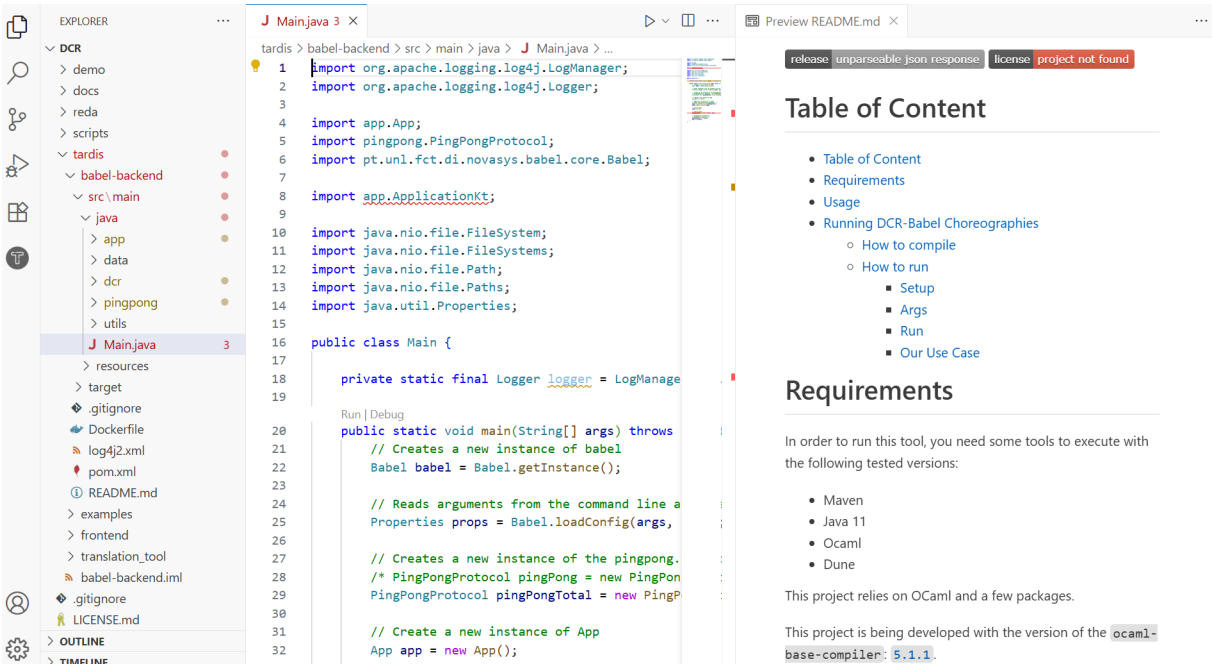

Figure 17: DCR Choreography integrated in the TaRDIS IDE.

When a new DCR Choreography project is created within TaRDIS, the extension automatically generates the necessary folder structure and configuration files to support execution. This predefined workspace ensures that all dependencies are properly set up, allowing users to focus on defining their choreography without worrying about manual setup. The project structure includes a dedicated workspace for the DCR logic, configuration files for managing dependencies, and a preconfigured environment that ensures compatibility with the required tools.

The TaRDIS extension simplifies the execution of DCR Choreographies by integrating directly with Docker, Maven, and other required tools. It provides an interactive interface where users can configure and execute their distributed workflows with minimal setup. Users start by pressing the compile button, then a menu pops-up on the right panel giving the programmer the option of choosing a `.tardisdcr` file from their workspace, which defines the communication behaviour of multiple participants (prosumers) in the distributed system. They can then specify the number of prosumers that will participate in the choreography execution. By default, the extension sets up three prosumers, but this value can be adjusted, as shown in Figure 18.

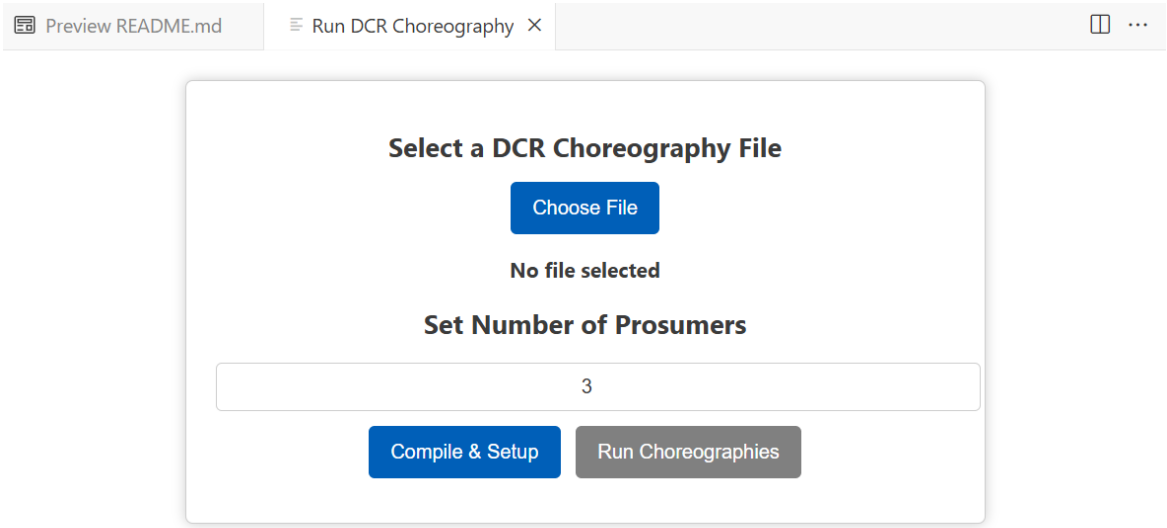

Figure 18: Running a DCR Choreography form.

Once the file and configuration are selected, the compilation and setup process begin. By clicking Compile & Setup, the extension verifies that Docker is running, executes necessary build scripts, compiles and packages the project using Maven, builds a Docker image for running the DCR Choreography, and creates a dedicated Docker network to allow communication between prosumers. These steps ensure that the execution environment is correctly set up before launching the choreography.

With the environment ready, users can execute the choreography by clicking Run Choreographies, which starts multiple terminals, each running a separate prosumer instance inside an isolated Docker container. Each prosumer is assigned a unique identifier (e.g., P_0, P_1, P_2), and the DCR logic is executed according to the provided specifications, simulating a distributed environment. Throughout the execution, users can observe real-time logs in VS Code's integrated terminal, allowing for direct monitoring of message exchanges, event triggering, and overall choreography behaviour.

By following this structured workflow, developers can easily simulate distributed systems using DCR choreographies, ensure correctness in message exchange and event handling through automated validation, seamlessly manage execution environments without manual configuration, and quickly debug and refine their workflows with real-time feedback in VS Code. To enhance usability, the TaRDIS extension also provides a graphical interface for managing

DCR execution. This interface allows users to select a `.tardisdcr` file, set the number of prosumers, and monitor execution progress while displaying any errors that may occur. This integration ensures that users can focus on developing their DCR-based communication logic without needing to manually manage complex execution environments.

3.4 T-WP4-03 JoinActors

JoinActors is a Scala 3 library (developed by DTU) for performing pattern matching on complex combinations of messages/events and conditions. The underlying matching algorithm implements a formal specification of “fair matching” ensuring that, if some incoming message can be matched by a pattern, then that message will be eventually matched and processed.

3.5 T-WP4-06 Java Typestate Checker (JaTyC)

Java typestate checker is a tool that verifies Java source code with respect to typestates. A typestate is associated with a Java class by the @Typestate annotation and defines the object's states, the methods that can be safely called in each state, and the states resulting from the calls. The tool statically verifies that when a Java program runs sequences of method calls obey to object's protocols, objects' protocols are completed, null-pointer exceptions are not raised, and a subclass' instances respect the protocol of their super classes.

3.6 T-WP4-07 Data Centric Concurrency (AtomiS)

This extended Java compiler is used to mark resources which need to be accessed in mutual exclusion; a type-checking and inference system ensures race freedom and produces deadlock free code.

3.7 T-WP4-08 Anticipation of Method Execution in Mixed Consistency Systems (Ant)

Ant is a tool to statically determine operations that can safely commute with other operations and use this information to allow the run-time to anticipate calls to commutable operations. The analysis takes into consideration the consistency policy of each operation.

3.8 T-WP4-09 Correct Replicated Data Types (VeriFx)

VeriFx is a language to design provably correct replicated data types (RDTs), supported by a library of verified conflict-free RDTs.

3.9 T-WP4-10 IFChannel

IFChannel is used to verify that the use of channels (generating and reacting to events) respects the secure information flow policy, e.g., confidential information of some group of

participants is not accidentally transmitted on a channel where non-members of the group can read. This includes implicit flows, e.g., we assume the attacker can see that communication is occurring.

3.10 T-WP4-11 PSPSP

PSPSP is a tool for verifying security protocols that setup and implement the channel (e.g., TLS) or change group memberships and the associated key infrastructure. This is internally used by TaRDIS for verifying security of the communication infrastructure that libraries provide.

3.11 T-WP4-12 CryptoChoreo

CryptoChoreo is an implementation of a verified choreography will not be secure if the actors do implement the behaviour expected by the top-down model. Cryptographic Protocols formulated as a choreography can be translated into local behaviours. Local behaviours can be checked with PSPSP and also an implementation can be derived from them.

3.12 T-WP4-13 (Sec)ReGraDa-IFC and DCR Choreographies

ReGraDa is a compiler and type checker with Dependent Information Flow Control for ReGraDa graphs, mapped onto a centralised graph database, currently being extended to a decentralised version using Actyx as backend runtime support.

3.13 T-WP5-01 Flower-based FL model training

The Flower-based FL model training tool (see D5.3 [17]) provides ML model training by utilizing federated learning solutions. The aim of the tool is to offer an AI/ML library with a set of different decentralized solutions, as well as to provide use case specific solutions. The implementations of FL algorithms are relying on the Flower framework [18]. The tool provides a simple user interface that does not necessarily require expert knowledge to start the training process. The user can select the task that needs to be solved, and the tool provides a list of applicable models and algorithms. The finished training process produces a trained ML model and a training status overview. This provides a customizable and reliable approach that supports the developers' decisions with ease of use.

3.14 T-WP5-02 Data preparation for Flower-based FL model training

The Data preparation for Flower-based FL model training tool (see D5.3 [17]) provides data preparation and preprocessing approaches for the ML model training, with the aim to overcome potential irregularities in the target data set. This includes common approaches, such as dealing with outliers, duplicates, missing values etc., but also some custom data preparation techniques, for example pseudo-labelling, that enables adding labels to an unlabelled data set, when a model requires them. The tool offers the facilitation of the ML training process (it

provides input for the T-WP5-01 tool), by supporting a more efficient process of the preparation of the data.

3.15 T-WP5-03 Flower-based FL Model Inference and Evaluation

The Flower-based FL model inference and evaluation tool (see D5.3 [17]) provides the possibility of getting output for the relevant data on a model trained by tool T-WP5-01. The output can be of different forms, depending on the needs regarding the relevant data, for example, predictions, forecasting, anomaly detections, and metrics. It offers a straightforward approach for gaining valuable insights into the quality of the trained model and for obtaining important conclusions.

3.16 T-WP5-04 PTB-FLA and MPT-FLA

As reported in D5.3 [17], PTB-FLA stands for Python Testbed for Federated Learning Algorithms. It consists of a framework for developing and testing distributed federated learning algorithms. PTB-FLA execution environment provides SPMD (single program multiple data) applications' launching facilities and the simple API (amenable both to AI & ML developers who do not need to be professionals and generative AI tools), which offers the generic centralized/decentralized federated learning algorithms that may be specialized by specifying client and server callback functions.

PTB-FLA is a completely independent solution based on pure Python, without additional dependencies, that is available as open source. It is directly compatible with the main AI & ML libraries and supports the development of both centralised and decentralised federated learning algorithms. This tool was designed to target small IoTs in edge systems, such as Raspberry Pi Pico W boards, ROS2 robots, etc. and supports the FL algorithm development on a single computer. With its simple API, the tool is easy-to-use by non-professional programmers, and it is amenable to LLMs such as ChatGPT. To facilitate easier development of the federated learning algorithms from sequential algorithms, a development paradigm was proposed to guide developers in transforming a referent sequential code into the target PTB-FLA code. The target PTB-FLA code is also easy to transform into the CSP formal model, which can then be used to formally verify the system properties, such as deadlock freedom, termination, etc.

MPT-FLA, which stands for MicroPython Testbed for Federated Learning Algorithms, was developed as a successor of PTB-FLA to support very small IoT edge devices with very limited computational capabilities. It is based on MicroPython, a lightweight version of Python. It supports decentralised applications whose instances can run on Raspberry Pi Pico W boards, robots, and PCs, connected to a Wi-Fi network.

The PTB-FLA and MPT-FLA are already being integrated with the TaRDIS IDE, namely with the inclusion of the creation of a PTB-FLA project, which includes several options, as seen in Figure 19.

The TaRDIS IDE now provides seamless support for the creation and management of PTB-FLA projects, offering a user-friendly environment for the development of federated learning

algorithms. When creating a PTB-FLA project, users have access to various options designed to streamline the setup and development process.

One of the key features is the automatic setup of a dedicated Python virtual environment (**venv_ptbfla**). This ensures that all necessary dependencies are pre-installed, guaranteeing compatibility with major AI & ML libraries and eliminating the need for manual configuration.

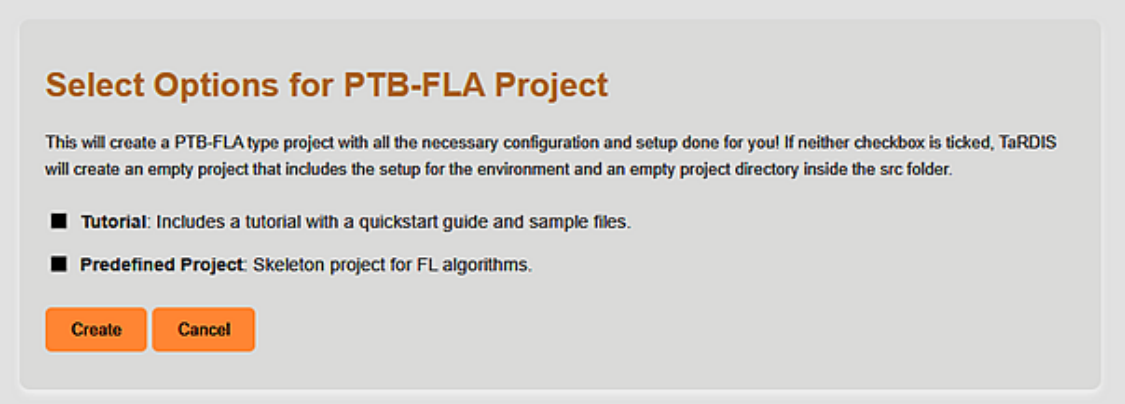

Figure 19: Configuration of a PTB-FLA project.

To accommodate different levels of expertise and project requirements, users can choose between several project structure options, the menu can be seen in Figure 19. The most basic option is the Empty Project, which provides a clean and minimal PTB-FLA project structure, allowing developers to configure it according to their needs. Alternatively, users may opt for the Guided Tutorial, which assists newcomers in navigating PTB-FLA development. In this mode, the left panel of the IDE displays a script for novice PTB-FLA programmers, while the right panel contains a README file with step-by-step instructions, ensuring a smooth learning experience.

For those who prefer a predefined starting point, the TaRDIS IDE offers three structured templates for federated learning algorithms, as depicted in Figure 20:

1. Centralized FLA, which follows a centralized server-client model.
2. Decentralized FLA, enabling fully distributed learning without a central coordinator.
3. TDM Peer Data Exchange, which allows time-division multiplexing for peer-to-peer data exchange.

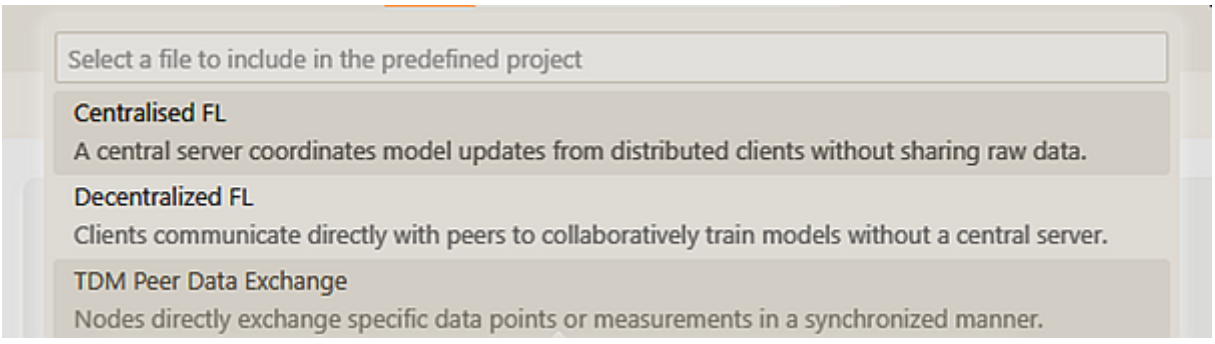

Figure 20: TaRDIS PTB-FLA form for selecting Federated Learning algorithms.

Additionally, every PTB-FLA project includes a custom implementation folder. Upon creation, the TaRDIS IDE automatically generates a `<project_name>_src` folder where users can develop their Python-based Federated Learning (FL) implementations. If one of the predefined FLA skeletons is selected, the corresponding boilerplate code is placed in this folder, significantly reducing the effort required to get started.

These enhancements allow developers to quickly create, customize, and experiment with PTB-FLA projects while ensuring seamless integration with AI & ML workflows. Whether starting from scratch, following a guided tutorial, or leveraging predefined federated learning templates, the TaRDIS IDE provides an efficient and structured environment for federated learning development.

3.17 T-WP5-05 Federated AI network orchestrator (FAuNO)

The Federated AI Network Orchestrator (FAuNO) is a tool providing state-of-the-art agents for Multi-Agent Reinforcement Learning (MARL) working in the collaborative, federated setting. FAUNO is compliant with the PettingZoo API and exploits and specialises the MARL framework for the planning, deployment, and orchestration of the complete TaRDIS framework through federated reinforcement learning and other relevant methodologies.

This framework is trainable with the TaRDIS tool PeersimGym, a MARL environment for training MARL agents, already reported in D5.3 [17].

The FAUNO tool was already integrated with the TaRDIS IDE, namely with the inclusion of the creation of a FAUNO project, which includes several options.

3.18 T-WP5-08 Lightweight, Knowledge Distillation and Pruning

Lightweight Functionalities and Energy-Efficient ML

The lightweight inference functionalities provided by these tools are: (i) The early-exit tool transforms a pre-trained deep neural network (DNN) model in a more lightweight version that includes multiple exits during the model feed-forward for purposes of providing quick inference at the cost of reduced accuracy. In addition, a distributed form of the early-exit tool was developed in order to implement the deployment of the early-exit in a distributed architecture, i.e., model segment among several edge nodes; (ii) The knowledge distillation tool transforms a pre-trained ML model in a more lightweight version in terms of network complexity, number of neurons and ultimately in terms of computational intensity. In specific, the original model is utilized to train a student, more lightweight model, essentially reducing the computational resources required during the inference process at the cost of decreased model accuracy; (iii) The pruning tool again transforms a DNN in a more lightweight version by nullifying the neuron connections that have a negligible impact on the DNN performance. To this end, the pruning functionality streamlines the inference process, in terms of latency and conservation of energy and computational resources.

3.19 T-WP5-09 Decentralised Federated Learning Framework (Fedra)

This tool provides a decentralised federated learning framework integrated with p2p communications between the participating nodes, specifically designed for swarm systems. Fedra leverages libp2p for peer-to-peer communications between the swarm members, enabling the secure model weight exchange for aggregating the federated global model, guaranteeing privacy and data ownership. Moreover, Fedra is model-agnostic, in the sense that different ML algorithms can be seamlessly integrated and utilized to train ML federated models, including models for forecasting, resource allocation, anomaly detection, among others.

It should be noted that the frameworks that have been developed for FL training cover, in principle, different requirements. The Flower-based framework is more standardised, being acknowledged to the open-source community, while revisions and updates are frequent. On the other hand, Fedra deals with completely decentralised learning using p2p communication, without the requirement of a centralized aggregator in the framework. Finally, PTB-FLA framework deals with a more lightweight version of FL, to be deployed in resource-constrained devices.

The VS Code integration of Fedra includes the setup and exploration of Fedra projects by providing an intuitive interface that handles repository management, environment setup, and documentation access. The extension automates the entire workflow—from cloning the repository to opening interactive documentation—and allows users to choose between a full setup, including building the documentation, or a basic project setup.

When a new Fedra project is initialized within VS Code, the extension guides the user through the following steps:

Folder Selection: Users select a folder where the Fedra repository will be cloned.

Repository Management: If the repository does not exist, the extension automatically clones it from GitHub. If the repository already exists, it pulls the latest changes to ensure the project is up to date.

Python Virtual Environment: The TaRDIS extension automatically creates a **venv** if it is not already present, ensuring a consistent Python environment for running Fedra. It also installs all required Python dependencies from requirements.txt, detects the user's platform (Windows, Linux, or macOS) and configures Python and pip commands accordingly to ensure compatibility across operating systems.

Intro Panel: A welcome panel guides users with two options as shown in Figure 21:

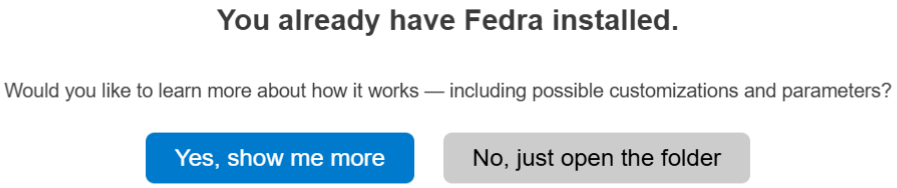

Figure 21: Fedra project after setup options

Open the project and explore the documentation as shown in Figure 22.

Open only the project in VS Code window to begin working with the framework.

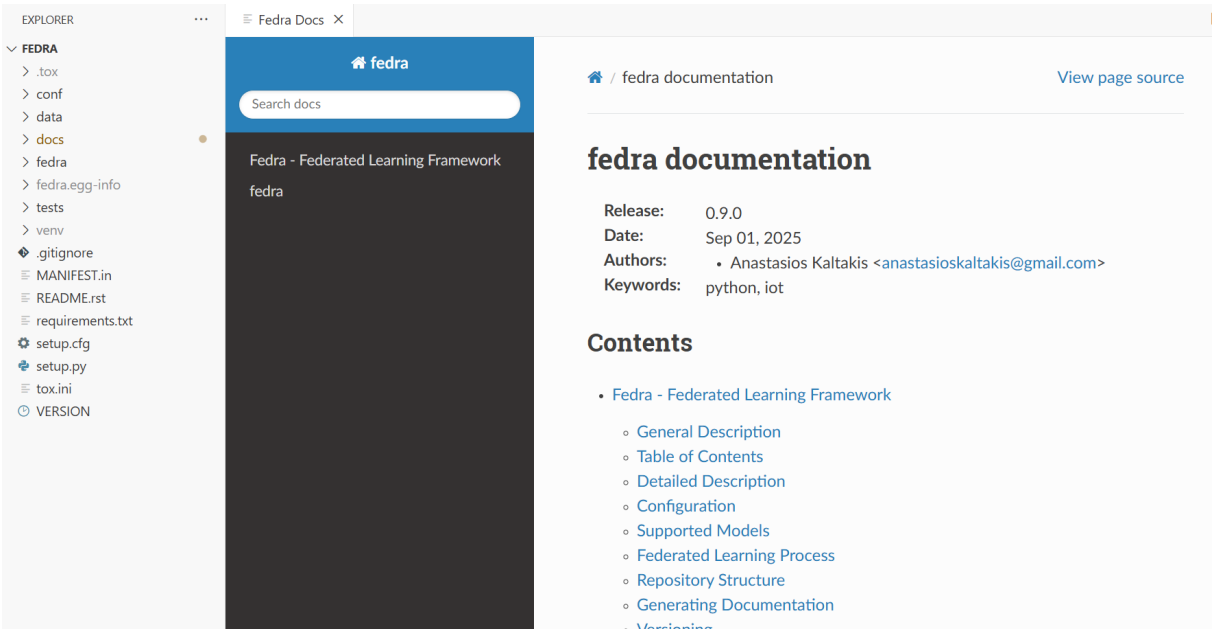

Figure 22: Showing Fedra documentation when clicking "Yes, show me more"

To serve the documentation, the extension starts a lightweight Python HTTP server in the project's docs/build/html directory. Users can explore the documentation interactively without leaving VS Code.

3.20 T-WP5-10 FLaaS

FLaaS (FL-as-a-Service), developed by Telefonica, is a practical federated learning framework for mobile environments that allows app developers to perform cross-device and cross-app (i.e., on-device cross-silo) FL. There are four core components of FLaaS: 1) App Developer Interface 2) FLaaS Server 3) Notification Service and 4) Client Devices. The FLaaS developer orchestrates the FL operations through the Admin Developer Interface and the clients' devices should have installed FLaaS local, which is a standalone service/app for Android devices. FLaaS may be used by a swarm developer in order to initiate and orchestrate a federated learning instance with Android devices and a cloud-based FL aggregator. Lastly, we stress here that, while the initial version of FLaaS was built outside of TaRDIS, the development and subsequent integration of the TaRDIS tools into FLaaS will result in an improved or more modular version of FLaaS.

The implementation of FLaaS into TaRDIS VS Code extension consists of a customized set up depending on the user's goal. First by clicking on the button "Setup FLaaS" it will clone the repository, followed by opening a webview inside VS Code. This webview allows the user to choose between a basic setup or a full setup, as depicted in Figure 23.

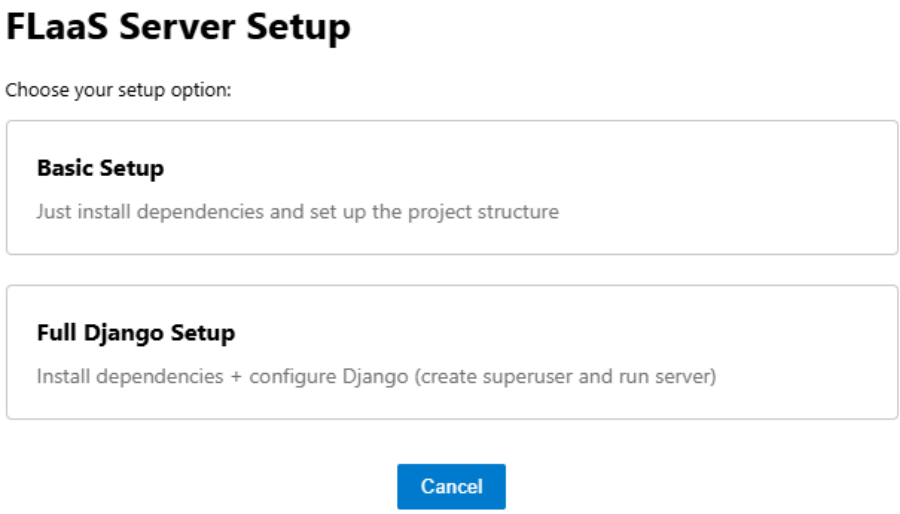

Figure 23: FLaaS Server Setup

Choosing the basic setup will install all the necessary dependencies and project structure to start working with a FlaaS project, while the full setup will guide the user through a step-by-step setup of Django, as well as installing all Django dependencies.

This setup includes a superuser configuration, automatically running the database server and launching a WebView for the Django administrator panel (as shown in Figure 24). The user can now login with the created user (Figure 25).

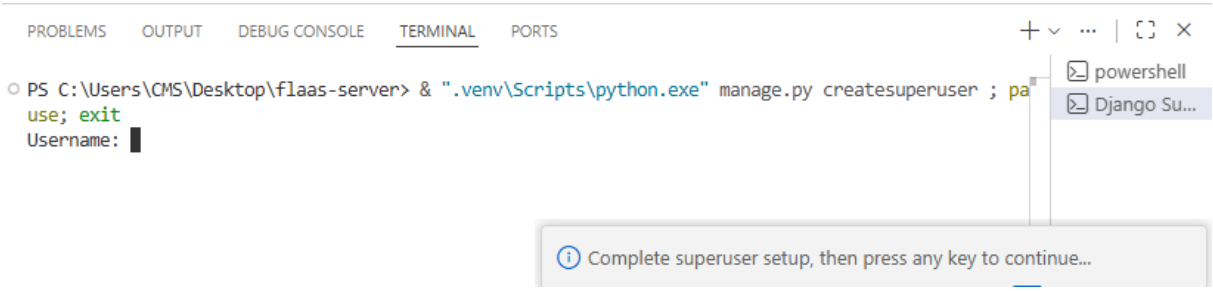

Figure 24: FLaaS superuser configuration.

Figure 25: FLaaS Django Login.

3.21 T-WP5-11 Simulator for Peer-to-Peer Networks

A simulation for training a Reinforcement Learning (RL) agent has been built as tool ML-Gym. It is divided into discrete time steps and can handle various network configurations, i.e., peer-to-peer networks but also networks where there is a hierarchy among the nodes. The simulator runs in Java and must be wrapped into a Python environment following the interface defined as a Markov Decision Process or a Markov Game, in the case of decentralised decision-making. Such RL training environments are embedded in an RL training framework, in our case, we chose PettingZoo. The developed environment models task offloading-based orchestration. It is an open platform that the team plans to extend to broader action spaces.

3.22 T-WP6-01 A Generic API for Decentralised Overlay and Communication Protocols

This tool consists of a collection of protocols (i.e., pre-made interactions between swarm participants for various shapes of communication, to be performed over network connections), a runtime component for managing instantiated protocols, and programming language bindings for interacting with the protocol manager as well as each protocol instance. This allows a TaRDIS application to use higher-level communication primitives than manually opening connections and sending or receiving bytes on them. Examples include efficient routing of messages between peers that are not directly connected or broadcasting from one peer to a group of peers, each offering a range of message delivery guarantees to select from.

3.23 T-WP6-02 An Epidemic and Scalable Global Membership Service

This membership abstraction (provided in the form of a library) allows a TaRDIS application to obtain information on the size of the surrounding swarm and the identities and health of its participants. In contrast to earlier work that only gave a partial view (focusing on a peer's neighbours), this tool aims to provide a global view, albeit with eventual consistency—i.e. after

Funded by the European Union

a membership change there is a delay before this change is reflected in the swarm view presented on all participating peers.

3.24 T-WP6-03 Actyx: Reliable event broadcast with configurable durability

Actyx is a middleware tool that will internally use the above tools for swarm communication and membership to provide even higher-level services to TaRDIS applications, namely the reliable and durable dissemination of event streams within the swarm. This is significant because individual events are quite small and typically don't warrant the overhead of being individually treated (e.g. for being identifiable or localisable) in a swarm system. Therefore, Actyx partitions the emitted events into streams that are then the unit of dissemination, leading to significant benefits in compressed event storage size. Storage resource usage can be controlled via configurable per-stream data retention policies. Actyx also introduces an eventually consistent global order between events that allows the resolution of conflicts arising from concurrent swarm behaviour in a fashion that does not compromise on system availability or resilience.

3.25 T-WP6-04 Babel

This internal tool allows a more formal expression of low-level communication behaviour of an application or algorithm and its validation in a variety of environmental scenarios (i.e., network availability and performance). It will be used to develop and test several of the TaRDIS tools, and it may also be of use to external developers, for example when they create their own communication protocols to be used via the generic communication protocol API. The Babel framework existed before the start of TaRDIS. In the context of TaRDIS Babel has evolved into a full ecosystem for supporting the development of highly decentralised swarm applications compatible with a variety of different devices. This ecosystem is composed of Babel-Swarm which enriched the framework with support for security, self-configuration, and self-management; and Babel-Android which transfers and expands the capabilities of Babel to the Android environment, including mobile phones and tablets.

This tool was integrated into the IDE, consisting of an option to create a Babel project, with multiple options available which may consider the expertise of the developer. When creating a new Babel project, the developer is guided through a structured process to ensure a smooth workflow. The process begins with the selection of a project type, where the user can choose to create a Java-based project utilizing Babel libraries. The IDE then prompts the developer to enter a project name and select a directory where the project should be created.

Once the project details are set, the extension automatically generates the required folder structure, including the necessary Maven configuration files. The setup includes predefined dependencies for Babel, ensuring that the project is ready to be developed without requiring manual dependency management. The generated structure provides a dedicated src folder, where Java classes and the main application logic can be implemented. This automated setup eliminates the need for manual environment configuration, allowing developers to focus on building their applications.

After setting up a new Babel project, the TaRDIS extension offers additional tools that enhance development productivity. Within the TaRDIS project view in the sidebar, users have access to buttons that facilitate common tasks, including the creation of essential Babel components and the importation of predefined communication protocols developed by NOVA. These options allow users to extend functionality of their project without any manual configuration of dependencies onto the `pom.xml` file or writing boilerplate code.

To assist developers in managing their project's communication structure, the extension provides a streamline process for importing existing Babel protocols. When clicking on the "**Import Protocol**" button, the user is presented with a categorized dropdown list of communication protocols available for integration. The protocol selection menu, as shown in the screenshot, allows the user to browse different protocol types, including options such as AntiEntropy, Eager Gossip Broadcast, Flood Broadcast and One Hop Broadcast. This categorization ensures that developers can quickly choose and locate the protocol that best suits their application needs, as seen in Figure 26.

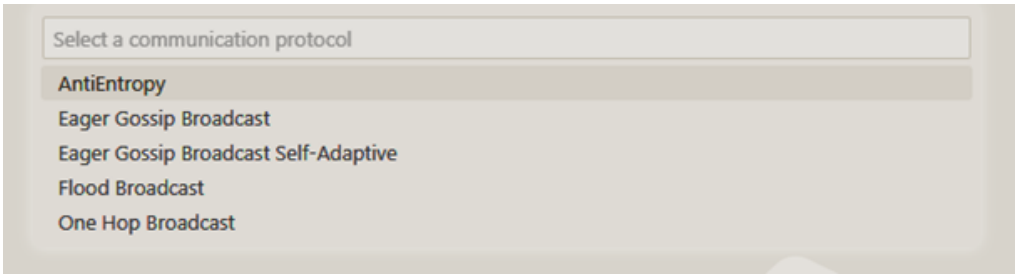

Figure 26: Babel selection of communication protocols

Once a protocol is selected, a detailed information panel appears within the right-side panel, providing an overview of the protocol's functionality, implementation details and configuration parameters. This information is presented in a formatted documentation view, making it easier for developers to understand how the protocol works before deciding to integrate it into their project. The protocol documentation outlines key aspects such as membership services, broadcast mechanisms and required configuration parameters. For example, in the case of the Eager Gossip Broadcast Protocol, the documentation specifies how it handles message dissemination and highlights relevant parameters like the fanout setting.

The information panel also includes two interactive buttons: **Import Protocol** and **Cancel**, as shown in Figure 27. The Import Protocol button confirms the user's selection and proceeds with integrating the protocol into the project. The extension automatically updates the pom.xml file to include the necessary Maven dependencies and repository links. Additionally, it injects the required initialization code into the main application, ensuring that the protocol is properly registered and ready for use. This automated integration significantly reduces the risk of misconfiguration and allows developers to start using the protocol without additional setup.

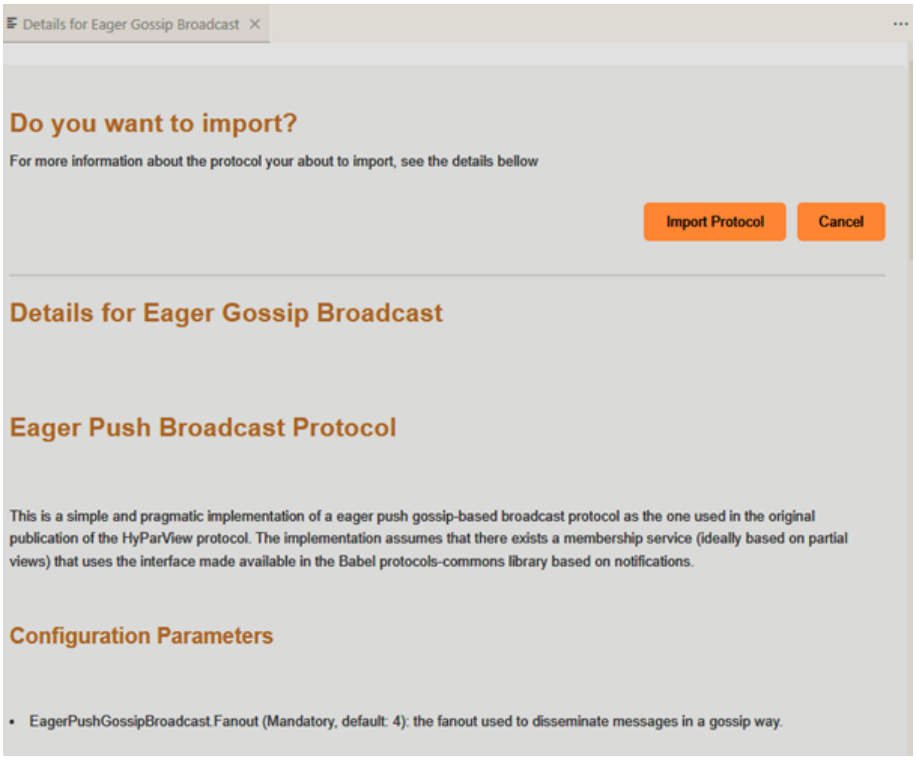

Figure 27: Babel form for confirmation of inserting a protocol.

If the developer decides that the selected protocol is not suitable, they can click the **Cancel** button, which closes the window without making any modifications to the project. This provides an intuitive and non-intrusive way to explore available protocols before committing to an import.

Beyond protocol integration, the TaRDIS extension further accelerates Babel projects development by simplifying the process of creating commonly used Babel components and classes. The **Create Class** button, as seen in Figure 28, enables users to generate structured Babel elements such as Messages, Protocols, Timers, Requests, and Replies. When selecting this option, the user is prompted to choose the type of class they wish to create and provide a name for it. Once confirmed, the extension generates the necessary Java class file with predefined templates, ensuring that it is structured correctly and follows best practices for Babel-based development.

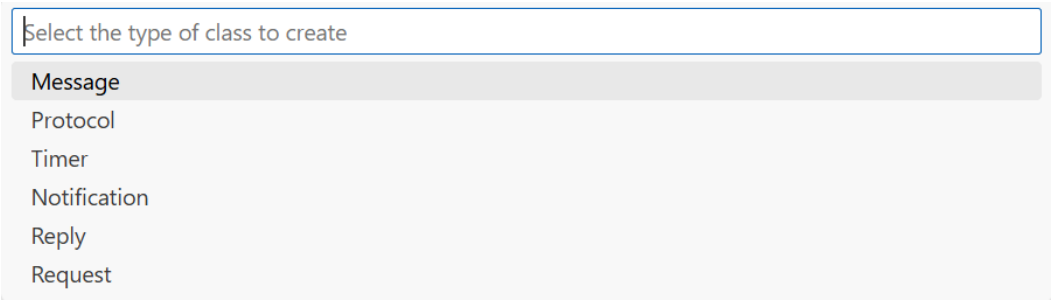

Figure 28: Babel form for creation of a class.

By incorporating these functionalities, the TaRDIS extension provides a comprehensive toolkit for developing distributed applications in Java using the Babel framework. The seamless

workflow, from project creation to protocol integration and component generation, allows developers to build robust communication-driven applications efficiently. This combination of automation, predefined templates, and interactive documentation makes TaRDIS an invaluable extension for managing Babel-based projects within Visual Studio Code.

3.26 T-WP6-05 Arboreal: Extending Data management from Cloud to Edge leveraging Dynamic Replication

Arboreal is a data management tool that replicates key-value bindings across data centres and dynamically distributes updates to these bindings within each data centre according to the declared interests of each edge node. Due to the way updates are propagated and using the included metadata, the system ensures so-called causal+ consistency which means that updates become visible at any swarm participant in an order where causality is preserved (i.e. you only see an effect once you have already seen the corresponding cause, and you will eventually see all changes) while ensuring that all replicas of a data objects eventually converge to the same value. This makes Arboreal a fully available and high-performance NoSQL database a.k.a. key-value store.

3.27 T-WP6-06 PotionDB: Strong Eventual Consistency under Partial Replication

PotionDB is a data management system designed to be deployed in a small number of nodes, potentially geo-distributed, and with support for partial replication. Unlike Arboreal, PotionDB supports a transactional API, thus providing a more powerful API for the application. Still under development, it is the support for materialised views over geo-partitioned data, providing a mechanism for supporting recurrent queries that are common in applications.

3.28 T-WP6-07 Integration of Storage Solutions into the TaRDIS Ecosystem

Similar in spirit to the generic API for communication, this library facilitates the use of a range of storage solutions by a TaRDIS application through a common API. Current solutions other than the above include the industry standard Cassandra (also with C3 enhancements for causal+ consistency), Engage, and Hyperledger Fabric. The latter will allow to easily leverage on blockchains in the design of TaRDIS use cases, which will provide a tamper-proof and publicly verifiable ledger where for instance, exchanges between members of the swarm can be reliably registered, for instance, energy exchanges among elements of the renewable energy community.

3.29 T-WP6-08 Distributed Management of Configuration based on Namespaces

This tool allows swarm administrators to inject the desired configuration for any participant or application running on it. Parameters are selected based on namespaces to isolate parts of

the system from each other, which allows a large measure of heterogeneity within the swarm: applications do not need to be co-designed to guard against interference, and even different versions can be clearly and cleanly separated. In addition, labels are used to allow fine-grained grouping of any kind of resources, allowing homogenous configuration where required.

