WORKSHOP 15.09.25

ORCE Fundamentals: A Hands-On Workshop

First FAP Workshop

info@facis.eu
www.facis.eu

CLOUD·EDGE CONTINUUM

Finanziert von der Europäischen Union
NextGenerationEU

Gefördert durch:
Bundesministerium für Wirtschaft und Energie
aufgrund eines Beschlusses des Deutschen Bundestages

01

Welcome & FACIS background

Lauresha Toska | Project Lead FACIS

Housekeeping Rules

Please note*

- This workshop will be recorded and made available online afterwards
- Please remain muted throughout the entire presentation
- Feel free to submit your questions in writing via the chat
- All questions will be addressed at the end of today's presentation.
- You are also welcome to ask your question/feedback verbally during the Q&A session following the presentation
  - To do so, please use the "raise hand" function.
  - We will call on you in turn – please unmute yourself, briefly state your name and company, and then ask your question.

AGENDA

01 Welcome & FACIS Background

02 WP1 FAPs & ORCE Fundamentals

03 Design & Deployment

04 Advanced Topics & Hands-On

About FACIS

FACIS (Federation Architecture for Composed Infrastructure Services)

- Official 8ra partner
- Project duration: November 2024 – January 2027
- Project management: eco – Association of the Internet Industry

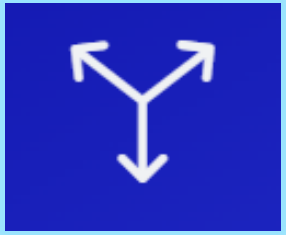
→ Addressing fragmentation through federation of services across providers without losing autonomy

→Provide open-source, reusable components deployable across industries (manufacturing, mobility, aerospace, etc.)

Key Deliverables:

Federation Architecture Patterns

- Modular blueprints/ orchestrated templates for building and operating trusted, federated digital ecosystems



Digital Contracting Service

- Tool to support bilateral and multilateral contract management
- Combines eSigning with qualified electronic signatures such as EUDI

SLA Governance Framework

- Base Taxonomy for multi-provider SLAs and representation of maschine-readable SLAs
- Playbook enabling consistent, transparent, and enforceable SLA governance across complex service ecosystems

- **Proof of Concept**
  - The practical application of the project results will be tested in proof-of-concept projects to demonstrate their feasibility in various domains.

02

WP1 FAP & ORCE Fundamentals

Hossein Rafieekhah | WP1 Lead

02

Understand the Core Concepts

Grasp the key objectives and design principles behind ORCE and its role in modern workflows.

Introduction to Orchestration Engines

The engine for automated tasks

Definition

- A software tool that manages and runs tasks in order (like a train conductor)
- Ensuring tasks reach the right endpoint at the right time

Purpose

- Automate repetitive tasks
- Ensure tasks are completed in the right order and on time

Benefits

Efficiency:	Saves time through automation
Consistency:	Reduces human error by enforcing standard execution
Flexibility:	Easily adapts to new steps, rules, or integrations
Acceleration:	Fast prototyping

XFSC Orchestration Engine (ORCE)

Driving Digital Transformation in Federated Ecosystems

Definition

ORCE is an open-source, low-code orchestration engine designed for the XFSC Toolbox. It uses a visual flow-based programming model and adds governance, security, and data-sovereignty features to support federated infrastructure.

Role in FACIS

- Ensures data flows correctly and securely across the ecosystems and FAPs.
- Connects and orchestrates multiple XFSC Services.

Why it matters

- **Speed:** Automates onboarding and accreditation workflows
- **Accuracy:** Reduces manual steps and human mistakes
- **Integration:** Connects heterogeneous services into one cohesive process

Key Features & Benefits of ORCE

Enhanced UI & Navigation: streamlined, modern interface
JSON-based GUI Generator: quickly build dashboards and forms
Specialized Nodes: ready-to-use building blocks for APIs, services, databases
Cloud & Kubernetes Integration: deploy locally or at scale with ease
Advanced Networking: secure and flexible inter-service communication
High Performance & Reliability: designed for production workloads
Visual Programming Environment: intuitive drag-and-drop editor
Wide Range of Pre-Built Nodes: supports multiple protocols (HTTP, MQTT, DB, ...)
Extensible Architecture: build custom nodes & plugins when needed
Lightweight & Scalable: runs on laptop, edge device, or cloud cluster
Strong Open-Source Community: active development, shared resources, examples

Flow-Based Programming (FBP)

Flow-Based Programming is the core paradigm that ORCE is built on. Instead of writing long, linear code, we define our application as a **network of independent nodes** that pass messages to each other. Each node performs a single, well-defined function, such as transforming data, calling an API, or making a decision and sends the result to the next node through a connection.

- **Programming paradigm:** Applications are networks of interconnected processes
- **Nodes as processes:** Each node encapsulates a single responsibility
- **Data flows via connections:** Messages move along “wires” between nodes
- **Visual representation:** Easier to understand, debug, and explain
- **Modularity & reusability:** Build components once and reuse them
- **Supports parallelism:** Multiple flows can run concurrently, improving performance

Advantages of Flow-Based Programming

Flow-Based Programming (FBP) lets us design applications as networks of connected nodes where data flows step by step.
This makes workflows visual, modular, and easier to maintain, scale, and debug.
It's ideal for building flexible, concurrent systems and speeding up development cycles.

Modularity & Reusability: build once, use many times
Simplified Code Organization: clear and structured flows
Clear Separation of Concerns: easier reasoning about each step
Maintainable Code: simpler to update and extend
Easier Parallelism: natural fit for concurrent processing
Enhanced Scalability: grow flows without breaking existing logic
Adaptable to Changing Requirements: swap nodes or rewire easily
Faster Development & Prototyping : rapid iteration with visual feedback
Accelerated Project Completion: less time from idea to production

ORCE in XFSC

ORCE is the orchestration engine at the heart of XFSC, ensuring that data and processes move securely and transparently across organizations.
It enables trusted automation by connecting federation services, enforcing policies, and logging every step.
This makes workflows auditable, interoperable, and scalable across multiple participants.

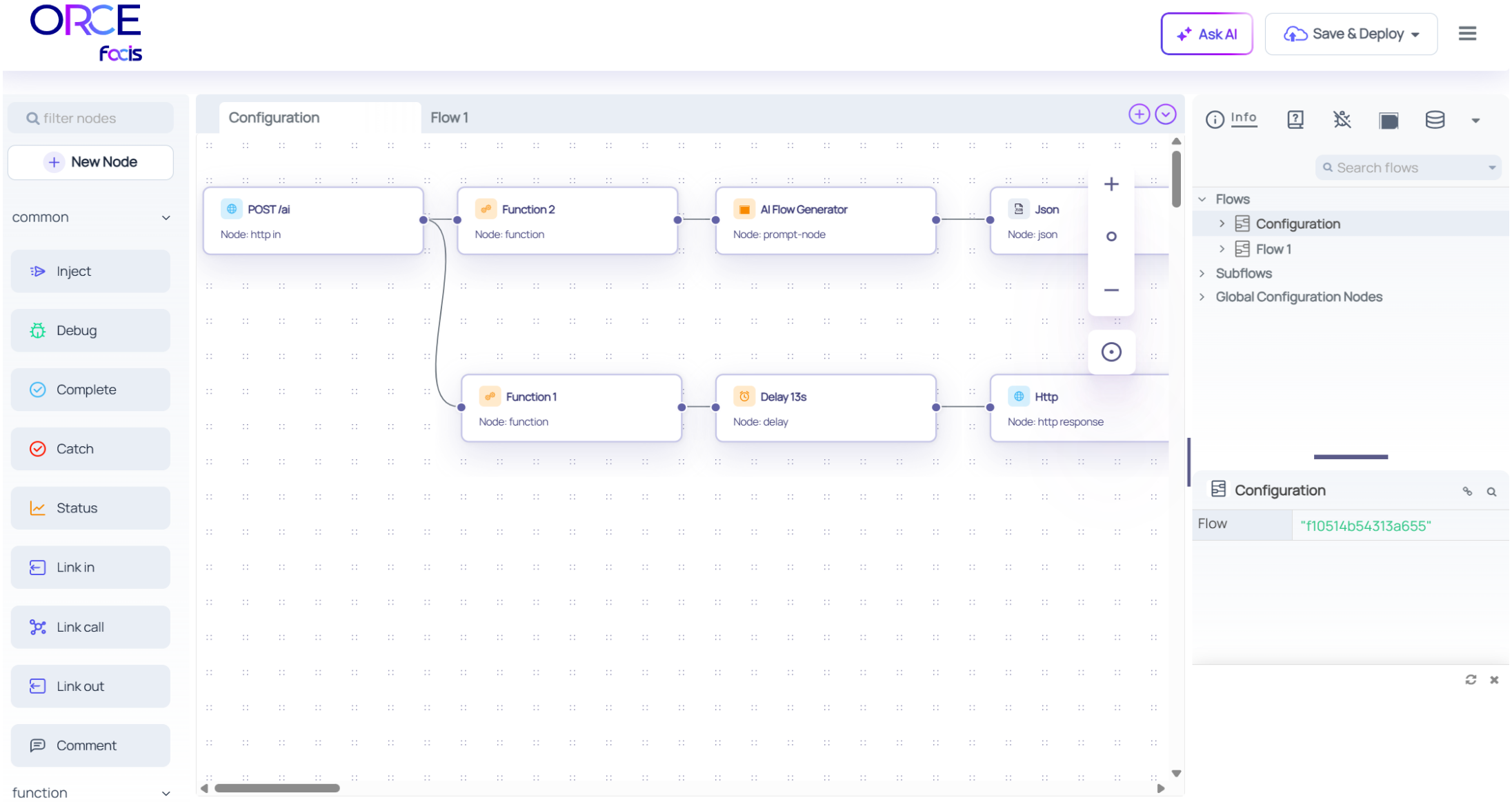
- **Central Role:** Orchestrates onboarding, accreditation, and data-sharing workflows
- **Trust & Transparency:** Logs every step for auditability and compliance
- **Federation Services Integration:** Connects to OCM, PCM, Catalogue, and more
- **Interoperability:** Enables cross-organization data exchange
- **Scalability:** Works from single provider to large federated ecosystems
- **Automation:** Reduces manual work, speeds up onboarding & validation

