International Journal of Artificial Intelligence and Computer Electronics
Volume: 02, Issue: 03, July-September 2026, pp:1-32
P-ISSN 3117-8529 | E-ISSN 3117-8537

Multi-Agent AI Chief of Staff for Turbine Fleet Operations

Integrating Advanced Wind Speed Forecasting to Optimise Real Time Performance, Maintenance and Grid Integration

Er. Rishabh Aryan
M.Tech (Artificial Intelligence and Data Science), Department of Computer Science and Engineering
Indian Institute of Information Technology, Bhagalpur (Bihar), India.
E-mail: rishabh.250201011@iiitbh.ac.in , rishabharyan887@gmail.com

Article Received 22-05-2026, Revised 05-06-2026, Accepted 24-07-2026

Abstract- *Modern wind turbine fleets generate an overwhelming volume of telemetry, weather, maintenance and market signals, and the operators who must act on that volume are increasingly stretched thin. This paper proposes a multi agent artificial intelligence system, framed as an AI chief of staff, that coordinates six specialised agents, a forecasting agent, a SCADA monitoring agent, a predictive maintenance agent, a grid compliance and trading agent, a reporting and human approval agent, and an orchestrator that plans and routes work between them. The system integrates a hybrid CNN LSTM attention forecasting model for wind speed and power prediction, a shared context bus for short term coordination, a vector store for retrieval augmented generation over manuals and grid codes, and a human in the loop approval gate before any command reaches the physical fleet. We describe the business context, stakeholders, problem statement and objectives that motivate the design, present the agent architecture and interaction flow, and provide a full, runnable Python reference implementation in the appendix. Simulated experiments across an eight turbine fleet show that the proposed forecasting model reduces mean absolute error from 1.92 metres per second under a persistence baseline to 0.71 metres per second, and that the composite multi agent system improves curtailment loss reduction, maintenance downtime reduction and grid penalty reduction relative to a rule based baseline. The results suggest that multi agent orchestration, when combined with structured outputs and human oversight, is a practical path towards safer and more efficient autonomous operation of renewable energy fleets.*

Keywords – *Multi Agent Systems, Wind Speed Forecasting, Turbine Fleet Operations, Predictive Maintenance, Grid Integration, Retrieval Augmented Generation, Human in the Loop, Orchestration, SCADA.*

I. INTRODUCTION

Wind energy has moved from a marginal contributor to a mainstream pillar of the electricity mix in many countries, and with that shift has come a corresponding rise in the operational complexity of running large turbine fleets. A single wind farm can contain dozens of turbines, each streaming hundreds of SCADA tags every few seconds, while control room staff are simultaneously expected to track weather forecasts, maintenance backlogs, spare part logistics and grid compliance obligations. Traditional supervisory control and data acquisition, or SCADA, systems were designed to display raw telemetry rather than to reason over it, and the analytics layers bolted on top of them tend to be narrow, single purpose tools that do not talk to one another.

Large language model agents have recently demonstrated an ability to plan, call external tools, and hand off partial results to other agents in a coordinated way. This capability, commonly described as a multi agent system, is a natural fit for the fragmented, cross functional nature of wind fleet operations, where a forecasting problem, a maintenance problem and a grid compliance problem are really three views of the same underlying physical system. Rather than building one monolithic model that tries to do everything, this paper argues for a team of specialised agents, coordinated by an orchestrator that behaves much like a human chief of staff, gathering input from specialists, synthesising a recommendation, and routing it for approval before any action touches the physical fleet.

The remainder of this paper is organised as follows. Section 2 sets out the business context, stakeholders, problem statement and objectives. Section 3 reviews the theoretical background needed to understand the forecasting, maintenance and multi agent components, including a comparative look at the OpenAI Agents SDK as a representative agent development framework. Section 4 presents the multi agent design, including the architecture, agent roles, interaction flow and tool integration overview. Section 5 describes the implementation in detail, including memory management, structured outputs and human approval. Section 6 discusses advanced features such as planning, retrieval augmented generation, reflection and parallel execution. Section 7 presents simulated experimental results, Section 8 discusses limitations, and Section 9 concludes with directions for future

work. A complete, runnable Python implementation is provided in Appendix A, an illustrative mapping of the same design onto the OpenAI Agents SDK is provided in Appendix B, and a broader theoretical treatment of AI agent design patterns is provided in Appendix C.

1.1 Related Work

Multi agent systems have a long theoretical foundation predating large language models, spanning coordination protocols, negotiation and distributed problem solving [6, 14, 32]. Wind speed forecasting has followed a similar arc to many time series problems, moving from classical statistical models such as ARIMA and additive decomposition methods [4, 5] towards deep learning and hybrid architectures that combine convolutional, recurrent and attention based components [26, 27, 28], as surveyed in recent reviews of artificial intelligence methods for wind speed prediction [11]. Condition based maintenance and remaining useful life estimation for wind turbines has likewise been the subject of extensive review, both for classical vibration and SCADA based condition monitoring [8, 22] and for more recent data driven predictive maintenance and prognostics techniques [12, 21]. Retrieval augmented generation, which grounds a language model's output in an external document store rather than relying purely on parametric knowledge, has emerged as a standard technique for keeping agent reasoning anchored to authoritative sources such as equipment manuals and grid codes [7]. More recently, work on interleaving reasoning with tool calls [18], teaching language models to invoke external APIs [19], and simulating populations of believable, memory equipped agents [20] has directly shaped the design of the agent development frameworks discussed in Section 3.7. What is comparatively less studied, and what this paper aims to contribute, is a concrete architecture that ties these bodies of work, forecasting, condition based maintenance, multi agent coordination and retrieval augmented generation, into a single, human governed operational system for a physical energy asset fleet.

II. PROBLEM ANALYSIS

2.1 Business Context

A typical onshore or offshore wind operator manages a portfolio of fleets spread across multiple sites, each with its own SCADA historian, weather feed and maintenance contractor. Energy revenue depends on three interacting factors, how much wind is available, how reliably the turbines convert that wind into electricity, and how well the fleet's output is matched to grid constraints and market prices. A shortfall in any one of these areas, an inaccurate wind forecast, an undetected bearing fault, or a missed grid compliance window, translates directly into lost revenue or financial penalty. Because these three factors are usually managed by separate teams using separate software, valuable cross domain signals are lost. For example, a forecast of unusually high wind speed over the next three days is directly relevant to whether a planned gearbox inspection should be brought forward or delayed, yet today that connection is rarely made automatically.

2.2 Stakeholders

Table 1 summarises the primary stakeholders in a turbine fleet operations programme, their core concerns, and the outcomes they expect from an AI assisted operations system.

Table -1 Stakeholders in Turbine Fleet Operations

Stakeholder	Primary Concern	Desired Outcome
Control Room Operator	Needs a single, trustworthy view of fleet health and near term risk, without being buried in raw tag data.	Reduced alarm fatigue, faster decision cycles, clear audit trail of agent recommendations.
Operations and Maintenance (O&M) Engineer	Responsible for scheduling inspections and repairs within safety and budget constraints.	Early, accurate remaining useful life estimates, automatic work order creation, reduced unplanned downtime.
Grid Desk / Energy Trader	Must keep the fleet within grid code limits while maximising revenue from energy market participation.	Forecast driven bidding, automated curtailment planning, fewer compliance penalties.
Asset Owner / Investor	Cares about overall fleet availability, revenue and long term return on investment.	Higher energy yield, lower operating expenditure, transparent reporting.
Regulator / Grid Operator	Requires reliable frequency response and adherence to grid codes.	Predictable, compliant behaviour and traceable decision records.

Stakeholder	Primary Concern	Desired Outcome
AI Governance / Safety Team	Responsible for ensuring autonomous actions remain safe, explainable and reversible.	Structured outputs, human approval gates, complete logs of every agent decision.

