



# NARRATE

Regenerative Resilient Smart Manufacturing Networks

## D6.1 AI Platform & Intelligent Manufacturing Custodian Early Release (R1)

2026/12/18



Grant Agreement No.: 101138094

Topic: HORIZON-CL4-2023-TWIN-TRANSITION-01-07

Call: HORIZON-CL4-2023-TWIN-TRANSITION-01

Type of action: HORIZON-RIA

# D6.1 AI PLATFORM & INTELLIGENT MANUFACTURING CUSTODIAN EARLY RELEASE (R1)

Work package	6
Task	6.1
Due date	30/11/2025
Submission date	18/12/2025
Deliverable lead	SYN
Version	1.0
Authors	Fabio Gioachin (SYN), Matteo Ronchetti (SYN)
Reviewers	David Correa (NUN), Tomás González (NUN), Jesús Ávila (NUN), María Gutiérrez (DHL)
Abstract	<p>This document defines the Software Requirements Specification (SRS) for the NARRATE Monitoring Platform, developed within the framework of the European project NARRATE – Regenerative Resilient Smart Manufacturing Networks.</p> <p>The objective of the platform is to provide supply chain stakeholders with a comprehensive solution for the detection, management, and response to risks and disruptive events that may affect production and logistics.</p>
Keywords	Intelligent Manufacturing Custodian, NARRATE Monitoring platform, Resilience, Orchestrator, Modular, Risk, Event,

---

Event-Driven, Disruption, Response Plan, Dashboard, History, Plugin, User, Building Block, Connect, Integration, Planner

---

Project co-funded by the European Commission in the Horizon Europe

---

Nature of the deliverable:	DEM
----------------------------	-----

---

Nature of the deliverable:	PU
----------------------------	----

---

## DOCUMENT REVISION HISTORY

Version	Date	Description of change	List of contributor(s)
0.1	2025/11/10	First draft	SYN
0.2	2025/12/01	Second draft	SYN
0.6	2025/12/08	Draft ready for contributors	SYN
0.7	2025/12/11	Document ready for reviewers	SYN
0.8	2025/12/15	Document reviewed	NUN
0.9	2025/12/17	Document reviewed	DHL
1.0	2025/12/18	Last revision by coordinator	AID

## STATEMENT ON MAINSTREAMING GENDER

The NARRATE consortium is committed to including gender and intersectionality as a transversal aspect in the project's activities. In line with EU guidelines and objectives, all partners – including the authors of this deliverable – recognise the importance of advancing gender analysis and sex-disaggregated data collection in the development of scientific research. Therefore, we commit to paying particular attention to including, monitoring, and periodically evaluating the participation of different genders in all activities developed within the project, including workshops, webinars and events but also surveys, interviews and research, in general. While applying a non-binary approach to data collection and promoting the participation of all genders in the activities, the partners will periodically reflect and inform about the limitations of their approach. Through an iterative learning process, they commit to plan and implement strategies that maximise the inclusion of more intersectional perspectives in their activities.

## DISCLAIMER

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Health and Digital Executive Agency (HADEA). Neither the European Union nor the granting authority can be held responsible for them.

## COPYRIGHT NOTICE

© NARRATE Consortium, 2023 - All rights reserved

No part of this publication may be translated, reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the written permission of the publisher or provided the source is acknowledged.



Except otherwise noted, the reuse of this document is authorised under the Creative Commons Attribution 4.0 International (CC BY 4.0) licence. This means that reuse is allowed provided appropriate credit is given and any changes are indicated.

NARRATE(2025). D6.1 AI Platform & Intelligent Manufacturing Custodian Early Release (R1)

The NARRATE Consortium is the following:

Participant number	Participant organisation name	Short name	Country
1	INSTITUTO TECNOLÓGICO METALMECÁNICO, MUEBLE, MADERA, EMBALAJE Y AFINES-AIDIMME	AID	ES
2	SCIENTIFIC ACADEMY FOR SERVICE TECHNOLOGY EV	SERV	DE
3	FRAUNHOFER GESELLSCHAFT ZUR FORDERUNGDER ANGEWANDTEN FORSCHUNG EV	FhG	DE
4	INSTITUT NATIONAL DES SCIENCES APPLIQUEES DE LYON	INSA	FR
4.1	INSAVALOR SA	INSA-V	FR
6	F6S NETWORK IRELAND LIMITED	F6S	IE
7	SYNESIS-SOCIETA CONSORTILE A RESPONSABILITA LIMITATA	SYN	IT
8	MEDITERRANEAN WOOD FACTORY S.L. - MEDWOOD	MED	ES
9	DHL EXEL SUPPLY CHAIN SPAIN SL	DHL	ES
10	NUNSYS SA	NUN	ES
11	BUDATEC GMBH	BUD	DE
12	BRUNEL UNIVERSITY LONDON	BUL	UK
13	POLICY LAB	PL	EE
14	BLUEBRIDGE SOLUTIONS UG	BLU	DE
16	SANTER REPLY SPA	REP	IT

## Table of Contents

1. Introduction.....	10
1.1. Purpose of the document.....	10
1.2. Tasks, Deliverable and Milestones .....	11
2. Architectures.....	11
2.1. Intelligent Manufacturing Custodian (IMC) .....	13
2.2. NARRATE Monitoring Platform (NMP) .....	14
2.3. NSQA .....	23
2.4. LLM – Large Language Models .....	24
2.5. Risks identification and Processing.....	25
2.6. Platform Library .....	26
2.7. Other Components.....	28
3. Typical workflow .....	29
3.1. Use case 1: Alternative Provider.....	31
3.2. Use case 2: Alerts and RAG data.....	45
4. Implementation & Deployment.....	51
5. Conclusions and next steps.....	52
6. References .....	53

## List of figures

Figure 1: NARRATE AI platform architecture .....	12
Figure 2: NARRATE AI platform architecture as today .....	13
Figure 3: NARRATE AI Platform Internal Architecture.....	14
Figure 4: Create an account .....	16
Figure 5: Login Page .....	16
Figure 6: IMC Chat Interface .....	17
Figure 7: Events list.....	17
Figure 8: Event Detail in a Chat .....	18
Figure 9: Response Plans list.....	19
Figure 10: Risk Types list .....	20
Figure 11: Edit Risk type.....	20
Figure 12: NMP History.....	21
Figure 13: NMP Dashboard.....	21
Figure 14: Plugins list .....	22
Figure 15: Config-Plugins .....	22
Figure 16: Event Test Simulation .....	23
Figure 17: Risk Identification Workflow [3] .....	25
Figure 18: Risk Identification Workflow.....	26
Figure 19: CEP Alert Example .....	26
Figure 20: Database and Knowledge Repository UML diagram.....	29
Figure 21: Data Flow .....	30

## List of tables

Table 1: Example of response plan.....	27
Table 2: Examples of Risk Types.....	27

## Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CEP	Complex Event Processing
DT	Digital Twin
EC	European Commission
IMC	Intelligent Manufacturing Custodian
KPI	Key Performance Indicator
LLM	Large Language Model
NMP	NARRATE Monitoring Platform
MSx	Milestone x
Mx	Month x
NSQ	Neuro Symbolic Question
NSQA	Neuro Symbolic Question Answering
RAG	Retrieval-Augmented Generation
SRS	Software Requirements Specifications
Tx.x	Task x.x
UC	Use Case
US	User Story
WP	Work Package

## EXECUTIVE SUMMARY

The document presents the Software Requirements and architectural vision of the NARRATE Monitoring Platform, an AI-driven system designed to detect, analyse, and mitigate risks in industrial supply chains and production environments. Its goal is to enhance industrial resilience by transforming heterogeneous operational events into actionable, explainable, and traceable responses, while shielding end users from the complexity of AI-based orchestration.

At the core of the platform is the Intelligent Manufacturing Custodian (IMC), built on LangChain, Large Language Models, and neuro-symbolic reasoning. The IMC interprets natural-language response plans and converts them into executable workflows composed of calls to domain-specific plugins exposed as microservices (e.g. suppliers, logistics, inventory, machines, digital twins). By combining LLM reasoning, rule-based constraints, and knowledge graphs, the IMC delivers adaptive yet auditable decision-making.

The NARRATE Monitoring Platform acts as the operational interface to this intelligence. Users interact through structured entities—Events, Risks, Response Plans, History, Dashboard, Plugins, Test, and free chat—rather than directly with the IMC. Events ingested from external detection systems via Kafka, RabbitMQ, and WebSockets are classified into risk types and linked to predefined or dynamically generated response plans. For each event, the platform automatically records the IMC's reasoning, plugin invocations, and outcomes, ensuring full traceability and explainability.

The early demonstrator focuses on two representative use cases: detection of raw-material delivery delays and prediction of machine failures to support maintenance planning. These scenarios anchor the design to real industrial disruptions. A dedicated Test module enables event simulation, replay, and operator training without affecting production data, supporting validation and continuous improvement.

From a technical perspective, the system follows a fully containerized microservice architecture with a React frontend, FastAPI backend, and PostgreSQL database. An event-driven communication model ensures scalability and decoupling, while a Metabase dashboard provides KPIs, historical analysis, and operational monitoring. Standardized APIs and modular building blocks allow the platform to evolve over time, integrating new AI models, knowledge graphs, and digital twins without re-architecture.

Overall, NARRATE delivers a unified, scalable framework for risk identification, intelligent response orchestration, and end-to-end visibility across manufacturing and supply-chain operations, enabling faster, more transparent, and more resilient industrial decision-making.

## 1. INTRODUCTION

The **NARRATE Monitoring Platform (NPM)** is a comprehensive solution whose mission is to detect, manage, and intelligently respond to risk events affecting supply chains and production processes. The execution of these processes is orchestrated by the **Intelligent Manufacturing Custodian (IMC)**.

The system aims to enhance industrial resilience by providing visibility, traceability, and actionable responses to disruptive situations that may compromise operational continuity.

The NARRATE Monitoring Platform is designed as a **modular, event-driven ecosystem** that enables the **detection, classification, and intelligent response** to risks and disruptive events in industrial supply chains and production environments, providing a comprehensive solution for value chain stakeholders.

At the core of the architecture lies the IMC (**Intelligent Manufacturing Custodian**), a central component built on **LangChain** and **Large Language Models (LLMs)** that acts as a task **orchestrator** and **planner**.

The IMC interprets response plans expressed in natural language and transforms them into structured sequences of actions executed via **plugins**.

These plugins are **APIs** deployed as **microservices**, each specialized in a specific domain (materials, products, suppliers, shipments, logistics, production, etc.), ensuring a modular, scalable, and easily maintainable design.

**The monitoring platform** connects directly to the IMC and plays a crucial role as the bridge or **interface between users and the system's intelligent core**. Users do not interact with the IMC at a technical level; instead, they access **structured entities** and views that encapsulate orchestration complexity, making the system accessible, explainable, and **user-friendly** in industrial environments.

Its architecture integrates advanced **AI capabilities**, particularly Large Language Models (LLMs) orchestrated through LangChain, with a microservices-based approach that ensures scalability, interoperability, and transparency.

The **IMC receives events** from external **detection systems** (e.g., Kafka topics, CEP engines, risk detection services) and, using natural language response plans stored in the **Platform Library**, determines the most suitable actions to mitigate or resolve the situation. These actions are executed by invoking plugins, which are APIs exposed as microservices providing domain-specific capabilities such as querying suppliers, verifying stock levels, checking logistics delays, or interacting with digital twins.

### 1.1. PURPOSE OF THE DOCUMENT

The purpose of this document is to:

- Define the Software Requirements Specification (SRS) for the NARRATE Monitoring Platform.
- Establish a **unified vision of the system's** functional and non-functional requirements.
- Describe the main entities: Events, Risks, Response Plans, History, Dashboard,

Plugins, and Test.

- Define the interaction flows between the IMC, the microservices, the users, and external systems (e.g., Kafka, CEP, risk detectors).
- Ensure that the specification enables a coherent, verifiable development, aligned with the industrial resilience objectives of NARRATE.

---

## 1.2. TASKS, DELIVERABLE AND MILESTONES

---

D6.1 delivers an early demonstrator partially integrating SMN main innovative building blocks and services including the NSQ, the AI-driven models, AI-driven sensing and production tools.

Objectives:

- O6.1: implement an interim AI Platform on an Industrial IoT framework integrating results from WPs 2, 3 & 4.
- O6.2: incorporate IMC, contextualization tools and interoperability mechanisms on top the AI Platform

At this stage of the early demonstrator, we are focusing on two specific Use Cases:

- Detect a delay in raw material delivery.
- Predict machine failures and planning maintenance.

---

## 2. ARCHITECTURES

---

The architecture of the **NARRATE AI Platform** is composed of several interconnected components, each one fulfilling a specific role within the system. Together, they enable the seamless flow from event detection to decision-making, execution of responses, and user interaction [1].

Design principles applied to all building blocks:

- **Functional encapsulation** (clear, measurable responsibilities).
- **Standardized interfaces** (API/DTPL/event bus, data formats, protocols).
- **Reusability and composability** (assembly into higher-level services/workflows).
- **Loose coupling, high cohesion** (replaceability and scalability).
- **Technology neutrality and evolvability** (plug-in LLMs/Knowledge Graphs without re-architecture).

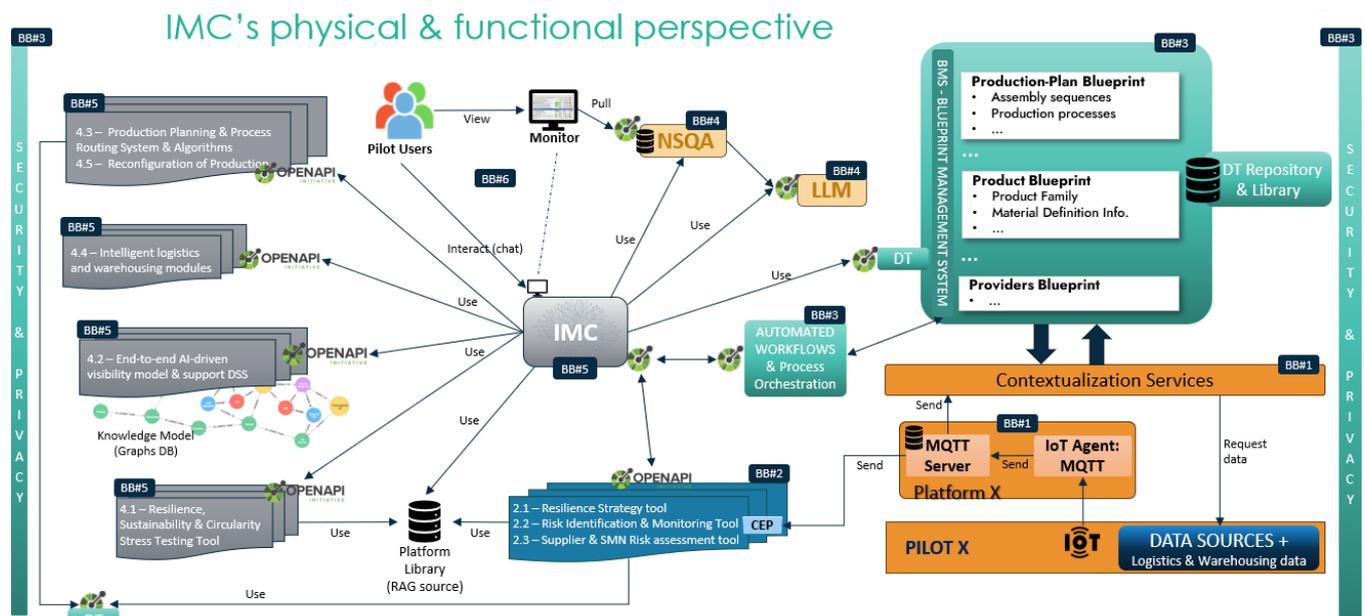


FIGURE 1: NARRATE AI PLATFORM ARCHITECTURE

Core building blocks and responsibilities (with IMC interface):

- **Data Ingestion Pipelines.** Consolidate heterogeneous streams (ERP, MES, domain tools).

