COoperative Cyber prOtectiOn for modern power grids

D4.1 COCOON Development Blueprint

Distribution Level	PU
Responsible Partner	UCY
Prepared by	Philippos Isaia, Irina Ciornei, Dimitris Theocharides, Filip Holik, Georgios Kryonidis, Hymanshu Goyel
Checked by WP Leader	UCY
Verified by Reviewer #1	Angelos Marnerides (UCY) 13/05/2024
Verified by Reviewer #2	Maria Michael (UCY) 13/05/2024
Verified by Reviewer #3	Awais Aziz Shah (UGLA) 16/05/2024
Verified by Reviewer #4	Charis Demoulias (AUTH) 16/05/2024
Approved by Project Coordinator	Angelos Marnerides (UCY) 16/05/2024

Co-funded by the European Union

Disclaimer

Deliverable Record

Planned Submission Date	17/05/2024
Actual Submission Date	16/05/2024
Status and version	FINAL

Version	**Date**	**Author(s)**	**Notes**
0.1 (Draft)	12/04/2024	Irina Ciornei (UCY)	ToC, Initial Structure – allocation of sections to partners
0.2 (Draft)	19/04/2024	Philippos Isaia, Irina Ciornei (UCY)	Chapter 3, and updates in all sections
0.3 (Draft)	25/04/2024	Filip Holik (UGLA), Hymanshu Goyel (TUD), Irina Ciornei, Philippos Isaia (UCY), Georgios Kryonidis (AUTH)	Contributions to all chapters
0.4	8/05/2024	Irina Ciornei, Philippos Isaia, Dimitris Theocharides (UCY), Filip Holik (UGLA), Georgios Kryonidis (AUTH)	Updates in all chapters
0.5	10/05/2024	Irina Ciornei (UCY)	Executive summary, conclusions, review of the full content of 1st full draft.
0.6	14/05/2024	Angelos Marnerides, Maria Michael (UCY)	Comments to 1st draft
0.6	15/05/2024	Irina Ciornei (UCY) and all contributors	Integration of all reviews and final edits
0.7	16/05/2024	Awais Aziz Shah (UGLA), Charis Demoulias (AUTH)	Comments to 2nd draft
1.0 (Final)	16/05/2024	Angelos Marnerides (UCY)	Final quality check for submission

Table of contents

Definition of Acronyms

AD	**Anomaly Diagnosis**
API	Application Programming Interface
AUTH	Aristotle University of Thessaloniki
BPF	Berkeley Packet Filter
CD	Continuous Delivery/Deployment
CI	Continuous Integration
COCOON	COoperative Cyber prOtectiON for modern power grids
COMML	COntrol Measurements and Monitoring Layer
CNN	Convolutional Neural Network
CPN	COCOON Programable Node
CSL	Cybersecurity Services Layer
CTD	COCOON Toolset Dashboard
DevOps	Development and Operation Process
DL	Deep Learning
DPDK	Data Plane Development Kit
DRL	Deep Reinforcement Learning
DRY	“Don't Repeat Yourself”
EPES	Electric Power and Energy Systems
FDII	False Data Injection Identification
GA	Grant Agreement
GitLab	Git based Version Control Software
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IOL	Instrumentation and Orchestration Layer
IT	Information Technology
JSON	JavaScript Object Notation
KIOS CoE	KIOS Innovation Centre of Excellence
LSTM	Long Short-Term Memory
ML	Machine Learning
MoSCoW	“Must have, Should have, Could Have, Won’t have”
MR	Merge Request
MS	Microsoft
NF	Network Function
OS	Operation System
OT	Operational Technology
P4	Programming Protocol-independent Packet Processors
PEP 8	Python Enhancement Proposals version 8.0
PV	Photovoltaic
QA	Quality Assurance
QoS	Quality of Service
REST	REpresentational State Transfer
RNN	Recurrent Neural Network
SC	Source Codes
SDN	Software-Defined Network
TCP	Transmission Control Protocol
TRL	Technology Readiness Level

TUD	Technical University of Delft
UCY	University of Cyprus
UGLA	University of Glasgow
UI	User Interface
URI	Uniform Resource Identifier
VM	Virtual Machine
WP	Work Package
XDP	eXpress Data Path

List of Figures

List of Tables

Executive Summary

Increased digitalization and automation merges Information Technology (IT) with Operational Technology (OT) for optimizing the operations of Electrical Power and Energy Systems (EPES). Furthermore, the heterogeneous landscape of stakeholders participating in both generation and newly established Ancillary Services (AS) markets through multiple in number and lower in power capacity Distributed Renewable Energy Sources (DRES), has significantly broadened the cyber-attack spectrum in modern power grids. To address this challenge, the COCOON Project (COoperative Cyber prOtectiOn for modern power grids) adopts an interdisciplinary approach that combines networked systems and power systems engineering research with real-world system operations. COCOON's bottom-up, systems-oriented approach aims to enable the creation of practical data-driven cyber-physical protection mechanisms, capable of operating in real-time when necessary, and enhancing aspects such as cyber-physical anomaly diagnosis, vulnerability assessment, cyber threat mitigation, and trustworthy information exchange.

The herein Deliverable D4.1: *COCOON Development Blueprint* presents the foundations of the framework to be employed within and during the development of the overall COCOON Programable Node (CPN) and cyber-protection services. Hence, this deliverable offers essential software development and deployment guidelines, initial design blueprints, and API function signatures for all three primary architectural layers of the CPN as firstly introduced in the original proposal document. It therefore adheres strictly to the project's bottom-up and system-oriented approach, without directly addressing the CPN system architecture itself, which is the scope of subsequent deliverables.

In particular, D4.1 details the COCOON DevOps process to be utilized by software developers within the project's lifecycle for developing, versioning and testing the envisaged COCOON software prototype such as to ensure that the software prototype reaches the desired Technology Readiness Level (TRL) of 7 as required. The COCOON DevOps process fosters collaboration and communication between software development and operations teams amongst project partners. This collaboration is a crucial element for addressing the complex requirements and challenges associated with adequately designing, implementing and deploying services underpinned by an SDN-enabled power grid data communication paradigm such as the one employed via the CPN programmable data-plane. Moreover, this document also details the abstractions to be utilised for the effective development and orchestration of the COCOON prototype across the three architectural layers of the CPN; namely, the COMML, the IOL and the CSL. Therefore, the high-level functional and non-functional system-level requirements, along with their importance, are included in all three layers abstracting the CPN functionality as part of this initial design phase. In order to appropriately capture the importance for these requirements the MoSCoW requirements prioritization framework is utilized and also discussed.

Furthermore, this deliverable provides an initial introduction to the CPN API software signatures across the three aforementioned layers introducing the fundamental software implementation abstractions. Focus is placed on illustrating how such abstract methods can be extended and utilised to formulate basic cyber protection service instantiation. Hence, in this deliverable two desired corresponding COCOON applications dealing with Anomaly Diagnosis and False Data Injection Identification are selected to demonstrate the practical use of abstract CPN API functions and how cross-layering is achieved through the CPN. Given the early stages of the project, it is worth mentioning that the deliverable outlines requirements relevant to the development of basic COCOON functionalities with respect to software implementation through selected core API software signatures. It is envisaged that the API for each corresponding CPN layer will evolve and be tailored according to requirements to be revealed via the four COCOON pilots.

1 Introduction

The COoperative Cyber prOtectiOn for modern power grids (COCOON) Project employs a multidisciplinary approach to compose a practical and collaborative cyber-physical defence mechanism, which genuinely fuses secure networked systems and power systems engineering research, complemented by system operations. COCOON will integrate a suite of comprehensive scenarios to be tested in actual operational environments of the power grid. Specifically, the COCOON's cybersecurity strategy for Electric Power and Energy Systems (EPES) aims to operate securely and optimally, to facilitate secure cross-domain information sharing, and to cater real-time power systems' needs for prompt cyber-physical threat detection and mitigation as required by the newly formed EU ACER Network Code for CyberSecurity (NCCS)[1]. COCOON follows a bottom-up, systems-oriented methodology which allows the creation of feasible data-driven cyber-physical protection mechanisms, capable of functioning in real-time when necessary. This approach will be thoroughly reflected in this first technical deliverable, which elaborates on the software development lifecycle of the COCOON software prototype and outlines the core COCOON API components and implementation philosophy.

1.1 Scope of the deliverable

This deliverable is the first technical deliverable for the COCOON project, and it kick-starts the reporting of activities carried out as part of Work Package 4 (WP4) - "Software Prototype Development", and specifically of Task 4.1 - " DevOps". It reflects the overall pipeline of the software development lifecycle to be followed in the COCOON project, with a special focus on the DevOps process and how this process will be applied within the experimental validation tests (e.g., using the UCY/KIOS CoE testbed infrastructure) as well as part of the pilot testing and validation phase.

Starting from the initial generic architecture of the COCOON Programable Node (CPN), this deliverable delves into the developing process of its architectural abstractions and blueprints. This methodology is also instrumental in the creation of the CPN APIs at the level of each architectural layer that seamlessly integrate various system components, ultimately aiming at ensuring optimal cross-layering during cyber-protection composition. The bottom-up design fosters a comprehensive understanding of the underlying system, allowing for the development of robust and adaptable data-driven cybersecurity solutions tailored to unique grid-oriented requirements.

Thus, this deliverable provides brief background information on architectural abstractions and blueprints, spanning the three architectural CPN layers: COntrol Measurements and Monitoring Layer (COMML), Instrumentation and Orchestration Layer (IOL) and the Cybersecurity Services Layer (CSL). Lastly, requirements via MoSCoW classification are stated for each of these architectural layers.

As part of indicating the composition of CSL services through the CPN API this deliverable includes hands-on examples of programmatically implementing the False Data Injection and Identification (FDII) tool that will be algorithmically developed in the frame of WP2, discussing its efficient integration with the remaining layers of the CPN node. Similarly, an overview and example of blueprints and software signatures for an Anomaly Diagnosis (AD) tool to be fully developed within WP1 is also provided.

1.2 Relation with other Work Packages and Tasks

This section provides the deliverable's purpose in relation with other tasks and deliverables within the project.

[1]https://eepublicdownloads.blob.core.windows.net/public-cdn-container/clean-documents/Network%20codes%20documents/NC%20CS/20241103_Regulation_(EU)_2019-943.pdf

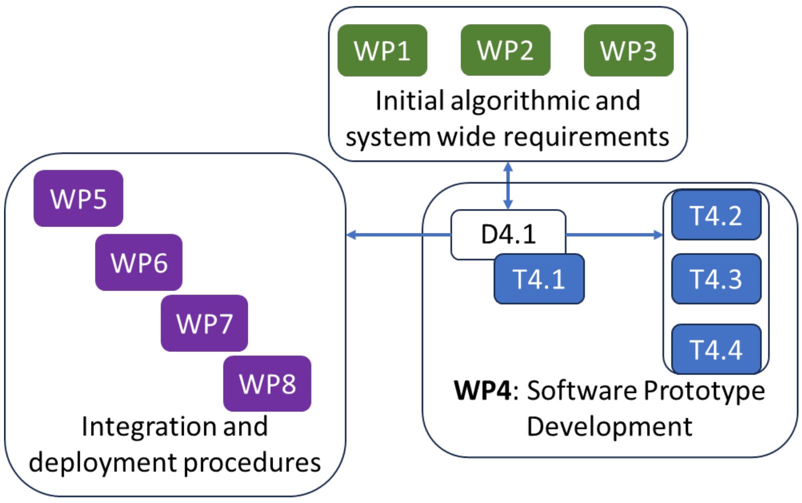

WP1
WP2
WP3
Initial algorithmic and system wide requirements
WP5
WP6
WP7
WP8
Integration and deployment procedures
D4.1
T4.1
T4.2
T4.3
T4.4
WP4: Software Prototype Development

Figure 1: The relationship of D4.1 with other WPs and deliverables

As depicted via Figure 1, D4.1 will serve as initial architectural design strategy for the whole COCOON development process, and it is also the input for several ongoing and future tasks and Work Packages (WPs). Figure 1 Thus, the prototype blueprints elaborated in this deliverable will constitute the input for the rest of the tasks in WP4 (e.g., T4.2 - "CPN System Implementation", T4.3 - "COCOON Services", and T4.4 - "COCOON Toolset", respectively), it has an interrelationship with the work carried out in WP1 to WP3, and it serves as initial guidelines principles for the integration of the COCOON solutions and their pilot-specific deployment procedures which are the scope of WP5 to WP8, respectively.

1.3 Methodology

The methodology employed for creating this deliverable involved two modes of collaborative interaction among partners (offline and online), as well as the integration and consistent collection and analysis of the work performed by all contributing partners as part of Task 4.1. The system-level requirements gathering, and analysis process commenced with their identification based on the initial information from the Grant Agreement (GA). These requirements were re-assessed considering the latest architectural and application-level advancements up to the time of this deliverable's completion, ensuring synchronization with ongoing work in WP1, WP2, and WP3.

An in-depth literature review, along with analysis of best practices in software design and development was also carried out to elaborate on the tailored COCOON DevOps Framework. The UCY KIOS research team leading this effort also used the internal know-how accumulated over several years of successful implementation of this process within other Research and Innovation Projects which also formed the basis for the solutions adopted here. Following the initial elicitation of requirements, numerous online workshops were conducted between the relevant technical partners engaged in the development, deployment, integration, and testing of the COCOON architectural components. The primary objective of these workshops was to agree on a uniform approach to assess the importance of the requirements corresponding to each of the COCOON architectural layers. It was determined that the MoSCoW framework was best suited for this initial requirement assessment, which was necessary for the development of the architectural abstractions and software blueprints.

In summary, the development of this deliverable was a comprehensive and collaborative effort, involving the integration of work from multiple partners and the application of a systematic methodology for requirements gathering and analysis. The use of the MoSCoW framework ensured a consistent and well-informed approach to assessing the importance of the identified requirements, ultimately contributing to the successful development of the COCOON's architectural components and software blueprints.

2 DevOps

2.1 Background

DevOps represents a transformative approach that integrates and enhances collaboration between software development (Dev) and Information Technology (IT) operations (Ops) teams through automation, promoting expedited and high-quality software delivery. This framework was developed in response to the isolated and often variant traditional structures of software creation and IT management, which were typically characterized by poor communication, divergent goals, and extended delivery schedules. By merging development and operational activities, DevOps encourages a seamless, continuous delivery environment that is well-aligned with strategic business goals and the rapid pace of technological progress.

The significance of DevOps in contemporary software engineering is profound. In today's competitive landscape, the ability to quickly update and deploy software is paramount, and DevOps offers essential methodologies to optimize these processes [1]. It introduces systematic practices such as Continuous Integration (CI) and Continuous Delivery/Deployment (CD), wherein code alterations are automatically examined and deployed, thereby markedly diminishing the likelihood of production problems and downtime. These strategies are crucial not only for reducing time to market but also for ensuring the applications' reliability and stability. Additionally, DevOps incorporates automated testing and continuous monitoring to swiftly detect and rectify defects, shifting from traditional reactive methods to a proactive, resilience-oriented approach. The enhanced operational transparency afforded by these practices improves decision-making capabilities across technical and business domains.