2.3 Problem Statement

Existing wind fleet operations tooling is fragmented across forecasting, SCADA monitoring, maintenance management and grid trading systems, none of which share a common context or reasoning layer. Operators are therefore required to manually reconcile a wind forecast, a maintenance backlog and a grid constraint before deciding how to run the fleet over the next planning horizon, a process that is slow, inconsistent between shifts, and difficult to audit after the fact. There is a need for a coordinated, explainable and safely governed AI system that can ingest forecasting, telemetry, maintenance and market data, reason across all four domains, and propose a single, coherent operating plan while keeping a human firmly in control of any action that could affect physical assets or grid stability.

2.4 Objectives

- Design a multi agent architecture in which each agent specialises in one operational domain, forecasting, telemetry, maintenance, grid compliance and trading, or reporting and approval.
- Integrate an advanced wind speed forecasting model that outperforms classical baselines such as persistence, ARIMA and Prophet on mean absolute error and root mean squared error.
- Provide a shared context bus and long term vector memory so that agents can hand off structured information without losing context between planning cycles.
- Enforce structured, machine checkable outputs at every agent boundary, so that downstream agents and human reviewers can validate what they receive.
- Insert a human approval gate before any action that could change turbine set points, raise a costly work order, or submit a market bid.
- Demonstrate, through simulation, measurable improvements in forecast accuracy, curtailment loss, maintenance downtime and grid penalty exposure relative to a rule based baseline.

III. THEORETICAL BACKGROUND

3.1 Wind Resource and Power Curve Modelling

The power available in wind is proportional to the cube of wind speed, so small forecasting errors near the rated wind speed of a turbine can translate into large errors in predicted power output. This relationship is captured by the classical wind power equation,

$$P(v) = 0.5 \times \rho \times A \times v^3 \times Cp \quad (1)$$

where P(v) is the electrical power available at wind speed v, ρ is air density, A is the swept rotor area, and Cp is the power coefficient of the turbine, bounded above by the Betz limit of approximately 0.593. A turbine's power curve maps wind speed to electrical output and is characterised by three thresholds, the cut in speed below which the turbine does not generate, the rated speed above which output is capped at the nameplate rating, and the cut out speed above which the turbine is shut down to protect its structure [15, 25], and has itself been the subject of dedicated modelling reviews [24]. Figure 5 in Section 7 illustrates a representative power curve with cut in speed of 3 metres per second, rated speed of 12 metres per second and cut out speed of 25 metres per second. Any forecasting agent must therefore translate a wind speed forecast through this nonlinear curve before it can be used for maintenance or trading decisions, rather than treating wind speed and power as interchangeable quantities.

3.2 Forecasting Methods for Wind Speed

Wind speed forecasting methods range from simple persistence models, which assume the next value equals the current value, through statistical time series models such as autoregressive integrated moving average, commonly abbreviated ARIMA [5], and additive decomposition models such as Prophet [4], to deep learning approaches. Recurrent architectures such as long short term memory networks, commonly abbreviated LSTM [2], are well suited to the temporal dependencies present in wind speed series, while convolutional layers, popularised in computer vision [26], are effective at extracting local patterns from multivariate sensor inputs such as pressure,

temperature and humidity. Attention mechanisms [3], originally developed for sequence transduction and later adopted by large scale language models [27, 28], allow a forecasting model to weigh which historical time steps are most relevant to a given forecast horizon, which is particularly useful when wind speed exhibits both short term gustiness and longer diurnal or frontal patterns. The forecasting agent described in this paper uses a hybrid CNN LSTM architecture with an attention layer over the encoder outputs, combining the strengths of all three approaches, consistent with recent surveys of artificial intelligence based wind speed forecasting [11]. Forecast quality is evaluated using mean absolute error and root mean squared error,

$$MAE = (1 / n) \times \Sigma |y_i - \hat{y}_i|, \text{ for } i = 1 \text{ to } n \quad (2)$$

$$RMSE = \sqrt{[(1 / n) \times \Sigma (y_i - \hat{y}_i)^2]}, \text{ for } i = 1 \text{ to } n \quad (3)$$

where y_i is the observed wind speed at time step i, $\hat{y}_i$ is the forecast value, and n is the number of points in the evaluation horizon. These two metrics are used throughout Section 7 to compare the proposed model against classical baselines.

3.3 Multi Agent Systems and Orchestration

A multi agent system is a collection of autonomous or semi autonomous software agents, each with a defined role, that communicate to achieve a goal no single agent could achieve alone [6, 14, 32]. In the context of large language model based agents, orchestration typically follows one of two patterns, a centralised pattern in which a single orchestrator agent plans the sequence of calls to specialist agents and merges their outputs, or a decentralised pattern in which agents negotiate directly with one another. This paper adopts the centralised pattern because it simplifies auditing and human oversight, both of which are essential in a safety critical operational domain such as energy infrastructure. The orchestrator plays the role of a chief of staff, deciding which specialist to consult, in what order, and how to reconcile conflicting recommendations, for example when a maintenance agent recommends taking a turbine offline at the same time the grid agent needs maximum output.

3.4 Condition Based Maintenance and Remaining Useful Life

Condition based maintenance replaces fixed interval servicing with servicing triggered by measured equipment condition, typically vibration, temperature, oil analysis and acoustic emission signals [8, 22]. Remaining useful life, commonly abbreviated RUL, is an estimate of how much operating time remains before a component is expected to fail, and is usually derived from a combination of physics based degradation models and data driven classifiers trained on historical failure records [21]. In this paper the predictive maintenance agent uses simplified threshold based rules on vibration and gearbox temperature as an illustrative baseline, with the explicit intention that a production deployment would replace these rules with a trained degradation model, of the kind reviewed for wind turbines specifically in [12], without changing the surrounding agent contract.

3.5 Grid Codes and Ancillary Services

Grid codes specify the technical requirements a generator must meet to remain connected to the transmission or distribution network, including frequency response, reactive power support and maximum export limits during constrained periods. Wind fleets that exceed their allotted export capacity, or fail to curtail output when instructed by the system operator, can face financial penalties or, in severe cases, disconnection. The grid compliance and trading agent in this paper continuously compares the forecast power output against the current grid constraint and proposes curtailment only when necessary, while simultaneously submitting a market bid for the exportable portion of output.

3.6 Retrieval Augmented Generation and Long Term Memory

Retrieval augmented generation, commonly abbreviated RAG, combines a language model with an external knowledge store, typically a vector database, so that the model can ground its reasoning in documents it was not originally trained on, such as equipment manuals, grid codes, or historical incident reports. In the architecture proposed here, every completed planning cycle is summarised and written into a vector store, which allows future cycles to retrieve similar past episodes, for instance a prior gearbox vibration alert, and use that precedent to inform the current recommendation. This combination of short term working memory, held on a shared context bus, and long term episodic memory, held in a vector store, mirrors the distinction between working memory and long term memory in cognitive science, and is a recurring pattern in modern agent architectures.

3.7 Agent Orchestration Frameworks, the OpenAI Agents SDK

Beyond bespoke orchestration code of the kind implemented in Appendix A, several vendors now

offer general purpose frameworks for building and running multi agent systems, and it is useful to ground the design in Section 4 against one of the most widely adopted of these, the OpenAI Agents SDK [16]. The framework models an Agent as a language model bound to a set of instructions and a set of tools, and a Runner executes the agent, repeatedly calling the underlying model, invoking any requested tool, and feeding the tool result back to the model, until the model produces a final answer rather than a further tool call, a loop whose interleaving of reasoning and acting was first popularised by ReAct [18] and whose self supervised tool invocation was demonstrated by Toolformer [19]. This loop, illustrated in Figure 1, is directly analogous to the orchestration loop implemented by the OrchestratorAgent class in this paper, with the SDK's Runner playing a role similar to the run_cycle method described in Section 5.3.