**IMC interface:** registers sources and SLAs; supervises data quality and lineage.

- **Digital Twin Registry & Blueprint Frames.** Maintain state, capabilities, and constraints per supplier.

**IMC interface:** DTPL queries/updates; event synchronization; capability/constraint governance.

- **Context-Aware Reasoning Engines (AI/Knowledge Graph).** Inference, anomaly detection, simulation and scenario analysis.

**IMC interface:** requests what-if studies, risk scoring, and mitigation recommendations.

- **Adaptive Orchestration & Scheduling.** Translates goals into executable MaaS workflows; optimizes under constraints.

**IMC interface:** receives objectives/constraints; returns executable plans; triggers dynamic reconfiguration.

- **Monitoring, Observability & Resilience.** Telemetry, SLO/SLA tracking, alerting, circuit breakers.

**IMC interface:** KPI dashboards; automated mitigation and failover triggers.

**Expected outcomes:** resilient supply chains, rapid reconfigurability, programmable MaaS via DTPL, end-to-end visibility and robust decision support [1].

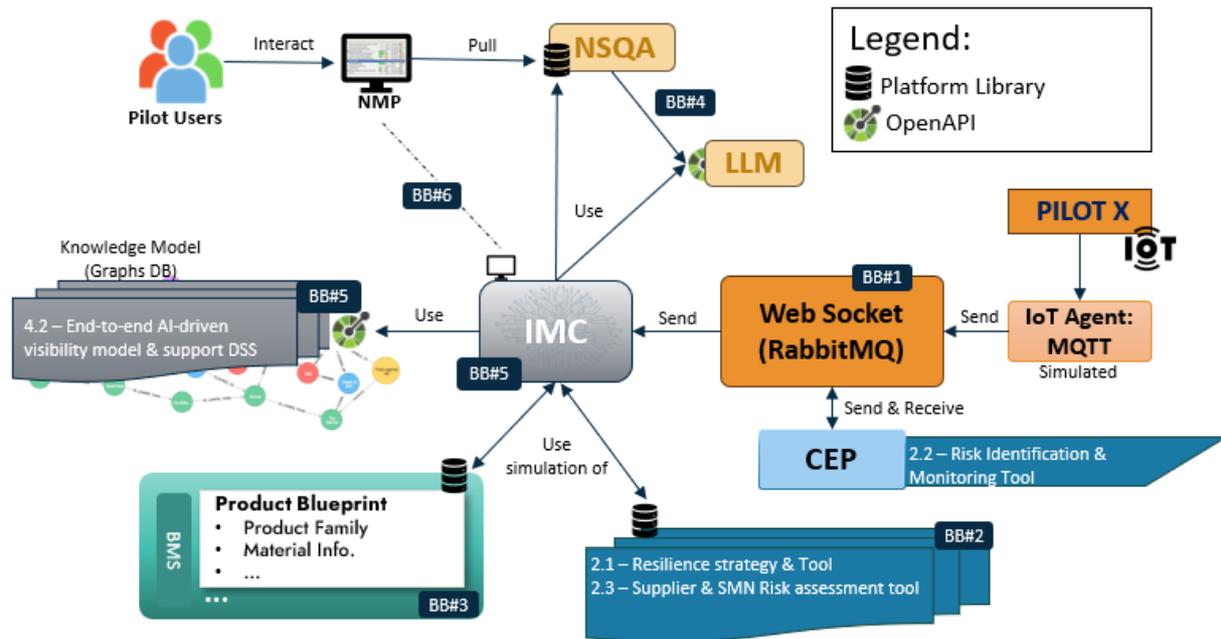


FIGURE 2: NARRATE AI PLATFORM ARCHITECTURE AS TODAY

Typical IMC ↔ building-block interaction flow integrated (end-to-end):

- **Set goals/policies** (e.g., lead time, sustainability, cost).
- **DTPL queries** Twins → retrieve harmonized state/capabilities/constraints (this interaction is simulated with mucked up blueprints repository).
- **Reasoning** → risks, scenarios, constrained optimization (the resilience strategy is simulated with mocked up Risk Types and Response Plans in the Platform Library).
- **Orchestration** → generate executable manufacturing plans and allocations.
- **Execution & monitoring** → feedback to DTs and KPI tracking to IMC.

## 2.1. INTELLIGENT MANUFACTURING CUSTODIAN (IMC)

The **Intelligent Manufacturing Custodian** is the intelligence core of the platform. It is implemented using LangChain and powered by Large Language Models (LLMs), which provide advanced reasoning and natural language understanding.

The IMC acts as a planner and orchestrator, interpreting response plans, applying rules, and coordinating plugin executions.

- **Planner:** Decomposes natural language response plans into a sequence of executable tasks.
- **LLM Reasoning:** Uses GPT-based models to interpret plans and adapt responses to the event context.
- **Rules Module:** Applies prioritization logic and operational constraints (e.g., prefer local suppliers, minimize lead times).
- **NSQA (Neuro-Symbolic Question Answering):** Enriches LLM reasoning with **graph-based ontologies**, ensuring explainability and consistency.

## 2.2. NARRATE MONITORING PLATFORM (NMP)

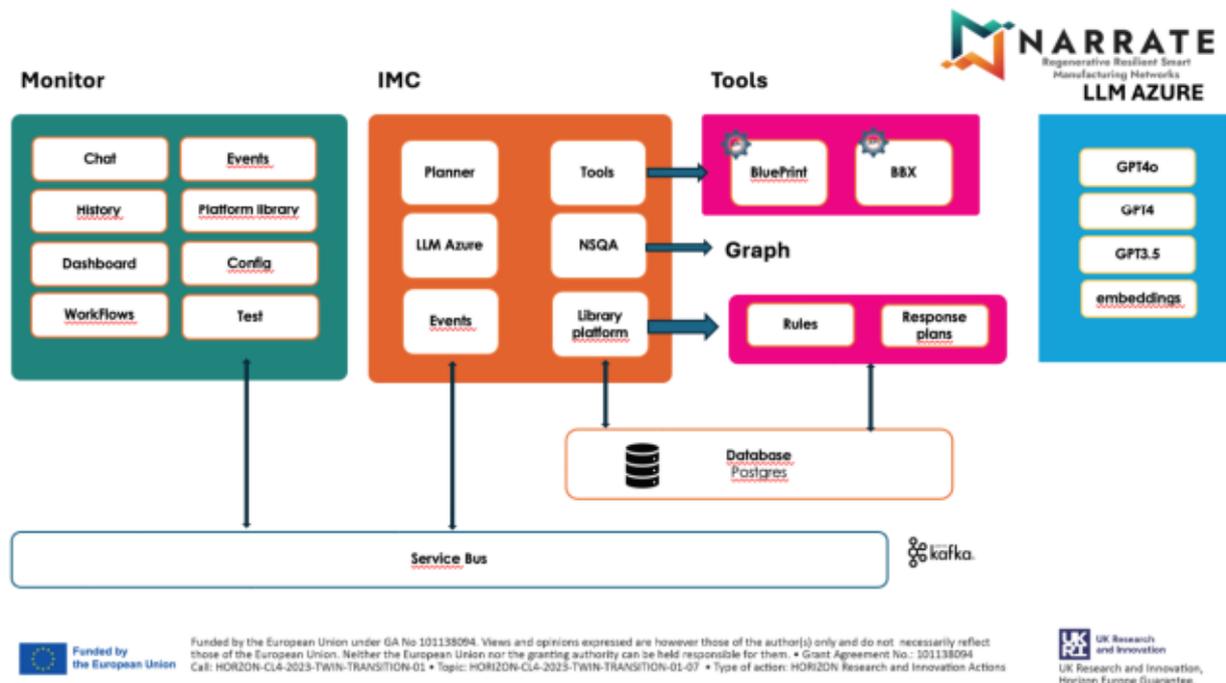


FIGURE 3: NARRATE AI PLATFORM INTERNAL ARCHITECTURE

The **NARRATE Monitoring Platform** is the bridge between users and the IMC. It abstracts the complexity of orchestration and provides intuitive access to the system's intelligence through structured entities — such as Dashboard, Chat, History and Test.

Each entity encapsulates a specific function, allowing users to monitor risks, track system behaviour, simulate scenarios, or consult the IMC through natural language queries. These interaction capabilities constitute the system's entry point.

The main system entities of the **NARRATE Monitoring Platform** are:

- **Events:** Represent incidents or occurrences registered in the platform. Events are automatically received through RabbitMQ/kafka, integrating external sources such as project Binding Blocks (CEP or other risk detection systems).

Each event is classified according to a risk type and associated with a response plan from the Platform Library. Events may correspond to a natural disaster, product defect, or supplier issue. Each **event automatically generates** an associated **chat** documenting the IMC's reasoning, the applied plan, and the invoked plugins. This ensures full and auditable traceability of the process.

- **Risks:** Represent the catalogue of risk types managed by the system. Each risk is **linked to a response plan** that specifies in natural language the actions the IMC must take when such a risk is detected. This **standardizes risk management** and provides both users and the IMC with a **common framework** for incident resolution.
- **Response Plans:** The set of natural language **instructions** that guide the IMC in acting against a specific risk. Stored in the **Platform Library**, these plans include the steps to follow, decision criteria, prioritization rules, and the plugins to be invoked. By leveraging LLMs, the IMC interprets these plans and **adapts** their execution **to the context** of each event.

- **History:** Serves as the **system's traceability repository**, recording all chats and IMC interactions. Each event generates its own history, documenting from the initial query to plugin execution and final decisions. History is critical for auditing, decision explainability, and continuous improvement.
- **Dashboard:** The platform's control panel, implemented in Metabase, offering a global view of the system's performance. The Dashboard provides metrics from high-level **KPIs** to detailed event histories, such as the number of events processed, their types, IMC response times, executed plans, and recommendations approved or overridden by users. It is directly connected to the IMC's database, ensuring consistent and up-to-date information.
- **Plugins:** The ecosystem of microservices that enable the IMC to execute actions. Each plugin exposes a **REST/OpenAPI** providing specific functionalities such as inventory queries, supplier alternatives, logistics delays, or warehouse stock checks. Their design as **independent microservices** ensures modularity, easy deployment, and seamless integration with external systems (ERP, WMS, PLM, Digital Twins, etc.), providing additional event and context data. Each plugin can evolve independently while maintaining interoperability through OpenAPI specifications.
- **Test:** A dedicated section for **event simulation** used to validate IMC behaviour and overall system functionality. It allows generating artificial events that follow the same workflow as real ones, testing response plan execution and plugin integration. This provides a **safe environment** for experimentation, training, and performance testing.
- **Free Chat with the IMC:** In addition to events, users can interact directly with the IMC via natural language queries not tied to specific events. In this case, the IMC also leverages available plugins to generate responses, extending the platform's intelligence beyond risk management.

Every request—whether generated by an event or initiated through the Free Chat—is routed through the Service Bus to the decision core.

All decisions and executions are logged in the History and feed the Dashboard, while risks and plans are managed and versioned in the Platform Library with persistence in the Postgres database. To validate end-to-end behaviour without impacting production data, the Test module enables simulations and event replays along the same application flow.

As shown in the following figures, all previously mentioned entities have been integrated and are now operating within the NMP. The subsequent figures illustrate the fundamental functioning of the platform, with a focus on the relevant User Stories, to enable the Pilot to understand how to interact directly with the NMP and indirectly with the IMC.

## 1. Login (Single Admin)

As a Pilot User, I can register a new account. At this stage of the development the Admin User Role is the only one existing. (FIGURE 4)

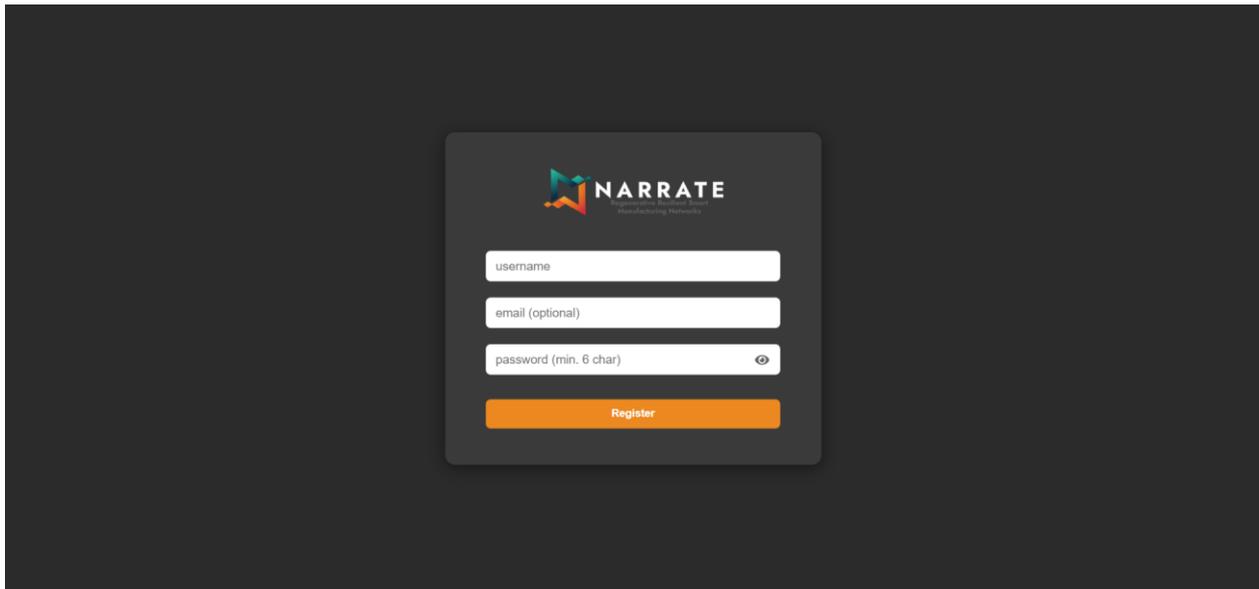


FIGURE 4: CREATE AN ACCOUNT

As a Pilot User, I want to log in with my credentials so that I can access the platform. (FIGURE 5)

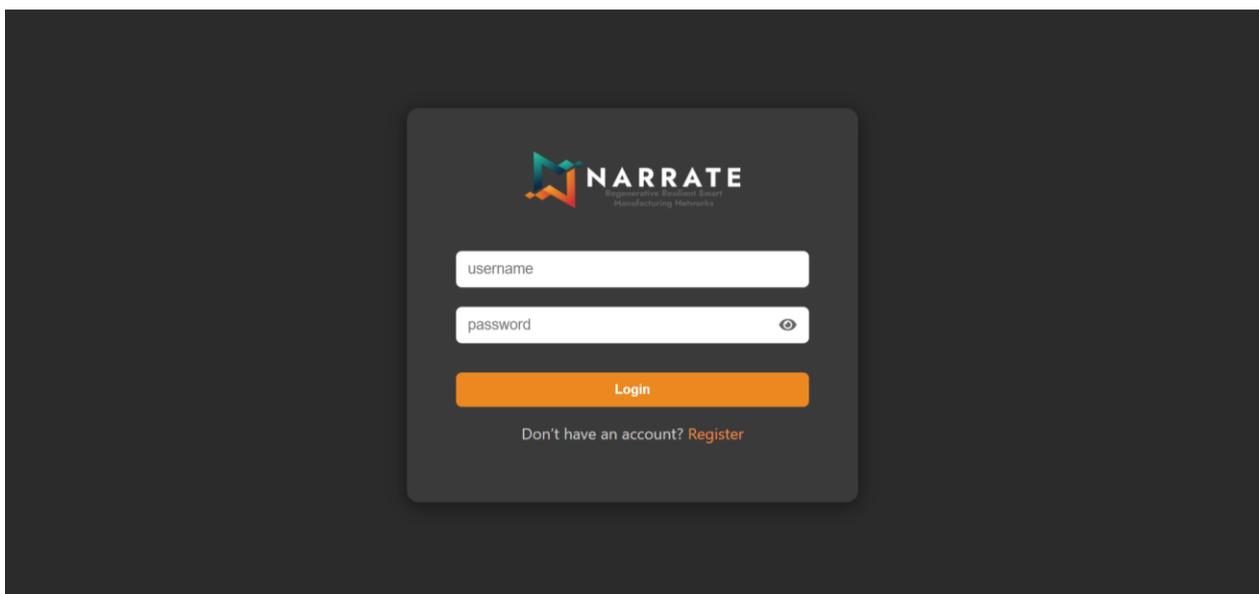


FIGURE 5: LOGIN PAGE

## 2. Home page – Ask to IMC (chat)

As soon as I Log in, the Home Page appears like an AI assistant chat interface.