CI is fundamental to contemporary DevOps methodologies, crucial for boosting code quality and accelerating development cycles. It entails regularly merging all developers' changes into a common mainline, thus reducing integration challenges and facilitating early detection and resolution of conflicts and defects. Integrating CI into the development process fosters a culture of regular, manageable updates that are consistently tested and validated, using automated builds and tests to provide immediate health feedback for the system, thereby enhancing productivity and reducing integration problems. Additionally, CI promotes a collaborative and transparent working environment, enabling team members to make well-informed decisions swiftly.

Following CI, CD marks the next advancement in software development practices, enabling automated software deployment to production settings. This method enhances automation beyond CI by deploying code modifications to test or production environments post-build automatically. This reduction in manual deployment tasks significantly streamlines the release process, making it quicker and more robust, thus allowing organizations to efficiently deliver new features, fixes, and updates to their clients. Frequent deployments mitigate the risks associated with large-scale releases by opting for smaller, more controlled updates that are simpler to manage and less disruptive. CD ensures constant production readiness through a stringent, automated pipeline that tests, validates, and deploys each change, thereby bolstering the software's reliability and enhancing stakeholder confidence in the development process. By integrating CD, organizations foster a proactive problem-solving culture, accelerate the feedback loop, and uphold a standard of excellence and accountability among development teams.

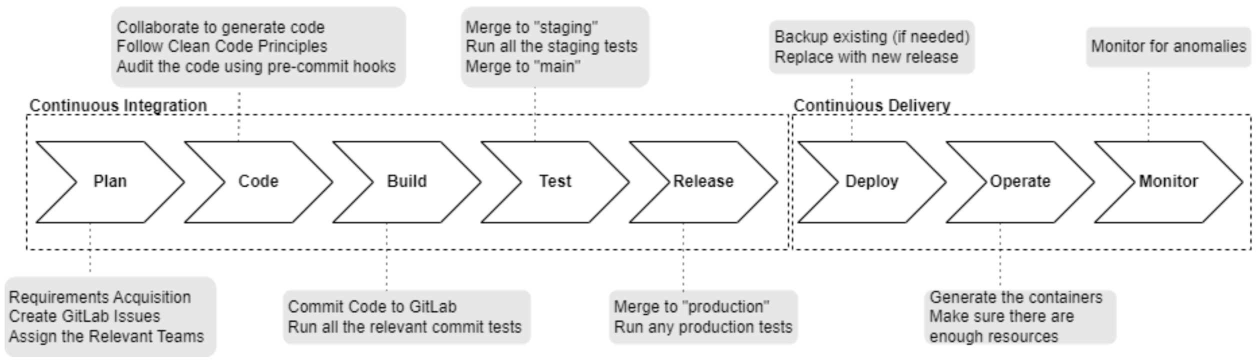


Figure 2 DevOps CI/CD Stages

In summary, Figure 2 gives a visual representation of CI/CD steps and direction), CI/CD includes the following steps:

1. Continuous Integration (CI) Stages
    a. Plan: Involves the initial planning of the software development cycle. It includes defining the project's requirements and setting up tasks.
    b. Code: Involved the developers who collaborate to write code, adhering to clean code principles and using tools like pre-commit hooks to audit the code for quality and standards compliance.
    c. Build: Involves compiling the code into executable or library files, preparing it for testing.
    d. Test: Involves running automated tests to ensure the code meets quality standards and behaves as expected.
    e. Release: Involved the process of code merging into the staging environment, which is as close as possible to the production environment. Here, it undergoes further integration and testing.
2. Continuous Delivery (CD) Stages
    a. Deploy: Involves automating the release of the application into a production environment, ensuring that the transition is smooth and stable.
    b. Operate: Involves the active management of the platform in the production environment, which includes ensuring adequate resources are available and the application is performing as intended.
    c. Monitor: Involves continuous monitoring and aims to detect and address any anomalies or issues that arise in the production environment. This helps maintain the health and performance of the application.

2.2 COCOON DevOps Practices

DevOps practices are designed to automate and streamline the software development and deployment process, ensuring that the delivery is fast, efficient, and of high quality. These practices include but are not limited to version control, automated testing, configuration management, containerization, and release management. Each practice plays a critical role in enhancing the agility and reliability of software development cycles. In the rest of this section, a detailed explanation of each practice used in COCOON will be provided.

2.2.1 Source Code

Source Code is the most critical component for a successful CI/CD pipeline. This involves how the code is written, audited, and stored. For COCOON the following guidelines should be taken into consideration:

1. Writing Code
   a. A style guideline (e.g., PEP8 for Python) should be followed for each programming language used for the codebase to be easier to read and maintain [2].
   b. Each function of class method in the code should be kept focussed, meaning that it should do one operation and do it well. If it performs multiple operations, it should be broken done into several more specific functions. This is done mainly to prevent function repetition and indicated by the DRY ("Don't Repeat Yourself") principle [3].
   c. The code should include clear and useful comments that explain why each function has been implemented. In addition, extensive documentation for complex logic and Application Programming Interfaces (APIs) should be included (for very complex logic, flow diagrams can and should be used).
2. Auditing Code
   a. Pre-commit hooks should be applied to identify issues prior to submitting any code for code review.
   b. Code linting should be included in the pipeline to perform a static code analysis to find any programmatic and stylistic errors. Code linting should be performed after any commit has been pushed to the codebase as well when a merge request is executed.
   c. Peer code reviews should be performed, by assigning Software Developers/Engineers to review code snippets of colleagues.
3. Storing Code
   a. All the code should be stored in a Git based Version Control Software (GitLab), further described in the Version Control section.
   b. Semantic Versioning 2.0.0 [4] should be used in all the software components and the whole platform to prevent any issues with incompatibilities.

2.2.2 Version Control

Version control is the management of changes to documents, software, and other collections of information. It is essential in DevOps for tracking revisions and ensuring that all team members are working on the latest version of a file. For the COCOON project, GitLab will be used for version control, and the following points act as guidelines.

1. Smaller and more frequent commits should be used to reduce complexity in merging and troubleshooting. This practice helps in isolating issues quickly by identifying the exact commit that introduced a problem.
2. Clear and descriptive commit messages that explain the rationale behind changes should be used. The Conventional Commits 1.0.0 [5] standard should be used for structured and consistent messages. To enhance traceability, linking commits to specific issues using GitLab's crosslinking features, such as mentioning an issue number in commit messages should be used.
3. A branching model that aligns with COCOON workflow should be used. GitLab Flow [6] is ideal for managing releases through dedicated channels, while the feature branch workflow supports continuous integration by isolating new changes in separate branches until they are ready to merge. More information for GitLab Flow is provided in the GitLab Flow Section.
4. Merge requests for code reviews before merging changes into the "staging" / "main" branches are essential. This ensures that all code is reviewed and tested, maintaining the quality and stability of your application. Detailed descriptions and tagging relevant reviewers are encouraged to facilitate effective reviews.
5. *main* and "*production*" branches should be protected to control who can commit directly to these branches. GitLab's protected branches feature should be utilized to restrict push and merge access to these branches to authorized personnel only, typically senior developers or CI/CD managers. This helps maintain the integrity and stability of the software and important branches.

6. Feature or other development branches should be regularly updated by merging changes from the *main* branch. This practice reduces the chances of conflicts during the merge-back process and keeps the feature branches up to date with the latest codebase developments.
7. GitLab's tagging feature should be used to mark release points in the repository. Tags can help to keep track of different versions of releases, manage deployment stages, and rollback to previous stable states if necessary. Semantic Versioning mentioned in the Source Code Section should be implemented in tagging.
8. Clear and comprehensive documentation should be maintained within the repository. This should include README files, setup guides, contribution guidelines, and any other documentation that helps team members understand and work with the project. Keeping documentation in the same repository as the code ensures that updates to documentation are synchronized with corresponding changes in the code.

2.2.3 Automated Testing

Automated testing is critical in DevOps to ensure that the new code does not break or degrade existing functionality, allowing teams to quickly identify problems. Thus, it helps in maintaining high code quality and operational efficiency. ***For the COCOON project, the following points should act as guidelines for automated testing.***

1. Clear Testing Standards should be established for what constitutes sufficient test coverage. This includes defining the required metrics, such as code coverage percentages, that must be met before code can be promoted through the CI/CD pipeline.
2. Automated testing should take place both locally (i.e., in the software developer/engineer development environment during code generation) as well as in the CI/CD pipeline. Using GitLab CI/CD pipelines, a predefined set of tests must run every time code is pushed to any branch, especially before merging into main branches like *staging*, *main*, *production*. This ensures that tests are run automatically, providing immediate feedback to developers on the impact of their changes.
3. Various testing types should be utilized:
   a. Unit Testing, to test individual components for correctness in isolation, using frameworks appropriate to the technology stack (e.g., *pytest* for Python[2]).
   b. Integration Testing, to ensure that combined parts of the application work together as expected.
   c. End-to-End Testing, to simulate user scenarios to verify the flow of the application from start to finish, ensuring all integrated components function correctly together.
   d. Performance Testing, to ensure that new code does not degrade performance. This should be performed in the same environment under the same circumstances to ensure that the environment is not impacting the performance.
4. Test Data Management is important to ensure repeatability and remove manual setup errors. Predefined scripts should be used to load and reset databases to a known state before tests run, ensuring consistency across test runs.
5. Mocking and Virtualization tools should be used to simulate external dependencies. This includes hardware, APIs and services to ensure tests can run in a self-contained environment without external dependencies. Furthermore, this approach helps in achieving reliable and faster test results.
6. Flaky tests that produce inconsistent results should be identified and addressed. These tests can undermine confidence in the testing process and should be either fixed or removed from the main test suite until their reliability can be assured.

[2] https://docs.pytest.org/en/8.2.x/

7. Test results should be reviewed and analysed regularly to identify trends and areas of improvement.
8. The deployment procedures should also be tested to ensure that the deployment scripts themselves do not become a failure point. Specifically, this process shall include testing rollback procedures.

2.2.4 Configuration Management

Configuration management is the maintenance of hardware and software systems in a desired, consistent state. It's especially crucial in COCOON DevOps to automate and streamline the provisioning and management of infrastructure. The following points will act as guidelines:

1. The configuration should be defined as "code", enabling consistent environments across development, testing and production. Furthermore, it enables automated deployments. The configuration code should be stored in the same version-controlled repositories used to store the rest of the code.
2. The configuration processes should be automated using scripts that can handle all aspects of configuration, including setting up servers, installing software, and configuring software elements. This ensures that the infrastructure can be recreated at any time with minimal effort.
3. Environments such as development, testing, staging, production, and others, should be kept as similar as possible, but with the necessary differences controlled and documented. This practice reduces issues such as some services working in development but not in production. Thus, it increases the reliability of continuous integration and deployment processes.
4. Roles for who can make changes to the infrastructure and configurations should be defined and enforced. Role-based access control helps in minimizing the risk of unauthorized changes and ensures that only qualified personnel can make critical changes.
5. Configurations should be audited regularly, to ensure that they comply with the project policies and security guidelines.
6. The configuration management processes should be fully integrated with the CI/CD pipelines. This integration helps in rolling out changes in a controlled and systematic manner, allowing for quick rollbacks if necessary.
7. A detailed documentation of the configuration management processes, policies, and configurations should be maintained. The documentation should include architecture diagrams, configuration settings, troubleshooting guidelines, and rollback procedures. This helps in quick onboarding of new team members and serves as a reference in case of issues.

2.2.5 Containerization

Containerization is a crucial aspect of DevOps, providing a standardized unit of software that packages code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. Here are some guidelines for effectively implementing containerization in the COCOON project:

1. Docker[3] should be used as the container platform since it has a vast ecosystem and support and suits all the needs of the project.
2. Minimal and efficient Base Images should be created (e.g., using Alpine Linux[4]) to minimize vulnerability surface and improve deployment times.

[3] https://www.docker.com/

[4] https://www.alpinelinux.org/

3. Application binaries and dependencies should be organized into layers to leverage Docker's caching[5] mechanisms effectively, which can significantly reduce build and deployment times.
4. Only the necessary packages and files should be included in the containers to reduce size and minimize security risks.
5. Environment variables should be used for configuration settings rather than hard coding them into the container image. This practice allows the same container to run in different environments (development, test, production), making your containers more portable and secure. For extra security, "Docker Secrets"[6] should be used in production.
6. Container Orchestration should be used to manage the lifecycle of containers. This can be a simple solution such as Docker Compose or a more advanced one such as Kubernetes[7] which can also help in scaling containers, maintaining container health, and managing service discovery among other benefits.
7. Building, testing and deployment of container images should be part of the CI/CD pipelines. This includes container security scans and compliance checks that will ensure that all the containers are secure and up to standards before deployment.
8. Containers should be designed to be stateless wherever possible. Any state or data that needs to be persistent should be stored in external storage systems such as databases or cloud storage services. This approach enhances the scalability and reliability of applications.
9. Strong security practices should be implemented, such as:
   a. Running the containers with the least privilege necessary.
   b. Scanning images for vulnerabilities regularly. (Snyk[8], Zed Attack Proxy[9], etc.)
   c. Using secrets management tools like HashiCorp Vault[10] to manage sensitive data.
   d. Ensuring network security policies are in place to control the traffic between containers.
10. Tools such as Prometheus[11] and Grafana[12] should be utilized to track the performance and health of containers. Monitoring helps in proactive issue resolution and performance optimization.
11. Clear documentation on how to build, run, and manage containers within the project should be provided. Furthermore, guidelines for troubleshooting common issues and best practices for deploying and maintaining containers should also be included.

2.2.6 Release Management

Release management in DevOps is a strategic process that encompasses planning, scheduling, and controlling the development, testing, and deployment of software releases. It's crucial for ensuring that updates are delivered in a controlled and systematic manner, aligning product development with business objectives. Here are comprehensive guidelines for effective release management in the COCOON project:

1. Release Policies should be established and be clear. They should include criteria for release readiness, approval processes, and rollback procedures. These policies should be agreed upon by both the development and operations teams to ensure consistency and compliance.

[5] https://docs.docker.com/build/cache/
[6] https://docs.docker.com/engine/swarm/secrets/
[7] https://kubernetes.io/
[8] https://snyk.io/
[9] https://www.zaproxy.org/
[10] https://www.vaultproject.io/
[11] https://prometheus.io/
[12] https://grafana.com/

2. Automated release tools should be utilized (GitLab CI) to automate various stages of the release process from code integration to deployment, reducing human error and increasing efficiency.
3. Environment management should be established, to maintain separate environments for development, testing, staging, and production. Each environment as mentioned in the Containerization Section must be as identical as possible to avoid issues that arise from environmental differences.
4. A regular release schedule should be implemented and by aligned with COCOON goals and deliverables. Regular, smaller releases can reduce risks compared to larger, infrequent releases. This approach also allows for quicker feedback from end users and the ability to iterate more rapidly on product features.
5. Release management involves various stakeholders including developers, testers, operations staff, research, and administrative personnel. All these teams should be aligned with the release schedule and requirements. Regular meetings and updates are essential for keeping all parties informed.
6. Monitoring tools (Prometheus and Grafana) should be used to track the performance and health of the application post-release. This data should be used to evaluate the success of the release and to plan improvements for future releases.
7. A detailed plan for rollback should be always in place. This means ensuring that we can quickly revert to a previous version of the application if a release goes wrong. Automated rollback capabilities in the deployment tools (GitLab CI) can facilitate this process.
8. A comprehensive documentation of each release process, including what was released, when, by whom, and any issues encountered should be maintained. This documentation is crucial for audit purposes and for continuous improvement.
9. A Continuous Feedback loop should be established with stakeholders to gather insights on the impact of releases. This feedback should be used to improve the quality and effectiveness of future releases.