Integration with XFSC Ecosystem

ORCE is built to work hand-in-hand with the XFSC ecosystem, following its policies and technical standards.
It acts as the glue between federation services, providers, and consumers, ensuring secure and trusted data flows.
This integration allows organizations to collaborate while preserving sovereignty and compliance.

- **Aligned with Trust Frameworks:** Technical enforcement data sovereignty, security, and policy rules
- **Service Connectivity:** Integrates with Catalogue, PCM, OCM, and other federation components
- **Interoperability:** Supports open protocols and standard data formats for seamless exchange
- **Use Cases:** Data sharing, federated onboarding, edge-to-cloud orchestration, AI, smart industry
- **Trust & Governance:** Built-in auditing, logging, and policy enforcement
- **Scalable Collaboration:** Supports single-provider pilots up to multi-organization federations

ORCE User Interface Tour



Mini Demo

Initial Q&A

03

Design & Deployment

Hossein Rafieekhah | WP1 Lead

Prerequisites and System Requirements

Before we dive into building flows, let’s make sure everyone has the right setup.
ORCE is lightweight, but there are some minimal requirements either Docker or Node.js for local runs, and some basic cluster tools if you want to try Kubernetes deployment.

Supported OS: Windows, Linux, macOS

Docker Setup (Recommended): Docker Engine (latest stable)

Node.js Setup (Alternative): Node.js v18+ (LTS) + npm

Kubernetes (Optional): kubectl CLI + running cluster (Kind/Minikube/K3s)

Hardware: Min 2 CPU / 2 GB RAM (Recommended: 4 CPU / 4 GB+ for larger flows)

Security Note: Change default admin password after first login

Step-by-Step Installation Guide

Let’s install ORCE and make sure everyone can access the flow editor before we start building workflows.
We’ll cover three options: **Docker Quick Start**, **Local Node.js Setup**, and **Kubernetes Deployment**.
Docker is the fastest option, Node.js is good for developers, and Kubernetes is ideal for production-like setups.

Docker Quick Start — ORCE

```
$ docker pull ecofacis/xfsc-orce:2.0.0
1.0.0: Pulling ecofacis/xfsc-orce:2.0.0
Digest: sha256:…
Status: Downloaded newer image for ecofacis/xfsc-orce:2.0.0
$ docker run -d --name xfsc-orce -p 1880:1880 -v orce_data:/data ecofacis/xfsc-orce:2.0.0
f1a2b3c4d5e6…
$ docker ps --filter name=xfsc-orce --format '{{.Names}}  {{.Status}}  {{.Ports}}'
xfsc-orce  Up 10s  0.0.0.0:1880->1880/tcp
OK Open http://localhost:1880 and log in with admin / xfsc-orce (change password after first login)
```

Local Setup — Build from Source (Node.js v18+)

```
$ git clone https://github.com/eclipse-xfsc/orchestration-engine.git
$ cd orchestration-engine
$ npm install
added X packages, audited Y packages
$ npm run build
> build completed successfully
$ npm start
> ORCE started on http://localhost:1880
OK Open the URL in your browser
```

Kubernetes Deployment — ORCE

```
$ kubectl apply -f orce-deployment.yaml
deployment.apps/orce created
service/orce created
$ kubectl get pods -l app=orce
NAME        READY   STATUS    RESTARTS   AGE
orce-xxxxx  1/1     Running   0          20s
$ kubectl port-forward svc/orce 1880:1880
Forwarding from 127.0.0.1:1880 -> 1880
OK Visit http://localhost:1880 (press Ctrl+C to stop forwarding)
```

Installation & Deployment

ORCE can run almost anywhere

on a developer laptop, inside a container, or in a production Kubernetes cluster.
Deployment should be fast and repeatable, whether for a single-user test or a federated-scale setup.
Here's how we move from installation to a running, ready-to-use ORCE instance.

Docker Deployment (Quickest)

- One-liner docker run for local development
- Persistent storage with – v orce_data:/data

Local Node.js Deployment

- Clone repo → npm install → npm start
- Ideal for devs who need to tweak source code

Kubernetes Deployment

- Declarative YAML: Deployment + Service
- Expose via kubectl port-forward, NodePort, or Ingress

Configuration & First Flow

Once ORCE is running, the next step is to configure your workspace and build your first “Hello ORCE” flow.
This will help participants understand the palette, message object (msg), and deployment cycle.
The goal is to go from a blank canvas to a working, visible output in the debug panel.

Configuration & First Flow

Configure Workspace

Open http://localhost:1880 and log in
Change default admin password (Settings → Users)
(Optional) Configure Project in Settings → Projects to enable version control

Build First Flow

- Drag **Inject Node** → Configure Payload ("Hello ORCE")
- Drag **Function Node** → Add timestamp:

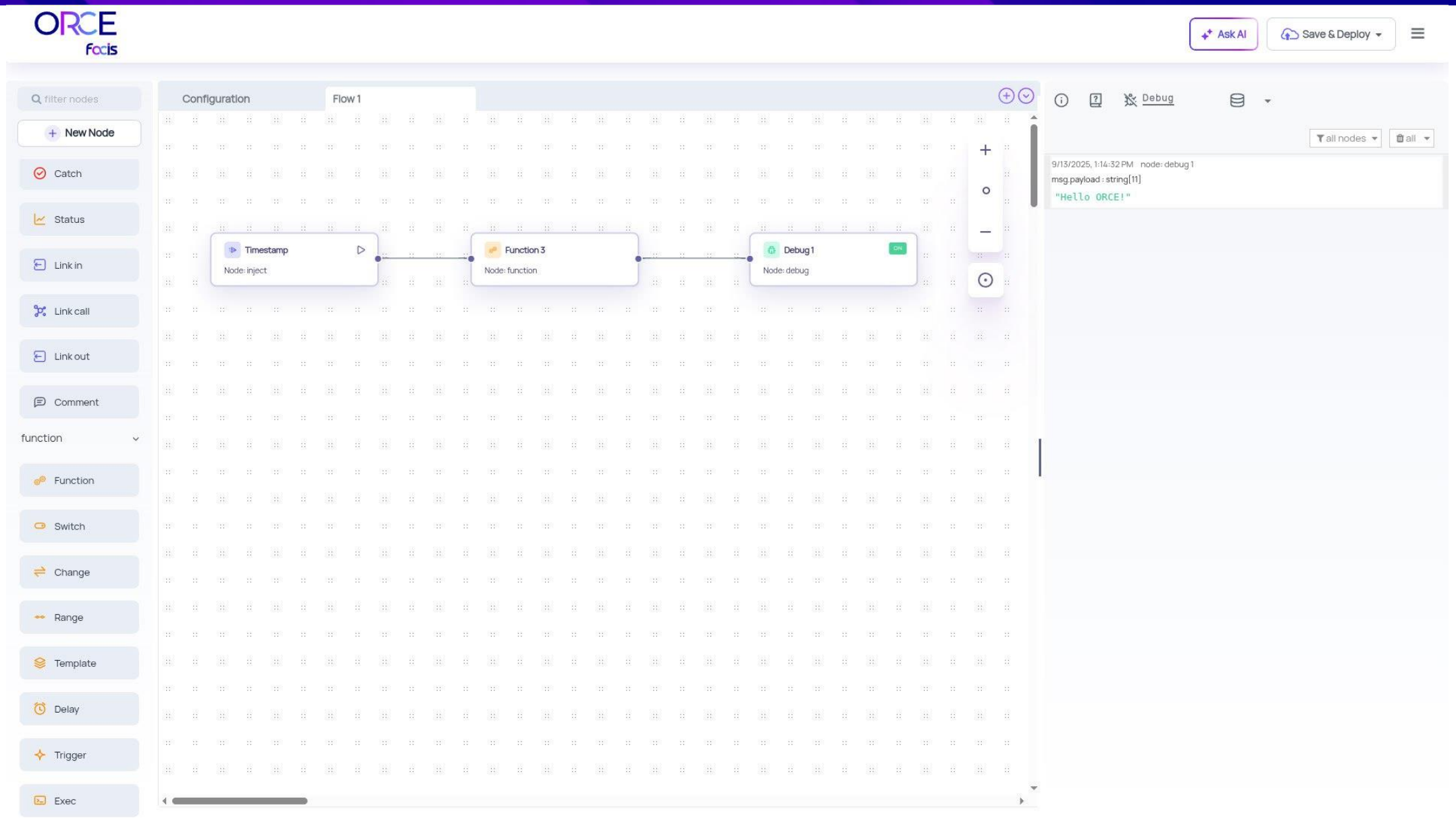
Function Node - JavaScript (ORCE)

```
// Add timestamp and return enriched msg
msg.payload = {
  text: msg.payload,
  timestamp: new Date().toISOString()
};
return msg;
```

Wire & Deploy

- Connect Inject → Function → HTTP → Debug
- Click **Deploy** (top right)
- Click the blue button on Inject node → Watch Debug panel output

ORCE Basics – Your First Flow in Action



Let's Start the Exercise!