As a Pilot User, if I want to ask questions to the IMC through a chat interface, I can directly insert my prompt in natural language inside the text box.

The response of the IMC is provided in the same chat interface as the question, and the conversation is stored in the History (accessible with the sidebar menu), allowing the user to open multiple chats and assessing multiple events at the same time. (FIGURE 6)

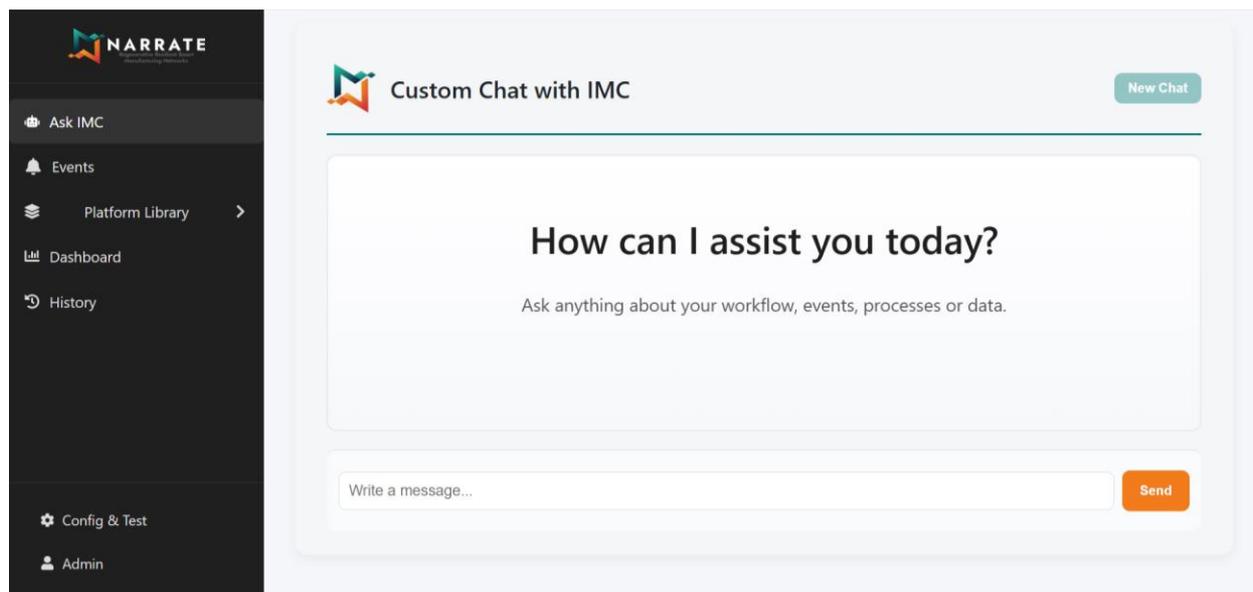


FIGURE 6: IMC CHAT INTERFACE

### 3. Events (Event List)

As a Pilot User, I want to see a real-time list of events to monitor incidents.

With the sidebar menu I can access the Event tab. In this view I can see the last disruptive events occurred, their description and their source. This view is updating in real time. It contains events associated with chat initiated by the User also.

I can filter the events base on the parameters available. (FIGURE 7)

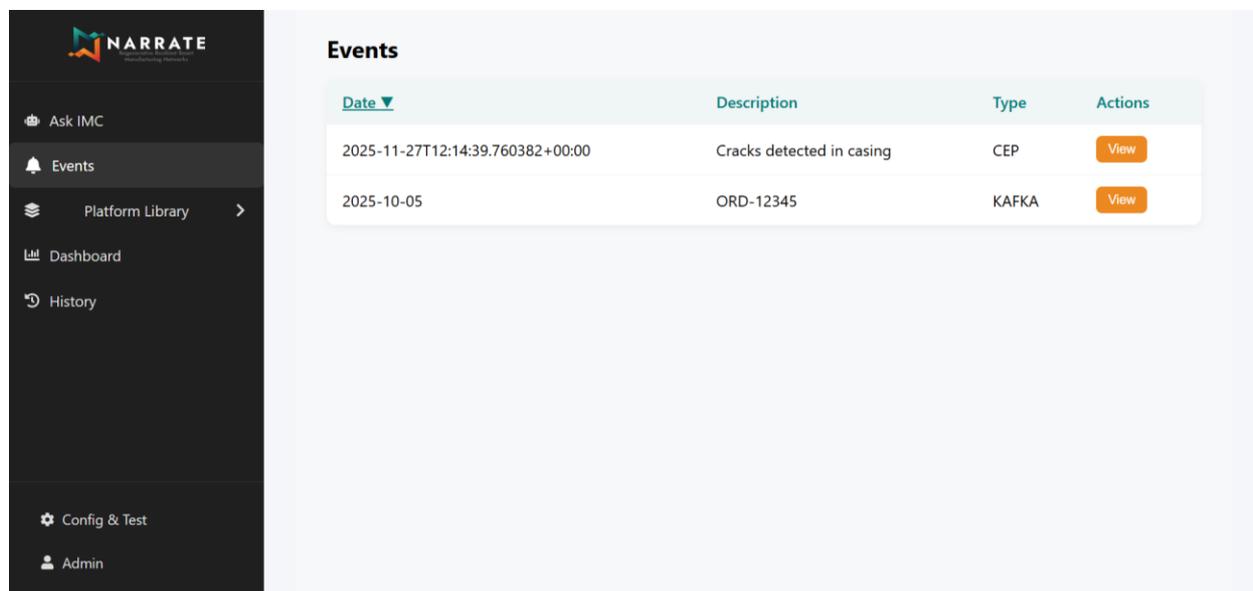


FIGURE 7: EVENTS LIST

### 4. Event Detail + Chat

As a Pilot User, I want to open an event and see details and chat to review IMC actions. Here I can see the response of the IMC in natural language, that understood the disruptive event based on the context provided, classified the risk type and searched for the right response plan to suggest.

I can follow the reasoning steps and the plugin used, reducing the chance hallucinations. (FIGURE 8)

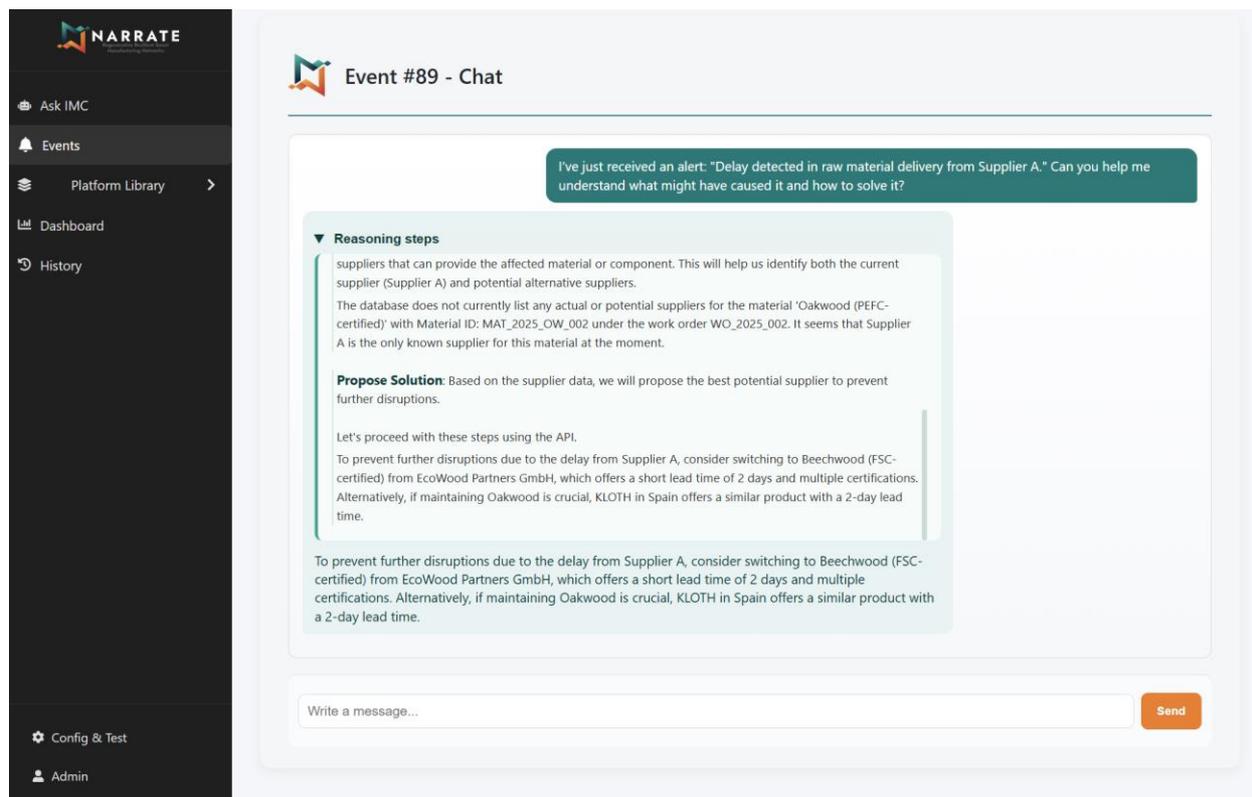


FIGURE 8: EVENT DETAIL IN A CHAT

### Query:

I've just received an alert: "Delay detected in raw material delivery from Supplier A." Can you help me understand what might have caused it and how to solve it?

### Response:

To prevent further disruptions due to the delay from Supplier A, consider switching to Beechwood (FSC-certified) from EcoWood Partners GmbH, which offers a short lead time of 2 days and multiple certifications. Alternatively, if maintaining Oakwood is crucial, KLOTH in Spain offers a similar product with a 2-day lead time.

### Reasoning steps:

**Check All Orders:** We need to retrieve all orders to identify which order is affected by the delay. Specifically, we need to find the order that includes the raw material supplied by Supplier A. Once identified, we will note the `productID` and `workOrderID` related to that order.

The order affected by the delay from Supplier A is: - Order ID: SO\_2025\_002 - Product ID: PROD\_2025\_002 - Work Order ID: WO\_2025\_002

**Get Product Details:** Using the `productID` from the affected order, we will retrieve detailed information about the product. This includes the components and materials used in the product. We need to identify the specific `materialID` or `componentID` that is affected by the delay.

The affected product, 'Premium Changing Table' (Product ID: PROD\_2025\_002), uses the material 'Oakwood (PEFC-certified)' with Material ID: MAT\_2025\_OW\_002. This material is used in both the Drawer Assembly and the Changing Table Frame, and is supplied by Premium Timber Imports from Poland. The delay from Supplier A affects this material.

**Find Suppliers:** With the `materialID` or `componentID` and the `workOrderID`, we will search for all suppliers that can provide the affected material or component. This will help us identify both the current supplier (Supplier A) and potential alternative suppliers.

The database does not currently list any actual or potential suppliers for the material 'Oakwood (PEFC-certified)' with Material ID: MAT\_2025\_OW\_002 under the work order WO\_2025\_002. It seems that Supplier A is the only known supplier for this material at the moment.

**Propose Solution:** Based on the supplier data, we will propose the best potential supplier to prevent further disruptions. Let's proceed with these steps using the API.

To prevent further disruptions due to the delay from Supplier A, consider switching to Beechwood (FSC-certified) from EcoWood Partners GmbH, which offers a short lead time of 2 days and multiple certifications. Alternatively, if maintaining Oakwood is crucial, KLOTH in Spain offers a similar product with a 2-day lead time.

## 5. Platform Library – Response Plan

As a Pilot Admin (or Risk manager) User, I want to create and edit response plans so the IMC can resolve events.