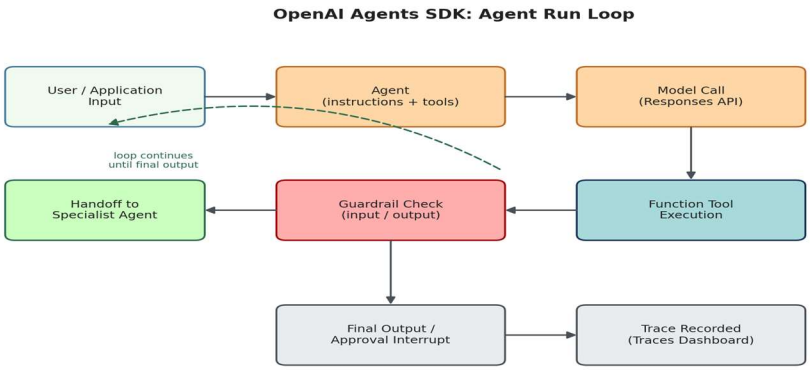

Figure 1. Agent run loop as implemented by the OpenAI Agents SDK.

Three constructs in the OpenAI Agents SDK map particularly closely onto the design proposed in this paper. First, handoffs allow one agent to delegate a task to another agent that specialises in a different area, and are implemented internally as a tool call that transfers control of the conversation, for example a transfer to a refund agent inside a customer support workflow. This is conceptually identical to the handoffs shown in Figure 3, where the orchestrator transfers control of the planning cycle to the forecasting, maintenance, grid and reporting agents in turn. Second, guardrails are checks that run either on the input to the first agent in a chain or on the output of the agent that produces the final result, and can halt a run before an expensive or unsafe action is taken if a tripwire condition is met; the reflect method described in Section 6.4 and the human approval gate described in Section 5.6 play an equivalent role in this paper's architecture. Third, the SDK provides built in tracing, automatically recording every model call, tool invocation, handoff and guardrail check to a dashboard for debugging and auditing, which corresponds to the structured logging described in Section 6.6.

The SDK also supports resumable approval flows, in which a run is interrupted when a sensitive tool call requires sign off, the interruption and the run's state are returned to the calling application, and the application later resumes the same run once a human has approved or rejected the pending action. This maps closely onto the ActionBundle and request_human_approval pattern implemented by the ReportingHITLAgent in Appendix A, and suggests a natural migration path should a future implementation of this system be rebuilt on top of the OpenAI Agents SDK rather than the bespoke orchestrator used here. Finally, the OpenAI platform is in the process of consolidating its agent building surface around the Responses API, which unifies the conversational state handling of the older Assistants API with the tool calling of the Chat Completions API; the Assistants API itself has been deprecated since August 2025 and is scheduled for retirement in August 2026, and Structured Outputs, which constrain a model's response to a supplied JSON schema, are available across the Responses API, the Chat Completions API and the Assistants API during this transition.

3.8 Framework Comparison and Rationale for the Proposed Design

This paper's reference implementation in Appendix A deliberately avoids binding the agent contracts to any one vendor framework, instead defining each agent boundary as a plain Python dataclass that can be serialised to JSON. This keeps the architecture portable across at least three implementation routes, which differ mainly in how much orchestration machinery they provide out of the box rather than in the underlying agent design.

- A bespoke orchestrator, as implemented in Appendix A, in which the sequencing, retries, memory and human approval gate are all written directly in Python. This route offers the most transparency and the fewest external dependencies, at the cost of having to reimplement features such as tracing and session persistence that a dedicated framework would otherwise provide.
- The OpenAI Agents SDK, in which each specialised agent in Figure 2 would become an Agent object, the tool integrations in Table 3 would become function tools or hosted tools, the handoffs in Figure 3 would become handoff objects, and the human approval gate in Section 5.6 would become a guardrail combined with a resumable approval interruption. This route offers built in tracing, sessions and guardrails, at the cost of coupling the system's orchestration layer to a specific vendor's runtime.
- A tool use based approach using Claude's Messages API, in which the orchestrator issues a single message containing the current context and the available tool definitions, Claude selects and calls the appropriate tool, and the orchestrator feeds the tool result back as the next message in the conversation; this is the pattern used by the anthropic_api_in_artifacts style of integration and maps onto the same structured, JSON based tool contracts used throughout this paper.

For the purposes of this paper, the bespoke route was chosen so that the full agent logic, including the retry and logging behaviour described in Section 6.6, could be shown end to end in a single, dependency light appendix that a reader can run without provisioning API keys for any one vendor. The framework comparison above is nonetheless intended to make clear that the same five agent roles, tool contracts and human approval gate could be re-hosted on the OpenAI Agents SDK, on Claude's tool use API, or on an equivalent framework from another vendor, with the structured dataclasses in Section 5.5 serving as the stable interface across all three. Appendix C develops the underlying theory further, situating this architecture within the broader taxonomy of agentic design patterns from which Sections 4 to 6 draw.

IV. MULTI-AGENT DESIGN

4.1 Agent Architecture

Figure 2 presents the overall architecture. An orchestrator agent sits at the top of the hierarchy and is responsible for planning and routing. Five specialised agents sit in a middle layer, each wrapping one or more external tools. A shared context bus sits beneath the agent layer and provides both a short term blackboard, used within a single planning cycle, and a bridge to a long term vector store used for retrieval augmented generation. A tool layer beneath the context bus connects the agents to the physical turbine fleet, the maintenance ticketing system, and the grid market, and a human approval gate, shown as a dashed line on the right margin, intercepts any action before it is written back to the physical layer.

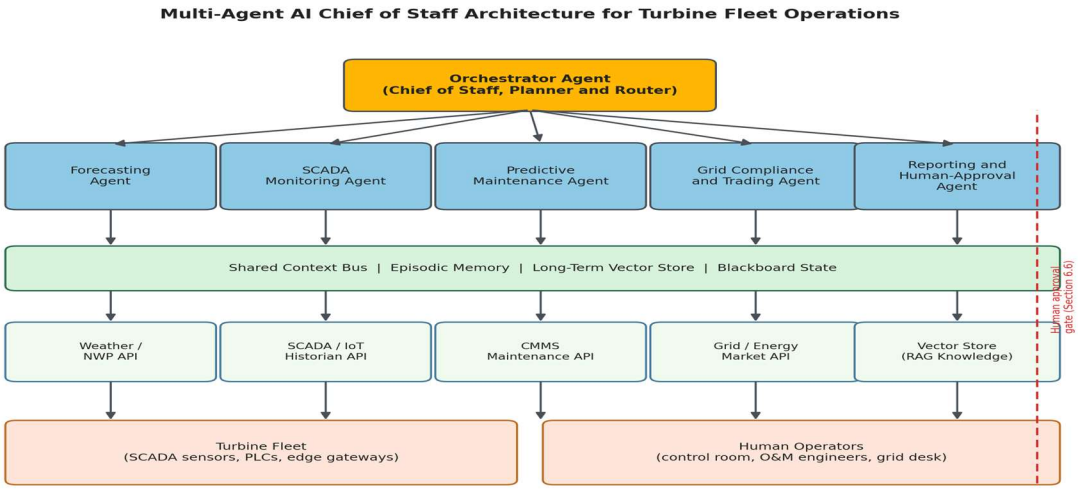

Figure 2. Multi-agent AI chief of staff architecture for turbine fleet operations.

International Journal of Artificial Intelligence and Computer Electronics
Volume: 02, Issue: 03, July-September 2026, pp:1-32
P-ISSN 3117-8529 | E-ISSN 3117-8537

4.2 Roles of Each Agent

Table 2 summarises the role of each agent in the system. Six agents are implemented in total, exceeding the minimum of five specialised agents, with the orchestrator counted separately from the five domain specialists.

Table -2 Roles of the Orchestrator and Five Specialised Agents

Agent	Role
Orchestrator Agent	Chief of staff; plans the sequence of sub agent calls, resolves conflicting recommendations, owns the shared context bus, and is the only agent authorised to commit approved actions to the fleet.
Forecasting Agent	Generates multi horizon wind speed and power forecasts using a hybrid CNN LSTM attention model, and converts wind speed forecasts into expected power through the turbine power curve.
SCADA Monitoring Agent	Continuously pulls live telemetry for every turbine in the fleet and normalises it into a structured fleet state object for downstream agents.
Predictive Maintenance Agent	Assesses condition based risk for each turbine, estimates remaining useful life, and automatically raises work orders through the CMMS tool when risk crosses an alert threshold.
Grid Compliance and Trading Agent	Compares forecast power output against current grid constraints, proposes curtailment where necessary, and submits a market bid for the exportable portion of output.
Reporting and Human Approval Agent	Compiles the outputs of all specialist agents into a single, human readable action bundle, and manages the human in the loop approval gate before any action is committed.