2.2.7 GitLab Flow

GitLab Flow [6] is a flexible, branch-based workflow that enhances the traditional GitFlow [7] and GitHub Flow [8] by incorporating issue tracking, environment-focused branches, and a strong emphasis on release management. This workflow is designed to accommodate the DevOps practices and CI/CD pipelines more effectively, making it a robust choice for COCOON which requires systematic control over production releases.

A clear structure with dedicated branches for staging and production allows for more controlled and gradual deployments. This approach reduces risks associated with direct deployments to production. The workflow is designed to integrate seamlessly with GitLab's built-in CI/CD features, facilitating automated builds, tests, and deployments directly within the workflow, thus enabling all the parts of the CI/CD pipeline. By separating production and staging branches, COCOON DevOps team can ensure that only fully tested and reviewed code makes it to production, thereby increasing the stability and reliability of releases. Furthermore, starting with issues for every task creates a clear audit trail of what changes have been made, why they were made, and who made them, improving project management and accountability. Using semantic versioning and tagging for releases makes it easy to identify and rollback to previous versions if a deployment introduces issues.

2.2.8 Use Case: GitLab Flow in COCOON

In the COCOON project we will need to implement several REST API endpoints for data retrieval. Using one of those endpoints as a use case we can indicate how GitLab Flow will be used.

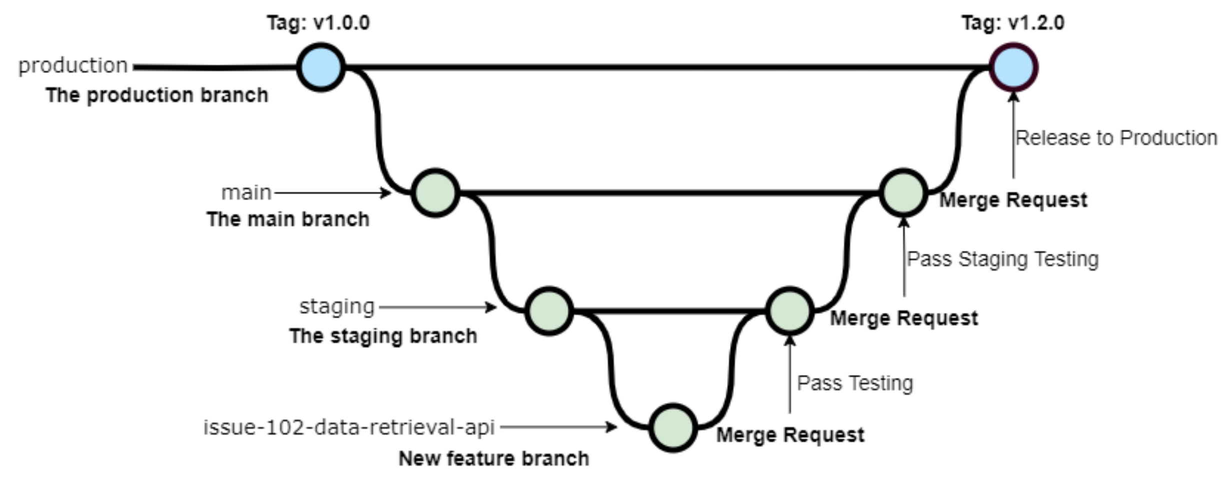


Figure 3 GitLab Flow Use Case for COCOON

Figure 3 provides a summary of the procedure and the process will be as follows:

1. An issue is created titled "*Implement Data Retrieval API*".
2. A developer creates a branch named "*issue-102-data-retrieval-api*" from the "*main*" branch.
3. The feature is developed, and a Merge Request (MR) is created targeting the "*staging*" branch.
4. After review and successful CI tests, the "*issue-102-data-retrieval-api*" branch is merged into "*staging*" (i.e., MR is completed) and deployed to the staging environment for further testing.
5. Once validated in the staging environment, the changes are merged into the "*main*" branch which essentially acts as a pre-production branch.
6. Once it is time to push features to production according to the release plan, the "*main*" branch is merged into the "*production*" branch and deployed. The production deployment triggers a new tag (e.g., "v1.2.0"), reflecting a minor version increase due to new functionality.

Using GitLab Flow, the COCOON project can manage complex releases with multiple stages of testing and review, ensuring high-quality outputs and systematic deployments. This workflow aligns well with the nature of COCOON that require rigorous testing and deployment strategies due to the critical nature of its applications.

2.3 COCOON CI/CD Pipeline

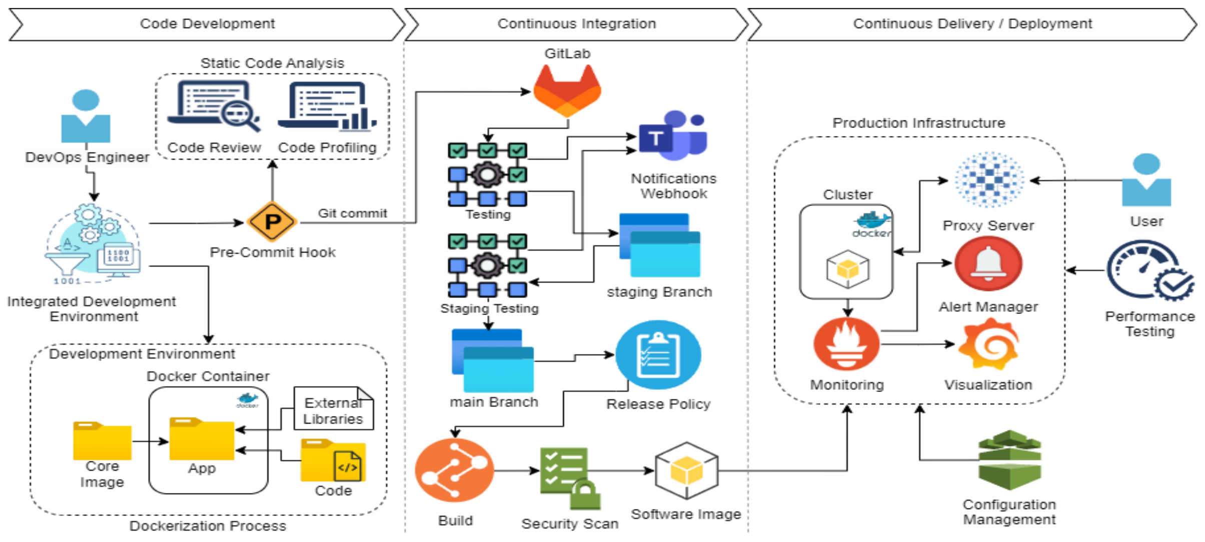


Figure 4 COCOON CI/CD Pipeline

Based on the guidelines provided in the COCOON DevOps Practices section, the initial version of the COCOON CI/CD Pipeline (Figure 4) can be summarised in the following steps:

1. DevOps Engineers write software and infrastructure code using an Integrated Development Environment (IDE).
2. The development environment used during the code development phase is generated using Docker Containers, which helps in ensuring consistency across different development settings.
3. Once the code is ready to be committed, a git-commit command initiates a Pre-Commit Hook. This hook performs static code analysis, including code review and profiling, to catch errors early.
4. Upon passing the pre-commit hook, the code is committed to GitLab where various tests are triggered. Any notifications generated during testing (successes, failures, etc.) are pushed to Microsoft (MS) Teams[13] via a webhook.
5. If the tests are passed, the feature branch is merged into the *staging* branch.
6. According to the *staging* to *main* branch guidelines, a new merge request from *staging* to the *main* branch will be triggered.
7. A new set of tests called the “staging tests” will be executed. Any notifications generated during testing (successes, failures, etc.) are pushed to MS Teams via a webhook.
8. If the tests are passed, the *staging* branch is merged into the *main* branch.
9. The Build process will be triggered according to the release policy. This process compiles the code into a software image.
10. The software image passes through a security scan before it gets the ready for deployment flag.
11. The built image is deployed to the production environment.
12. A reverse proxy in the production environment manages user requests, providing an additional layer of security and load management.
13. Prometheus, Alert Manager, and Grafana are used for real-time monitoring and alerting of the production infrastructure.
14. The system's performance is assessed at predefined intervals to ensure it meets required standards.

2.4 COCOON Release Policy

The Release Policy is designed to govern the planning, scheduling, and implementation of software releases for the COCOON project. The goal is to ensure that all deployments are conducted in a controlled, repeatable, and efficient manner to maintain high quality and stability of the production environment. It applies to all software releases, including new features, bug fixes, performance enhancements, and security updates, across all platforms and environments managed within the COCOON project framework.

2.4.1 Release Scheduling

A detailed release calendar will be maintained and shared with all stakeholders, highlighting the planned release dates, freeze periods, and code cut-off dates. This calendar will be reviewed and adjusted based on business needs and project milestones. The releases are divided into three categories:

1. Major Releases: Scheduled quarterly, these releases include significant feature additions and changes to the application. They require extensive testing and approval.
2. Minor Releases: Conducted monthly, these include incremental feature improvements and minor bug fixes that enhance functionality without drastic changes to the system.

[13] https://www.microsoft.com/en-us/microsoft-teams/

3. Patch Releases: Issued as needed, typically bi-weekly or to respond to urgent issues. These focus on critical bug fixes and security patches.

2.4.2 Approval Process

Releases must be approved by both the development team leaders and the technical coordinator to ensure alignment with technical standards and operational stability. The approval process will be based on the following criteria:

1. Completion of all scheduled tests with results meeting predefined success criteria.
2. Documentation review by the Quality Assurance (QA) team to ensure all changes are accurately captured and compliant with standards.
3. Final sign-off by the technical coordinator or the person responsible for the technical delivery of the platform.

During the approval process, a readiness review will be conducted, mainly by the technical coordinator. This readiness review will include:

1. Verification of feature completion and bug fixes against release requirements.
2. Assessment of test results for coverage and defect resolution.
3. Review of operational and support documentation for updates reflecting the new release.

2.4.3 Environment Definitions

For all the releases, the following environments will be used:

1. Development Environment: Used by developers for coding and initial testing, not reflective of production settings.
2. Testing Environment: Mirrors the production environment as closely as possible where all integration and performance tests are conducted.
3. Staging Environment: A pre-production environment used for final user acceptance testing and to simulate production deployments.
4. Production Environment: The live environment where releases are deployed after thorough testing and approvals.

Releases must be promoted through the environments in sequential order, with mandatory validation checks at each stage to ensure compatibility and performance integrity.

2.4.4 Communication Plan

Regular updates will be communicated through meetings, emails, and a dedicated MS Teams channel. These updates will cover release progress, potential risks, and key decisions. Stakeholders, including clients and end-users, will be informed about upcoming releases, expected features, and potential system downtimes through an official email from the COCOON technical team.

2.4.5 Monitoring and Documentation

Tools such as Prometheus and Grafana will be employed to monitor the system's performance and stability post-release. Any anomalies or degradation in service will trigger an immediate review. In addition, all issues identified during post-release monitoring will be documented and reported through an issue tracking system (GitLab). Regular reports will be generated to evaluate the effectiveness of the release process and to identify areas for improvement.

Furthermore, each release should be accompanied by comprehensive documentation, including:

1. Release notes summarizing new features, fixes, and known issues (changelog).
2. Technical guides detailing configuration changes and new capabilities.
3. Rollback procedures in case of deployment failure.

2.5 COCOON Rollback Strategy

The Rollback Strategy provides a structured approach for reversing deployments that do not meet operational or performance expectations in the production environment. The objective is to minimize disruption and maintain service stability through rapid and controlled reversion to previous software versions when necessary. It applies to all software components deployed in the COCOON project across all operational environments.

2.5.1 Pre-Deployment Checks

A repository should be maintained that includes all the software version releases, with metadata including release date, change log, and environment settings. This includes all the documentation indicated in the release policy (i.e., changelog, configuration changes, rollback procedures), as well as backup created before each deployment.

Before the deployment of a new version, automated backup systems that capture and store the current state of the production environment should be initiated. After completion, the backups should be tested for data integrity and application functionality to be verified.

2.5.2 Rollback Procedure

Since the COCOON project will not be deployed at TRL 9 but only up to 7, it is safe to proceed with a manual rollback procedure that will follow the steps listed below.

1. Initial Assessment: Quickly gather a cross-functional team to assess the impact and decide on initiating a manual rollback.
2. Communication: Notify all stakeholders about the issue and the decision to rollback, outlining expected downtimes and impacts.
3. Execution: Manually restore the previous application version using documented procedures. This includes a) restoring databases and services to their former state using backups and b) reverting any changes made to the environment configurations.
4. Verification: After rollback, conduct tests to ensure that the application is functioning as expected and that the issues introduced by the failed release have been resolved.
5. Monitoring: Intensively monitor the application post-rollback to ensure stability and functionality.

2.5.3 Documentation and Post-Rollback Actions

A detailed documentation of all steps taken during the manual rollback process, including times, actions, individuals involved, and any issues encountered should be maintained. This will be reviewed post-rollback in case something goes wrong, and in addition, it can be used as a guideline for future rollback actions.

Furthermore, after the rollback actions have been completed, a thorough review of the failed release should be conducted to identify the root causes of the failed release. Any outcome from the thorough review should be documented and disseminated to the relevant personnel. Furthermore, the effectiveness of the rollback process should be evaluated and updated if necessary.

2.5.4 Communication and Regular Testing

A comprehensive report should be provided to all stakeholders detailing the cause of the failure, actions taken, outcomes, and future prevention strategies. Regular drills for rollback should be scheduled to practice the rollback procedures to ensure all team members are familiar with the processes. Realistic testing scenarios that cover a range of failure modes should be used to ensure robustness of the rollback procedures.

3 CPN architectural abstractions

3.1 Background

Architectural abstractions and blueprints are essential tools in software engineering, serving as high-level representations of complex systems [9]. Abstractions focus on the system's key aspects, omitting unnecessary details to facilitate understanding and communication [10]. Blueprints build upon these abstractions, providing structured designs that guide software development [11]. In cybersecurity software architectures which specifically target critical infrastructures such as the power grid, abstractions might represent for example secure data flow among components, while blueprints detail the placement of encryption modules, access control systems, and secure communication protocols. Hence, ensuring resilience against cyber threats while maintaining operational and efficient the energy management of the grid [12].

To develop the COCOON architectural abstractions presented in this document, we originally focused on the initial COCOON architecture (Figure 5) from the Grant Agreement (GA).

