Software Requirements Specification

for

Federation Architecture for Composed Infrastructure Services (FACIS)

FACIS EasyStack Builder ICAM

(FACIS.ESB_ICAM)

Version 1.0 (November 2025)

Published by
eco – Association of the Internet Industry (eco – Verband der Internetwirtschaft e.V.)
Lichtstrasse 43h
50825 Cologne, Germany

Table of Contents

1 Introduction and Background

The EasyStack Builder (ESB) provides the essential automation layer needed to deploy future Federation Architecture Patterns (FAPs) efficiently and consistently across different cloud providers. It ensures that FAPs relying on the XFSC Orchestrator (ORCE) can be rolled out quickly, reliably, and with far less manual setup effort-regardless of the target platform.

The ESB already offers a set of standardized, reusable deployment configurations compatible with multiple environments, such as T-Systems Open Sovereign Cloud, IONOS DCD, and STACKIT Cloud. This enables federated components such as the ORCE and the Federated Catalogue to be provisioned automatically using low-code integration and Helm-based automation. These capabilities have already been validated in the FACIS IONOS Cloud Test Environment, demonstrating stable, repeatable end-to-end deployments of key federation services.

Next and subject of the current tender, the EasyStack Builder is being expanded with additional modules that automate the deployment, configuration, and lifecycle management of trust, identity, and credentialing services. These core services are essential for secure collaboration in federated ecosystems and will further strengthen cross-provider interoperability.

This is important, because scalable digital federation in Europe depends on the ability to deploy federated services consistently, repeatably, and across heterogeneous cloud environments. Without a unified deployment layer, every FAP would require custom setup work-slowing down adoption and limiting interoperability. The EasyStack Builder removes these barriers by delivering a reusable, generalized deployment framework, making federation not only technically feasible but operationally practical.

The purpose of this tender is to commission the development of a software suite of Builder modules for Identity, Credential, and Access Management (ICAM) for EasyStack Builder by using the Eclipse Project XFSC[1] as the foundation. The delivery of the software is provisionally by Q1 of 2026. The final clarification will be communicated after the award of the tender.

Previously, the FACIS project successfully delivered the foundational components[2]of the ESB framework, which enabled automated orchestration and deployment of individual Federation Services. The key deliverables from this phase included:

- ESB Core Framework providing standardized orchestration capabilities through ORCE.

[1] https://projects.eclipse.org/projects/technology.xfsc
[2] https://github.com/eclipse-xfsc/smartdeployment/tree/main/Easy%20Stack%20Builder%20(ESB)

- Builder Nodes for ORCE and Federated Catalogue, ensuring low-code integration and Helm-based automation.
- Integration within the FACIS IONOS Cloud Test Environment, validating end-to-end deployment of federated components.

Building upon this foundation, FACIS ESB ICAM introduces an extended module suite, aimed at automating the deployment, configuration, and lifecycle management of core trust, identity, and credentialing services within the FACIS project. These modules collectively enhance interoperability and provide a reusable automation layer across federation environments.

Table 1 ESB ICAM: Overview of Modules

Module[3]	**Component Deployed**
OCM Builder (W-Stack)	Organizational Credential Manager
PCM Builder (Cloud)	Personal Credential Manager
TSA Builder	Trust Service API
AAS Builder	Authentication & Authorization Service

Each Builder module must be fully compatible with Kubernetes (v1.29+) and capable of deploying its target service on any CNCF-compliant cluster without additional manual setup or pre-configuration.
All deployments shall be executed through the ESB Core using Helm-based orchestration and declarative low-code workflows within ORCE.

Deployment Responsibility and Complexity Overview

Each Builder module must be deployed, validated, and fully operationalized by the contractor, including script rework, configuration tuning, and fixing any deployment-related issues that arise during integration. The contractor is solely responsible for ensuring that each module runs as expected within the Kubernetes test environment provided by the client and that all Helm and ORCE automation workflows function correctly without manual intervention.

It is important to note that each module has a distinct deployment complexity level, depending on its architecture, dependencies (e.g., Keycloak, TLS, domain setup), and the amount of shell-level automation required. Table 2 illustrates the relative

[3] "OCM W-Stack" and "PCM Cloud" refer to extended Builder variants, incorporating workflow-level automation and cloud-native configuration capabilities.

complexity level for installation, configuration, and runtime validation of each Builder module based on source-level implementation requirements.

Table 2 ESB ICAM: Deployment Complexity

Module	**Deployment Complexity**	**Description**
OCM Builder (W-Stack)	High	Multi-layer orchestration with Keycloak realm creation, credential management, and TLS domain mapping; requires strong Kubernetes and Helm scripting skills.
PCM Builder (Cloud)	Medium-High	Integration with Keycloak for user credential policies and certificate setup; moderate shell and Helm customization needed.
AAS Builder	Medium-High	Multi-service stack (auth-server, key-server, test-server) with complex OIDC configuration and validation.
TSA Builder	Medium	Standard Helm-based service with signing key and policy injection; moderate setup effort.