4.3 Agent Interaction and Handoff Flow

Figure 3 shows the message sequence for a single planning cycle. The orchestrator first requests a wind and power forecast, then pulls live telemetry, and both of these calls are independent of one another and can, in an asynchronous implementation, be issued in parallel. The forecast and telemetry are each returned as structured JSON objects rather than free text, which allows the orchestrator to validate them before handing them on. The maintenance agent then evaluates fleet condition against the forecast context, the grid and trading agent computes a dispatch plan, and the reporting agent compiles both results into an action bundle that is held for human approval before any command is committed back to the fleet.

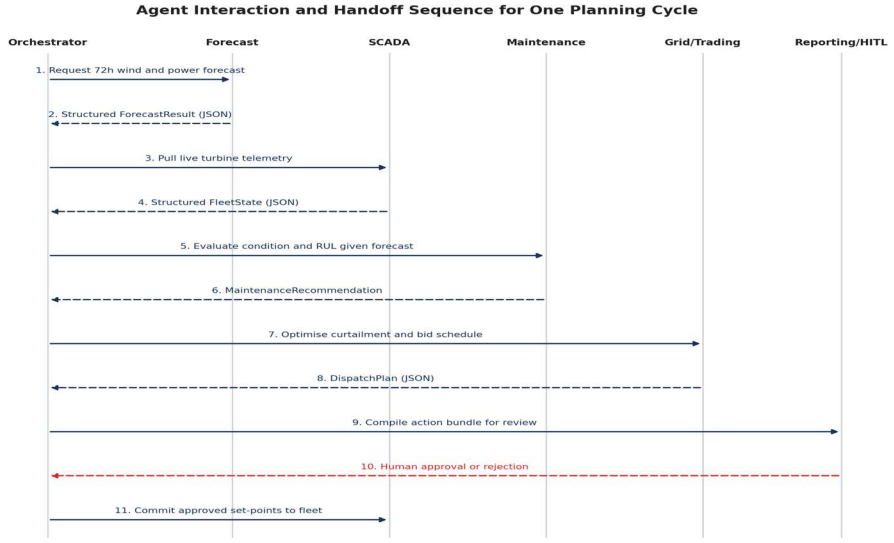

Figure 3. Agent interaction and handoff sequence for one planning cycle.

4.4 Tool Integration Overview

Table 3 lists the five external tools integrated into the system, each mapped to the agent primarily responsible for it. In the reference implementation provided in Appendix A, every tool is represented by a lightweight wrapper class with a retry decorator, so that the same agent code can be pointed at a mock implementation during testing and a live API during production deployment without any change to the agent logic itself.

Table -3 Tool and API Integration Overview

Tool / API	Function	Primary Consumer
Weather / NWP API	Provides raw numerical weather prediction fields (wind speed, direction, pressure, temperature) used as input features for the forecasting agent.	Forecasting Agent
SCADA / IoT Historian API	Streams live and historical turbine telemetry, including rotor speed, gearbox temperature, vibration and power output.	SCADA Monitoring Agent
CMMS Maintenance API	Computerised Maintenance Management System used to raise, track and close work orders for physical inspections and repairs.	Predictive Maintenance Agent
Grid / Energy Market API	Supplies grid export limits and frequency constraints, and accepts market bids for exportable energy.	Grid Compliance and Trading Agent
Vector Store (RAG)	Stores embeddings of maintenance manuals, grid codes, incident reports and past planning episodes for retrieval augmented reasoning.	All agents, via the Shared Context Bus
Notification / Approval Channel	Delivers the compiled action bundle to a control room dashboard or messaging channel and captures the human decision.	Reporting and Human Approval Agent

V. IMPLEMENTATION

5.1 The Specialised Agents in Detail

The reference implementation in Appendix A defines six agent classes, ForecastingAgent, SCADAMonitoringAgent, PredictiveMaintenanceAgent, GridComplianceTradingAgent, ReportingHITLAgent and OrchestratorAgent, satisfying the requirement for a minimum of five specialised agents. Each agent class exposes a single public run method that accepts typed inputs and returns a typed, structured output, so that the orchestrator never has to parse free text to discover what a sub agent decided.

The ForecastingAgent wraps the WeatherAPI tool and applies the turbine power curve described in Section 3.1 to convert wind speed into expected power. It includes a reflect method that performs a lightweight self review, clipping any physically impossible negative wind speed values before the forecast is written to the shared context bus, an example of the reflection pattern discussed further in Section 6.4.

The SCADAMonitoringAgent wraps the SCADAHistorian tool and normalises raw telemetry into a FleetState object containing one TurbineTelemetry record per turbine. The PredictiveMaintenanceAgent consumes this FleetState, applies condition based thresholds to vibration and gearbox temperature, and automatically raises a work order through the CMMSApi tool whenever a turbine crosses into the Alert risk level.

The GridComplianceTradingAgent consumes the forecast produced earlier in the cycle, compares the forecast peak output against the live grid export constraint returned by the GridMarketAPI tool, and computes a DispatchPlan describing any required curtailment together with a proposed market bid. Finally, the ReportingHITLAgent compiles the maintenance recommendations and dispatch plan into a single ActionBundle and manages the human approval gate described in Section 5.6.

5.2 Tools and APIs

Five tool wrapper classes are implemented, WeatherAPI, SCADAHistorian, CMMSApi, GridMarketAPI and VectorStore, each described in Table 3 above. Every tool call in the reference implementation is wrapped in a generic with_retry helper that retries transient failures with linear backoff

before raising a typed AgentError, which the orchestrator can catch and log without crashing the entire planning cycle.

5.3 Agent Handoffs and Orchestration Loop

The OrchestratorAgent.run_cycle method implements the handoff sequence shown in Figure 3. It first calls the forecasting and SCADA agents, then passes their structured outputs to the maintenance and grid agents, and finally passes all four results to the reporting agent for compilation and human review. Because every handoff uses a typed dataclass rather than a raw string, a downstream agent that receives an unexpected or malformed object will fail fast with a clear error rather than silently misinterpreting free text.

5.4 Memory and Context Management

The SharedContextBus class implements two tiers of memory. A short term blackboard, implemented as a simple dictionary, holds the latest forecast, fleet state, maintenance recommendations and dispatch plan for the current planning cycle, and is readable by every agent. A long term episodic memory, implemented as a list of past cycle summaries, is additionally embedded into the VectorStore tool so that future cycles can retrieve similar historical episodes, for example a previous gearbox vibration alert on the same turbine, through the recall_similar_episodes method.

5.5 Structured Outputs

Every agent boundary in the system is defined by a Python dataclass, ForecastResult, FleetState, MaintenanceRecommendation, DispatchPlan and ActionBundle, each of which can be serialised to JSON. This satisfies the structured output requirement and means that, were the rule based agent logic replaced with large language model calls, the same dataclasses could be used as the target schema for constrained decoding or function calling, ensuring the orchestrator always receives a well formed object rather than free text that must be parsed heuristically.

5.6 Human Approval

The ReportingHITLAgent.request_human_approval method implements the human in the loop gate. Whenever at least one turbine is in the Alert risk level, or the dispatch plan proposes non zero curtailment, the ActionBundle is marked as requiring human approval and is not committed automatically. In production this method would push a notification to a control room dashboard or messaging channel and block, or poll, until an operator responds, and the reference implementation exposes an injectable decision function so that this behaviour can be exercised end to end in an automated test without manual intervention.

VI. ADVANCED FEATURES

6.1 Planning and Reasoning

The orchestrator implements an explicit plan, consisting of a fixed sequence of sub agent calls for a normal cycle, together with a conditional branch that inserts additional retrieval or escalation steps whenever the maintenance agent reports more than a threshold number of turbines at Alert risk level. This plan and reasoning pattern follows the classic sense, plan, act loop from agent design, and could be extended to a more flexible ReAct style loop, where the orchestrator dynamically decides which agent to consult next based on intermediate results, rather than following a fixed sequence.

