Deployed AI Agents for Industrial Asset Management: CodeReAct Framework for Event Analysis and Work Order Automation

Nianjun Zhou[*1†], **Dhaval Patel**[*1], **Anamitra Bhattacharyya**[2]

[1]IBM, Research Division, Yorktown Heights, NY 10598, USA
[2]IBM, Maximo Division, Lowell, MA 01851-5114, USA
jzhou@us.ibm.com, pateldha@us.ibm.com, abhattacharyya@us.ibm.com

Abstract

Maintenance of mission-critical industrial assets is frequently hindered by fragmented data, inconsistent record-keeping, and limited access to analytical expertise, resulting in reactive rather than predictive practices. We present *CodeReAct*, an AI-powered agentic framework deployed in large-scale facilities to automate event analysis and work order (WO) management. CodeReAct extends the ReAct paradigm by embedding executable Python code within the Thought–Action–Observation (TAO) loop, enabling natural language interaction, grounding heterogeneous alerts and work orders into structured Business Objects (BOs), and dynamically invoking analytic functions for forecasting, anomaly correlation, and maintenance recommendations. This architecture reduces manual data science intervention, improves adaptability, and supports reuse across asset types.

Deployed in a mission-critical data center and productionized in Maximo, CodeReAct manages pumps, chillers, AHUs, compressors, cooling towers, and other mechanical and electrical systems. Evaluation with 36 representative maintenance utterances showed that outer-loop reflection and adaptive temperature improved task completion by up to 20%, while ablation studies confirmed the importance of reasoning in addition to code execution. Business validation revealed seasonal failure patterns, bundling opportunities, and predictive accuracy trends. In production, site engineers reported 25–40% faster diagnostics, fewer unplanned downtime events, and reduced reliance on specialized analysts. Lessons learned highlight the importance of structured BOs for grounding analytics, runtime safeguards to mitigate hallucinations, and adaptive model control for consistent execution. These results demonstrate how deployed agentic AI can deliver measurable business value in predictive and strategic maintenance planning.

Introduction

Mission-critical facilities depend on mechanical, electrical, and control systems to operate without interruption. Failures in these systems can cascade across operations, with industry estimates placing the cost of unplanned downtime at **$100,000 to over $1 million per hour** depending on the sector. Timely fault detection, accurate diagnosis, and efficient maintenance planning are therefore essential for operational resilience (Serradilla et al. 2022; Gulati and Smith 2009). In our case, the system is deployed across enterprise data centers, where chillers, CRACs, AHUs, boilers, and pumps support high-density computing workloads. Yet in practice, maintenance teams face persistent obstacles: **(1) Fragmented data sources** across IoT platforms, building management systems, OEM feeds, CMMS, and spreadsheets (Balali et al. 2020); **(2) Inconsistent records**, with work orders lacking standardized codes or containing incomplete, jargon-heavy text (Chuang et al. 2020; Goyal et al. 2016); **(3) Alert overload**, as IoT platforms generate thousands of daily notifications, many spurious or low-priority (Langbridge et al. 2024); and **(4) Limited analytical expertise**, leaving front-line engineers without timely support for root cause analysis or forecasting (McGuire, Ariker, and Roggendorf 2013; Tawil et al. 2024).

These gaps delay fault diagnosis, reinforce reactive maintenance, and waste labor and materials. Traditional preventive maintenance, which is based on fixed schedules or OEM recommendations, cannot adapt to changing conditions, often leading to unnecessary servicing and missed early warning signs (Gulati and Smith 2009). While IoT sensors and anomaly detection models create opportunities for condition-based maintenance, their impact is frequently undermined by the “last-mile” problem: advanced analytics exist, but are not embedded into daily workflows (Petsinis, Naskos, and Gounaris 2021; Ye et al. 2024; Neuberg 2003).

To address this, we developed and deployed the *CodeReAct* framework: an AI-powered agent that extends the Large Language Model (LLM) ReAct paradigm (Yao et al. 2022) with **executable Python code generation** in the Thought–Action–Observation loop, integrates structured **Business Objects (BOs)** for consistent interpretation of maintenance data, and employs adaptive reflection mechanisms for reliable decision-making. During the development, deployment, and production process, *CodeReAct* has shown measurable gains: **25–40%** faster diagnostics, fewer unplanned downtime events, reduced dependence on specialists, and actionable insights such as seasonal failure patterns and maintenance bundling opportunities. By embedding agentic AI into daily workflows, it transforms asset management from reactive response to predictive and strategic planning (Tao et al.

[*]These authors contributed equally.
[†]Corresponding author.

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

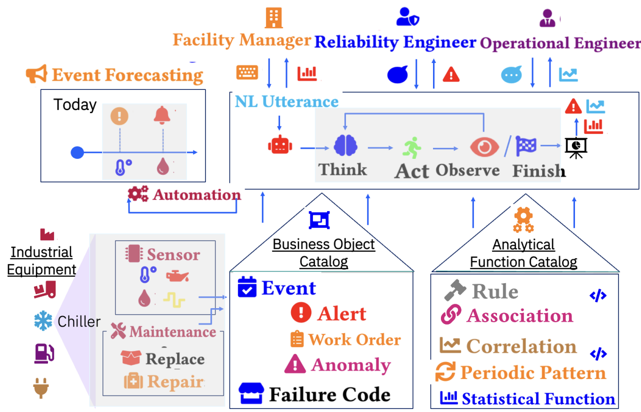


Figure 1: Schematic view of the deployed AI agent-driven framework for event forecasting and predictive maintenance.