2 Common Objectives

All Builder modules developed under this tender shall adhere to a unified set of objectives and quality criteria to ensure interoperability, maintainability, and compliance across FACIS. Each module must be deliverable as a self-contained, declarative deployment unit that integrates seamlessly with the EasyStack Builder Core and executes through the ORCE runtime environment.

General Technical Objectives

- Provide **declarative, low-code deployment** of the target component through ORCE UI, supporting dynamic parameter binding and contextual variable substitution.
- Support **Helm-based, fully automated Kubernetes deployment** across multiple environments.
- Ensure **zero-touch operation**, meaning no manual system configuration or external intervention should be required beyond the defined Helm and ORCE inputs.
- Deliver **idempotent deployment logic**, ensuring repeated runs do not create duplicate resources or conflicting states.
- Implement **standardized input/output contracts, structured logging**, and complete uninstall/rollback mechanisms for clean resource removal.

- Integrate fully with the **ORCE monitoring and context system**, exposing deployment status and endpoint information as structured JSON outputs.
- Provide **modular test automation** (Helm dry-run, connectivity, ingress readiness) and include verifiable evidence of successful deployment.
- Include **documentation**, covering node usage, deployment parameters, known limitations, and troubleshooting steps.
- Enforce **TLS 1.3 encryption** and secure credential handling, with all sensitive values stored as **Kubernetes Secrets** and masked in all logs.
- Align with **FACIS technical compliance** standards and Eclipse XFSC open-source governance requirements.

Each Builder module must demonstrate a reproducible, secure, and fully automated deployment lifecycle from configuration to validation. All modules should be interoperable within the FACIS federated architecture, capable of integration testing in the infrastructure, and ready for acceptance verification under the official FACIS runtime environment.

3 Detailed Module Specifications

3.1 OCM Builder Module

3.1.1 Purpose and Context

The **Organizational Credential Manager (OCM)** manages the issuance, lifecycle, and revocation of organization-level Verifiable Credentials (VCs).
The **OCM Builder Node** automates the deployment and configuration of this service through Helm, Kubernetes, and ORCE-based low-code orchestration, ensuring fully reproducible and auditable installation.

This module serves as the foundational trust component for organizational identities, supporting credential issuance policies, Keycloak integration, and TLS-secured endpoints.

3.1.2 Objectives

- Deliver a **complete ORCE Builder Node** for automated deployment of the OCM service.
- Implement **Helm-based Kubernetes deployment** with automated namespace creation, ingress configuration, and TLS secret injection.
- Automate **Keycloak realm, client, and credential provisioning** via REST APIs.
- Provide **idempotent lifecycle management**, supporting deploy, validate, reconfigure, and uninstall operations.

- Enable **structured JSON output contracts** for downstream ORCE node integration.
- Include **comprehensive documentation** and test results covering both deployment and integration validation.

3.1.3 Scope

In Scope:

- ORCE UI node (HTML/JS/JSON schema) and logic implementation in Node.js.
- Helm charts and bash automation scripts for deployment and teardown.
- Keycloak integration for realm, client, and credential management.
- Full integration testing on the FACIS IONOS Kubernetes test environment.
- Documentation, README, and operational guidelines.

Out of Scope:

- Development or modification of PCM, TSA, or AAS software.
- Manual system administration or external configuration beyond Helm execution.

3.1.4 Functional Requirements

Table 3 Functional Requirements for OCM

ID	Requirement	Description
FR-1	Deployment Automation	Full Helm-based deployment with namespace creation, ingress setup, and TLS installation.
FR-2	UI Tabs	Must include Instance Info, Kubernetes & Domain, Keycloak & Credentials.
FR-3	Parameter Validation	Validate kubeconfig, domain, and TLS parameters dynamically before deployment.
FR-4	Keycloak Integration	Automate creation of realm, client, and secrets using REST APIs; store secrets securely.
FR-5	Output Contract	JSON { ocmUrl, keycloakUrl, clientSecret, externalIp, status }.
FR-6	Uninstall Flow	Execute helm uninstall, remove namespace, and clean up Keycloak resources.
FR-7	Resource Optimization	Enforce CPU/RAM quotas and per-namespace resource limits.
FR-8	Logging & Monitoring	Provide structured logs and expose real-time status to ORCE context.

3.1.5 Execution Overview

ORCE UI → Node.js Logic → Helm/Bash Layer → ORCE Context Output

Layered Breakdown:

- **Frontend Layer:** ORCE UI node for configuration input and status monitoring.
- **Backend Logic:** Node.js scripts invoking Helm and shell automation.
- **Execution Layer:** Bash scripts performing Kubernetes operations (namespace, ingress, TLS).

All processes must be idempotent and non-blocking, ensuring repeatable, error-free deployments.

3.1.6 Deployment Lifecycle

1. User configures OCM Builder parameters in ORCE UI.
2. Input validation checks kubeconfig, TLS, and domain integrity.
3. Helm chart deploys the OCM service, ingress, and secrets.
4. Keycloak realm and clients are automatically created via REST APIs.
5. ORCE monitors progress and updates the context with deployment status.
6. On success, output JSON is generated for downstream nodes.
7. Optional uninstall flow removes all associated Kubernetes and Keycloak resources.