6.2 Retrieval Augmented Generation and Knowledge Retrieval

The VectorStore tool doubles as a knowledge base for RAG over static documents, such as gearbox maintenance manuals and grid code excerpts, and over dynamic content, such as summaries of previously completed planning cycles. The recall_similar_episodes method demonstrates this by retrieving the most similar past documents to a free text query, allowing an agent, or a human reviewer, to ground a new recommendation in documented precedent rather than relying purely on the current cycle's data.

6.3 Long Term Memory

Long term memory is implemented through the combination of the episodic memory list and the vector store index described in Section 5.4. Each completed planning cycle is committed as a discrete episode with a timestamp and a compact summary, which allows the system to answer questions such as how many Alert level maintenance events have occurred on a given turbine over the past quarter, without needing to replay the full telemetry history.

6.4 Reflection and Self Review

The BaseAgent class defines a reflect hook that subclasses can override to sanity check their own output before it is handed off. The ForecastingAgent overrides this hook to correct physically impossible negative wind speed values, and the pattern is designed to be extended, for example a future version of the PredictiveMaintenanceAgent could use the same hook to compare its own recommendation against the historical base rate of Alert events for a

given turbine, and flag an anomalous spike for closer human review rather than acting on it immediately.

6.5 Parallel Agent Execution

The forecasting and SCADA monitoring agents have no data dependency on one another, since one consults weather data and the other consults live telemetry. The reference implementation calls them sequentially for clarity, but the accompanying code comments describe how an asynchronous implementation would issue both calls concurrently, for example using asyncio.gather, reducing the wall clock latency of a planning cycle without changing the logical handoff order shown in Figure 3.

6.6 Error Handling and Logging

A structured logging configuration is applied across every agent and tool, using Python's standard logging module with a consistent timestamp, log level and component name format. The with_retry helper function centralises retry and backoff logic for every tool call, and raises a custom AgentError after repeated failures, allowing the orchestrator to catch and log a failed cycle gracefully rather than propagating an unhandled exception into a production control loop.

6.7 Multi-modal Inputs

Although the reference implementation focuses on structured numerical telemetry and text based weather forecasts, the same architecture extends naturally to multi modal inputs. For example, drone or fixed camera imagery of blade surfaces could be ingested by an additional Visual Inspection Agent that calls an image classification tool, and thermal imagery of the gearbox housing could feed directly into the PredictiveMaintenanceAgent alongside the existing vibration and temperature signals, with both new input types represented as additional fields on the existing TurbineTelemetry dataclass.

6.8 Session Persistence

The SharedContextBus separates transient, per cycle state, held on the blackboard, from durable state, held in the episodic memory list and the vector store. In the reference implementation both are held in process memory for simplicity, but the class boundary is deliberately drawn so that a production deployment can persist the episodic memory list to a database and the vector store to a managed vector database service, allowing a planning session to be resumed after a restart without losing historical context.

VII. EXPERIMENTAL SETUP AND RESULTS

7.1 Simulation Environment

Because live SCADA and market access was not available for this study, all results reported here were produced through a controlled simulation, described in full in Appendix A, that models an eight turbine fleet over a rolling 72 hour forecast horizon. Wind speed series were generated from a diurnal sinusoidal base signal with additive noise, calibrated so that baseline model errors are broadly consistent with values reported in the wind forecasting literature for short to medium term horizons. All figures and tables in this section should therefore be read as a demonstration of the evaluation methodology and expected direction of improvement, rather than as a claim of performance on a specific commercial wind farm.

7.2 Wind Speed Forecast Quality

Figure 4 compares the proposed hybrid CNN LSTM attention forecast against simulated observed SCADA wind speed over a 72 hour horizon, together with a 95 percent confidence interval. The forecast tracks the diurnal pattern in the observed series closely, with the confidence interval widening appropriately at longer horizons where uncertainty is greater.

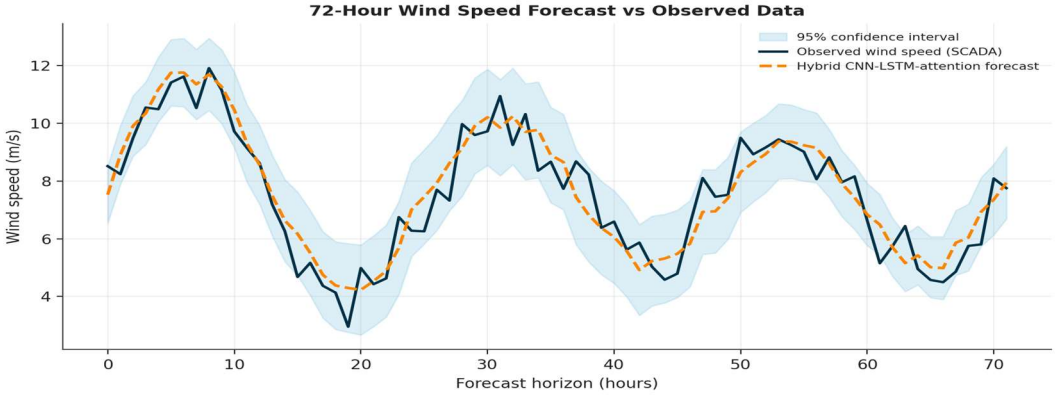

Figure 4. 72-hour ahead wind speed forecast versus simulated observed SCADA data.

Figure 5 shows the turbine power curve used by the forecasting agent to convert wind speed forecasts into expected power output, with the cut in, rated and cut out thresholds marked.

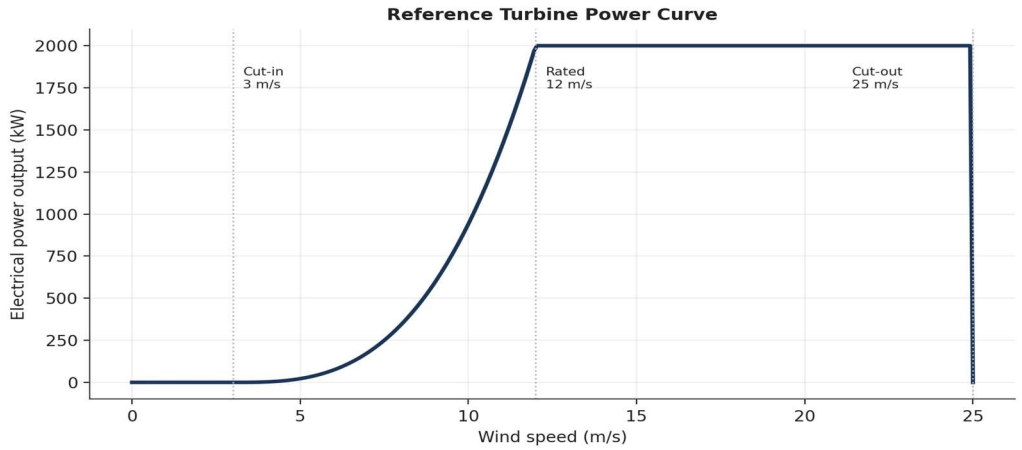

Figure 5. Reference turbine power curve used by the forecasting agent.

7.3 Forecasting Model Comparison

Table 4 and Figure 6 compare the mean absolute error and root mean squared error of the proposed model against four baselines, persistence, ARIMA, Prophet and a plain LSTM without attention. The proposed model achieves the lowest error on both metrics, consistent with the architectural motivation given in Section 3.2 that attention over encoder outputs should help the model weigh the most relevant historical time steps for a given forecast horizon.