2024; Jimenez et al. 2023; La Rosa, Hulse, and Liu 2024).

Operational Context and Maintenance Challenges

The facilities portfolio studied in this work spans thirty-two geographically distributed sites, each maintaining a diverse mix of mechanical, electrical, and control assets to support critical operations. Assets include air handling units (AHUs), computer room air conditioners (CRACs), centrifugal chillers, boilers, and heat exchangers (HXUs), among others. Many of these systems are tightly coupled, so failures in one component can cascade to others, complicating both diagnosis and maintenance prioritization. Table 1 lists the top 10 largest sites, not the full 32-site portfolio; the remaining sites have smaller asset counts but follow the same maintenance and integration workflow.

Scale and Complexity of Asset Portfolios

Across the network, asset counts reach into the tens of thousands and span multiple generations of equipment and control technologies. Annual maintenance programs may involve hundreds of thousands of work orders and continuous streams of IoT-enabled alerts and anomaly notifications (Serradilla et al. 2022; Petsinis, Naskos, and Gounaris 2021). Seasonal load variations, such as summer cooling peaks or winter heating demands, drive distinct failure patterns across site types and climate zones.

Asset distribution varies substantially across sites. For example, **POKMAIN** operates 164 AHUs, 5 boilers, 124 CRAC units, 11 chillers, and 28 HXUs to support dense computing loads, while other sites maintain more specialized configurations. Table 1 highlights this heterogeneity by listing the ten largest sites by asset counts. The variation underscores challenges in scheduling maintenance, managing spares, and optimizing performance at scale.

Fragmented Data and Pain Points

Although large volumes of operational data are collected, they often reside in siloed platforms with inconsistent identifiers and limited cross-referencing. This fragmentation delays analysis and can obscure patterns already present in historical records. Table 2 summarizes representative asset signals, primary data sources, and recurring pain points. These include unstructured work order text, inconsistent coding schemes, alert overload, and reliance on scarce expert analysts—factors that hinder the scalability of traditional maintenance practices.

Site Name	AHU	Boiler	CRAC	Chiller	HXU
BLDMAIN	412	4	145	12	5
RCHMAIN	259	0	18	6	53
POKMAIN	164	5	124	11	28
BRMM	113	0	0	7	0
GDLPLANT	101	0	0	0	0
AUBHILLS	23	0	72	2	0
RTPMAIN	73	0	0	3	0
DUBMAIN	49	9	0	7	0
STLMAIN	24	0	36	4	0
YKTMAIN	42	3	9	0	8

Table 1: Top 10 Sites by Important Asset Counts

Example: Seasonal Maintenance Burden

Figure 2 illustrates these issues at the **POKMAIN** site. Preventive maintenance (blue) appears relatively evenly distributed across the year, while corrective maintenance (red) is clustered in summer months. This seasonal concentration reflects load-driven stress and highlights the limitations of rigid, calendar-based scheduling. Such recurring patterns emphasize the need for approaches that adapt to actual operating conditions.

Traditional Maintenance Practices

Historically, maintenance programs combined calendar-based preventive schedules with reactive corrective actions. Preventive intervals were typically based on OEM guidelines or fixed cycles, without adjustment to real-time asset conditions (Chuang et al. 2020; Goyal et al. 2016). Data from IoT platforms, BMS, OEM feeds, CMMS, and manual logs were rarely integrated, requiring engineers to reconcile sources before analysis. The combination of asset diversity, global scale, and operational risk often prevented a unified picture of asset health, with predictive patterns discovered only after downtime had already occurred.

These operational realities, illustrated by Tables 1 and 2 as well as Figure 2, motivate the structured representation of asset and event data that we present in the following section. In addition, Figure 3 summarizes how these challenges informed a staged deployment strategy, moving from early research concepts through client validation to system and product integration.

Deployment Environment and Integration

The *CodeReAct* framework (Yao et al. 2022; Wang et al. 2024) is a general-purpose LLM agent paradigm that integrates reasoning and code execution within a Thought–Action–Observation (TAO) loop. Building on this

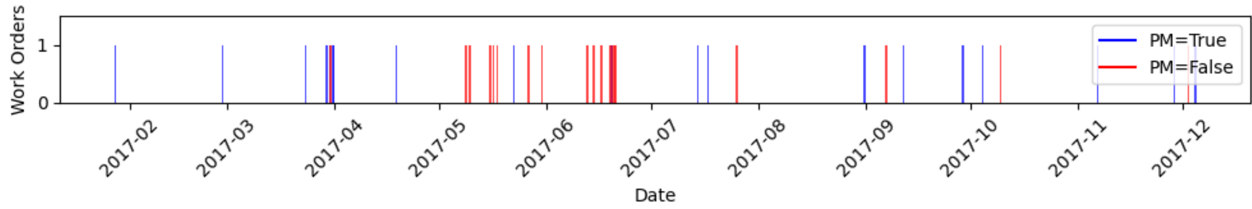


Figure 2: Work Order Events in 2017 for a Chiller at the POKMAIN site: Preventive Maintenance (Blue: evenly spaced) vs. Corrective Maintenance (Red: clustered in summer). Seasonal clustering highlights limitations of traditional scheduling.