3.1.7 UI Schema

- **Instance Info:** Instance name, version, and status fields.
- **Cluster & Domain:** kubeconfig upload, domain, ingress, and TLS configuration.
- **Keycloak & Credentials:** realm name, client setup, and credential display (masked).

All input fields must support validation, default values, and contextual bindings to ORCE variables.

3.1.8 Security

- Enforce TLS 1.3 on all ingress endpoints.
- Store credentials and secrets as Kubernetes Secrets with masking in logs.
- Apply role-based access control (RBAC) for namespace and service account permissions.
- Validate Keycloak endpoints for certificate and token correctness before acceptance.

3.1.9 Testing & Deliverables

- **Integration Tests:** Validate Helm deployment, Keycloak API connection, and TLS readiness on IONOS Kubernetes.
- **Dry-Run Validation:** Include Helm dry-run results to verify parameter integrity.
- **Deliverables:**
    - ORCE Builder Node.
    - Helm charts and bash scripts for deployment/uninstall.
    - Test logs and validation reports.
    - Deployment guide and technical README.

      Short demonstration video (optional) showing end-to-end deployment.

3.2 PCM Builder Module

3.2.1 Purpose and Context

The **Personal Credential Manager (PCM)** is responsible for issuing, managing, and revoking individual-level VCs for users participating in the FACIS federation.
The **PCM Builder Node** provides a fully automated deployment pipeline for this service, using Helm charts, Kubernetes orchestration, and ORCE low-code workflows to ensure seamless integration and configuration.

This Builder automates all deployment stages – from namespace creation to Keycloak configuration and credential policy setup – allowing operators to deploy and manage PCM without any manual intervention.

3.2.2 Objectives

- Deliver a **self-contained ORCE Builder Node** that automates the complete PCM service deployment lifecycle.
- Enable **Helm-based Kubernetes deployment**, including namespace setup, ingress creation, and TLS configuration.
- Automate **Keycloak realm and client setup**, handling user credential policies and authentication rules.
- Support **credential policy customization** (e.g., validity duration, revocation logic, and credential type definitions).
- Provide **idempotent automation**, allowing safe re-deployment or updates without causing resource conflicts.

- Output standardized **JSON contracts** compatible with other FACIS Builder nodes.
- Deliver all **technical documentation, integration test reports, and deployment instructions**.

3.2.3 Scope

In Scope:

- ORCE UI node (HTML/JSON/JS) for configuration input and runtime status.
- Helm automation templates and bash scripts for deployment and cleanup.
- Integration with Keycloak for user credential issuance and policy enforcement.
- Documentation, README, and acceptance testing on the FACIS IONOS Kubernetes environment.

Out of Scope:

- Configuration or maintenance of OCM, TSA, or AAS software.
- Manual system changes outside Helm or ORCE orchestration.

3.2.4 Functional Requirements

Table 4 Functional Requirements for PCM

ID	Requirement	Description
FR-1	Dual-Component Deployment	Deploy both the credential management API and the user interface, with automated namespace and ingress setup.
FR-2	Credential Policy Configuration	Define credential types, expiry duration, and revocation policies dynamically through ORCE UI.
FR-3	Keycloak Integration	Auto-provision Keycloak realm, client, and roles required for credential issuance.
FR-4	Output Contract	JSON { pcmUrl, issuerId, keycloakUrl, clientSecret, status }.
FR-5	Parameter Validation	Validate kubeconfig, TLS, and Keycloak configuration prior to deployment.
FR-6	Uninstall Flow	Helm uninstall and cleanup of all related Kubernetes and Keycloak resources.
FR-7	Resource Optimization	Apply CPU/RAM resource limits for deployed namespaces.
FR-8	Logging & Monitoring	Provide structured logs and real-time ORCE context updates for deployment progress.

3.2.5 Execution Overview

ORCE UI → Node.js Logic → Helm/Bash Automation → ORCE Output Context

Component Responsibilities:

- **Frontend (ORCE Node):** Collects input parameters and displays progress indicators.
- **Backend Logic (Node.js):** Executes Helm commands and automation scripts.
- **Execution Layer (Shell/Helm):** Handles namespace, TLS, ingress, and deployment verification.
- **Integration Layer (Keycloak REST):** Configures realm, clients, and credentials dynamically.

Each component must operate idempotently and provide clear error codes in the event of deployment failure.

3.2.6 Deployment Lifecycle

1. User provides input parameters (domain, kubeconfig, TLS, credential policy) in ORCE UI.
2. ORCE validates the parameters and generates Helm configuration values.
3. Helm deploys the PCM components (API + frontend) and configures ingress with TLS.
4. Keycloak realm and clients are automatically provisioned through REST APIs.
5. Once deployment completes, ORCE verifies ingress and connectivity.
6. JSON output is generated for use in dependent Builder nodes.
7. Optional uninstall flow removes PCM resources and Keycloak entities.

3.2.7 UI Schema

- **Instance Info:** Displays instance name, status, and version.
- **Cluster & Domain:** kubeconfig upload, ingress configuration, TLS parameters.
- **Keycloak & Credentials:** Realm, client, and authentication configuration.
- **Credential Policy:** Define validity, revocation, and credential type options.