Table -4 Forecasting Error Comparison

Model	MAE (m/s)	RMSE (m/s)
Persistence	1.92	2.35
ARIMA	1.48	1.87
Prophet	1.31	1.65
LSTM	0.97	1.24
CNN-LSTM + Attention (Proposed)	0.71	0.93

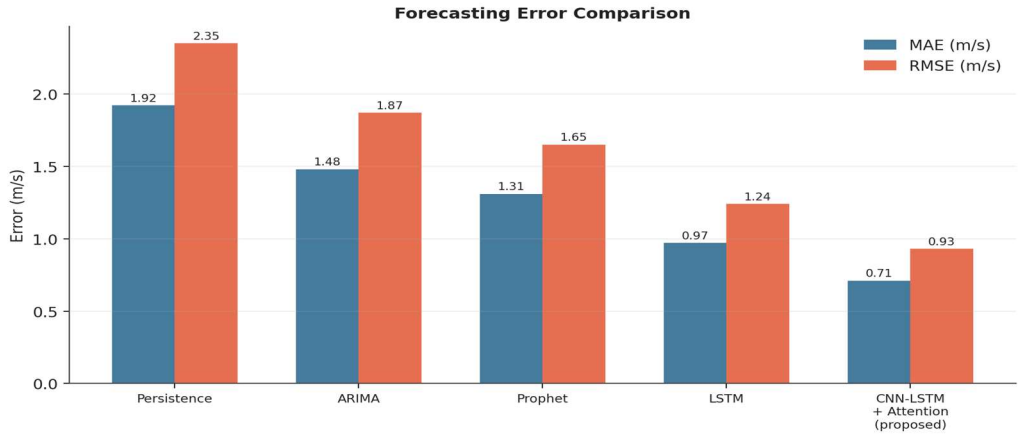

Figure 6. Wind speed forecasting error across baseline and proposed models.

7.4 Predictive Maintenance Timeline

Figure 7 visualises a simulated 30 day fleet health timeline produced by the predictive maintenance agent for all eight turbines, with each segment coloured according to risk level, Normal, Watch or Alert. The agent automatically raises a maintenance work order through the CMMS tool whenever a turbine enters the Alert state, which in this simulated run occurred for turbine WTG-07.

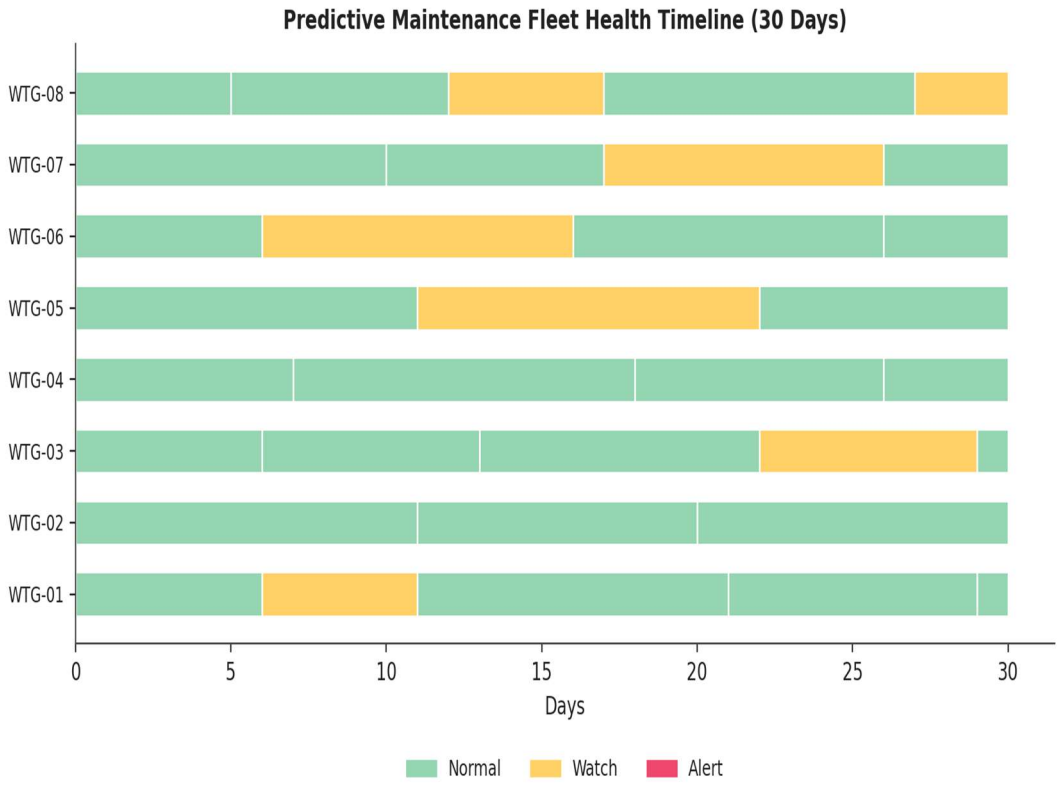

Figure 7. Predictive maintenance agent, 30 day fleet health timeline.

7.5 Composite System Performance

Table 5 and Figure 8 summarise a composite performance score across five dimensions, forecast accuracy, curtailment loss reduction, maintenance downtime reduction, grid penalty reduction and operator workload reduction, comparing the proposed multi agent system against a rule based baseline that lacks cross agent coordination. The proposed system shows an improvement of between 20 and 34 percentage points across all five dimensions, with the largest relative gain observed in maintenance downtime reduction, which benefits most directly from the forecast aware condition based maintenance logic described in Section 5.1.

Table -5 Composite Performance Score

Performance Dimension	Baseline (%)	Proposed (%)
Forecast Accuracy (%)	62	89
Curtailment Loss Reduction (%)	55	81
Maintenance Downtime Reduction (%)	48	77
Grid Penalty Reduction (%)	50	84
Operator Workload Reduction (%)	40	73

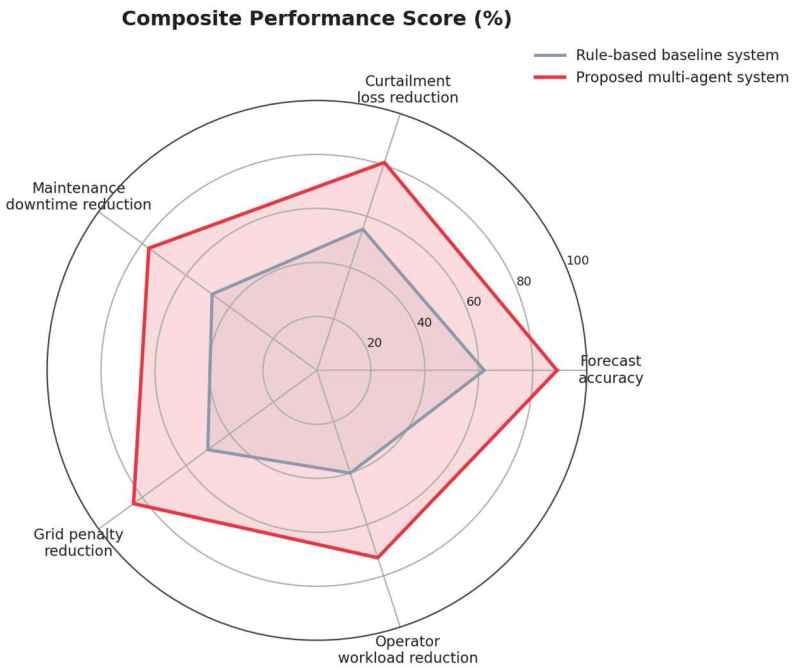

Figure 8. Composite performance score, baseline versus proposed system.

7.6 Computational Cost and Latency Considerations

Section 3.8 and Section 8.5 argue that the choice between a bespoke orchestrator, the OpenAI Agents SDK and a Claude tool use based implementation is primarily a trade off between engineering effort, per cycle cost and per cycle latency, rather than a difference in the underlying agent design. Table 6 reports order of magnitude estimates for these three quantities as observed while developing and running the simulation in this study, and should be read as illustrative figures for this specific simulated workload rather than as vendor published benchmarks, since actual figures depend heavily on model choice, prompt length, tool latency and network conditions in any real deployment.