Hands-on time – build, click, debug & have fun [

Understanding Messages (msg)

Every flow in ORCE passes a single object called msg between nodes.
Understanding the msg structure is essential it's how data travels through your workflow.
You can inspect, modify, and enrich msg in Function nodes or via Debug panel.

msg = Message Object,the container for data moving through the flow

Key Properties:

msg.payload → main data content (string, object, array, …)
msg.topic → optional label or category for routing
msg.headers → HTTP headers (when using HTTP nodes)
msg.status / msg.error → info about node state or errors

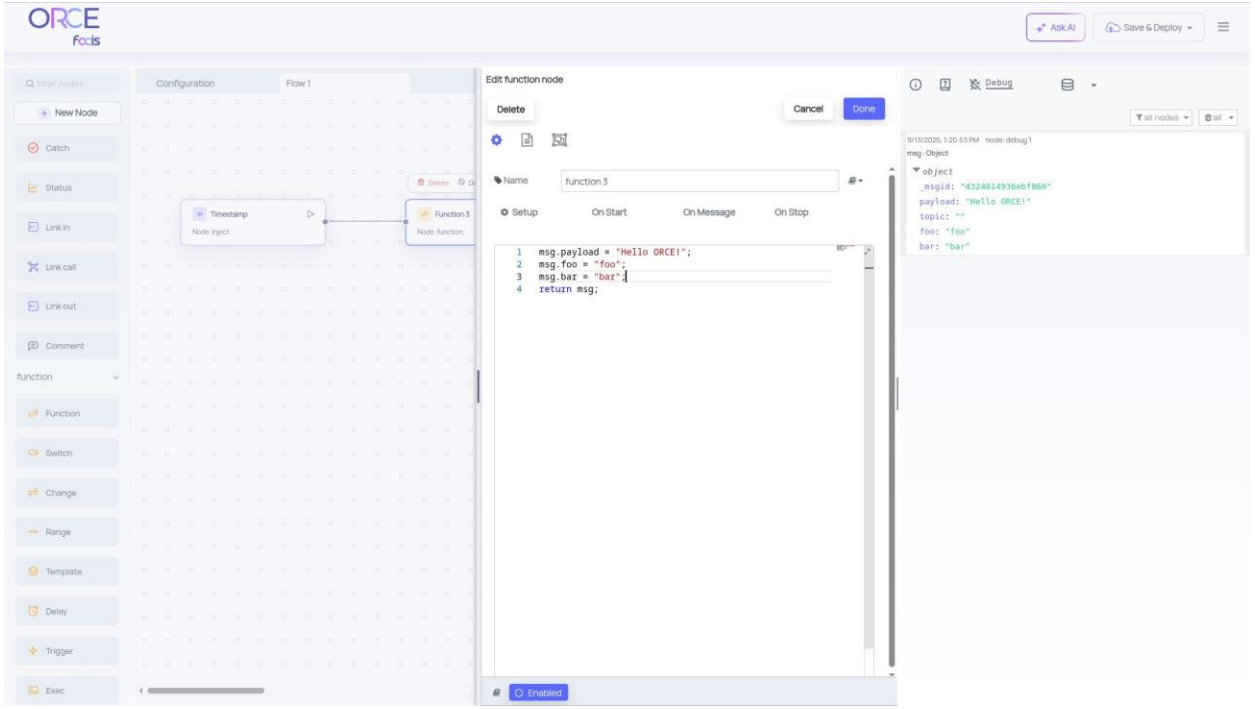
Modifiable Anywhere: Function nodes can read/change/add new **properties:**

Debug Node: shows full msg structure in real-time

Tip: Use Debug + return msg; often to verify transformations

Cloning: use RED.util.cloneMessage(msg) when you need to branch without mutating original

Context: store values in flow / global context if data must persist beyond a single message



Palette & Editor Overview (Deep Dive)

The ORCE editor is where you visually design, connect, and manage your workflows.
Understanding the palette and editor layout makes you faster and helps keep your flows clean and organized.
In this section, we'll explore the key components of the interface.

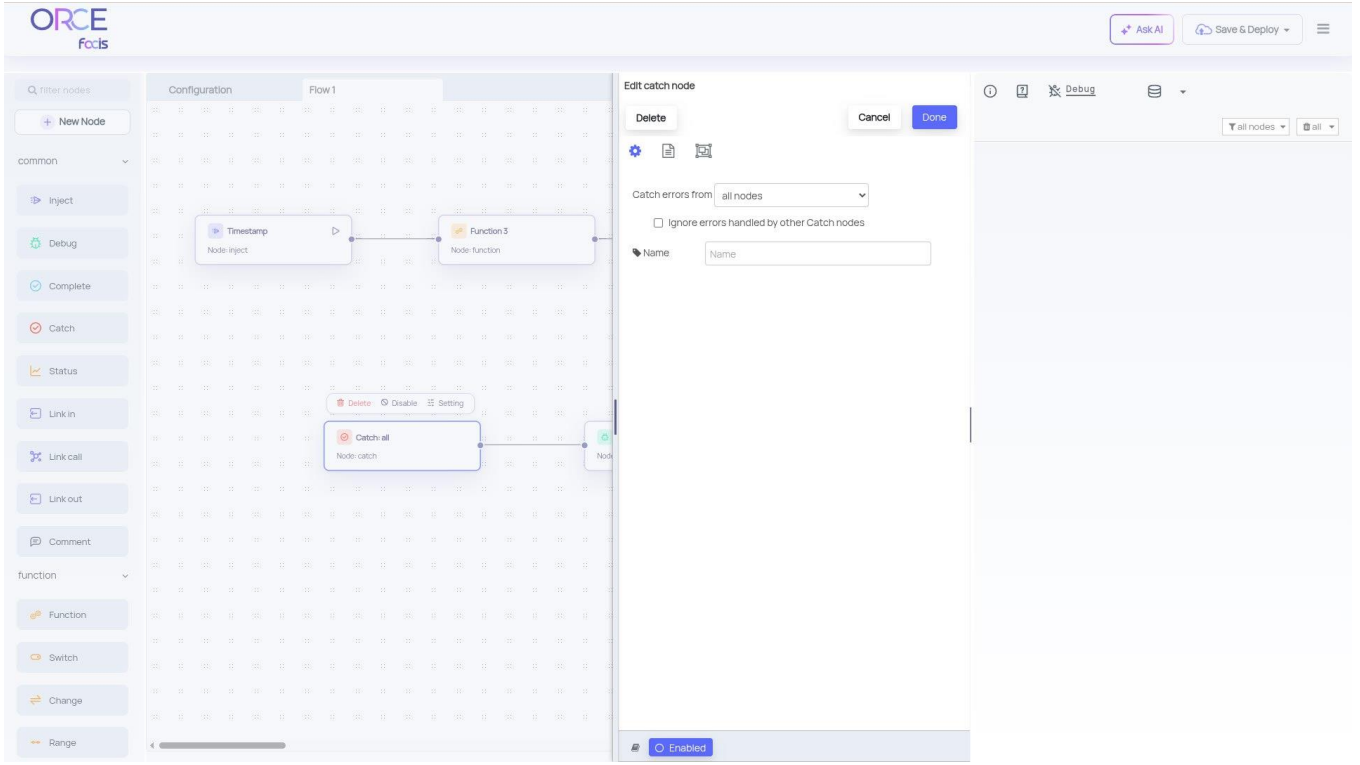
Palette:
Node categories (Input, Output, Function, Network, Dashboard…)
Quick search with the filter bar
Install new nodes from the library / manage packages

Canvas:
Main design area drag & drop nodes
Wire nodes together to define data paths
Group nodes or use Subflows for modularity