All input fields must include validation and contextual default values bound to ORCE variables.

3.2.8 Security

- All PCM endpoints must enforce TLS 1.3 and secure ingress rules.

- Store credentials and secrets as Kubernetes Secrets, fully masked in all logs.
- Support RBAC for access to namespaces and Keycloak APIs.
- Apply Keycloak token validation and HTTPS certificate verification prior to activation.

All outputs must be verified through automated Helm tests and endpoint validation routines.

3.2.9 Testing & Deliverables

- **Integration Tests:** Verify deployment flow, credential issuance via Keycloak, and API accessibility.
- **Helm Dry-Run Validation:** Must include pre-deployment configuration validation.
- **Deliverables:**
    - ORCE Node (UI + JS + JSON schema).
    - Helm charts and bash scripts for deployment/uninstall.
    - Integration test logs and verification reports.
    - Technical documentation and README.
    - Optional short demo video for deployment flow.

Complexity Rating: Medium-High
The PCM Builder combines Keycloak automation, credential policy configuration, and Helm-based orchestration. It requires solid expertise in Kubernetes, Helm, and Keycloak API automation.

3.3 TSA Builder Module

3.3.1 Purpose and Context

The **Trusted Service API (TSA)** provides timestamping, signature verification, and trust-policy enforcement across the FACIS federation.
It ensures the integrity, authenticity, and non-repudiation of data exchanges between federated participants.

The **TSA Builder Node** automates the complete deployment lifecycle of the TSA service using Helm, Kubernetes, and ORCE low-code orchestration, enabling a fully repeatable setup process.
This includes certificate management, policy injection, and validation of trust signatures according to FACIS and eIDAS-compliant requirements.

3.3.2 Objectives

- Deliver a fully automated **ORCE Builder Node** for TSA deployment.
- Implement **Helm-based Kubernetes automation** for namespace creation, ingress, and TLS configuration.
- Manage **signing keys**, **trust policies**, and **certificate chains** through automated configuration scripts.
- Provide support for **JSON-LD and eIDAS-compatible signature validation**.
- Output structured **JSON results** for downstream validation components.
- Ensure deployment consistency and idempotence, and support implementation of security controls in line with ISO/IEC 27001, NIST and CIS Benchmarks, and with eIDAS requirements where applicable.

3.3.3 Scope

In Scope:

- ORCE UI Node and backend logic (Node.js) for TSA deployment configuration.
- Helm automation and bash scripting for namespace, ingress, and TLS setup.
- Integration with Keycloak or external trust frameworks for secure token validation.
- Documentation and testing within the FACIS IONOS Kubernetes environment.

Out of Scope:

- Policy management for OCM or PCM credential issuance.
- Manual system-level key installation or external CA management.

3.3.4 Functional Requirements

Table 5 Functional Requirements for TSA

ID	Requirement	Description
FR-1	Helm Deployment	Automated deployment via Helm, including namespace, ingress, and TLS configuration.
FR-2	Key Management	Support upload of PEM/JWK keys and certificate chains for signing and verification.
FR-3	Policy Configuration	Enable trust-policy enforcement and schema validation through ORCE UI.
FR-4	Output Contract	JSON { tsaUrl, keyId, policyStatus, status }.

FR-5	eIDAS Compliance	Optional flag for activating eIDAS-compliant signature validation.
FR-6	Uninstall Flow	Full teardown: helm uninstall, namespace cleanup, and key deletion.
FR-7	Resource Optimization	Limit CPU/RAM resources per namespace for lightweight execution.
FR-8	Logging & Monitoring	Structured JSON logs for each automation stage; exposed via ORCE context.

3.3.5 Execution Overview

ORCE UI → Node.js Logic → Helm/Bash Automation → Key Management + Policy Engine → ORCE Context Output

Component Layers:

- **Frontend:** ORCE UI Node for parameter configuration and policy upload.
- **Backend Logic:** Node.js script controlling Helm commands and key injection routines.
- **Execution Layer:** Bash scripts for Kubernetes operations (namespace, TLS, ingress).
- **Integration Layer:** API layer handling trust-policy enforcement and key validation.

Each deployment must be idempotent, ensuring that re-execution does not overwrite existing valid configurations.

3.3.6 Deployment Lifecycle

1. User defines parameters (domain, TLS, policy file, key set) in ORCE UI.
2. ORCE validates inputs and triggers Helm deployment.
3. Namespace, ingress, and TLS certificates are automatically created.
4. PEM/JWK keys and trust policies are injected into the running service.
5. ORCE monitors the deployment status and validates the TSA endpoint.
6. Final JSON output is generated with trust-policy verification results.
7. Optional uninstall flow removes all TSA-related Kubernetes resources.

3.3.7 UI Schema

- **Instance Info:** Service name, version, and status.
- **Cluster & Domain:** kubeconfig, ingress configuration, TLS setup.
- **Trust Configuration:** PEM/JWK key upload, policy injection, eIDAS compliance toggle.

All inputs must support file validation, parameter checks, and be context-aware through ORCE variable binding.