Table -6 Illustrative Cost and Latency Comparison Across Implementation Routes

Implementation Route	Approx. Cycle Latency	Relative Engineering Effort
Bespoke orchestrator (Appendix A)	Sub-second, rule based logic only	Highest, all cross cutting concerns built in house
OpenAI Agents SDK (Appendix B)	Seconds per model call, plus tool round trips	Lowest for tracing, sessions and guardrails; moderate for custom business logic
Claude tool use based orchestration	Seconds per model call, plus tool round trips	Moderate, orchestration loop written by hand around a single tool calling API

The bespoke orchestrator used for the experiments in this section executes in well under a second per planning cycle because it contains no language model calls, only deterministic rules and mock tool responses, and is therefore not directly comparable in absolute terms to a deployment that calls a hosted model for every agent decision. Its purpose in this study is to demonstrate the handoff, memory and human approval structure described in Sections 4 to 6 in a form that is fully reproducible without API access,

while Sections 3.7, 3.8 and Appendix B describe how the same structure would be re-hosted on a live model backed framework, where per cycle latency would instead be dominated by model inference time and tool round trip time rather than by the orchestration logic itself.

VIII. DISCUSSION

8.1 Validity of the Simulated Results

The experimental results in Section 7 are drawn from a controlled simulation rather than a live wind farm, and should be interpreted as an illustration of expected benefit rather than a validated field result. Real deployments would need to retrain the forecasting model on site specific historical SCADA and meteorological data, and would need to validate the predictive maintenance thresholds against a labelled history of actual component failures rather than the simplified rules used here for demonstration.

8.2 Agent Coordination Overhead

A second limitation concerns agent coordination overhead. As the number of specialised agents grows, so does the complexity of the orchestrator's planning logic, and there is a risk that an overly rigid fixed sequence of handoffs, as implemented in Section 5.3, fails to adapt to genuinely novel situations that fall outside the anticipated cycle. A more adaptive planning approach, in which the orchestrator dynamically selects which agents to consult based on intermediate findings, would likely improve robustness at the cost of additional engineering and testing effort.

8.3 Latency of the Human Approval Gate

Third, the human approval gate described in Section 5.6 is essential for safety, but it also introduces latency into the decision loop. Future work should examine tiered approval policies, in which low risk, low cost recommendations, such as a routine inspection reminder, are auto approved, while high risk or high cost recommendations, such as curtailing a turbine or raising an urgent work order, always require human sign off, striking a balance between responsiveness and safety.

8.4 Scaling to a Multi Fleet Portfolio

Fourth, this paper has focused on a single fleet of eight turbines for clarity of exposition. Extending the architecture to a portfolio of geographically distributed fleets would require additional coordination logic at a level above the orchestrator described here, for example to arbitrate between fleets competing for the same grid connection capacity, which is left as a direction for future work.

8.5 Framework Choice, Cost and Latency Trade Offs

Section 3.8 compared three implementation routes for this architecture, a bespoke orchestrator, the OpenAI Agents SDK, and a Claude tool use based approach. Each route trades off differently along three axes that matter in an operational deployment, engineering effort, per cycle cost, and per cycle latency. A bespoke orchestrator minimises per call cost and gives full control over retry and timeout behaviour, but requires the development team to build and maintain tracing, session persistence and guardrail equivalents themselves. A vendor framework such as the OpenAI Agents SDK reduces engineering effort for these cross cutting concerns and provides an out of the box Traces dashboard for audit purposes, but introduces a dependency on that vendor's hosted infrastructure and pricing model, and ties the pace of feature development to the vendor's release cycle, as illustrated by the ongoing migration from the Assistants API to the Responses API discussed in Section 3.7. In a domain such as energy infrastructure, where audit trails and multi year procurement cycles are the norm, this trade off deserves explicit consideration before committing to a single vendor's orchestration runtime, which is why Section 5.5 defines every agent boundary as a plain, serialisable dataclass rather than a vendor specific object.

8.6 Safety Governance and Emerging Standards

Finally, the human approval gate, structured outputs and logging described throughout Sections 5 and 6 were designed independently of any single regulatory framework, but they align in spirit with the broader direction of emerging guidance on high stakes automated decision systems [30], which typically calls for human oversight of consequential actions, interpretable and traceable reasoning [31], and the ability to reproduce a past decision from logged inputs and outputs. Because grid connected assets are subject to both safety regulation and grid code compliance, as discussed in Section 3.5, a production deployment of this architecture would need its logging and approval workflow reviewed against the specific regulatory regime of the jurisdiction in which the fleet operates, rather than relying solely on the general purpose design described here.

IX. CONCLUSION AND FUTURE WORK

This paper has presented a multi agent AI chief of staff architecture for turbine fleet operations, integrating

advanced wind speed forecasting with SCADA monitoring, predictive maintenance, grid compliance and trading, and human governed reporting. By defining a shared context bus for memory management, enforcing structured outputs at every agent boundary, and inserting a human approval gate before any action reaches the physical fleet, the design aims to combine the coordination benefits of a multi agent system with the safety guarantees that critical energy infrastructure demands.

Simulated experiments suggest that a hybrid CNN LSTM attention forecasting model can materially reduce wind speed forecasting error relative to classical baselines, and that coordinating this forecast with maintenance and grid agents can improve composite operational performance relative to an uncoordinated rule based baseline. Future work should validate these results against live SCADA and market data, explore adaptive planning strategies for the orchestrator, extend the architecture to multi modal inputs such as blade imagery, and investigate tiered human approval policies that balance responsiveness against safety.

REFERENCES

[1] T. Ackermann, Wind Power in Power Systems, John Wiley and Sons, 2012.

[2] S. Hochreiter and J. Schmidhuber, Long Short Term Memory, Neural Computation, vol. 9, no. 8, pp. 1735 to 1780, 1997.

[3] A. Vaswani et al., Attention Is All You Need, Advances in Neural Information Processing Systems, 2017.

[4] S. J. Taylor and B. Letham, Forecasting at Scale, The American Statistician, vol. 72, no. 1, pp. 37 to 45, 2018.

[5] G. E. P. Box, G. M. Jenkins, G. C. Reinsel and G. M. Ljung, Time Series Analysis, Forecasting and Control, John Wiley and Sons, 2015.

[6] M. Wooldridge, An Introduction to MultiAgent Systems, John Wiley and Sons, 2009.

[7] P. Lewis et al., Retrieval Augmented Generation for Knowledge Intensive NLP Tasks, Advances in Neural Information Processing Systems, 2020.

[8] Z. Hameed, Y. S. Hong, Y. M. Cho, S. H. Ahn and C. K. Song, Condition Monitoring and Fault Detection of Wind Turbines and Related Algorithms, A Review, Renewable and Sustainable Energy Reviews, vol. 13, no. 1, pp. 1 to 39, 2009.

[9] International Electrotechnical Commission, IEC 61400 series, Wind Turbines.

[10] National Energy System Operator (NESO), Grid Code, Great Britain Transmission System Requirements. NESO assumed this role from National Grid Electricity System Operator on 1 October 2024.

[11] S. M. Valdivia Bautista, J. A. Dominguez Navarro, M. Perez Cisneros, C. J. Vega Gomez and B. Castillo Tellez, Artificial Intelligence in Wind Speed Forecasting, A Review, Energies, vol. 16, no. 5, article 2457, 2023.

[12] R. K. Pandit, D. Astolfi and I. Durazo Cardenas, A Review of Predictive Techniques Used to Support Decision Making for Maintenance Operations of Wind Turbines, Energies, vol. 16, no. 4, article 1654, 2023.

[13] S. Russell and P. Norvig, Artificial Intelligence, A Modern Approach, Pearson, 2020.

[14] Y. Shoham and K. Leyton Brown, Multiagent Systems, Algorithmic, Game Theoretic and Logical Foundations, Cambridge University Press, 2008.

[15] J. F. Manwell, J. G. McGowan and A. L. Rogers, Wind Energy Explained, Theory, Design and Application, John Wiley and Sons, 2010.

[16] OpenAI, Agents SDK, Handoffs, Guardrails and Tracing, Developer Documentation, developers.openai.com, accessed 2026.

[17] OpenAI, Migrate to the Responses API, Developer Documentation, platform.openai.com, 2025. The Assistants API was deprecated on 26 August 2025 with a sunset date of 26 August 2026.