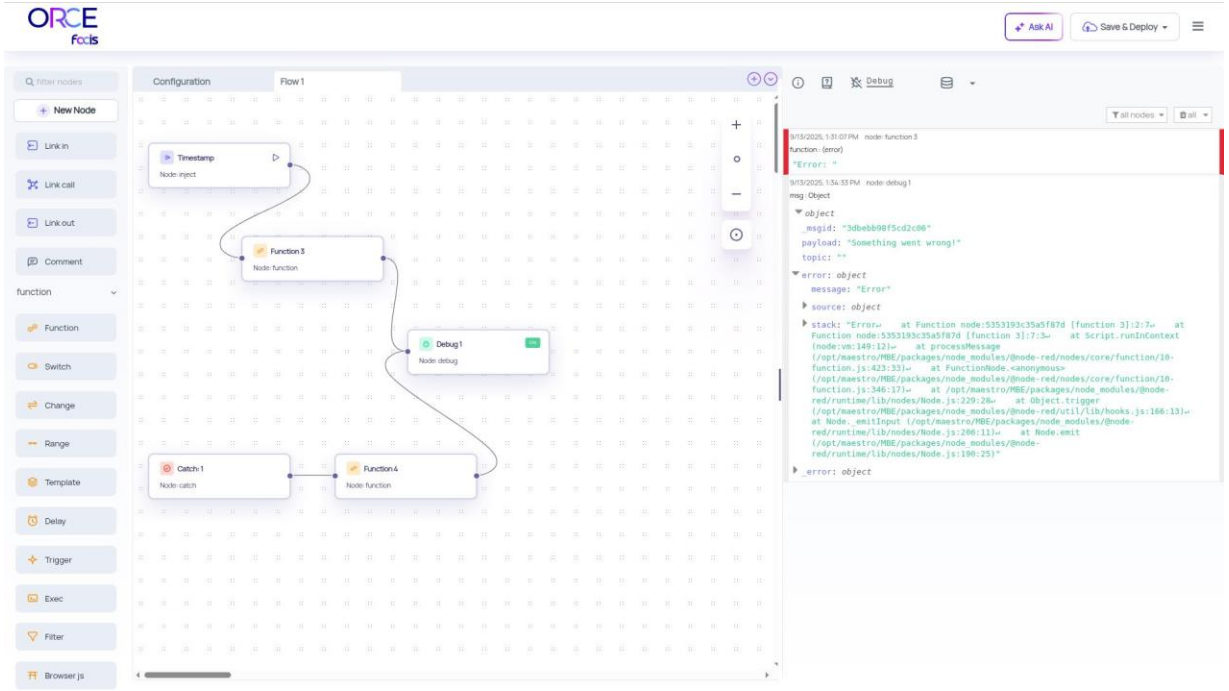
Top Toolbar:
Deploy button to apply changes
Status indicators (running, stopped, errors)
Projects menu, Import/Export JSON options

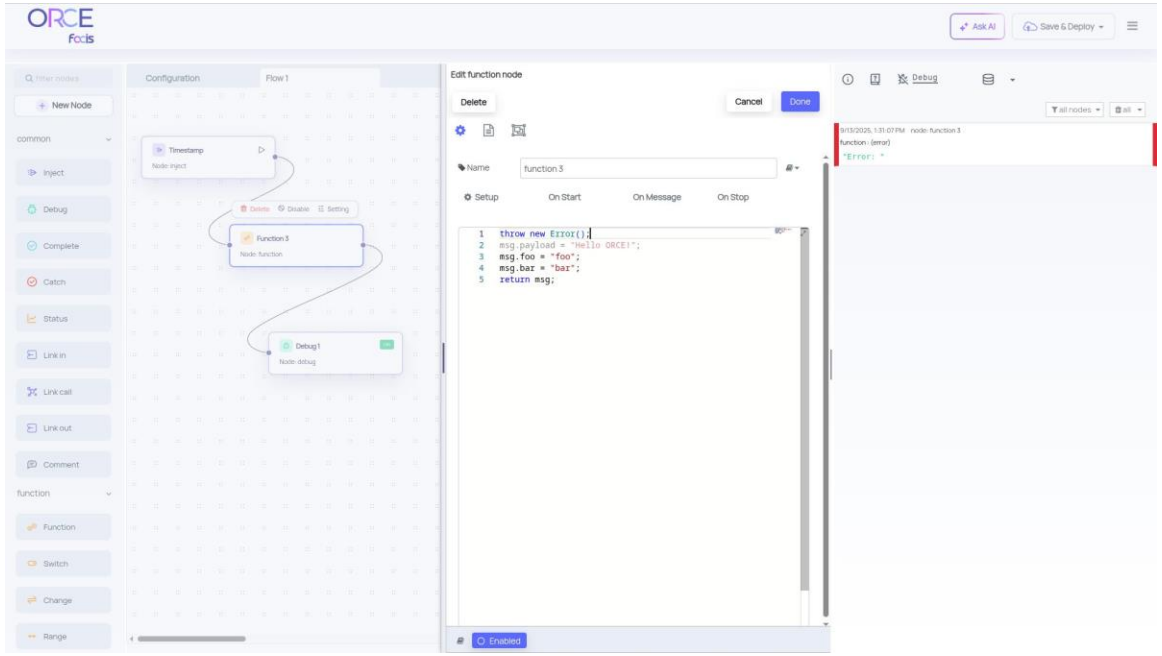
Sidebar:
Info: documentation for the selected node
Debug: view live msg output
Configuration Nodes: manage global settings

Pro Tips:
Use Snap/Align to keep flows neat
Rename nodes with meaningful names
Add Comment nodes for documentation



Error Handling Basics (use Catch node demo)





- Here is the catch node in action, it diverted the flow to a set of other instructions.

- suppose there are any kind of errors in the flow. They can be process blockers.
- in programming there is a try/catch pattern of dealing with exceptions
- you can use the catch node to just suppress any kind of errors globally or specifically to help develop the flows step by step

Tips & Best Practices for Workflow Design

Good workflow design makes your automation easier to maintain, debug, and scale.
Follow these best practices to build flows that are reliable, readable, and production-ready.
Small improvements in naming, structure, and error handling save a lot of time later.

Keep Flows Modular

- Use Subflows for repeated patterns
- Break big flows into smaller, focused segments

Name Everything Clearly

- Give nodes meaningful names (not just function1)
- Use Comment nodes to document complex logic

Test Early & Often

- Use Debug nodes on every branch during development
- Test with both normal and edge-case data

Handle Errors Gracefully

- Always include Catch nodes for critical steps
- Provide fallback values or alternate paths where possible

Use Context Wisely

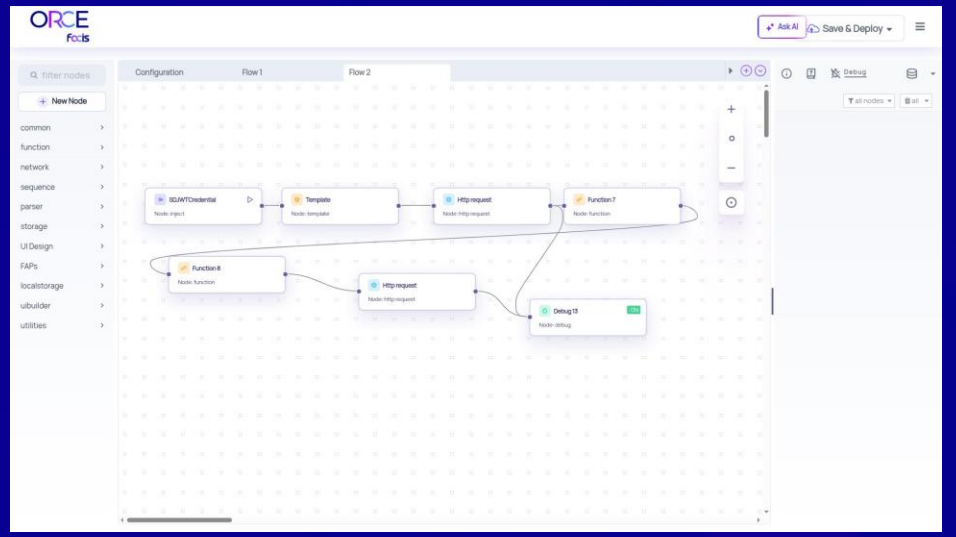
- flow context for per-flow state, global for shared state
- Don’t overload context with unnecessary data

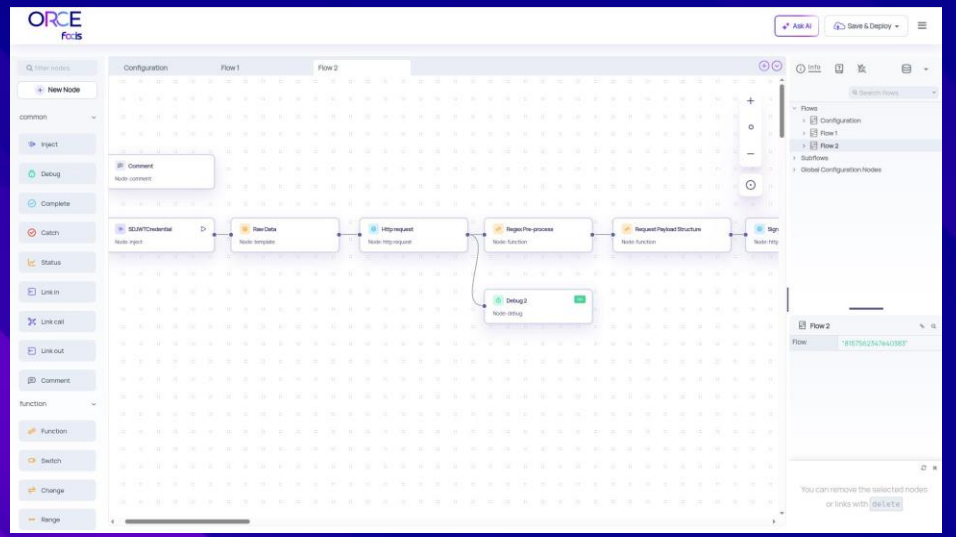
Version Control & Deploy Safely

- Enable Projects in ORCE to track changes
- Deploy incrementally and monitor Debug panel