Asset Types & Typical Signals	
Chillers	Load (tons), kW, kW/ton, evaporator/condenser inlet/outlet temperatures, flow rates
Pumps	Head, flow, motor current, vibration
AHUs	Supply/return temperatures, static pressure, fan speed
Cooling towers	Approach temperature, fan VFD%, basin level
Compressors and Generators	kW, FLA%, oil pressure/temperature, vibration
Primary Data Sources	
IoT/BMS	Time-series telemetry; rule-based alerts
CMMS	Work orders (preventive/corrective), timestamps, labor
SME artifacts	Failure modes, procedures, asset metadata
Recurring Pain Points	
Labeling	Unstructured WO text; inconsistent failure codes
Fragmentation	Siloed systems; difficult joins across IDs and time
Data quality	Misaligned timestamps, inconsistent units, dropouts
Alert overload	High alert volumes with unclear prioritization
Seasonality	Corrective WOs cluster by climate and asset type
Expert bottlenecks	Advanced analytics require scarce specialists

Table 2: Operational Context at a Glance: Representative Assets, Data Sources, and Recurring Pain Points Motivating an Agentic-driven Approach

paradigm, we developed our own open-source implementation, *ReActXen* (Rayfield et al. 2025), and extended it for industrial asset management. Our contribution lies not in introducing CodeReAct itself, but in adapting and operationalizing it for mission-critical maintenance workflows. Specifically, we embed domain-grounded **Business Objects (BOs)**, standardized failure codes, analytic functions, and structured natural language utterances into the CodeReAct loop, enabling the agent to operate directly on heterogeneous maintenance data. This domain adaptation allows the agent to reason over structured records, execute analytic pipelines, and return actionable outputs within a production deployment at the POKMAIN data center in Poughkeepsie, NY. Due to page constraints, the visualizations of the standardized failure/maintenance taxonomy and the BO-based linkage between alerts and work orders are provided in the extended version of this work[1]

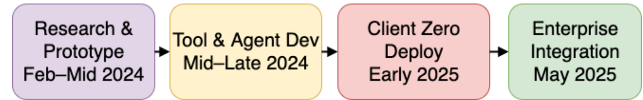


Figure 3: Timeline of research, prototype, and deployment stages for the CodeReAct framework, showing progression from early concept to enterprise-wide integration.

Business Objects (BOs)

A central contribution of this deployment is the introduction of **Business Objects (BOs)**: schema-driven representations that unify heterogeneous maintenance data into a consistent, machine-readable form. Implemented as `Pydantic` classes in Python, BOs include typed attributes, validation rules, and serialization into String (digesteable by LLM) or JSON. This ensures that records from CMMS, IoT sensors, OEM alerts, and anomaly detectors are normalized into a common structure so that they are consumable both by analytic functions and by LLMs that interpret structured objects.

In total, we developed a set of **BOs**, each capturing on average ten attributes (additional details are provided in the extended version[1]). These cover the key entities of industrial asset management:

- **FailureCode BO** (4 attributes): Represents a standardized failure or maintenance code an asset experienced, aligned with ISO 14224.
- **Equipment BO** (5): Captures asset-level metadata such as ID, type, location, site, and status.
- **Date BO** (1): Defines temporal anchors such as start date, end date.
- **DateRange BO** (2): Defines temporal anchors such as time ranges of interested;

[1] The full appendix and algorithms, including expanded related work, detailed definitions of Business Objects, the complete set of utterance categories, additional experimental results, and other supplementary materials, will be provided in the arXiv version.

- **WorkOrder BO** (10): Encodes preventive/corrective work orders with description, standardized codes, priority, scheduling, and status.
- **Event BO** (12): Aggregates alerts, anomalies, and work orders into a unified event tied to a root cause.
- **Alert BO** (8): Stores IoT/BMS alert signals including rule ID, condition, severity, and triggering time and duration.
- **Anomaly BO** (7): Encapsulates KPI anomalies detected by ML models, including type, magnitude, and affected asset.
- **Recommendation BO** (6): Suggests work orders, bundles, or preventive plans generated by the agent.

Technical Definition. A BO defines a schema with attributes, validation logic, and serialization methods. For example, a Failure/Maintenance Code BO can be defined as:

```
1 class FailureCodeBO(BaseModel):
2     category: str # Broad class (e.g.,
          Routine Checks)
3     primary_code: str # ISO14224-based
          primary code
4     secondary_code: str # More specific
          sub-code
5     description: str # Human-readable
          explanation
```

Example Instantiation. An instance of this BO may represent a compressor overhaul:

```
1 FailureCodeBO(
2     category="Maintenance and Routine
          Checks",
3     primary_code="MT006",
4     secondary_code="MT006a",
5     description="Compressor Overhaul"
6 )
```

Key Contribution with BO Specification. Standardizing heterogeneous data into BOs achieves two goals: (i) it eliminates fragmentation across systems by providing a single schema for diverse inputs, and (ii) it enables seamless interaction with LLMs, which can interpret and reason over BOs to generate analytic logic and maintenance recommendations. Although not yet fully implemented, BOs can also support access-control: individual attributes may be role-gated, such that when a user lacks permission, the value is replaced with "Hidden". This design ensures sensitive data is protected while preserving schema consistency. Additional details of the BO structure, including full function descriptions and summary tables, are provided in the extended version of this work[1].