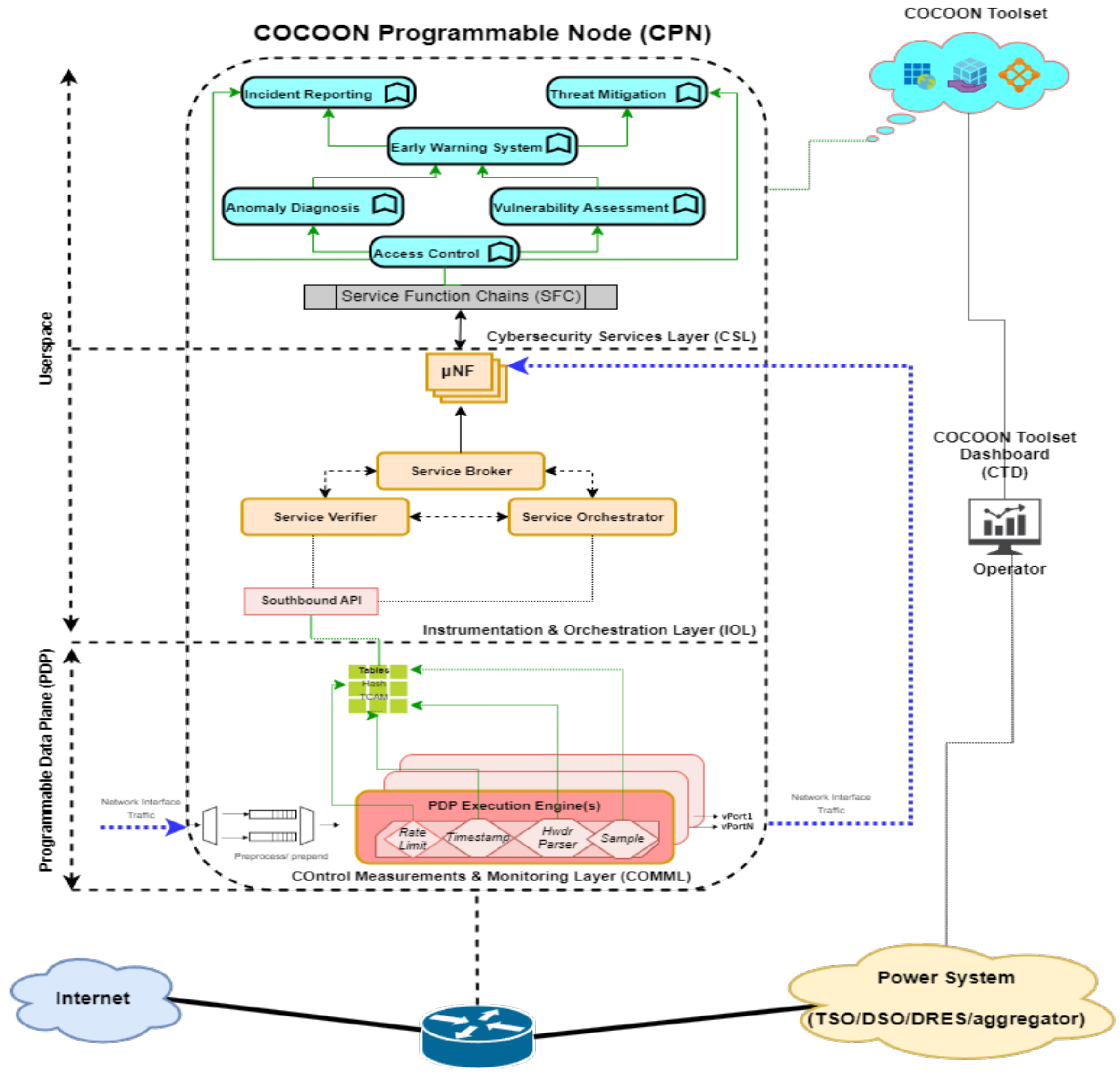


Figure 5 High-level architecture of the COCOON Programmable Node (CPN) enabling cyber protection functionalities to power system operators through the COCOON Toolset and the COCOON Toolset Dashboard (CTD)

We subsequently conducted a decomposition process based on the predefined CPN architectural layers, such as COMML, IOL, and CSL, along with their interaction paths. Specifically, the construction of blueprints for each architectural layer involves a meticulous process that begins with a thorough understanding of the system's cross-layering as well as independent requirements. Each layer's blueprints are carefully designed to ensure that they fulfill their intended role in the system while maintaining a clear separation of concerns. The MoSCoW framework is employed to prioritize the requirements for each layer, ensuring that the most critical features are addressed first.

The MoSCoW framework is a prioritization technique used in requirements analysis, categorizing requirements into four groups: Must have, Should have, Could have, and Won't have [13]. This framework will be used in the context of this deliverable to ensure focus on critical requirements while managing stakeholder expectations. For architectural abstractions explicit to the COCOON architecture, the MoSCoW framework is used to help prioritize especially security-related features. For instance, encryption and access control might be "Must haves," Intrusion Detection (ID) systems "Should haves," advanced threat analytics "Could haves," and biometric authentication "Won't haves" due to cost or feasibility constraints. This approach streamlines the design process, ensuring the most critical security features are implemented effectively. In the context of this deliverable, by using the MoSCoW framework at the level of each architectural is intended to construct blueprints that accurately reflect the priorities and needs of each layer, ensuring a robust, scalable, and efficient architecture that meets the project's objectives and delivers value to the end-users.

As illustrated via Figure 5, the CPN adheres to a cross-layering paradigm and its bottom architectural layer is the COMML. The COMML is responsible for packet processing for measurement, monitoring and control purposes and will be using the eBPF technology (a successor of the Berkeley Packet Filter (BPF)). eBPF allows execution of a custom code within a limited instruction set. Hence. COMML will utilize customized IP packet switching pipeline developed within the BPFabric tool, and it will be supported on Data Plane Development Kit (DPDK)[14] interfaces and OS user space switches.

Right above the COMML lays the IOL being responsible for management of all nodes and at the same time acts as a middleware between nodes and services running in the CSL. The IOL will utilize a reliable Transmission Control Protocol (TCP)-based communication with nodes, which would include messages for eBPF functions installation and removal. The layer will be built on top of the BPFabric controller, and it will provide a northbound API to the CSL layer adhering to an SDN-enabled paradigm.

CSL integrates all the COCOON cyber-protection services which will be deployed and accessed via the COCOON Toolset Dashboard (CTD) through an interface over cloud-based services abstracted within the COCOON Toolset. The CTD will align with operator requirements, while empowering technologies that implement control, monitoring, and measurement elements on the CPN programmable data plane. Services offered via the COCOON Toolset will be subjected to verification processes jointly conducted through the interface of the northbound and southbound CPN API to ensure their trustworthiness and efficient synthesis. Two examples of CSL applications are listed in this deliverable: (i) False Data Injection Identification (FDII), and, (ii) Anomaly Diagnosis (AD). The FDII will be developed and implemented as a dedicated open-source software tool written in Python. The core of the FDII algorithm will use the fundamentals of state estimation techniques [14]. The tool will be developed from scratch by employing well-established Python libraries that will be further analyzed in the following deliverables: a) D2.1*: Generic methodology for power grid state estimation* and b) D2.2: *Report on the refinement of the generic power grid state estimation for PV plants and energy communities*. The AD tool will be developed as part of WP1 (Task T1.4) and it will be detailed as part of the following

[14] https://doc.dpdk.org/guides/prog_guide/bpf_lib.html

deliverables: a) D1.3: *DL-based Diagnosis and DRL-based Threat Mitigation in IT and OT environments* and b) D1.4: *Finalised algorithmic cyber-physical attack diagnosis and mitigation.*

3.2 COCOON blueprints

Figure 6 shows the high-level blueprint of the COCOON architecture which is inspired by properties of software-defined networks (SDNs)[15].

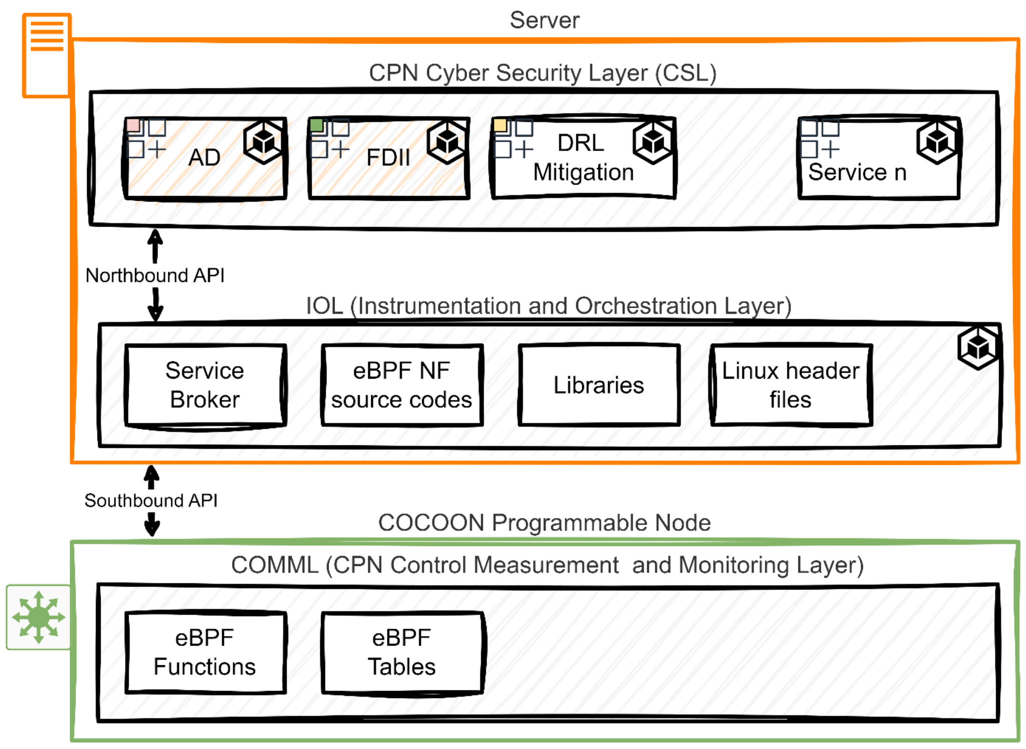


Figure 6 COCOON blueprint architecture indicating some exemplar CSL services

It is to be highlighted that the adoption of an SDN-based architecture offers several key advantages over legacy architectures. Specifically, SDN provides centralized management and control of network devices through a logically centralized controller (or orchestrator, as in the case of COCOON), enabling greater network agility, flexibility, and programmability. This centralization simplifies network configuration, reduces operational complexity, and facilitates the deployment of innovative services. Additionally, SDN enhances network security and visibility by allowing administrators to implement fine-grained policies and monitor network-wide traffic, ultimately leading to improved overall network performance and resource utilization. As already presented in previous sections, the overarching CPN blueprint at the architectural level is composed of three layers: CSL with high-level applications, IOL acting as a middleware, and COMML providing the data plane functionality. Cross-layering is achieved via northbound and southbound APIs between the layers.

The northbound API serves as a critical interface for communication between the control plane and achieves effective cross-layering between low-level functionalities of the CPN and requirements distilled by CSL applications. Within the COCOON architecture the SDN-controller role is served by a more elaborated orchestrator, the IOL, and the application plane refers to the CSL. Through the northbound API applications and network services are facilitated with the necessary capability to

[15] https://opennetworking.org/sdn-definition/

programmatically interact with the COCOON orchestrator, thereby enabling dynamic configuration of network policies and the retrieval of network information when and if required.

By contrast, the southbound CPN API functions as the interface between the control plane (IOL) and the infrastructure or data plane (COMML). This API allows the CPN orchestrator to manage and manipulate network devices, such as switches and routers, by translating high-level instructions into device-specific commands, ultimately ensuring proper forwarding and processing of network traffic.

For the sake of clarity and simplified presentation, Figure 6 illustrates exemplar CSL services such as Anomaly Diagnosis (AD), False Data Injection Identification (FDII) and Deep Reinforcement Learning Mitigation (DRL). These CSL services can be implemented on the same device as IOL, or separately.

3.3 COMML blueprints

COMML is providing data plane functionality and will support packet processing operations such as to forward and drop the packet, or the ability to extract information from the packet header fields or payload, to store this information and provide to the IOL via the southbound API. The functionality will be implemented in customized eBPF functions and the ability to store the information will be implemented with eBPF tables[16] as shown in Figure 6.

The order of eBPF functions will define the processing pipeline for incoming traffic. Functions can perform specific actions on the packets (such as to extract a header field, modify, or drop the packet). Every eBPF function can have one or more eBPF tables for storing data as this is the only mechanism for achieving statefull functionality in eBPF. A function is connected to an event (for example a packet received) and after its code is executed, all data not saved to the table is lost. The tables will be controlled by the IOL.

The following table presents the COMML requirements based on the MoSCoW framework in order to demonstrate clearly the envisaged features.

Table 1 COMML requirements summary and prioritization

Requirement (Name/Description)	**MoSCoW**
Packet processing: Ability to read and write on network interfaces and to parse and de-parse packet information for further analysis.	M
Packet forwarding: Forward the traffic with minimum overhead.	M
Packet control: Ability to define basic packet actions – drop and re-route.	M
Statistics monitoring: Ability to collect and provide various data using Extended Berkeley Packet Filter (eBPF) tables structures.[17]	M
Packet modification: Ability to modify packet header information.	S
Threat mitigation policies: port blocking.	S
Support for Advanced packet queueing features: Techniques for Quality of Service (QoS) and additional threat mitigation – packet throttling, traffic re-shaping, load-balancing.	C
Support for Deep Reinforcement Learning (DRL) threat mitigation algorithm: DRL algorithm for selecting appropriate threat mitigation policy.	M

[16] https://docs.kernel.org/bpf/maps.html
[17] https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h

Southbound API: Communication between IOL and CPN supporting the previous requirements.	M
P4[18] support: Ability to run Programming Protocol-independent Packet Processors (P4) programs won't be utilized as all the required functionalities can be done with eBPF.	W
eXpress Data Path (XDP) Linux Operation System (OS) hooks: Ability to use XDP hooks won't be utilize due to the use of DPDK.	W
Kernel-space Linux modules: Won't be utilized due to the use of DPDK (will use user-space instead).	W

3.4 IOL blueprints

The IOL will serve as the middleware between CSL and COMML and will perform a controller role. It will have a service broker functionality which will be responsible for management of eBPF micro-network functions (NFs) and interaction with both APIs (northbound and southbound). An example of a micro-NF is a forwarding functionality, or a function counting received packets. IOL will provide a platform for installation of custom eBPF source codes, which can then be dynamically deployed by the broker to the nodes to the appropriate slot within the processing pipeline. The layer's functionality will be supported by appropriate libraries (such as Twisted[19] and ProtoBuf[20]) and Linux header files (linux/if_ether.h, linux/ip.h, etc.).