I can access the correct tab from the side menu, from the Platform Library section, I can select "Response Plan". The Response Plans have their name and description as parameters, they also have the chance to be added, edited or deleted. (FIGURE 9)

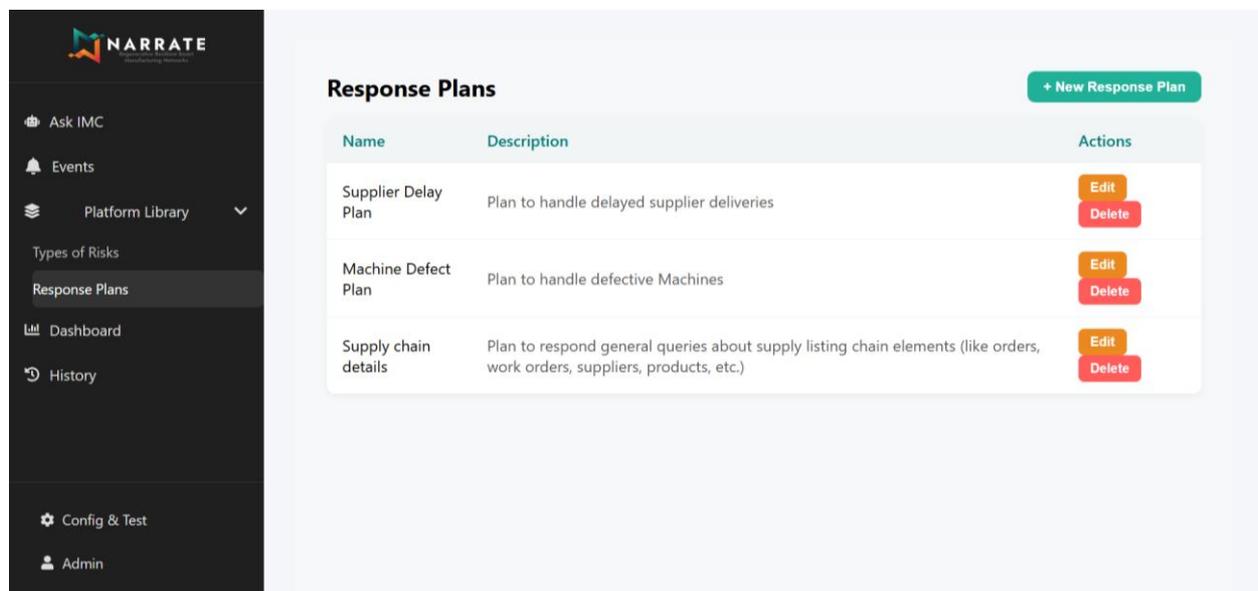


FIGURE 9: RESPONSE PLANS LIST

## 6. Platform Library – Type of Risks

As a Pilot Admin (or Risk Manager) User, I want to define and maintain risk types so the IMC can classify events. The Risk Types have Name, Description, Severity and at least one associated executable Response Plan as parameters. (FIGURE 10)

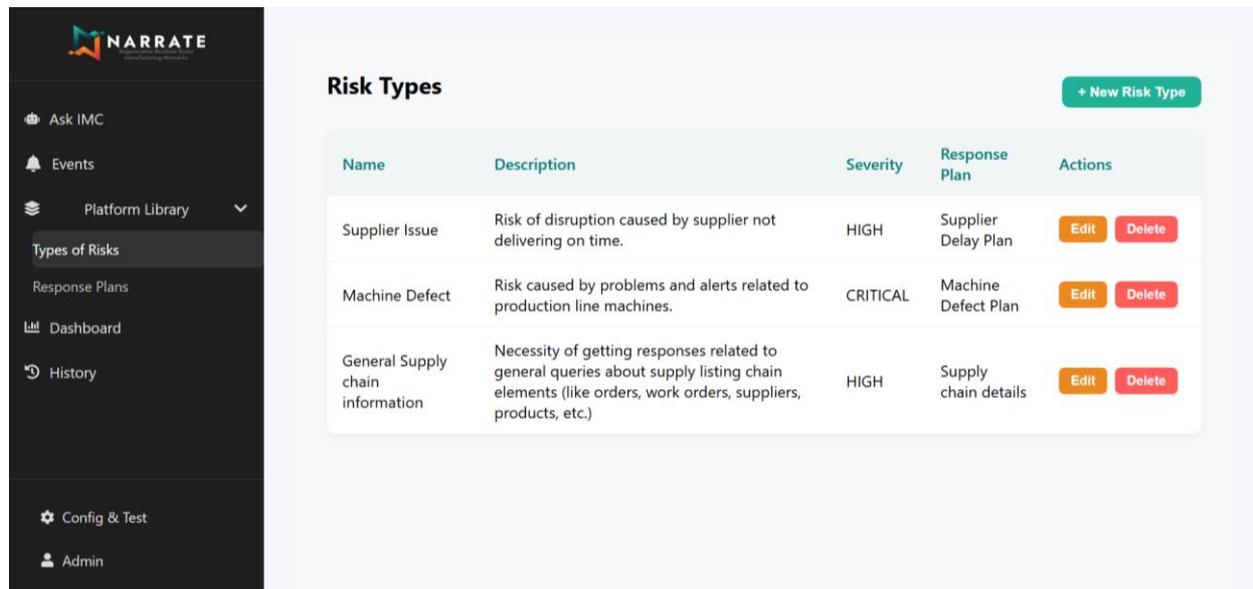


FIGURE 10: RISK TYPES LIST

I can view the previous edits of the risk type and make new ones. (FIGURE 11)

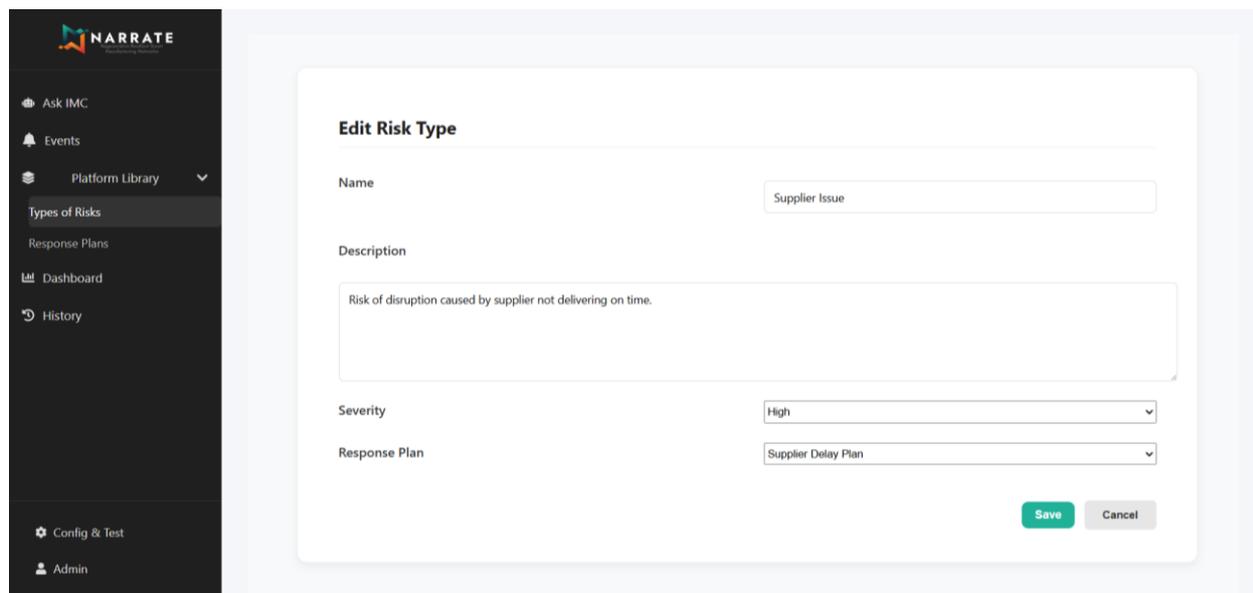


FIGURE 11: EDIT RISK TYPE

## 7. History (List and Detail)

As a Pilot User, I want to see a list of past chats and open details for review.

I can access the History from the side bar menu.

The list has ID, date, event associated, risk associated, response plan associated as parameters. I can filter the chats by any possible parameter.

The detailed chat has the same characteristics of the chat interface.

I can export in PDF/JSON the chat. (FIGURE 12)

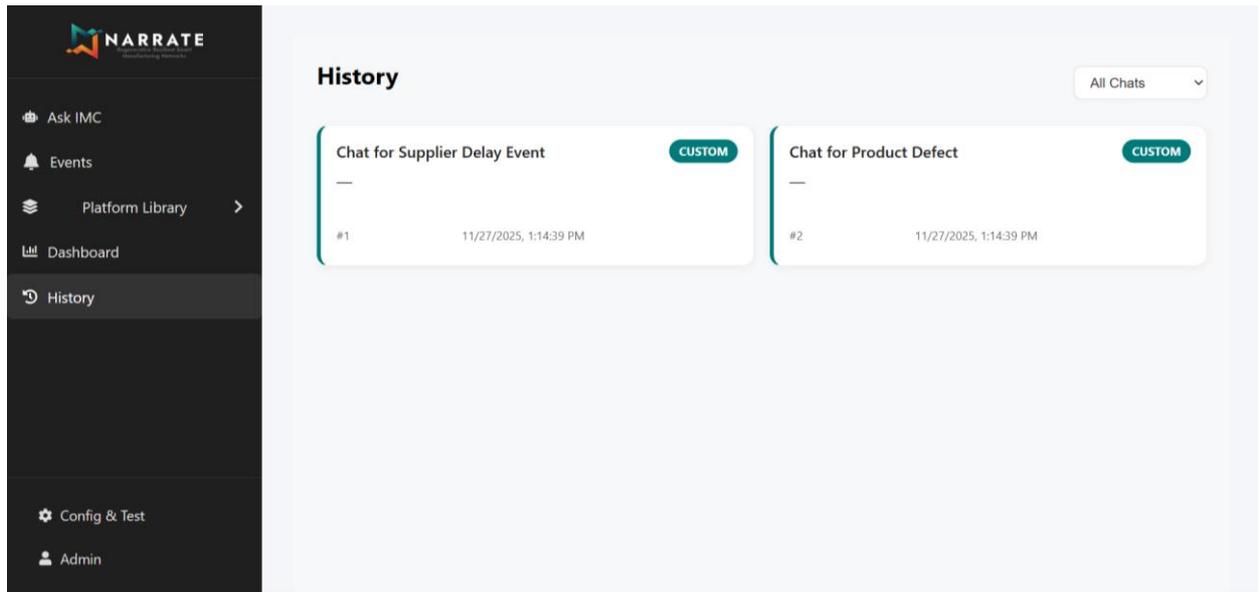


FIGURE 12: NMP HISTORY

## 8. Dashboard (Embedded Metabase)

As Pilot User, I want to access an embedded dashboard to see KPIs and metrics.

I can access the Dashboard from the side bar menu.

I can see the Metabase frame, statistics, KPIs, information in real time.

I can filter every information by the parameters accepted. (FIGURE 13)

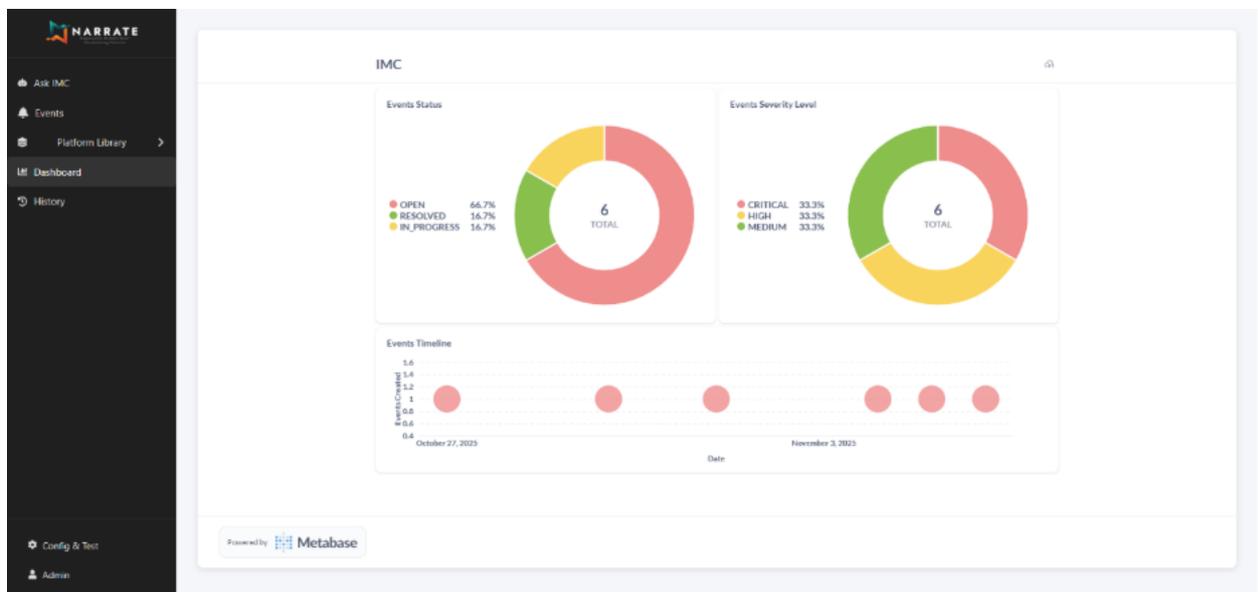


FIGURE 13: NMP DASHBOARD

## 9. Config-Plugins

As a Pilot Admin User, I want to register and configure plugins so the IMC can use them.

I can access the Plugin menu from the side bar menu, in the Platform Library section. The Plugins have Name, Description, URL, Status as parameters.

They also have the chance to be added, edited or deleted. (FIGURE 14)

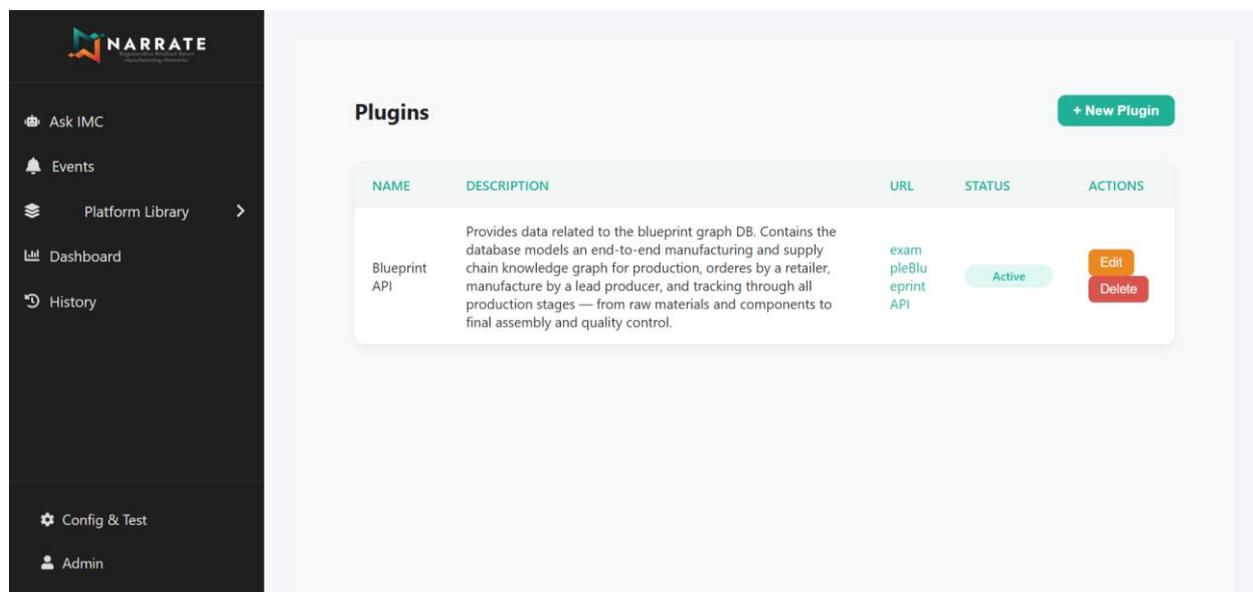


FIGURE 14: PLUGINS LIST

I can give edit plugin parameters to give a better description of the tool in a hypothetical future development scenario. (FIGURE 15)