3.30 T-WP6-09/10 Telemetry Acquisition for Decentralised Systems

These tools (one for containers and one for Babel protocols) consist of a manager, registries, and exporters for a wide variety of measurements performed within a running swarm. Due to the dynamic nature and usually complex communication topology of such systems, dedicated tooling is necessary to transport this data from the device where it originates to the person or ML algorithm that monitors the swarm. The purpose is to enable the smooth operation of the system, which includes quick incident response as well as proactive management of resources to avoid incidents. This is currently being evolved to take advantage of in-network processing by swarm elements to ensure that telemetry can be effectively and efficiently processed and disseminated to elements of the swarm that make decisions related to the current configuration.

Funded by the European Union

4 TaRDIS Developer Stories

The TaRDIS Development Approach is designed to be flexible and compatible with different Software Development Life Cycles (SDLC). The TaRDIS tool stack can be integrated with existing SDLCs for swarm application development.

The main contribution of this deliverable is a catalogue of *recipes* illustrating the **TaRDIS Development Approach**. These recipes were collected from the TaRDIS case studies and can serve as an initial set of blueprints for swarm application development. That said, the possible scenarios for the TaRDIS Development Approach are, naturally, much wider, so new recipes may be developed in the future.

4.1 Introduction

Decentralized, heterogeneous swarm applications present challenges that traditional, centralized programming paradigms are not prepared to address. The TaRDIS project tackles these challenges by offering a distributed development approach and supporting toolset. This simplifies the development and operation of such applications.

The main contribution of this document is the introduction of TaRDIS development recipes. These recipes are structured, reusable templates. Concrete case studies serve as guidelines for solving specific problems within the swarm application domain. Developers struggle to integrate tools like communication middleware, formal verification libraries, and federated learning frameworks.

The TaRDIS recipes serve as blueprints for effectively combining these tools to achieve a desired functional outcome. The TaRDIS approach is agnostic to the Software Development Lifecycle. In this catalogue, some recipes follow a Waterfall-like approach, others use Scrum, and others use DevOps. By including verification activities, the recipes guide developers to spot bugs, such as deadlocks or confusion in swarm protocols, before finishing implementation. The TaRDIS IDE supports the transition from problem domain (requirements) to solution domain (executable code). It acts as the primary interface for invoking recipe-defined tools and managing a project's heterogeneous dependencies.

By documenting these models and APIs in a programmer-oriented format, this report aims to lower the barrier to entry for developers seeking to harness the power of swarms, ensuring that the resulting systems are not only performant but also robust and privacy-preserving.

This section is structured as follows:

Section 4.2: Development Approach Overview establishes the technical and management framework for the TaRDIS methodology. It defines the core activities and presents the structure that underpins the software process across the different recipes.

Section 4.3: TaRDIS Recipes provides a comprehensive catalogue of 28 specialized recipes across domains such as aerospace, manufacturing, energy distribution, and intelligence for smart-home environments.

Funded by the European Union

Section 4.4: TaRDIS IDE Overview and Discussion outline the centralization of these tools within the TaRDIS IDE, an extension of the Visual Studio Code environment.

4.2 Development Approach Overview

4.2.1 What is a software development approach?

4.2.1.1 Defining software development approach

A **software development approach** defines a set of activities and associated results which produce a software system. It establishes the technical and management framework for applying methods, tools, and personnel to software development tasks. It also guides software developers as they work by identifying their tasks and roles within the development process.

4.2.1.2 Software development approaches activities

The development process encompasses several activities that typically occur, regardless of the specific development approach. These activities include a **feasibility study**, **software specification**, **development**, **verification and validation**, **maintenance** and **evolution**. A broader view of the software process also includes its **deployment** and **operation**, although these activities were traditionally separated from the development itself. This is no longer the case with some modern approaches, such as DevOps.

The **feasibility study** is a focused effort to determine if a system contributes to organisational objectives, if it can be engineered with the available technology and resources, including budget, and the extent to which it will integrate well in the existing ecosystem of systems the new system is expected to interact with.

The **software specification** aims to elicit both functional and non-functional requirements from relevant stakeholders, producing a detailed system specification closely aligned with the problem domain.

The **software development** includes the design and implementation of the software system. This design encompasses both high-level (architectural) design, where the main system components and their relationships and distribution among devices are defined, and lower-level design of the components, their interfaces, used databases, and so on. The design provides a transition from the problem domain to the solution domain. The detailed design specifications are then implemented, thus creating executable software systems.

The software also undergoes a process of **verification and validation**. Verification checks the extent to which software is built according to its specifications. Validation assesses the extent to which the software meets its external stakeholders' (typically, its customers) requirements.

Successful software systems are likely to change over their lifetime due to changes in requirements, customer needs, or in the context they operate in (e.g., due to legal constraints evolution). Software **maintenance** and **evolution** is the process guiding this change over time.

Finally, **delivery** and **operations** include the processes, activities, and tools involved in deploying, operating, and supporting software applications throughout their lifecycle. This includes installing, configuring, testing, releasing, monitoring, and maintaining software products in operational environments [19].

4.2.1.3 Software Development Life Cycle Paradigms

Software life cycles can typically be categorised into one of the following paradigms [20]: **predictive**, **iterative**, **evolutionary**, **incremental**, or **continuous**.

Predictive life cycles have the project scope, time, and cost determined in an early phase [21]. They depart from the assumption that the set of requirements will not undergo substantive changes, except in the event of severe, unforeseen circumstances. The waterfall model [22] is an example of a predictive life cycle.

Iterative life cycles have the project scope determined early in the life cycle but routinely update time and cost estimates as the project team's understanding of the product increases. The product is developed through a series of cycles, with successive increments to the product's functionality [23][21][24].

Evolutionary life cycles involve the product or service changing over its lifetime, either because requirements change or because they are introduced progressively, rather than as a complete set.

Incremental life cycles are adaptive project life cycles in which the product is produced in a series of iterations that add functionality within a specified time frame. The software product is considered complete only after its final iteration [23][21][24]. In these life cycles, the various development activities, from requirements to testing, are performed to ensure the incremental completion of the software product or service [25].

Continuous development is a set of practices that support frequent releases of new systems using automated tools.

4.2.1.4 Software Development Life Cycles

There is a plethora of software development life cycles that instantiate the paradigms mentioned in section 4.2.1.3. In this section, we present some of the most representative ones.

Waterfall Model

The waterfall model is a predictive, document-driven, model where each phase results in deliverables, including documentation, that serve as inputs to subsequent phases. This results in an order that is followed sequentially, where the next phase cannot start until the previous one ends. The approach is inspired by other engineering approaches and provides visibility to management, performing better in projects where requirements are well-understood and unlikely to change. The approach is suitable for large projects with multi-disciplinary teams, or strong subsystem integration needs. It leads, however, to an inflexible partition of the project into stages, with potentially high costs in the event of iteration and slow reaction to change, as well as a long time before the customer can see the system. That said, it is possible to have

Funded by the European Union

iterative variations of the waterfall model, where new functionalities are added in each iteration. This approach is followed in several of the examples provided in this report.

Agile and Scrum

Agile methods were created as an alternative to the apparent overhead associated with predictive, heavy-weight approaches (*e.g.*, the Waterfall model) used in large-scale software projects [20]. In contrast, Agile methods are lightweight and have relatively short iterative life cycles, and emphasize individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan [26]. There are several Agile approaches, including Rapid Application Development (RAD), eXtreme programming (XP), Scrum, Feature-Driven Development (FDD) and Lean Software Development. Agile approaches are particularly suited for creating software with volatile requirements, due to their focus on quickly adapting to change. However, most Agile methods suffer from a lack of visibility for management, resulting from self-organising teams, insufficient documentation practices, and a focus on short cycles which may hamper a longer-term perspective. Although an agile approach, Scrum attempts to mitigate this lack of visibility for management, with a *Scrum master* managing the activities within a *sprint*. A product owner selects which items are put in the product backlog, what are their relative priorities, and estimated associated effort. The development team members then select and implement the items from the backlog. By the end of each sprint, a new working version of the system is tested and released. The process also includes short daily scrum meetings, to monitor progress. Agile approaches make it easier to react to change, when compared to planned approaches like the waterfall approach, which have a longer feedback loop and require a heavy collaboration between different roles in the software team.

Agile approaches struggle with infrequent (often manual) error-prone deployments. There is a split between delivery (the process of deploying applications into production environments in a consistent and reliable way. The delivery phase also includes deploying and configuring the fully governed foundational infrastructure that makes up production environments) and operation (maintaining, monitoring and troubleshooting software in production environments).

DevOps and Continuous Delivery

Continuous delivery is a set of practices and principles that are aimed at building, testing and releasing software faster and more frequently. It includes three key components: Continuous Integration, Continuous Deployment and Continuous Experimentation. In Continuous Integration, developers frequently commit their code changes into a central repository, with automatic builds and acceptance tests, that serve as a quality gate, providing immediate feedback to developers, so they can fix any detected issues. The process is continuously repeated, to keep the codebase deployable. Continuous deployment means the decision to deploy is often automatic, although human oversight may also be required (and supported), when necessary. Continuous experimentation implies running experiments, comparing different versions of the software, on a continuous basis in the production environment, using strategies such as a canary release, A/B testing, or a dark launch. These practices are crucial for DevOps, a combination of development and operation (in the old days one team developed and handed it over to others to operate it). In DevOps, the team that develops, also operates

and runs, gets feedback and evolves the software system, so the traditional split between the development, quality assurance, and the operations team does not exist.

4.2.2 TaRDIS Principles & the Software Development Approach

The TaRDIS development approach supports the development, maintenance and safe evolution of heterogeneous swarm applications. By design, the TaRDIS development approach is integrated into existing development methodologies. In this document we present a set of **TaRDIS development recipes** that illustrate, via TaRDIS case studies, several examples of how TaRDIS can be used in the context of several software development approaches.

4.2.3 TaRDIS Development Recipes Overview

This section presents an overview of the TaRDIS development recipes. In the following chapter, we will show the TaRDIS development recipes one by one. Here, we have compiled all the recipes together to provide a quick overview. We recommend using this overview to find the recipes that best match your use case. This section concludes with a summary of the presentation approach used to describe the various recipes.

4.2.3.1 Recipes Overview

A **TaRDIS development Recipe** is a structured reusable template, instantiated with a concrete case study, that provides guidelines for solving specific problems within the swarm application development domain. The recipes can be used as blueprints that illustrate how to combine TaRDIS tools to achieve a desired outcome.

4.2.3.2 Recipes Structure

We present TaRDIS recipes following a common document structure. Each recipe has a descriptive **title** for it. This is followed by a brief description of the **scenario and requirements** covered by the recipe. Then, we present an **activity diagram**, where we depict the roles and responsibilities within the development team. This is followed by a description of the **software development life cycle** used in this recipe, as well as which of the process steps are supported by which TaRDIS tools, and how. We then present an overview of the **software architecture**, including the set of **tools** used to build a running system from this recipe, and how these tools combine within the architecture. We also include a discussion of if and how **verification** activities play a role in the recipe, and which tools support it. Finally, we discuss if and how **Machine Learning** tools are used in the recipe.

4.2.3.3 Tool Usage Matrix

This section presents a consolidated view mapping TaRDIS tools to their functions in the development recipes, offering TaRDIS users a quick overview of the mapping of the TaRDIS ecosystem to the development process (see Table 1).

Funded by the European Union

Table 1: TaRDIS tools usage matrix.

TaRDIS tool	Primary function	Applied in recipes
Actyx Toolkit (machine-runner and machine-check libraries)	Finite state machine implementation & static protocol verification.	4.3.1, 4.3.2, 4.3.3
Babel	Decentralized communication middleware.	3.4, 3.5, EDP Use Case (3.7-3.16)
DCR Choreographies	Declarative specification, verification, & execution of multi-party workflows.	EDP Use Case (3.7-3.16)
PTB-FLA	Framework for decentralized/federated ML applications (P2P data exchange).	3.4, 3.5
Fedra/Pruning Tool	Specialized toolset for decentralized FL (LSTM models) & model pruning.	EDP Use Case (3.7-3.16)
Coordination Application	ML-based component to select & trigger DCR choreographies.	EDP Use Case (3.7-3.16)
Nimbus	Fully decentralized storage system for scalable data sharing.	3.4, 3.5
Flower-based FL Tool	Standalone solution for FL models for anomaly detection.	3.17
Early-Exit (EE) & D-Exit Tool	Framework for splitting a DNN for distributed deployment & inference.	3.18
TaRDIS IDE Extension	VS Code extension to streamline project creation &	3.6, 3.12 (Applicable to all)

	deployment.	

4.3 TaRDIS RECIPES

4.3.1 Composition of (Preexisting) Swarms

4.3.1.1 Scenario and requirements

Use Case

Composition of independently developed swarms to create a larger, functionally integrated system (see Figure 29).

- **Scenario**: Two or more swarms, possibly preexisting, each realizing a given swarm protocol, with distinct roles and behaviours, are combined to model a larger coordinated system.
- **Requirements**:
  - **Modularity**: Swarms must be implemented to allow adaptation for composition without modification.
  - **Confusion-freeness**: Swarm protocols must be confusion-free and without concurrency.
  - **Role-Based Interfaces**: Shared roles act as interaction points between protocols. Each event type may only be emitted by one role.
- **Result**: A swarm that:
  - Correctly realises the composition of all input swarm protocols.
  - Is eventually faithful.
  - Contains an adaptation of each machine of an input swarm.

Funded by the European Union

4.3.1.2 Activity diagram

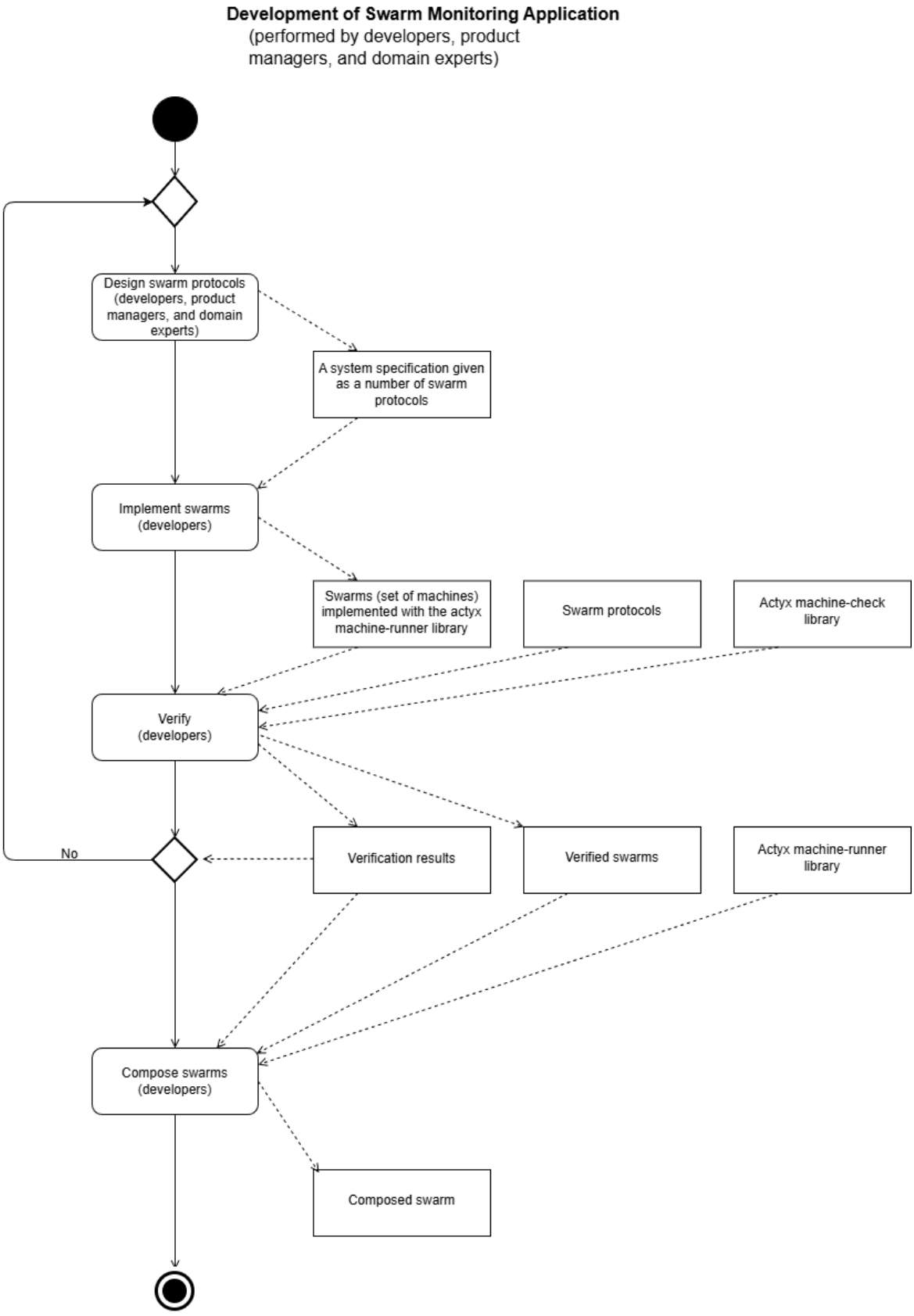

Figure 29: Composition of (preexisting) swarms.

- **Phase 1 (Design swarm protocols)**
  - Developers, product managers, and domain experts design the swarm protocols or update them in order to ensure they pass the verification phase. Some swarm protocols used to form the system may already exist and have implementations. These can be reused.
- **Phase 2 (Implement swarm protocols)**
  - The developers develop swarms implementing the swarm protocols using the `machine-runner` library or update swarms in order to ensure they pass the verification phase.
  - The result of this phase is a set of swarms with their corresponding swarm protocols implemented using the `machine-runner` library.
- **Phase 3 (Verify swarm protocols)**
  - The inputs to this phase are: a set of swarms with their corresponding swarm protocols implemented using the `machine-runner` library.
  - The `machine-check` library automatically checks whether (1) the swarms realise the corresponding swarm protocols, (2) the swarms are non-concurrent and confusion-free, and (3) that the swarms interface correctly.
  - The outputs of this phase are the verification results which are either positive or a detailed message about which property is not satisfied and why.
  - If the verification fails, the team returns to the design phase and updates the swarm protocols.
- **Phase 4 (Compose Swarms)**
  - The inputs to this phase are a set of swarms with their corresponding swarm protocols implemented using the `machine-runner` library.
  - The `machine-runner` library automatically adapts all machines in the given swarms such that they form a new swarm that realises the composition of the corresponding input swarm protocols.
  - The output is an eventually faithful swarm that realises the composition and consists of adaptations of the input machines.

4.3.1.3 Software Development Life Cycle (SDLC)

The software development life cycle used to develop a composed swarm could be described as a variation of the Waterfall model with some iteration (see Figure 30). Specifically, the phases could be described as:

1. **Conceptualization, requirements analysis and design**
   - Corresponds to the first activity. This would be carried out by product managers, domain experts and developers. The end goal is to develop a specification of the system as a number of swarm protocols that can be composed. Some swarm protocols used to form the system may already exist and have implementations from other use cases. These can be reused.
2. **Implementation**
   - Corresponds to the second activity. Here, each swarm protocol specified in the previous activity is implemented as a swarm of machines using the Actyx machine-runner tool.
3. **Verification**

- Corresponds to the third activity. The *well-formedness* swarm protocols are verified and the machines implemented in the previous step are verified against their swarm protocols.
- If verification fails, the first phases are repeated

4. **Deployment**
   - The swarms verified in the previous step are composed and the resulting composed swarm is deployed. Further assessments on the deployed system may lead to revising the requirements or implementation, and thus to an iteration of the previous phases

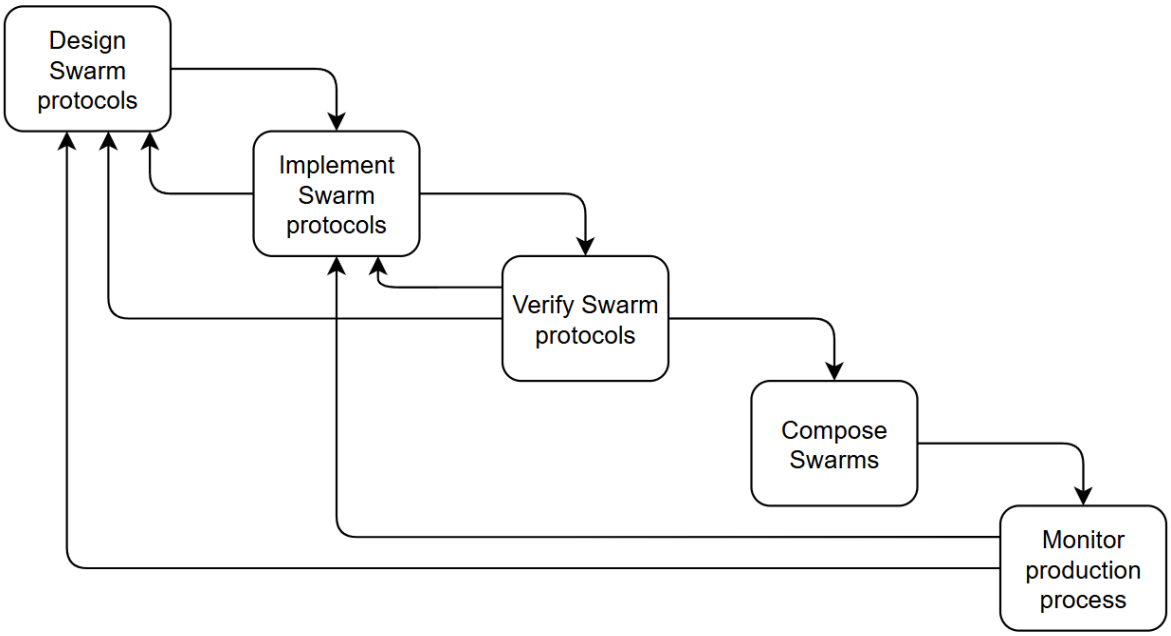

Figure 30: Composition of (preexisting) swarms SDLC.

4.3.1.4 Architecture and Tools

- **Tools:**
  - Custom version of the Actyx toolkit developed as part of research on compositional implementation and verification of swarms. Specifically, the custom versions of the `machine-check` and `machine-runner` libraries.
- **Architecture:**
  - The `machine-check` library is used to:
    - Statically verify swarm protocols and swarms implementing swarm protocols.
    - Infer and verify the conditions under which correctly implemented swarms can be composed into a larger swarm.
  - The `machine-runner` library is used to:
    - Implement the initial swarm modules.
    - Automatically adapt these swarms into a functionally integrated swarm system.
    - Deploy the finite state machines making up swarms.

Funded by the European Union

- The machine-runner library in turn depends on:
  - The Actyx SDK for event transmission
  - The `machine-check` library for the conditions under which a number of swarms can be safely composed.

4.3.1.5 Verification

Static Verification of inputs is automatically performed by the `machine-check` library before the composition ensuring that:

- Machines from each swarm behave correctly when projected onto the composed system. This involves checking their state transitions and event emissions to align with the original protocols.
- Each swarm protocol is confusion-free and non-concurrent.
- Each event type is only emitted by one role in all inputs. This ensures that the shared roles between protocols allow communication without conflicts.

Additional modular verification:

The input machines as well as the swarm protocols are modelled as finite state machines which allows for a multitude of additional verification tasks. Since the composition only restricts the original behaviour, it is usually sufficient to verify the input machines and protocols individually.

Additional verification of the composition:

Additionally, the composed swarm protocol can be constructed automatically by the `machine-check` library and be verified against different properties, which are not suitable for modular verification (e.g. liveness properties).

Verified Result:

The composed swarm constructed by the `machine-runner` library is guaranteed to be eventually faithful to the input swarm protocols.

4.3.1.6 Machine Learning

Machine learning is not used in this recipe.

4.3.2 Monitoring of Swarm Activity and Quality of Service

4.3.2.1 Scenario and Requirements

Use Case

Runtime monitoring of swarm activity and quality of service.

- **Scenario:**
  - Runtime monitoring of swarm systems using fair join pattern matching.
  - Gather information about swarms e.g. to quantitatively measure quality of service.
  - The events generated at runtime by the machines in a swarm are monitored for message combinations that match certain join patterns.
  - An example pattern arising in a factory automation setting that could be expressed and recognized using join patterns is a time-out between when a machine on the factory shop floor fails and when it is brought back into operation (provided corresponding events are emitted).

- **Requirements**:
  - A *swarm* implemented with the Actyx toolchain.
- **Result:**
  - A swarm implemented using the Actyx toolkit supporting monitoring with join patterns with:
    - *A forwarding node* that joins the swarm and forwards every emitted event to a monitoring node.
    - *A swarm monitoring node* that receives all events emitted in the swarm and looks for combinations of messages that fits one of the given join patterns.

4.3.2.2 Activity Diagram

Figure 31 presents the activities leading to the development of a swarm monitoring system.

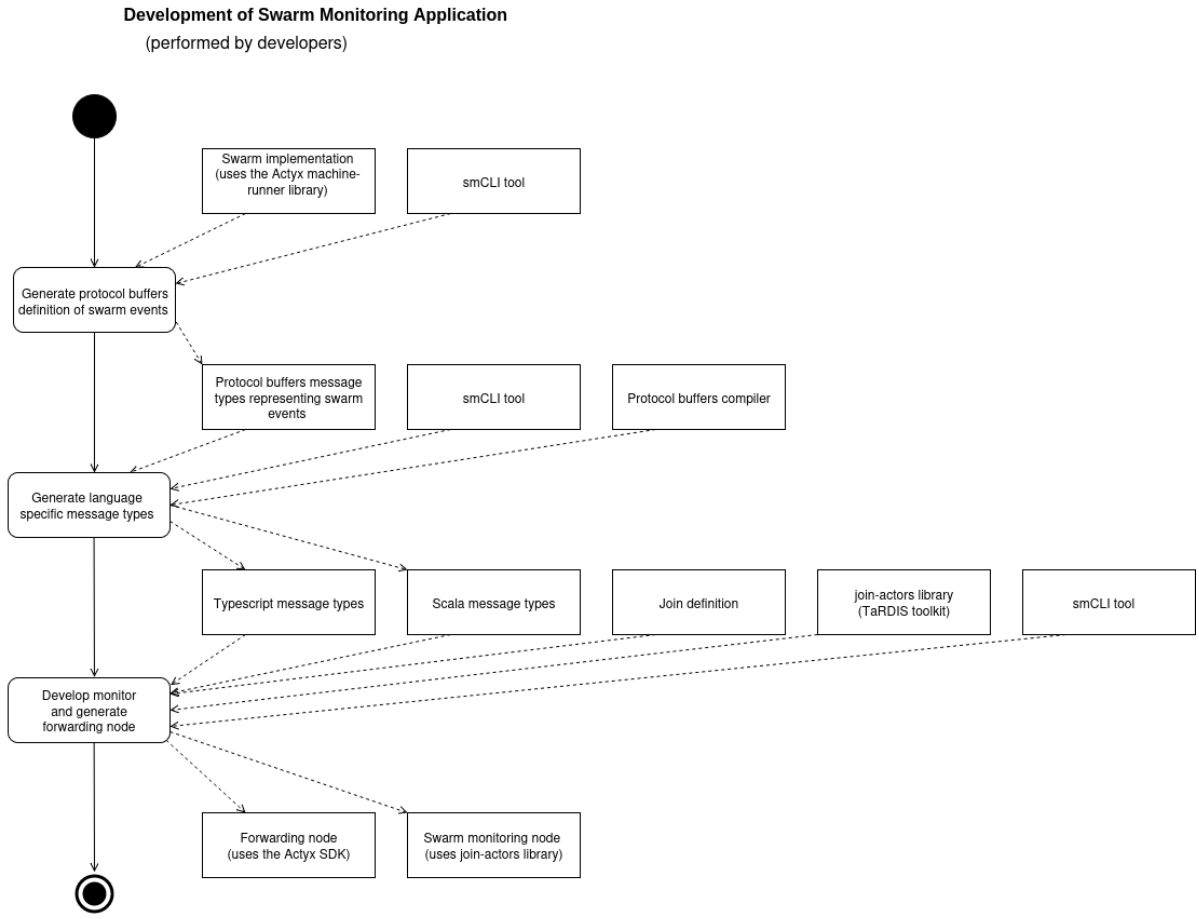

Figure 31: Development of swarm monitoring applications.

- **Phase 1 (Develop Protocol Buffers definition of swarm events)**
  - The input is a swarm implemented using the machine-runner library.
  - Using a custom CLI-tool called smCLI,[2] the event types of the swarm (TypeScript object types) are translated to a language independent format (protocol buffers) to ensure a common, efficient and type-safe representation message format used between the swarm and the monitor.

[2] smCLI is available in the TaRDIS toolkit demo

- The output of this phase is a set of protocol buffers message definitions – language independent representations of the swarm event types.

- **Phase 2 (Generate language specific message formats)**
  - The inputs to this phase are: (1) the protocol buffers definition of the swarm events, (2) the smCLI tool, and (3) a protocol buffers compiler.
  - The outputs of this phase are TypeScript and Scala representations of the messages. The TypeScript representations are generated by the smCLI tool (using a protocol buffers-to-Typescript compiler under the hood) and the Scala representation is generated using a protocol buffers-to-Scala compiler.
- **Phase 3 (Develop monitor and code to deploy forwarding node)**
  - This phase requires (1) the TypeScript representations of the message types, (2) the Scala representations of the message types, (3) the join-actors library, (4) a join definition, and (5) the smCLI tool.
  - The join definition describes the message combinations that messages received by the monitor from the swarm are to be matched against. During this phase (1) the join definition is implemented using the declarative API provided by the join-actors library (possibly with the assistance of a domain expert) and wrapped in an actor that receives messages from a network communication endpoint; then, (2) the forwarding is automatically generated using the smCLI tool.
  - The outputs to this phase are the forwarding node and the swarm monitoring node.
  - Ongoing work aims to automatically generate the boilerplate code for deploying the monitoring node.

4.3.2.3 Software Development Life Cycle

The software development life cycle used to develop a monitored swarm could be described as a variation of the Waterfall model. Developing the forwarding node and the TypeScript specific message types is done by applying the smCLI tool. Developing the monitoring node is more involved as it requires designing the join definition and implementing it using the join-actors library. Generating Scala specific message types is done automatically by applying a protocol buffers-to-Scala compiler in a build script.

4.3.2.4 Architecture and Tools

- **Tools:**
  - The Actyx toolchain (specifically the machine-runner library and the Actyx SDK).
  - The join-actors library.
  - The smCLI tool.
- **Architecture:**
  - Protocol buffers message types are automatically generated based on the event types of the swarm under consideration.
  - Language specific representations of the message formats are generated (one for TypeScript and one for Scala).

- A *forwarding node* using the Actyx SDK is automatically generated and joins the swarm. The forwarding node receives all messages emitted in the swarm, but does not influence the execution of the swarm. Whenever it receives a message it forwards it to the *swarm monitoring node*.
- The swarm monitoring node is a Scala program that listens for messages on a communication endpoint and processes all incoming messages using an actor written with the join-actors library.
- The join patterns used for message matching by the monitor depend on the application and are written by a developer with domain expertise.

4.3.2.5 Verification

- The swarm under monitoring is statically verified during its development to guarantee its fidelity to a swarm protocol.
- The monitor is guaranteed to match messages *fairly*: when a join pattern can match multiple combinations of messages, the combination that consumes the oldest messages is prioritised.

4.3.2.6 Machine Learning

Machine learning is not used in this recipe.

4.3.3 Reprogramming critical elements of a manufacturing process

4.3.3.1 Scenario and requirements

Use case

The use case involves developing a software system intended to reprogram critical elements of the manufacturing process within a factory, ensuring that it is:

- Reusable
- Controllable
- Testable
- Easier to implement
- Maintainable

Reprogramming the manufacturing process to accommodate different or additional workloads is often challenging and susceptible to errors. Production must typically be halted to enable engineers to update and install new software, which takes place during scheduled maintenance periods. This procedure requires detailed planning and adherence to time-intensive protocols, regardless of the chosen development methodology.

This use case seeks to develop a software solution addressing these challenges. Leveraging Actyx and its ecosystem (mainly the machine-runner and machine-check libraries) as the foundation, the solution abstracts the concepts of factory components and machines, supporting a configurable and programmable pipeline based on declarative implementations.

4.3.3.2 Activity Diagram

The activity diagram could be a variation of those in sections 4.3.1.2 and 4.3.2.2, depending on the extent and purpose of the reprogramming.

4.3.3.3 Software Development Life Cycle

Development and testing follow the standard software process found in most of the standard software projects. As such, Agile (Scrum, Kanban, Waterfall, etc) are all compatible development methodologies for Actyx-enabled projects.

However, due to the nature of the implementations that require Actyx (applications that span across the software and physical world, extra care must be taken for requirements analysis and design prior to implementation. Figure 32 represents the software development process funnel followed in this recipe.

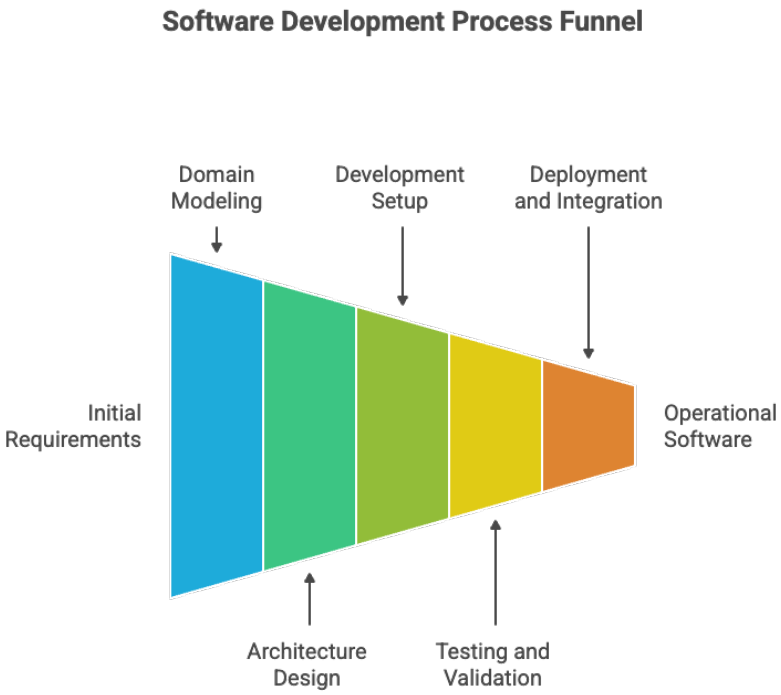

Figure 32: Software development process funnel.

- **Initial Requirements**: Capture the business goals, factory constraints, and expected outcomes. Define KPIs (throughput, downtime reduction, flexibility) and identify the stakeholders (operators, engineers, managers). This phase sets the scope and success criteria
- **Domain Modelling**: Translate requirements into an event-first model: identify key assets (machines, conveyors, AGVs, etc.), their states, and the events that describe their behaviour. Define event schemas and state transitions, which will become the foundation for Actyx machines.
- **Architecture Design**: Decide how assets are partitioned into Actyx asset runners. Define communication flows, connectors to external systems (PLCs, ERP, MES), and

deployment topology (nodes, networks). Select supporting libraries (machine-runner, machine-check, UIs).

- **Development & Setup**: Establish the coding environment (TypeScript/NodeJS), repo structure, CI/CD pipelines, and containerization strategy (Docker-compose or Helm). Prepare stubs and simulators for connectors so development can proceed without immediate access to real machines.
- **Testing and Validation** (see dedicated section on verification below)
- **Deployment and Integration**: Package runners, and Actyx nodes into deployable containers. Deploy first in staging (shadow mode), then to production with controlled rollout. Integrate with customer infrastructure, train operators, and confirm with site acceptance tests

4.3.3.4 Architecture and Tools

An Actyx solution (Figure 33) is based on using machine-runner and machine-check libraries to abstract away the operations dedicated to message construction and state management.

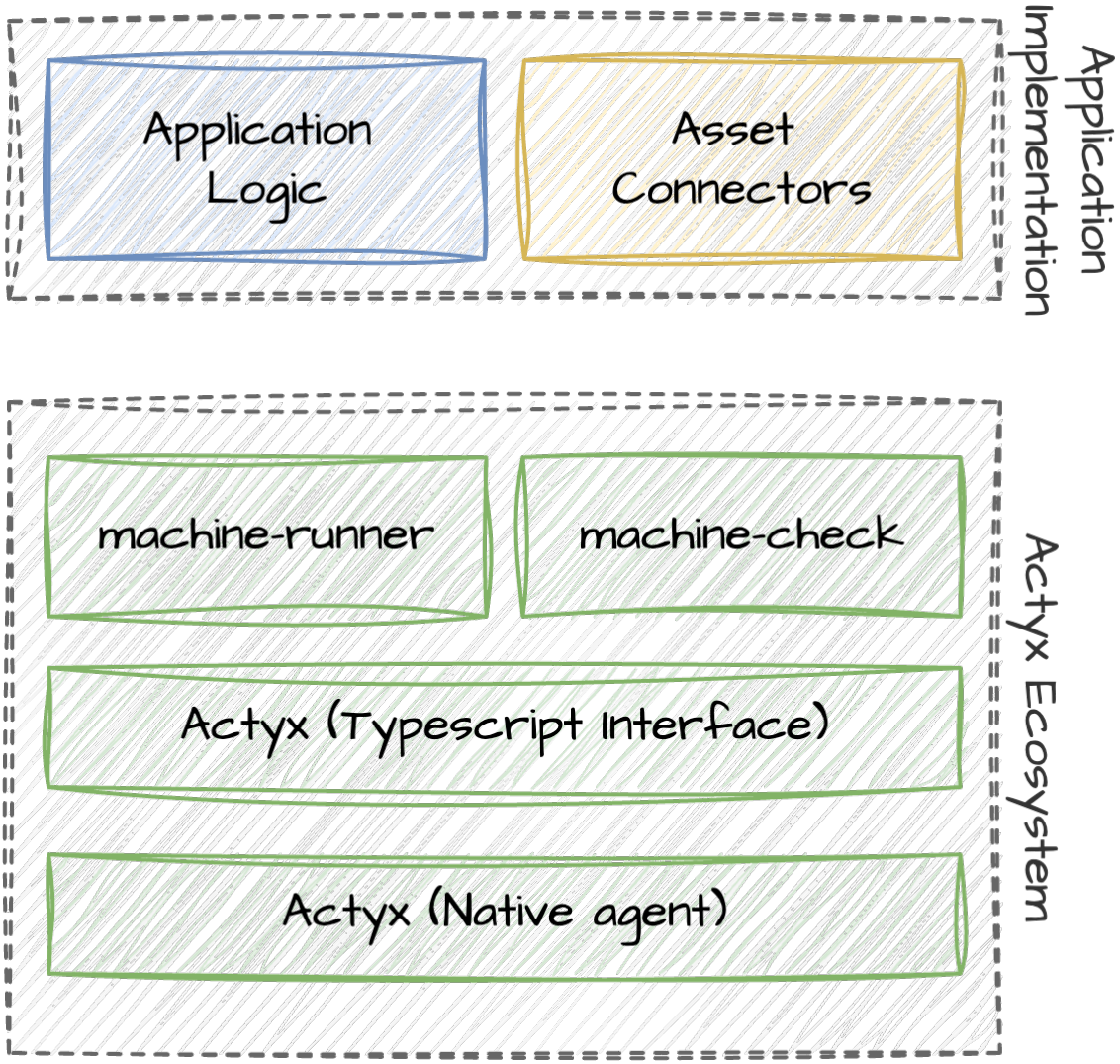

Figure 33: Actyx architecture overview.

Actyx and its associated components are employed to deliver a comprehensive solution:

- **Actyx** (native) is the middleware that implements message distribution across the system.
- The **Actyx TypeScript** library provides APIs used to implement programs (a.k.a. *machines*) that send and receive messages over the Actyx (native) middleware.
- The **Actyx machine-runner** TypeScript library provides mechanisms for defining states and coordinating actions among system components.
- **Actyx machine-check** assists in verifying implementations. It is an invaluable tool that can check protocol conformance before deployment, without having to resort to installing the application and making integration tests.

The following components will be developed and customized for each application implementation:

- **Asset Connectors**, implemented in TypeScript, function as intermediaries between the state managed by the machine-runner and the actions required of each asset. An asset may include a factory machine, an individual receiving input, or any entity that is represented within the machine-runner framework.
- **Application Logic**: The core application logic is designed with reference to key requirements and leverages the libraries available in the Actyx SDK. Protocols will be constructed within this logic to accurately represent entities relevant to the application domain.

Each TaRDIS Solution using Actyx relies on the Actyx being the central broker of the messages between subsystems. In the following simplified example (Figure 34), each machine protocol is implemented on top of the machine-runner interfaces provided by Actyx. They use a connector to give messages to the actual real-life machines.