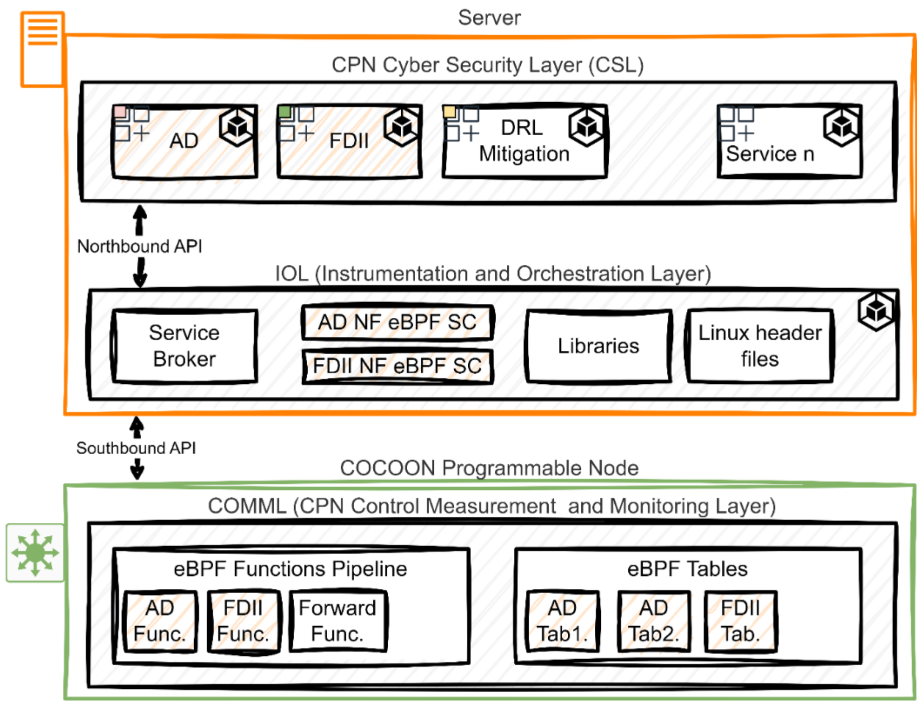


Figure 7 An example of services implementation within the CPN architecture

Figure 7 shows architectural details with two example services – Anomaly Diagnosis (AD) and False Data Injection Identification (FDII). These services will have custom eBPF NFs prepared as source codes (SC). These services will then be installed by the service broker upon request from CSL. The

[18] https://opennetworking.org/p4/
[19] https://twisted.org/
[20] https://protobuf.dev/overview/

broker can also query eBPF tables in COMML and the received data can be subsequently provided to the CSL.

The following table presents the IOL requirements compiled based on the MoSCoW framework.

Table 2 IOL requirements summary and prioritization

Requirement (Name/Description)	**MoSCoW**
Micro-network functions: Ability to utilize custom eBPF functions for support of services in the CSL.	M
User-space Linux modules: Include of required Linux kernel header files.	M
Service function chain: Ability to manage order of execution of micro-network functions.	M
Service broker: Management of micro-network functions and APIs.	M
Processing graphs: Binary verification, trusted execution and memory isolation won't be implemented as it is provided by eBPF.	W
Northbound API: Communication between CSL and IOL.	M

3.5 CSL blueprints

3.5.1 Blueprint for the CSL microservices architecture

Figure 8 provides the architecture of the CSL which will run in one or more virtual machines and will include a series of microservices in isolated Docker containers.

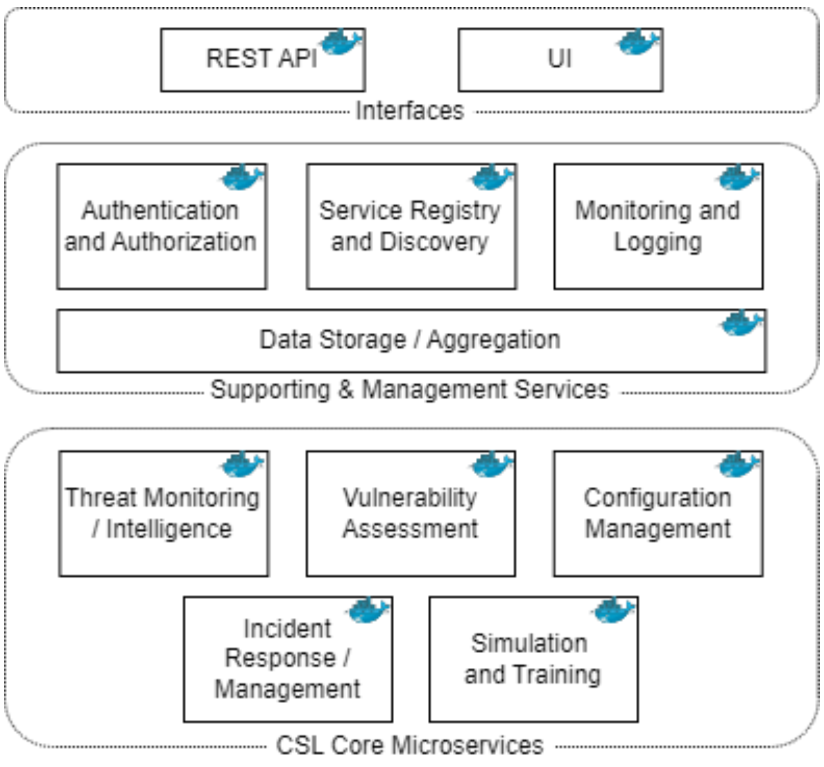

REST API
UI
Interfaces
Authentication and Authorization
Service Registry and Discovery
Monitoring and Logging
Data Storage / Aggregation
Supporting & Management Services
Threat Monitoring / Intelligence
Vulnerability Assessment
Configuration Management
Incident Response / Management
Simulation and Training
CSL Core Microservices

Figure 8 CSL Microservices Architecture

These microservices include the Core CSL services which will perform the core functionality alongside supporting, management, and interfacing services. The latter allow for better monitoring of the CSL

Core microservices and interfacing them with other external to the CSL components. Listed below are all the services in the CSL Architecture:

1. REST API Gateway: It serves as the single entry-point for all clients, directing traffic to appropriate services and handling load balancing and security.
2. User Interface (UI): It provides dashboards for real-time monitoring, risk assessments, and managing incidents and configurations.
3. Authentication and Authorization Service: It ensures that only authorized personnel can access the various components of the CSL, managing credentials and access controls.
4. Service Registry and Discovery: It keeps track of all the microservices in the CSL, facilitating service discovery and management.
5. Monitoring and Logging: It monitors the health of microservices and logs activities and errors, providing insights into the system's operation and helping troubleshoot issues.
6. Data Storage / Aggregation: It stores logs, incident reports, configurations, and other data securely. Different databases will be used for structured and unstructured data.
7. Threat Monitoring / Intelligence Service: It gathers and analyzes threat data from various sources to keep the system updated with the latest security threats and vulnerabilities.
8. Vulnerability Assessment Service: It automatically scans Information Technology (IT) and Operational Technology (OT) systems to detect vulnerabilities and provides reports on potential security weaknesses.
9. Configuration Management Service: It manages and stores configuration settings for IT and OT devices, ensuring that all systems are up to date and configured according to security best practices.
10. Incident Response / Management Service: It handles security incidents by logging details, notifying stakeholders, and coordinating response strategies.
11. Simulation and Training Service: It provides a virtual environment to simulate attacks on cyber-physical systems, and modules for training sessions helping users understand and operate the security features effectively.

Table 3 provides the requirements of the CSL along with their prioritization following the MoSCoW framework.

Table 3 CSL Functional Requirements with MoSCoW Prioritization

Requirement (Name/Description)	**MoSCoW**
Real-Time Threat Data Integration / Monitoring: The CSL must gather and integrate data from external and internal sources to maintain an updated threat database.	M
System Scanning: Scan IT and OT systems to detect vulnerabilities regularly and on demand.	M
Risk Profiling: Assess risk levels based on vulnerability data and known threats, assigning risk scores to critical components.	M
Incident Logging and Tracking: Log all incidents and enable tracking through the entire incident management lifecycle.	M
Access Control: Restrict access to authorized personnel only, based on roles.	M
Customizable Test Scenarios: Allow users to customize test scenarios that replicate real-world attacks.	S
Centralized Configuration Management: Centrally manage configuration changes across IT and OT systems.	S
Customizable Layouts: Provide customizable dashboard layouts for users to tailor their experience.	C
Service Health Checks: Perform regular health checks on services to ensure optimal performance.	C

Comprehensive Metrics: Provide comprehensive metrics on system performance and security posture.	C
Cross-Platform Compatibility: Supporting multiple operating systems environments.	W
Extensive Customization Features: Extensive customization of interfaces and configurations.	W
Full Text Search Capabilities: While useful, implementing a comprehensive search feature across all logged data and historical records will not be included in the initial release.	W
Voice Command Features: Voice command capabilities for hands-free operation will not be developed at this stage.	W

3.5.2 Blueprint for the Anomaly Diagnosis (AD) tool

As already mentioned, the CSL will integrate a suite of cybersecurity services among which there is an Anomaly Diagnosis (AD) tool. The AD is designed as a part of WP1 (Task T1.4) to serve as an early warning and alerting system, and its detailed functionality and development process will be presented in the corresponding subsequent deliverable of the aforementioned Task. For the purpose of this deliverable, we provide a brief understanding on the scope of the AD tool and how the blueprint of it was elaborated during an initial design phase adhering to the core CPN software abstraction properties. Specifically, the AD tool will operate within the CSL at the digital substation level, continuously collecting real-time network data to detect and alert on any possible anomalies. The AD tool will consist of a Deep Learning (DL)-based framework for diagnosis of anomalous traffic in the network.

The AD tool will be developed using Python programming language using state of the art DL and Machine Learning (ML) libraries such as Pytorch[21], scikit-learn[22] and TensorFlow[23]. It would consist of two major algorithms, one for prediction and the other for diagnosis.

1. Prediction model: The prediction-based model will employ a Recurrent Neural Network (RNN) -based solution (e.g., – type Long Short-Term Memory (LSTM)) which will be used to predict the network behaviour for all the devices present in the network.
2. Classification model: The classification model will employ a Convolutional Neural Network (CNN) - based model which will be used to classify events in the network. The model will be trained to classify normal behaviour with anomalous and will take input from the prediction model along with the actual data from the network.

In relation with the CPN system architecture, the AD tool will contain of the following major parts,

1. Network data collection: The model will collect information from the CPN using functions defined in the following chapter of this document for the COCOON's APIs. It will receive real-time packet information from the network which will be processed continuously with minimum time delay in the order of milliseconds (ms).
2. Training and testing: The data collected will be used to train and test the DL models. During this phase, the models are evaluated for accuracy, and adjustments are made as needed. If the training data is deemed insufficient, additional data collection may be required. The training of the model will be done using the Technical University of Delft (TUD) High Performance Computing infrastructure.

[21] https://pytorch.org/docs/stable/library.html
[22] https://github.com/scikit-learn/scikit-learn
[23] https://www.tensorflow.org/resources/libraries-extensions

3. Anomaly diagnosis: The trained detection model will communicate with the CPN in case an anomalous traffic is detected.

Table 4 Anomaly Diagnosis requirements with MoSCoW Prioritisation

Requirement (Name/Description)	**MoSCoW**
Network function: Ability to communicate with CPN to exchange API requests	M
Information storage: Ability to store temporary data from the network locally for further processing.	
Data collection: Ability to receive real-time network traffic from the devices in the network via CPN	M
Anomaly diagnosis report/ Alert: A module to communicate the anomaly diagnosis report to the connected device on the same network every few minutes.	M
Loading trained model: Ability to communicate with remote server (eg Gitlab/GitHub) to download and deploy the trained model	M
Model testing: Ability to test the trained model locally using a defined set of network data	C
Model training: Ability to train the model locally on the defined ML architecture.	C
Mitigation control: A method to stop the attacker from doing further damage to the network	W
Continuous model training: A method to train and update the model continuously with the incoming packets	W

3.5.3 Blueprint for the False Data Injection Identification (FDII) tool

In the frame of WP2, a FDII method will be developed. Its distinct characteristic is the efficient integration of the physical system, i.e., the photovoltaic (PV) power plant (T2.2) and/or the distribution grid where the energy communities are connected (T2.3), in the identification process. This way, an additional security level is introduced that can identify even phenomenally stealthy attacks. The proposed method will be implemented as a dedicated open-source software tool written in Python that will run as a service at the CPN cyber security layer (CSL), as shown in Figure 6. Using the Northbound API (see Figure 6), the FDII tool will acquire a set of electrical measurements needed to fully monitor the operating condition of either the PV power plant (T2.2) or the distribution grid where the energy communities are connected (T2.3). To facilitate the efficient integration of the FDII tool with the Northbound API, a set of requirements are foreseen according to the MoSCoW prioritization listed in Table 5. These requirements mainly involve: (a) the communication protocol used for the data acquisition from the electrical power and energy system (EPES) components, (b) the type of data that will be captured, (c) the sampling rates, etc.

Table 5 FDII Requirements with MoSCoW Prioritisation

Requirement (Name/Description)	**MoSCoW**
Acquire measurements based on IEC 60870-5-104 protocol used in supervisory control and data acquisition (SCADA) systems. SCADA systems are widely used by distribution system operators to monitor and control the main network elements in distribution grids. This system will be also used to control/monitor the energy communities pilot installation in WP5.	M

Acquire measurements based on Modbus communication protocol. This is a well-established protocol used in PV plants to control/monitor each individual inverter. This protocol will be also used in the PV plant pilot installation in WP8.	M
Predefined data capture. The data to be captured will be determined offline and remain constant during the online operation. Ability to capture at least the following: (a) active and reactive power of each PV inverter, (b) the voltage magnitude at its point of interconnection with the grid, (c) the power flowing at the low-voltage side of the high-voltage/medium-voltage transformer.	M
Continuous/live tracking of the captured data with a maximum sampling rate equal to 2 Hz.	M
The data acquisition will be triggered on demand by the CSL, i.e., the FDII, based on the considered ancillary service, reaching sampling rates up to 2 Hz.	S
Configurable/Selective data capture. The data to be captured can vary during the real-time operation and can be modified on demand.	C
Send active and reactive power setpoints either at PV plant or inverter levels via the IEC 60870-5-104 protocol.	C
Data acquisition with sampling rates up to 10 Hz.	C
Communication protocol agnostic FDII method.	W
Data acquisition with sampling rates larger than 10 Hz.	W

4 COCOON API

4.1 Background

An Application Programming Interface (API) is a set of protocols and mechanisms which enables different software components to communicate with each other. They define methods and data formats which the applications use to set or get information, or to perform a specific task.

The COCOON API as decomposed for each layer is crucial to provide communication between the three layers. Hence, three inter-related APIs will co-exist and based on the following technologies:

- Python functions – internal communication will use custom Python functions and data structures.
- REpresentational State Transfer (REST) – external communication will use REST API[24] based on the Uniform Resource Identifier (URI)[25] and the Hypertext Transfer Protocol (HTTP)[26].
- JavaScript Object Notation (JSON) – exchange of more complex data via REST will be realized with JSON format[27].
- Transmission Control Protocol (TCP) communication – communication between control and data plane will be realized with custom SDN-like messages implemented over the TCP.