3.3.8 Security

- Enforce TLS 1.3 for all external communication.
- Store private keys and certificates as Kubernetes Secrets (never in plaintext).
- Implement RBAC for TSA namespace and configuration operations.
- Support cryptographic signature verification for integrity of all policies and keys.
- Ensure compatibility with the FACIS and eIDAS standards.

3.3.9 Testing & Deliverables

- **Integration Testing:** Validate policy injection, signing/verification functions, and endpoint availability.
- **Dry-Run Validation:** Provide Helm dry-run logs and parameter verification results.
- **Deliverables:**
  - ORCE Node (UI + JS + JSON schema).
  - Helm and bash scripts for deployment and uninstall.
  - Integration test logs and trust-validation reports.
  - Documentation and operational guide (README).
  - Optional short demo video showcasing TSA setup.

Complexity Rating: Medium
TSA Builder requires moderate Kubernetes and Helm expertise, with additional understanding of key management, JSON-LD, and trust-policy configuration.

3.4 AAS Builder Module

3.4.1 Purpose and Context

The **Authentication & Authorization Service (AAS)** provides centralized authentication and authorization across all FACIS federation components.
It implements OpenID Connect (OIDC) and OAuth 2.0 flows to enable secure identity exchange, single sign-on (SSO), and service-to-service authentication among federated modules.

The **AAS Builder Node** automates the complete deployment of the AAS microservices stack including the auth-server, key-server, and test-server using Helm, Kubernetes, and ORCE low-code orchestration.

This Builder ensures that all authentication endpoints and token exchanges are properly configured and validated in a fully automated manner.

3.4.2 Objectives

- Deliver an **automated ORCE Builder Node** capable of deploying the full AAS service stack.
- Automate **Helm-based Kubernetes deployment**, covering namespace creation, ingress configuration, and TLS enforcement.
- Configure **Keycloak realms and clients** for OIDC/SIOP authentication flows automatically.
- Validate the deployed environment via test endpoints and connectivity checks.
- Provide **structured JSON outputs** representing deployment results and endpoint metadata.
- Guarantee **idempotent and secure** deployment cycles across all FACIS test clusters.

3.4.3 Scope

In Scope:

- ORCE UI Node for input configuration and runtime monitoring.
- Helm charts and shell scripts for deployment, teardown, and validation.
- Keycloak configuration for realm, client, and token exchange setup.
- Documentation, test reports, and user guides.

Out of Scope:

- Non-OIDC authentication protocols or external identity provider integration.
- Manual adjustment of Keycloak configuration outside the automated workflow.

3.4.4 Functional Requirements

Table 6 Functional Requirements for AAS

ID	Requirement	Description
FR-1	Multi-Service Deployment	Deploy auth-server, key-server, and test-server using Helm charts and automated namespace setup.
FR-2	Keycloak Integration	Auto-provision Keycloak realm and clients for SSI OIDC and SSI SIOP authentication.
FR-3	Verification Endpoint	Deploy and validate /demo endpoint to confirm authentication functionality.
FR-4	Output Contract	JSON { aasAuthUrl, keyServerUrl, testUrl, status }.

FR-5	Security	Enforce TLS 1.3, secure credential handling, and RBAC for namespaces.
FR-6	Uninstall Flow	Helm uninstall and Keycloak cleanup for all deployed services.
FR-7	Resource Optimization	Apply CPU and memory quotas for each service within the namespace.
FR-8	Logging & Monitoring	Provide structured logging and real-time ORCE context updates during deployment.

3.4.5 Execution Overview

ORCE UI → Node.js Logic → Helm/Bash Automation → Keycloak REST API → AAS Microservices → ORCE Context Output

Component Layers:

- **Frontend (ORCE Node):** User input and configuration parameters.
- **Backend Logic (Node.js):** Executes Helm and automation scripts, monitors Helm status.
- **Execution Layer:** Bash scripts handling namespace, ingress, and TLS setup.
- **Integration Layer:** Keycloak REST calls and token validation APIs.

Each deployment must support idempotent execution and handle rollback scenarios automatically in case of deployment failure.

3.4.6 Deployment Lifecycle

1. User enters configuration data (domain, TLS, Keycloak parameters) in ORCE UI.
2. ORCE validates parameters and generates Helm configuration files.
3. Helm deploys auth-server, key-server, and test-server within the namespace.
4. Keycloak realm and clients are provisioned automatically for OIDC/SIOP.
5. ORCE monitors the deployment process and performs endpoint connectivity tests.
6. Upon success, JSON output is generated with service URLs and statuses.
7. Uninstall flow removes all resources and clears Keycloak configurations.

3.4.7 UI Schema

- **Instance Info:** Display service version, status, and runtime info.
- **Cluster & Domain:** kubeconfig upload, ingress domain, and TLS parameters.
- **Keycloak & Authentication:** Realm name, client IDs, redirect URIs, token lifespan.
- **Verification:** Simple /demo endpoint test and status indicator.

All inputs must be validated at runtime and bound to ORCE context variables for automation chaining.