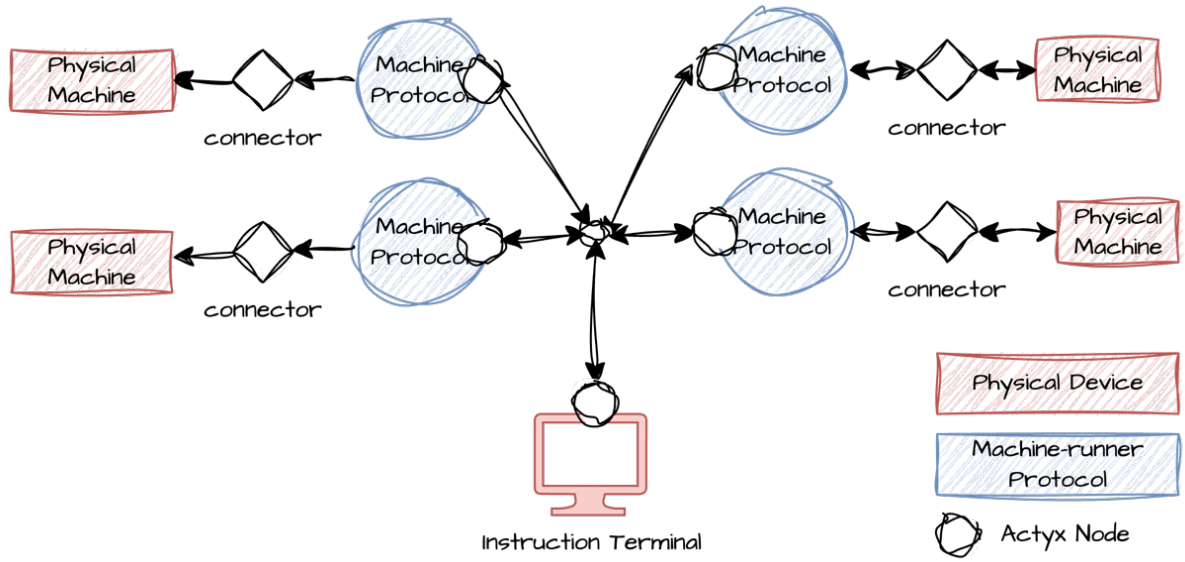

Figure 34: Components in the "Reprogramming critical elements of a manufacturing process" recipe architecture.

The machine-runner serves as the central component of the Asset Runner, overseeing both state management and process execution. At runtime, it executes configurations provided via external files.

4.3.3.5 Verification

The validation of this use case recipe is based on a combination of testing and monitoring (with metrics collection) described in the subsections below

Testing

Testing is conducted in multiple layers, progressing incrementally from code-level validation to real-world integration:

- **Unit Testing** – This phase checks that coding standards and styles are followed. Unit tests utilizing the machine-check library are developed to assess if protocols function as intended under cause-and-effect scenarios.
- **Integration Testing** – Larger subsystems are tested collectively, often using mocks to simulate external dependencies. Mocks are employed to replicate the behaviour of the main application as if external systems were present.
- **Manual Testing with Docker** – Applications are bundled and run in containers for manual testing, moving one step closer to production environments. This step tests connectors, using two main approaches:
    - Connector Testing – Two methods include:
        - Using connectors that return immediately (via Actyx) without executing actual actions.
        - Using external triggers to simulate Actyx behaviour.

On-premises Customer Testing – The final stage involves system integration and testing at the customer site, which includes:

- Testing the end-to-end production process.
- Validating connectors against real machines or PLC programs simulating factory operations.

This approach uses iterative development and multi-level testing to verify that the solution is controllable, reusable, testable, and maintainable prior to deployment in production environments.

Monitoring and metrics collection

Metrics Collection from the main application is based on Prometheus metrics, a universal standard format for aggregating statistics from exposed containers and applications (see Figure 35).

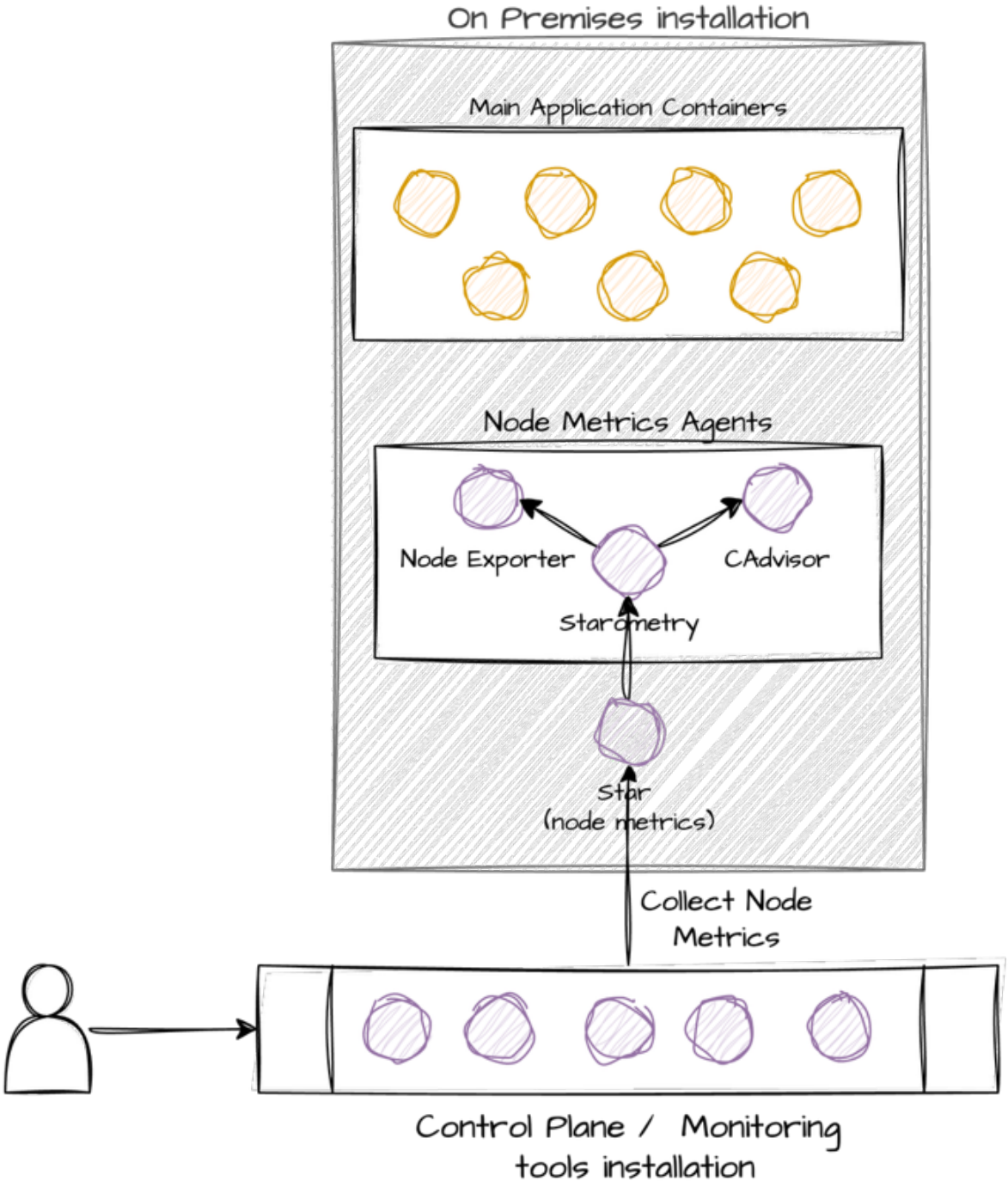

Figure 35: Monitoring and metrics collection.

The metrics mechanism is a component of the TaRDIS Toolbox and comprises the following primary tool categories:

- **On-Premises Installation Tools (Node Tools)**:
  - **Node Exporter**: Collects system-wide metrics from the node.

- **cAdvisor**: Gathers metrics from active containers, including the main Actyx application containers and asset runners.
- **Starometry**: Aggregates collected log data.

- **Control Plane**: This category encompasses tools responsible for aggregating, analysing, and presenting the collected information. Users access these tools to view comprehensive metrics and relevant operational data.
- **Prometheus**: Provides a robust system for querying metrics.
- **Grafana**: Facilitates visualization of the collected data.
- **Databases and Auxiliary Tools**: Support data storage and ancillary operations.

4.3.3.6 Machine Learning

Machine learning is not used in this recipe.

4.3.4 Distributed Orbit Determination and Time Synchronization of a constellation

4.3.4.1 Scenario and Requirements

The first scenario proposed by GMV for the use case of Distributed (Decentralized) **O**rbit **D**etermination and **T**ime **S**ynchronization (ODTS) represents the simulation of the nominal operation of a satellite constellation, in which ODTS tasks are performed in a distributed manner. This approach prioritizes onboard and autonomous determination over the commonly used centralized ground-based process, which heavily depends on links with ground stations. This scenario constitutes the core of what is intended with this use case, while the second scenario -described later- represents an enhancement that increases the swarm's autonomy capabilities and improves its resilience against potential communication failures.

This scenario leverages the use of the TaRDIS toolkit to carry out simulations and investigate the feasibility, parameter tuning and performance of a real-time, distributed ODTS process. In this way, the constellation is simulated (specifically, in GMV's use case, a LEO (Low Earth Orbit) mega-constellation, due to the growing relevance of such configuration in cutting-edge real-world projects) as a swarm, where the nodes are the individual satellites, also including the network of ground stations that monitor them. The diagram in Figure 36 outlines the processes followed by the actors involved in the scenario. It is worth noting that, in this scenario, it is assumed that each node is aware of the peers it must connect at any given time (i.e., this connection schedule has been provided via telecommand from ground control, and all of the links are viable and expected to occur as planned).

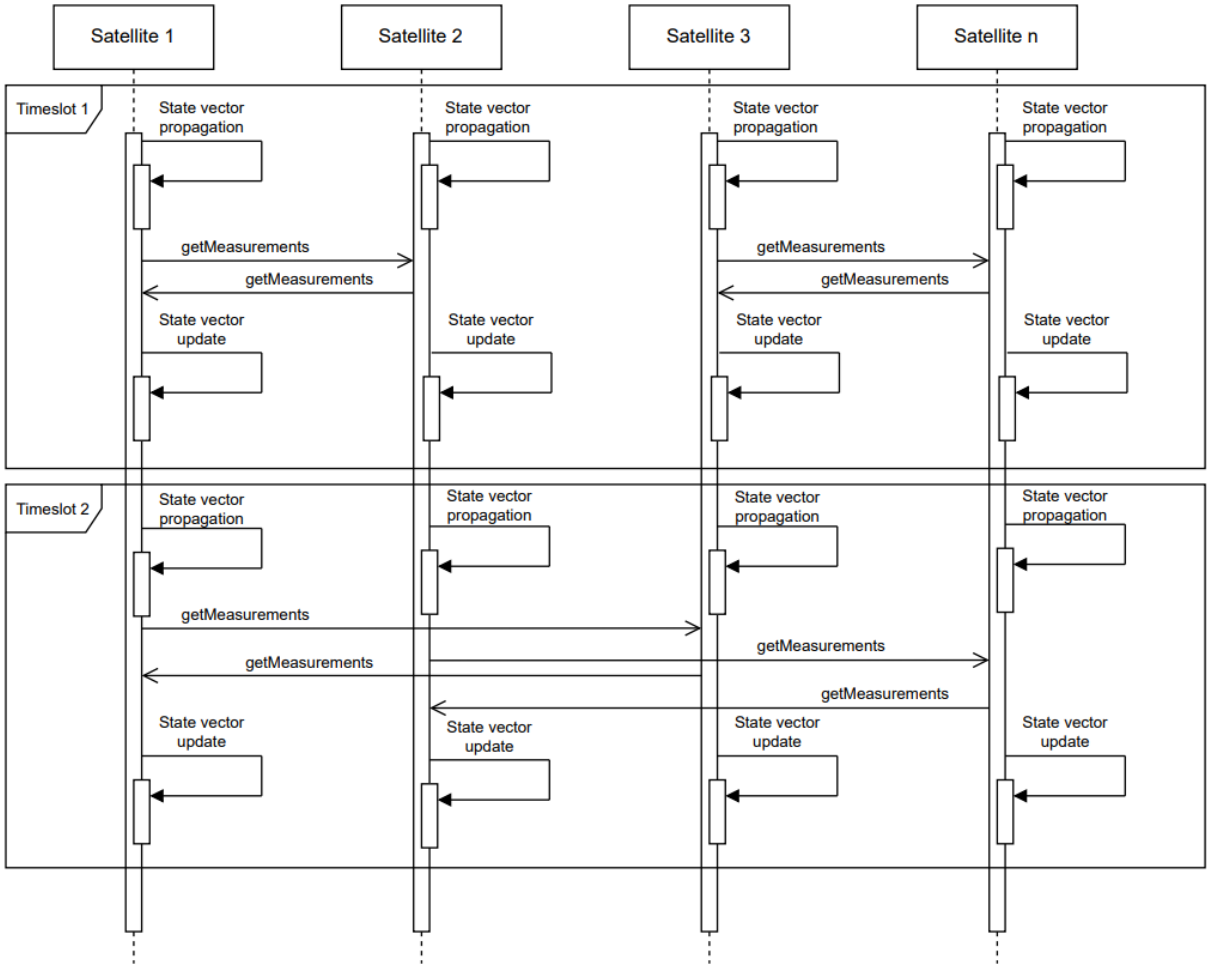

Figure 36: Orbit Determination and Time Synchronization.

Starting from predefined initial conditions and also providing connection schedules between the different nodes as input, each node must be capable of performing the following actions in each timeslot.

1. Propagate its state vector (position, velocity and clock information) through time from the last timeslot to the current timeslot: predict the evolution of its state based on the updated estimates and the dynamic models used on-board.

2. Establish connections with scheduled peers: initiate communication links with the designated nodes according to the predefined schedule.

3. Exchange state information and time data: share its current orbit and timing estimates with connected peers and receive the corresponding data from them.

4. Fuse received data with local estimates: Apply a data fusion algorithm such as Kalman Filter to integrate the information from peers with its own on-board estimations

5. Update its state vector: refine its orbit and clock estimates based on the fusion process, ensuring improved accuracy with each iteration.

With the help of the TaRDIS toolkit, two main objectives are pursued.

- The application of Machine Learning techniques to the ODTS process, aiming to investigate and test the impact of new methodologies on the performance of orbit determination and time synchronization compared to commonly used techniques, such as the Extended Kalman Filter. The goal is to assess whether Machine Learning can offer improvements in accuracy, robustness or computational efficiency, particularly in scenarios involving large constellations and limited inter-satellite communication opportunities.

- Perform a fully decentralized simulation that enables testing the level of autonomy achieved with the implemented distributed ODTS process, in contrast to the typically used centralized schemes. The use of PTB-FLA allows each satellite in the constellation to be simulated as an independent application instance, with the exchange of information and measurements between them emulated through the Babel framework. This setup provides a realistic environment to evaluate the behaviour of the system under distributed conditions, assessing aspects such as scalability, fault tolerance and the ability of the constellation to maintain accurate state estimation without heavily relying on centralized ground-based control.

4.3.4.2 Activity Diagram

GMV Use Case Development Methodology

The GMV UC development methodology is used to govern the GMV UC software development process, which takes the GMV baseline together with the scenarios and requirements as its input and produces the GMV UC software at its output. To ensure that the final code is correct by construction, the process is a sequence of transformational phases. Within some of the phases there are some parallel and iterative sub activities, which are abstracted away for the sake of simplicity.

The GMV UC development methodology is composed of machine learning methods (e.g., a differentiable program for end-to-end training using techniques like stochastic gradient descent [27], knowledge distillation, model pruning, etc.), Python TestBed for Federated Learning Algorithms (PTB-FLA) development paradigm [28], and Babel's generalization named the Generic Framework for Building Dynamic Decentralized Systems (GFDS) [29].

The GMV UC development methodology is supported by the TaRDIS toolkit, and it takes the top-down development approach (Fig 1.) from (1) the sequential decentralized code, (2) to the decentralized multiprocessing PTB-FLA code running on a local host, and (3) to the decentralized and distributed PTB-FLA/Babel code running on a computer network. The development is conducted through the following four phases, whose names are given according to the code they produce: (1) the referent code, (2) the PTB-FLA code, (3) the PTB-FLA/Babel code, and (4) the final code.

The development methodology recognizes the three distinct roles: a developer (in charge of development & operations a.k.a. devops, including testing), a machine learning (ML) developer, and a formal verification (FV) engineer. Developers are engaged in all the phases, ML developers are engaged in Phase 1, and FV engineers are engaged in Phase 4.

The development methodology is traditionally illustrated by an activity diagram wherein roles are shown in different swim lanes (Figure 37). However, when there are many artifacts, activity diagrams with swim lanes might become rather complex. Therefore, in this document an alternative approach without the swim lanes is used. The first activity diagram shows the whole process, i.e., the activities for the first role (developers), whereas the next two activity diagrams show the activities for the second role (ML developers) and the third role (FV engineers), respectively.

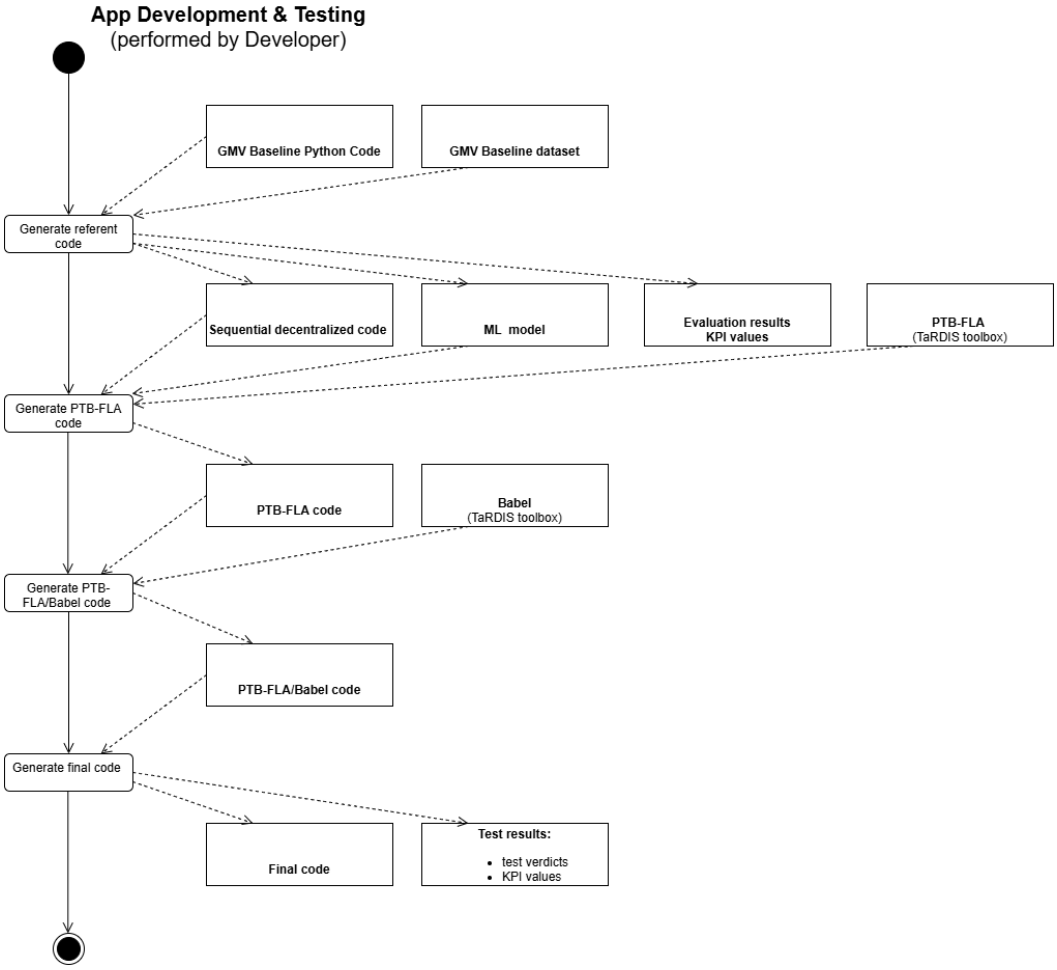

Figure 37: App development and testing activity diagram.

Phase 1 (Referent code)

This phase is the first phase of the PTB-FLA paradigm. The input to this phase is: (1) the GMV baseline Python code and (2) the dataset produced by baseline simulations, see inputs to the activity “Phase 1” in Figure 37. In this phase, the mentioned dataset and the adequate ML methods are used to train the orbit determination and time synchronization (ODTS) ML models. The dataset must be composed of orbital simulation data, the measurements that the satellites exchange, and the output of the baseline GMV algorithm.

The output of Phase 1 is the *sequential decentralized* code (or the referent code), see Figure 9, which performs a *sequential* loop over a satellite constellation and within the loop performs a *decentralized* processing of satellite local data for each satellite.

ODTS Model development process is further described in Figure 38, and its outputs are: the trained model, the evaluation results, and the KPI values. As shown in Figure 38, if the evaluation results and the KPI values are not satisfactory, then the process starts from the beginning.

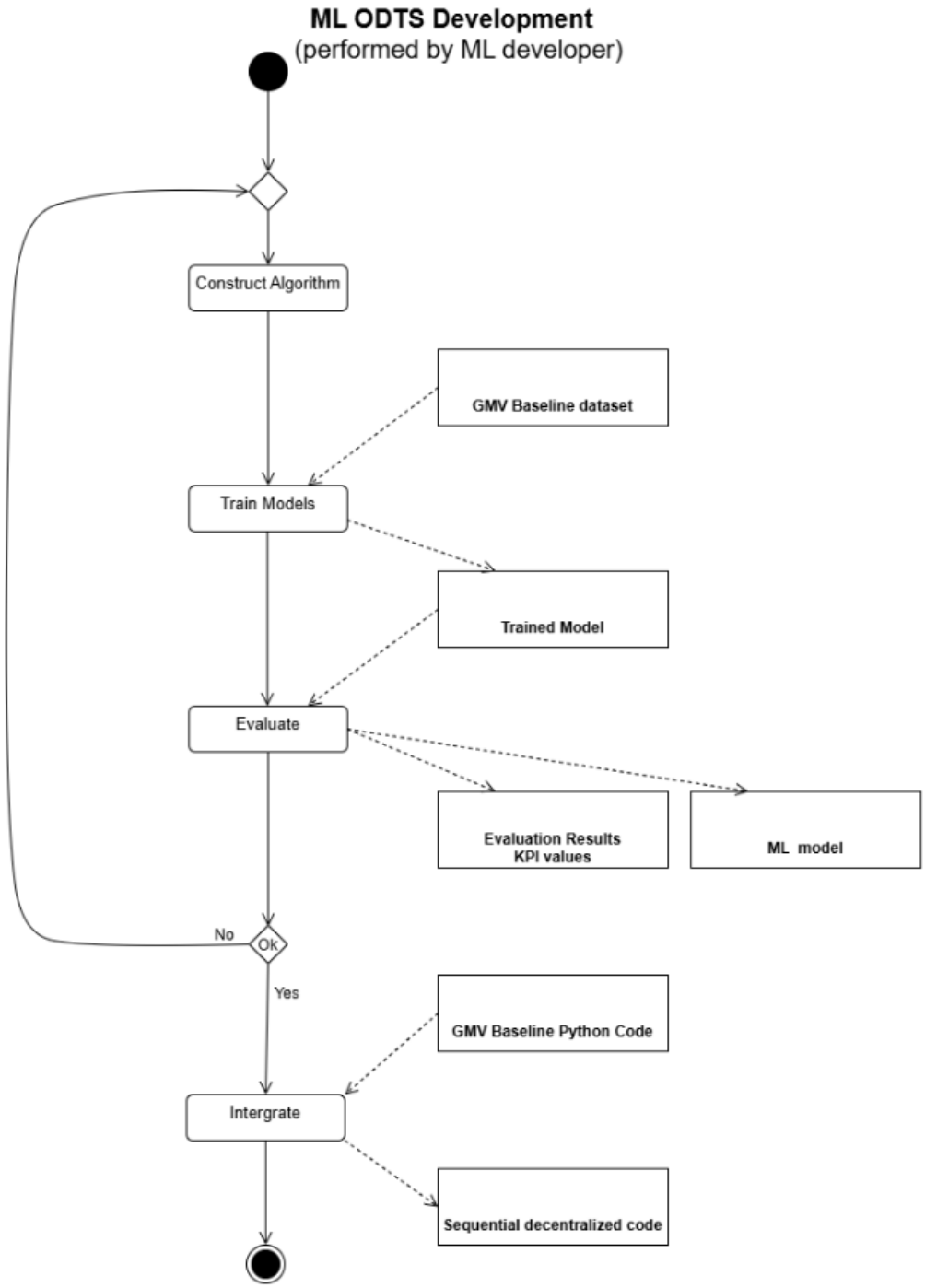

Figure 38: ODTS ML methods development activity diagram.

Phase 2 (PTB-FLA code)

This is the second and final phase of the PTB-FLA paradigm for applications utilizing generic Time Division Multiplexing (TDM) algorithms (get1Meas and getMeas). The inputs to this

phase are the sequential decentralized code, the trained ML model, and the PTB-FLA tool, see Figure 37. In this phase, the PTB-FLA is integrated into the sequential decentralized code by adding calls to PTB-FLA's constructor, generic TDM algorithms, and destructor. The output of Phase 2 is the decentralized multiprocessing PTB-FLA code, see Figure 37, which in accordance with the single program multiple data (SPMD) pattern may be instantiated and launched by the PTB-FLA launcher as a swarm (or group) of processes running on a single computer i.e., on a local host. Each of these processes plays the role of a single satellite or a ground station.

Phase 3 (PTB-FLA/Babel code)

This phase is based on Babel's GFDS approach. The input to this phase is the PTB-FLA code, see Figure 37. In this phase, PTB-FLA code is integrated with the PTB-FLA Adapter for Babel and the required Babel services: the message passing broadcast service, the satellite membership service, and the distributed storage service Nimbus. The output of this phase is the decentralized and distributed PTB-FLA/Babel code, which requires the system startup that is conducted on all the target computer network nodes in the following two stages. In stage one, the required Babel services and the local PTB-FLA Adapter instance are started by the local instance of the main Babel application, and in stage two, PTB-FLA instances are launched by the respective local PTB-FLA launchers.

Phase 4 (Final code)

This phase is based on the GMV use case recipes. The input to this phase is the PTB-FLA/Babel code, see Figure 37. In this phase, the input code is fully verified and evaluated in accordance with the GMV UC recipes i.e., the corresponding test cases are executed and the defined KPIs are measured.

PTB-FLA has been formally verified using CSP before becoming a part of the TaRDIS toolbox, as illustrated in Figure 39, following the procedure explained in the GMV use case recipes.

The output of this phase is the final GMV UC code and the test and evaluation report.

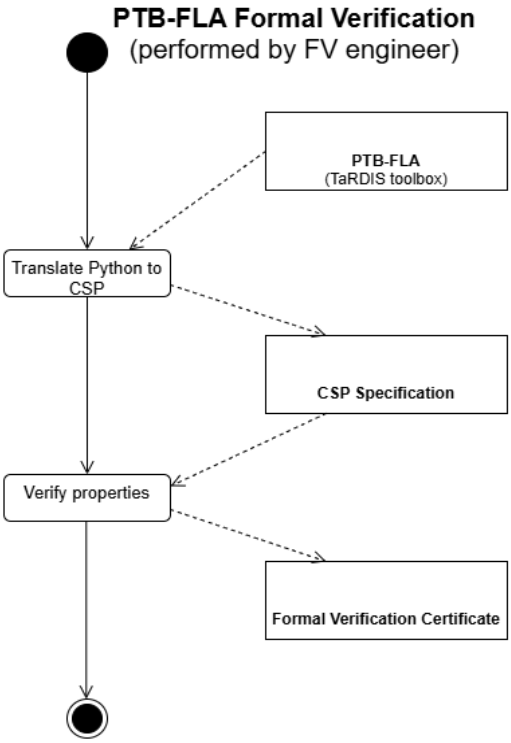

Figure 39: PTB-FLA Formal Verification activity diagram.

4.3.4.3 Software Development Life Cycle

In this section we map the software development methodology used in the GMV use cases to the waterfall model life cycle (Figure 40).

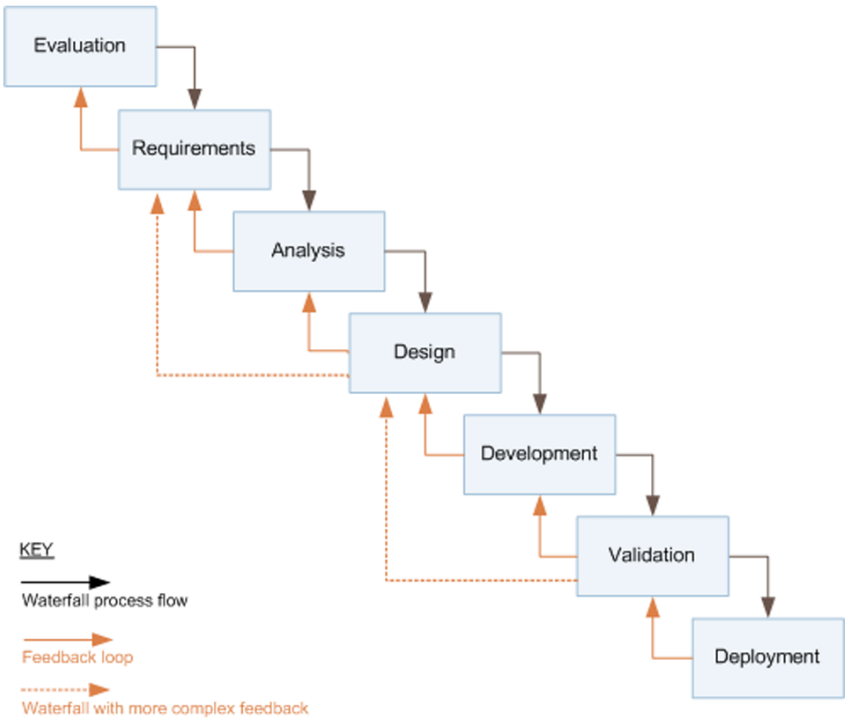

Figure 40: Software development life cycle at GMV.

Evaluation: this process was made ad-hoc while making the application for the TaRDIS project.

Requirements: defined during WP2, but with the final product in mind (ODTS algorithms + ML models). The final code (output of the Phase 4) developed with the help of TaRDIS must be capable of supporting the design, execution, evaluation and testing of such a product.

Analysis: This phase was again centred in the final product, based on state-of-the-art studies in decentralized ODTS algorithms. This process helped the design of the GMV baseline python code, performed in the next phase.

Design: In this phase, the GMV baseline python code is generated, which is the input for the first phase of the development approach.

Development: contained in the development methodology/activity diagrams

Validation: part of this is included in the development methodology/activity diagrams. Further tests/evaluation must be done after the activities shown in the diagrams (such as an evaluation of the final code inside a satellite board).

Deployment: once the final code is fully developed (with the help of the TaRDIS toolkit) and tested (both in the simulated environment and outside it), the final product can be deployed in a real environment (satellite/representative board)

4.3.4.4 Architecture and Tools

This section provides a broad overview regarding the different tools used in this scenario and their interactions. An architecture diagram is also provided with the complete scenario execution flow.

Tools

(From WP5)

GMV UC PTB-FLA Application:

The PTB-FLA application serves as a focal point of the system, dictating the communication tempo, rescheduling the inter satellite links (ISLs), calling the orbit determination time synchronization (ODTS) algorithms, and simulating node crash failures. ISL communication between satellite nodes (i.e., PTB-FLA application instances) is conducted either using the PTB-FLA function get1Meas or the PTB-FLA function getMeas. During communication, satellite nodes exchange data over the computer network using the PTB-FLA Babel adapter and protocols written in the Babel framework, propagating the data over the network.

PTB-FLA:

PTB-FLA offers two peer-to-peer data exchange algorithms: get1Meas, for direct communication between pairs of nodes, and getMeas for exchanging data with an arbitrary number of nodes. Another benefit of PTB-FLA framework is its standardized development process from the sequential machine learning code to the federated code with the use of the PTB-FLA federated learning development paradigm. With the use of PTB-FLA Babel adapter

it is possible to run a distributed federated learning application over a network without any modifications to the application code.

Nimbus:

Nimbus is designed as a fully decentralized storage system, by providing a scalable and efficient data storage without relying on a central authority. Nimbus is used to store and share data among the different nodes of the constellation, namely, the satellite state vector, ephemerides, etc. It will use the satellite membership service to communicate with the other nodes (i.e., it may not be possible to guarantee that every satellite receives the same data in a specific time due to eventual nature of the data store, or possible failures).

Synchronization Protocols:

The synchronization protocols ensure that data is correctly propagated through the different nodes. Nimbus operates on top of these protocols to send and receive information, as well as update its communication pool (i.e., its incoming and outgoing links) through neighbourhood updates arriving from the membership protocol.

Satellite Visibility Model:

The satellite visibility model is in charge of calculating, or receiving from PTB-FLA, the satellites reachable from the current node in different time slots. The model outputs a set of available satellites for the ongoing timeslot (i.e., as dictated by PTB-FLA) and shares this information with the membership protocol to allow communication.

Membership Protocol:

A tailor-made membership service for the use case receives information from the visibility model (i.e., the available neighbour satellites for the current timeslot) and updates the membership accordingly. Moreover, since satellite communication paths may be obstructed due to external events (i.e., antenna alignment, passing debris, etc.) in the communication phase, the protocol is able to, in real-time, to update its membership pool to adapt to such situations.

Architecture

The tools described above interact with each other to enable inter-satellite communication and distributed orbit determination and time synchronization of the constellation.

The overall architecture is depicted below in Figure 41, while a detailed explanation regarding the interactions with WP5 and WP6 tools are provided in Figure 42.

It is important to note that the arrows from PTB-FLA to “Satellite visibility model” and from the latter to the “Membership Protocol” represent blocking procedure calls (i.e., sending a message and waiting for an acknowledgement). This ensures that the next time slot is only started when both parties agree to the correct satellite neighbourhood to the current slot.

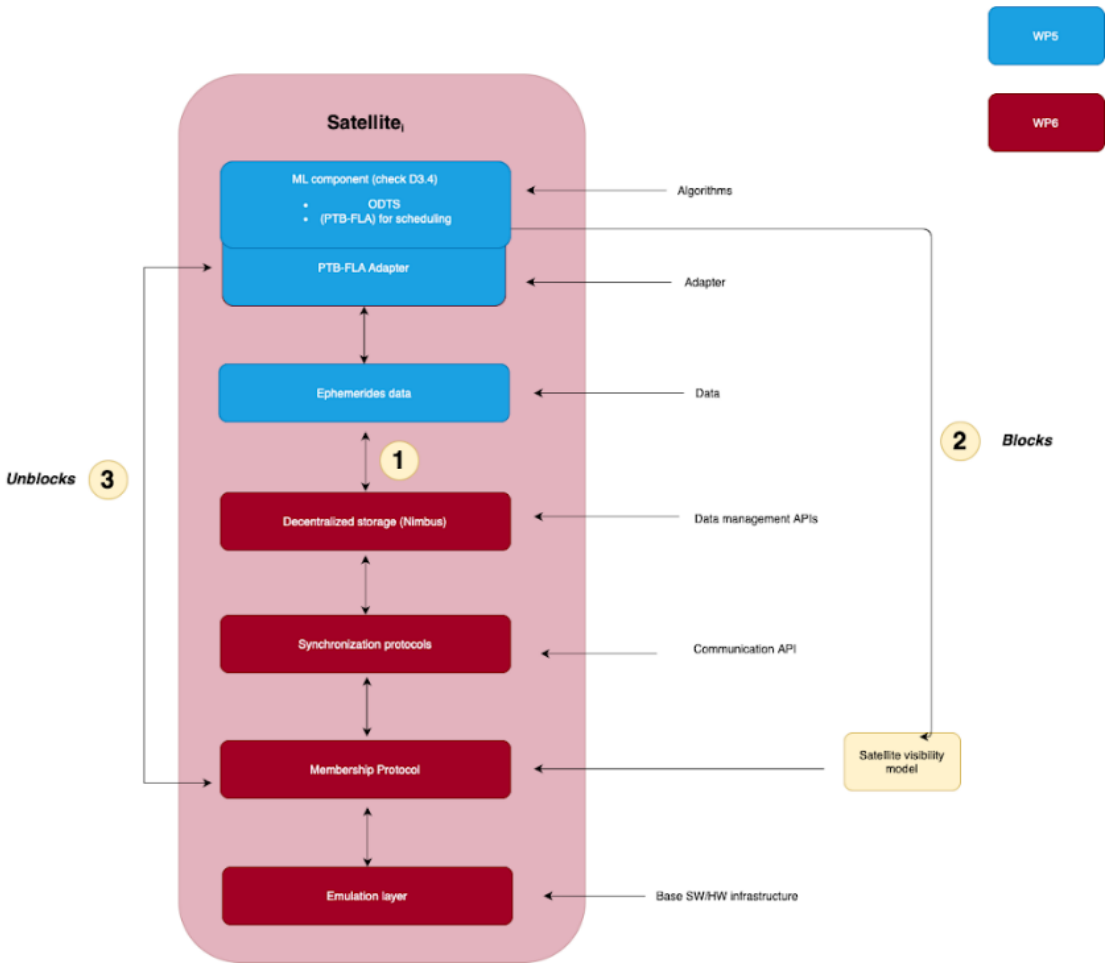

Figure 41: Scenario 1 architecture.

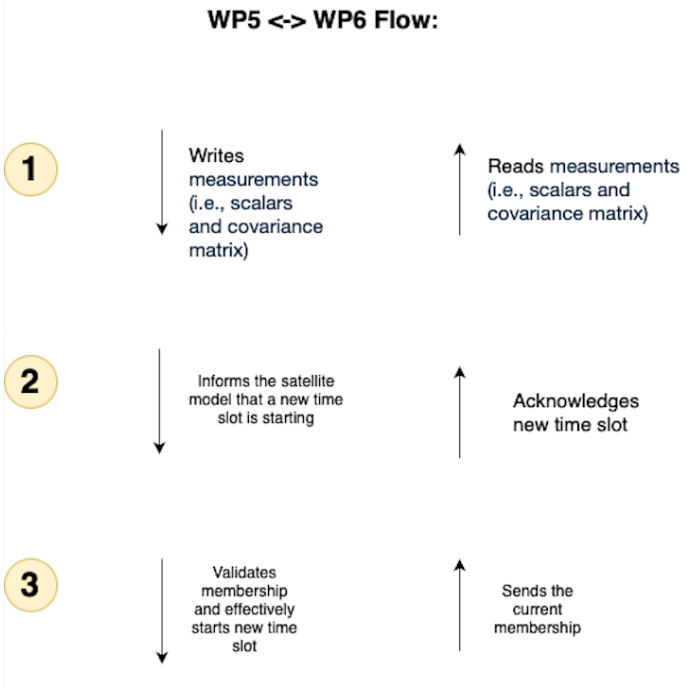

Figure 42: WP5 and WP6 workflow.

Execution Flow

An overview of the execution flow demonstrated in the scenario, offering a step-by-step outline of how the process unfolds, is as follows:

1. PTB-FLA advances each timeslot.
2. PTB-FLA sends the next timeslot to the “satellite visibility model” and waits for the acknowledgement (to be sure that all the Babel components are ready to advance for the next timeslot).
3. The “satellite visibility model” advances the timeslot, accordingly, sends the information to the “membership service” to update the neighbourhood for the current slot, and waits for the acknowledgement.
4. The “membership service” sends the acknowledgement to the “satellite visibility model”.
5. The “satellite visibility model” sends the acknowledgement to the PTB-FLA so communication with Babel components for this slot can start.
6. PTB-FLA requests the current membership for the satellite and checks if the schedule for the current time slot number is valid, and if not it starts the dynamic (re)scheduling and eventually communicates the changes to the “membership service” (note: dynamic scheduling will not be completed in the current time-slot, this is why the term “eventually” is used).

7. PTB-FLA instances exchange their data through Nimbus.
8. Repeat steps 1-7.

4.3.4.5 Verification

The term “verification” is intended in a broad sense here, including:

- Static verification, i.e., analyse the system before running it, to ensure that (some categories of) bugs will never occur at runtime.
- Runtime verification, which may include:
    - Monitoring, i.e., have monitors that run alongside the system and report errors and unwanted events as soon as they occur in the running system.
    - Testing, e.g., via a systematic test suite, fuzzing.

WP5

PTB-FLA Static Formal Verification:

PTB-FLA offers four generic algorithms, namely, the centralized FL, the decentralized FL, the time division multiplexing (TDM) data exchange between pairs of nodes, and the TDM data exchange with an arbitrary number of peer nodes. At present, the first two generic algorithms have been formally verified, and the verification of the latter two algorithms, based on the same approach (presented below), is currently in progress.

The generic centralized and decentralized FL algorithms are formally verified using the Communicating Sequential Processes calculus (CSP) process algebra and the Process Analysis Toolkit (PAT) model checker, in a two-phase process.

In phase 1, the fact that PTB-FLA Python code follows a restricted actor-based programming model is leveraged to systematically construct its formal description using CSP. The description is expressed in CSP# language used by PAT, where CSP processes represent instances of the generic algorithm. This systematic approach ensures that the CSP models accurately correspond to the actual Python code.

In phase 2, two properties of generic algorithms are proven: the deadlock freedom (a safety property) and the successful termination (a reachability and a liveness property). Callback functions that contain the actual ML&AI application processing are not modelled. The only assumption about them that is necessary for the correctness verification is that they terminate (i.e., have the termination property), which is expected from all correct processing in any case.

C Code Static Verification:

In the case of the code of the models that is migrated to the C language for a potential deployment on a representative on-board system, it must undergo a static verification process, for instance by utilizing the Cppcheck tool. Such tests would also ensure the compliance with some of the safety properties required for on-board software, and the requirements RF-GMV-22 and RF-GMV-23.

Dynamic (Runtime) / Monitoring

For runtime verification, monitoring tasks are integrated through the insertion of assertions within the codebase. Assertions are diagnostic constructs that evaluate specific conditions at runtime and trigger predefined error handling mechanisms when violations occur. This approach enables early detection of anomalous behaviour and supports the validation of system properties during operational execution.

Dynamic (Runtime) / Testing

Test cases (i.e. run the scenario, offline analysis of produced files).