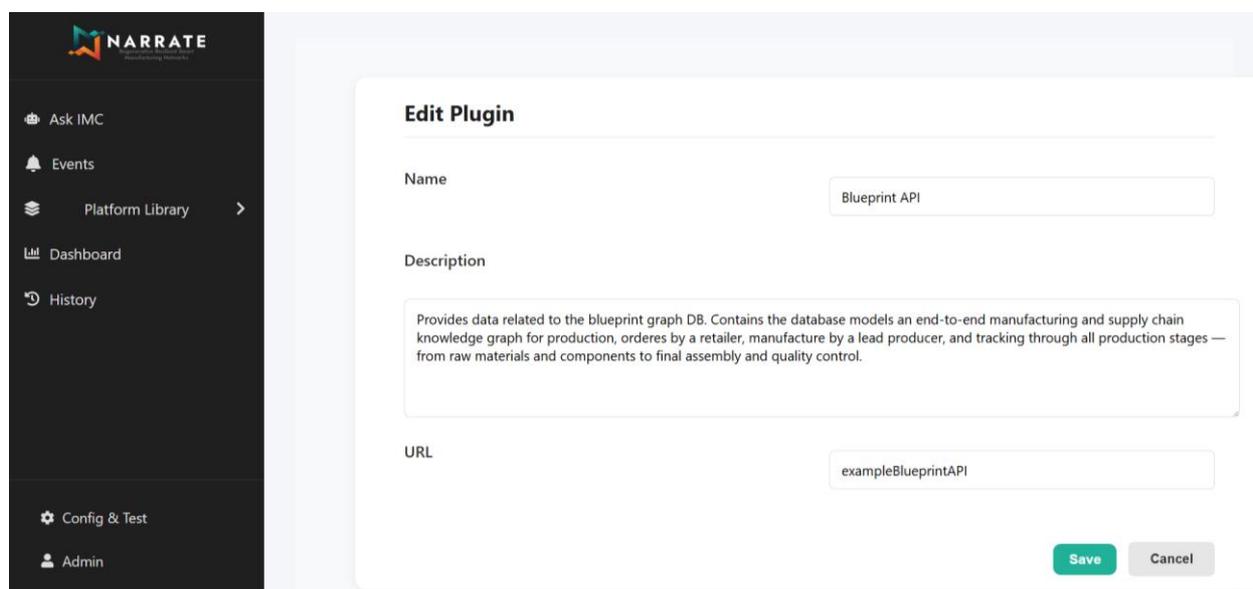


FIGURE 15: CONFIG-PLUGINS

## 10. Test (Simulations)

As a Pilot Admin User, I want to send simulated events so I can validate IMC behaviour.

I can fill the text boxes in the FIGURE 16 or upload a compatible JSON file to generate a test alert. I can run a simulation of a determined event, see them in the history, and even duplicate past events to test the Platform Orchestration and the SMN behaviour. (FIGURE 16)

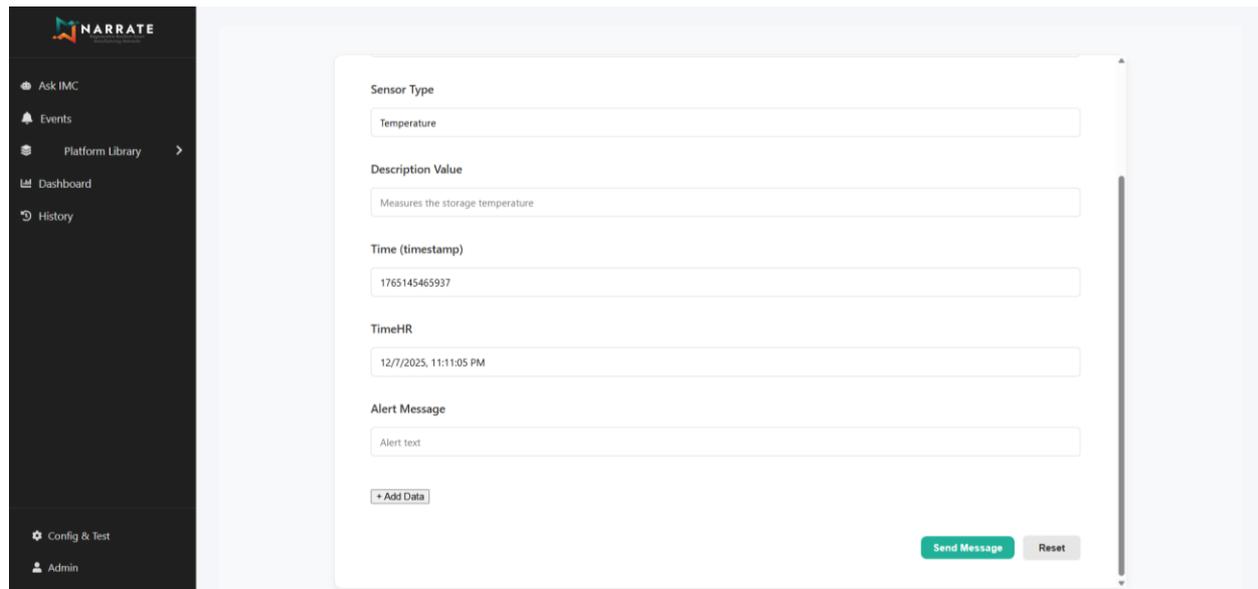


FIGURE 16: EVENT TEST SIMULATION

### 2.3. NSQA

The Neuro-Symbolic Question Answering component extends the IMC’s reasoning capabilities by integrating symbolic expert knowledge with the adaptive natural language understanding provided by LLMs. While the IMC acts as the high-level planner and orchestrator, the NSQA module is responsible for structured decision support grounded in explicit knowledge representations and stepwise reasoning. This ensures that the system’s behavior remains consistent, explainable, and auditable, even when dealing with ambiguous or evolving operational contexts [2].

The NSQA combines three complementary knowledge layers:

- **Expert Rules**

These are natural language policies, heuristics, and operational constraints curated by domain specialists. Rules codify domain expectations and preferences, such as supplier priorities or procedural requirements. The NSQA injects these rules into every reasoning process, enabling consistent decision-making aligned with organizational policy.

- **Response Plans**

Each risk type defined in the Platform Library is associated with a response plan. These plans specify the recommended course of action, describe the steps that should be executed, and identify the plugins or external data sources that must be consulted. NSQA interprets these plans, decomposes their logic into actionable steps, and adapts them when contextual information requires dynamic adjustment.

- **Facts from External Knowledge Sources**

The NSQA enriches expert knowledge with real-time or on-demand information retrieved from external APIs and RAG-based information services. These facts provide grounding for the reasoning process and allow response plans to be tailored to the current state of the supply chain, asset conditions, or external events.

At runtime, the NSQA instantiates a **neuro-symbolic reasoning agent**, which integrates these knowledge layers into a unified decision-making workflow. The agent combines reflective reasoning with external tool invocation to iteratively produce well-founded conclusions. It documents every thought, decision, plugin call, and retrieved fact, creating a transparent audit trail stored in the History entity of the NARRATE Monitoring Platform.

Key capabilities of the NSQA component include:

- **Interpretation of natural language inputs** from Events or user prompts into actionable reasoning tasks.
- **Structured decision support**, where each step is governed by a combination of symbolic rules and contextual data.
- **Transparent reasoning**, with detailed intermediate steps, justifications, and rule activations.
- **Consistency and explainability**, ensured through the interplay between symbolic constraints and LLM-based inference.
- **Dynamic adaptation** of response plans whenever contextual information modifies the expected workflow.

In the overall, the NSQA operates as a specialized reasoning layer that reinforces the IMC's decision-making with semantic rigor and explainable intelligence. It bridges natural language, expert knowledge, and machine-actionable workflows, ensuring that risk management actions are both operationally valid and fully traceable within the system.

This is explained in detail in the deliverable D3.3 [2].

---

## 2.4. LLM – LARGE LANGUAGE MODELS

---

Large Language Models constitute the primary cognitive engine of the Intelligent Manufacturing Custodian, enabling advanced reasoning, natural language understanding, and context-aware decision support across the NARRATE AI Platform. Their integration is foundational to transforming human instructions, unstructured inputs, and heterogeneous event descriptions into structured, actionable plans aligned with manufacturing objectives and operational constraints.

Operating as both interpreters and planners, the LLMs embedded within the IMC support several core capabilities:

- **Natural Language Interpretation.** LLMs translate user prompts, event descriptions, response plans, and expert rules into structured semantic representations that the IMC can operationalize. This includes extracting entities, identifying intent, and inferring the relationships required to classify risks and contextualize actions.
- **Adaptive Reasoning.** Through chain-of-thought and tool-augmented reasoning, the LLMs determine how to decompose complex instructions into actionable task units. They evaluate conditions, apply prioritization logic, select relevant plugins, and adapt predefined plans based on contextual parameters such as supplier availability, lead times, or inventory status.
- **Dynamic Orchestration.** The IMC relies on LLMs to map high-level natural language plans into concrete sequences of plugin invocations. This translation is performed

dynamically, allowing the system to adjust execution based on intermediate results retrieved from external APIs, Knowledge Graph sources, or Digital Twins.

- **Knowledge Integration.** LLMs operate in synergy with the NSQA module, consuming expert rules, risk types, and response plans as part of their reasoning context. They incorporate both symbolic knowledge and real-time external facts, enabling decisions that remain grounded, explainable, and aligned with operational objectives.
- **Explainability Support.** The LLM generates coherent, natural language explanations that accompany every executed response plan. These explanations detail the reasoning steps, plugin decisions, and contextual interpretations, and are automatically recorded in the History component to provide full auditability.
- **Continuous Adaptation.** Although the platform does not automatically retrain models, the IMC leverages LLM context management and prompt engineering to adapt its behavior over time. This includes incorporating user feedback, updated rules, new response plans, and evolving operational constraints without requiring model fine-tuning.

## 2.5. RISKS IDENTIFICATION AND PROCESSING

This BB#2 module is responsible for identifying risks within the SMN across the entire stack, ranging from machine-level operational risks to geopolitical exposure and supply chain vulnerabilities.

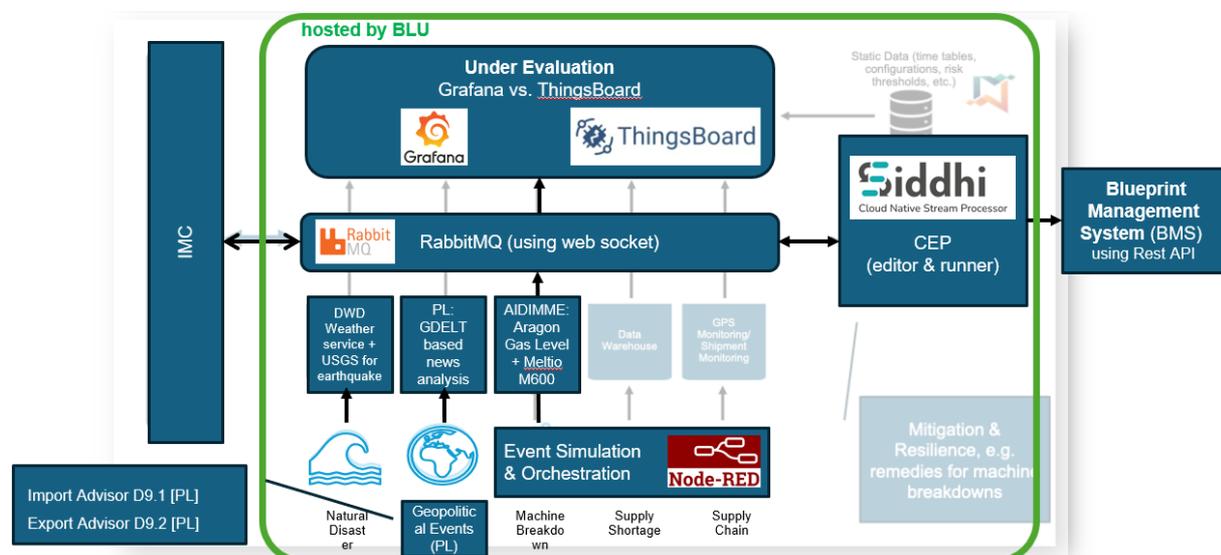


FIGURE 17: RISK IDENTIFICATION WORKFLOW [3]

The Complex Event Processor (CEP) publishes detected risk events to RabbitMQ, which acts as a message bus to ensure reliable, asynchronous, and decoupled delivery of alerts. The IMC consumes these events and is responsible for propagating them to the NMP in real time [3].

The communication channel between the CEP side and the IMC is a bidirectional WebSocket connection managed by the NMP (Figure 17, FIGURE 18) [3].

A persistent WebSocket session is established between the CEP and the IMC to the endpoint “ws://events.bluebridgesolutions.de”, enabling the CEP to deliver alerts to the IMC. Upon receiving an alert, the IMC generates a user-facing notification and forwards it to the NMP, which records it as a historical event accessible to the user. When applicable, the NMP may send acknowledgements or control-related feedback back to the IMC.

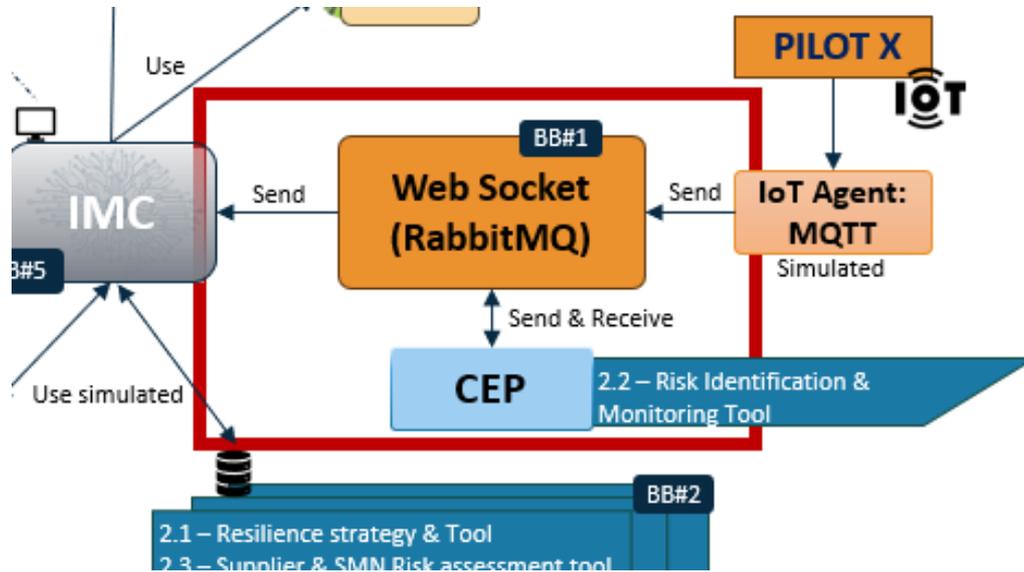


FIGURE 18: RISK IDENTIFICATION WORKFLOW

This WebSocket-based integration guarantees low-latency, two-way communication and allows the IMC to be continuously informed about newly identified risks in real time [3].