Analytical Functions

Analytical functions operate directly on BOs to deliver site-relevant capabilities. Each function is defined along two complementary aspects: *design* and *implementation*. The **function design** specifies the semantic contract—its name, task, inputs, outputs, and an illustrative example. For instance, `predict_next_work_order_probability` operates on an equipment identifier and a date range, returning a list of tuples (`ISO_Failure_Code`, probability). The **functional implementation** realizes this design as executable code. Implementations are atomic, domain-specific units that can be reused or extended by LLMs for in-context learning. For example:

```
1 @validate_inputs
2 def get_events(equipment: Equipment,
      date_range: DateRange) ->
      PickleFilePath:
3     """Retrieve all events, including new
          alerts or anomalies,
4       for a specific equipment within a
          given time range."""
```

Building on this design–implementation framework, we implement five categories of functions spanning over a dozen techniques (with detailed examples in the supplementary version[1]): data standardization (e.g., NLP-based entity extraction mapping CMMS free text to ISO 14224 codes), correlation and linking (e.g., aligning alerts and work orders via Allen's Interval Algebra (Allen 1983)), pattern recognition (e.g., frequent sequence mining for seasonal maintenance trends), forecasting and prediction (e.g., RUL estimation (Serradilla et al. 2022) and anomaly-driven forecasting (Langbridge et al. 2024)), and recommendation and bundling (e.g., grouping corrective work orders to reduce downtime (Neuberg 2003)). Collectively, these functions serve as reusable analytic building blocks grounded in the BO schema, providing consistent inputs and outputs to support scalable, domain-aware decision making across diverse assets and maintenance contexts.

Utterance Design

Interaction with the CodeReAct framework occurs through *utterances*: natural language queries posed by operators or scheduled tasks. Each utterance maps human intent onto BOs and analytic functions, ensuring queries can be executed without ambiguity. An utterance must specify four elements: (1) target asset, (2) intended action (retrieval, summarization, prediction), (3) timeframe, and (4) optional constraints.

In practice, we developed a representative set of **36 utterances** across nine categories, ensuring coverage of retrieval, summarization, forecasting, bundling, and anomaly analysis (full examples are provided in the extended version[1]). For instance:

> *"Retrieve all corrective work orders for Chiller 9 in 2017 and group them into bundles where tasks occur within two weeks of each other."*

This utterance specifies the asset (Chiller 9), action (retrieval + bundling), timeframe (2017), and constraint (bundle within two weeks). Such structured utterances form the basis for experiments in Section Experiments and Lesson Learned.

In the deployment, BOs and analytic functions are embedded into the site's ecosystem through API connectors and ingestion pipelines that normalize alerts, fetch and enrich work orders, and assemble related records into coherent event objects. At runtime, the *CodeReAct* agent retrieves these BO instances, applies analytic functions to extract insights or predictions, and returns recommended work orders to the CMMS for review and approval. This integration eliminates data fragmentation, reduces reliance on scarce experts, and embeds decision-support directly into daily operations.

Alternative designs were studied during system development. First, RAG (retrieval-augmented) agents without code execution were simpler but unable to perform multi-step event correlation, cross-source reasoning, or deeper analytical tasks. Second, a SQL-like tool provided strong precision for structured data retrieval but lacked flexibility for deep analysis capabilities. Third, fixed-function pipelines without auto-coding lacked the flexibility to handle the variety of utterance scenarios encountered in practice. These comparisons motivated the mixed reasoning–execution architecture of *CodeReAct*, which balances interpretability, flexibility, and maintainability.

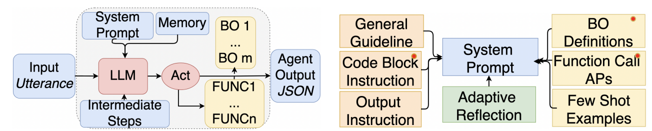
Solution Framework – Integration with *CodeReAct* Framework

Building on the deployment environment described in previous section, this section presents the architecture of the *CodeReAct* framework. At its core is the *CodeReAct* Agent, which extends the ReAct paradigm (Yao et al. 2022) by embedding Business Objects (BOs) and analytic functions into a reasoning–execution loop. This design enables real-time event analysis and automated work order (WO) management, allowing maintenance teams to query operational data in natural language while producing validated, actionable outputs. Our design follows prior work on LLM-agent integration in industrial contexts (Ye et al. 2024; Tao et al. 2024), but introduces runtime Python execution and reflection mechanisms tailored for mission-critical facilities.

Framework of CodeReAct Agent

The *CodeReAct* Agent is an LLM-powered reasoning and execution component tailored for industrial asset management. It extends the ReAct (Reason + Act) paradigm by embedding executable Python code generation directly into the Thought–Action–Observation (TAO) loop. This enables the agent not only to reason about a user request in natural language but also to synthesize, execute, and validate code against live operational data (check the details at (Rayfield et al. 2025)).

The system prompt initializes the agent with: 1) **Business Object (BO) definitions** unifying alerts, anomalies, and work orders; 2) **Analytic function signatures** for standardization, correlation, forecasting, and optimization; and 3) **Operational context**, including asset types (e.g., chillers, pumps, AHUs), available data sources (e.g., sensor streams, CMMS logs, OEM fault codes), and integration points to enterprise systems. This context grounds the agent so that



(a) CodeReAct Agent with BOs and Functions

(b) System Prompt (red dot boxes enable code generation)

Figure 4: Agent and System Prompt Composition

utterances are interpreted with respect to the correct equipment, data feeds, and site-specific constraints.