This recipe is strictly related to the tests designed for Scenario 01 of GMV's use case. Specifically, the application of this recipe, along with the analysis and/or comparison of the resulting ODTS metrics and performance, forms the backbone of test cases T-GMV-101, T-GMV-102 and T-GMV-103. These test cases are used to verify compliance with requirements RF-GMV-01, RF-GMV-02, RF-GMV-03, RF-GMV-04, RF-GMV-05, RF-GMV-10, RF-GMV-11, RF-GMV-12 and RF-GMV-17, as well as with the KPI K-U-05.

All these requirements are related to the decentralization of the ODTS process in a safe way (i.e. ensuring that the solution obtained is converged, the correctness of the satellite communications under nominal operation and testing that in the occasion of a communication failure (in absence of measurements) a satellite can still propagate its state vector).

As said above, in all these cases the runtime verification is performed by carefully analysing the output data.

WP6

Static Verification

Not applicable.

Dynamic (Runtime) / Testing

Not applicable.

Dynamic (Runtime) / Monitoring

Babel supports a metrics collection system embedded into its Core. This module collects node-related metrics, namely both hardware and kernel related metrics (e.g., CPU usage time, memory usage, etc.). Moreover, it's also possible to collect custom metrics. For example, the synchronization protocols may decide to store custom metrics such as latency or throughput.

This approach enables one to monitor the current status of the process in real-time, or store such information later on to be analysed offline after the scenario execution.

4.3.4.6 Machine learning

PTB-FLA:

Python TestBed for Federated Learning Algorithms (PTB-FLA) is a lightweight Federated Learning (FL) framework that follows Single Program Multiple Data (SPMD) pattern and

provides simple development experience to ML and AI developers who are not as experienced with distributed systems development. It is based on pure Python and leverages multiprocessing primitives to communicate between the nodes. Overall, it offers four generic algorithms, which are implemented as the PTB-FLA API functions: fl_centralized, fl_decentralized, get1Meas, and getMeas, respectively.

ODTS ML Methods & Models:

The proposed Machine Learning algorithm leverages historical position data and environmental variables. The use of time-series techniques and exogenous variables enables the model to achieve low positioning errors at a reduced computational cost, providing a faster and reliable orbit determination for an increasing number of space objects.

Physics-Informed Neural Networks (PINNs) and Neural Ordinary Differential Equations (NeuralODEs) are also used to enhance the model's accuracy and generalizability. The first technique guarantees that the model's predictions are consistent with known physical principles, such as the effects of gravitational forces and atmospheric drag, reducing the need for larger datasets to cover unseen conditions. NeuralODEs embeds differential equations directly into the architecture of the ML model, which allows the propagation process to be modelled to be constrained by orbital mechanics during both training and inference.

4.3.5 ISL re-scheduling capabilities - Scenario 2 recipe

4.3.5.1 Scenario and Requirements

This use case recipe is related to the second main scenario considered for GMV's use case. This scenario has undergone several modifications throughout the project, reflecting the evolving interests regarding the applicability of this functionality – modifications both in its conceptual approach and in the tools initially intended for its implementation.

The situation addressed in this recipe concerns a partial or total communication failure of a node. Although the affected node must still be capable of propagating its state vector to provide the best possible navigation solution using its available resources, a contingency measure is proposed: executing a re-scheduling process to minimize the impact of such failure on the rest of the swarm. This approach aims to preserve the overall system performance and maintain the robustness of the ODTS process in degraded operational conditions.

As indicated in the nominal operation, simulations start with a preloaded communication schedule on each node, which is generated using a scheduling algorithm developed by GMV. However, in the event that a node loses the ability to establish connections through one of its antennas (partial failure) or both (total failure), the impact extends beyond its own state estimation. Such a failure also prevents all other satellites scheduled to link with the affected node from obtaining measurements, thus disrupting their estimation processes as well. In this situation, the constellation must autonomously trigger an on-board re-scheduling process, which generates a new orchestration of connections that accounts for the limitations introduced by the failed satellite – specifically, by excluding it from the pool of feasible links. This adaptive mechanism is essential to preserve the integrity and performance of the ODTS process of the healthy satellites.

In this scenario, the tools from the TaRDIS toolkit used are the same as those in the previous one, and no additional Machine Learning components are introduced beyond those previously described. The key difference lies in the tasks performed at the application layer: while the node continues to follow its predefined connection scheme and computes its navigation solution, it must simultaneously work on generating an updated communication schedule. This new schedule must be based on the ephemeris information of the constellation, taking into account the uploaded network topology resulting from the failure.

Thus, the tasks that the application must perform in response to a failure event are as follows:

1. **Failure diagnosis.** In the messages exchanged between satellites, in addition to the data required for the ODTS process and the updated ephemerides of the constellation, a failure dictionary is included. This dictionary is progressively filled with failure information reported by the satellites. If connection issues with a satellite are detected over several consecutive timeslots, that satellite is considered to have failed – either partially (one antenna) or totally (both antennas), depending on the number of reports per slot. When this threshold is reached, the re-scheduling process is triggered.

2. **Snapshot of current ephemerides and re-scheduling execution.** A snapshot of the most up-to-date constellation ephemerides is taken, and the re-scheduling algorithm is executed. This algorithm is a simplified version of the baseline scheduling algorithm, adapted to on-board constraints such as limited computational power and memory. While this on-board generated schedule may not be as optimal as the one generated on the ground, it is sufficient to prevent satellites from attempting to link with failed peers. This avoids the occurrence of timeslots without measurements, which would significantly degrade orbit determination accuracy.

3. **Synchronized application of the new schedule.** Once the re-scheduling process is completed, the newly generated schedule must be applied in a coherent and consistent manner across the entire constellation. That is, all satellites must operate with an identical communication schedule and apply it in a synchronized fashion to ensure the integrity of the network and continuity of the distributed ODTS process.

A final clarifying note must be made: the scenario will ultimately not include the application of Reinforcement Learning agents, as initially proposed. Although certain requirements related to the use case were defined with the expectation of leveraging this methodology – and it was indeed a path that was originally intended to be explored – it has since been concluded that RL is not suitable for the objectives pursued in this context. This decision is based on an evaluation on its applicability, considering the operational constraints and the nature of the problem, which demand deterministic and lightweight on-board decision-making processes.

4.3.5.2 Activity Diagram

See section **Error! Reference source not found.**.

4.3.5.3 Software Development Life Cycle

See section **Error! Reference source not found.**.

4.3.5.4 Architecture and Tools

This scenario uses the same tools and architecture described in section 4.3.4.4.

4.3.5.5 Verification

Since this recipe is related to an exceptional failure scenario that falls outside the nominal operation of the simulation, it should be understood that all static and dynamic monitoring verification methods detailed in the previous recipe – concerning the tools from both WP5 and WP6 – also apply in this case. However, regarding dynamic verification tied to testing tasks, this recipe introduces new opportunities that will support the assessment of compliance with the requirements linked to Scenario 02 of GMV's use case. Therefore, test cases T-GMV-201, T-GMV-202, T-GMV-203, T-GMV-204 and T-GMV-205 are directly associated with this recipe, and once again, the verification process relies on executing the simulation under the predefined conditions detailed above and analysing the output data obtained.

4.3.5.6 Machine Learning

The Machine Learning techniques applied for this scenario are the same as described in section 4.3.4.6.

4.3.6 IDE – How to deploy a project with the different tools available

4.3.6.1 Scenario and Requirements

This tool is present within every single test use case scenario established, so this recipe and its description apply to every test scenario.

The primary objective behind the development of TaRDIS was to simplify how the developers interact with distributed systems by automating intricate configurations and providing user-friendly workflows. Building from that starting point, the consortium developed extensions for existing best-of-breed IDEs to be versatile and adapt its features to suit their specific project requirements. The initial extension was built for the Eclipse IDE, but after a survey that was posed to the developer community, the path switched into the customisation of the Visual Studio Code (VS Code) IDE, currently the most popular especially in the new generations of developers. The objective then is to make an extension to this IDE in order to customise it for the development of swarm environments using TaRDIS.

To achieve this, the proposed extension supports multiple project types: Babel-based Java projects, PTB-FLA projects for federated learning, and DCR choreographies for distributed execution. It ensures seamless integration with essential tools such as Maven for Java projects, Python virtual environments for PTB-FLA, and Docker for running DCR-based prosumers. This focus on usability, flexibility and cross-platform compatibility was key to creating an extension that enhances productivity while maintaining robust functionality.

For this particular use-case, the proposed scenario to build the TID environment is to:

- Create a base GMV workspace in VS Code, a common repository where all the project elements will be present.

- Invoke, from the IDE, each of the tools such as PTB-FLA and Babel, that will be seamlessly integrated and perform the development actions that are stated in each of the tool's descriptions in this document.

4.3.6.2 Activity Diagram

See section **Error! Reference source not found.**.

4.3.6.3 Software Development Life Cycle

See section **Error! Reference source not found.**.

4.3.6.4 Architecture and Tools

The TaRDIS Visual Studio Code extension is a comprehensive toolbox designed to streamline the development and management of distributed systems and projects. The extension provides a range of functionalities, including project creation, protocol importing, and class generation, to help developers quickly and efficiently set up and manage their projects. This manual outlines how to use the TaRDIS extension and its current features.

For a detailed description of the tools and frameworks used in the TaRDIS IDE extension (Node.js, TypeScript, VS Code API, Webview API, Maven, Docker, Yeoman, etc.), see Section 4.4.

Tools required for operation:

To ensure seamless operation, the following tools are required:

- **Java Development Kit (JDK):** Java 21 is essential for Babel-based projects, providing the runtime environment for Java applications.
- **Node.js:** Serves as the runtime environment for the extension itself, managing dependencies and executing commands.
- **Maven:** Used for dependency management and build automation in Java projects.
- **Python:** Required for PTB-FLA projects, with virtual environments ensuring package isolation.
- **Docker:** Facilitates containerized execution of DCR choreographies.
- **Visual Studio Code:** Serves as the primary development environment.
- **Git:** Enables version control, code sharing, and collaborative development.

4.3.6.5 Verification

This tool is mainly for performing development, although it can be foreseen that some of the TaRDIS verification tools such as the C Code Static Verification or the Dynamic (Runtime) / Monitoring or Testing may be in the future also integrated in the IDE, thus allowing the IDE the capability of invoking them and hence allow verification and validation of swarm environments.

4.3.6.6 Machine Learning

ML tools are not directly used in this recipe, although the IDE may integrate AI/ML tools such as PTB-FLA or FEDRA, but these shall then be described separately on their own sections.

4.3.7 EDP Use Case Development Methodology

4.3.7.1 Scenario and Requirements

The EDP UC development methodology is used to guide a user to simulate the EDP UC software development process, which considers the EDP baseline with the test scenarios, the requirements that were defined and the inputs necessary that will lead the EDP UC software to its output.

4.3.7.2 Activity Diagrams

The EDP UC development methodology is supported by the TaRDIS toolkit and it takes a sequential approach by using multiple tools such as the Coordination Application, the DCR Choreographies, the Babel and the Fedra/Pruning tool to achieve the desired objectives of the use case. The development is conducted through three phases.

Phase 1: Initialization/Training

In the first phase, it is necessary for the developer to have a defined energy community universe, in other words, the number of prosumers and the community orchestrator information, the topology of the community and the geoinformation of each peer. It is also necessary to have the historical dataset of the production and consumption of these same prosumers within an energy community in order to train the Fedra model.

After defining the community, it is necessary to use community configuration to define the training parameters for the ML models. With these parameters defined, then you can use the historical data of each prosumer and train the local Fedra/Flower models. Then, the Pruning tool can be used to create a lightweight trained model for forecasting.

Then, this process of forecasting the production and consumption of the prosumers within the energy community will be done hourly.

The output of this phase will be the forecast production and consumption of energy, and the community configuration assigned to the known energy community universe, as shown on Figure 43.

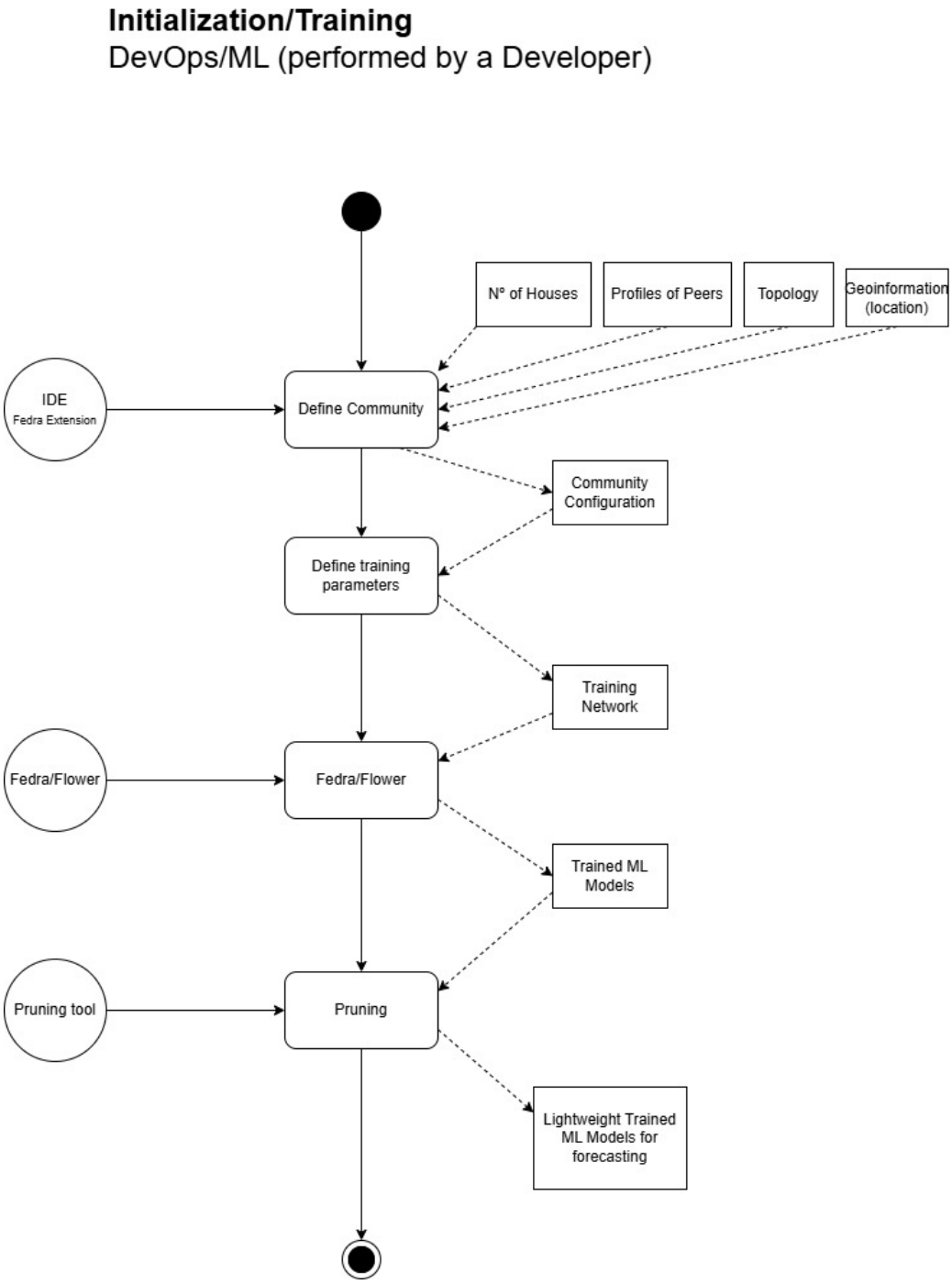

Figure 43: Phase 1: Initialization and training activity diagram.

Phase 2: Defining DCR Choreography model for usage scenario

In this second phase of the project, the developer will develop a DCR Choreography model that will define activities that each participant can perform in a given usage scenario (DCR Interface events).

ReGraDa / Dynamic Condition Response (DCR) Choreographies provide a declarative, event-driven, and stateful approach to the specification of workflows within a swarm application.

The DCR process methodology defines activities that are essential for a DCR model to correctly capture desired system behaviour:

1. Determine roles of the participant and their activities in the choreography
2. Determine flows (how the activities/events should be executed)
3. Determine constraints and rules (is there any constraints/conditions for some events?)

A simple activity diagram for the DCR Choreography model development process is given in Figure 44.

The base input for the DCR Choreography model development is the end-user requirements describing desired system behaviour. These requirements should be defined by domain experts (in this use case, EDP experts).

From these requirements the developer should be able to determine roles of all the participants in a given usage scenario.

Once the role of the participants (peers) is determined, the developer is using TaRDIS DCR Editor tool to define activities each peer is performing in a given choreography (usage scenario). During this process the developer will identify events and their interactions, i.e., defining the process flow.

As some events in the system can have certain preconditions or some interactions should not be possible at all, the developer will again consult the requirements and enhance the DCR Choreography model with such constraints - hence defining rules for choreography execution.

After the working draft of the DCR Choreography model is defined, the validation of the model, using the embedded testing runtime environment, is performed in consultation with domain experts. This validation determines if the DCR Choreography model captures correctly desired overall system behaviour.

Funded by the European Union

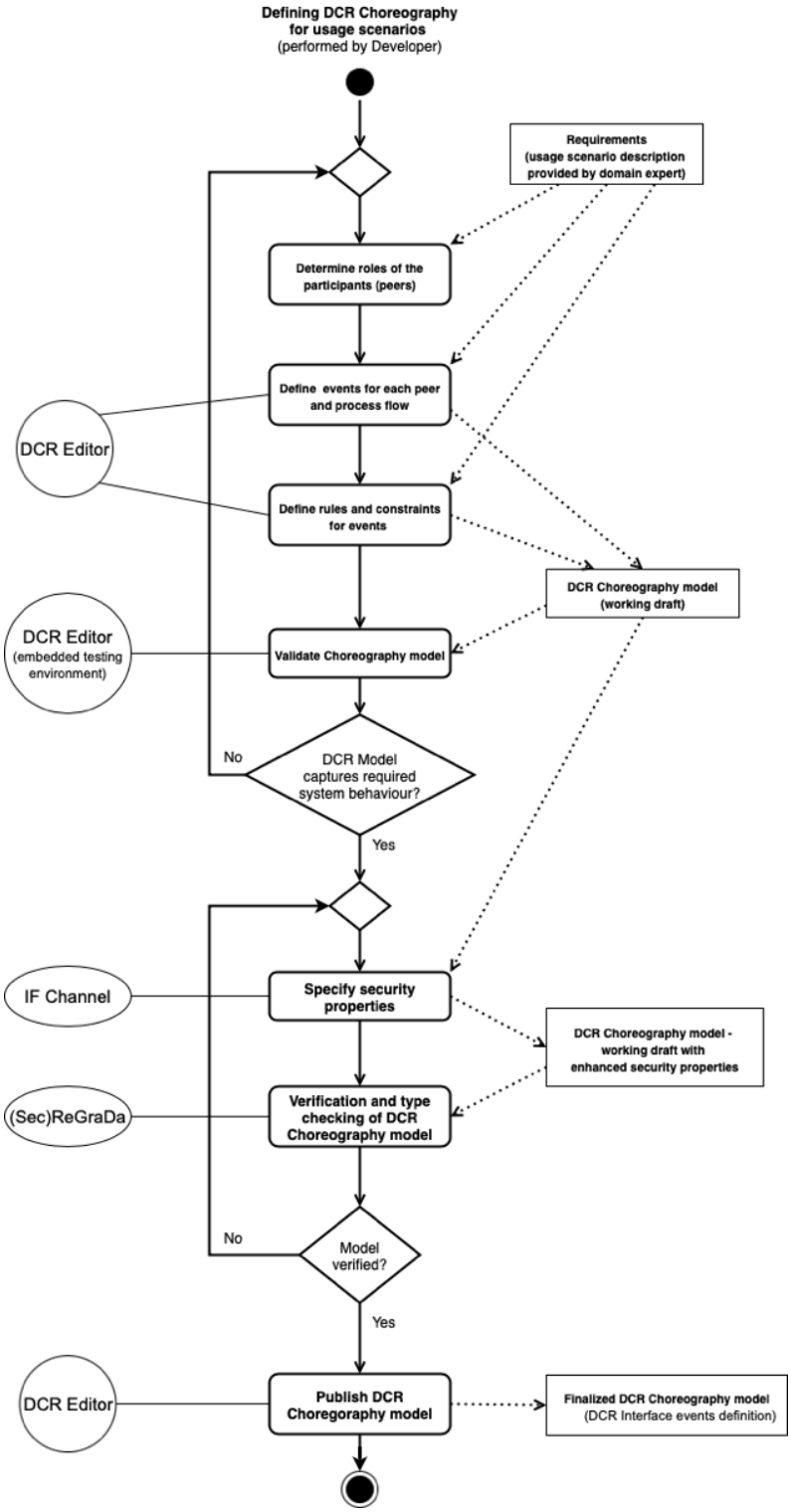

Figure 44: Phase 2: Defining DCR Choreography model for usage scenario activity diagram.

As DCR methodology is intended to allow for incremental development, it is expected that these steps are performed in iterations, by refining the model, and capturing more precise behaviour in each cycle.

After the working draft of the DCR Choreography model is validated to be a good representation of desired system behaviour, security aspects should also be taken into

consideration. By using IFChannel and (Sec)ReGraDa the developer can define security constraints on the DCR Choreography model and perform verification.

Once the verification is completed, the developer can publish the DCR Choreography model for use in other phases.

The development process is well suited for agile development methodology, where the process model can be developed incrementally as knowledge about fine details of the process is growing over time. But the same development lifecycle can also be implemented using the waterfall approach.

Phase 3: Development

In the third phase, the developer should have the participants' profiles, the trained Model, the community topology and the DCR Interface events. For this phase, it is necessary for the developer to have the coordination application which is a ML algorithm python code that enables a set of actions based on data from a database that needs to be updated hourly as well. This database is stored in every single prosumer, so it only has information of its own self. So, the developer needs to populate the data it needs initially such as the type of peer and then obtain the rest of the information from the output of the Fedra model which was explained in phase 1.

After confirming it has all of these tools ready, then the developer can start defining the policies that will be used later to enable events. After defining these policies, it is necessary to code it via the coordination application. After this, then it is necessary to validate the combined policies via a test report. After analysing this test report, it is necessary to assert if the policies are satisfactory or not. If not, then they need to be defined again. If they are satisfactory, then this phase is finished. Bear in mind that this step is done simultaneously by each peer of the energy community.

The output of this phase will be the set of enabled events for each peer to use by the Babel framework, as shown in Figure 45.

Funded by the European Union

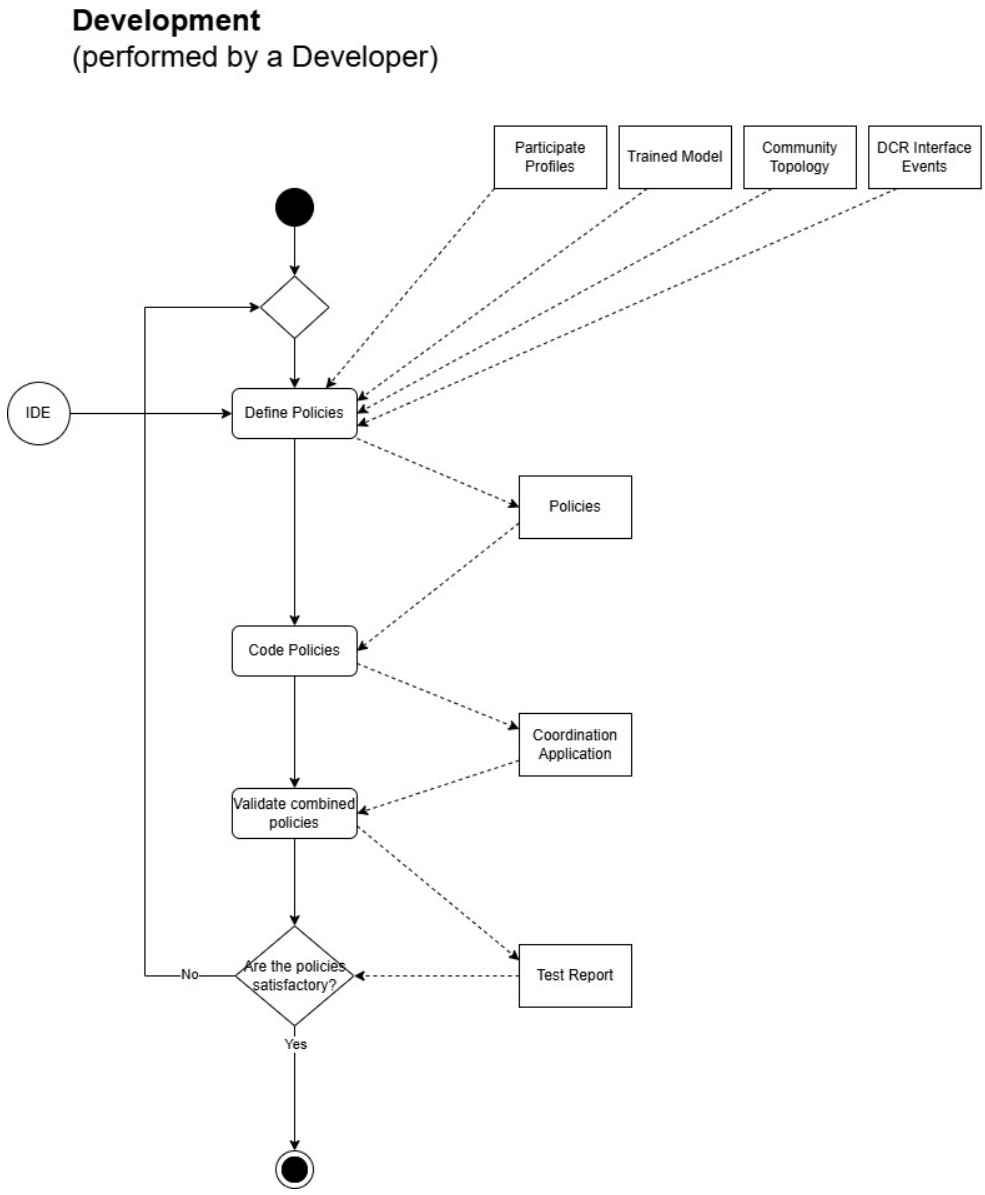

Figure 45: Phase 3: Development activity diagram.

Phase 4 (Babel Code/final code)

This phase is based on the use case recipe of the EDP use case recipes. The input of this phase is the policies with the enabled events for each prosumer that was defined on the previous phase as well as the community topology and chosen protocols for deployment (Figure 46).

Initially the configuration for each node is created through the information passed as input (i.e., the necessary parameters to enable node communication). With this information the appropriate communication and membership protocols are chosen and populated with the configuration information from the previous steps.

Finally, the configured protocols and the corresponding DCR events are combined to define the interactions in the system, leveraging the communication mechanisms provided by Babel.

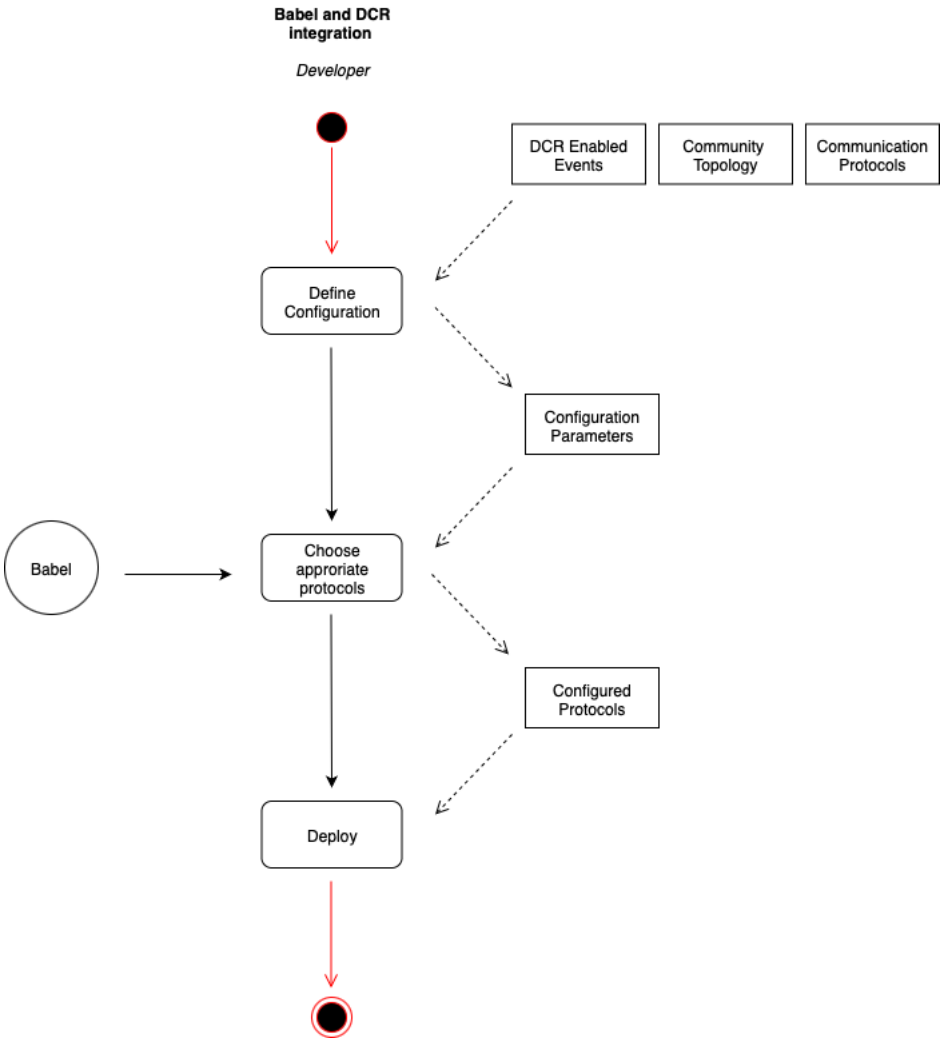

Figure 46: Fourth phase diagram.

As such, by using the Babel services, each prosumer will act according to the list of events enabled by the DCR choreography according to the protocols defined in the input on top of the community topology defined by the communication protocols.

The output of this phase will be the final code ready for deployment on devices, as shown in Figure 46.

4.3.8 DCR Choreographies – Create choreographies for different peers, which will define the set of actions available

4.3.8.1 Scenario and Requirements

ReGraDa / Dynamic Condition Response (DCR) Choreographies provide a declarative, event-driven, and stateful approach to the specification of workflows within a swarm application. The language supports both graphical and textual representation, providing a high-level abstraction that is both intuitive and human-readable, as well as machine-executable. An Energy community is a network of peers, which can be consumers or producers, in a delimited geographical region, who collectively manage and share energy resources. The TaRDIS

toolbox aims to enable each peer to connect within its community and make automatic agreements about energy exchange.

There are specific roles that any peer can play in the energy community - consumer, producer (or both as prosumer), and the community orchestrator, responsible for the external interactions with other communities' orchestrators and Distribution System Operator (DSO).

DCR Choreographies are used to specify global behaviour of the swarm and define each peer's role in certain scenarios, defining the events that are available in the system at a given time in the different scenarios, as well as actions that each peer may take upon detecting certain events.

A DCR choreography specifies the messages exchanged between peers as well as constraints on control flow that define causality between events and messages in the system's logic.

4.3.8.2 Activity Diagram

Phase 2 (Defining DCR Choreography model for usage scenario) of the UC development methodology (see section 4.3.7.2):

- The tools that support this phase are the DCR Editor and the runtime support system. The developer should have the requirements of all the scenarios that they intend to support.
- The first task is to specify all the roles of the participants and the corresponding parameterisation.
- The second task is to define all events and their relations and constraints that are needed to support the process. This output will be used to determine the behaviour of the coordination application. This defines the choreography.
- The Choreography can be validated for well-formedness using the tools linked in the DCR Editor.
- This concludes the iterative first step. Any errors or improvements can be introduced by repeating the previous steps. The output of this phase will be the set of enabled events for each peer to use by the Babel framework.
- Next, we can specify security levels for each event and check for information-flow control.
- Finally, the DCR compiler, accessible through the TaRDIS IDE, produces the compiled version of the choreography that can be shipped to the Babel nodes.

4.3.8.3 Software Development Life Cycle

In this example, the life cycle of a choreography corresponds to an iterative life cycle. For each iteration there is a short waterfall-like development life cycle. The life cycle is determined by the organization developing the choreography, rather than by TaRDIS itself. Other lifecycles (e.g. Scrum) could easily be used instead. The analysis of a choreography running on the can

give feedback to add new messages between participants, constraints between such events, or change the roles of the participants.

4.3.8.4 Architecture and Tools

The DCR Editor (T-WP3-03) and accompanying command line compiler are used to define DCR Choreographies. This enables specification of choreographies, extending DCR Graphs with communication, and the compiler enables compilation of specified choreographies to managed services (using Babel) and produces the application that will run locally on the participant's device. These applications determine the actions that participants can execute to participate in specified choreography.

The DCR Editor is available as part of the TaRDIS Toolbox IDE. When a new project utilising DCR Choreography is created within the TaRDIS Toolbox, the DCR Editor extension automatically generates the necessary folder structure and configuration files to support execution and ensures that all dependencies are properly set up. The project structure includes a dedicated workspace for the DCR logic, configuration files for managing dependencies, and a preconfigured environment that ensures compatibility with the required tools. The user is guided through the steps of compiling and running the choreography. The TaRDIS extension simplifies the execution of DCR Choreographies by integrating directly with Docker, Maven, and other required tools. It provides an interactive interface where users can configure and execute their distributed workflows with minimal setup.

The DCR Editor is a standalone tool to define the DCR Choreographies. Once compiled, DCR Choreographies integrate into the overall architecture of TaRDIS applications in several ways:

- (Sec)ReGraDa DCR for static and dynamic verification of data confidentiality using information flow control techniques.

- Uses the Babel communication framework and APIs as runtime support

4.3.8.5 Verification

- **Static verification:**
    - The provided compiler ensures that DCR Choreographies are statically checked for correctness during translation to the executables running on each participant’s devices. The property being checked is called projectability.
- **Runtime verification:**
    - The DCR Editor provides an environment for running and testing DCR Choreographies, running each participant’s behaviour in a Docker container, simulating a distributed environment. Throughout the execution, users can observe real-time logs in VS Code’s integrated terminal, allowing for direct monitoring of message exchanges, event triggering, and overall choreography behaviour.

4.3.8.6 Machine learning

Machine learning does not play a role in this specific recipe.

4.3.9 Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator

4.3.9.1 Scenario and requirements

The goal of the Coordination Application is to select an adequate DCR choreography for a given prosumer/community orchestrator, based on the forecast values produced by the Fedra/Pruning tools. The application uses data from Fedra/Pruning tools, such as the forecasted production and consumption of energy, which will then be used to calculate the energy balance of that peer. Another feature that will be used is the peer's type, which can be a prosumer or a community orchestrator, the action which will be directly linked to the energy balance variable (for example, if the energy balance is negative then action should be buy, if the energy balance is positive, then the action should sell, and so on) and monitor the performance from the perspective of the current peer. Based on these variables, it will use a decision tree algorithm to select which event will trigger in order to start the process that will be used later on for this prosumer.

Bear in mind that this tool is present within every single test use case scenario established, so this recipe will be present in every single test use case scenario.

4.3.9.2 Activity Diagram and Software Development Life Cycle

Phase 3 (Development) of the UC development methodology (see section 4.3.7.2):

- The tools that support this phase are the DCR Choreographies, the Fedra Model and the Coordination Application.
- The developer should have the participants' profiles, the trained Model, the community topology and the DCR Interface events. For this phase, it is necessary for the developer to have the coordination application which is a ML algorithm python code that enables a set of actions based on data from a database that needs to be updated hourly as well. This database is stored in every single prosumer, so it only has information of its own self. So, the developer needs to populate the data it needs initially such as the type of peer and then obtain the rest of the information from the output of the Fedra model which was explained in phase 1.
- After confirming it has all of these tools ready, then the developer can start defining the policies that will be used later to enable events. After defining these policies, it is necessary to code it via the coordination application. After this, then it is necessary to validate the combined policies via a test report. After analysing this test report, it is necessary to assert if the policies are satisfactory or not. If not, then they need to be defined again. If they are satisfactory, then this phase is finished. Bear in mind that this step is done simultaneously by each peer of the energy community.

- The output of this phase will be the set of enabled events for each peer to use by the Babel framework.

4.3.9.3 Software Development Life Cycle

The development life cycle process that corresponds to this recipe is a waterfall development life cycle. In this example, the life cycle is iterative. For each iteration there is a short waterfall-like development life cycle. The life cycle is determined by the organization developing the choreography, rather than by TaRDIS itself. Other lifecycles (e.g. Scrum) could easily be used instead.

4.3.9.4 Architecture and Tools

The coordination application is used in a centralized location for this recipe. Each prosumer/community orchestrator will run this application locally based on the information gathered from the forecast algorithms. Depending on the type of peer, the corresponding DCR choreography is used within the component's application. If it is a consumer, it will do a particular DCR choreography; if it is a producer, it will do another DCR choreography, and so on. By doing this, we can apply all choreographies to every single component. Based on whether the peer is a producer or consumer, the corresponding forecasted data that is available through the Fedra and Pruning Tools is used. This strategy even allows a component to assume a completely different role if needed, e.g., a producer transitioning into a Community Orchestrator.

The process is schedule driven. This means the time scheduler triggers each single tool to be used, in this use case. After the schedule starts, the Fedra and Pruning tools provide the forecast data for the next hour, for production and consumption for each producer and consumer that exists in the energy community. Then, the community energy balancing application starts (bear in mind that this app is running on every single component/local machine). Through a decision tree algorithm, the defined choreography will run within that peer, which, by consequence, will define the set of actions available for itself, as shown in Figure 47.

Funded by the European Union

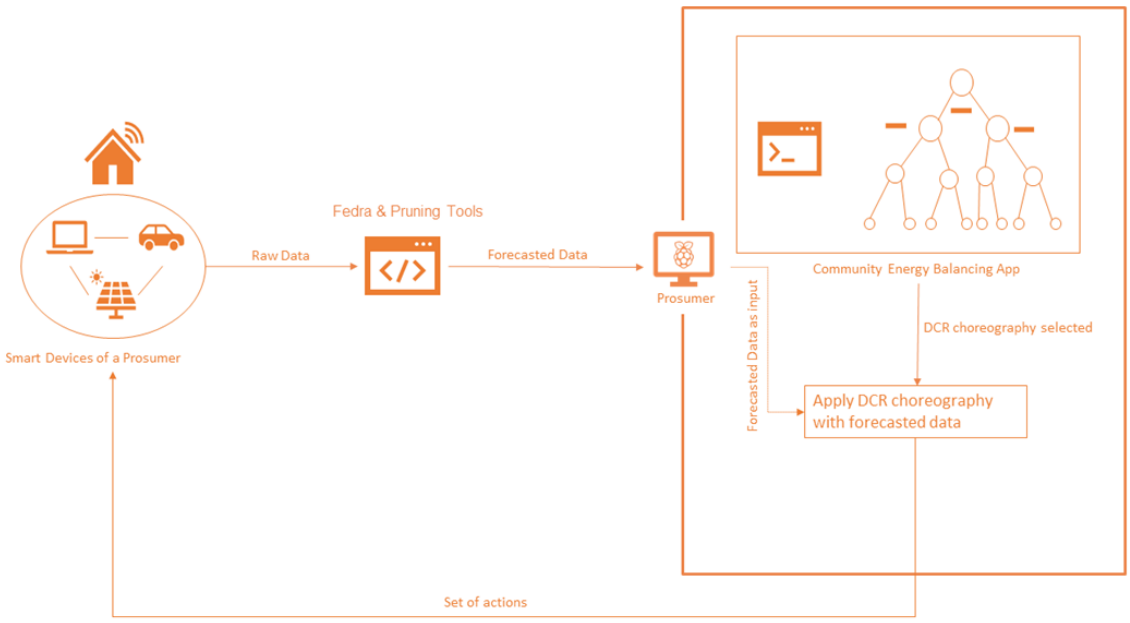

Figure 47: Community energy balancing application.

4.3.9.5 Verification

Verification does not play a role in this specific tool (Coordination Application), however the DCR choreographies do have verification such as:

- **Static verification**:
    - The provided compiler ensures that DCR Choreographies are statically checked for correctness during translation to the executables r unning on each participant's devices. The property being checked is called projectability.
- **Runtime verification:**
    - The DCR Editor provides an environment for running and testing DCR Choreographies, running each participant's behaviour in a Docker container, simulating a distributed environment. Throughout the execution, users can observe real-time logs in VS Code's integrated terminal, allowing for direct monitoring of message exchanges, event triggering, and overall choreography behaviour.

4.3.9.6 Machine Learning

Initial learning

To understand which type of classifier should be used for the coordination application, several algorithms such as decision trees, Random Forest, GradientBoosting and others were tested. Before testing the several algorithms, some steps were made first such as solving class imbalance, analysing feature quality, data preprocessing and also hyperparameter tuning. After doing all of these steps, then we trained the algorithms.

Models' usage

Models are called during an algorithm that will call every single tool that will be used in this use case. By using the forecasted data from the Fedra/Pruning tools then you can use these algorithms to calculate the type of choreography.

Continuous learning process

There is no continuous learning process in this recipe.

4.3.10 Fedra/Pruning Tool – Use FL algorithms to train ML models for forecasting the energy production and consumption of smart homes

4.3.10.1 Scenario and Requirements

This tool is present within every single test use case scenario established, so this recipe, and its description, applies to every test scenario.