4.2 COCOON API

The COCOON architecture will inherit properties and adhere to the software interfaces of two existing types of APIs in a similar fashion to the actual CPN architecture: northbound and southbound as shown in Figure 6. These interfaces will handle communication between CSL and IOL (northbound), and between IOL and CPN (southbound). The southbound API must handle communication between different physical devices and will therefore be based on TCP which ensures reliable delivery of messages. The northbound API can be implemented in two different variants based on the final implementation of the CSL:

1. REST API– if IOL and CSL services run on different virtual machines (VMs), containers, different devices, or more advanced type of communication is required, the API should be done via REST API with JSON files for more complex data exchange.
2. Python API – if IOL and CSL run in the same VM and do not require inter-process communication, the API can be done with Python methods and custom objects for more complex data exchange.

4.3 COMML API

The southbound API will use custom developed TCP-based messages sent between COMML and IOL in both directions. The following message types will be implemented:

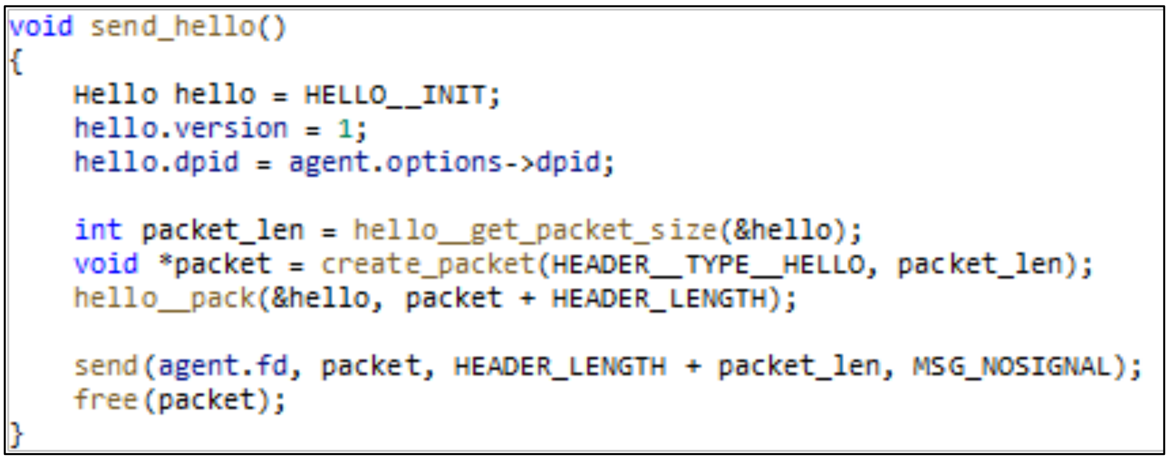
- *Hello* – for nodes discovery, connection establishment and keepalive.
- *Packet in* – for forwarding received packets on the node to the IOL.
- *Packet out* – for sending packets from IOL to the node.
- Function operations – for management of eBPF functions on the node
    - o *Add / remove / list.*

[24] https://www.ibm.com/topics/rest-apis
[25] https://www.techtarget.com/whatis/definition/URI-Uniform-Resource-Identifier
[26] https://www.w3.org/Protocols/
[27] https://www.json.org/json-en.html

- Tables operations – for management of eBPF tables
    - *List table* – will get the table content.
    - *Table entry get / update /* delete.

This API will allow extraction of packet data either via eBPF tables (for selected header fields), or via the entire message forwarding to the IOL.

The *hello* initialization message example is provided in Figure 9 below.

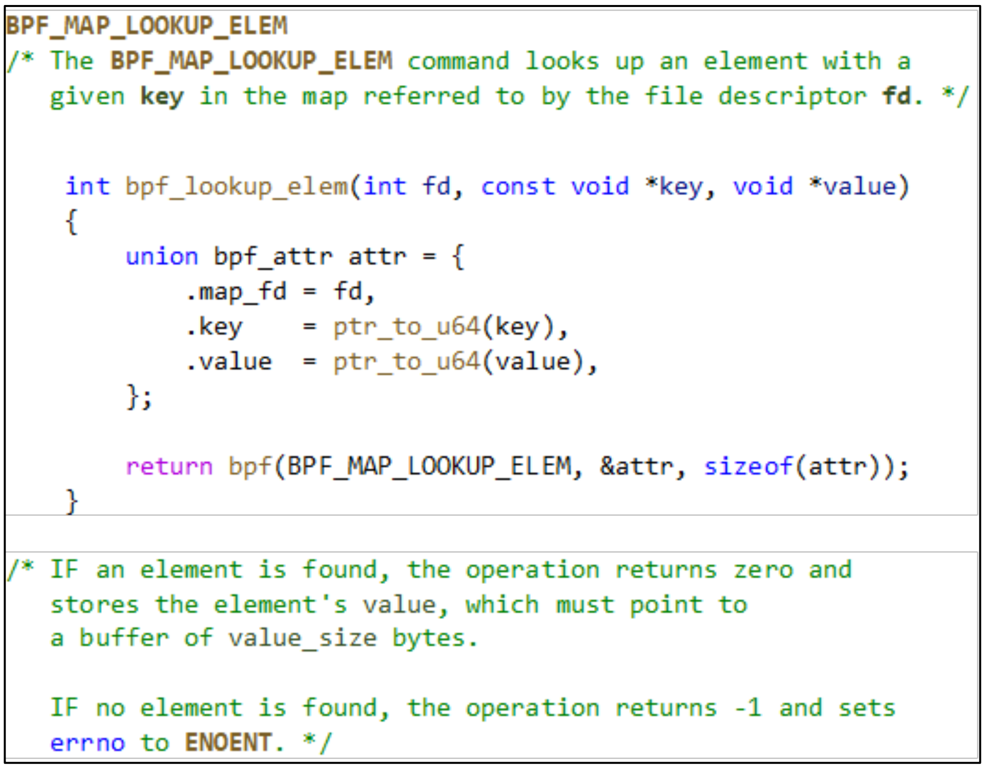


```c
void send_hello()
{
    Hello hello = HELLO__INIT;
    hello.version = 1;
    hello.dpid = agent.options->dpid;

    int packet_len = hello__get_packet_size(&hello);
    void *packet = create_packet(HEADER__TYPE__HELLO, packet_len);
    hello__pack(&hello, packet + HEADER_LENGTH);

    send(agent.fd, packet, HEADER_LENGTH + packet_len, MSG_NOSIGNAL);
    free(packet);
}
```

Figure 9 Example of signature code for the Hello initialization message

Tables operation for reading data will be realized using the *bpf_map_lookup_elem* kernel helper function[28] as shown in Figure 10 below.

[28] https://docs.kernel.org/bpf/map_array.html



```
BPF_MAP_LOOKUP_ELEM
/* The BPF_MAP_LOOKUP_ELEM command looks up an element with a
   given key in the map referred to by the file descriptor fd. */

    int bpf_lookup_elem(int fd, const void *key, void *value)
    {
        union bpf_attr attr = {
            .map_fd = fd,
            .key    = ptr_to_u64(key),
            .value  = ptr_to_u64(value),
        };

        return bpf(BPF_MAP_LOOKUP_ELEM, &attr, sizeof(attr));
    }

/* IF an element is found, the operation returns zero and
   stores the element's value, which must point to
   a buffer of value_size bytes.

   IF no element is found, the operation returns -1 and sets
   errno to ENOENT. */
```

Figure 10 Documentation of the bpf_map_lookup_elem command[29]

4.4 IOL API

The IOL API is the synthesis of the message exchange between the northbound (NB) API and the southbound (SB) API. The IOL API deals with the orchestration of requests exchanged between the NB and SB APIs, and thus acts as a middleware. Exemplar functions include methods for managing eBPF functions and tables:

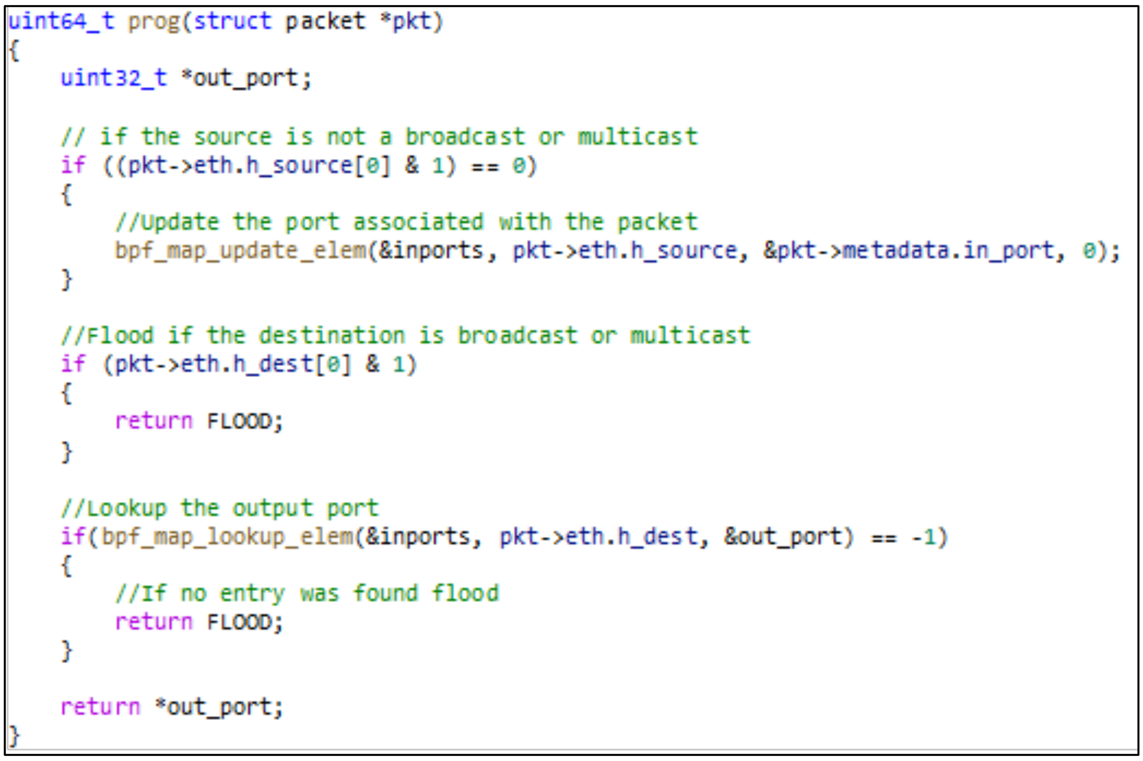
- Function operations – manages eBPF functions
  - *Add / remove / list*
- Tables operations - manages eBPF tables
  - *List table*
  - *Table entry get / update / delete*

Furthermore, there will be COCOON wrapped methods for most common operations, this will include:

- *cocoon_monitor_config*(*string*[] *headerFields*, *int nodeID*) - returns *buffer ID* (*int*).
  - The function will start collecting defined *headerFields* for all traffic on the specified node and will store the data in a buffer.
- *cocoon_monitor_get_data*(*int bufferID*) - returns a structure with configured data received from the last function call (deletes the buffer).
  - The function will collect the data from the buffer and deletes its content.

A code signature example of an eBPF function for simple switch forwarding logic is provided in Figure 11 below.

[29] https://man7.org/linux/man-pages/man2/bpf.2.html



```
uint64_t prog(struct packet *pkt)
{
    uint32_t *out_port;

    // if the source is not a broadcast or multicast
    if ((pkt->eth.h_source[0] & 1) == 0)
    {
        //Update the port associated with the packet
        bpf_map_update_elem(&inports, pkt->eth.h_source, &pkt->metadata.in_port, 0);
    }

    //Flood if the destination is broadcast or multicast
    if (pkt->eth.h_dest[0] & 1)
    {
        return FLOOD;
    }

    //Lookup the output port
    if(bpf_map_lookup_elem(&inports, pkt->eth.h_dest, &out_port) == -1)
    {
        //If no entry was found flood
        return FLOOD;
    }

    return *out_port;
}
```

Figure 11 Example of signature code for the eBPF function for simple switch forwarding logic

4.5 CSL API

The COCOON CSL Layer plays a pivotal role in SDN architectures like the CPN, by providing vital security services and functionalities. In the context of the COCOON Toolset, which develops specific API for control, monitoring, and measurement elements within the programmable data plane, the CSL ensures these tools meet the necessary security standards. This security provisioning includes the implementation of secure communication channels, robust access control mechanisms, and stringent data integrity checks. These measures are designed to protect the network's control, monitoring, and measurement components from potential cyber threats, ensuring they operate securely and comply with established network security policies.

In following sections we provide three comprehensive examples of code signatures for the CSL API.

4.5.1 *Cocoon Toolset Dashboard (CTD)*

The COCOON toolset is designed to define and refine the software architecture for the CPN/CSL, aligning it with the operator requirements. This alignment is facilitated through specific applications available on the COCOON Toolset Dashboard (CTD), which are utilized during the configuration phase to collect user-interface requirements and setup applications. This dashboard is a critical component, serving as the hub for initializing and managing the software tools provided by COCOON.

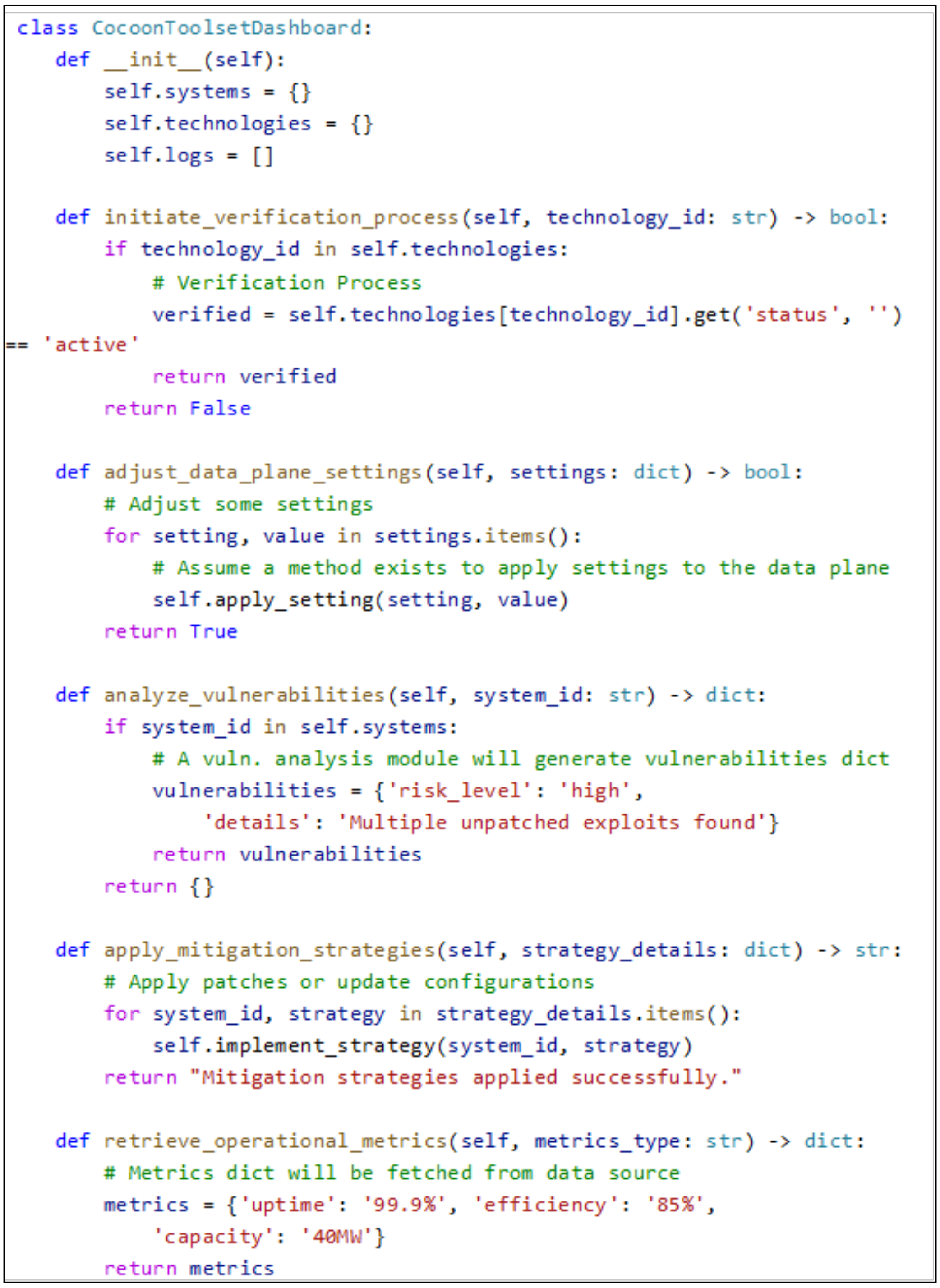
Additionally, the COCOON Toolset enhances grid operator's capabilities by offering sophisticated tools for vulnerability assessment and risk profiling. These tools employ a cyber-physical, graph-based approach to data provenance, which supports both Information Technology and Operational Technology (IT/OT) convergence in Electric Power and Energy Systems (EPES) deployments. This approach significantly improved the accuracy of cybersecurity threat predictions and risk assessments.