Each utterance (additional examples are provided in the extended version[1]) specifies an asset, an intended action (retrieval, analysis, prediction), a required time frame, and optional constraints. Guided by the system prompt, the agent dynamically invokes analytic functions mapped to BOs (full function lists are also included in the extended version[1]). These functions are executed as Python code blocks, ensuring transparent integration with live data streams and reproducible outputs.

Operational Workflow

At runtime, the *CodeReAct* agent executes a Thought–Action–Observation (TAO) loop enhanced with Python code generation and execution. Figure 5 illustrates this process for the utterance:

> *"Using your knowledge of the asset, suggest a corrective work order for alert RUL0014 ('Chiller – Cooling Substance ΔT Low') on Chiller 9, observed on March 15, 2009, when the load was above the upper bound in a closed-loop water-cooled system."*

Unlike a text-only ReAct agent, *CodeReAct* extends beyond retrieval to contextual reasoning validated against live telemetry before issuing a concrete work order. Embedding executable Python code within the TAO loop enables complex reasoning and data retrieval in a **single LLM call**, reducing token consumption and mitigating error accumulation. The agent also supports **batch and cross-asset analysis** by dynamically generating queries over multiple Business Objects (BOs), reuses **domain-specific analytic functions** (e.g., alert–WO correlation, anomaly detection, sequence mining), and persists results in **structured formats** (JSON, Pickle) for direct CMMS integration and auditable execution.

For robustness in production, several safeguards and workflow extensions are integrated, including **reflection placeholders** to validate intermediate results before recommendations are issued, **deadlock detection** and **error recovery** mechanisms to parse execution failures and regenerate corrected code, **adaptive thresholding** to tune anomaly detection parameters dynamically from historical statistics and reduce false positives, **loop variations** (no-think, single-reflection, multi-reflection) to balance diagnostic accuracy and latency, and **runtime Python validation** with **adaptive LLM temperature control** to suppress unsafe outputs and stabilize execution. More details of the execution process are provided in the extended version[1].

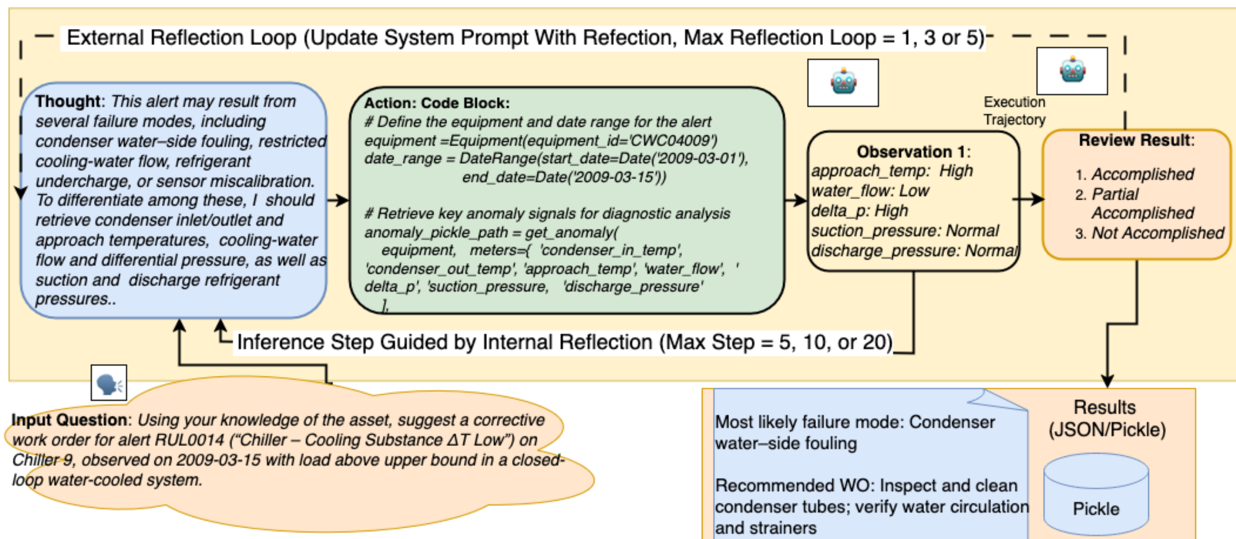


Figure 5: Execution trajectory of the *CodeReAct* Agent for a representative alert. The TAO reasoning–execution loop shows how the agent interprets the input, generates diagnostic code, observes sensor anomalies, and reflects to infer the most likely failure mode. Details of inner and outer reflection are provided in Section Experiments and Lesson Learned.

Automated Real-Time Work Order Management

An "online agent" instance (Figure 6) operationalizes these capabilities by: 1) retrieving alerts, anomalies, and unresolved WOs; 2) fetching historical context; 3) executing validated utterances for prioritization and recommendations; 4) pushing results into CMMS and notifying stakeholders; and 5) archiving outputs and execution logs for auditability.

This near-real-time triage reduces reliance on scarce expert analysts and accelerates maintenance decision-making.

Together, these mechanisms ensure CodeReAct produces valid, actionable insights in mission-critical environments.

Experiments and Lesson Learned

This section evaluates the proposed framework through structured experiments, assessing its capabilities based on various combinations of parameters (Table 3), focusing on utterance design and performance analysis. Additional details can be found in the extended version[1]

Utterance Selection and Categories

Thirty-six utterances were crafted to span a wide range of chiller maintenance tasks, incorporating the Business Objects and analytic functions introduced in the extended version[1]. Each utterance specifies an asset, an intended action (e.g., retrieval, summarization, prediction, bundling), a time frame, and constraints for clarity and reproducibility.