The scenario is described as follows: Leveraging historical measured data from a smart home, this recipe enables the training of a Machine Learning (ML) model to forecast two critical parameters: the time-varying energy generated by renewable sources and the energy consumption in the household, as shown in Figure 48.

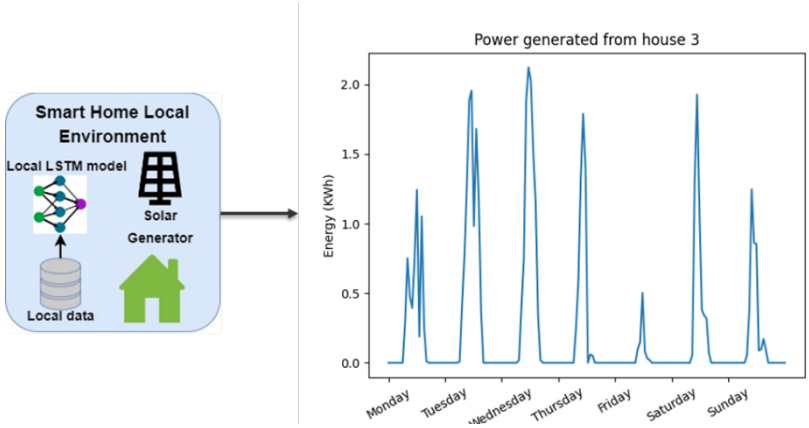

Figure 48: Historical local data of generated energy in the smart home local environment.

Since these two parameters are time-varying, the Long Short-Term Memory (LSTM) networks are used for forecasting purposes as ML models. Taking into account that we have multiple houses inside an energy community, the target is to perform collaborative training of these two ML models (one for energy generation forecasting and one for energy consumption forecasting) in a decentralized manner, as depicted in Figure 49. The decentralized federated learning (FL) presented here promotes collaborative shared intelligence among different smart homes/clients, while ensuring the accuracy of their data, since updates of the ML model parameters are only shared across the FL participants. Finally, this configuration does not require a central server/agent that acts as an aggregator of a global ML model.

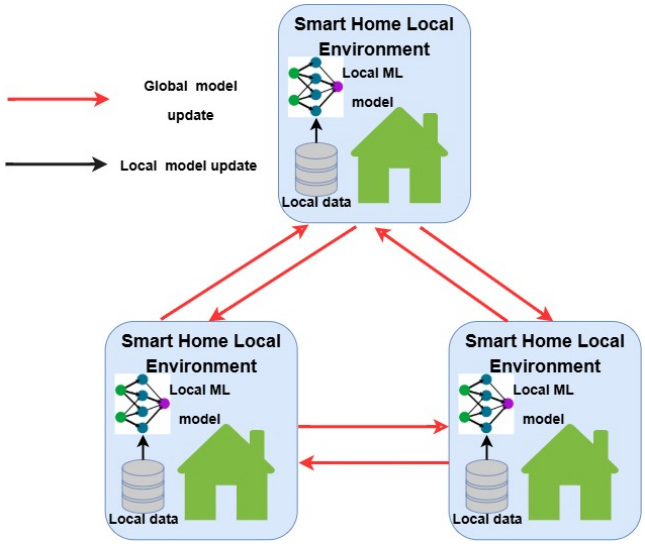

Figure 49: Decentralized configuration of the smart homes and the local LSTM models.

Once the FL training is completed, the trained LSTM models are stored locally in each smart home, to be used "online" (during runtime) for inference purposes. The trained LSTM models can then be transformed into more lightweight versions that exhibit increased energy efficiency with respect to the original models.

4.3.10.2 Activity Diagram

Phase 1 (Initialization/Training) of the UC development methodology:

- Developer/ML developer that will define the energy community parameters (number of prosumers, community orchestrator, topology of the community, information of each peer that will participate in the FL framework).
- Test/modify the python code that performs the FL training of the ML model (LSTM model for forecasting) and the availability of the dataset in each node/prosumer, defining also the training hyperparameters (e.g., learning rate, number of epochs, number of FL rounds, minimum number of peers in the FL framework)
- Once the configuration is completed by the developer/ML engineer, the training is initiated when the required number of peers connect to the FL framework.
- The process is finalized and the trained local ML models are stored at the location of the peers. The developer/ML engineer then selects to prune the local ML models by using the pruning TaRDIS tool, configuring the pruning rate and the required accuracy level.
- The pruned lightweight ML models are stored at the location of the peers/prosumers and are ready for online inference (online predicting the upcoming energy consumption/generation of each node).

4.3.10.3 Software Development Life Cycle

In this example, the life cycle is a waterfall-like development life cycle. The life cycle is determined by the organization developing the choreography, rather than by TaRDIS itself. Other lifecycles (e.g. Scrum) could easily be used instead.

4.3.10.4 Architecture and Tools

The **Fedra** tool developed in WP5 will be used to host the decentralised, federated learning of the two LSTM models (one for energy generation and one for energy consumption forecasting) that are deployed locally on the edge devices/nodes/clients (smart homes). The **Fedra** framework includes the code base of the ML training for the LSTM models that are deployed at each edge node and starts the FL training process with the local datasets. After several federated rounds (in each round multiple epochs are used to train the local models), the FL training is concluded, quantifying the training and testing losses. Then, the trained LSTM models are transformed in their lightweight version by using the **Pruning** tool, which outputs the new models, ready for online inference with real-time input samples. Figure 50 presents an overview of the architecture.

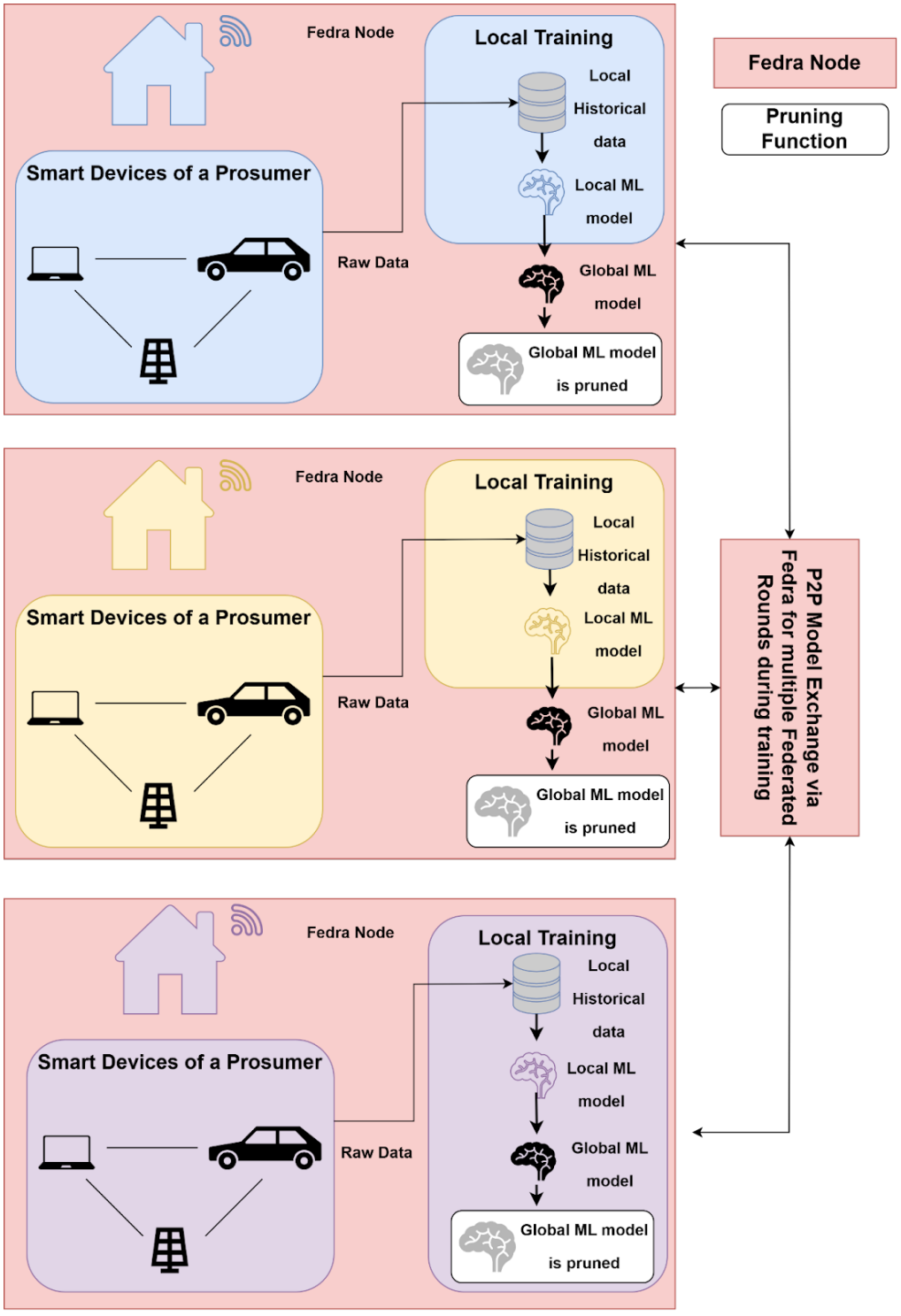

Figure 50: Fedra architecture.

In terms of requirements or dependency with other TaRDIS tools, **Fedra** framework uses the libp2p library for direct communication between the peers/nodes in the decentralized configuration, enabling the peer-to-peer communication among the participating client nodes.

4.3.10.5 Verification

Verification does not play a role in this specific recipe.

4.3.10.6 Machine learning

This section describes how ML tools are used in this recipe and how they integrate in the system development and deployment.

Initial learning

The initial learning is based on a training python code that is integrated inside the Fedra tool and is application-specific (for the energy forecasting). For instance, we have also developed a DRL code for smart home energy management system that can also be integrated inside Fedra (or a developer can integrate his/her third-party base code for training an ML model for another application). Fedra initializes the FL training and monitors the process for the federated rounds (training and testing losses).

Models' usage

The trained LSTM models are not used in this recipe.

Continuous learning process

There is no direct continuous learning process in this recipe. However, a developer can re-run the Fedra tool with additional training data, for instance, having gathered additional data from the real smart home environment. To this end, by selecting the subset of the newly observed data that have not been utilized during the previous training, the developer can perform a soft retraining/tuning of the pre-trained LSTM models.

Continuous learning process integration in the system

Not applicable.

4.3.11 Babel - Layer of communication between entities

4.3.11.1 Scenario and Requirements

This tool is present within every single test use case scenario established, so this recipe, and its description, applies to every test scenario.

The main objective of the Babel tool in this use case is to enable the communication between the different entities present.

Babel offers a specialized membership service for energy markets, to allow data exchange between the different nodes in the system (i.e., consumers, prosumers, communication orchestrator, etc.).

To achieve this, Babel acts as a middleware (Figure 51) which receives the output from the tools above (*e.g.*, DCR Choreographies for the EDP Use Case) and sends its content to the corresponding destinations by leveraging the previously mentioned membership service.

When data arrives at a node through Babel's network components, Babel notifies the corresponding tools interacting directly with Babel to relay the information sent by other nodes in the community.

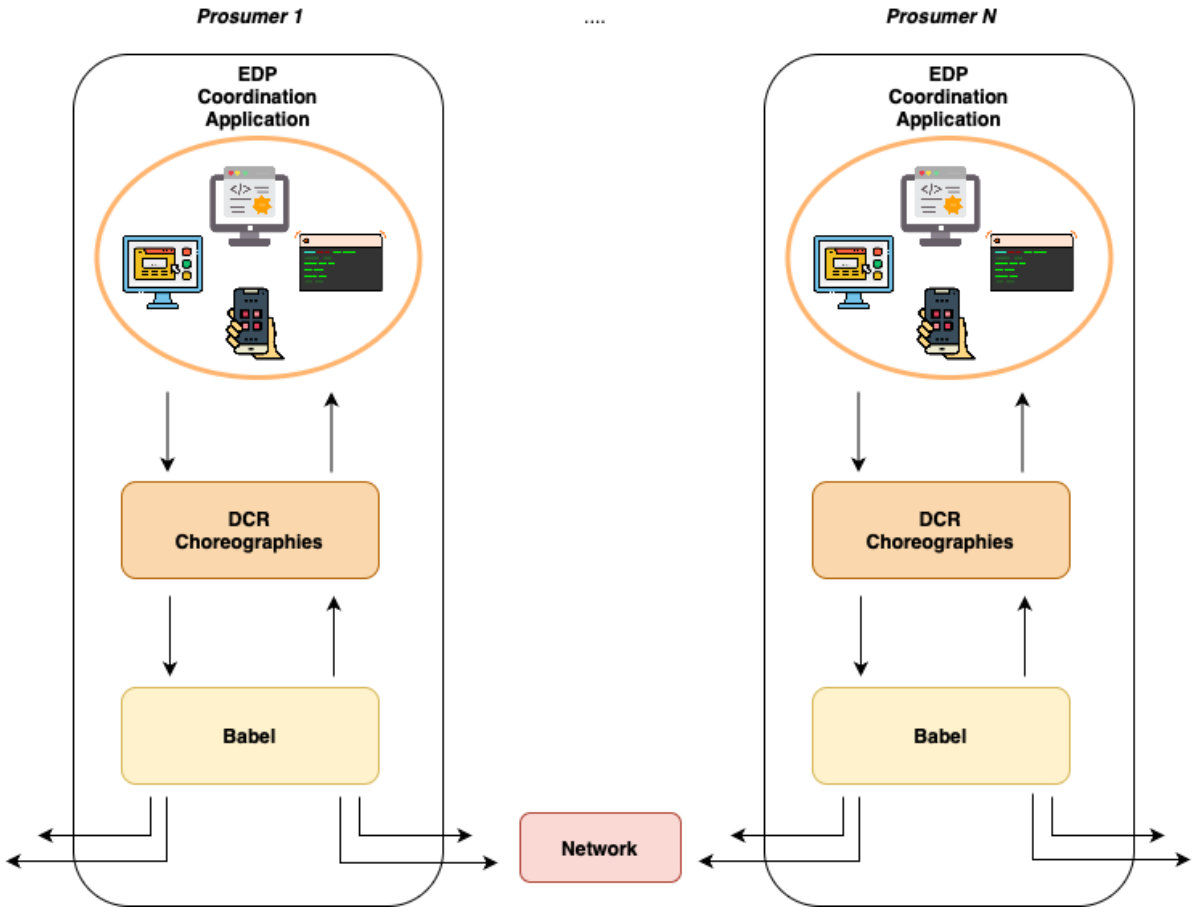

Figure 51: Babel EDP Use Case overview.

4.3.11.2 Activity Diagram

Phase 4 (Final Code and Deployment) of the UC development methodology :

1. The developer collects the input from the previous phase (i.e., DCR enabled events) as well as the communication topology and the appropriate Babel protocols for the scenario.

2. With this information, the configuration parameters are generated (e.g., various neighbours in the community, etc.) in order to be passed on to the protocols;

3. The Babel developer passes the configuration to the chosen communication and membership protocols, thus generating ready to be deployed instances of the protocols

4. In the final step the final code is generated with the DCR events that define the information flow encoded and the proper Babel protocols to ensure communication between the different entities of the energy community.

4.3.11.3 Software Development Life Cycle

Babel uses the *DevOps* methodology to provide continuous delivery and integration during its different updates.

4.3.11.4 Architecture and Tools

The Babel framework focuses on easing the complexity of building distributed protocols. Babel's design emphasized simplicity and performance, enabling developers to focus on the core logic of protocols without being encumbered by the intricacies of low-level operations by offering extensible networking channels, providing developers with the flexibility to implement a wide range of communication patterns, from peer-to-peer interactions to client-server models.

It possesses an event-driven model where protocols are depicted as state machines reacting to events such as timers, network messages, or intra-process notifications. Developers design protocols independently to facilitate the reuse of existing

implementations and reduce development overhead, while supporting comprehensive support for security mechanisms including protocol authentication, encrypted communications, and access control.

In order to keep up with the ever-increasing complexity of swarm systems, Babel offers automated mechanisms for the dynamic configuration of protocols and applications (i.e., self-configuration) and capabilities for autonomously managing protocol lifecycles, monitoring performance, and optimizing resource allocation to ensure system robustness and efficiency (i.e., self-management).

Moreover, Babel is also available in Android devices, recognizing the critical role of mobile platforms as user interfaces in swarm applications.

4.3.11.5 Verification

Static Verification

Not applicable.

Dynamic (Runtime) / Testing

Not applicable.

Dynamic (Runtime) / Monitoring

Babel supports a metrics collection system embedded into its Core. This module collects node-related metrics, namely both hardware and kernel related metrics (e.g., CPU usage time, memory usage, etc.). Moreover, it's also possible to collect custom metrics. For example, users

may decide to store custom metrics such as protocol related metrics, or tracing information collected by the related tools watching over running applications.

All collected metrics are stored in a centralized place inside a specialized time series database, which can be later on retrieved to monitor or visualize the current execution status of the process.

4.3.11.6 Machine Learning

Machine learning does not play a role in this specific recipe.

4.3.12 Combining Tools: Fedra/Pruning Tool with Coordination App – Coordination recipe based on forecast data

4.3.12.1 Scenario and Requirements

This combination of tools is present within every single test use case scenario established, so this recipe, and its description, applies to every test scenario.

The target of this use case is to use the outputs of the "*Fedra/Pruning Tool – Use FL algorithms to train ML models for forecasting the energy production and consumption of smart homes"* recipe, i.e., the trained (and pruned) LSTM models in runtime for online inference, as depicted in Figure 52.

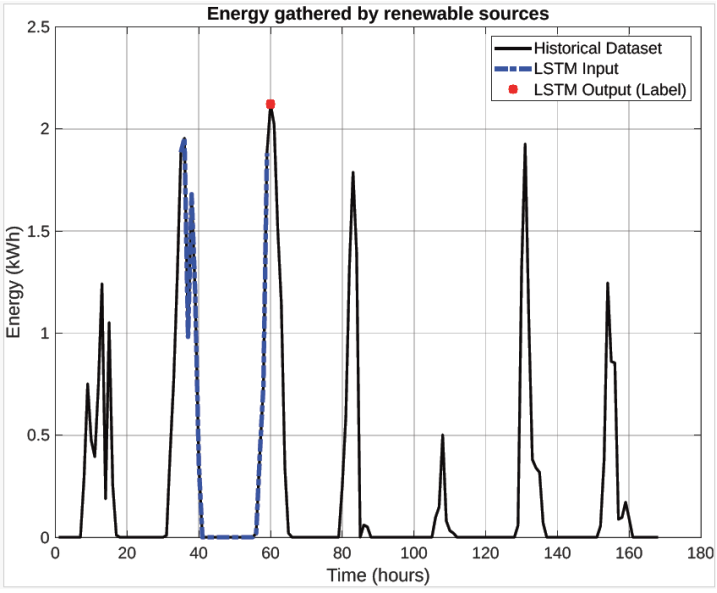

Figure 52: Input and Output of the LSTM models during the online inference.

To this end, the LSTM models provide the forecasting values for energy consumption and generation, as depicted in Figure 53 for the period of 1 week. These values are then acknowledged to the coordination application in runtime.

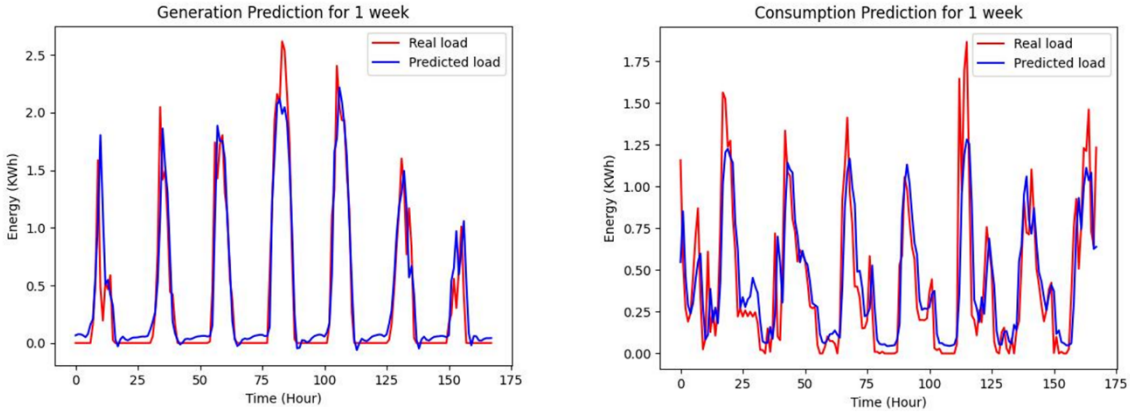

Figure 53: Sample online inference of the trained LSTM models for the energy generation (left) and the energy consumption (right) for the period of 1 week.

By using the outputs of these tools, we can use the EDP coordination application to calculate the type of choreography that should be used later on by the prosumer/community orchestrator.

This application's objective is to coordinate the different components: consumers, producers and the Community Orchestrator that are within an energy community. The coordination itself is based on trying to match everyone's needs with the lowest possible cost for each one. Fedra/Pruning's forecasts of consumption and production of energy will be used to directly calculate two features that will be used by the decision tree algorithm present in the coordination application.

The first feature being the energy balance of that component which will be either negative, positive or neutral (0).

By calculating the energy balance, we can then create the second feature which is the action. This feature will also have three unique values. If the energy balance is negative, then the component should buy energy. If the energy balance is positive, then the component should sell energy. If the energy balance is neutral, then the action value should be on hold.

Another feature that will be used to define the type of choreography is a feature called error_exists which is a Boolean value. If the previous actions that were done before this procedure worked out perfectly, without any issues, then the value is false. If there was some kind of error during the previous processes, then the value is true. The output of the type of choreography will go from 1 to 8 which means there are, for now, 8 different types of choreography that enable a set of actions differently.

Bear in mind that these tools are present within every single test use case scenario established so this recipe will be present in every single test use case scenario.

4.3.12.2 Activity Diagram

The software development life cycle of this recipe is the combination of the two recipes that describe the Coordination Application and Fedra. It is present on Phase 1 and Phase 3 of the UC development methodology (see section 4.3.7.2).

4.3.12.3 Software Development Life Cycle

The development life cycle process that corresponds to this recipe is an iterative model with a waterfall-like development life cycle in each iteration.

4.3.12.4 Architecture and Tools

The Fedra/Pruning tools and the coordination application are not used online in this recipe.

What is actually used are the trained LSTM models of each individual smart home that have been derived by the Fedra/Pruning tools in the recipe "*Fedra/Pruning Tool – Use FL algorithms to train ML models for forecasting the energy production and consumption of smart homes*". In this context, the required assets for this scenario are the pre-trained local LSTM models as objects and the Python code that performs the inference (input data pre-processing and feedforward operation of the ML models) (Figure 54).

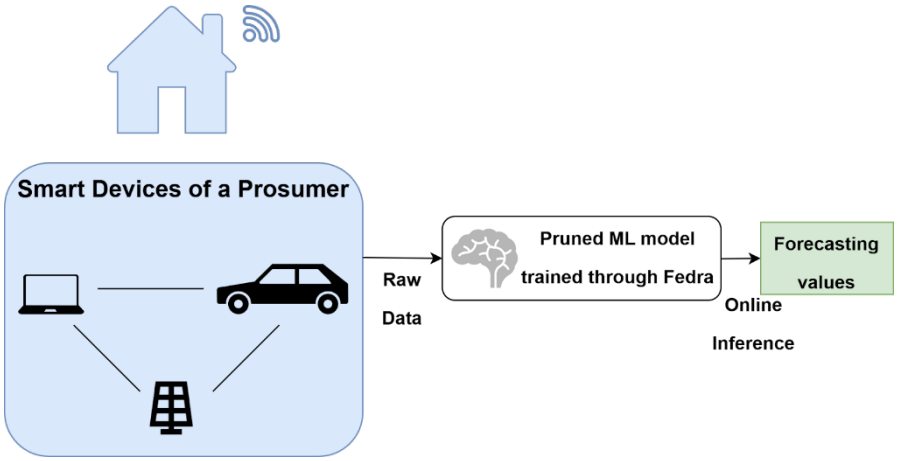

Figure 54: From prosumer smart devices to forecasting values.

As described in the recipe *"3.9. EDP Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator",* each prosumer/community orchestrator will run this application locally based on the information gathered from the forecast algorithms. Depending on the type of component, the corresponding DCR choreography is used within the component's application.

4.3.12.5 Verification

For the Fedra/Pruning and the coordination application tools, verification does not play a role.

4.3.12.6 Machine Learning

Describe if and how ML tools are used in this recipe and how they integrate in the system development and deployment:

Initial learning

- Fedra:

- The training has been performed in the recipe "*Fedra/Pruning Tool – Use FL algorithms to train ML models for forecasting the energy production and consumption of smart homes"*.

- EDP Coordination Application:
  - The training of the coordination application has been performed in the recipe *"EDP Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator".*

Models usage

- Fedra
  - The trained LSTM models are used for real-time online inference, yielding the forecasted values for the energy generation and energy consumption.
- EDP Coordination Application
  - In the coordination application, it is used locally by each entity after the forecasted values from the Fedra/Pruning tool becomes available.

Continuous learning process

- Fedra
  - There is no direct continuous learning process in this recipe. Similar to what was described in the "*Fedra/Pruning Tool – Use FL algorithms to train ML models for forecasting the energy production and consumption of smart homes"* recipe.
- EDP Coordination Application
  - For the coordination application, no.

4.3.13 DCR Choreographies with Coordination App – A decision tree that based on the input will define the type of choreography that a peer will have

4.3.13.1 Scenario and Requirements

This combination of tools is present within every single test use case scenario established, so this recipe, and its description, applies to every test scenario.

As it was described in the recipe *"3.9. EDP Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator",* the outputs of the Fedra and Pruning tools which is the forecasts of consumption of production of energy allow the EDP coordination application to calculate the type of choreography that should be used by the prosumer/community orchestrator.

The output of the type of choreography will go from 1 to 8, which means there are 8 different types of choreography that enable a set of actions differently.

So, after the Fedra/Pruning tool and the coordination application compute, the output of the coordination application will be linked to a unique choreography that was created previously. After the selection of the choreography, it will then perform the operations described on the DCR choreography selected.

As it is described in the recipe *"DCR Choreographies – Create choreographies for different peers which will define the set of actions available"*, DCR Choreographies are used to specify the global behaviour of the swarm and define each identity behaviour in certain scenarios - defining events that are available in the system in certain scenarios as well as actions that each identity may take upon detecting certain events. The output of the coordination application will be used to define the type of DCR choreography used.

4.3.13.2 Activity Diagram

The software development life cycle of this recipe is the combination of the two recipes that describe the Coordination Application and DCR Choreographies. It is present in the Phase 2 (Defining DCR Choreography model for usage scenario) and Phase 3 (Development) of the UC development methodology (see section 4.3.7.2).

4.3.13.3 Software Development Life Cycle

The development life cycle process that corresponds to this recipe is iterative. For each iteration there is a short waterfall-like development life cycle. The life cycle is determined by the organization developing the choreography, rather than by TaRDIS itself. Other lifecycles (e.g. Scrum) could easily be used instead.

4.3.13.4 Architecture and Tools

As it is described in the recipe *"EDP Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator",* each prosumer/community orchestrator will run this application locally based on the information gathered from the forecast algorithms. Depending on the type of component, the corresponding DCR choreography is used within the component's application. So, to work to this point, it is necessary to have the Fedra/Pruning tool and the coordination application available and with its processes run.

After receiving the input from the Fedra/Pruning tools then, as it is described in the recipe *"EDP Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator",* the coordination application will then calculate what type of choreography will be used for each prosumer or community orchestrator within the energy community.

4.3.13.5 Verification

DCR Choreographies

- **Static verification.**

- The DCR Choreography compiler ensures that all checked choreographies are statically checked for well-formedness and projectability during the translation to the executables to be distributed by the participants devices.

- **Runtime verification:**

  - DCR Editor provides an environment for running and testing DCR Choreographies, running each individual identity as a Docker container, simulating a distributed environment. Throughout the execution, users can observe real-time logs in VS Code's integrated terminal, allowing for direct monitoring of message exchanges, event triggering, and overall choreography behaviour.

EDP Coordination application

- There's no verification process.

4.3.13.6 Machine Learning

DCR Choreographies

Machine learning does not play a role in this tool within this specific recipe.

EDP Coordination Application

As it was described in the recipe *"EDP Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator":*

Initial learning

To understand which type of classifier should be used for the EDP coordination application, we tested several algorithms such as decision trees, Random Forest, GradientBoosting and others. Before testing the several algorithms, some steps were made first such as solving class imbalance, analysing feature quality, data preprocessing and also hyperparameter tuning. After doing all of these steps, then we trained the algorithms.

Models usage

They are called during an algorithm that will call every single tool that will be used in this use case. By using the forecasted data from the Fedra/Pruning tools then you can use these algorithms to calculate the type of choreography.

Continuous learning process

This recipe does not include a continuous learning process.

Learning process integration in the system

Not applicable to this recipe.

4.3.14 DCR Choreographies with Babel Stack – Layer of communication between identities with a defined set of actions

4.3.14.1 Scenario and Requirements

This combination of tools is present within every single test use case scenario established, so this recipe, and its description, applies to every test scenario.

The target of this recipe is to use the outputs of the "DCR Choreographies – Create choreographies for different peers which will define the set of actions available " recipe, i.e., the entities participating in the choreography for each specific test scenario, as the input for "Babel - Layer of communication between entities", i.e., the target nodes to send information to.

DCR Choreographies are used to specify and verify the global (coordinated) behaviour of all participants in the swarm. DCR Choreographies specify the sequence of actions that each participant can take by limiting the events that can be triggered in each state of the choreography. DCR Choreographies are a declarative mechanism that provides a flexible setting and does not limit the participants in relation to a global lock. The coordination application, linked to the participant's device, is limited to executing the actions made available by a local DCR Choreography layer, which executes the local behaviour of that particular participant. The global specification is translated into those local specifications, guaranteeing that all participants have the information necessary to follow the global process. The result of the DCR Choreography compiler is fed into a control and coordination layer in the Babel application layer, which intermediates the raw communication between participants and the coordination application connected to the devices. The compilation of DCR Choreographies also includes verifying the confidentiality of data.

Babel receives information from the DCR Choreographies tools, regarding the information that should be propagated (e.g., a forecast of a specific node) and the corresponding destination. For example, assuming that the DCR Choreographies tool outputs that prosumers P(1), P(2) and P(3) should receive forecast *f*, Babel efficiently propagates this information to each prosumer using its membership service. When a message arrives at a node, Babel notifies the DCR Choreographies module that new information is available.

4.3.14.2 Activity Diagram

The software development life cycle of this recipe is the combination of the two recipes that describe the Babel Stack and DCR Choreographies. It is present in the Phase 2 (Defining DCR Choreography model for usage scenario) and Phase 4 (Final Code and Deployment) of the UC development methodology (see section 4.3.7.2)

4.3.14.3 Software Development Life Cycle

Babel uses the *DevOps* methodology to provide continuous delivery and integration during its different updates. The life cycle of a choreography corresponds to an iterative life cycle. For each iteration there is a short waterfall-like development life cycle. The life cycle is determined by the organization developing the choreography, rather than by TaRDIS itself. Other lifecycles

(e.g. Scrum) could easily be used instead. The analysis of a choreography running on the can give feedback to add new messages between participants, constraints between such events, or change the roles of the participants

4.3.14.4 Architecture and Tools

Each node in the swarm, namely, each prosumer/community orchestrator, should run a local instance of Babel in its tool stack. Babel sends information to the other nodes in the swarm through the membership service and communicates primitives most suited for each communication scenario.

Thus, to work to this point, it is necessary to have DCR Choreographies and Babel tools available. DCR Choreographies are executed by one layer of the DCR stack. Each participant is parameterized with the specification for their own behaviour and acts accordingly. The DCR execution stack provides a REST API to allow communication between the Babel Layers and the coordination app (as shown in Figure 55). The communication between the DCR layer and the rest of the Babel stack works using Babel mechanisms (events).

The architecture of the two tools is depicted in Figure 55:

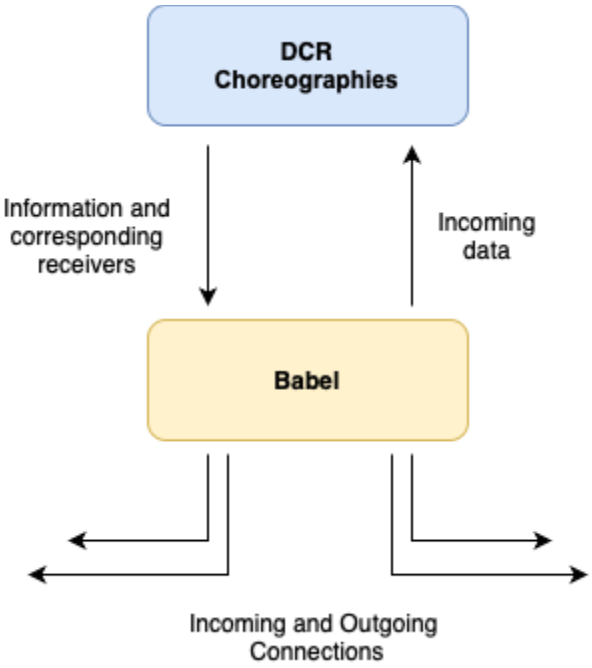

Figure 55: Connection and share of information between DCR choreographies and Babel.

4.3.14.5 Verification

DCR Choreographies

- Static verification.
    - Provided compiler ensures DCR Choreographies are statically checked for correctness during translation to executables.
- Runtime verification:

- DCR Editor provides an environment for running and testing DCR Choreographies, running each individual identity as Docker container, simulating a distributed environment. Throughout the execution, users can observe real-time logs in VS Code's integrated terminal, allowing for direct monitoring of message exchanges, event triggering, and overall choreography behaviour.

Babel

Not applicable in this recipe.

4.3.14.6 Machine Learning

Machine learning does not play a role in these tools within this specific recipe.

4.3.15 Use Case Tools - Multi-Level Grid Balancing – Configuration on how to create a decentralized system that allows peer-to-peer communication and energy transaction between prosumers within energy communities

4.3.15.1 Scenario and Requirements

The objective of this recipe is to, in a sense, replicate with the different tools described above, the EDP use case or a use case with a similar goal. The aggregation of all of these tools which then simulate the EDP use case is present within every single test use case scenario established so this recipe will be present in every single test use case scenario.

4.3.15.2 Activity Diagram

The software development life cycle of this recipe is the combination of all the previous recipes that describe every single individual and combining tools. It is present in every single Phase of the UC development methodology. See diagrams in figures 28, 29 and 30.

4.3.15.3 Software Development Life Cycle

The development life cycle process that corresponds to this recipe can be seen as a waterfall development life cycle.

4.3.15.4 Architecture and Tools

For this recipe it is necessary to use all of the previous recipes. The first step is to launch the IDE where you can create a TaRDIS project and select all of the tools that were identified in the recipes above, as described in the recipe *"How to deploy a project with the different tools available"*.

After creating the project, it is necessary historical data regarding the consumption and production of energy of a smart home or other prosumer profile. This allows the user to train the Fedra/Pruning tools and forecast consumption and production of energy for the short-term future. You can find more information regarding this process in the recipe "*Fedra/Pruning Tool*

– Use FL algorithms to train ML models for forecasting the energy production and consumption of smart homes".

By using the forecasted data from the Fedra and Pruning tools then, it is necessary to use the coordination application to select the type of choreography based on the peer's information and its energy balance. By using the procedure within the recipe *"EDP Coordination application - How to define a set of enabled events through a DCR choreography for a prosumer/community orchestrator"*, the type of choreography is now defined which allows the peer a set of actions to do and proceeds to follow to do the steps described in the recipe *"DCR Choreographies - Create choreographies for different peers which will define the set of actions available"*.

The last step is to use the Babel stack tool which allows the entities to communicate with each other and start the transactions between the prosumers and community orchestrators. This procedure is described in the recipe *"Babel - Layer of communication between entities".*

By doing all of these recipes sequentially, it allows to replicate a multi-level grid balancing system, as shown on Figure 56, Figure 57 and Figure 58.

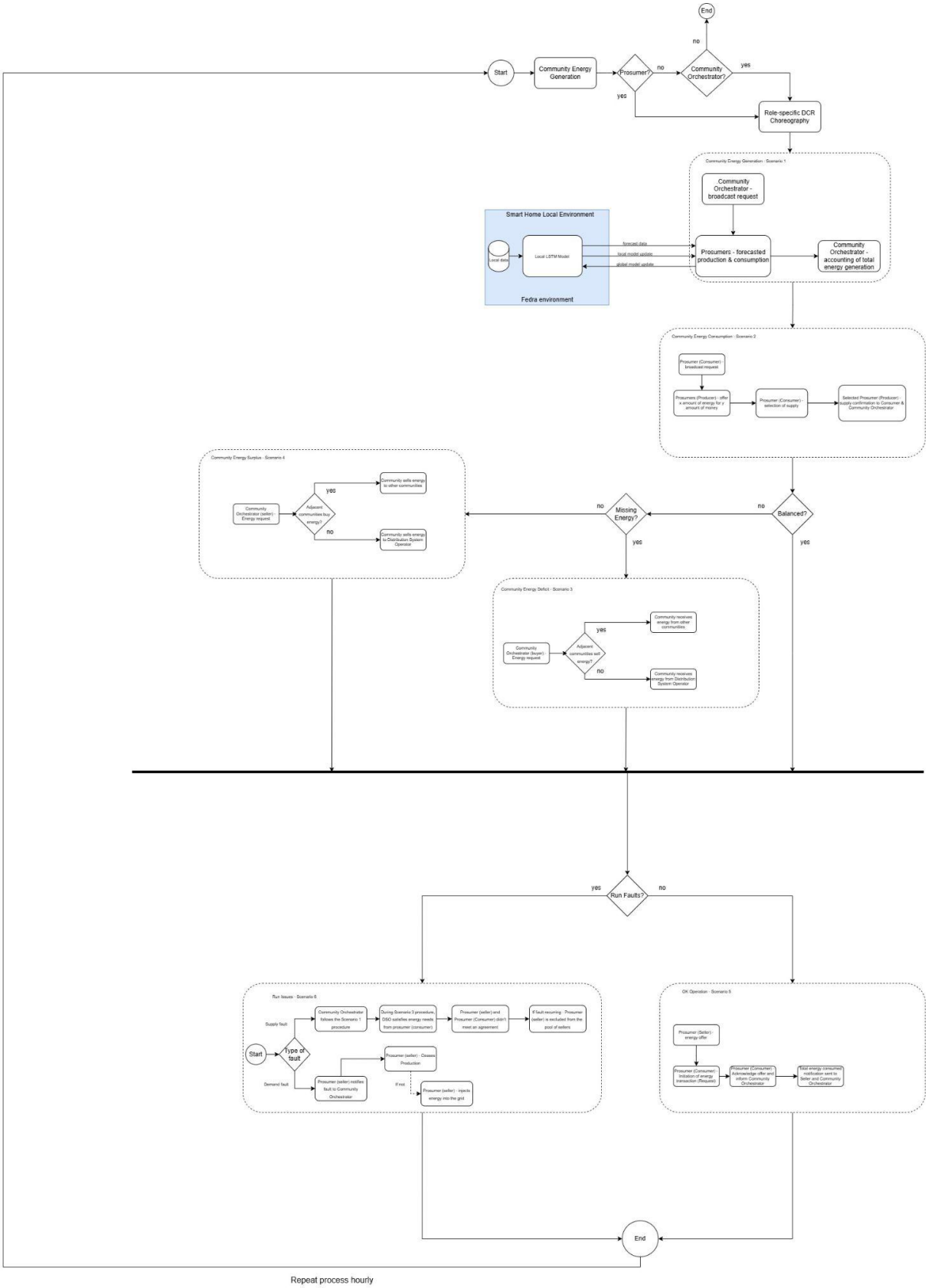

Figure 56: Multi-level Grid Balancing activity diagram.

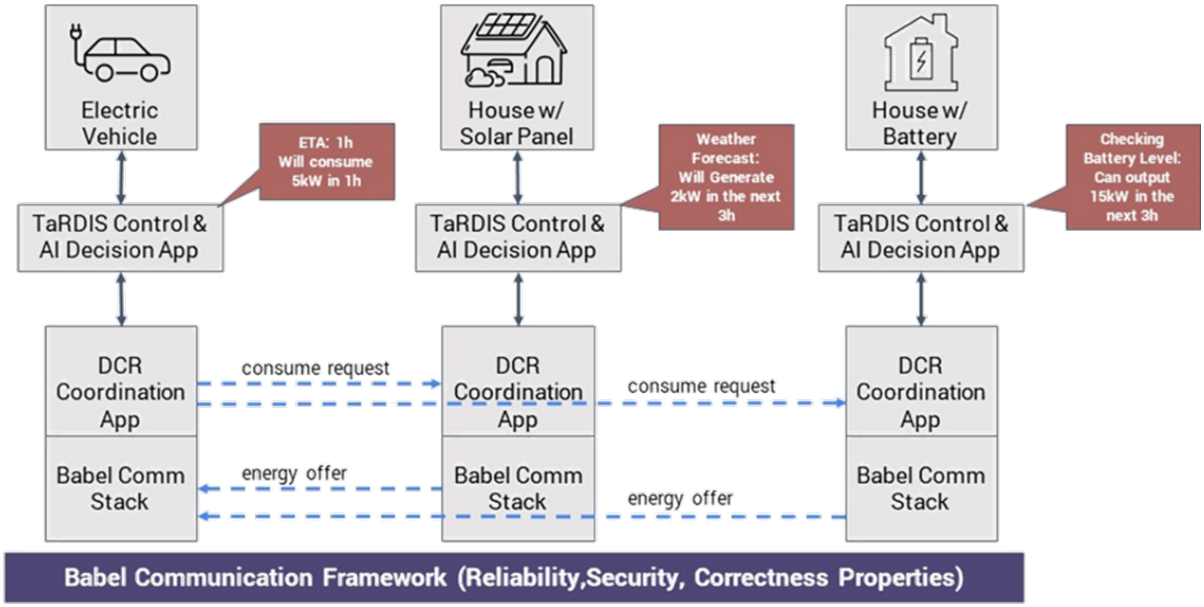

Figure 57: Multi-level Grid Balancing architecture.