Keep It Clean

- Align nodes and wires for readability
- Group related nodes visually, think of it as a map for future maintainers





COFFEE BREAK

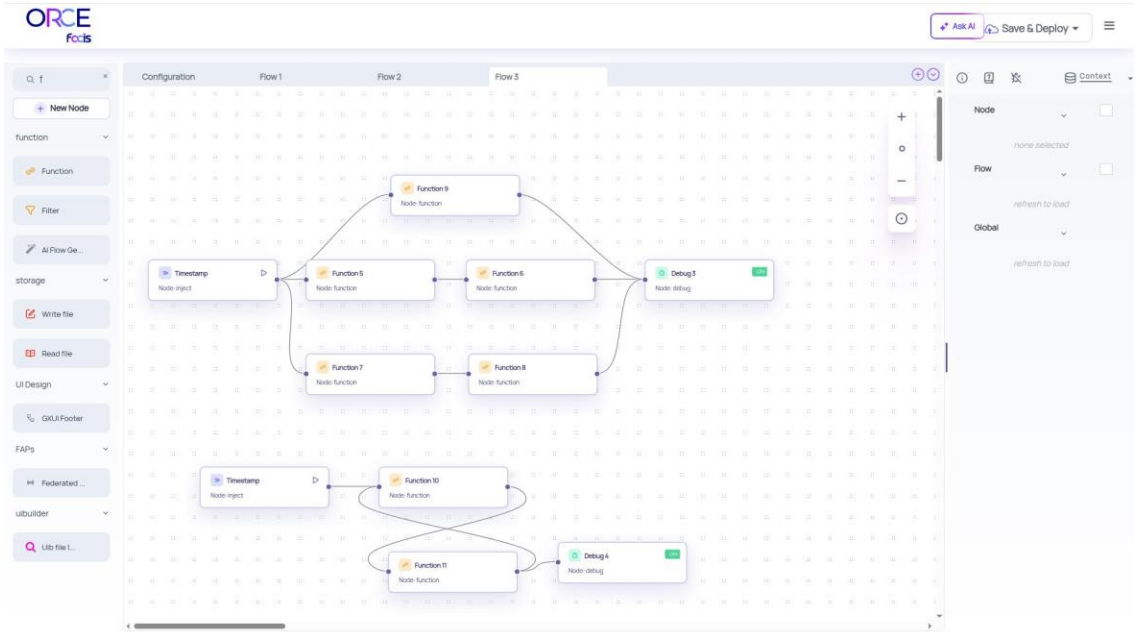
04

Advanced Topics & Hands-On

Hossein Rafieekhah | WP 1 Lead

Flow Patterns & Complex Workflows

As workflows grow, simple linear flows are not enough.
Complex automations need branching, merging, error handling, and state tracking.
ORCE supports several common flow patterns to keep designs scalable and maintainable.



Common Flow Patterns

Branching (Fan-Out):

- Split a message into multiple parallel paths
- Example: Enrich data from two different APIs simultaneously

Merging (Fan-In):

- Combine results from multiple branches back into one message
- Use Join or context storage to wait until all branches complete

Looping / Iteration:

- Use Split node + link-call or subflows for batch processing
- Ideal for handling arrays (e.g., process each provider in a list)

Error-First Pattern:

- Route errors into a dedicated Catch flow
- Optionally retry with exponential backoff, then log/alert

Delay & Rate-Limiting:

- Prevent overwhelming external systems
- Use Delay node to throttle high-frequency events

Stateful Flows:

- Track intermediate results with flow or global context
- Useful for multi-step onboarding or approval processes

Working with External APIs & Services

```
// Function Node: Build API Request
msg.method = "POST";
msg.url = env.get("ACCREDIT_URL"); // e.g. https://api.example/accredit
msg.headers = { Authorization: "Bearer " + env.get("ACCREDIT_TOKEN") };
msg.payload = { providerId: msg.payload.providerId };
return msg;

// Switch Node: Routing
if (msg.payload.status === "approved") {
    // → Create catalogue entry
} else if (msg.payload.status === "pending") {
    // → Notify + Delay(2s) → retry (max 3)
} else {
    // → Compose error → Log → HTTP 400
}

// Catch Node: Error Handling
// Send msg.error.message & msg.error.source to Debug/Log
// Optionally use Status node for red indicator
```

Most ORCE flows integrate with external services. You need a consistent pattern for calling APIs, handling responses, and enforcing security/compliance.

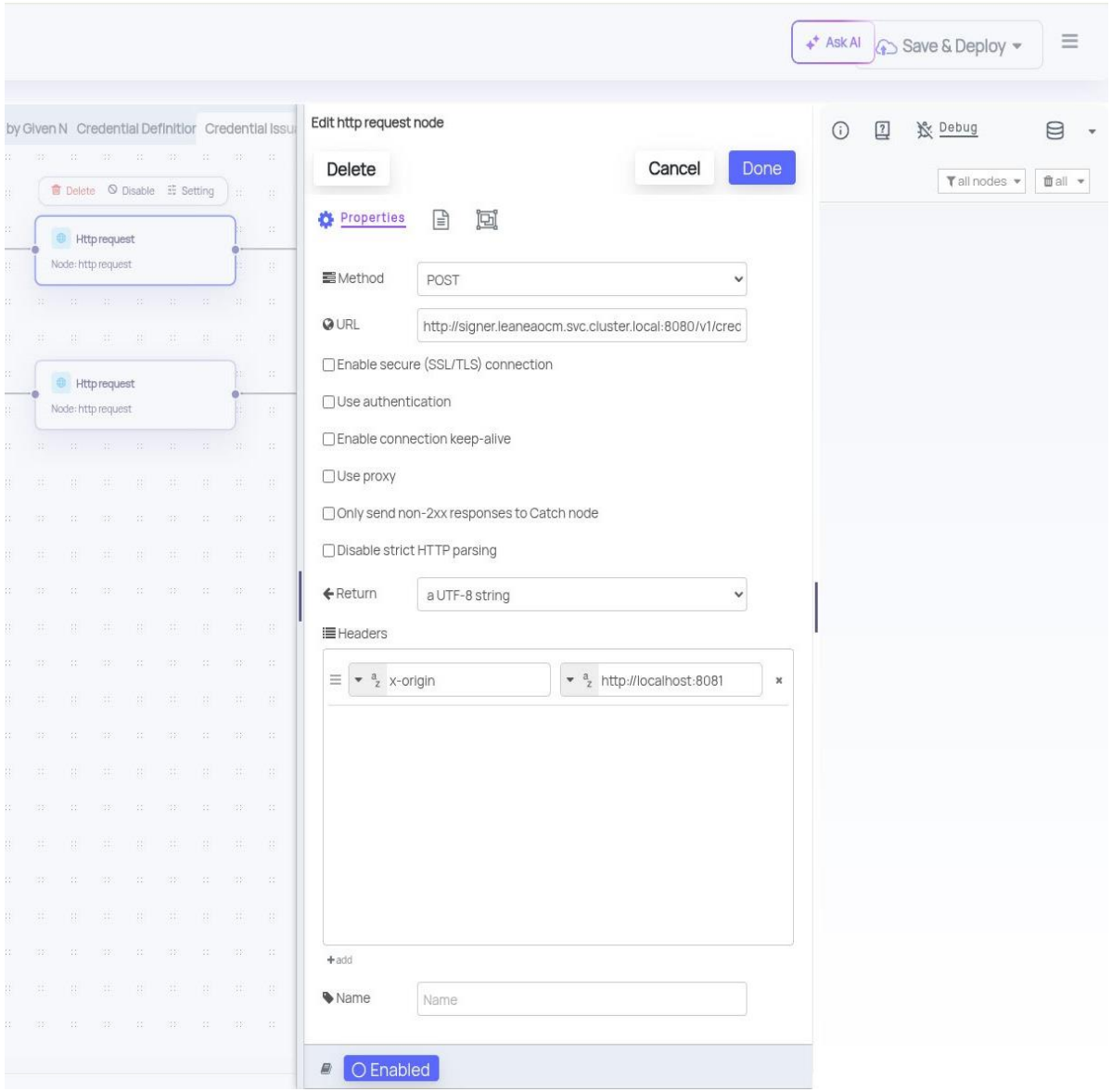
Integration pattern:

- HTTP In → Function (build request) → HTTP Request → Switch (status) → Next step / Retry / Error path
- Log everything important for auditability (Gaia-X).