3.4.8 Security

- All services must enforce TLS 1.3 and use verified certificate chains.
- All credentials stored as Kubernetes Secrets, masked in UI and logs.
- Apply RBAC policies for service accounts with least-privilege principles.
- Tokens and credentials validated before service activation.
- Support OIDC discovery endpoints (.well-known/openid-configuration) for interoperability.

3.4.9 Outputs

Example of ORCE JSON output structure:

```
{
 "aasAuthUrl": "https://aas.facis.example/auth",
 "keyServerUrl": "https://aas.facis.example/key",
 "testUrl": "https://aas.facis.example/demo",
 "status": "Deployed"
}
```

Each output parameter must be verified through test endpoints to confirm successful deployment and token exchange.

3.4.10 Testing & Deliverables

- **Integration Tests:** Validate OIDC flow, client authentication, and Keycloak token issuance.
- **Dry-Run Validation:** Include Helm dry-run logs and pre-deployment checks.
- **Deliverables:**
  - ORCE Builder Node (UI + JS + JSON schema).
  - Helm charts and bash scripts for automated deployment/uninstall.
  - Integration test results and validation logs.
  - Technical documentation and user guide (README).
  - Optional short demo video showing authentication validation.

Complexity Rating: Medium-High
AAS Builder integrates multiple microservices and Keycloak-based authentication flows.

4 Non-Functional Requirements

All Builder modules (OCM, PCM, TSA, AAS) must meet the following non-functional standards to ensure secure, reliable, and consistent operation within FACIS:

- **Performance:** Each module must deploy in under 10 minutes on a standard FACIS Kubernetes cluster and support idempotent redeployment without conflict.
- **Security:** Enforce TLS 1.3, store all secrets as Kubernetes Secrets, and mask credentials in logs.
- **Reliability:** Ensure automatic rollback on failure and maintain ≥ 99 % deployment success in test environments.
- **Maintainability:** Follow modular code structure (Node.js + Helm + Bash) with clear documentation and reusable templates.
- **Interoperability:** Operate on any CNCF-compliant cluster and integrate with the EasyStack Builder Core using standardized JSON I/O.
- **Testing:** Include automated Helm dry-run and integration tests verifying ingress, Keycloak connectivity, and output validation.

5 Technical Implementation Requirements

All Builder modules must be implemented following the unified technical framework defined for the EasyStack Builder Core and the FACIS ORCE runtime environment.
These requirements ensure that each Builder Node is consistent, maintainable, and interoperable within FACIS.

5.1 Frontend Layer

- Implemented as **ORCE UI components** in **Node.js**, using the standard ORCE node schema (.html, .js, .json).
- All input parameters (e.g., kubeconfig, domain, TLS, credentials) must be **declaratively defined** and bound dynamically to ORCE context variables.
- The UI must provide **real-time validation and feedback** for all required inputs.
- Each UI node must follow a **three-tab structure**: Instance Info, Cluster & Domain, and Authentication/Credentials.

- Outputs must be displayed in structured JSON format within ORCE for downstream automation.

5.2 Backend Logic Layer

- Written in **JavaScript (Node.js)** and executed inside the **ORCE runtime container**.
- The logic must call **Helm**, **kubectl**, and **bash-based scripts** to handle deployment, validation, and teardown.
- The backend must be fully **idempotent**, handling retries without duplicating resources.
- All configurations must be externalized and injected via ORCE parameters.
- Logging must be **structured (JSON)** and report all key execution stages and exit codes.

5.3 Execution Layer

- Each Builder Node must include a **Linux-based Shell Automation Layer** responsible for system-level operations:
  - Namespace and service creation
  - TLS secret generation and certificate installation
  - Ingress rule and domain configuration
  - Resource cleanup on uninstall
- Automation scripts must reside under a dedicated /scripts directory and be callable from Node.js via child_process.spawn.
- Each script must return **structured JSON output or status codes** to ORCE for logging and monitoring.
- Scripts must be **non-root**, rely only on Kubernetes service account permissions, and support **repeatable, conflict-free execution**.

5.4 Deployment Engine

- Implemented with **Helm charts** stored in a /helm directory.
- Helm templates must include:
  - Namespace creation
  - TLS and ingress rules
  - Environment variable templating

    - Cleanup logic using helm uninstall and validation hooks
- Charts must be reusable and compatible with any **CNCF-compliant Kubernetes v1.29+** environments.
- Values must be defined in values.yaml and dynamically injected via ORCE runtime parameters.

5.5 Integration Interfaces

- All Builder modules must communicate through **REST APIs** over **HTTPS** with **JSON-based payloads**.
- Endpoints (e.g., OCM → PCM → TSA → AAS) must be configurable and validated at runtime.
- Output contracts must conform to a **shared JSON schema**, enabling inter-node chaining inside ORCE.
- Any external service (e.g., Keycloak, external CA) must use authenticated REST calls secured with **OAuth 2.0 bearer tokens**.

5.6 Configuration and Security

- All secrets, credentials, and tokens must be stored as **Kubernetes Secrets**.
- Sensitive values must never appear in logs or plain configuration files.
- Helm values and environment variables must be validated before execution.
- Enforce **TLS 1.3** on all ingress and service endpoints.
- Follow the **FACIS security baseline** and **Eclipse XFSC governance rules** for credential handling.