The utterances cover nine categories: retrieving work orders, summarizing multi-source events, filtering meaningful alerts, reviewing past issues, detecting failures early, predicting upcoming work orders, recommending corrective work orders, bundling tasks to optimize schedules, and analyzing anomalies and alerts for root-cause diagnosis. Representative examples include retrieving corrective work orders for a given chiller in 2017, summarizing event histories for Chiller 9, recommending work orders in response to a KPI threshold violation, and bundling corrective tasks within a two-week interval to reduce downtime. The complete categorization, with examples and ID counts, is provided in the extended version[1].

Experiment Results

The evaluation consists of three set of experiments. 1) **Problem-Solving Evaluation**, which assesses the framework's ability to generate appropriate and insightful responses using the utterances provided in the extended version[1]; 2) **Adaptive Capability Assessment**, which evaluates the framework's capabilities in address an utterance task with the absent of analytic function APIs; and 3) **Ablation Study**, which focuses on the evaluation of the performance of Dynamic Python code generation versus predefined functions/tools. Table 3 outlines the parameters used in these experiments.

Problem-Solving Capability We evaluated task completion across the 36 utterances using a review agent, which classified each attempt as *Accomplished*, *Partially Accomplished*, or *Not Accomplished*. Accomplished cases were further validated by inspecting execution trajectories to ensure that the correct analytic functions were invoked with appropriate parameters. Granite completed 72% of utterances in the baseline (Table 4), while Mistral-Large achieved 56% (Table 5). Failures primarily stemmed from syntax errors, looping, or missing variables.

Outer reflections, which refine prompts across iterations, improved completion by up to 20% (Figure 7). However, the distribution of Thought–Act–Observation steps followed a long-tail pattern (Figure 8a), reflecting deadlocks and repetitive loops. To mitigate these issues, we refined the execution algorithm (details provided in the extended version[1]) by

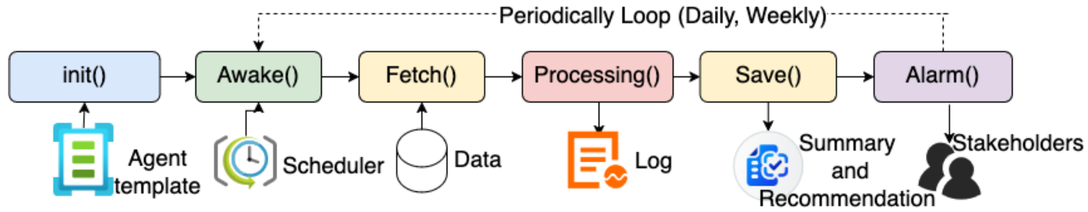


Figure 6: Deploying the online agent for real-time work order management with CMMS integration and event-driven updates.

Parameter	Candidate	Description
CodeReAct Model	Granite, Mistral-Large	The LLM are used for processing input questions, generating solutions, and executing actions.
Review Model	Granite, Mistral-Large	The LLM is used to review the trajectory of a whole iteration and decide whether the task is completed.
Inner Loop	10, 20	Number of steps in the inner loop of iteration
LLM Temp.	0, 0.1, 0.3, 0.5	Controls thought-act-observation process; A higher values expected to break the deadlock of execution.
Outer loop	1, 3, 5	Maximum number of reflections to refine execution with update system prompt if iteration > 1.

Table 3: Experiment Parameters

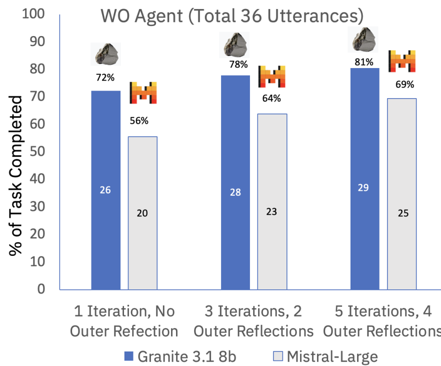


Figure 7: Impact of Outer Reflection (T=0, Inner step = 20)

incrementally increasing the model temperature across iterations. This controlled variation in code generation helped the model escape loops and further improved completion rates (Figure 8b).

To address these challenges, we incrementally increase the model temperature, which improves completion rates (Figure 8b) alongside additional iterations. These insights enable us to refine the execution algorithm (details provided in the extended version[1]) by introducing controlled variation in code generation (Action step) through a gradual increase in model temperature as iterations progress.

Code Adaptive Capability Assessment The adaptability relies on the description of the system prompt (details in Figure 4b. This experiment introduces a structured methodology to assess the adaptability by updating the system

ID	-	Review Message
1	P	Returned file path instead of JSON output.
11	N	Syntax errors prevented work order retrieval.
13	N	Incorrect date types and insufficient data.
15	N	Unterminated string error blocked execution.
17	N	Undefined functions and classes caused failure.
24	N	Failed to bundle work orders due to missing var.
27	N	Did not identify alerts due to improper filtering.
29	N	Invalid code blocked warning message generation.
30	N	Import error prevented execution of functions.
33	N	No relevant data found for work order reco.