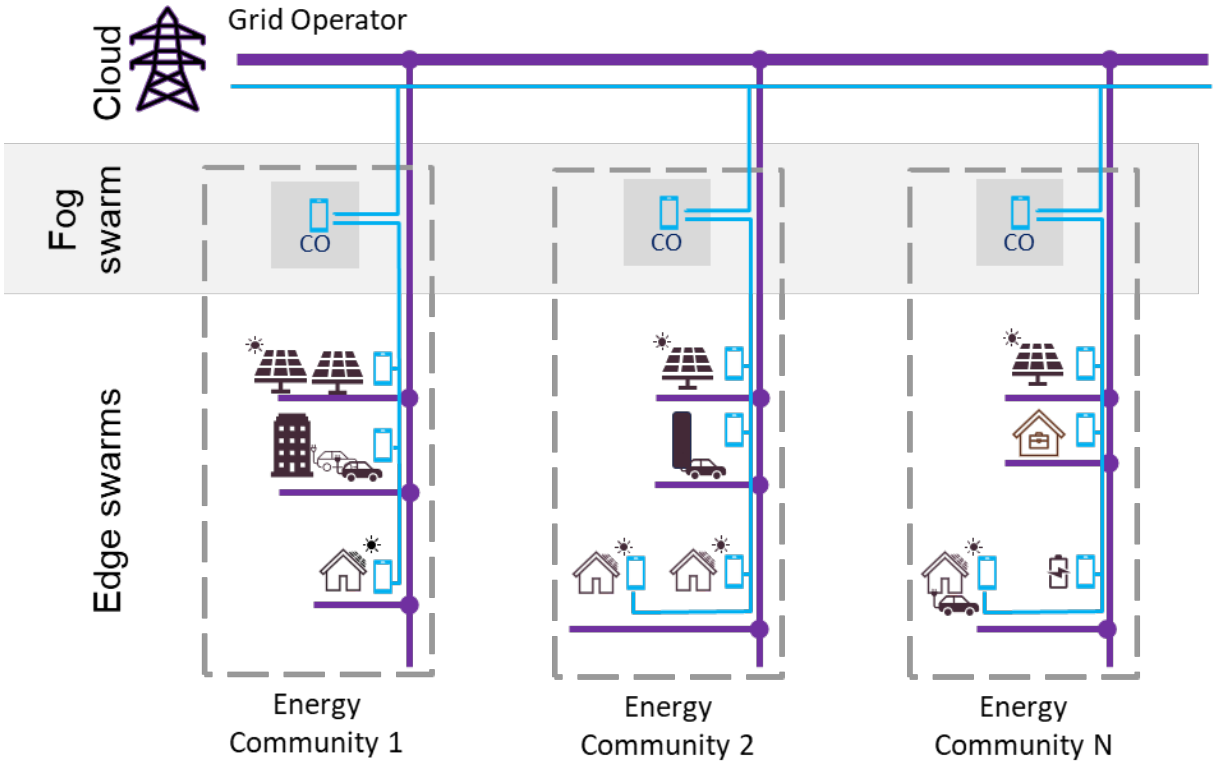

Figure 58: Multi-level Grid Balancing with TaRDIS (blue lines) architecture.

4.3.15.5 Verification

DCR Choreographies

- Static verification.
    - Provided compiler ensures DCR Choreographies are statically checked for correctness during translation to executables.
- Runtime verification:
    - DCR Editor provides an environment for running and testing DCR Choreographies, running each individual identity as Docker container, simulating a distributed environment. Throughout the execution, users can observe real-time logs in VS Code's integrated terminal, allowing for direct monitoring of message exchanges, event triggering, and overall choreography behaviour.

EDP Coordination Application

Verification does not play a role in this specific tool.

Fedra/Pruning Tool

Verification does not play a role in this specific tool.

Babel Stack

Static Verification

Not applicable.

Dynamic (Runtime) / Testing

Not applicable.

Dynamic (Runtime) / Monitoring

Babel supports a metrics collection system embedded into its Core. This module collects node-related metrics, namely both hardware and kernel related metrics (e.g., CPU usage time, memory usage, etc.). Moreover, it's also possible to collect custom metrics. For example, users may decide to store custom metrics such as protocol related metrics, or tracing information collected by the related tools watching over running applications.

All collected metrics are stored in a centralized place inside a specialized time series database, which can be later on retrieved to monitor or visualize the current execution status of the process.

IDE

This tool is mainly for performing development, although it can be foreseen that some of the TaRDIS verification tools may be in the future also integrated in the IDE, thus allowing the IDE

the capability of invoking them and hence allow verification and validation of swarm environments.

4.3.15.6 Machine Learning

DCR Choreographies

Machine learning does not play a role in this specific tool.

EDP Coordination Application

Initial learning

To understand which type of classifier should be used for the EDP coordination application, several algorithms such as decision trees, Random Forest, GradientBoosting and others were tested. Before testing the several algorithms, some steps were made first such as solving class imbalance, analysing feature quality, data preprocessing and also hyperparameter tuning. After doing all of these steps, then we trained the algorithms.

Models usage

They are called during an algorithm that will call every single tool that will be used in this use case. By using the forecasted data from the Fedra/Pruning tools then you can use these algorithms to calculate the type of choreography.

Continuous learning process

Not in this recipe.

Integration process

Not applicable to this recipe.

Fedra/Pruning Tool

Initial learning

The initial learning is based on a training python code that is integrated inside the Fedra tool and is application-specific (for the energy forecasting). For instance, we have also developed a DRL code for smart home energy management system that can also be integrated inside Fedra (or a developer can integrate his/her third-party base code for training an ML model for another application). Fedra initializes the FL training and monitors the process for the federated rounds (training and testing losses).

Models usage

The trained LSTM models are not used in this recipe.

Continuous learning process

There is not really a learning process associated to this recipe. However, a developer can re-run the Fedra tool with additional training data, for instance, having gathered additional data from the real smart home environment. To this end, by selecting the subset of the newly

observed data that have not been utilized during the previous training, the developer can perform a soft retraining/tuning of the pre-trained LSTM models.

Babel Stack

Machine learning does not play a role in this specific tool.

IDE

ML tools are not directly used in this tool, although the IDE may integrate AI/ML tools such as PTB-FLA or FEDRA, but these shall then be described separately on their own sections.

4.3.16 Flower-based FL tool - Anomaly detection of factory workflows

4.3.16.1 Scenario and Requirements

This scenario considers workflow anomaly detection (integrated to the Flower-based FL tool), based on smart factory data in settings where ground-truth labels are unavailable. While the tool was not integrated into a live factory or swarm system during the project, the tool was tested on synthetically generated ACT data (described in D5.3 and KPIs reported in D7.3). Here, we present a recipe that describes how it could be applied in a real swarm-oriented industrial scenario.

In such a setting, multiple machines, production lines, or factory units would form a logical swarm of autonomous data owners, each observing local workflow events. The identification of anomalous workflows at the level of individual swarm members can provide valuable insights into process deviations, contributing to improved automation, resilience, and reliability of industrial and logistics processes.

4.3.16.2 Activity Diagram and Software Development Life Cycle

The activity diagram illustrates the conceptual workflow for configuring, training, and evaluating an anomaly detection model using the Flower-based federated learning tool, which includes the TaRDIS tools T-WP5-01/02/03. The activity diagram should be understood as a conceptual representation of the tool-supported workflow, illustrating a hypothetical swarm-based learning process rather than a physically deployed system during the project. The tool is designed to be usable by both developers and non-expert users, as it does not require in-depth machine learning expertise to perform model training and inference. For simplicity, these roles are collectively referred to as: user.

No custom software development is required from the user side. Instead, the user incrementally configures the learning workflow through the tool interface. First, the learning task is selected; in this recipe, the task is workflow anomaly detection. Next, a dataset is selected for training. In the context of this recipe, simulated data derived from a TaRDIS use case provider are used for validation purposes. However, in a real-time swarm scenario, in a hypothetical deployment, it could operate on real data, on federated client nodes.

The user then decides whether to train a new model from scratch or to reuse a pre-trained model. Model selection can be performed manually, or it could be supported by the tool's model recommendation functionality, which requires a representative data sample to suggest suitable

models. Default hyperparameter values are provided for the selected model and can either be accepted or adjusted by the user.

The tool can also offer an optional client selection optimization mechanism, which can be enabled to weight and select federated clients according to predefined criteria, or disabled in Favor of a default configuration. In a hypothetical swarm-oriented scenario, each federated client conceptually represents a swarm agent (e.g., a machine or production unit) contributing locally trained model updates.

Once configuration is completed, the federated training process is initiated. Optionally, model compaction can be applied via knowledge distillation to obtain a more compact model. After training, the user selects and executes inference on a test dataset and inspects the resulting anomaly detection outcomes.

Although this recipe does not represent a deployed software development process, the workflow shown in the activity diagram resembles a waterfall-like life cycle (Figure 59), as it consists of clearly distinguishable and mostly sequential phases: task definition, model configuration, training, evaluation, and reuse. The resulting model can be reused or retrained with modified configurations in subsequent experimental iterations, which can be interpreted as a limited form of maintenance within an experimental setting.

Funded by the European Union

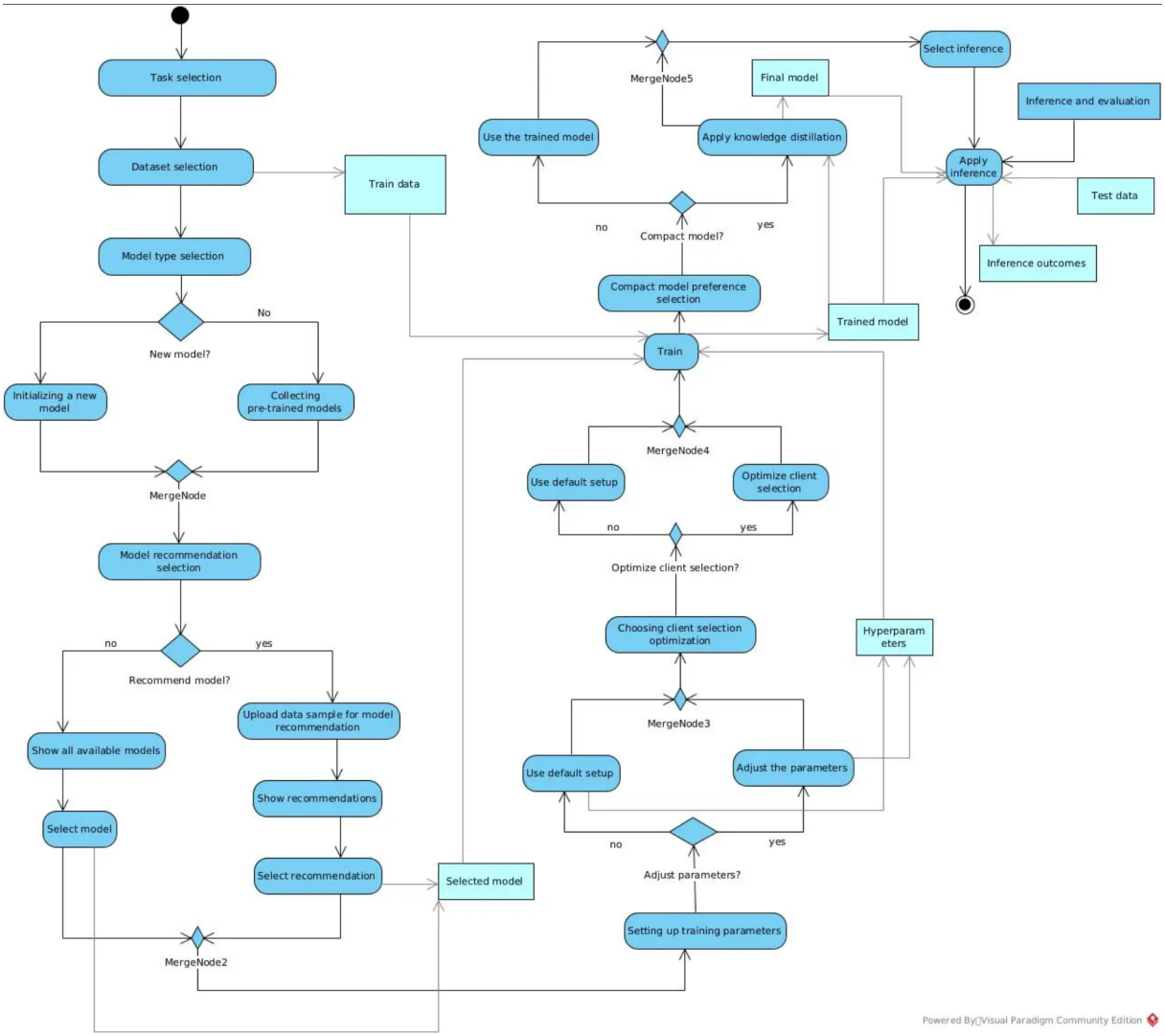

Figure 59: Configuring, training, and evaluating an anomaly detection model.

4.3.16.3 Architecture and Tools

The Flower-based federated learning (FL) tool represents a synthesis of the Flower-based FL model training tool (T-WP5-01), the data preparation tool for Flower-based FL model training (T-WP5-02), and the Flower-based FL model inference and evaluation tool (T-WP5-03), which are integral components of the TaRDIS architecture. Together, these tools provide a standalone federated machine learning solution within the TaRDIS toolbox.

The tool enables the configuration, training, inference, and evaluation of federated learning models implemented using the Flower framework. Among the supported learning tasks is anomaly detection, which is the focus of this recipe. In the context of the project, the tool was validated using simulated, event-based factory data and operated as an independent component, without integration into a live factory or swarm infrastructure.

From an architectural perspective, the Flower framework naturally supports swarm-oriented learning paradigms, where multiple federated clients collaboratively train a shared model under the coordination of a central aggregation server. In a hypothetical swarm-based industrial scenario, each federated client could conceptually represent a swarm agent (e.g., a machine,

production line, or factory unit), operating on local data, while the aggregation server would enable collective model updates without requiring centralized data collection.

Although the tool operates in a standalone manner, the resulting anomaly detection outcomes can support downstream analysis and decision-making processes, and the architecture is compatible with future integration into swarm-based or distributed industrial systems.

4.3.16.4 Verification

The Flower-based federated learning tool includes integrated facilities for data preparation, model training, inference, and evaluation, which collectively support verification activities within this recipe. Prior to model training or testing, datasets are pre-processed to ensure consistency and to prevent data-related runtime errors during experimental execution.

Verification was primarily performed in an offline and experimental setting, using simulated data. Model inference and evaluation are integral parts of the tool and enable the user to assess training stability, anomaly detection behavior, and model performance during and after the execution of federated learning experiments. These capabilities provide relevant insights into both the training and testing processes, without relying on runtime verification in a deployed system. In a hypothetical swarm-based deployment, the same mechanisms could support runtime monitoring and validation of anomaly detection outcomes across federated clients operating on real factory data.

4.3.16.5 Machine Learning

The Flower-based federated learning (FL) tool is an integral part of the TaRDIS toolbox and supports the application of various FL-based machine learning solutions on distributed datasets. In this recipe, the focus is on workflow anomaly detection, which was validated using simulated data derived from a TaRDIS use case provider (ACT) and relies on unsupervised learning, as ground-truth labels are typically unavailable in industrial settings.

One of the implemented approaches integrates two complementary machine learning methodologies: clustering using the K-means algorithm and representational learning through autoencoders, inspired by the HUNOD (Hybrid UNsupervised Outlier Detection) [30] method. This approach is designed to be robust to inexact or noisy labels, which are common in automatically generated factory workflow data. Learning is decentralized through the Flower framework, with global model updates performed using the FedAvg aggregation algorithm.

In addition to this approach, an alternative solution has been developed based on the Transformer model architecture, drawing on the principles of the Anomaly Transformer [31]. Owing to their ability to model long-range dependencies via attention mechanisms, Transformer-based models represent the state of the art in time-series anomaly detection. Within the federated learning context provided by the Flower framework, this architecture aims to enhance the accuracy and robustness of anomaly detection under heterogeneous and distributed data conditions.

Although model training and evaluation were conducted in an experimental setting, the federated learning paradigm enables iterative and potentially continuous learning. In a hypothetical swarm-based industrial deployment, models could be periodically updated as new

data become available at individual swarm agents, allowing the anomaly detection system to adapt over time without centralized data collection.

4.3.17 Early-Exit and D-Exit tool - Decentralized Early-Exit of Inference Deployment in Swarm Systems

4.3.17.1 Scenario and Requirements

This scenario describes how the early-exit (EE) tool and the decentralised early-exit of inference framework for swarm systems (D-Exit) tool can be utilized by a swarm developer or an ML operator to train and deploy a lightweight ML model in a distributed swarm system considering the resource-constrained edge devices.

EE and D-Exit use a combination of distributed computing, early-exit strategies, and peer-to-peer networking between the node participants. The EE and D-Exit tools were not integrated in any of the use cases during the project, since additional techniques for lightweight inference have been developed and showcased (pruning, knowledge distillation). Therefore, EE and D-Exit were demonstrated in simulated scenarios with varying topologies, configurations and number of nodes and we herein present a recipe that describes how it could be applied in a real-world swarm scenario.

Following the early-exit of inference concept, we split a DNN model in individual parts, each part containing one early-exit. The deployment of the different parts of the model according to the early-exit splitting can be conducted directly at the swarm nodes. Therefore, the same copy of an early-exit model part can be hosted in multiple swarm nodes for redundancy purposes, as well as for minimizing the inference latency and for sharing the computational burden of the processing. To this end, the energy capacity and resources of specific swarm members (e.g., battery-powered) are not individually reduced, enabling the energy efficiency and the resilience of the swarm system.

4.3.17.2 Activity Diagram

The combination of the EE and D-Exit tools is designed to be quite user-friendly, abstracting most of the required development operations from the user, as well as keeping simple the required knowledge related to ML models. The workflow of this recipe (also depicted in Figure 60) can be described as follows:

- The Developer/ML developer defines the configuration parameters of the EE model training, i.e., the splitting strategy of the original model (number of early exits, hidden layers/model part), the training dataset, the confidence intervals of each exit and also provides the original model.

- Once the configuration is completed by the developer/ML engineer, the training of the EE model is initiated from the IDE and the user receives the trained model variant, as well as is notified about the achieved specifications of the resulting model.

- The developer/ML developer defines the configuration parameters of the D-Exit tool, providing the trained EE model from the previous step, the swarm topology (number of peers that will host the first model part, the intermediate model part and the final model

part) and status (number of peers required to initialize the D-Exit framework) and the peer connection strategy (e.g., random, closest, etc.).

- The D-Exit configuration is used as input to initiate the D-Exit tool, where the relevant model parts are deployed directly to the swarm peers, ensuring the connection between the hierarchically higher layers to complete the overall inference process.
- The pretrained EE model parts are ready and can be used for online inference in the swarm.

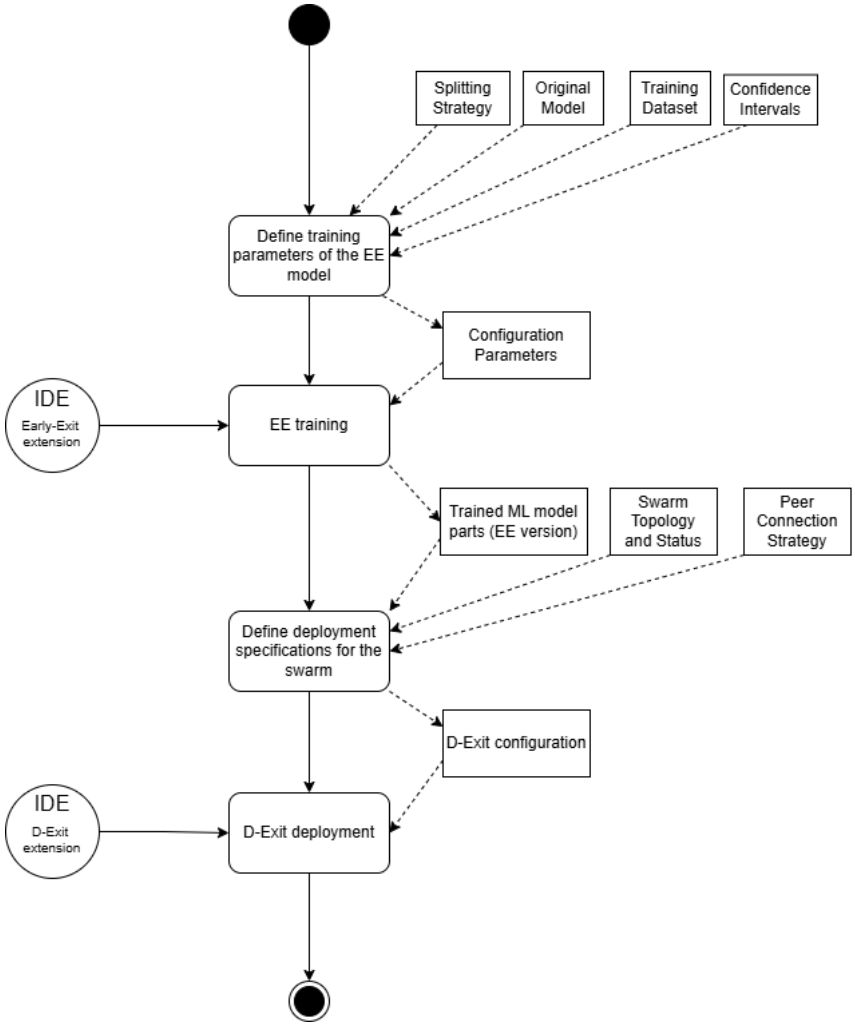

Figure 60: Early-exit model training and decentralized early-exit of inference deployment in swarm systems activity diagram.

4.3.17.3 Software Development Life Cycle

The described procedure is not strictly a software development process, being more relevant to MLOps, since the target is to train a lightweight EE model and deploy it in the decentralized swarm system, to be then used for online inference. The workflow described in the activity diagram corresponds to a waterfall-like development life cycle, where the configurational model split and the training parameters are initially defined to train the lightweight model. Once the EE version of the model is prepared, the ML developer uses the D-exit framework to deploy

the ML model in the swarm, considering the swarm topology and configuring the peer connection strategy. The ML model and the deployment strategy can be reused or later modified for different environmental conditions and experimental scenarios, which is considered a form of maintenance.

4.3.17.4 Architecture and Tools

Initially, the EE tool is used to train a DNN model with multiple early exits or model branches, in order to save computational resources and minimize the inference latency. The D-Exit tool enables the configuration and the deployment of the pieces (the early-exit parts) of an ML model in a swarm system, to be used for inference. D-Exit is model-agnostic, and it has been tested in simulated environments with a small number of swarms. Although the tool operates in a standalone manner, it can be integrated with multiple use cases that use ML models for inference at the edge.

From an architectural perspective, D-Exit includes the following elements (see Figure 61):

- **IoT/End Device**: a resource-constrained node that host the first model part and serves as the entry point for inference requests initiated by an end-user (in case of a smartphone) or by the functional operation of the device (e.g., IoT sensor, camera, etc.).
- **Edge Node**: Hosts a part of the model that requires increased computational complexity and receives the inference requests conveyed by the IoT/End Device.
- **Cloud Node**: The Cloud Node represents the final hierarchical layer of the D-Exit framework, exhibiting increased computational capacity in terms of resources.
- **Network layer**: Forms the communication backbone of the D-Exit framework, including peer discovery and efficient routing protocols.
- **Network State Management**: Acts as a memory of the swarm system, tracking the statuses of the peers that participate in the D-Exit framework, having an overview of the overall swarm topology.

Funded by the European Union

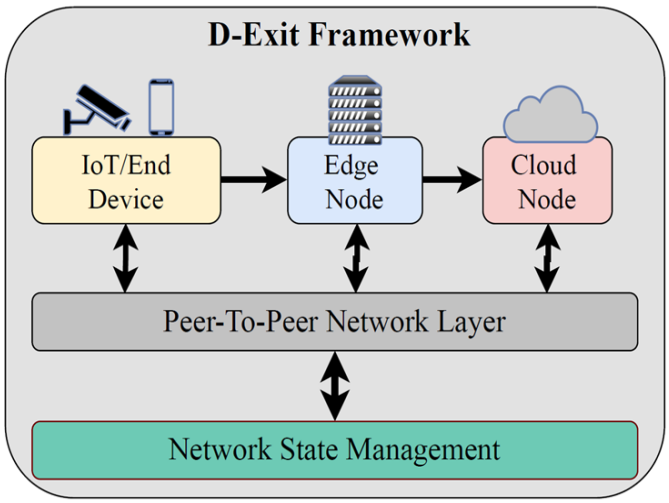

Figure 61: D-Exit framework architecture, illustrating the internal modules and their interconnection.

Regarding the implementation in a swarm system, the same copy of the early-exit model part is hosted in multiple swarm nodes for redundancy. Moreover, the swarm nodes share the computational burden of the inference processing and different numbers of hidden layers are included in each device and in each exit branch. Since the swarm system may be heterogeneous in terms of computational resources, the nodes with increased capacity assume the functional role of cloud nodes and the nodes with limited capabilities and strict energy constraints host the first part of the model, assuming the role of IoT/End device. For purposes of balancing the computational load within the swarm system, we use C < E < I (see Figure 62).

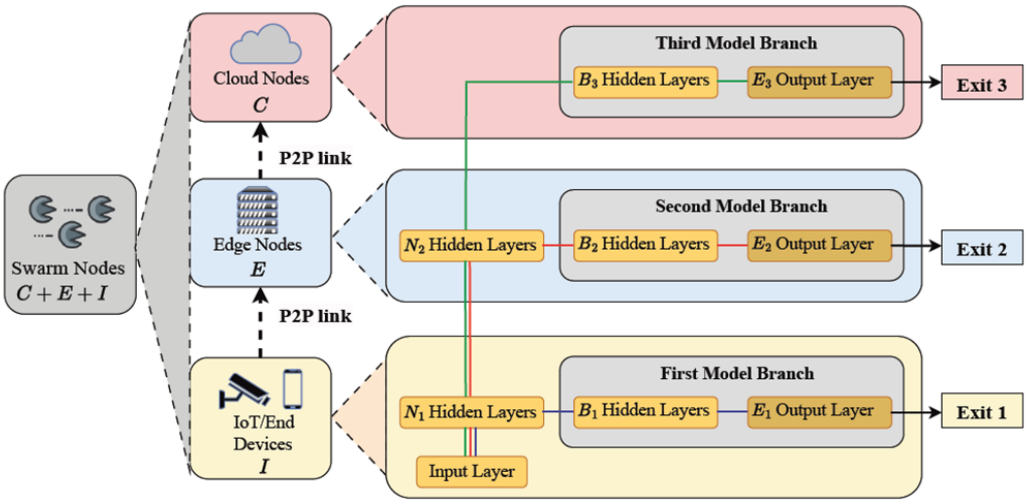

Figure 62: Computational load balancing within the swarm system illustrating the lightweight model split across the different hierarchical layers.

4.3.17.5 Verification

Verification does not play a role in this specific recipe.

4.3.17.6 Machine Learning

The EE and D-Exit tools are an individual part of the TaRDIS toolbox that can be used for the training and deployment of pre-trained ML models in swarm systems. The tools abstract from the developer all the relevant information related to swarm membership, peer discovery mechanisms and communication protocols between peers, enabling him to effortlessly train, deploy the pre-trained model and use it for online inference, while at the same time balancing the trade-off between the overall model accuracy, the computational savings and the inference latency.

It should be noted that the model needs to be pre-trained in its early-exit format before it can be deployed for inference. We have numerically evaluated the EE and D-Exit framework, using a representative subset of the CIFAR- 10 dataset, consisting of 32×32×3 RGB images across 10 object classes and as base model architecture we have used the Visual Geometry Group (VGG-16) that includes 138M parameters, 13 convolutional layers and 3 fully connected layers [32]. To this end, three versions of the VGG model were created using the EE tool, based on the splitting strategy: (i) Front-weighted (50/30/20%) distribution per exit, strengthening the IoT/End device model parts; (ii) Rear-weighted (35/25/40%) distribution per exit, representing increased computation capabilities of Cloud nodes; (iii) Evenly-weighted (33/34/33%) distribution per exit, constituting a fair policy of sharing the computational load among the swarm peers.

The pre-trained model variants can be then deployed in a swarm system with D-Exit. A topology with 8 IoT/End Devices, 4 Edge nodes and 2 Cloud nodes has been configured, hosting the corresponding early-exit model splits in a 2D space of 20×20 m^2. The developer can also optionally select between different peer connection strategies. In the evaluation of our simulations scenarios, we tested: (i) *Closest strategy*, where the peers send their feedforward data to their closest node that contains the higher-layer model split; (ii) *First-available strategy*, where the peers forward the inference data to the higher-layer node that is available, i.e., currently not executing additional operations; (iii) *Random strategy*, where the peers choose randomly the next-layer node in case of low confidence results.

4.3.18 TID Use Case Recipes

4.3.18.1 Overview

This overview of the TID use case recipes includes 3 main parts:

- **Individual Tools.** The first part details each of the tools used in the use case, focusing on its requirements (i.e., input information) and its functionality (i.e., expected output). It also presented an architecture overview of each tool, information regarding their verification guarantees, and information regarding their ML components. These are covered from section 4.3.19 to section 4.3.24.

- **Combining Tools.** The second part focuses on the combination of different tools to achieve more complex results. This composes a detailed description of their interaction, the respective input and output of each tool and overall architecture. These are covered from section 4.3.25 to section 4.3.28.

Funded by the European Union

- **Use Case Tools.** Finally, the third and final part encompasses all of the tools presented above to describe the execution of one or more use-case scenarios.

4.3.18.2 Activity Diagram

Our use case development methodology provides a structured framework for developing distributed and privacy-enhanced AI applications on top of the FLaaS platform and supported by the TaRDIS toolkit. The methodology adopts a modular, top-down approach to ensure correctness-by-construction and integration of advanced machine learning and system components. It starts from legacy implementations and iteratively evolves the code through multiple stages, ultimately producing a fully functional, decentralized, and verified federated learning application.

The development process is primarily carried out by the developer, supported at specific stages by the ML developer and the formal verification (FV) engineer. While the ML developer is engaged early in training and validating machine learning models, the FV engineer plays a critical role during the final verification and KPI evaluation of the system.

First, we focus on the development activities performed by the developer, which are illustrated in the "App Development and Testing" activity diagram (Figure 63).

The process begins with updating the FLaaS code and its dependencies. At this stage, the developer refactors the legacy codebase, integrates necessary updates, and prepares the environment for incorporating advanced functionalities. Once the foundation is set, differential privacy mechanisms are added. This involves generating the differential privacy (DP) code, integrating either central or local DP techniques, and ensuring that the model training and update mechanisms adhere to defined privacy constraints. Evaluation metrics are considered to verify the effectiveness of the DP mechanisms.

Following the privacy integration, the developer generates the split learning (SL) code. In this phase, the ML model is partitioned to support collaborative training across distributed nodes, where only intermediate representations are shared between the client and server. This facilitates a more privacy-preserving and resource-efficient training process. The next phase involves generating the helper code. Here, the helper component, typically containerized using Docker, is equipped with asynchronous execution capabilities using FastAPI, Uvicorn, and threading utilities. This helper acts as a relay or mediator in split learning, executing partial models and returning the results to the coordinating components.

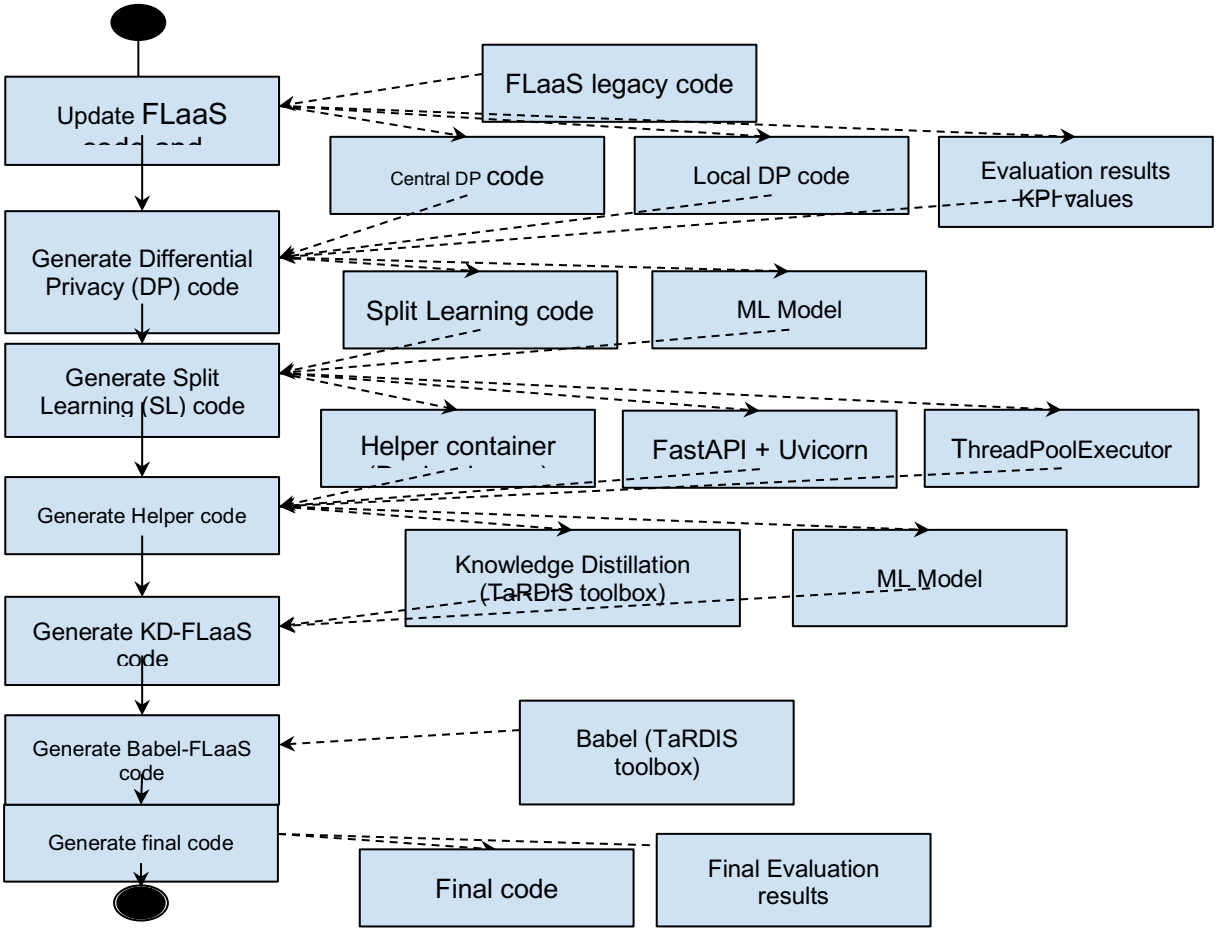

Figure 63: App development and testing (performed by developer).

After the helper infrastructure is in place, knowledge distillation (KD) is incorporated. This step involves generating the KD-FLaaS code, where the TaRDIS toolbox is used to support teacher-student learning pipelines. The knowledge distillation process allows lightweight models to be trained on edge devices by transferring knowledge from larger, more complex models, thus reducing computational load and communication overhead without sacrificing performance.

The penultimate phase of the methodology is the generation of the Babel-FLaaS code. Babel, as part of the TaRDIS toolkit, provides decentralized system functionality. In this phase, the updated FLaaS code is extended with Babel services to support decentralized deployment across a computer network. This integration enables the application to scale beyond a single host and operate as a dynamic swarm of coordinated nodes.

Finally, the developer generates the final use case code. This phase involves comprehensive testing and evaluation. The code is validated for correctness, performance, and compliance with all key performance indicators (KPIs). The result of the methodology is a production-ready, fully evaluated, and privacy-enhanced distributed learning application, accompanied by a complete test and evaluation report.

This methodology, supported throughout by the TaRDIS toolkit and governed by a role-aware activity structure, ensures that complex AI use cases can be systematically developed, extended, and deployed with strong guarantees of correctness and performance.

Once the final code has been generated using the FLaaS-based use case development methodology, the training phase begins. This phase is primarily conducted by the end user and focuses on executing the FL workflow to produce a performant and privacy-preserving machine learning model. It is structured to ensure modularity, flexibility, and reproducibility, and it supports advanced features such as Differential Privacy (DP), Split Learning (SL), Helper-based orchestration, and Knowledge Distillation (KD), as previously integrated into the FLaaS infrastructure.

The training process (Figure 64) starts with the creation of a FLaaS project. During this step, the user selects from a set of configurable modules including DP, SL, Helper, and KD options. Additionally, a machine learning model and a pre-processed dataset are loaded into the system to define the initial training environment.

Following project creation, the user proceeds to define the training parameters. These parameters include the number of clients participating in the training, the number of training rounds to be executed, and the number of applications (apps) to be instantiated. These settings are essential for configuring the federated learning environment, balancing workload across nodes, and controlling the scope of training iterations.

Once the training parameters are defined, the system sets up and initializes all clients. This initialization step ensures that each client device or simulation instance is configured with its respective data partition, model segment (if using SL), and security or privacy parameters.

Training is then started. The server initiates the training round by distributing the current global model weights to all clients. Each client performs local training on its respective dataset using the received model weights. The result of this computation is a set of local gradient updates or model updates.

Clients then send their computed updates back to the server. The server aggregates these updates using a defined aggregation method, which may incorporate privacy-preserving mechanisms, robustness techniques, or adaptive weighting strategies depending on the use case.

After aggregation, the server evaluates whether the termination condition has been met. This condition is typically based on the number of completed rounds. If the condition is not met, training continues with another round starting from the new global model.

This cycle of distribution, local training, update collection, and aggregation is repeated until the termination condition is satisfied. At that point, training is finalized and the system outputs the final model, which represents the fully trained version of the application-specific machine learning model under the chosen FLaaS configuration.

The training workflow ensures a fully decentralized, privacy-aware, and modular federated learning process that adheres to the specifications and architecture established during the development methodology. This clear separation between development and training phases enables reuse of the same FLaaS infrastructure across different scenarios and model configurations while ensuring transparency and traceability at each step.

Funded by the European Union

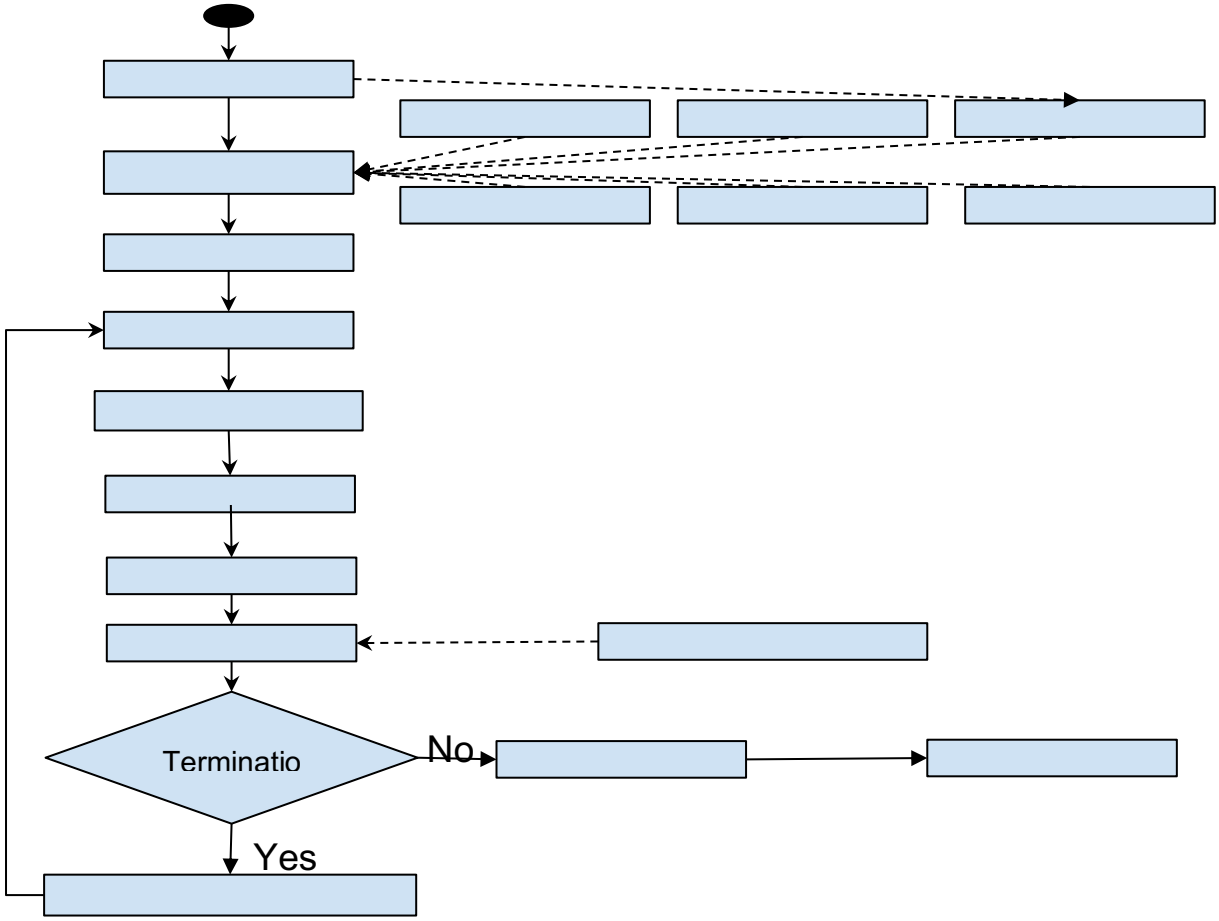

Figure 64: Training (performed by user).