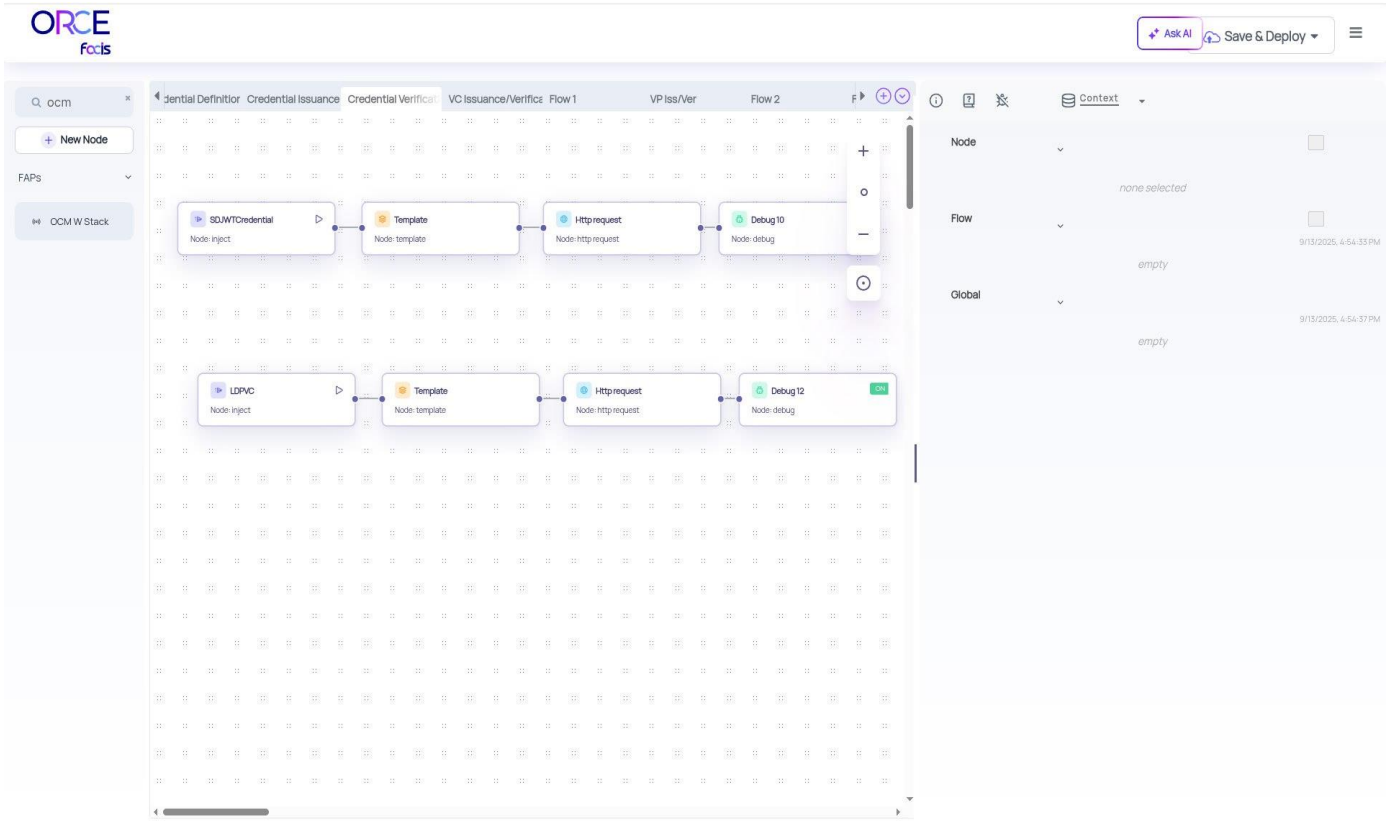
Working with External APIs & Services

Best practices

- Use credentials/env vars for tokens (never hardcode).
- Timeout + retry strategy (with caps).
- Rate limiting (Delay/Throttle) to respect quotas.
- Structured logging and Catch node for errors.
- Prefer JSON; validate schema before use.



Context & Data Persistence (Tracking Provider Status)



-in real-world onboarding flows, you often need to track a provider's progress across multiple steps.

-ORCE supports context storage so you can persist data between messages and across nodes.

-This ensures that status, counters, and intermediate results are never lost, even if flows are asynchronous.

Key Concepts

Context Types:

- msg → lives only during a single message
- flow → shared across all nodes in one flow
- global → shared across all flows in the runtime

Why Use Context:

- Track provider onboarding state (e.g., requested, approved, published)
- Store partial data (credentials, timestamps) between HTTP calls
- Avoid duplicating work if the flow restarts

Best Practices

- Reset context after a workflow completes to avoid stale data.
- Debug context: use a Debug node to print flow.keys() to see what's stored.
- Security: be careful when storing sensitive data — encrypt or hash if required.
- Persistence: enable context storage in ORCE settings (file-based or DB-backed) to keep state after a restart.

Provider → Step 1 (request) → Flow sets status=requested → Step 2 (approval) → updates context → Step 3 (published) → writes to catalogue

Context - Save Provider Status

```
// Function Node: Save Provider Status to flow context
let state = flow.get("providerStatus") || {};
state[msg.payload.providerId] = {
  status: msg.payload.status,
  updatedAt: new Date().toISOString()
};
flow.set("providerStatus", state);
return msg;
```

Context - Read Provider Status

```
// Function Node: Read Provider Status from flow context
let state = flow.get("providerStatus") || {};
msg.payload = state[msg.payload.providerId] || { status: "unknown" };
return msg;
```

Context Persistence - Settings Snippet

```
// Enable context persistence (settings.js or settings.json)
// Choose one store; here we use file-based 'default'
contextStorage: {
  default: {
    module: "localfilesystem"
  }
}
```

Scenario-Driven Mini Project

Gaia-X Wizard Workflow with validation, routing, dashboard display

Now it’s time to bring everything together!
In this mini project, participants will build a realistic **end-to-end ORCE flow** that simulates a Gaia-X provider accreditation process, including data validation, external API call, dashboard visualization, and error handling.

Learning Outcomes

Designing an end-to-end workflow (trigger → process → result)

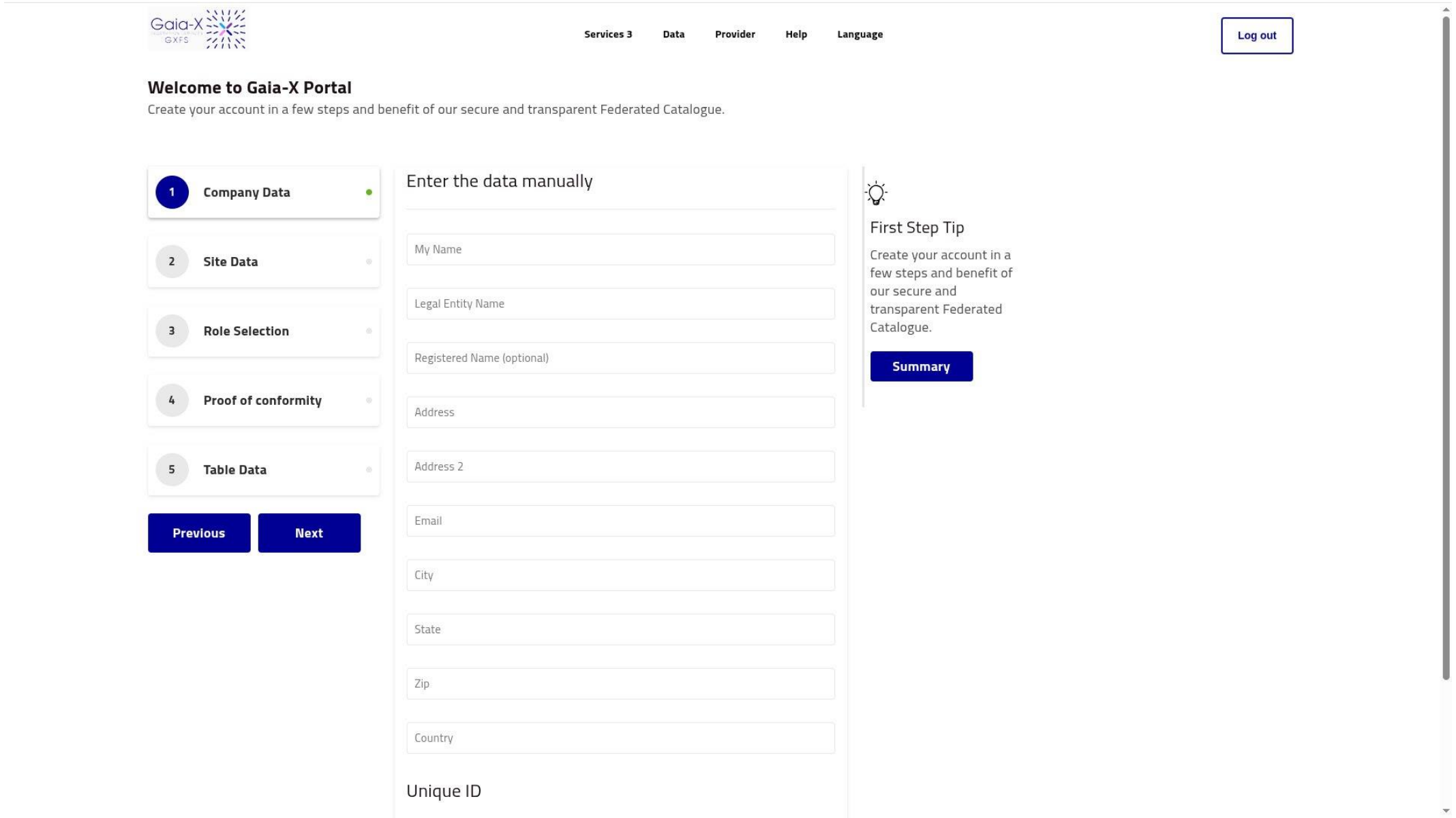
Using Function nodes for validation and transformation