For visualization and analysis, the Grafana tool is utilized, enabling an effective display of data received from RabbitMQ[30] via its comprehensive dashboard features. Grafana Loki[31] is also incorporated for efficient log management, providing a robust solution for handling and analyzing log data.

CTD Code Signatures (Shown in Figure 12 and Figure 13) provide a high-level Python structure for some of the most important functionalities of CTD. The following list provides these functionalities with a high-level description:

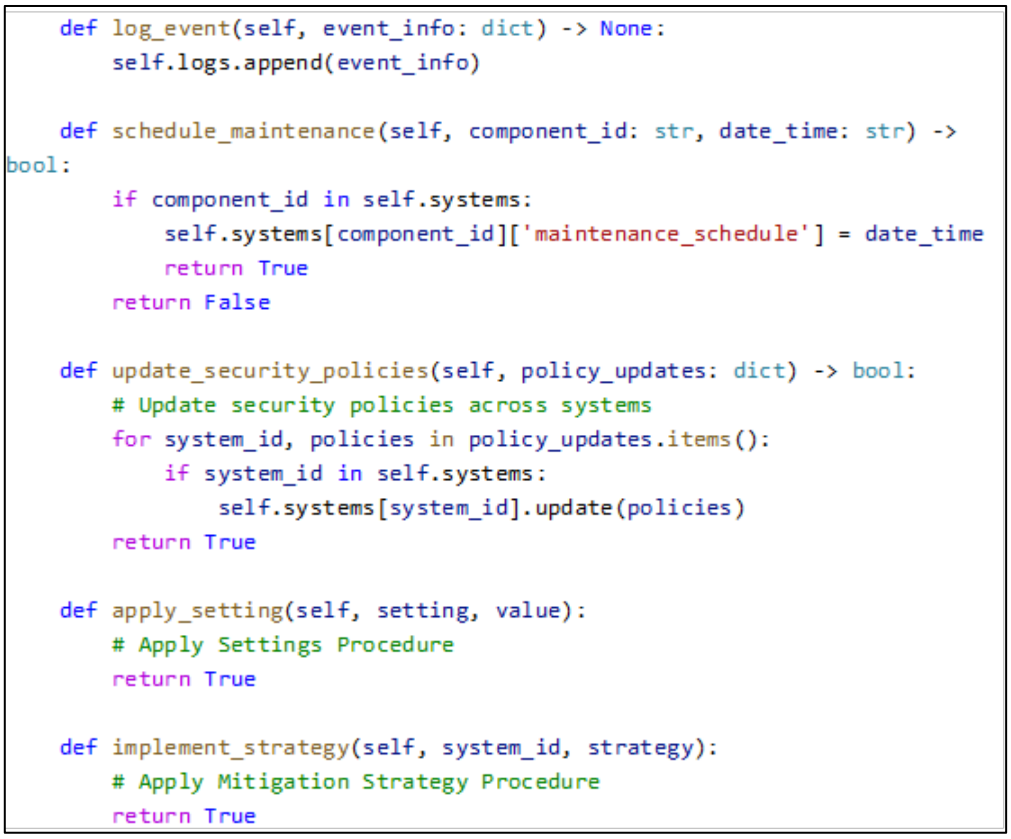
1. "***initiate_verification_process***". Initiates a verification process for a specified technology identified by "*technology_id*". It checks if the technology exists in the "*technologies*" dictionary and verifies its status. It returns "*True*" if the technology is active and presumed verified, otherwise "*False*".
2. "***adjust_data_plane_settings***". Adjusts settings on the programmable data plane based on the provided "*settings*" dictionary. It iterates through each setting and applies it using a hypothetical method "*apply_setting*". It returns "*True*" if the settings have been adjusted, otherwise "*False*".
3. "***analyze_vulnerabilities***". Analyzes and reports vulnerabilities for a system specified by "*system_id*". It checks if the system exists and simulates a vulnerability analysis. It returns a dictionary with vulnerability details if the system exists; otherwise, an empty dictionary.
4. "***apply_mitigation_strategies***". Applies specified mitigation strategies detailed in "*strategy_details*" to address identified risks or vulnerabilities. It iterates over the strategies, applying each to its respective system using the "*implement_strategy*" method. It returns a string message indicating successful application of strategies.
5. "***retrieve_operational_metrics***". Retrieves operational metrics based on a specified "*metrics_type*". It simulates the fetching of metrics data such as uptime, efficiency, and capacity. It returns a dictionary of the fetched metrics.
6. "***log_event***". Logs an event by appending "*event_info*" to the "*logs*" list. This function takes a dictionary containing event details and stores it, expanding the log history. There's no return value as the function solely appends to the log list.
7. "***schedule_maintenance***". Schedules maintenance for a component at a specified date and time ("*date_time*"), identified by "*component_id*". It adds or updates a maintenance schedule in the "*systems*" dictionary for the specified component. It returns "*True*" if the component exists and the schedule is set, otherwise "*False*".
8. "***update_security_policies***". Updates security policies for systems based on the provided "*policy_updates*". It iterates over the policy updates, applying each to the corresponding system if it exists. It returns "*True*" after updating all provided policies, indicating successful updates.
9. "***apply_setting***". A helper method that applies a single setting to the data plane. It prints a message to indicate the setting being applied, serving as a placeholder for actual implementation.
10. "***implement_strategy***". A helper method that applies a mitigation strategy to a specified system. It prints a message to indicate the application of the strategy, providing a simple visual confirmation for this example.

[30] https://www.rabbitmq.com/
[31] https://grafana.com/oss/loki/



```python
class CocoonToolsetDashboard:
    def __init__(self):
        self.systems = {}
        self.technologies = {}
        self.logs = []

    def initiate_verification_process(self, technology_id: str) -> bool:
        if technology_id in self.technologies:
            # Verification Process
            verified = self.technologies[technology_id].get('status', '')
== 'active'
            return verified
        return False

    def adjust_data_plane_settings(self, settings: dict) -> bool:
        # Adjust some settings
        for setting, value in settings.items():
            # Assume a method exists to apply settings to the data plane
            self.apply_setting(setting, value)
        return True

    def analyze_vulnerabilities(self, system_id: str) -> dict:
        if system_id in self.systems:
            # A vuln. analysis module will generate vulnerabilities dict
            vulnerabilities = {'risk_level': 'high',
                'details': 'Multiple unpatched exploits found'}
            return vulnerabilities
        return {}

    def apply_mitigation_strategies(self, strategy_details: dict) -> str:
        # Apply patches or update configurations
        for system_id, strategy in strategy_details.items():
            self.implement_strategy(system_id, strategy)
        return "Mitigation strategies applied successfully."

    def retrieve_operational_metrics(self, metrics_type: str) -> dict:
        # Metrics dict will be fetched from data source
        metrics = {'uptime': '99.9%', 'efficiency': '85%',
            'capacity': '40MW'}
        return metrics
```

Figure 12 CTD Code Signatures 1



```
    def log_event(self, event_info: dict) -> None:
        self.logs.append(event_info)

    def schedule_maintenance(self, component_id: str, date_time: str) ->
bool:
        if component_id in self.systems:
            self.systems[component_id]['maintenance_schedule'] = date_time
            return True
        return False

    def update_security_policies(self, policy_updates: dict) -> bool:
        # Update security policies across systems
        for system_id, policies in policy_updates.items():
            if system_id in self.systems:
                self.systems[system_id].update(policies)
        return True

    def apply_setting(self, setting, value):
        # Apply Settings Procedure
        return True

    def implement_strategy(self, system_id, strategy):
        # Apply Mitigation Strategy Procedure
        return True
```

Figure 13 CTD Code Signatures 2

4.5.2 Implementation Use Case: Anomaly Diagnosis (AD)

The Anomaly diagnosis tool deployed on the CSL layer in the substation will seamlessly integrate with the Cocoon Programmable Node (CPN) to share reports and alerts for events as in Figure 14. It would collect the network packets and analyze the data to classify abnormal events from normal events using the AI algorithms mentioned previously.

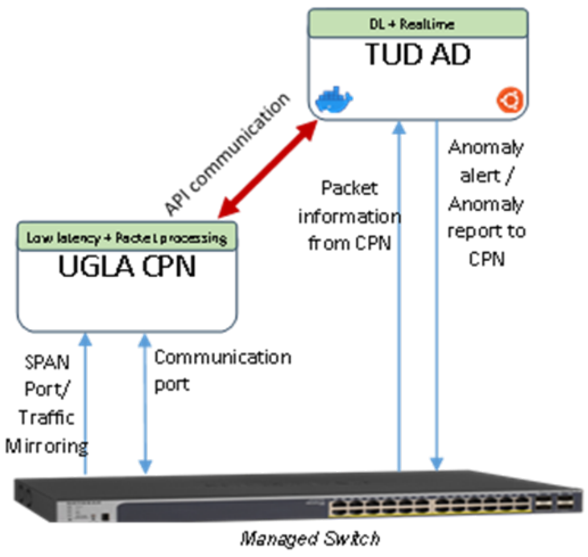

DL + Realtime
TUD AD
API communication
Packet information from CPN
Anomaly alert / Anomaly report to CPN
Low latency + Packet processing
UGLA CPN
SPAN Port/ Traffic Mirroring
Communication port
Managed Switch

Figure 14 Anomaly Diagnosis deployment and integration with CPN at substation

The tool will generate periodic reports, which will be forwarded to the CPN. In case of significant events, the alerting system of the AD tool will be triggered, and the alert will be sent instantaneously to the CPN.

The deployment will be done in a virtual containerized environment and the communication with CPN will be using standard protocols as shown in Figure 14.

The implementation of this communication is to be done through function calls. The functions are as defined below:

Function to be called for getting Input to the AD from CPN

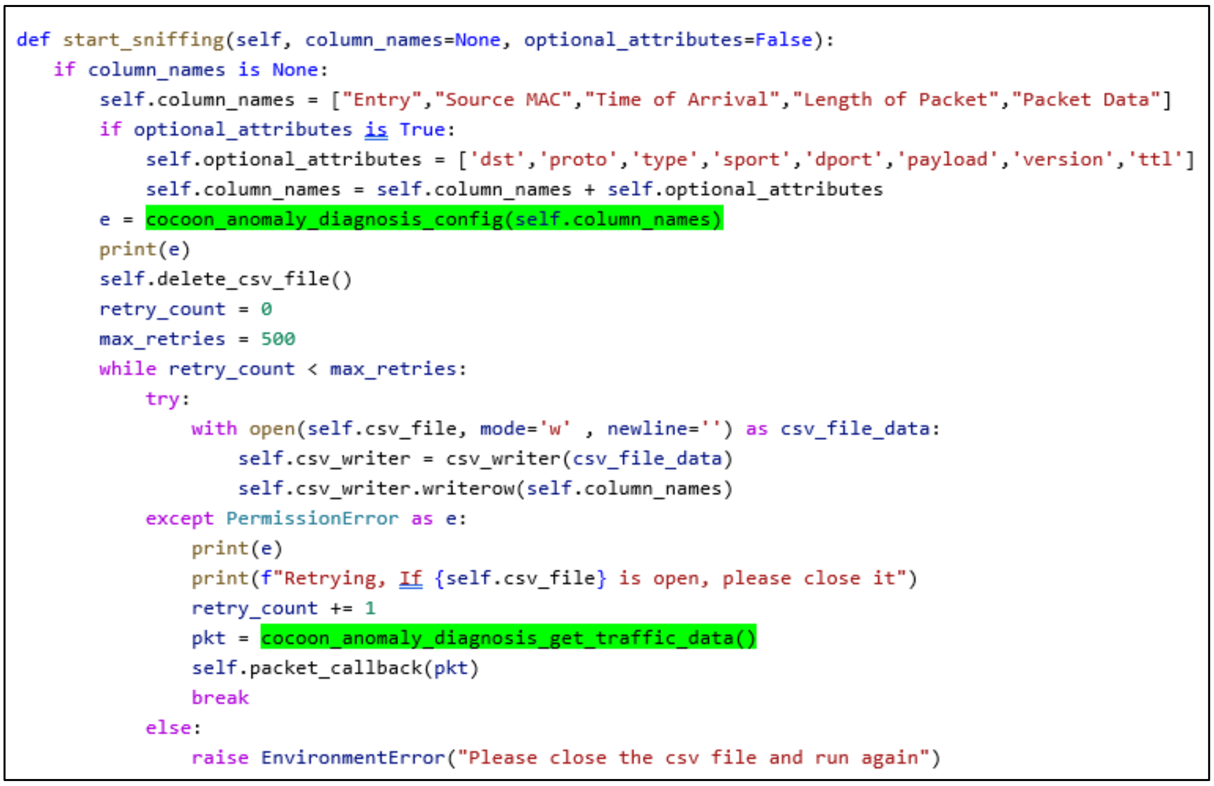
a. Config function: Used to configure the acceptance of packet details from the CPN. The CPN will start capturing the specified details (Packet arrival time, Packet length, etc.) once this function is triggered.
   i. cocoon_anomaly_disgnosis_config(list of strings config = ["time of arrival", "Packet length", "source mac"])
   ii. Return: string specifying if the config is accepted by the IOL of CPN
       e: Error in setting config
       1: No error
b. Get traffic data function: Get the data from the CPN. The CPN will return all the packets stored in the buffer and reset the buffer once all the information is passed.
   i. cocoon_anomaly_diagnosis_get_traffic_data()
   ii. Return: Json object with specified information in config