4.3.18.3 Software Development Life Cycle

These recipes follow an iterative waterfall development process.

4.3.19 Federated Learning as a Service (FLaaS)

4.3.19.1 Scenario and Requirements

Federated Learning as a Service (FLaaS) is a Federated Learning (FL) framework in mobile environments, enabling both cross-device and cross-app FL. In the context of the TID use case, FLaaS enables distributed training of smart home applications (e.g., smartphones, personal assistants, smart TV) through an intuitive and practical interface.

4.3.19.2 Activity Diagram

See section 4.3.18.2.

4.3.19.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.19.4 Architecture and Tools

FLaaS is a standalone TaRDIS tool comprising 4 components. The FLaaS architecture is depicted in Figure 65, and the main components are described in the following:

1. **App developer interface:** consists of an admin interface where the developer can bootstrap, reconfigure, execute and monitor FL projects. This component also exposes high-level APIs for FL workflow (task definition, client selection, hyperparameter tuning).

2. **FLaaS server:** consists of the cloud-hosted service responsible for orchestrating and monitoring FL processes.

3. **Notification Service:** is responsible for pushing communications to client devices through silent push notifications related to the system activity

4. **Client devices:** consists of Android-based devices responsible for executing local ML training/inference and communicating model deltas back to the server via secure channels.

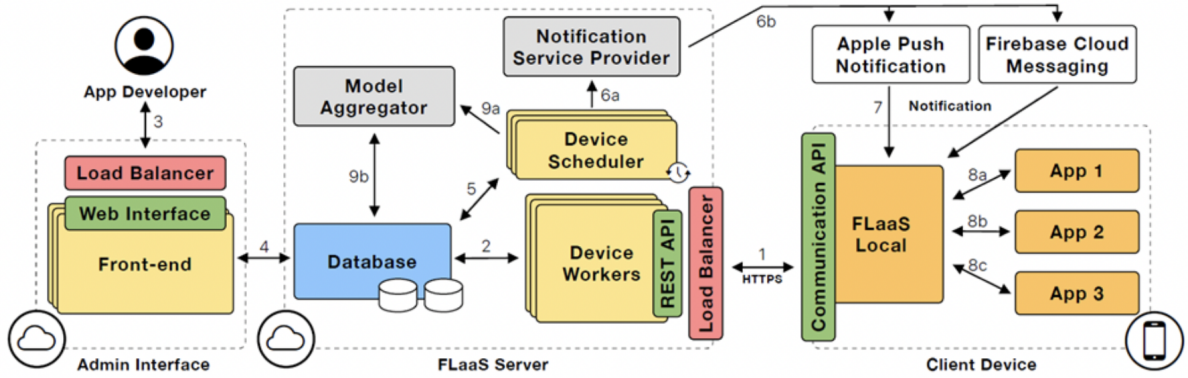

Figure 65: FLaaS architecture.

4.3.19.5 Verification

Static verification is performed by checking FL project descriptors, hyperparameter bounds, and model-delta schemas for consistency before any training begins. At runtime, FLaaS emits per-round metrics (client participation rates, round durations, delta sizes and potential error codes) into the monitoring dashboard, where configurable alerts flag anomalies.

4.3.19.6 Machine Learning

Machine Learning (ML) models are at the core of the working process of FLaaS. Similar to other FL frameworks, ML models are continuously trained in FLaaS in a federated manner, computing updates at the distributed devices and aggregating model weights at the central server. Additionally, developers can bootstrap pre-trained models (e.g., CNNs for object detection) or define new models.

Hyperparameters such as number of rounds, client fraction, local epochs, batch size, and optimiser settings (learning rate, momentum) are configurable via the interface. As described in the architecture above, the server orchestrates periodic training rounds, triggering clients via the Notification Service when they meet resource and connectivity criteria.

4.3.20 Differential Privacy

4.3.20.1 Scenario and Requirements

While FL protects privacy by keeping user data on local devices, it remains vulnerable to gradient inversion attacks, where an adversarial (yet protocol-compliant) server attempts to reconstruct training data from received gradients. This recipe focuses on mitigating such risks using Differential Privacy (DP). Specifically, it targets the "honest-but-curious" server scenario, where the server aggregates gradients but attempts to infer individual client data. This recipe explores two complementary DP approaches: central DP (server-side) and local DP (client-side). Central DP seeks to balance privacy and utility, while local DP prioritises stronger privacy at the cost of model accuracy.

4.3.20.2 Activity Diagram

See section 4.3.18.2.

4.3.20.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.20.4 Architecture and Tools

This recipe integrates both central and local DP mechanisms in a federated learning system, using the TaRDIS toolkit alongside the Flower.ai framework.

Central DP (server side):

Clipping:

Each client's model update is clipped by an L2 norm threshold SSS. This limits any single client's influence on the global model. A static clipping threshold is used across all rounds.

Noise Addition:

Gaussian noise $N(0, \sigma^2)$ is added to the sum of the clipped updates. The noise scale is computed as: σ = (noise_sigma * S) / (number of sampled clients)

Local DP (client side):

Each client independently perturbs their update with Gaussian noise before uploading. This avoids reliance on a trusted aggregator but results in higher utility degradation. For an (ϵ,δ)-DP guarantee with sensitivity Δ, the noise scale σ is chosen accordingly to satisfy the Gaussian mechanism.

4.3.20.5 Verification

Runtime Monitoring:

Privacy accounting tools monitor the cumulative privacy budget (ϵ) throughout training. Logs include clipping ratios, update norms, and effective noise scale per round.

Testing:

A suite of unit and integration tests ensures the clipping and noise addition steps operate correctly. Differential privacy guarantees are validated via analytical privacy accounts.

Static Verification:

Static analysis checks ensure the clipping bounds and DP parameters (e.g., σ, ϵ, δ) are correctly initialised and never exceed user-defined limits.

4.3.20.6 Machine Learning

This recipe builds on standard FL pipelines:

Initial model training:

A pre-trained model is deployed to all clients. Clients perform local training on their private data and send updates to the server (with or without local DP).

Global Aggregation:

The server aggregates clipped and noised updates using weighted averaging.

Continuous Learning: The system trains continuously across multiple FL rounds. At each round, the privacy tracker updates the cumulative privacy loss and reports model utility and other metrics.

Model Evaluation:

After training, the model is evaluated on a held-out global test set to assess the impact of DP on accuracy.

4.3.21 Split Learning

4.3.21.1 Scenario and Requirements

Split Learning (SL) partitions Neural Networks (NNs) at a designated “cut” layer, executing the initial layers on client devices and the remaining layers on a server, exchanging only intermediate activations to preserve data privacy and minimise on-device computation. In a nutshell, SL splits the NN model into parts and allows clients (devices) to offload the largest part as a processing task to a computationally powerful helper.

In the context of TID smart-home use-cases, enabling SL allows devices with limited computational resources to participate in the training by offloading part of the computations to the FLaaS Server (thus the addition of the server training and ML library at the server).

4.3.21.2 Activity Diagram

See section 4.3.18.2.

4.3.21.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.21.4 Architecture and Tools

The system architecture in SL consists of a client-side component responsible for executing the initial layers of a neural network and a server-side component that processes the deeper layers. Communication between the two entities involves the client transmitting the intermediate activations at the cut layer, along with the corresponding labels, to the server. The server then completes the forward pass, computes the loss, and initiates the backwards pass.

During backpropagation, the server returns the partial gradients corresponding to the cut layer back to the client. This coordinated process enables synchronous, step-wise training, where the client manages data ingestion and the initial forward pass, while the server handles the remaining forward and backward computations.

4.3.21.5 Verification

Static Verification:

- Ensures the model is correctly split at a valid cut layer and both parts are initialised with matching weight shapes.
- Configuration validation ensures correct alignment of optimiser settings and learning rates across both sides.

Runtime Verification:

Monitoring:

- Logs activation sizes and loss/accuracy at both ends.
- Verifies gradient transmission to avoid shape mismatches or silent failure.

Testing:

- Correct gradient flow across cut layers
- Loss consistency between split and end-to-end models

4.3.21.6 Machine Learning

The ML process in SL follows a distributed approach to training a single model. An initial NN architecture is selected and partitioned into client-side and server-side sub-models. Both parts are initialized with appropriate parameters, and synchronized training begins. The client performs the forward pass on its local data, up to the cut layer, and sends activations to the

server. The server completes the forward computation, calculates the loss, conducts backpropagation through its layers, and sends the gradient of the cut layer back to the client.

The client then completes the backward pass on its side and updates its local weights. This collaborative procedure is repeated across many training epochs. Continuous learning is supported by iteratively improving the full model across clients and over time. The final model can be reassembled and deployed as a complete network or remain in its split form for real-time inference in constrained environments.

4.3.22 Knowledge Distillation

4.3.22.1 Scenario and Requirements

Knowledge Distillation (KD) is a model compression technique that transfers knowledge from a large, high-capacity model (referred to as the *teacher*) to a smaller, more efficient model (the *student*). The goal is to maintain much of the predictive performance of the teacher while reducing the model's computational complexity, size, and inference cost. In the context of TID's smart-home use case, KD is essential for enabling on-device inference in environments where computational resources, memory, and communication bandwidth are limited. The distilled model can be deployed on edge devices such as sensors, IoT hubs, or embedded controllers, supporting real-time decision-making without the need for constant server interaction. This aligns with the requirements of energy efficiency, low-latency operation, and minimal data transmission in smart-home scenarios.

4.3.22.2 Activity Diagram

See section 4.3.18.2.

4.3.22.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.22.4 Architecture and Tools

The architecture of the KD module consists of two main components (Figure 66): the *teacher model* and the *student model*. The teacher is a pre-trained, high-capacity model that performs well on the target task but is too heavy for direct deployment on edge devices. The student is a smaller neural network trained to mimic the behaviour of the teacher.

During training, both models process the same input data. The student learns not only from the ground truth labels (as in standard supervised learning), but also from the soft targets generated by the teacher model, typically the logits or softened probability distributions from the final layer. This dual supervision enables the student to capture richer information, including inter-class relationships and decision boundaries encoded in the teacher's output. The overall training objective is a weighted combination of the standard classification loss and the distillation loss (e.g., Kullback–Leibler divergence between teacher and student outputs).

Funded by the European Union

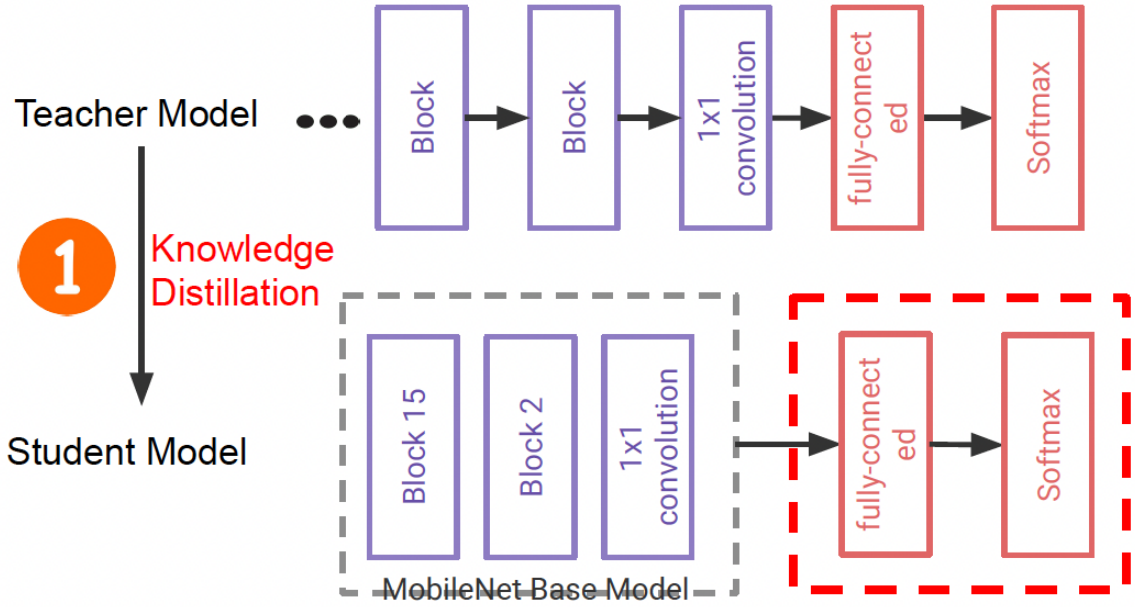

Figure 66: KD architecture.

4.3.22.5 Verification

Verification in KD focuses on ensuring functional correctness and performance fidelity between the teacher and the distilled student model. Static checks verify that the student model has a valid architecture and compatible output dimensions for distillation. Runtime monitoring includes tracking the distillation loss, student model accuracy, and divergence from the teacher's predictions. Evaluation is conducted across both in-distribution and out-of-distribution samples to ensure robustness and generalisation.

Testing procedures include performance benchmarking to confirm that the student meets resource constraints such as memory usage, inference time, and model size. Validation suites are used to compare accuracy before and after distillation.

4.3.22.6 Machine Learning

The ML workflow begins with a high-capacity teacher model that is either pre-trained or obtained from a prior ML process. The student model is initialised with a smaller architecture and trained using a dataset shared with or similar to that of the teacher. During training, the student receives supervision from both the ground truth labels and the teacher's outputs, using a composite loss function that balances classification and distillation losses.

Once training is complete, the student model is evaluated and validated independently. If performance targets are met, the student is exported for deployment on smart-home devices. In many cases, this training process is one-off, but it may also be adapted into a continuous learning setup where the teacher is periodically updated, and the student is retrained to capture new knowledge. In systems with multiple devices or evolving environments, a lightweight online update mechanism may also be incorporated.

4.3.23 Babel - A Layer of communication between entities

Babel is described in detail in Section 4.3.11. The same architecture, verification approach, and tooling apply to this recipe. Machine learning does not play a role in this specific recipe.

4.3.24 IDE - How to deploy a project with the different tools available

4.3.24.1 Scenario and Requirements

This tool is present within every single test use case scenario established, so this recipe and its description apply to every test scenario.

The primary objective behind the development of TaRDIS was to simplify how the developers interact with distributed systems by automating intricate configurations and providing user-friendly workflows. Building from that starting point, the consortium developed extensions for existing best-of-breed IDEs to be versatile and adapt its features to suit their specific project requirements. The initial extension was built for the Eclipse IDE, but after a survey that was posed to the developer community, the path switched into the customisation of the Visual Studio Code (VS Code) IDE, currently the most popular especially in the new generations of developers. The objective then is to make an extension to this IDE in order to customise it for the development of swarm environments using TaRDIS.

To achieve this, the proposed extension supports multiple project types: Babel-based Java projects, PTB-FLA projects for federated learning, and DCR choreographies for distributed execution. It ensures seamless integration with essential tools such as Maven for Java projects, Python virtual environments for PTB-FLA, and Docker for running DCR-based prosumers. This focus on usability, flexibility and cross-platform compatibility was key to creating an extension that enhances productivity while maintaining robust functionality.

For this particular use-case, the proposed scenario to build the TID environment is to:

- Create a base TID workspace in VS Code, a common repository where all the project elements will be present.
- Invoke, from the IDE, each of the tools, such as Babel, that will be seamlessly integrated, and perform the development actions that are stated in each of the tools descriptions in this document.

4.3.24.2 Activity Diagram

See section 4.3.18.2.

4.3.24.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.24.4 Architecture and Tools

The TaRDIS Visual Studio Code extension is a comprehensive toolbox designed to streamline the development and management of distributed systems and projects. The extension

provides a range of functionalities, including project creation, protocol importing, and class generation, to help developers quickly and efficiently set up and manage their projects. This manual outlines how to use the TaRDIS extension and its current features.

For a detailed description of the tools and frameworks used in the TaRDIS IDE extension (Node.js, TypeScript, VS Code API, Webview API, Maven, Docker, Yeoman, etc.), see Section 4.4.

Tools required for operation:

To ensure seamless operation, the following tools are required:

- **Java Development Kit (JDK):** Java 21 is essential for Babel-based projects, providing the runtime environment for Java applications.
- **Node.js:** Serves as the runtime environment for the extension itself, managing dependencies and executing commands.
- **Maven:** Used for dependency management and build automation in Java projects.
- **Python:** Required for PTB-FLA projects, with virtual environments ensuring package isolation.
- **Docker:** Facilitates containerized execution of DCR choreographies.
- **Visual Studio Code:** Serves as the primary development environment.
- **Git:** Enables version control, code sharing, and collaborative development.

4.3.24.5 Verification

This tool is mainly for performing development, although it can be foreseen that some of the TaRDIS verification tools may be in the future also integrated in the IDE, thus allowing the IDE the capability of invoking them and hence allow verification and validation of swarm environments.

4.3.24.6 Machine Learning

ML tools are not directly used in this recipe, although the IDE may integrate AI/ML tools such as PTB-FLA or FEDRA, but these shall then be described separately on their own sections.

4.3.25 Differential Privacy Integration and Coordination with FLaaS

4.3.25.1 Scenario and Requirements

This recipe addresses the integration of differential privacy (DP) into a baseline federated learning scenario by embedding privacy-preserving mechanisms directly into both the **FLaaS Server** and the **FLaaS Local App**. The objective is to enable privacy-preserving training in cases where the central server may not be fully trusted. To mitigate gradient inversion and

data reconstruction risks, this recipe introduces **global differential privacy** (applied at the server level) and **local differential privacy** (applied at the client level).

4.3.25.2 Activity Diagram

See section 4.3.18.2.

4.3.25.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.25.4 Architecture and Tools

The architecture extends the FLaaS (Federated Learning as a Service) system with modular differential privacy (DP) components that ensure privacy preservation at both the server and client levels.

FLaaS Server Integrates **Central Differential Privacy** by performing clipping of incoming client updates and adding calibrated Gaussian noise to the aggregated sum. It is built on **Django**, which serves as the backend control plane for managing privacy configurations and orchestrating the FL workflow.

The **Django admin interface and backend services** (Figure 67) handle: Selection of the **DP type**: either **Central DP** (applied on the server) or **Local DP** (applied on the client); Configuration and distribution of **privacy parameters** (ϵ,δ) for both Central and Local DP; Secure delivery of these configurations to clients through RESTful APIs or secure messaging services.

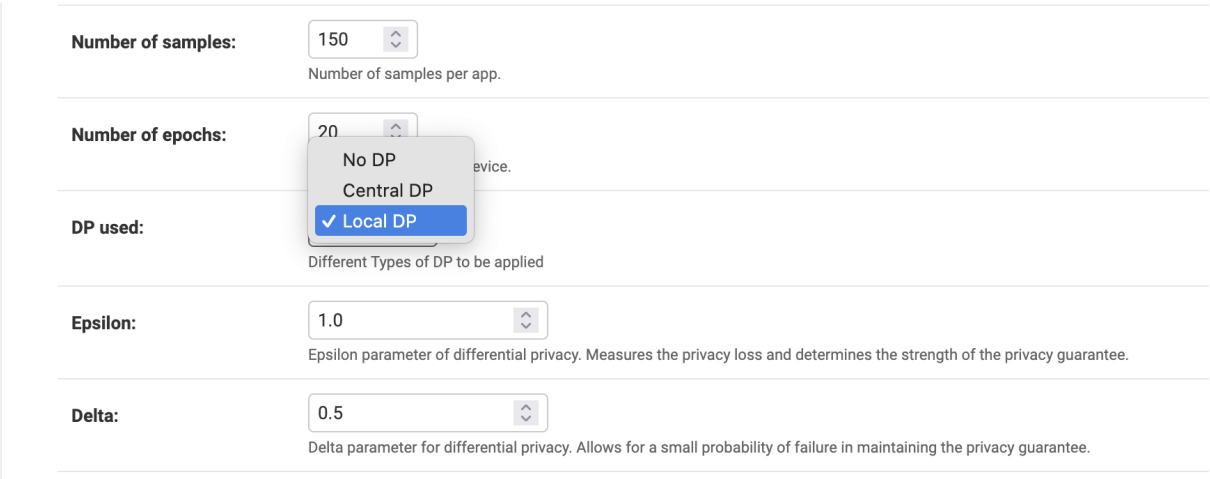

Figure 67: Django admin interface and backend services.

FLaaS Local App: The client application supports **Local Differential Privacy (Local DP)** by adding Gaussian noise to local model updates before they are sent to the server. This ensures privacy even in the presence of an untrusted server. DP parameters (ϵ,δ) are received from the Django server during the FL setup phase.

FLaaS Privacy Module: A reusable DP code integrated into both client and server environments.

Toolkits Used:

TaRDIS Toolkit and IDE for system orchestration, policy configuration, and verification. **Opacus** DP library for efficient, scalable DP noise application.

4.3.25.5 Verification

Static Verification:

- Configuration checks using TaRDIS IDE ensure DP parameters are within acceptable bounds.
- Static analysis tools verify that clipping is applied before noise and that privacy accounting logic is correctly wired.

Runtime Verification:

- **Monitoring**: FLaaS Privacy Module emits logs and metrics on update clipping ratio, privacy budget consumption, and client-level DP adherence.
- **Testing**: Unit tests ensure the correct calculation of noise scale and clipping behaviour.

4.3.25.6 Machine Learning

Initial Learning:

A base model (MobileNet) is initialised on the FLaaS Server and distributed to all participating clients. Alongside the model, the server also dispatches the differential privacy configuration, specifying whether **Local DP** or **Central DP** is to be used, and providing the associated parameters (ϵ,δ).

Local Training:

Each client trains the model locally on its private dataset using standard ML techniques. If **Local DP** is enabled, the client's FLaaS Local App clips gradients and adds noise before sending the update to the server. If **Central DP** is used, raw local updates are sent to the server, where clipping and noise addition are performed before aggregation.

Model Aggregation and Update:

The FLaaS Server aggregates the (potentially noisy) updates using federated averaging. This global model is then redistributed to clients for the next round of training.

Continuous Learning:

The training proceeds iteratively across multiple communication rounds, allowing the model to converge while respecting a predefined privacy budget. The **privacy accountant** (part of the FLaaS Privacy Module) tracks the (ϵ,δ) values throughout the training lifecycle.

Model Usage and Deployment:

Upon convergence, the final trained model can be deployed centrally (e.g., in a cloud-based service) for online inference or sent back to client devices to enable on-device inference, supporting privacy-aware, distributed applications.

4.3.26 Resource-Aware Split Learning with FLaaS

4.3.26.1 Scenario and Requirements

In many smart-home deployments, edge devices lack the compute power or energy budget to train entire deep models locally. At the same time, privacy concerns and communication costs discourage shipping raw data to a central server. This recipe shows how to integrate SL into the FLaaS framework so that each client executes only the initial layers of a neural network (on private data) and offloads the deeper layers to the FLaaS server.

4.3.26.2 Activity Diagram

See section 4.3.18.2.

4.3.26.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.26.4 Architecture and Tools

The implementation of SL into FLaaS implies modifications in both the FLaaS Server and FLaaS local app, as described in Figure 68.

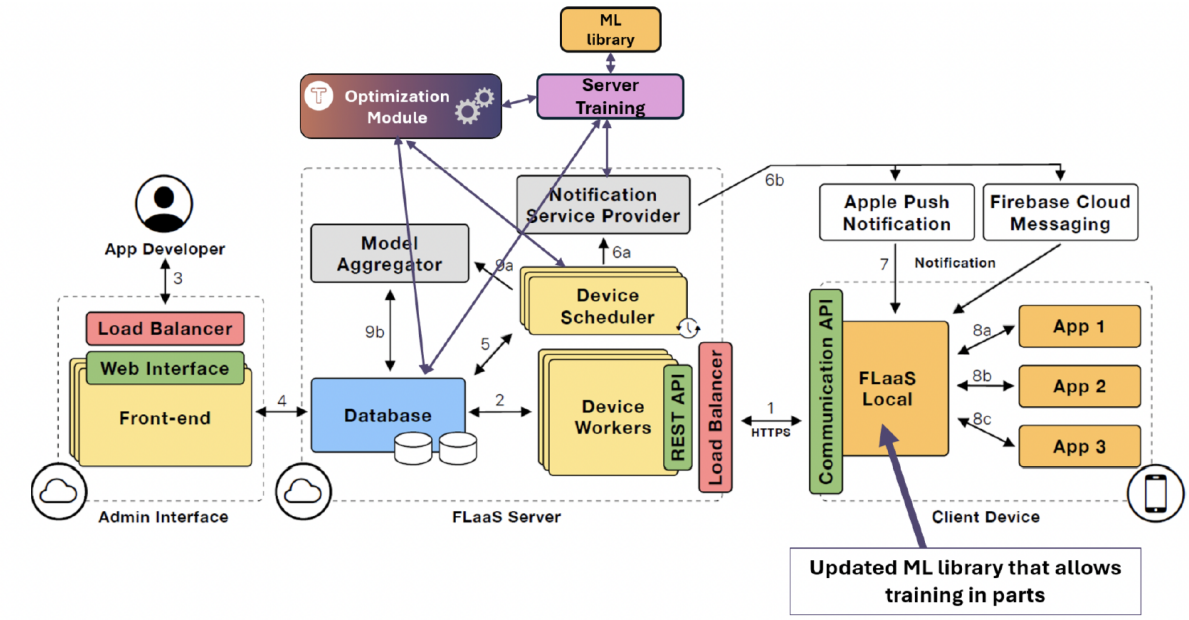

Figure 68: FLaaS Server and local app.

The SL integration extends both the FLaaS Server and the FLaaS Local App. On the server side, the FLaaS backend hosts the downstream sub-model, i.e., the layers following the cut

point. It then receives client activations and labels, completing the forward pass, computing the loss, and executing backpropagation through its sub-model.

Once the server-side backward pass is done, it returns only the gradients corresponding to the cut layer to the client. On the client side, the FLaaS Local App incorporates an updated ML runtime library that supports partitioned training: after receiving the cut-layer gradients, it performs the backward pass through its initial layers and updates local weights.

4.3.26.5 Verification

Static analysis validates that the shapes of activations from the client sub-model match the expected input of the server sub-model. Runtime verification consists of collecting per-round metrics: activation tensor sizes, forward/backward latencies on client and server, and round-completion rates.

4.3.26.6 Machine Learning

The main ML novelty with respect to the recipe of the standalone FLaaS includes ML components to allow training in parts. FLaaS Training begins with a base model that is programmatically split at layer k, chosen based on a device capabilities profile. In each synchronous round, the server triggers clients, which in turn load their local data batches, execute the forward pass through their sub-model up to layer k, and serialise the resulting activations along with labels for transmission.

The server completes the forward and backwards passes on its sub-model, computes the loss, and extracts only the gradients for the cut layer, which it sends back. Clients then apply these gradients to finish backpropagation locally and perform an optimizer step. In federated scenarios with multiple clients per round, the server might additionally aggregate weight deltas via federated averaging before updating the global model. Training proceeds iteratively until convergence. Once the full model is trained, it can be reassembled for centralized evaluation, or its lightweight client sub-model can be deployed for on-device inference under constrained resources.

4.3.27 Energy-Efficient Inference with FLaaS – Knowledge Distillation for Smart-Home Model Deployment

4.3.27.1 Scenario and Requirements

This recipe introduces an energy-efficient training and deployment strategy by integrating KD into the FLaaS framework. The objective is to generate compact, low-power models suitable for smart-home environments, where devices such as sensors, thermostats, and control hubs have limited memory, processing power, and battery life. In this context, a high-capacity model (the *teacher*) is first trained using federated learning across distributed clients or centrally on the FLaaS Server. This model is then used to guide the training of a lightweight *student* model, which is efficient enough to run inference directly on edge devices. The approach balances model accuracy with energy efficiency, enabling privacy-preserving, real-time predictions without frequent communication with the server.

4.3.27.2 Activity Diagram

See section 4.3.18.2.

4.3.27.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.27.4 Architecture and Tools

The architecture (Figure 69) is structured into two key phases: teacher model generation and student model distillation. In the first phase, the FLaaS Server loads a powerful pre-trained teacher model. This model serves as the source of knowledge for the distillation process.

In the second phase, the FLaaS server conducts the knowledge distillation process. It uses the teacher model to guide the training of a smaller student model by leveraging the teacher's predictions—typically soft logits or probability distributions—as learning targets. The student model is trained using a composite loss function that combines standard classification loss (using ground truth labels) with a distillation loss, the Kullback–Leibler divergence between the outputs of the teacher and the student. The distillation process is modulated by a temperature parameter and a weighting factor to balance the contribution of hard and soft targets.

Once training is complete, the resulting student model is much smaller in terms of parameters and computational complexity, while still retaining high performance on the target task. The FLaaS Server then sends the distilled student model to the FLaaS Local App on each client. This deployment strategy reduces both communication cost and local training cost, due to a smaller model size.

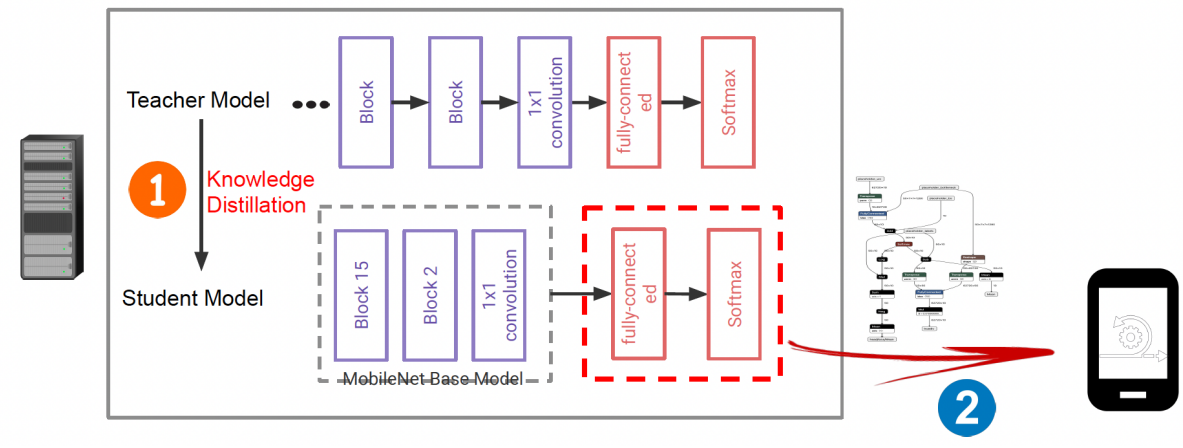

Figure 69: Knowledge Distillation

4.3.27.5 Verification

Static verification is performed before training begins to confirm that the student model architecture is compatible with the teacher model's outputs. This includes validating parameter shapes, output dimensions, and the configuration of the distillation loss components (e.g., temperature scaling and loss weights). The configuration of the training process, including

optimizer settings, learning rate, and batch size, is also checked to ensure consistency with the distillation strategy.

Runtime verification involves continuous monitoring during the distillation process. The FLaaS Server logs and tracks both the classification loss and the distillation loss to ensure convergence and stability. Key training metrics such as accuracy, loss curves, and divergence from teacher predictions are visualized through the TaRDIS Toolkit. These metrics help verify that the student model is successfully capturing the relevant behaviour of the teacher.

After training, the student model undergoes **performance evaluation** to validate that it meets the target criteria for accuracy, model size, inference latency, and memory consumption. These metrics are benchmarked against the original teacher model to ensure acceptable degradation (if any).

Finally, **deployment verification** ensures that the exported student model is compatible with the FLaaS Local App runtime environment. This includes testing for correct model loading, inference performance under constrained resources, and resilience to communication faults or updates. Where necessary, fallback mechanisms or version checks can be included to ensure reliability on device.

4.3.27.6 Machine Learning

The machine learning workflow in this recipe is centred on using knowledge distillation to compress a high-capacity model into an efficient version suitable for deployment on edge devices. The process begins with a pre-trained teacher model. This model exhibits high performance on the target task but is too large or computationally expensive for use in constrained environments.

On the FLaaS Server, a compact student model is initialized. The student is trained using the same input data as the teacher, but its learning targets include not only the ground truth labels (for standard supervised training) but also the soft outputs (logits or probability distributions) produced by the teacher. These soft targets provide richer information about the class relationships and decision boundaries learned by the teacher. The training objective combines classification loss (e.g., cross-entropy) with a distillation loss (e.g., Kullback–Leibler divergence), regulated by a temperature scaling factor and a weight that balances the two terms.

Once the student has converged, it is evaluated against the original teacher model on a validation dataset. If the student achieves an acceptable trade-off between accuracy and efficiency, it is exported and distributed to the FLaaS Local App for inference. The student model is significantly smaller, requiring fewer parameters, lower memory, and reduced computational resources, making it well-suited for real-time inference on smart-home devices.

There is no continuous learning in the standard setup; however, the process can be extended to support periodic retraining. In such cases, as new data becomes available or a teacher model is updated, the distillation process can be repeated to refresh or fine-tune the student. This allows the system to adapt to changes in user behaviour, device usage patterns, or environmental conditions over time.

4.3.28 Privacy-Preserving Learning through decentralized training in smart homes

4.3.28.1 Scenario and Requirements

This recipe describes the deployment of a comprehensive, privacy-preserving, and energy-efficient machine learning pipeline in a smart-home environment using the FLaaS platform. Smart-home devices such as sensors, cameras, and control hubs often operate under constraints in terms of computation, memory, communication bandwidth, and energy. At the same time, they are required to process private data (e.g., voice, motion, temperature patterns) locally to provide personalized and timely responses.

To meet these requirements, the recipe integrates SL, DP, and KD into the FLaaS framework. The goal is to enable lightweight training and inference, ensure data privacy, and support real-time deployment on heterogeneous devices, all while minimising server load and communication costs.

4.3.28.2 Activity Diagram

See section 4.3.18.2.

4.3.28.3 Software Development Life Cycle

See section 4.3.18.3.

4.3.28.4 Architecture and Tools

The recipe architecture combines multiple tools within FLaaS. The system operates in modular phases:

In the SL setup, the NN models are partitioned at a predefined cut layer. The initial layers are executed on the client-side (on the smart-home devices), while the deeper layers are executed on the FLaaS Server. Clients perform the forward pass on their local data and send intermediate activations to the server. The server completes the forward computation, calculates the loss, performs backpropagation, and returns partial gradients to the clients. This reduces local computational cost and limits the exposure of raw data.

Differential Privacy is applied to protect sensitive data during training. The FLaaS server may enable Local DP, where clients add noise to updates before sharing them, or Central DP, where clipping and noise addition are done on the server. Privacy parameters such as (ϵ,δ) are managed through the Django-based FLaaS control plane and enforced by the privacy module embedded in both server and client components.

Once training is complete, KD is employed to compress the high-capacity trained model into a lightweight student model. This process is carried out on the server using the full model and original training data or representative inputs. The resulting compact student model is then deployed to smart-home devices, supporting efficient and low-latency inference while maintaining strong performance.

Funded by the European Union

4.3.28.5 Verification

Verification occurs across all stages of the pipeline. Static checks ensure that model partitioning in SL is valid, student models in Knowledge Distillation are architecturally compatible, and DP parameters are within acceptable bounds. Runtime verification is conducted using the TaRDIS monitoring suite, which tracks training metrics (e.g., loss, accuracy), privacy guarantees (e.g., cumulative (ϵ,δ)), model size, and latency.

Testing includes validating communication between server and client (for both activations and gradients), ensuring privacy-preserving mechanisms are properly applied, and comparing teacher and student model performance. Post-deployment validation ensures that models run correctly on smart-home devices and respond in real time under energy and memory constraints.

4.3.28.6 Machine Learning

The system supports an end-to-end ML workflow. Models are trained in a hybrid federated-split setup, incorporating either Local DP or Central DP for privacy. After convergence, server-side distillation produces an optimized student model. This model is sent to smart-home devices for real-time inference.

Model updates can be triggered periodically or in response to behavioural drift. In future deployments, the student model can be retrained or fine-tuned using new data while continuing to preserve privacy and energy efficiency. The use of split computation during training, combined with DP for privacy and KD for deployment, ensures that smart-home intelligence is delivered securely, efficiently, and scalable.

4.4 TaRDIS IDE Overview

4.4.1 Scenario and requirements

This tool is present within every single test use case scenario established, so this recipe and its description apply to every test scenario.

The primary objective behind the development of TaRDIS was to simplify how the developers interact with distributed systems by automating intricate configurations and providing user-friendly workflows. Building from that starting point, the consortium developed extensions for existing best-of-breed IDEs to be versatile and adapt its features to suit their specific project requirements. The initial extension was built for the Eclipse IDE, but after a survey that was posed to the developer community, the path switched into the customisation of the Visual Studio Code (VS Code) IDE, currently the most popular, especially among the new generations of developers. The objective then is to make an extension to this IDE in order to customize it for the development of swarm environments using TaRDIS.

To achieve this, the proposed extension supports multiple project types: Babel-based Java projects, PTB-FLA projects for federated learning, and DCR choreographies for distributed execution. It ensures seamless integration with essential tools such as Maven for Java projects, Python virtual environments for PTB-FLA, and Docker for running DCR-based

prosumers. This focus on usability, flexibility and cross-platform compatibility was key to creating an extension that enhances productivity while maintaining robust functionality.

For this particular use-case, the proposed scenario to build the EDP environment is to:

- Create a base EDP workspace in VS Code, a common repository where all the project elements will be present.
- Invoke, from the IDE, each of the tools, such as FEDRA, Babel, DCR Choreographies, and eventually the Community Orchestrator tool as well, that will be seamlessly integrated and perform the development actions that are stated in each of the tools' descriptions in this document.

4.4.2 Activity Diagram

As can be seen in Figure 70, the TaRDIS IDE activity diagram follows closely a typical software development life cycle diagram:

Initialisation

- The user (developer) starts by opening the IDE application and initialise it, e.g., opening the TaRDIS extension, configuring the environment, dimensioning the swarm, boundaries and constraints, identifying and defining all environment initial settings, concluding with the establishment of a development workspace, the "base" for the swarm development.
- Alternatively, the developer may start with an already existing workspace, developed previously or acquired as a base template for the development of the swarm.

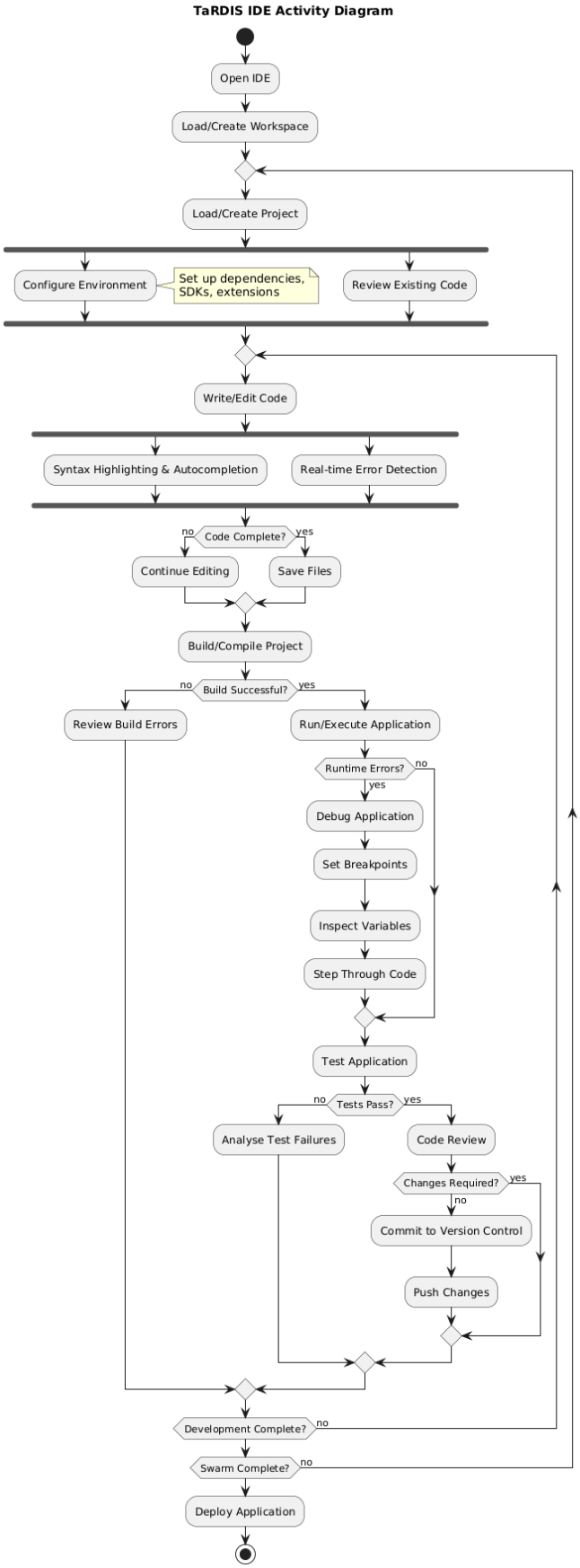

Figure 70: TaRDIS IDE activity diagram.