[*] T=0, Inner step=20, P- Partial accomplishment, N - Not accomplished

Table 4: Granite-8-1-8b Unaccomplished Tasks (Success rate 72%, without Extra Outer Iteration/Reflection)

prompt. We selected six tasks with varying levels of complexity, as shown in Table 6. The corresponding utterances are presented in the extended version[1]. To streamline the evaluation, we removed all analytical function descriptions, retaining only two essential functions for retrieving work orders and events, but kept the descriptions of business objects.

Next, we introduce additional guidelines to two challenging utterances to assess **CodeReAct**'s capabilities to complete tasks without relying on pre-defined functions. We denote **U** as the original utterance, while **U+** and **U++** represent utterances with progressively detailed instructions (as shown in Table 7). The results, summarized in Table 6, indicate that **CodeReAct**, without assistance from analytical function APIs, successfully handles filtering and summarizing tasks. However, it encounters challenges in solving bundling and work order prediction tasks. By incorporating additional guidelines in the extended utterances, only the bundling task is effectively accomplished.

Ablation Study: Impact of the *Act*-Only Setting To evaluate the role of explicit reasoning in CodeReAct, we ablated the *Thought* step, retaining only the *Act–Observe* loop (akin to CodeAct). Using the Granite-3.1-8B model on 36 utterances, we observed a consistent and substantial reduction in task completion across all outer-loop configurations. With 1 outer iteration, task completion dropped from 24 to 12 (-50%); with 3 iterations, from 28 to 16 (-43%); and with 5 iterations, from 29 to 20 (-31%), despite identical temperature (0.0) and 20 inner-loop steps. These results indicate that executable actions alone are insufficient: intermediate reasoning plays a critical role in grounding actions, sustaining

ID	S	Review Message
2	N	Syntax errors in JSON caused looping.
3	N	Datetime handling led to repetitive failures.
4	N	Repeated syntax errors prevented valid JSON.
8	P	Looping errors in JSON formatting.
9	N	Incomplete code execution with syntax errors.
11	N	Incorrect file path returned not the bundled WOs.
12	N	Unterminated string literal caused repetitive failures.
13	N	JSON formatting errors led to invalid responses.
15	N	Looping due to undefined variable 'total_work_orders'.
17	N	Failure to initialize 'recommended_primary_codes' correctly.
18	N	Incorrect API handling caused repeated failures.
20	N	Looping in JSON formatting.
24	N	Failure to execute 'find_bundles' function correctly.
31	N	Syntax error in JSON formatting caused looping.
34	P	Failure to check corrective WOs and recommend one.

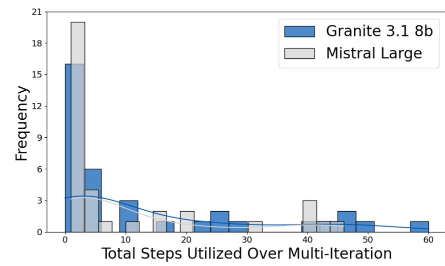
* T=0, Inner step=20, P: Partial Accomp., N: Not Accomplished

Table 5: Mistral-Large Unaccomplished Tasks (Success rate 56%, without Extra Outer Iteration/Reflection)

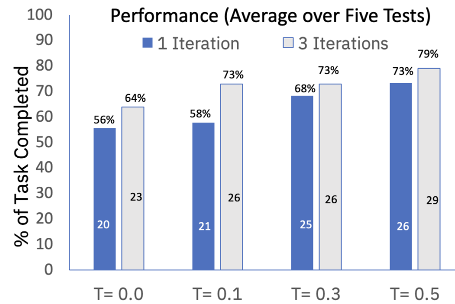
Category	ID	U	U+	U++
Get Work Orders (distribution)	1	Yes	-	-
Get Work Orders (filtering)	3	Yes	-	-
Summarize Events	5	Yes	-	-
Bundle Work Orders[4]	24	Hall	Yes-	Yes
Predict Work Orders[5]	14	No	No	Hall

[0] Mistral Large, T=0.1, inner step=20, no outer reflect, 5 tests
[1] Yes - Task completed successfully; Yes-: Only one test failed due to Hallucination; Hall - Hallucination output; No - Task not completed

Table 6: Code Adaptability Assessment[0]



(a) Dist. of # Steps Utilized (T=0,Iteration=5)



(b) Impact of Temp/Reflect (Mistral-Large)

Figure 8: Performances (T=0, Inner step = 20)

Category	U+ Utterance	U++ Utterance (trimmed)
Bundle Work Orders (ID 24)	(U) + Bundles should not have overlap.	(U, U+) + Group corrective work orders based on actual finish dates within an allowed time gap. Discard bundles with only one work order.
Predict Work Orders (ID 14)	(U) + Using a Markov approach and specify the state with the primary failure code.	(U, U+) + Use a Markov model to forecast work orders based on state transitions from historical data and update predictions using transition probabilities.

Table 7: Updated Utterances

progress across iterations, and improving overall reliability.

Lessons Learned from Deployment

Deployment at the POKMAIN data center surfaced several key insights. **(1) Accuracy vs. usability:** Engineers valued interpretable outputs and auditable logs as much as accuracy, as these supported trust and iterative refinement. **(2) Data normalization:** ISO 14224-aligned BOs required substantial upfront effort but were essential for scalable, cross-site analytics. **(3) Safeguards:** Reflection, validation, and domain-scoped functions reduced unsafe execution, albeit with modest latency and token overhead. **(4) Localization:** Heterogeneous asset layouts across sites required local configuration, limiting a fully uniform deployment strategy. **(5) Human–AI collaboration:** Automation accelerated triage, yet oversight remained necessary for critical work orders. **(6) ROI and baselines:** Early gains (25–40% faster diagnostics, fewer outages) were promising, but stakeholders emphasized the need for long-term validation and comparisons with simpler baselines (e.g., ReAct, Data Interpreter (Hong et al. 2024)). **(7) Strategic queries:** Engineers requested long-horizon (5–10 year) forecasts for WO volumes, alert patterns, and replacement planning, underscoring the system's relevance for life-cycle asset management.