Figure 19 shows an example of the JSON structure of a CEP alert in a basic scenario, where a machine exceeds a threshold value.

```

{
  "eventId": "11111",
  "eventType": "sensorAlert",
  "sensorType": "Temperature",
  "descriptionValue": "Measures the storage temperature",
  "time": 1242352352352,
  "timeHR": "Mo 10:30:10, 05-10-2025",
  "alert": "Alert: Machine {machine} exceeds the threshold value. Actual: {currentValue} Threshold: {valueThreshold}",
  "data": {
    "valueThreshold": 0,
    "currentValue": 1,
    "units": "°C",
    "machineId": "machine1"
  }
}
    
```

FIGURE 19: CEP ALERT EXAMPLE

## 2.6. PLATFORM LIBRARY

The Platform Library is a repository used to store Risk Types and Response Plans (TABLE 1, TABLE 2), ensuring a seamless workflow among entities and building blocks, as these matrices constitute the starting point for evaluating and actuating an executable plan.

TABLE 1: EXAMPLE OF RESPONSE PLAN

ID	Name	Description	Instructions
1	Supplier Delay Plan	Plan to handle delayed supplier deliveries	<p>1. Check all the orders, find the one affected and save the field product id (productID) related to that order and save the field Work order id (workOrderID).</p> <p>2. Search for that product id to get the details. From the components and materials, get the component id or material id affected (materialID or componentID).</p> <p>3. Search for all the suppliers that can provide that material or component using the material id or component id and the Work order id to get the actual supplier and the potential supplier. ((materialID or componentID) and workOrderID)\nOnce you get all the data, propose the best potential supplier to prevent disruptions.</p>
2	Machine Defect Plan	Plan to handle defective Machines	<p>Use the context retrieved from the rag tool to answer the questions related to machine maintenances.</p> <p>Give clear and complete response, and do not skip any step of the process.</p> <p>It is critical that you stick to the content of the context received.</p>
3	Supply chain details	Plan to respond general queries about supply listing chain elements (like orders, work orders, suppliers, products, etc.)	Use de endpoint that you think fits most in order to answer the intentions of the user query.

TABLE 2: EXAMPLES OF RISK TYPES

ID	Name	Description	Severity
1	Supplier Issue	Risk of disruption caused by supplier not delivering on time.	HIGH
2	Machine Defect	Risk caused by problems and alerts related to production line machines.	CRITICAL
3	General Supply Chain information	Necessity of getting responses related to general queries about supply listing chain elements (like orders, work orders, suppliers, products, etc.)	HIGH

At the current stage of the project, the Platform Library simulates the Resilience Strategies & Tool, providing Risk Types and Response Plans. It is connected to the database and knowledge repositories (FIGURE 20) of the Platform ecosystem.

This BB#2 (Resilience strategies & Tool) module is responsible for providing Risk Types and Response Plans to Platform Knowledge. The module relies on a Suprabase repository for the persistence of the corresponding tables (Table 1, Table 2). Next steps on this matter will be mentioned in chapter 6.

Risk classifications and response strategies are defined by the IMC, with the support of the NSQA, both for testing solutions against disruptive scenarios and for reacting to actual events. These steps enable the IMC to coherently orchestrate the appropriate tools required to manage the Event.

---

## 2.7. OTHER COMPONENTS

---

- **Event Ingestion and Service Bus**
  - RabbitMQ/Kafka: Serves as the event ingestion layer, ensuring reliable and scalable handling of external events. Each event is enriched with metadata before being processed by the IMC.
  - Service Bus: Connects the Monitoring Platform, IMC, and Plugins, ensuring asynchronous communication and decoupled workflows.
- **Database and Knowledge Repositories** (FIGURE 20)
  - Postgres Database: Central storage for events, risks, response plans, chats, and historical data.
  - Platform Library: Repository containing the catalogue of risks and response plans, editable and versioned.
  - Knowledge Graph / Ontologies: Support NSQA reasoning and enhance explainability.

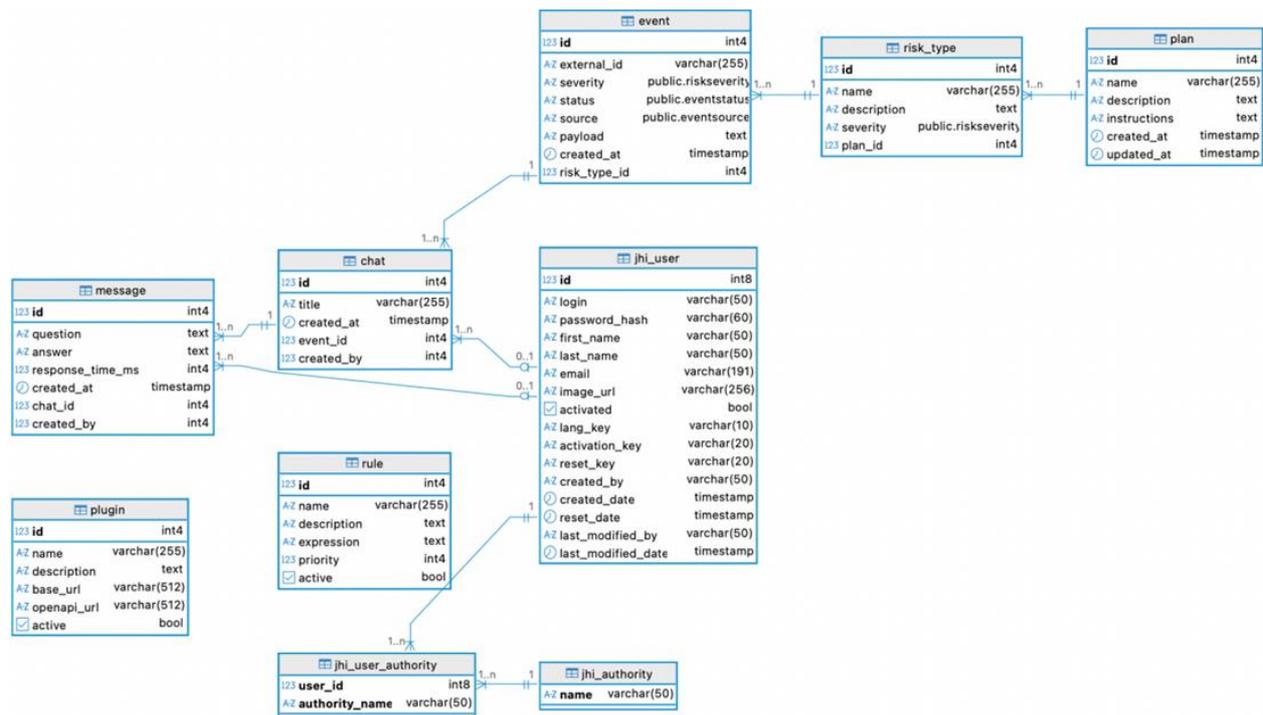


FIGURE 20: DATABASE AND KNOWLEDGE REPOSITORY UML DIAGRAM

o **Test and Simulation Environment**

The Test module allows **safe experimentation**:

- Simulate events to validate response plans and plugin interactions.
- Replay past events to analyse system behaviour.
- Train operators by exposing them to realistic scenarios without impacting production data.

### 3. TYPICAL WORKFLOW

The NARRATE Monitoring Platform provides a unified, intelligent, and explainable environment in which the complexity of AI-driven orchestration is abstracted into user-facing entities. This architectural approach enables organizations to respond to risks with speed, transparency, and confidence, in alignment with the project’s overarching objective of industrial resilience and smart digitalization.

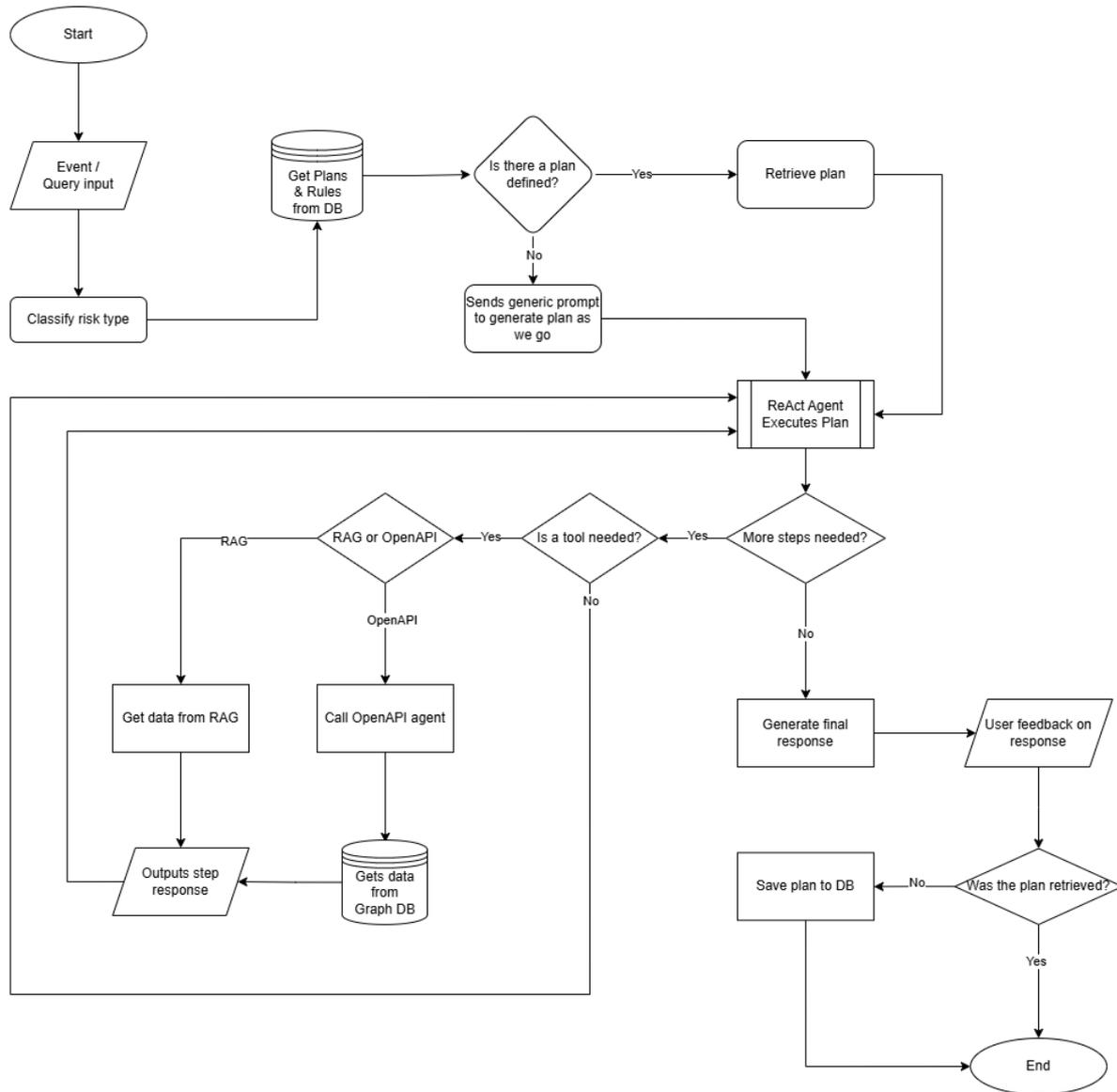


FIGURE 21: DATA FLOW

The typical workflow can be initiated in two different ways:

- With the reception of an Event (for example, a supplier disruption, an equipment malfunction, or an emerging geopolitical tension) through the CEP message bus.
- The user may initiate the process directly by submitting a natural language query through the NMP chat interface. Regardless of the entry point, the system treats the input as a risk that requires contextual analysis and potential mitigation.

The workflow continues following this actions sequence to handle the event received/generated.

- The IMC evaluates the content of the Event and classifies it into a Risk type by consulting the Platform Library. This categorization step relies on semantic interpretation capabilities that allow the platform to map heterogeneous and ambiguous descriptions into structured operational domains. The classification determines which portions of the encoded expert knowledge, and which external information sources will be relevant for subsequent reasoning.

- Once the Risk type has been identified, the corresponding Response Plan is retrieved, if available. These Response Plans encapsulate domain expertise in the form of procedural knowledge, expected decision structures, and possible external information dependencies. When no predefined plan exists, the IMC Planner may generate a coherent alternative based on expert rules, contextual information, and the historical behavior of the system in comparable situations. This dynamically generated Response Plan preserves traceability and can later be validated by users.
- The IMC then leverages the LLM to interpret natural language instructions and translate them into a sequence of structured plugin calls. This translation process includes selecting relevant external data sources, determining which analytical operations are required, and orchestrating a chain of reasoning steps. Throughout execution, the system produces intermediate justifications, records tool usage, and formulates human-readable explanations that reflect both the underlying logic and the data gathered.
- Execution results are stored in the database and visualized via the Dashboard, implemented in Metabase. The dashboard aggregates KPIs, trends, and system-level indicators that help stakeholders understand not only the final outcomes but also the quality and reliability of intermediate inferences. This provides a consolidated view of system performance, operational resilience, and the effectiveness of executed Response Plans.

This workflow is deeply explained in the deliverable 3.3 in section 6 (NSQA Processing Pipeline) [2]

From a technical perspective, the system adheres to the following key principles:

- Event-driven ingestion via RabbitMQ/kafka, enabling real-time processing and system scalability.
- Task orchestration powered by LLMs, supporting adaptability, natural language reasoning, and explainability.
- Microservices-based plugins, each exposing specialized capabilities and integrated through standardized OpenAPI contracts.
- Automatic refresh of the Events and History views to ensure continuous situational awareness.
- Traceability by design, with every decision, action, and plugin invocation persistently stored in History.
- A user-centric interface that supports both structured monitoring of Events and conversational interaction with the IMC.

Additionally, the Test module enables the simulation of Events, supporting validation, training, and experimentation without reliance on real incidents.

Examples of this workflow are shown in the deliverable 3.3 in section 8 (Examples and case studies) [2]

---

### 3.1. USE CASE 1: ALTERNATIVE PROVIDER

---

Examples of this workflow are shown in the deliverable 3.3 in section 8 (Examples and case studies) [2]

**Input:** The provider of the material beechwood for order SO\_2025\_001 won't be able to deliver on time. How do I solve it?

Steps:

### 1. Get input via event or query

The system receives incoming information either as a natural-language query from a user or as a structured event generated by another component through RabbitMQ.

Regardless of format or origin, the input is normalized and transformed into a consistent internal representation.

This unified preprocessing ensures that both human-initiated requests and machine-triggered alerts enter the same reasoning pipeline for risk analysis.

```
async def query(query_input: QueryInput, db: Session) -> QueryResponse:
    """
    """
    event = get_event_from_query(query_input)

    chat_manager = ChatManager()
    plan_manager = PlanManager()
    rules_manager = RulesManager()
    plugin_manager = PluginManager()
    risk_type_manager = RiskTypeManager()
    event_manager = EventManager()

    orchestrator_agent = OrchestratorAgent(
        chat_manager=chat_manager,
        plan_manager=plan_manager,
        rules_manager=rules_manager,
        plugin_manager=plugin_manager,
        risk_type_manager=risk_type_manager,
        event_manager=event_manager
    )

    response = await orchestrator_agent.get_response(event, db)

    return response
```

```
async def process_message(msg, db: Session):
    """
    """
    event = get_event_from_message(msg)

    chat_manager = ChatManager()
    plan_manager = PlanManager()
    rules_manager = RulesManager()
    plugin_manager = PluginManager()
    risk_type_manager = RiskTypeManager()
    event_manager = EventManager()
```

```
orchestrator_agent = OrchestratorAgent(
    chat_manager=chat_manager,
    plan_manager=plan_manager,
    rules_manager=rules_manager,
    plugin_manager=plugin_manager,
    risk_type_manager=risk_type_manager,
    event_manager=event_manager
)
return await orchestrator_agent.get_response(event, db)
```

## 2. Select risk type

The system loads all existing risk types from the knowledge base and uses a large language model to compare the incoming input against these definitions.