5.7 Testing Framework

- Each Builder must include automated tests under a /test directory:
    - Helm installation and uninstall tests
    - Ingress reachability and endpoint validation
    - Connectivity checks with dependent components (Keycloak, TSA, AAS)
    - JSON schema validation for outputs
- All test results must be exportable in structured format (JSON or Markdown).
- Tests must run successfully on IONOS Cloud test cluster or any FACIS-approved Kubernetes environment.

Summary:

Every Builder Node must be fully automated, reproducible, and compliant with the FACIS technical baseline (Kubernetes v1.29+, Helm, ORCE runtime, TLS 1.3, Kubernetes Secrets, RBAC, Apache 2.0, and XFSC governance), combining Node.js logic, Helm based orchestration, and secure Bash automation under a unified open-source governance model.

6 Deliverables Summary

Each Builder module must include the following deliverables:

- **ORCE Builder Node (UI + Logic):** Complete .html, .js, and .json files.
- **Helm Charts & Bash Scripts:** Deployment, uninstall, and validation automation.
- **Integration Tests:** Helm dry-run results, connectivity tests, and output validation.
- **Documentation:** Technical README, configuration guide, and troubleshooting notes.
- **Test Report & Logs:** Evidence of successful deployment on the FACIS IONOS cluster.
- **Optional Demo Video:** Short clip (≤3 min) demonstrating full deployment lifecycle.

7 Project Governance & Ownership

All outputs remain under the ownership of the FACIS project and shall be open sourced under Eclipse Foundation (Apache 2.0 License).

8 Validation & Acceptance Criteria

Acceptance of the EasyStack Builder ICAM modules will be based on a combination of successful runtime behaviour, verified outputs, and documented evidence.

8.1 Core Acceptance Criteria

Acceptance will be based on the following core criteria, which apply to all Builder modules (OCM W-Stack, PCM Cloud, TSA Builder, AAS Builder):

- **Successful functional deployment**
  Each module can be deployed through the ORCE UI without manual intervention, using the agreed workflows and parameters.
- **Idempotent lifecycle behavior**
  Repeated execution of the Deploy and Uninstall workflows results in a consistent state with no duplicate resources or residual artefacts.

- **Verified outputs per module**
  Each module produces the expected JSON output contract in ORCE context (for example `ocmUrl`, `pcmUrl`, `tsaUrl`, `aasAuthUrl`, `status`), and all referenced endpoints are reachable.
- **Documented tests, user guides and video tutorials**
  Test procedures, deployment instructions, configuration guides and, where applicable, short demo videos are delivered and accepted by the client.
- **Security and compliance checks**
  TLS 1.3 is enforced on all external endpoints, sensitive credentials are stored as Kubernetes Secrets and masked in logs, and RBAC follows least privilege principles.
- **Target environment validation**
  All modules are deployed and successfully tested on the client provided Kubernetes test environment, as specified in this SRS.

8.2 Module-Specific Acceptance References

For runtime sign off, the following module level acceptance IDs and references are used:

- **OCM W-Stack**
  - **Acceptance ID:** IDM.OCM.W-STACK.00046
  - **Goal:** Creates and publishes the public profile endpoint that serves the self-description of the organization based on selected VCs.
  - **Reference:** https://eclipse.dev/xfsc/ocmwstack/ocmwstack/#idmocmw-stack00046-create-public-profile-endpoint
- **PCM Cloud**
  - **Acceptance ID:** IDM.PCM.E1.00020
  - **Goal:** Synchronization functionality between different personal wallet instances (cloud PCM and smartphone wallet).
  - **Reference:** https://eclipse.dev/xfsc/pcme1/pcme1/#idmpcme100020-sync-wallets
- **TSA Builder**
  - **Acceptance ID:** IDM.TSA.E1.00063
  - **Goal:** An administration API for policies is available.
  - **Reference:** https://eclipse.dev/xfsc/tsae1/tsae1/#user-interfaces
- **AAS Builder**
  - **Acceptance Target**: Keycloak administration console accessible in browser and basic authentication flows working as specified.
  - **Reference:** https://github.com/eclipse-xfsc/aas/blob/main/doc/install/keycloak/index.md