Conclusion

We presented *CodeReAct*, a deployed AI agent framework for automated industrial work order analysis and recommendation. Used in daily operations at the POKMAIN data center and productized in Maximo, it delivered 25–40% faster diagnostics, reduced unplanned outages, and less reliance on specialists. While demonstrated on chillers, the approach generalizes to other critical assets. Deployment experience emphasizes ISO-aligned data models, execution safeguards, and interpretable outputs. Overall, *CodeReAct* shows how AI agents can move maintenance from reactive response to predictive planning.

References

Allen, J. F. 1983. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11): 832–843.

Balali, F.; Nouri, J.; Nasiri, A.; and Zhao, T. 2020. *Data Intensive Industrial Asset Management*. Springer.

Chuang, C.; Ningyun, L.; Bin, J.; and Yin, X. 2020. Condition-based maintenance optimization for continuously monitored degrading systems under imperfect maintenance actions. *Journal of Systems Engineering and Electronics*, 31(4): 841–851.

Goyal, D.; Saini, A.; Dhami, S.; Pabla, B.; et al. 2016. Intelligent predictive maintenance of dynamic systems using condition monitoring and signal processing techniques—a review. In *2016 international conference on advances in computing, communication, & automation (ICACCA)(Spring)*, 1–6. IEEE.

Gulati, R.; and Smith, R. 2009. *Maintenance and reliability best practices*. Industrial Press Inc.

Hong, S.; Lin, Y.; Liu, B.; Liu, B.; Wu, B.; Zhang, C.; Wei, C.; Li, D.; Chen, J.; Zhang, J.; Wang, J.; Zhang, L.; Zhang, L.; Yang, M.; Zhuge, M.; Guo, T.; Zhou, T.; Tao, W.; Tang, X.; Lu, X.; Zheng, X.; Liang, X.; Fei, Y.; Cheng, Y.; Gou, Z.; Xu, Z.; and Wu, C. 2024. Data Interpreter: An LLM Agent For Data Science. arXiv:2402.18679.

Jimenez, C. E.; Yang, J.; Wettig, A.; Yao, S.; Pei, K.; Press, O.; and Narasimhan, K. 2023. Swe-bench: Can language models resolve real-world github issues?, 2024. *URL https://arxiv.org/abs/2310.06770*.

La Rosa, R.; Hulse, C.; and Liu, B. 2024. Can Github issues be solved with Tree Of Thoughts? *arXiv preprint arXiv:2405.13057*.

Langbridge, A.; O'Donncha, F.; Rayfield, J. T.; and Eck, B. 2024. Optimal Transport for Efficient, Unsupervised Anomaly Detection on Industrial Data. In *2024 IEEE International Conference on Big Data (BigData)*, 2142–2151. IEEE.

McGuire, T.; Ariker, M.; and Roggendorf, M. 2013. Making data analytics work: Three key challenges. *McKinsey & Company*.

Neuberg, L. G. 2003. Causality: models, reasoning, and inference, by judea pearl, cambridge university press, 2000. *Econometric Theory*, 19(4): 675–685.

Petsinis, P.; Naskos, A.; and Gounaris, A. 2021. Analysis of key flavors of event-driven predictive maintenance using logs of phenomena described by Weibull distributions. arXiv:2101.07033.

Rayfield, J. T.; Lin, S.; Zhou, N.; and Patel, D. C. 2025. ReAct Meets Industrial IoT: Language Agents for Data Access. In Potdar, S.; Rojas-Barahona, L.; and Montella, S., eds., *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, 364–382. Suzhou (China): Association for Computational Linguistics. ISBN 979-8-89176-333-3.

Serradilla, O.; Zugasti, E.; Rodriguez, J.; and Zurutuza, U. 2022. Deep learning models for predictive maintenance: a survey, comparison, challenges and prospects. *Applied Intelligence*, 52(10): 10934–10964.

Tao, W.; Zhou, Y.; Wang, Y.; Zhang, W.; Zhang, H.; and Cheng, Y. 2024. Magis: Llm-based multi-agent framework for github issue resolution. *arXiv preprint arXiv:2403.17927*.

Tawil, A.-R. H.; Mohamed, M.; Schmoor, X.; Vlachos, K.; and Haidar, D. 2024. Trends and challenges towards effective data-driven decision making in UK Small and Medium-sized Enterprises: Case studies and lessons learnt from the analysis of 85 Small and Medium-sized Enterprises. *Big Data and Cognitive Computing*, 8(7): 79.

Wang, X.; Chen, Y.; Yuan, L.; Zhang, Y.; Li, Y.; Peng, H.; and Ji, H. 2024. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*.

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2022. ReAct: Synergizing Reasoning and Acting in Language Models. *ArXiv*, abs/2210.03629.

Ye, C.; Hu, Z.; Deng, Y.; Huang, Z.; Ma, M. D.; Zhu, Y.; and Wang, W. 2024. MIRAI: Evaluating LLM Agents for Event Forecasting. arXiv:2407.01231.