Handling async flows (delay + retry logic)

Persisting data in context across multiple messages

Creating a dashboard widget to visualize provider status

ORCE Dashboard Widgets (Real-Time Status View)



Scenario-Driven Mini Project-Provider Accreditation Flow

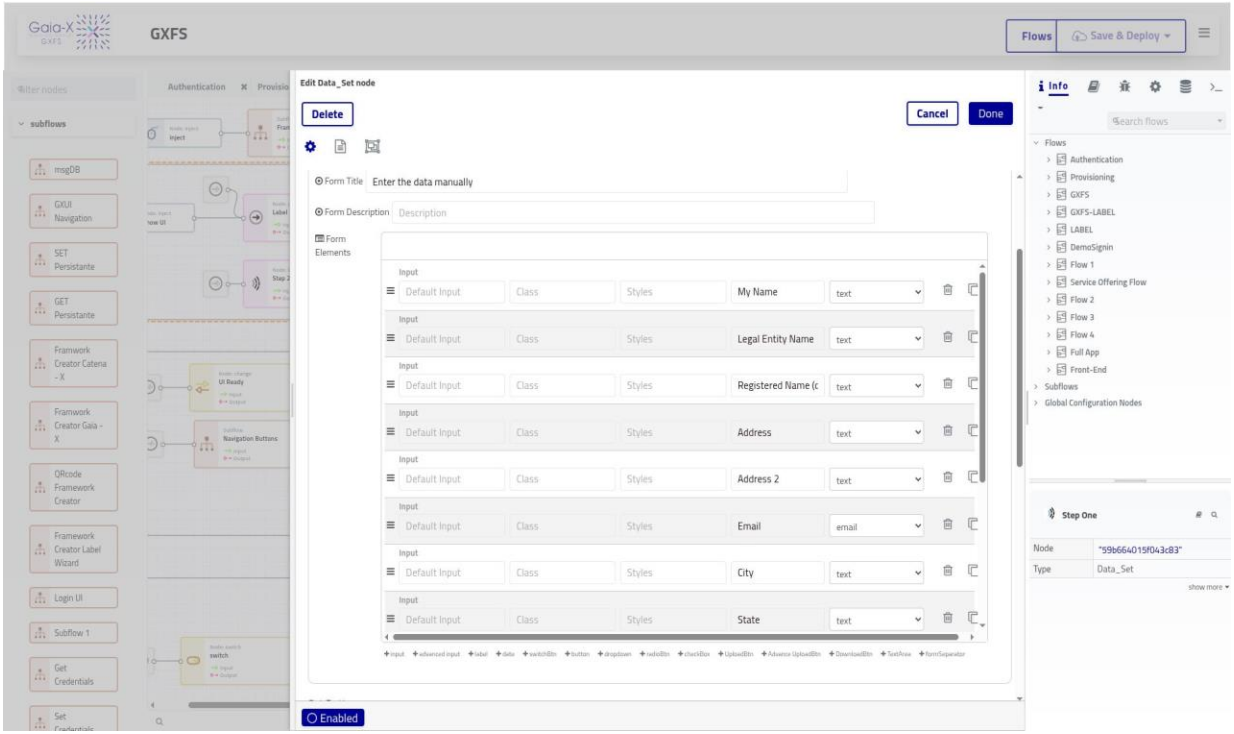
Scenario Steps

Trigger:

- HTTP In /onboard-provider receives JSON:
- { "providerId": "P-12345", "orgName": "Acme Corp" }

Validation:
Function node checks payload fields → sends error if missing.

- **Context:**
  Stores initial status = requested in **flow context.**
- **External Call:**
  HTTP Request → call **Accreditation API** → response contains status.
- **Decision (Switch Node):**
  - **Approved:** Update context, call Create Catalogue Entry API, return **200 OK.**
  - **Pending**: Notify user, Delay 10s, retry up to 3 times.
  - **Rejected:** Update context to rejected, log error, return 400 Bad Request.
- **Dashboard:**
  Send final status to dashboard widget for **live monitoring.**
- **Error Handling:**
  Catch node logs msg.error and triggers a **Status node** for visual alert.



Pro Tips & Next-Level Best Practices

Use AI to Speed Up Development

- Generate Function Node code from prompts (e.g., PII masking, validation)
- Auto-build flow templates from textual requirements

Performance & Scaling

- Use Subflows for repeated logic to reduce complexity
- Monitor memory/CPU usage when running many concurrent flows
- Deploy to Kubernetes with horizontal scaling and load balancing

Security & Governance

- Store secrets in credentials / environment variables, never hardcode
- Add audit logs for all API calls (Gaia-X compliance)
- Validate incoming data with schemas to avoid injection attacks

Testing & CI/CD

- Keep test flows for QA environments
- Automate deployments with Git integration + CI/CD pipelines

Documentation & Collaboration

- Comment nodes and use consistent naming conventions
- Use version control to track changes across team members

Generate Function Node with AI (PII masking)

Prompt:

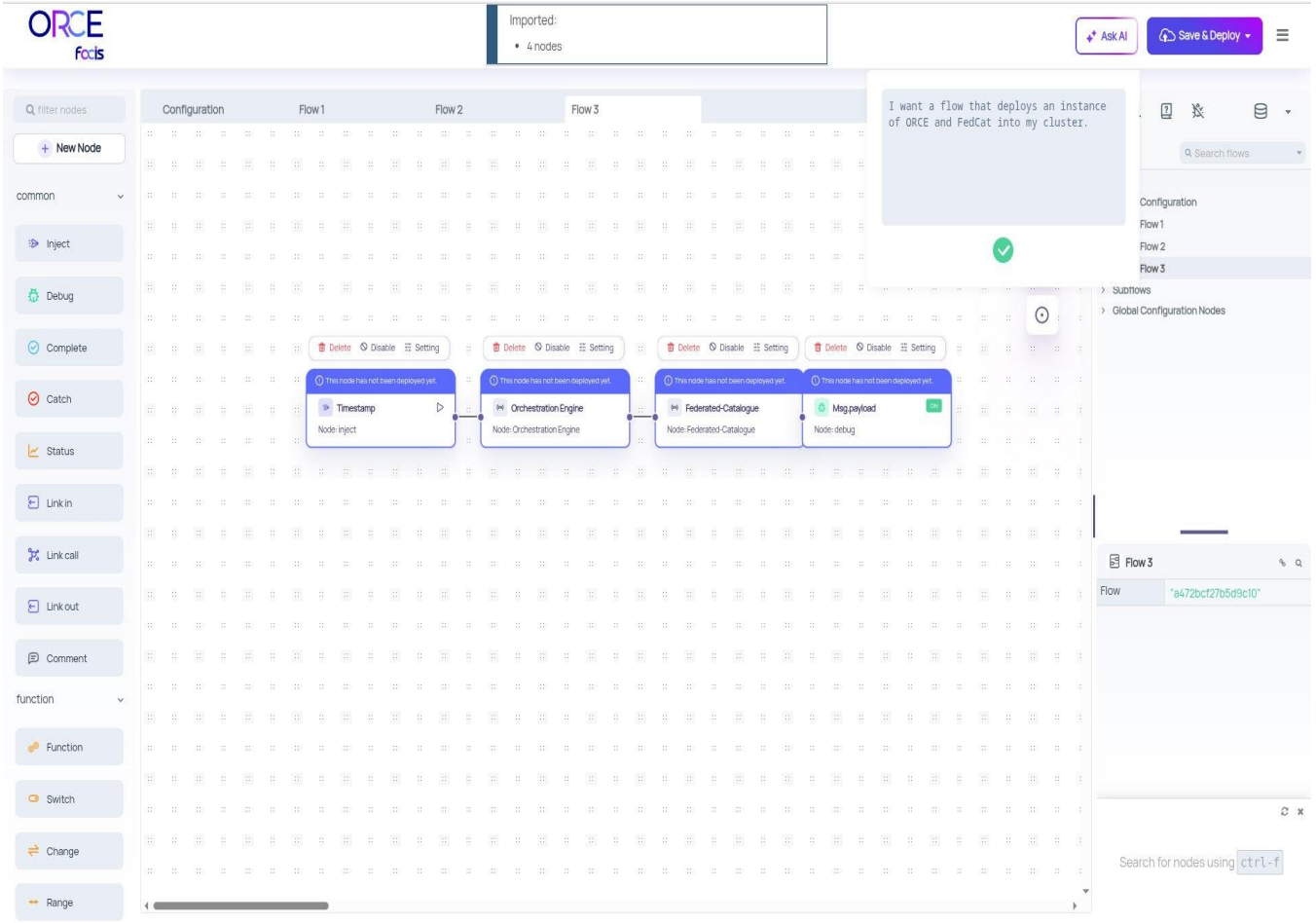
- “Write a Function Node that masks emails and phone numbers in msg.payload.”