Through semantic similarity assessment, it identifies which risk type best matches the meaning and context of the input.

If no suitable match exists, a new risk type is automatically generated and stored, enabling the system to adapt to previously unseen risks.

```
async def get_response(self, event_query: EventModel, db: Session) ->
QueryResponse:
    """ """

    plan, decision, chat_id = await self._select_plan_advanced(
        event_query=event_query, db=db
    )

[...]
```

```
async def _select_risk_type(
    self, possible_risk_types: List[RiskTypeData], event: EventModel
) -> tuple[str, str]:
    """
    Use an LLM to select the most appropriate risk_type given the user's
    query.
    """

    user_query = event.query

    risk_blocks = "\n\n".join(
        f"PLAN ID {rt.id}:\n{rt.description}" for rt in possible_risk_types
    )

    template = f"""
    You are an expert orchestrator that decides which risk type description
    best fits a user's question.

    Here are the available risk type descriptions:
```

```

{risk_blocks}

User's question:
"{user_query}"

Decide which risk type description is most related to this question.
Respond ONLY with the ID (example: "1").
"""

prompt = PromptTemplate(template=template)

try:
    result = await (prompt | self.llm).ainvoke({})
    decision = result.content.strip() if result else ""
except Exception as e:
    print(f"Plan selection failed via LLM: {e}")
    decision = ""

return decision

```

### 3. Get plans from DB based on risk

After the risk type is determined, the system retrieves all response plans associated with that category from the database.

It evaluates these plans -considering metadata, usage history, or semantic annotations- to select the one that most closely fits the situation.

If none is appropriate, the system records this and prepares for dynamic plan creation during reasoning.

```

class PlanManager(PlanManagerBase):
    async def get_plans(self, db: Session) -> list[PlanData]:
        plan_list_object: list[Plan] = db.scalars(select(Plan)).all()
        return [
            PlanData(
                id=plan.id,
                name=plan.name,
                description=plan.description,
                instructions=plan.instructions
            ) for plan in plan_list_object
        ]

    async def get_plan(self, id: int, db: Session) -> PlanData:
        plan: Plan | None = db.scalars(select(Plan).where(Plan.id == id)).first()

        if plan is None:
            return None

        return PlanData(

```

```

        id=plan.id,
        name=plan.name,
        description=plan.description,
        instructions=plan.instructions
    )

class RulesManager(RulesManagerBase):
    async def get_rules(self, db: Session) -> list[RuleData]:
        rule_object: list[Rule] = db.scalars(select(Rule)).all()
        return [
            RuleData(
                id=rule.id,
                name=rule.name,
                description=rule.description,
                expression=rule.expression,
                priority=rule.priority
            ) for rule in rule_object
        ]

```

#### 4. Plan and execute using ReAct agent

A ReAct agent is constructed using the chosen plan (or an empty scaffold), all expert rules, external agents, and known facts.

It proceeds through iterative cycles of reasoning, tool-calling, and reflection, gathering information and adjusting steps as needed.

The process continues until the agent reaches a well-justified response, producing both an actionable recommendation and a transparent reasoning trace.

```

async def get_response(self, event_query: EventModel, db: Session) ->
QueryResponse:

[... ]

    elif str(decision) == "1":
        react_agent = ReActAgent(
            question=event_query.query,
            plan_prompt=plan,
            criteria_prompt=rules,
            history=history,
            tools=tools,
        )
        react_agent.build_agent()

def build_agent(self):
    if not self.plan_prompt:
        self.plan_prompt = default_plan_template
    self.final_prompt = self._build_react_prompt(
        self.plan_prompt, self.criteria_prompt, self.history

```

```
)
self.planner = load_chat_planner(self.llm, self.final_prompt)
self.executor = load_agent_executor(tools=self.tools, llm=self.llm)
self.react_chain = PlanAndExecute(planner=self.planner,
executor=self.executor)
self.logger.info("Successfully build ReAct agent")

[...]
base_prompt = """
You are an intelligent reasoning agent designed to answer user questions
accurately and efficiently.
You have access to various tools to help you complete and execute your reasoning.

Your objective:
- Follow the provided plan to reason through the question and use tools to gather
data to answer.
- Evaluate your reasoning and final answer against the provided criteria.
---

PLAN:
__PLAN__
---

CRITERIA:
__CRITERIA__
---

CHAT HISTORY:
__HISTORY__

USER'S QUERSION:
{question}
---
"""

[...]

def plan_and_execute(self):
self.logger.info(f"{self.__class__.__name__}: Getting agent response.")
max_retries = int(settings.max_retries_parser_error_func_call)
attempts = 0

callback_handler = ReActCallbackHandler(self.plan_actions)

while attempts < max_retries:
try:
result = self.react_chain.invoke(
```

```

        {"input": f"Use the available tools to generate a response:
{self.question}"},
        config={"callbacks": [callback_handler]},
    )

    self.logger.info(result)

    reasoning_steps = self._get_reasoning_steps()

    return ResponseModel(
        question=self.question,
        answer=result["output"],
        context=ContextModel(
            reasoning_steps=reasoning_steps,
            success=True
        )
    )

except OutputParserException as e:
    attempts += 1
    if attempts == max_retries:
        try:
            reasoning_steps = self._get_reasoning_steps()
        except:
            reasoning_steps = None
        return ResponseModel(
            question=self.question,
            answer=None,
            context=ContextModel(
                reasoning_steps=reasoning_steps,
                success=False,
                error_message=str(e)
            )
        )

    self.logger.warning(
        f"Retrying {attempts}/{max_retries} times openapi_chain
response because of Exception {e}... "
    )

except Exception as e:
    if "list index out of range" in str(e):
        try:
            reasoning_steps = self._get_reasoning_steps()
        except Exception:
            reasoning_steps = []

    return ResponseModel(

```

```

        question=self.question,
        answer=(
            "The system could not generate a tool-based plan, "
            "so here is a direct answer:\n\n" +
self._fallback_answer()
        ),
        context=ContextModel(
            reasoning_steps=reasoning_steps,
            success=False,
            error_message=f"Planner error: {str(e)}"
        )
    )

self.logger.error(
    f"{self.__class__.__name__}: An exception occurred: {e}"
)
import traceback
self.logger.error(f"Exception type: {type(e)}")
self.logger.error(traceback.format_exc())
# Returning something for testing
try:
    reasoning_steps = self._get_reasoning_steps()
except:
    reasoning_steps = None
return ResponseModel(
    question=self.question,
    answer=None,
    context=ContextModel(
        reasoning_steps=reasoning_steps,
        success=False,
        error_message=str(e)
    )
)

```

#### 4.1 Calls to OpenAPI agent

During execution, the ReAct agent may identify the need for external information or system interaction and invokes the appropriate OpenAPI agent.

These agents act as structured connectors to external services, enforcing schema validation and standardized request formats.

By delegating calls through these agents, the system ensures reliability, safety, and consistent integration with operational APIs.

```

def get_tool(plugin: PluginInfo):
    agent = OpenApiAgent(plugin=plugin)
    agent.build_agent()

    tool = Tool(
        name=plugin.title,

```

```

        description=(
            plugin.description
        ),
        func=lambda question: agent.ask_agent(question)
    )
    return tool

[...]

class OpenApiAgent:
    def __init__(self, plugin: PluginInfo):
        self.logger: Logger = logger
        self.id = plugin.id
        self.title = plugin.title
        self.url = plugin.url
        self.schema_url = plugin.schema_url
        self.verify_ssl = plugin.verify_ssl
        self.prompt = plugin.prompt
        self.auth_type = plugin.auth_type
        self.auth_params = plugin.auth_params
        self.config_params = plugin.config_params
        self.llm = llm

    def build_agent(self):
        self.logger.debug(f"{self.__class__.__name__}: Building agent.")

        secret_key = settings.encrypt_secret_key

        # Load and parse OpenAPI schema
        schema = self._load_openapi_schema()
        self.spec = OpenAPISpec.from_spec_dict(schema)

        # Set headers based on auth type
        self.headers = self._build_headers(secret_key)

        # Load optional config parameters
        self._load_config_params(secret_key)

        # Construct final prompt
        self._build_prompt_template()

        self.openapi_chain = self._build_chain()

    def ask_agent(self, question):
        self.logger.critical(f"{self.__class__.__name__}: Getting agent
response.")
        max_retries = int(settings.max_retries_parser_error_func_call)
        attempts = 0

```

```

while attempts < max_retries:
    try:
        response = [
            self.openapi_chain.invoke(
                {"question": question}
            )
        ]
        self.logger.info(f"Raw chain output: {response}")
        return response

    except OutputParserException as e:
        print(e)
        attempts += 1
        if attempts == max_retries:
            return []
        self.logger.warning(
            f"Retrying {attempts}/{max_retries} times openapi_chain
response because of Exception {e}... "
        )

    except Exception as e:
        self.logger.error(
            f"{self.__class__.__name__}: An exception occurred while
querying the plugin: {e}"
        )
        return []

def _load_openapi_schema(self):
    try:
        schema = get_openapi_schema(self.schema_url, self.url)
    except Exception as e:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=f"Error while retrieving the schema: {e}",
        )
    return schema

def _build_headers(self, secret_key):
    if self.auth_type == "bearer_token":
        decrypted = EncryptDecrypt.decrypt_secret_key(
            self.auth_params, secret_key=secret_key
        )
        parsed = json.loads(decrypted.replace("'", ''))
        self.auth_token = parsed.get("bearer_token")
    return {
        "Authorization": f"Bearer {self.auth_token}",
        "Content-Type": "application/json",
        "Accept": "application/json",
    }

```

```

    }

    self.auth_token = None
    return {
        "Content-Type": "application/json",
        "Accept": "application/json",
    }

def _load_config_params(self, secret_key):
    self.parameters = {}

    if self.config_params:
        decrypted = EncryptDecrypt.decrypt_secret_key(
            self.config_params, secret_key=secret_key
        )
        if decrypted:
            self.parameters = json.loads(decrypted.replace("'", ''))

def _build_prompt_template(self):
    self.prompt_template = PromptTemplate(
        input_variables=["question"], template=self.prompt
    )

def _build_chain(self):
    return get_custom_openapi_chain(
        self.spec,
        self.llm,
        verbose=True,
        headers=self.headers,
        params=self.parameters,
        verify_ssl=self.verify_ssl,
        prompt=self.prompt_template,
    )

```

## 4.2 OpenAPI calls to NSQA API

When an OpenAPI agent is triggered, it formulates and sends requests directly to the NSQA API or related backend endpoints.

This enables the retrieval of domain-specific data, validation of system states, or execution of predefined operations required by the reasoning cycle.

The results returned by the NSQA API are then fed back into the ReAct agent's working memory as new facts or observations.

```

@router.get("/orders", response_model=OrdersResponse)
def get_orders():
    """
    Returns detailed sales order information including
    customer, product, and manufacturer data.
    """

```

```

records = run_query(ORDERS_QUERY)
if not records:
    raise HTTPException(status_code=404, detail="No orders were found.")
return {"count": len(records), "orders": records}

@router.get("/product/{product_id}", response_model=ProductDetailsResponse)
def get_product_details(product_id: str):
    """
    Returns detailed information of a product by ID.
    """
    records = run_query(PRODUCT_DETAILS_QUERY, {"product_ID": product_id})
    if not records:
        raise HTTPException(status_code=404, detail=f"Product '{product_id}' not
found.")
    return {"count": len(records), "productDetails": records}

@router.get("/suppliers/", response_model=SuppliersByMaterialResponse)
def get_suppliers_by_material(workorder_id: str, input_id: str):
    """
    Returns suppliers related to a specific Material or Product Component and
Workorder ID.
    The input ID can be either a `materialID` or a `componentID`.
    Workorder ID is the parameter `workOrderID`.
    """
    records = run_query(SUPPLIERS_BY_MATERIAL, {
        "workOrderID": workorder_id,
        "inputID": input_id
    })
    if not records:
        raise HTTPException(
            status_code=404,
            detail=f"Material or component id '{input_id}' not found."
        )
    return {"input_id": input_id, "count": len(records), "suppliers": records}

@router.get("/workorders/", response_model=WorkOrdersResponse)
def get_all_workorders():
    records = run_query(ALL_WORKORDERS)
    if not records:
        raise HTTPException(
            status_code=404,
            detail=f"No workorders were found."
        )
    return {"count": len(records), "workorders": records}

```

```
@router.get("/all-suppliers/", response_model=SupplierListResponse)
def get_all_suppliers():
    records = run_query(ALL_SUPPLIERS)
    if not records:
        raise HTTPException(
            status_code=404,
            detail=f"No suppliers were found."
        )
    return {"suppliers": records}

@router.get("/all-products/", response_model=ProductsResponse)
def get_all_products():
    records = run_query(ALL_PRODUCTS)
    if not records:
        raise HTTPException(
            status_code=404,
            detail=f"No products were found."
        )
    return {"count": len(records), "products": records}
```

### 4.3 ReAct agent reasoning steps

After receiving responses from external calls, the ReAct agent incorporates the new information into its internal context.

It applies expert rules, interprets the implications of new facts, and decides the next action in the reasoning chain.

This loop of reasoning, acting, and reflecting continues until the agent converges on a stable and well-supported resolution.

#### Reasoning steps:

##### 11. Check all orders

**Task:** Find the order with ID SO\_2025\_001 and retrieve the associated product ID and work order ID.

**Response:**

- **Product ID:** PROD\_2025\_001
- **Work Order ID:** WO\_2025\_001

##### 12. Get product details

**Task:** Retrieve product details using the product ID to identify the beechwood material/component ID.

**Response:**

- **Material Name:** Beechwood (FSC-certified)
- **Material ID:** MAT\_2025\_BW\_001
- **Material Type:** Wood
- **Material Origin:** Germany, Black Forest

##### 13. Find suppliers

**Task:** Using the beechwood material ID and work order ID, find all potential suppliers.

**Response:**

- **Actual Supplier:**

- Supplier Name: *EcoWood Partners GmbH*
- Supplier ID: *MS\_002*
- Location: *Mannheim, Germany*
- Lead Time: *2 days*
- Certifications: *FSC, PEFC, ISO 9001*
- Supplying Date: *2025-09-17*
- **Potential Suppliers:**
  - *ForestLine Timber AG:*
    - ✓ Supplier ID: *MS\_003*
    - ✓ Location: *Zurich, Switzerland*
    - ✓ Lead Time: *5 days*
    - ✓ Certifications: *FSC, ISO 45001*
  - *GreenWood Supplies Ltd.*
    - ✓ Supplier ID: *MS\_001*
    - ✓ Location: *Freiburg, Germany*
    - ✓ Lead Time: *3 days*
    - ✓ Certifications: *FSC, ISO 14001*

#### 14. Propose a solution

**Task:** Recommend the best supplier to avoid disruptions.

**Response:**

- **Best Supplier:** *EcoWood Partners GmbH* — shortest lead time (2 days), certified (FSC, PEFC, ISO 9001), geographically close.
- **Backup Supplier:** *GreenWood Supplies Ltd.* — 3-day lead time and close proximity.