9 Terms and Abbreviations

Table 7 Terms and Abbreviations

Term or Abbreviation	Full Name	Meaning and Role in This Document
AAS	Authentication and Authorization Service	Central authentication and authorization service implementing OIDC and OAuth 2.0 flows for FACIS components.
API	Application Programming Interface	Programmatic interface of services such as OCM, PCM, TSA, AAS, and Keycloak, usually exposed over HTTP or HTTPS.
Bash	Bourne Again Shell	Shell environment used for the Linux Shell Automation Layer; scripts under the /scripts directory are written in Bash.
Builder Node	EasyStack Builder Node	A self-contained ORCE node (UI plus logic plus Helm plus scripts) that automates deployment and lifecycle of a specific service such as OCM or PCM.
CA	Certificate Authority	Internal or external authority that issues TLS certificates used by ingress and trust services.
CNCF	Cloud Native Computing Foundation	Industry body that defines Kubernetes conformance; in this tender a "CNCF compliant cluster" means a Kubernetes cluster such as IONOS or T Systems that passes CNCF conformance tests.
eIDAS	Electronic Identification, Authentication and Trust Services	European framework for electronic identification and trust services; TSA can optionally enforce eIDAS compatible validation.
ESB	EasyStack Builder	Short form used in the title for the EasyStack Builder framework
ESB ICAM	EasyStack Builder Identity, Credential, and Access Management	Extension suite of 4 modules for EasyStack Builder
Helm	Helm package manager for Kubernetes	The deployment engine used by all Builder modules to install, upgrade, and uninstall services on Kubernetes.
Helm chart	Helm chart	A Helm package containing templates and values that define how OCM, PCM, TSA, and AAS are deployed.

HTML	HyperText Markup Language	Markup language used for the UI part of Builder Nodes (.html files).
HTTPS	HyperText Transfer Protocol Secure	Secure transport protocol required for all REST based APIs and web endpoints in the system.
Ingress	Kubernetes Ingress	Kubernetes resource that exposes services over HTTP or HTTPS using domains and TLS certificates.
IONOS	IONOS Cloud	Cloud provider hosting the FACIS Kubernetes test environment where all Builder modules must be validated.
JS	JavaScript	Programming language used for the Node.js logic of ORCE Builder Nodes (.js backend files).
JSON	JavaScript Object Notation	Data format used for output contracts, ORCE context values, logs, and structured results.
JSON-LD	JSON for Linked Data	JSON based format used by TSA for linked data signatures and optional eIDAS compatible validation.
JWK	JSON Web Key	JSON based key format used by TSA for signing and verification of trust data.
Kubeconfig	Kubernetes configuration file	Configuration file that provides credentials and cluster access; users upload it in ORCE UI to let Builder Nodes talk to the cluster.
Kubernetes	-	Container orchestration platform on which all Builder modules must deploy their target services. Version v1.29+ is required.
Kubernetes Secrets	Kubernetes Secret Objects	Native Kubernetes objects used to store credentials, keys, and sensitive values securely for all Builder modules.
Namespace	Kubernetes Namespace	Isolated resource space in Kubernetes; each Builder module creates and manages its own namespace for deployment and teardown.
Node.js	Node.js	JavaScript runtime used to execute backend logic for Builder Nodes inside the ORCE runtime container.
OAuth 2.0	OAuth 2.0 Authorization Framework	Authorization standard used by AAS for tokens, bearer authentication, and API protection.

OCM	Organizational Credential Manager	Service that issues, manages, and revokes organization level Verifiable Credentials inside the FACIS federation.
OIDC	OpenID Connect	Identity protocol on top of OAuth 2.0 implemented by AAS for authentication flows and SSO.
ORCE	Orchestration Engine	The EasyStack Builder runtime environment that executes Builder Nodes including UI, Node.js logic, Helm, and shell scripts.
ORCE UI	ORCE User Interface	Visual interface inside ORCE where users configure Builder parameters, trigger deployments, and view status or JSON outputs.
PCM	Personal Credential Manager	Service that issues, manages, and revokes individual level Verifiable Credentials for users in the FACIS federation.
PCM Cloud	Personal Credential Manager Cloud Variant	Cloud native Phase 2 variant of the PCM Builder that targets CNCF compliant Kubernetes clusters.
PEM	PEM encoded key or certificate	File format for keys and certificates that TSA and other modules can use for signing and TLS.
RBAC	Role Based Access Control	Access control mechanism applied to Kubernetes namespaces and service accounts, enforcing least privilege.
REST	Representational State Transfer	HTTP based style of API used for Keycloak and other service interactions in this tender.
SIOP	Self-Issued OpenID Provider	Self-sovereign OpenID flow used by AAS for SSI SIOP based authentication.
SSI	Self-Sovereign Identity	Identity model referenced in the AAS module for SSI OIDC and SSI SIOP flows.
SSO	Single Sign On	User experience where one login provides access to multiple FACIS services, enabled by AAS using OIDC and OAuth 2.0.
TLS	Transport Layer Security	Encryption protocol required for all external endpoints; the document mandates TLS 1.3.
TLS 1.3	Transport Layer Security version 1.3	Specific TLS version required for all ingress and service endpoints across Builder modules.

TSA	Trusted Service API	Service that provides timestamping, signature verification, and trust policy enforcement across the FACIS federation.
T-Systems	T-Systems Cloud	Additional cloud provider whose Kubernetes clusters are examples of CNCF compliant targets for the Builder modules.
URL	Uniform Resource Locator	Address of HTTP or HTTPS endpoints; appears in JSON outputs such as ocmUrl, pcmUrl, aasAuthUrl.
VC(s)	Verifiable Credential(s)	Standardized digital credentials (for organizations or individuals) whose lifecycle is managed by OCM and PCM.
XFSC	Eclipse Cross Federation Services Components	Eclipse open-source project under which FACIS outputs are governed and licensed.
YAML	YAML Ain Not Markup Language	Configuration format used for Helm templates and values.yaml files.