Generated Code:

- let s = JSON.stringify(msg.payload);
- // Mask email addresses
- s = s.replace(/[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}/ig, "[email]");
- // Mask phone numbers
- s = s.replace(/\b(\+?\d[\d -]{7,}\d)\b/g, "[phone]");
- msg.payload = JSON.parse(s);
- return msg;

Steps in Demo:

- Copy-paste prompt into AI tool (or integrated node generator).
- Paste generated code into Function Node → Deploy.
- Inject sample payload with email/phone → show Debug output (masked).



Prompt-to-Flow example

Prompt:

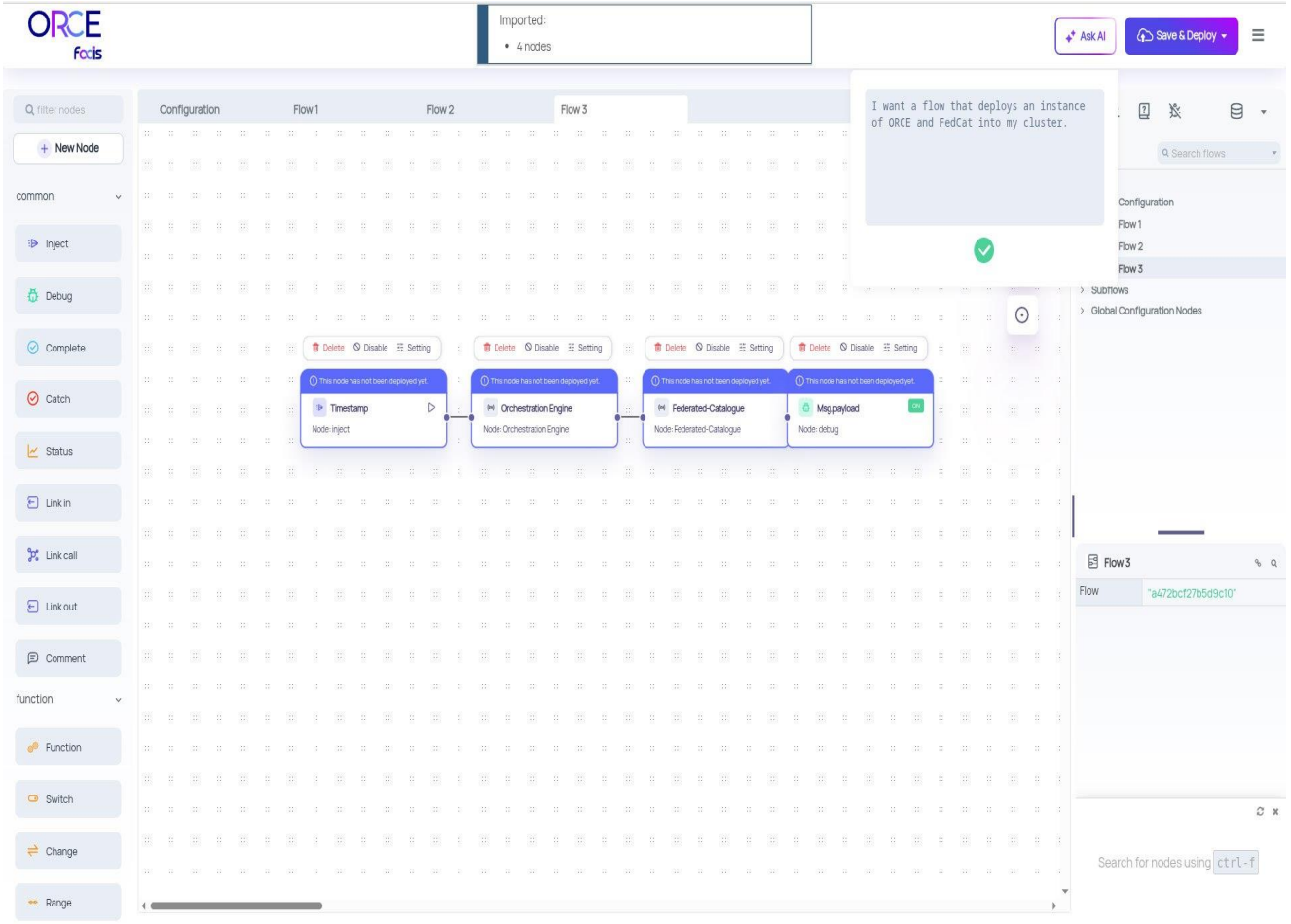
- “Build a flow that receives an HTTP POST with provider data, validates it, and logs to console.”

Expected Output:

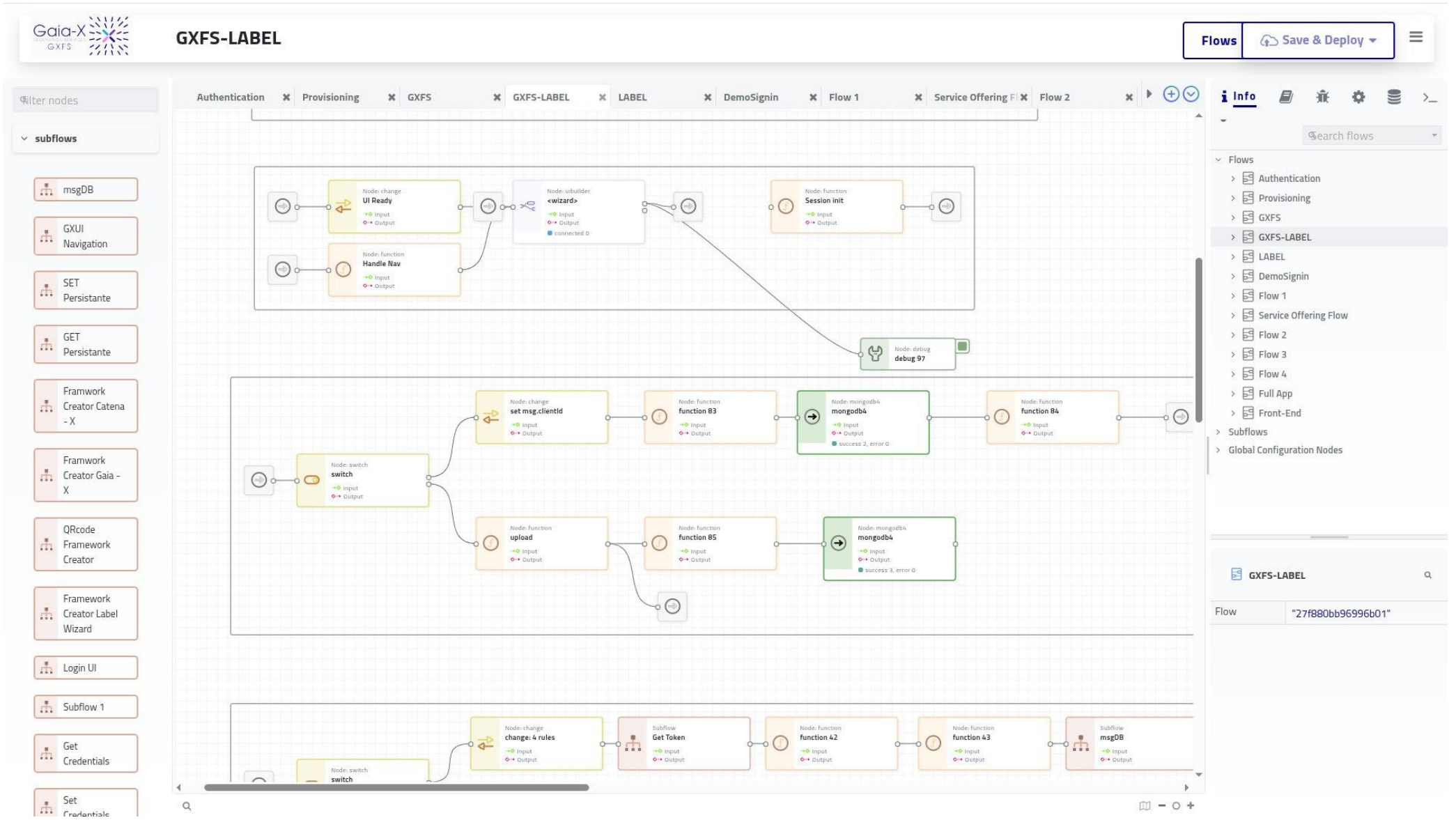
- AI generates JSON flow with:
  - HTTP In → Function (validate) → Debug
- Import JSON into ORCE editor → Deploy → Test with curl.

Key Takeaways:

- **AI speeds up development** – fewer manual steps, faster iteration
- Great for **boilerplate code**, validation logic, and flow skeletons
- Always **review & test AI output** before production
- Perfect complement to human design, not a replacement



End-to-End Showcase



Final Q&A

Get in touch with us

Homepage!

Newsletter!

info@facis.eu
www.facis.eu

CLOUD·EDGE CONTINUUM

Finanziert von der Europäischen Union
NextGenerationEU

Gefördert durch:
Bundesministerium für Wirtschaft und Energie
aufgrund eines Beschlusses des Deutschen Bundestages

Thank you for your participation!