#### 5. Final response generation

Once the reasoning process concludes, the system compiles a structured output that includes the recommended action or solution.

Alongside the result, it generates an explanatory narrative clarifying why each step was taken and how the conclusion was reached.

This final response ensures transparency, traceability, and readiness for user review or automated downstream execution.

```
reasoning_steps = self._get_reasoning_steps()

return ResponseModel(
    question=self.question,
    answer=result["output"],
    context=ContextModel(
        reasoning_steps=reasoning_steps,
        success=True
    )
)
```

[...]

```
response: ResponseModel = react_agent.plan_and_execute()
```

[...]

```
return QueryResponse(chat_id=chat_id, response=response)
```

### Final output:

Based on the supplier information provided, the best potential supplier to prevent disruptions for the order SO\_2025\_001 is EcoWood Partners GmbH. This supplier is already the actual supplier, located in Mannheim, Germany, with a lead time capability of 2 days on average, which is the shortest among the options. They also hold relevant certifications such as FSC, PEFC, and ISO 9001, ensuring quality and sustainability standards are met. If a backup supplier is needed, Greenwood Supplies Ltd. could be considered due to their relatively short lead time of 3 days and proximity in Freiburg, Germany.

## 3.2. USE CASE 2: ALERTS AND RAG DATA

Input: Alert (Nozzle Operation Time Error): MELTIO M600 exceed the threshold value.  
Actual: 301 h Threshold: 300 h

### 1. Get input via event or query

The system receives incoming information either as a natural-language query from a user or as a structured event generated by another component through RabbitMQ.

Regardless of format or origin, the input is normalized and transformed into a consistent internal representation.

This unified preprocessing ensures that both human-initiated requests and machine-triggered alerts enter the same reasoning pipeline for risk analysis.

```
async def query(query_input: QueryInput, db: Session) -> QueryResponse:
    """
    """
    event = get_event_from_query(query_input)

    chat_manager = ChatManager()
    plan_manager = PlanManager()
    rules_manager = RulesManager()
    plugin_manager = PluginManager()
    risk_type_manager = RiskTypeManager()
    event_manager = EventManager()

    orchestrator_agent = OrchestratorAgent(
        chat_manager=chat_manager,
        plan_manager=plan_manager,
        rules_manager=rules_manager,
        plugin_manager=plugin_manager,
        risk_type_manager=risk_type_manager,
        event_manager=event_manager
    )

    response = await orchestrator_agent.get_response(event, db)
```

```
return response
```

```
async def process_message(msg, db: Session):
    """
    """
    event = get_event_from_message(msg)

    chat_manager = ChatManager()
    plan_manager = PlanManager()
    rules_manager = RulesManager()
    plugin_manager = PluginManager()
    risk_type_manager = RiskTypeManager()
    event_manager = EventManager()

    orchestrator_agent = OrchestratorAgent(
        chat_manager=chat_manager,
        plan_manager=plan_manager,
        rules_manager=rules_manager,
        plugin_manager=plugin_manager,
        risk_type_manager=risk_type_manager,
        event_manager=event_manager
    )
    return await orchestrator_agent.get_response(event, db)
```

## 2. Select risk type

The system loads all existing risk types from the knowledge base and uses a large language model to compare the incoming input against these definitions.

Through semantic similarity assessment, it identifies which risk type best matches the meaning and context of the input.

If no suitable match exists, a new risk type is automatically generated and stored, enabling the system to adapt to previously unseen risks.

```
async def get_response(self, event_query: EventModel, db: Session) ->
QueryResponse:
    """ """

    plan, decision, chat_id = await self._select_plan_advanced(
        event_query=event_query, db=db
    )

[...]
```

```
async def _select_risk_type(
    self, possible_risk_types: List[RiskTypeData], event: EventModel
) -> tuple[str, str]:
    """ """
```

```

    Use an LLM to select the most appropriate risk_type given the user's
query.
    """

    user_query = event.query

    risk_blocks = "\n\n".join(
        f"PLAN ID {rt.id}:\n{rt.description}" for rt in possible_risk_types
    )

    template = f"""
    You are an expert orchestrator that decides which risk type description
best fits a user's question.

    Here are the available risk type descriptions:

    {risk_blocks}

    User's question:
    "{user_query}"

    Decide which risk type description is most related to this question.
Respond ONLY with the ID (example: "1").
    """

    prompt = PromptTemplate(template=template)

    try:
        result = await (prompt | self.llm).ainvoke({})
        decision = result.content.strip() if result else ""
    except Exception as e:
        print(f"Plan selection failed via LLM: {e}")
        decision = ""

    return decision

```

### 3. Get plans from DB based on risk

After the risk type is determined, the system retrieves all response plans associated with that category from the database.

It evaluates these plans—considering metadata, usage history, or semantic annotations—to select the one that most closely fits the situation.

If none is appropriate, the system records this and prepares for dynamic plan creation during reasoning.

```

class PlanManager(PlanManagerBase):
    async def get_plans(self, db: Session) -> list[PlanData]:
        plan_list_object: list[Plan] = db.scalars(select(Plan)).all()

```

```

return [
    PlanData(
        id=plan.id,
        name=plan.name,
        description=plan.description,
        instructions=plan.instructions
    ) for plan in plan_list_object
]

async def get_plan(self, id: int, db: Session) -> PlanData:
    plan: Plan | None = db.scalars(select(Plan).where(Plan.id == id)).first()

    if plan is None:
        return None

    return PlanData(
        id=plan.id,
        name=plan.name,
        description=plan.description,
        instructions=plan.instructions
    )

class RulesManager(RulesManagerBase):
    async def get_rules(self, db: Session) -> list[RuleData]:
        rule_object: list[Rule] = db.scalars(select(Rule)).all()
        return [
            RuleData(
                id=rule.id,
                name=rule.name,
                description=rule.description,
                expression=rule.expression,
                priority=rule.priority
            ) for rule in rule_object
        ]

```

#### 4. Calls RAG tool to get data

When the ReAct agent determines that additional contextual or historical information is needed, it invokes the RAG (Retrieval-Augmented Generation) tool.

The RAG tool queries embedded knowledge sources—such as document collections, logs, or domain records—to retrieve semantically relevant information.

The retrieved data is then injected into the agent's working memory, enriching its factual context and enabling more accurate reasoning in subsequent steps.

```

if str(decision) == "2":
    response: ResponseModel = query_chroma_rag(event_query.query)

```

```

def query_chroma_rag(question: str):

```

```

"""Query an existing Chroma DB using Azure Chat and embeddings."""

embeddings = AzureOpenAIEmbeddings(
    api_key=settings.azure_openai_api_key,
    azure_endpoint=settings.azure_openai_base_url,
    azure_deployment=settings.azure_openai_embedding_model_deployment_name,
    openai_api_version=settings.azure_openai_api_version
)
with open("imc/databases/chroma_mock/mock_data.txt", "r", encoding="utf-8")
as f:
    additional_context = f.read()

vectordb = Chroma(
    persist_directory="imc/databases/chroma_mock",
    embedding_function=embeddings
)
retriever = vectordb.as_retriever(search_kwargs={"k": 3})
llm = LLM

prompt = ChatPromptTemplate.from_template(
    """You are an intelligent assistant helping answer user questions based on
the provided context.

Question:
{input}

Context:
{context}

Additional context:
{additional_context}

Provide a clear and concise answer based only on the context above.
If the context does not contain enough information, say you don't know.
"""
)

combine_chain = create_stuff_documents_chain(llm, prompt)
rag_chain = create_retrieval_chain(retriever, combine_chain)

result = rag_chain.invoke({"input": question,
"additional_context":additional_context})
print("Answer:\n", result["answer"])
print("Context:")
for d in result.get("context", []):
    print("-", d.page_content[:200], "... \n")

return ResponseModel(

```

```
question=question,
answer=result["answer"],
context=ContextModel(
    context_documents=result["context"]
)
)
```

**RAG Context:**

```
"context_documents": [
    {
        "id": null,
        "metadata": {},
        "page_content": "7. Machine Error – AL06 Cooling Error\n\nSolution:\n\nThe AL06 error indicates improper cooling pressure or flow (Page 62). Follow the documented procedure: adjust the O2 sensor, probe, and door valves to 1–1.4 L/min and verify that head and gantry flow auto-regulate to approx. 8 L/min and 2.4 L/min.\n\nIf pressure remains above 0.50 MPa, adjust the chiller bypass until stable operation at 0.45–0.48 MPa is achieved.\n\n8. Nozzle Operation Time – Current: 301 h (Limit: 300 h)\n\nSolution:\n\nThe wire nozzle has exceeded its typical lifespan of 50–300 hours (Page 178). Replace the nozzle following the inspection and removal instructions to avoid print defects.\n\nContinued use past this threshold may cause beam misalignment and deposition quality issues.\n\n9. Nozzle Needs Replacement Message",
        "type": "Document"
    },
    {
        "id": null,
        "metadata": {},
        "page_content": "9. Nozzle Needs Replacement Message\n\nSolution:\n\nSince the operating hours exceed the recommended limit, replace the nozzle immediately following the 5-step maintenance procedure. Visually inspect the nozzle and compare it to wear levels A, B, and C to confirm the condition.\n\nReinstall the shield gas nozzle after replacement to restore proper operation.\n\n10. Power Peak Consumption – Current: 11.2 kW (Limit: 10.6 kW)\n\nSolution:\n\nA peak consumption of 11.2 kW exceeds the typical maximum noted in the specifications (Page 38). Check the electrical installation, cable sizing, and power distribution to ensure they match the machine requirements.\n\nHigh peak draw may signal issues with the chiller, lasers, or internal electronics, and should be investigated to prevent electrical overloads.",
        "type": "Document"
    },
    {
        "id": null,
        "metadata": {},
        "page_content": "Regulate chiller bypass\n\nIf the chiller still displays the error, adjust the chiller bypass by opening the valve little by little and repeat the cooling process until the displayed value is around 0.45-0.48MPa.\n\nOnce the chiller is running without any alarm, check the flow meters located on the Meltio M600 left side panel:\n\nFrom left to right, approximately values should be set to:\n\nHead - 8 - 8.5L/min (error displayed if below 7.8L/min)\n\nGantry - 2.2 - 2.5L/min (error displayed if below
```

```
2.1L/min)\nO2 sensor - 1 - 1.5L/min (error displayed if below 0.7L/min)\nProbe - 1 -  
1.5L/min (error displayed if below 0.7L/min)\nDoor - 1 - 1.5L/min (error displayed if below  
0.7L/min)\n\n-----\n\n## ERROR HANDLING\n\n1. Storage Temperature –  
Current: 65 °C (Limit: 50 °C)",  
    "type": "Document"  
    }  
  ]
```

## 5. Return final response

After retrieving and integrating information from the RAG tool, the system synthesizes the enriched context and generates a final response that incorporates the newly retrieved knowledge.

This response includes the context of how the RAG-sourced data influenced the outcome, ensuring transparency and justifying why the final recommendation or answer is valid.

```
return QueryResponse(chat_id=chat_id, response=response)
```

### Final response

The alert indicates that the nozzle operation time for the MELTIO M600 has exceeded the threshold value, with the actual time being 301 hours compared to the threshold of 300 hours. The solution is to replace the wire nozzle, as it has surpassed its typical lifespan of 50–300 hours. Continued use beyond this threshold may lead to beam misalignment and deposition quality issues. Follow the inspection and removal instructions to replace the nozzle and avoid print defects.

## 4. IMPLEMENTATION & DEPLOYMENT

The platform is a fully containerized, scalable, and data-driven management system designed for flexible deployment and high maintainability. Its architecture combines modern web technologies with powerful backend logic, a relational data layer, and an integrated analytical dashboard, making it suitable for both development environments and production infrastructures.

The platform is composed of a **React-based frontend**, a **Python FastAPI backend**, and a **PostgreSQL database**, all fully **containerized with Docker**.

The architecture ensures scalability, portability, and consistency across development, testing, and production environments.

### SYSTEM OVERVIEW

The system follows a microservice-oriented architecture where each component runs in an independent Docker container.

A central **Docker Compose** configuration orchestrates the startup and interconnection of all services, which include:

- A **front-end service** providing the user interface, built with **React** and served through **Nginx**.

- A **back-end service** implemented in **Python (FastAPI)**, responsible for the business logic, database management, and WebSocket connection.
- A **PostgreSQL database** for data persistence and relational storage.

A **Metabase** dashboard is integrated into the system to provide powerful visualization and analytical capabilities. It connects directly to the PostgreSQL database, enabling the creation of charts, reports, and custom dashboards without additional coding.

Through Metabase, users can gain insights into operational data, monitor performance indicators, and support decision-making through data analytics.

Connections to the remote database are established using **SSL/TLS encryption** to maintain data integrity and security. The modular design also allows easy integration with external authentication providers or message brokers if needed.

---

## 5. CONCLUSIONS AND NEXT STEPS

---

Conclusions and achievements:

- Successfully developed and delivered a fully operational software platform by adhering to established software-engineering best practices and leveraging state-of-the-art technologies, including containerization, artificial intelligence, large language models, and real-time monitoring frameworks.
- Designed and deployed a user-friendly interface that enhances accessibility, usability, and overall interaction efficiency for both technical and non-technical stakeholders.
- Integrated multiple tools within a coherent ecosystem and enabled seamless communication and data exchange among them, thereby improving interoperability and workflow continuity.
- Deployed the platform on a publicly accessible internet endpoint, ensuring secure, reliable, and scalable access for all project partners and relevant external stakeholders.

Next steps:

- Ensure operational use cases across all pilot systems, including a fully functional Complex Event Processing (CEP) component sending real data to the IMC.
- Increase the robustness and reliability of existing tools, with emphasis on NSQA and LLMs.
- Initiate the implementation of new tools, prioritizing those currently available in simulated form (e.g., Blueprints and Resilience Strategies).
- Develop new interfaces to enhance integration and communication among tools, focusing on D3.1, D3.2, D3.4, D4.3, and D4.4.
- Determine the operational framework for accessing the NMP, evaluating alternatives such as internal cloud deployment, proprietary systems, or the Narrate platform.
- Assess which additional functionalities can be incorporated into the IMC (e.g., automated email dispatch, customizable workflows, integration with external APIs) to improve tool efficiency and inter-tool communication.
- Continue implementing and delivering incremental updates to the platform.

---

## 6. REFERENCES

---

- [1] Michael Papazoglou, Amal Elgammal, Bernd Kraemer, NARRATE (2025). D1.4 ARCHITECTURAL REQUIREMENTS, 2025/05/29
- [2] González, Tomás; Correa, David, NARRATE (2024). D3.3(a) End-to-end AI-driven visibility model & support DSS (a), 2025/10/12
- [3] FRANK WERNER, BLU 5<sup>th</sup> GA meeting D2.2, 2025/11/25