Development

After the initialisation, the development shall be performed in an iterative manner, as to accomplish the multiple complex interactions and activities, the developer may iteratively:

- create/load software projects specified in different programming languages, or which use diverse tools, e.g., create a TaRDIS DCR choreography using the DCR editor, or create a workflow using the TaRDIS workflow editor. According to the usage of each of the tools and frameworks, additional configuration may be required and needed, as well as the definition of tools, SDKs, APIs and other supporting tools
- The work performed in those tools may then generate its outcomes (e.g., code or other development specifications), which may afterwards be analysed and validated by other tools, e.g., as the nuScr tool or other tools to perform static or dynamic analysis of those outcomes
- The resulting outputs of these tools may be in the shape of code or other text or data specification that can be edited by the IDE's rich set of editors, which include syntax highlighting or analysis tools

Build and Execution

- The resulting outputs of these tools may be interpreted or transformed with the support of code builders and compilers, generating specifications that may be used directly or indirectly by the target business, such as the Actyx middleware and runtime frameworks
- The IDE can dynamically gather tools and information from other projects and data sources, support the building and execution of the application with debuggers and tools that analyse metrics and KPIs, can run the project step by step, analyse the project data, and other activities
- The IDE supports the concepts of test campaigns, assertions and frameworks for unit tests
- If the outcomes of the tests are satisfactory, the application can reach its objectives otherwise it may result in a new iteration of development and corrections.

Multiple Projects on the same Workspace

- The heterogeneous and complex context of swarm applications may require more than one development activity. Hence, after defining a DCR Choreography, the developer may require additional resources, such as the establishment and training of an AI/ML environment, by creating additional projects e.g., using FEDRA, PTB-FLA or FAuNO over the same workspace which already has some precious information about the swarm activities
- These projects may result in outcomes that are blended with the previously obtained information, thus enriching the workspace and making it more complex

Engagement and Dissemination

- The TaRDIS IDE seamlessly integrates with the most popular source code repositories and engagement platforms e.g., GitHub, GitLab. This increases the visibility of the source

code and allows it to be shared, versioned, and distributed within a community of developers

4.4.3 Software Development Life Cycle

The TaRDIS IDE is being developed on a SCRUM-based lifecycle (Figure 71), where each of the TaRDIS tools, and particularly the ones in this use-case were reported in the development backlog, and accordingly, the CMS team has been meeting each of the tool owners, presenting the development team's vision of what the TaRDIS tool's functionalities are. Then the team performs a 15-day sprint to perform the integration of the tool in the Integrated Development Environment, including documentation, APIs and centralisation of functionalities. The team then has a follow-up meeting with the tool owner for early validation of the approach, the expectations are aligned and the team finalizes the integration approach.

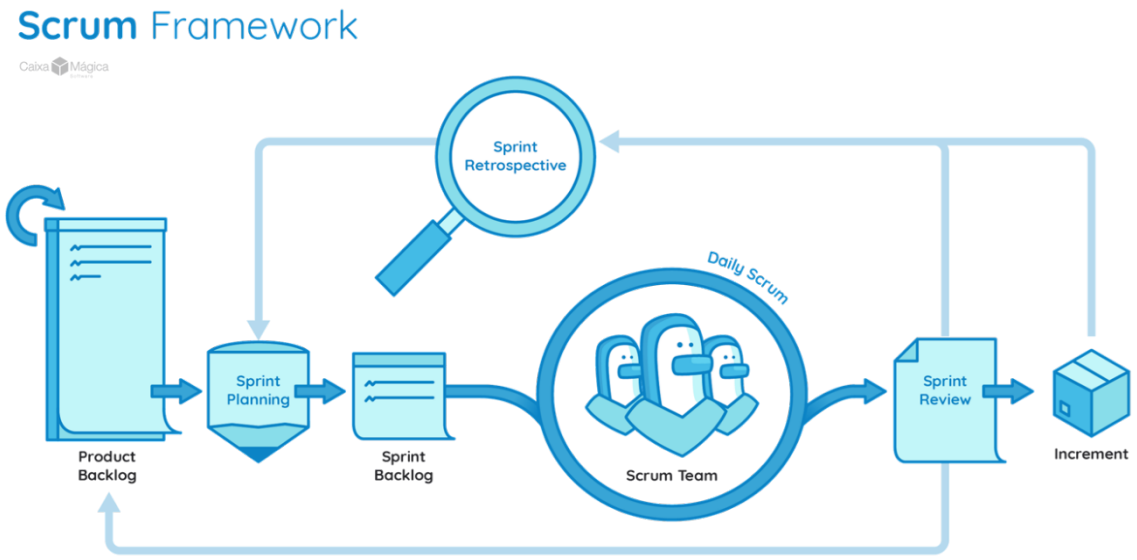

Figure 71: Scrum development lifecycle.

4.4.4 Architecture and Tools

The TaRDIS Visual Studio Code extension is a comprehensive toolbox designed to streamline the development and management of distributed systems and projects. The extension provides a range of functionalities, including project creation, protocol importing, and class generation, to help developers quickly and efficiently set up and manage their projects. This manual outlines how to use the TaRDIS extension and its current features.

Tools and frameworks

Node.js

Node.js was used as the runtime environment to build the extension and execute its operations. It facilitated seamless integration with the Visual Studio Code API while providing

an extensive ecosystem for dependency management through `npm` and helped build the structure of the extension with existing `npm` packages like 'yo' and '`generator code`'.

TypeScript

TypeScript was chosen as the primary development language for its static typing, modern JavaScript features, and strong compatibility with the Visual Studio Code API. This choice ensured the extension's reliability, maintainability and ease of debugging as the tools provided by the `yo` Node Package Manager facilitated the integration using this language.

Visual Studio Code API

The VS Code API enabled the extension to interact directly with the IDE, allowing for the registration of commands, dynamic file manipulation, and the rendering of interactive webviews. This API served as the backbone for the extension's core functionality. The API documentation can be found in the link https://code.visualstudio.com/api/references/VS Code-api.

Webview API

The Webview API played a crucial role in creating visually rich and interactive panels that allowed users to make choices. These panels were used for user inputs, project configuration, protocol documentation and DCR choreography management. This ensured the extension maintained a consistent design while delivering clarity to users.

Maven

For Java-based projects, Maven was employed to manage project dependencies and build configurations. The extension has the ability to automatically update the pom.xml file to include the required Babel libraries and repositories at the press of a button, eliminating any manual configuration steps.

Python Virtual Environment

For PTB-FLA projects, the extension created a dedicated virtual environment (venv_ptbfla) with pre-installed dependencies. This environment ensured compatibility with machine learning libraries, providing a ready-to-use workspace for federated learning development.

Docker

Docker was integrated into the extension for running DCR choreographies. The extension automated the creation of Docker containers, networks, and images, ensuring each prosumer instance operated in an isolated yet interconnected environment.

Markdown Renderer

A custom markdown renderer was implemented to display documentation within webview panels. This allowed users to view formatted protocol specifications, configuration parameters, and API documentation directly within VS Code.

Yeoman

The initial project structure was scaffolded using Yeoman's generator for VS Code extensions, providing a modular foundation for the codebase and ensuring a consistent development workflow.

Tools required for operation:

To ensure seamless operation, the following tools are required:

Java Development Kit (JDK)

Java 21 is essential for Babel-based projects, providing the runtime environment for Java applications.

Node.js

Serves as the runtime environment for the extension itself, managing dependencies and executing commands.

Maven

Used for dependency management and build automation in Java projects.

Python

Required for PTB-FLA projects, with virtual environments ensuring package isolation.

Docker

Facilitates containerized execution of DCR choreographies.

Visual Studio Code

Serves as the primary development environment.

Git

Enables version control, code sharing, and collaborative development.

4.4.5 Verification

This tool is mainly for performing development, although it can be foreseen that some of the TaRDIS verification tools may be in the future also integrated in the IDE, thus allowing the IDE the capability of invoking them and hence allow verification and validation of swarm environments.

4.4.6 Machine Learning

ML tools are not directly used in this recipe, although the IDE may integrate AI/ML tools such as PTB-FLA or FEDRA, but these shall then be described separately on their own sections.

4.5 Discussion

4.5.1 Overview

The TaRDIS development approach is essentially independent of the chosen software development life cycle. The different recipes illustrate how the TaRDIS toolset was used in different life cycles. A common theme that emerges when analysing the different use cases and the recipes emerging from them is that the choice of the software development life cycle aligns with the practice of the organisations conducting the use case, rather than with the usage of TaRDIS. This supports one of the goals underpinning the TaRDIS project: the TaRDIS Development Approach should be agnostic to the different software development life cycles, so it could be adopted regardless of the particular chosen life cycle.

The life cycles adopted in the different use cases include planned life cycles, with nuanced versions of the waterfall model, but also agile approaches with Scrum, and even the usage of a DevOps approach.

4.5.2 TaRDIS tools usage throughout the life cycle

Concerning the usage of the different TaRDIS tools, they span the life cycle, with a clearer focus on the implementation stage of the life cycle, regardless of the particular life cycle in place. We now discuss the TaRDIS tools role in each life cycle phase.

4.5.2.1 Requirements elicitation and analysis

The requirements elicitation and analysis phase is dominated by activities conducted manually by domain experts and developers.

Requirements elicitation and analysis typically start with a feasibility study, to determine if a swarm can be engineered with resource constraints (e.g. battery life of an IoT sensor, or the computing power of a satellite board), and how the swarm integrates with the broader ecosystem. During the requirements process, it is essential to capture the use case business goals and identify the corresponding high-level Key Performance Indicators (KPIs), such as throughput, downtime reduction, or mission-specific goals such as autonomous orbit determination.

Another key activity concerns domain modelling, where key components like satellites, prosumers and sensors, their relationships, and expected behaviour need to be specified.

The process includes identifying both functional (e.g., perform autonomous ODTS onboard LEO mega-constellations, or forecasting energy production and consumption using geo-information and historical datasets), and non-functional requirements (e.g., having low reliance on ground stations and resilience against inter-satellite link failures, or supporting differential privacy).

These requirements are then codified into protocols governing the desired swarm behaviour. The recipes collected in this report illustrate how high-level requirements can be translated into decentralized systems using TaRDIS tools including Babel, PTB-FLA and the DCR Editor.

Funded by the European Union

The recipes included in this document and listed on Table 2 span from defining core swarm logic, industrial applications such as manufacturing and aerospace, decentralized intelligence concerning energy and federated learning.

Table 2: Recipes scenarios and key requirements summary.

Recipe	Scenario	Key requirements
3.1	Composition of preexisting swarms.	Modularity, confusion-freeness, role-based interfaces.
3.2	Runtime monitoring of swarm activity/QoS.	Actyx-implemented swarm, join definition (patterns/guards).
3.3	Reprogramming manufacturing processes.	Reusability, controllability, event-first domain modelling.
3.4	Nominal satellite constellation ODTS.	Connection schedules, state vectors, RF-GMV-01 to 17.
3.5	ISL re-scheduling during satellite failure.	Failure diagnosis (dictionary), synchronous schedule updates.
3.6 / 3.24	Project deployment via TaRDIS IDE.	Support for Babel, PTB-FLA, DCR, Maven, Python venv, and Docker.
3.7	EDP Energy Community development.	Community universe topology, historical production/consumption data.
3.8	Creating DCR Choreographies for peers.	Participant roles (consumer/producer), event constraints/relations.
3.9	Coordination of energy events.	Energy balance variables, peer type, decision tree algorithm.
3.10	Energy forecasting using Fedra/Pruning.	Historical local data, LSTM networks, FL training parameters.
3.11	Babel communication between energy entities.	Specialized membership service, middleware data propagation.

Recipe	Scenario	Key requirements
3.12	Coordination based on forecast data.	Pre-trained local LSTM models, energy balance features.
3.13	Mapping Coordination App to DCR.	DCR choreography types.
3.14	DCR choreographies via Babel stack	REST API for coordination-to-Babel communication, data confidentiality verification.
3.15	Multi-level Grid Balancing	Sequential execution of all EDP recipes, historical prosumer data.
3.16	Factory workflow anomaly detection.	Unsupervised learning task selection, federated client criteria.
3.17	Distributed Early-Exit inference.	Model splitting strategy, confidence intervals, swarm topology.
3.19	FLaaS (Federated Learning as a Service).	App developer interface, cloud-hosted server, Android client devices.
3.20	Differential Privacy (DP).	Mitigation of gradient inversion, noise scales (sigma), (ϵ,δ)-DP parameters.
3.21	Split Learning (SL).	Designated cut layer, matching weight shapes, intermediate activations.
3.22	Knowledge Distillation (KD).	High-capacity teacher model, efficient student model, composite loss.
3.23	Babel for TID smart-home entities.	Android support, membership service for market data exchange.
3.24	IDE for TID project deployment.	Base TID workspace, Java 21 JDK, Python virtual environments.

Recipe	Scenario	Key requirements
3.25	DP Integration with FLaaS.	Django admin interface for parameter (ϵ,δ) configuration, Opacus library.
3.26	Resource-Aware Split Learning.	Defined "cut" layer, matching weight shapes for sub-models, forward/backward latency monitoring.
3.27	Energy-Efficient Inference (KD).	High-capacity teacher model, student model architecture, temperature scaling parameters
3.28	Privacy-preserving smart-home learning.	Integration of SL, DP, and KD on heterogeneous devices.

Recipes 3.1 and 3.2 focus on swarm engineering. Recipe 3.1 addresses the challenge of combining independently developed swarms into an integrated system, while recipe 3.2 provides a scenario for runtime monitoring of the swarm activity and the emerging quality of service.

Recipes 3.3 through 3.5 focus on industrial (3.3) and aerospace applications (3.4 and 3.5). Recipe 3.3 concerns a manufacturing environment where reprogramming a process often implies costly production halts, which make non-functional requirements such as reusability, controllability, testability, implementation ease, and maintainability crucial. The recipes resort to an event-first domain model that identifies key assets, such as machines, conveyor belts, or AGVs, their states and transitions. Recipes 3.4 and 3.5 focus on satellite constellations, autonomous orbit determination, and autonomous rescheduling to handle communication losses without ground intervention.

Recipes 3.6 through 3.15 focus on a multi-level decentralized energy market with prosumers and community orchestrators making agreements concerning energy production and consumption. These recipes rely on historical datasets for supporting forecast models with Fedra, role-based participant profiles (DCR) and specialized communication middleware (Babel) to manage peer-to-peer transactions.

Recipe 3.16 targets factory anomaly detection using unsupervised learning on smart-factory data. Recipe 3.17 introduces “early-exit” strategies for deploying lightweight ML models on resource-constrained edge devices. Recipes 3.19 through 3.28 illustrate Federated Learning as a Service (FLaaS) with requirements focusing on Differential privacy to prevent data reconstruction attacks, Split Learning to offload computation to a server, and Knowledge Distillation to compress models into lightweight versions for sensors.

Funded by the European Union

4.5.2.2 Design

The design phase supports the transition from the problem domain (requirements) to the solution domain (executable systems). In this phase, TaRDIS tools support architectural design, defining system components and their distribution among devices, and lower-level specification of protocols, and of the logic underpinning swarm interactions, listed in Table 3.

Table 3: Recipes design activity summary.

Recipe	Core Design Activity	Specific Tool or Output
3.1	Designing modular, confusion-free swarm protocols.	Finite State Machines
3.2	Designing **join definitions** and patterns for QoS monitoring.	join-actors library
3.3	**Event-first domain modeling**; partitioning assets into runners.	Event schemas and state transitions
3.4	Generating baseline Python code for ODTS algorithms.	GMV baseline code
3.5	Designing deterministic on-board **re-scheduling algorithms**.	Failure dictionary
3.6	Designing **interactive ebviews** and automated **project scaffolding** for distributed systems.	VS Code Extension / Yeoman
3.7 / 3.8	Defining roles, activities, flows, and constraints for choreographies.	DCR Choreography model
3.9	Designing **event-enablement policies** and **decision tree logic** to trigger DCR choreographies	Coordination Application (Python ML code)
3.10	Configuring energy community parameters and FL hyperparameters.	Fedra configuration
3.11	Designing independent protocols as state machines.	Babel networking channels
3.12	Designing the mapping logic between ML forecasts (energy balance) and coordination actions.	Decision tree / LSTM models

Funded by the European Union

Recipe	Core Design Activity	Specific Tool or Output
3.13	Designing a **decision tree** to select appropriate choreographies.	Coordination application logic
3.16	Task definition and model selection/recommendation.	Flower-based FL model
3.17 / 3.21	Designing **model splitting strategies** (cut layers) for distributed inference.	D-Exit / Split Learning code
3.22 / 3.27	Designing teacher-student architectures for model compression.	Lightweight student model
3.28	Orchestrating NN model partitioning, privacy parameter enforcement, and teacher-student model distillation.	FLaaS Platform (Integrated SL/DP/KD pipeline)

The design phase focuses on three main areas: protocol and choreography specification, domain and architecture modelling, and decentralized intelligence configuration.

Protocol and choreography specification formalize how swarm members interact. Examples include swarm composition (Recipe 3.1), where developers need to design confusion-free protocols to support a safe integration of pre-existing swarms, and recipes 3.7 and 3.8, which use the DCR editor to model roles, activities, flows and constraints for choreographies, leading to a DCR Choreography model.

Domain and architecture modelling may start with identifying assets, such as AGVs, their states and events describing their behaviour, with Event-first domain modelling, e.g., in a manufacturing context (Recipe 3.3). Modelling can also focus in monitoring scenarios, where designers create a join definition encoding message patterns and guards to monitor at runtime (Recipe 3.2).

Machine learning recipes require specialized design steps for distributed environments. For instance, the design of splitting strategies for Neural Networks enables the distribution of different portions of the model to run in different devices (e.g., in recipes 3.17 and 3.21). The design of knowledge distillation pipelines offers another form of knowledge distillation, where a complex model guides the training of a lightweight model for usage in edge components (e.g., in recipes 3.22 and 3.27).

4.5.2.3 Implementation

The main goal for this stage is to evolve detailed design specifications into executable software systems. The implementation phase includes the usage of several of the TaRDIS toolset

elements, including libraries for swarm logic, decentralized communication middleware, and automated machine learning frameworks.

One of the components of implementation concerns swarm logic and finite state machines and relies on the usage of the `machine-runner` library for the implementation of swarm logic protocols and state transitions management, for coordination. In addition, TypeScript asset connectors can act as intermediaries between the software state and the physical factory machines.

Babel protocols are configured to implement decentralized communication. For DCR-based systems, implementation includes using the DCR compiler to translate global specifications into local executable code for individual nodes.

Several recipes rely on decentralized and federated intelligence, often adopting the Single Program Multiple Data pattern. Developers use PTB-FLA to integrate federated learning logic with their decentralized Python code. In other recipes, developers use the FLaaS framework to integrate Differential Privacy, Split Learning, and Knowledge Distillation.

The TaRDIS IDE, implemented as a VS Code extension, provides an integrated development environment orchestration, automating the setup of the implementation environments, across several languages (e.g., Java, Python, and TypeScript). It scaffolds project structures using Yeoman, manages Java dependencies with Maven, sets up Python virtual environments for ML, and automates Docker containerization for deployment testing.

Table 4 summarizes the key implementation activities in each recipe.

Table 4: Implementation activities.

Recipe	Key Implementation Activities	Primary Tools / Languages
3.1	Developing swarms to realize protocols; adapting machines for composition.	machine-runner / TypeScript
3.2	Translating TypeScript types to Protocol Buffers; generating forwarding nodes.	smCLI, join-actors / Scala
3.3	Coding asset connectors; setting up CI/CD and containerization.	Actyx SDK / TypeScript
3.4 / 3.5	Integrating generic TDM algorithms; exchange of data over computer networks.	PTB-FLA, Babel / Python

Recipe	Key Implementation Activities	Primary Tools / Languages
3.6 / 3.24	Project scaffolding; automatic update of Maven/Python configurations.	TaRDIS IDE / Node.js, TS
3.7 / 3.9	Coding policies via coordination app; populating peer databases.	Coordination App / Python
3.8 / 3.13	Compiling choreographies to managed services; linking logic to DCR.	DCR Editor, Compiler / Docker
3.10	Integrating FL training code; pruning models for energy efficiency.	Fedra, Pruning Tool / Python
3.11 / 3.23	Generating protocol instances; encoding information flow for destination nodes.	Babel Middleware / Java
3.12	Implementing Python code for online inference and feedforward operations.	Fedra, Coordination App / Python
3.14	Implementing REST APIs to bridge Babel layers and coordination apps.	DCR Execution Stack / REST
3.15	Sequential execution of Fedra, Coordination, DCR, and Babel recipes.	Full TaRDIS Stack
3.16	Incremental workflow configuration; dataset selection via tool interface.	Flower-based FL Tool / Python
3.17	Defining model splitting strategies; deploying parts to swarm peers.	EE & D-Exit Tool
3.18 / 3.19	Refactoring FLaaS legacy code; generating helper code via FastAPI.	FLaaS / Android, Django
3.20 / 3.25	Integrating clipping and Gaussian noise addition.	Opacus, Flower.ai

Recipe	Key Implementation Activities	Primary Tools / Languages
3.21 / 3.26	Partitioning NNs; implementing client-side/server-side sub-models.	FLaaS / ML libraries
3.22 / 3.27	Training student models with composite loss; temperature scaling.	FLaaS Server / Python
3.28	Orchestrating integrated SL, DP, and KD pipelines.	FLaaS Platform

4.5.2.4 Verification

Verification in TaRDIS is integrated throughout the development lifecycle. Its main focus areas are the correctness of decentralized protocols and the integrity of the machine learning models. In the recipes, we observe several layers of verification activities, including static analysis to catch defects before execution, formal verification to prove system properties, runtime monitoring to gauge system behaviour during execution, and systematic testing to bridge the gap between simulation and reality.

Static analysis and formal verification aim at “correctness by construction”. The `machine-check` library statically verifies that swarm protocols are confusion-free and non-concurrent. In federated learning, PTB-FLA uses CSP and the PAT model checker to prove properties such as deadlock freedom and successful termination (e.g., in Recipe 3.4). The DCR compiler provides projectability checks, so that DCR Choreographies are statically checked for correctness during translation to locally executable code.

TaRDIS tools support runtime monitoring of decentralized systems. Babel supports metrics collection on nodes (both hardware and kernel) and protocols (e.g. latency and throughput). Tools like Prometheus and Grafana collect statistics from containers, in manufacturing settings. Machine learning recipes with FLaaS do runtime monitoring of the cumulative privacy budget, in Differential Privacy, and distillation loss, in Knowledge Distillation.

Recipes involving DCR Choreographies use a testing runtime environment within Docker to simulate distributed environments and monitor message exchanges. Industrial and aerospace use cases progress from unit testing (using `machine-check`) to integration testing with mocks, and finally on-premises site acceptance tests.

Using tools like machine-check and the DCR compiler early in the design phase is more efficient than late-stage testing. These tools ensure protocols are mathematically sound before the application code is executed. In Federated Learning, verification activities (like privacy accounting) have revealed that Local Differential Privacy provides stronger security but results in higher utility degradation compared to central approaches, requiring rigorous post-training validation. For aerospace applications (Recipe 3.5), deterministic algorithms were preferable to Reinforcement Learning because their behaviour can be strictly verified against a "failure

Funded by the European Union

dictionary" under operational constraints, suggesting a preference for deterministic decision-making in safety-critical scenarios. Recipes 3.12 through 3.15 highlight that the greatest verification challenge is not individual tools, but the inter-tool communication (e.g., ensuring a coordination app triggers the correct choreography), necessitating integrated test reports. Table 5 summarizes the verification activities in these recipes.

Table 5: Verification activities.

Recipe	Primary Verification Activity	Tools and Metrics Used
3.1	Static verification of protocol modularity and confusion-freeness.	machine-check library
3.2	Protocol fidelity checks and fair message matching guarantees.	smCLI, join-actors
3.3	Multi-layer testing: unit, integration (mocks), and Site Acceptance Tests.	machine-check, Prometheus, Grafana
3.4 / 3.5	Formal verification of algorithms; C-code safety checks; ODTS metric analysis.	CSP/PAT, Cppcheck, assertions
3.6 / 3.24	Foreseen integration of static and runtime verification within the IDE.	TaRDIS IDE
3.7	Validation with domain experts; security and confidentiality verification.	DCR Editor, IFChannel, (Sec)ReGraDa
3.8	Static projectability checks; runtime testing in Docker containers.	DCR Compiler, Docker
3.9 / 3.13	Verification of triggered choreographies (projectability and well-formedness).	DCR Editor, VS Code logs
3.10 / 3.12	No verification currently plays a role in these recipes.	N/A
3.11 / 3.23	Real-time monitoring of hardware, kernel, and protocol metrics.	Babel Core, time-series DB

Funded by the European Union

Recipe	Primary Verification Activity	Tools and Metrics Used
3.14	Static correctness and data confidentiality verification.	DCR Compiler, (Sec)ReGraDa
3.15	Aggregate verification of DCR, coordination policies, and Babel stacks.	DCR Editor, Prometheus
3.16	Data preprocessing; evaluation of training stability and performance.	Flower-based FL Tool
3.17	No verification currently plays a role in this recipe.	N/A
3.19	Consistency checks for project descriptors and hyperparameter bounds.	FLaaS Server, admin dashboard
3.20	Cumulative privacy budget tracking; static analysis of clipping bounds.	Privacy accounting tools
3.21 / 3.26	Weight shape matching; gradient flow and latency monitoring.	FLaaS SL runtime
3.22 / 3.27	Distillation loss tracking; student-teacher divergence evaluation.	FLaaS KD module
3.28	Holistic verification of SL, DP, and KD layers.	TaRDIS monitoring suite

4.5.2.5 Deployment

TaRDIS supports deployment by automating intricate configurations through a centralized IDE and providing specialized libraries for decentralized execution. Deployment activities across the recipes focus on three primary areas: infrastructure automation, model optimization for edge devices, and middleware orchestration.

The TaRDIS IDE supports infrastructure automation, and, as such, deployment (see, e.g. Recipes 3.6 and 3.24). The TaRDIS IDE automates the creation of Docker containers, networks, and other elements, to support decentralized components, such as DCR prosumers, operating in isolated, but interconnected environments. The IDE also manages Python virtual environments for federated learning and Maven configurations for Java-based Babel projects.

In industrial and aerospace scenarios, deployment is done in physical hardware, rather than simulated environments. In recipe 3.3 runners and nodes are packaged into containers and staged, before being rolled out to production. In recipes 3.4 and 3.5, in the aerospace domain, the final code is deployed in satellite boards, following rigorous verification.

The deployment of communication layers using Babel (e.g., recipes 3.4 and 3.5), includes creating instances of Babel protocols to which developers provide configuration parameters, such as community neighbours and topology to support node-to-node communication. As for DCR Choreographies (e.g., in recipe 3.8) a compiler transforms global specifications into local executables that are then shipped to participant devices (Babel nodes).

In Federated Learning scenarios (FLaaS), deployment involves a cloud-hosted server orchestrating client devices (e.g., recipes 3.19, 3.28). When exporting lightweight models to resource-constrained devices, such as sensors or IoT hubs, techniques like pruning (recipe 3.10) or Knowledge Distillation (e.g., recipe 3.27) can be used.

Managing the complex configurations benefits from an IDE that hides this complex and error-prone activity behind a user-friendly interface. Deploying raw AI models to the edge is often unfeasible, due to resource constraints. Techniques like pruning, knowledge distillation, and early-exit strategies are therefore essential for making decentralized intelligence viable in real-world swarms. In the aerospace domain, deterministic decision-making is preferable over reinforcement learning, as critical on-board re-scheduling undergoes strict operational constraints and requires predictable behaviour.

Table 6 summarizes the deployment activities in these recipes.

Table 6: Deployment activities.

Recipe	Key Deployment Activities	Primary Infrastructure / Tool
3.1	Composing verified swarms; deploying finite state machines.	machine-runner library,
3.2	Automated generation and deployment of forwarding nodes.	smCLI tool,
3.3	Packaging runners/nodes; staging in "shadow mode"; production rollout.	Docker-compose / Helm
3.4 / 3.5	Final code deployment on satellite/representative hardware boards.	Satellite Hardware

Recipe	Key Deployment Activities	Primary Infrastructure / Tool
3.6 / 3.24	Automating Docker, Maven, and Python venv configurations.	TaRDIS IDE (VS Code),
3.7 / 3.15	Generating final code for energy community prosumer devices.	P2P Energy Devices
3.8 / 3.14	Shipping compiled local choreographies to Babel-enabled nodes.	DCR Compiler / Docker,
3.9 / 3.13	Triggering local choreography instances based on forecast data.	Local Peer Node
3.10 / 3.12	Storing pruned LSTM models locally for online inference.	Edge Smart Home Devices,
3.11 / 3.23	Generating and deploying protocol instances with neighbour configs.	Babel Middleware
3.16	Conceptual deployment of federated clients as swarm agents.	Simulated Factory Nodes,
3.17	Partitioning and deploying model parts (IoT, Edge, Cloud) for redundancy.	D-Exit Framework,
3.18 / 3.19	Orchestrating Android-based devices and cloud-hosted servers.	FLaaS / Mobile Platforms,
3.20 / 3.25	Deploying DP-configured models with noise addition parameters.	FLaaS Local App / Django,
3.21 / 3.26	Deploying partitioned sub-models (initial layers to client, deep to server).	FLaaS Server & Local App
3.22 / 3.27	Exporting and distributing compact "student" models to sensors/hubs.	IoT Sensors / IoT Hubs,

Funded by the European Union

Recipe	Key Deployment Activities	Primary Infrastructure / Tool
3.28	Unified deployment of SL, DP, and KD layers in smart homes.	Heterogeneous Smart Devices

4.5.2.6 Operation

Once deployed, decentralized applications run in a live environment, where they need to be monitored for continuous feedback and evolution, ideally leveraging practices inspired by DevOps principles. The operation activities in these recipes cluster around three main areas: runtime monitoring, online inference, and autonomous self-management.

Most recipes use specialized tools to observe their runtime behaviour. Babel collects hardware, kernel, and custom protocol runtime metrics and stores them on a time-series database for real-time visualization. Manufacturing recipes use Prometheus and Grafana dashboards. Some recipes (e.g. recipe 3.2) use join pattern matching to identify specific message combination, such as detecting a factory machine failure from a timeout event.

Several recipes use machine learning models in their operational environment. After trained and pruned, LSTM models are stored locally for online inference so they can be used in real-time decision making (e.g. providing energy production and consumption forecasts). Federated Learning recipes support soft retraining by gathering data in the operational environment and using that data to tune pre-trained models without requiring a complete redeployment.

Swarm environments usually require high levels of autonomy. Babel supports mechanisms for self-configuration and self-management so that protocols can dynamically optimize resource allocation and manage their own lifecycles to ensure robustness. In safety-critical aerospace scenarios, nodes monitor for failures using a failure dictionary and can autonomously trigger on-board re-scheduling to maintain the integrity of the satellite constellation if a peer is lost.

Table 7 summarizes the main operation activities in the recipes.

Table 7: Operation activities.

Recipe	Key Operation Activities	Tools and Metrics
3.1	Monitoring composed swarm fidelity to protocols.	machine-runner library
3.2	Fair join pattern matching for QoS monitoring.	smCLI, join-actors library
3.3	Live performance tracking in production (shadow/production modes).	Prometheus, Grafana, Node Exporter

Recipe	Key Operation Activities	Tools and Metrics
3.4 / 3.5	Real-time failure diagnosis; autonomous ISL re-scheduling.	Babel metrics, failure dictionary
3.6 / 3.24	Integrated log viewing and container management.	VS Code integrated terminal, Docker
3.7 / 3.15	Real-time energy transaction management on prosumer devices.	Babel Stack, DCR execution layer
3.8 / 3.14	Monitoring message exchanges and event triggers via real-time logs.	VS Code integrated terminal
3.9 / 3.13	Logic-driven event enablement; online role transitioning.	Coordination Application
3.10 / 3.12	Local online inference for energy forecasting; soft retraining.	Pruned LSTM models
3.11 / 3.23	Autonomous protocol lifecycle management; self-optimization.	Babel Self-management module
3.16	Stability assessment; anomaly detection in factory workflows.	Flower-based inference tool
3.17	Distributed inference; energy-efficient load balancing across nodes.	D-Exit Framework, Early-Exit tool
3.19	FL workflow monitoring; push-notification-driven system activity.	FLaaS Server, Notification Service
3.20 / 3.25	Runtime privacy accounting (tracking cumulative ϵ).	Privacy accounting tools, Django
3.21 / 3.26	Monitoring forward/backward latencies and activation tensor sizes.	Split Learning runtime

Funded by the European Union

Recipe	Key Operation Activities	Tools and Metrics
3.22 / 3.27	Tracking distillation loss and student divergence from teacher predictions.	KD module, student model inference
3.28	Holistic runtime tracking of privacy, model size, and latency.	TaRDIS monitoring suite

5 CONCLUSIONS

This document presents the updated approach towards the definition of an Integrated Development Environment tool suited for the TaRDIS toolbox integration efforts.

In conclusion, the integration of Visual Studio Code (VS Code) with the toolset explored in the TaRDIS research project provides a coherent and extensible environment for software development in heterogeneous swarm settings. By combining the native capabilities of VS Code with project-specific tools, the deliverable shows how the TaRDIS IDE can reduce fragmentation across the toolchain and support more consistent development workflows.

Throughout the study, it became evident that the combination of VS Code and the TaRDIS toolbox tools not only enhances code editing and debugging experiences but also fosters a smoother collaboration across various stages of development. The use of version control systems, integrated testing frameworks, and advanced extensions—combined with VS Code's seamless support for languages, frameworks, and debugging tools—has created an ecosystem that can adapt to a diverse range of project requirements and developer preferences.

Furthermore, the extension architecture in VS Code proved to be a key advantage, allowing for highly specific customization to match the unique demands of different project types. The ability to incorporate third-party tools, such as continuous integration/deployment pipelines, Docker containers, and cloud platforms, extends VS Code's capabilities, making it an ideal choice for developers working on projects of varying scope and complexity.

From a research perspective, the integration of these tools has highlighted areas of potential improvement in the development cycle, including streamlining the process for tool configuration, ensuring robust documentation for new users, availability of templates, examples and other support, and refining workflows for collaborative development. The overall results confirm that the synergies created by leveraging VS Code as the central hub for development significantly enhance the overall software development lifecycle.

Ultimately, this research reinforces the importance of using a flexible, well-supported, and extensible Integrated Development Environment (IDE) like Visual Studio Code as a highly added value in conjunction with the TaRDIS toolbox, as it fosters an environment conducive to innovation, rapid prototyping, and efficient code delivery in modern software projects. The outcomes suggest that further research into refining these integrations and expanding the set of supported tools will continue to improve both individual and team-based software development efforts in the future.

This document not only presented the customisation of the VS Code IDE to cope with the TaRDIS development needs but also showed how the customisation and integration of each of the selected tools was performed, from the point of view of the integration process and from the point of view of the operational instructions.

Funded by the European Union

The development of decentralized, heterogeneous swarm applications requires a framework capable of managing the complexities of coordination, communication, and limited edge resources. This report details the TaRDIS development approach, simplifying the engineering of such systems through a suite of programming models, APIs, and integrated tools. This framework helps bridge the gap between high-level conceptual requirements and executable decentralized logic. This report presents a recipe catalogue including 28 recipes taken from the TaRDIS project. These recipes illustrate the application of the toolbox across a wide range of scenarios, including autonomous satellite orbit determination (Aerospace), dynamic reprogramming of manufacturing processes (Industrial), multi-level grid balancing (Energy), and privacy-preserving federated learning (Smart Homes). Each recipe includes a scenario description, activity diagrams, tool usage matrices, and verification strategies to ensure developers can replicate complex decentralized behaviours with a high degree of confidence.

A notable feature of these recipes is the SDLC-agnostic nature of the TaRDIS toolkit. The various case studies illustrate that the toolbox can be easily integrated into existing organizational practices, whether they utilize predictive Waterfall models, iterative Agile/Scrum methodologies, or continuous DevOps frameworks. This is further enhanced by the TaRDIS IDE, which abstracts the configuration of multi-language, multi-platform environments.

Where possible, the approach adopts a "correctness-by-construction" philosophy. By incorporating formal verification methods (such as CSP/PAT for federated learning) and static analysis tools (such as the machine-check library for protocol modularity) directly into the early phases of the development life cycle, the TaRDIS approach delivers essential safety and liveness guarantees for mission-critical swarms. These verification layers, complemented by runtime monitoring via the Babel middleware and Prometheus-based dashboards, ensure system integrity from initial design through live operations.

Funded by the European Union

REFERENCES

[1] D3.2 – First release of TaRDIS development environment, 2024, TaRDIS project, https://www.project-tardis.eu/download/d3-2-integrated-development-environment

[2] D3.4 – Second release of TaRDIS development environment, 2025, TaRDIS project, https://project-tardis.eu/download/d3-4-second-report-on-integrated-development-environment

[3] Stack Overflow 2025 developer survey on most popular technologies, https://survey.stackoverflow.co/2025/technology#most-popular-technologies-dev-envs-dev-envs, accessed on 2026-02-15.

[4] GitHub, Code OSS open-source code, https://github.com/code-oss-dev/code, accessed on 2026-02-15.

[5] Microsoft (n.d.). Code editing. Redefined. Visual Studio Code. https://code.visualstudio.com, accessed on 2026-02-15.

[6] Microsoft (2024, January 2). *Extension API*. Visual Studio Code. https://code.visualstudio.com/api, accessed on 2026-02-15.

[7] Git (n.d.). Local-branching-on-the-cheap. https://git-scm.com, accessed on 2026-02-15.

[8] Node NPM package “yo”, https://www.npmjs.com/package/yo, accessed on 2026-03-16.

[9] Node NPM package Yo Code - Extension and Customization Generator, https://www.npmjs.com/package/generator-code, accessed on 2026-03-16.

[10] D3.5 – Report on the final iteration of the application model and APIs, 2025, TaRDIS project, https://project-tardis.eu/download/d3-5-final-report-on-programming-model-and-apis

[11] Oracle, Java Downloads, https://www.oracle.com/java/technologies/downloads, accessed on 2026-03-16.

[12] Node, Node.js homepage, https://nodejs.org/en, accessed on 2026-03-16.

[13] Apache, the Apache Maven Project, https://maven.apache.org, accessed on 2026-03-16.

[14] Python, https://www.python.org, accessed on 2026-03-16.

[15] Docker, https://www.docker.com, accessed on 2026-03-16.

[16] ESLint, https://eslint.org, accessed on 2026-03-16.

[17] D5.3 – Final report on Distributed AI and AI-based orchestration, 2025, TaRDIS project, https://www.project-tardis.eu/download/d5-3-final-report-on-distributed-ai-and-ai-based-orchestration

[18] Flower – The Industry Standard for Enterprise-Grade Federated AI, https://flower.ai, accessed on 2026-03-16.

[19] ACM, “Guide to Software Operations - Exploring the Crucial Role of Operators in Software Operations”, 2025. Accessed: Oct. 09, 2025. [Online]. Available: https://www.computer.org/resources/software-operations-guide, accessed on 2025-10-09.

Funded by the European Union

[20] Washizaki, H., “Guide to the Software Engineering Body of Knowledge”, v4-0a, IEEE Computer Society Press, 2025. https://computer.org/swebok accessed on 2026-02-15.

[21] Project Management Institute, “A Guide to the Project Management Body of Knowledge - PMBOK Guide, 7th Edition”, Project Management Institute, 2021. https://www.pmi.org/standards/pmbok, accessed on 2025-10-09

[22] Royce, W., “Managing the development of large software systems: concepts and techniques”, in *Proceedings of the 9th International Conference on Software Engineering*, Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338. DOI: https://doi.org/10.5555/41765.41801

[23] Farley, D., “Modern Software Engineering: Doing What Works to Build Better Software Faster”, Pearson Education. 2022.

[24] Sommerville, I., “Software Engineering”, 10th ed. Pearson Education, 2016.

[25] ISO, “ISO/IEC/IEEE 24765:2017 Systems and software engineering — Vocabulary”, 2017. https://www.iso.org/standard/71952.html, accessed on 2025-10-09.

[26] Beck, K., *et al.*, “Manifesto for Agile Software Development”, 2001. https://agilemanifesto.org/, accessed on 2025-10-09.

[27] Acciarini, G., Güneş, A., Izzo, D., “Closing the gap between SGP4 and high-precision propagation via differentiable programming”, Acta Astronautica 226 (2025) 694–701, https://doi.org/10.1016/j.actaastro.2024.10.063

[28] Popovic, M., Kastelan, I., Djukic, M., Basicevic, I., “A Federated Learning Algorithms Development Paradigm” In: Engineering of Computer-Based Systems. ECBS 2023. Lecture Notes in Computer Science, vol 14390. Springer, Cham. https://doi.org/10.1007/978-3-031-49252-5_4

[29] Jesus, D., Leitão, J., “A Generic Framework for Building Dynamic Decentralized Systems (GFDS)”, draft-jesus-gfds-00, https://datatracker.ietf.org/doc/draft-jesus-gfds/00/ accessed on 2025-02-28.

[30] Savic, M., Atanasijevic, J., Jakovetic, D., Krejic, N., “Tax evasion risk management using a Hybrid Unsupervised Outlier Detection method”. Expert Syst. Appl. 193, 2022, https://doi.org/10.48550/arXiv.2103.01033

[31] Xu, J., Wu, H., Wang, J., Long, M., “Anomaly Transformer: Time Series Anomaly Detection with Association Discrepancy”. arXiv preprint arXiv:2110.02642, 2022, https://doi.org/10.48550/arXiv.2110.02642

[32] Spantideas, S., Giannopoulos, A., Kaltakis, A., Assimakis, N., Trakadas, P., “D-Exit: A Decentralised Early Exit of Inference Framework for Swarm Systems”, 15th International Conference Dependable Systems, Services and Technologies Greece, Athens, December 19-21, 2025, presented and accepted for publication.

Funded by the European Union