Function to be called to publish the Output from AD to the CPN

a. Anomaly diagnosis report: The function to be called periodically in case of no anomaly and instantaneously when the anomaly occurs. This will be passing the report/alert in case of event.
   i. cocoon_anomaly_diagnosis_set_status(json object:{})
      Json object: {

      Anomaly_status: Boolean 1 = Anonaly/ 0 = NoAnomaly,
      Summary: str
      Src_mac: str (Null if no anomaly)
      details: string (Other details in case of anomaly or no anomaly)
      }
   ii. Return: string specifying if the status message is accepted by the IOL of CPN
       e: Error in setting accepting
       1: No error

The implementation of these functions can be found in Figure 15 below.



```python
def start_sniffing(self, column_names=None, optional_attributes=False):
    if column_names is None:
        self.column_names = ["Entry","Source MAC","Time of Arrival","Length of Packet","Packet Data"]
        if optional_attributes is True:
            self.optional_attributes = ['dst','proto','type','sport','dport','payload','version','ttl']
            self.column_names = self.column_names + self.optional_attributes
        e = cocoon_anomaly_diagnosis_config(self.column_names)
        print(e)
        self.delete_csv_file()
        retry_count = 0
        max_retries = 500
        while retry_count < max_retries:
            try:
                with open(self.csv_file, mode='w' , newline='') as csv_file_data:
                    self.csv_writer = csv_writer(csv_file_data)
                    self.csv_writer.writerow(self.column_names)
            except PermissionError as e:
                print(e)
                print(f"Retrying, If {self.csv_file} is open, please close it")
                retry_count += 1
                pkt = cocoon_anomaly_diagnosis_get_traffic_data()
                self.packet_callback(pkt)
                break
            else:
                raise EnvironmentError("Please close the csv file and run again")
```

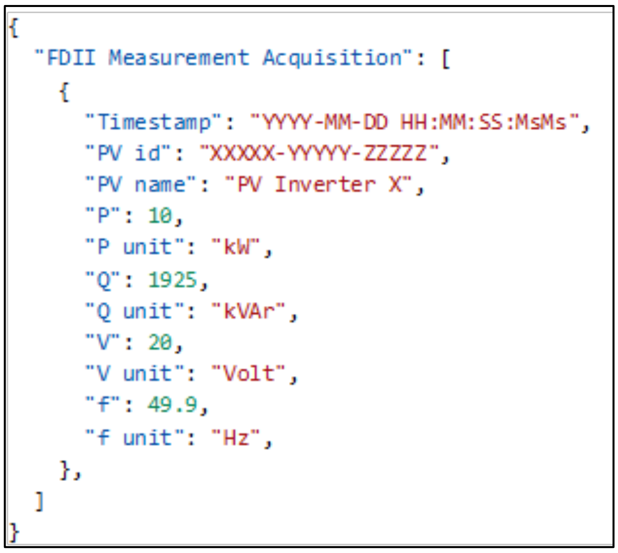
Figure 15 COCOON API implementation for Anomaly Diagnosis (AD) with Cocoon Programmable Node (CPN)

4.5.3 Implementation Use Case: False Data Injection Identification (FDII)

The successful acquisition of electrical measurements from the EPES constitutes one the main prerequisites for the effective implementation of the FDII tool. These measurements will be acquired from the instrumentation and orchestration layer (IOL) foreseen in the proposed CPN architecture through the Northbound API (see Figure 6) following the requirements presented in Section 3.5.3. This communication will be established through either Python function calls or REST API (JSON) in case the various CPN layers and services lie on the same or different VMs, respectively. In both cases, the following function calls are foreseen:

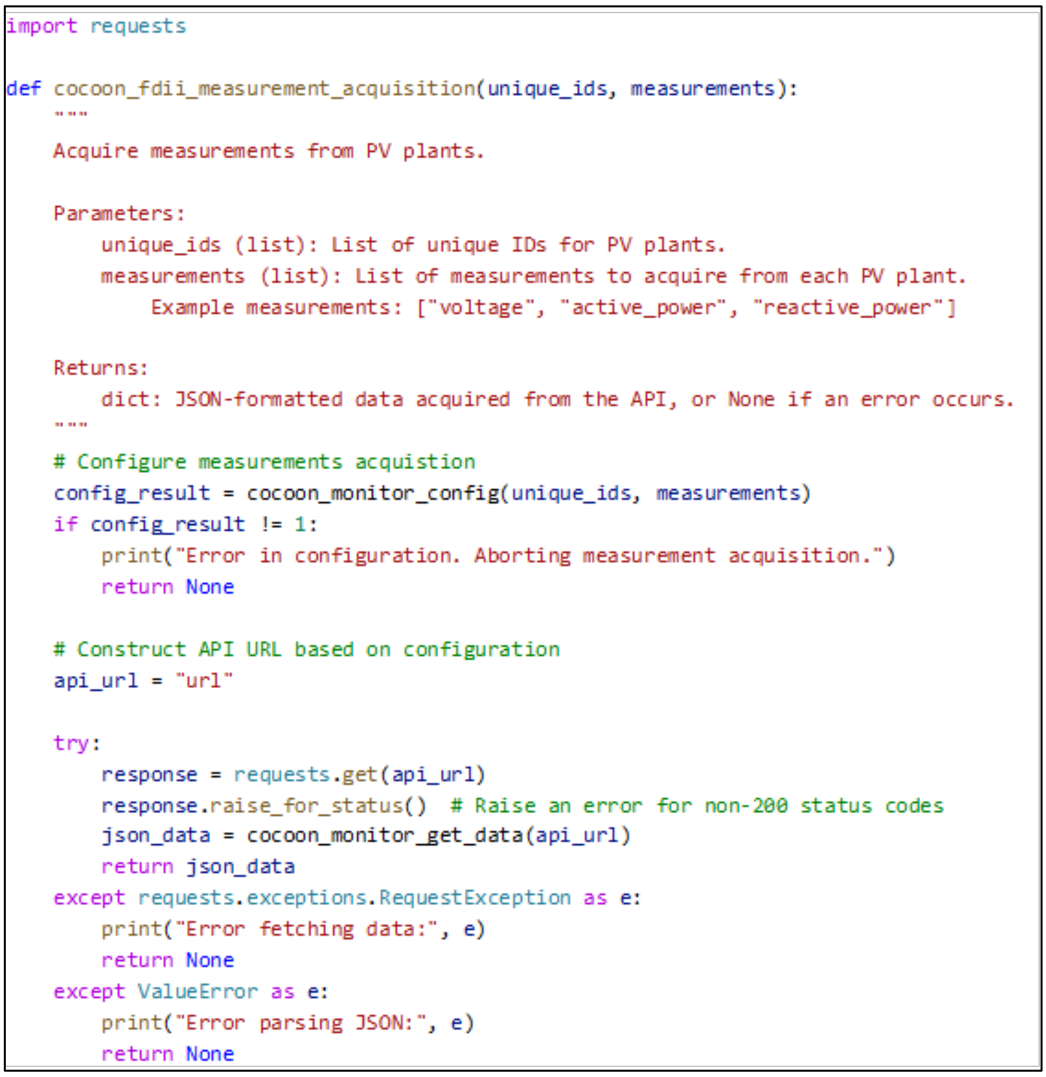
1. Configuration function (*cocoon_monitor_config*). The main scope of this function is to determine the type of the measurements needed by the FDII tool. An indicative but not restrictive list of arguments is given below:
    - *elements_list*: A list containing the unique identifiers of the EPES components that are monitored, e.g., PV power plants, inverters within a PV power plant, additional measurement devices within the distribution grid, etc.
    - *measurements_list*: A list containing for each of the above declared elements, the acquired measurements, e.g., active power (P), reactive power (Q), voltage magnitude (V), etc.

This function will return a string specifying whether the proposed configuration is accepted by the IOL (*1*) or not (*e*).



```
{
  "FDII Measurement Acquisition": [
    {
      "Timestamp": "YYYY-MM-DD HH:MM:SS:MsMs",
      "PV id": "XXXXX-YYYYY-ZZZZZ",
      "PV name": "PV Inverter X",
      "P": 10,
      "P unit": "kW",
      "Q": 1925,
      "Q unit": "kVAr",
      "V": 20,
      "V unit": "Volt",
      "f": 49.9,
      "f unit": "Hz",
    },
  ]
}
```

Figure 16 JSON structure for measurement configuration of FDII tool



```python
import requests

def cocoon_fdii_measurement_acquisition(unique_ids, measurements):
    """
    Acquire measurements from PV plants.

    Parameters:
        unique_ids (list): List of unique IDs for PV plants.
        measurements (list): List of measurements to acquire from each PV plant.
            Example measurements: ["voltage", "active_power", "reactive_power"]

    Returns:
        dict: JSON-formatted data acquired from the API, or None if an error occurs.
    """
    # Configure measurements acquistion
    config_result = cocoon_monitor_config(unique_ids, measurements)
    if config_result != 1:
        print("Error in configuration. Aborting measurement acquisition.")
        return None

    # Construct API URL based on configuration
    api_url = "url"

    try:
        response = requests.get(api_url)
        response.raise_for_status()  # Raise an error for non-200 status codes
        json_data = cocoon_monitor_get_data(api_url)
        return json_data
    except requests.exceptions.RequestException as e:
        print("Error fetching data:", e)
        return None
    except ValueError as e:
        print("Error parsing JSON:", e)
        return None
```

Figure 17 Example code of measurement acquisition of FDII tool

2. Measurements acquisition (*cocoon_monitor_get_data*). This is the core of the interaction between the Northbound API and the FDII tool aiming to acquire all the measurements determined after calling the above configuration function. The output will be a JSON file containing the requested measurements as indicatively presented in Figure 16. An indicative code example for requesting these measurements from the IOL layer of the CPN node is presented in Figure 17.
3. FDII report (*cocoon_fdi_set_status*): The scope of this function is to inform about the outcome of the FDII analysis. The function will return 1 or 0 in case false data are identified or not respectively. Additionally, a JSON file similarly to the one presented in Figure 16 will be returned containing only the PVs with false data.

5 Conclusions

In summary, the adoption of DevOps principles and practices can significantly enhance the software development and delivery process within the COCOON development and delivery of cybersecurity services for power grid stakeholders. This adoption may ultimately lead to improved business outcomes and competitive advantages for the end users of these solutions. By fostering collaboration, automation, and continuous learning within the presented COCOON's DevOps process on optimized pathway for faster and higher quality software delivery, enhanced quality and stability, increased efficiency, reduced costs, greater flexibility and scalability, and security integration throughout the SDLC (DevSecOps), has been created.

The presentation of the CPN architectural blueprints, along with its requirements and integration with network services, is also a crucial step towards developing a comprehensive cybersecurity solution for power grids, which will be subsequently elaborated in the upcoming deliverables of the COCOON Project. The blueprints elaborated in this document provide a clear understanding of the various components and their cross-layering interactions, enabling the creation of a robust and secure CPN-enabled ecosystem of cyber-protection services for EPES.

Through the herein deliverable it was also shown that the extension of the blueprints with northbound and southbound APIs, further enhances the CPN's functionality and interoperability. The northbound REST-based API, facilitates communication between the CSL and IOL, while the southbound API enable interaction between the IOL and COMML. Thus, these APIs ensure seamless data exchange and integration between the different layers of the CPN, allowing a more flexible, scalable, and secure architecture.

Further to the above, the identification of requirements for two examples of services to be integrated within the CLS, the AD and the FDII, offered a first flavour on how to use DevOps principles, architectural abstractions in the form of blueprints such that to develop effective cybersecurity solutions under the CPN paradigm. Specifically, the requirements pertaining to the AD tool were identified. The AD tool, once deployed at the substation CSL layer, will detect anomalous behaviour in network traffic and send alerts to the CPN. The tool will receive network traffic data via custom-developed APIs, ensuring efficient and secure data transmission.

Similarly, in the case of the FDII tool, a set of requirements have been identified and relevant functions have been presented. These FDII requirements and functions were considered such that to allow fully configurable acquisition of grid measurements from various EPES components from IOL and CPN through the northbound API.

In conclusion, the key outcomes presented in the document provide a solid foundation for the development of the unified COCOON solution for EPES. Via leveraging DevOps principles, well-defined architectural blueprints, and the CPN API assurance on the required TRL promised by this project as well as a secure and efficient operation of modern EPES will be ultimately achieved.

References

[1] J. Díaz, D. López-Fernández, J. Pérez and Á. González-Prieto, “Why are many businesses instilling a DevOps culture into their organization?,” *Empirical Software Engineering,* vol. 26, p. 1–50, 2021.

[2] G. Van Rossum, B. Warsaw and N. Coghlan, “PEP 8 – Style Guide for Python Code,” 2001. [Online]. Available: https://www.python.org/dev/peps/pep-0008/. [Accessed 16 April 2024].

[3] A. Hunt and D. Thomas, The pragmatic programmer: From journeyman to master, Addison-Wesley, 2015.

[4] T. Preston-Werner, “Semantic Versioning 2.0.0,” 2011. [Online]. Available: https://semver.org/. [Accessed 16 April 2024].

[5] “Conventional Commits 1.0.0.,” [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/. [Accessed 16 April 2024].

[6] “What is GitLab Flow?,” [Online]. Available: https://about.gitlab.com/topics/version-control/what-is-gitlab-flow/. [Accessed 16 April 2024].

[7] V. Driessen, “A successful Git branching model,” 2011. [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/. [Accessed 16 April 2024].

[8] “GitHub flow,” [Online]. Available: https://docs.github.com/en/get-started/using-github/github-flow. [Accessed 16 April 2024].

[9] A. Moniruzzam et D. Hossain, «Comparative Study on Agile software development methodologies,» arXiv preprint acXiv: 1307.3356, 2013 July 12.

[10] M. Unterkalmsteiner, T. Gorschek et et. al, «Evaluation and Measurement of Software Process Improvement - A Systematic Literature Review,» *IEEE Transactions on Software Engineering,* vol. 38, n° %12, pp. 398-424, March-April 2012.

[11] M. P. Papazoglou, W. -J. van den Heuvel et J. E. Mascolo, «A Reference Architecture and Knowledge-Based Structures for Smart Manufacturing Networks,» *IEEE Software,* vol. 32, n° %13, pp. 61-69, July-Aug. 2019.

[12] S. Manson et D. Anderson, «Cybersecurity for Protection and Control Systems: An Overview of Proven Design Solutions,» *IEEE Industry Applications Magazine,* vol. 25, n° %14, pp. 14-23, July-Aug. 2019.

[13] G. Blokdik, MoSCoW Method: Beginner's Guide - Second Edition, CreateSpace Independent Publishing Platform, 2017.

[14] A. Abus et . A. G. Exposito, Power System State Estimation: Theory and Implementation, CRC Press, 2004.