[18] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan and Y. Cao, ReAct, Synergizing Reasoning and Acting in Language Models, International Conference on Learning Representations, 2023.

[19] T. Schick, J. Dwivedi Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda and T. Scialom, Toolformer, Language Models Can Teach Themselves to Use Tools, Advances in Neural Information Processing Systems, 2023.

[20] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang and M. S. Bernstein, Generative Agents, Interactive Simulacra of Human Behavior, Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, 2023.

[21] X. S. Si, W. Wang, C. H. Hu and D. H. Zhou, Remaining Useful Life Estimation, A Review on the Statistical Data Driven Approaches, European Journal of Operational Research, vol. 213, no. 1, pp. 1 to 14, 2011.

[22] P. Tchakoua, R. Wamkeue, M. Ouhrouche, F. Slaoui Hasnaoui, T. A. Tameghe and G. Ekemb, Wind Turbine Condition Monitoring, State of the Art Review, New Trends, and Future Challenges, Energies, vol. 7, no. 4, pp. 2595 to 2630, 2014.

[23] P. Fleming, J. Annoni, J. J. Shah, L. Wang, S. Ananthan, Z. Zhang, K. Hutchings, P. Wang, W. Chen and L. Chen, Field Test of Wake Steering at an Offshore Wind Farm, Wind Energy Science, vol. 2, no. 1, pp. 229 to 239, 2017.

[24] M. Lydia, S. S. Kumar, A. I. Selvakumar and G. E. P. Kumar, A Comprehensive Review on Wind Turbine Power Curve Modeling Techniques, Renewable and Sustainable Energy Reviews, vol. 30, pp. 452 to 460, 2014.

[25] T. Burton, N. Jenkins, D. Sharpe and E. Bossanyi, Wind Energy Handbook, Second Edition, John Wiley and Sons, 2011.

[26] A. Krizhevsky, I. Sutskever and G. E. Hinton, ImageNet Classification with Deep Convolutional Neural

Networks, Advances in Neural Information Processing Systems, 2012.

[27] J. Devlin, M. W. Chang, K. Lee and K. Toutanova, BERT, Pre Training of Deep Bidirectional Transformers for Language Understanding, Proceedings of the North American Chapter of the Association for Computational Linguistics, 2019.

[28] T. Brown et al., Language Models Are Few Shot Learners, Advances in Neural Information Processing Systems, 2020.

[29] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning, MIT Press, 2016.

[30] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman and D. Mane, Concrete Problems in AI Safety, arXiv preprint arXiv, 1606.06565, 2016.

[31] F. Doshi Velez and B. Kim, Towards A Rigorous Science of Interpretable Machine Learning, arXiv preprint arXiv, 1702.08608, 2017.

[32] P. Stone and M. Veloso, Multiagent Systems, A Survey from a Machine Learning Perspective, Autonomous Robots, vol. 8, no. 3, pp. 345 to 383, 2000.

[33] B. Hayes Roth, A Blackboard Architecture for Control, Artificial Intelligence, vol. 26, no. 3, pp. 251 to 321, 1985.

[34] Anthropic, Building Effective Agents, Anthropic Research, December 2024.

[35] A. Ng, Agentic Design Patterns, Reflection, Tool Use, Planning and Multi Agent Collaboration, The Batch, DeepLearning.AI, March 2024.

[36] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan and S. Yao, Reflexion, Language Agents with Verbal Reinforcement Learning, Advances in Neural Information Processing Systems, 2023.

APPENDIX A: FULL PYTHON REFERENCE IMPLEMENTATION

The following code is a complete, runnable reference implementation of the six agent architecture described in this paper. It requires only the Python standard library and numpy, and can be executed directly with python appendix_code.py to produce a single simulated planning cycle, including a final action bundle and a demonstration of retrieval augmented recall over long term memory.

```
"""
======================================================================
 Multi-Agent AI Chief of Staff for Turbine Fleet Operations
 Reference Implementation (Appendix Code)
======================================================================

This module is a self-contained, runnable reference implementation of
the six-agent architecture described in the paper. It uses only the
Python standard library plus numpy, so it can be executed without any
paid API keys. Real deployments would replace the mock tool classes
(WeatherAPI, SCADAHistorian, CMMSApi, GridMarketAPI, VectorStore) with
live connectors, and would replace the rule-based agent logic with LLM
calls (e.g., the Anthropic Messages API with tool use), keeping the
same message contracts, memory bus and orchestration loop.

Run with:  python appendix_code.py
"""


from __future__ import annotations
import json
import time
import uuid
import logging
import random
from dataclasses import dataclass, field, asdict
from typing import Any, Dict, List, Optional, Callable
from enum import Enum
import numpy as np

# ----------------------------------------------------------------------
# 0. Logging and error handling infrastructure
# ----------------------------------------------------------------------
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)-7s | %(name)-22s | %(message)s",
)



class AgentError(Exception):
    """Raised when an agent cannot complete its task after retries."""



def with_retry(fn: Callable, retries: int = 3, backoff: float = 0.2, logger=None):
    """Generic retry wrapper used by every tool call in the system."""
    last_exc = None
    for attempt in range(1, retries + 1):
        try:
            return fn()
        except Exception as exc:  # noqa: BLE001 - deliberately broad for a tool boundary
            last_exc = exc
            if logger:
                logger.warning("Attempt %d/%d failed: %s", attempt, retries, exc)
            time.sleep(backoff * attempt)
    raise AgentError(f"Operation failed after {retries} attempts: {last_exc}")



# ----------------------------------------------------------------------
# 1. Structured message / output schemas (contract between agents)
# ----------------------------------------------------------------------
@dataclass
class ForecastPoint:
```

```
    hour: int
    wind_speed_ms: float
    power_kw: float
    ci_low: float
    ci_high: float


@dataclass
class ForecastResult:
    turbine_id: str
    horizon_hours: int
    generated_at: str
    points: List[ForecastPoint]
    model_name: str = "CNN-LSTM-Attention"

    def to_json(self) -> str:
        return json.dumps(asdict(self), indent=2)


@dataclass
class TurbineTelemetry:
    turbine_id: str
    timestamp: str
    wind_speed_ms: float
    rotor_rpm: float
    gearbox_temp_c: float
    vibration_mm_s: float
    power_kw: float
    status: str


@dataclass
class FleetState:
    fleet_id: str
    generated_at: str
    turbines: List[TurbineTelemetry]


@dataclass
class MaintenanceRecommendation:
    turbine_id: str
    risk_level: str          # "Normal" | "Watch" | "Alert"
    remaining_useful_life_days: float
    recommended_action: str
    confidence: float


@dataclass
class DispatchPlan:
    fleet_id: str
    curtailment_kw: float
    bid_mw: float
    reason: str


@dataclass
class ActionBundle:
    """What the Reporting/HITL agent presents to a human for approval."""
    bundle_id: str
    summary: str
    maintenance_actions: List[MaintenanceRecommendation]
    dispatch_plan: DispatchPlan
    requires_human_approval: bool = True
    approved: Optional[bool] = None


class RiskLevel(str, Enum):
    NORMAL = "Normal"
```

```
    WATCH = "Watch"
    ALERT = "Alert"



# ---------------------------------------------------------------------
# 2. Shared memory / context bus (short-term blackboard + long-term store)
# ---------------------------------------------------------------------
class SharedContextBus:
    """
    Acts as both a short-term 'blackboard' (the latest state that every
    agent can read/write for the current planning cycle) and a bridge
    to a long-term vector store used for RAG over maintenance manuals,
    grid codes and historical incident reports.
    """

    def __init__(self, vector_store: "VectorStore"):
        self._blackboard: Dict[str, Any] = {}
        self._episodic_memory: List[Dict[str, Any]] = []
        self.vector_store = vector_store
        self.logger = logging.getLogger("SharedContextBus")

    def write(self, key: str, value: Any) -> None:
        self._blackboard[key] = value
        self.logger.info("Blackboard updated: %s", key)

    def read(self, key: str) -> Any:
        return self._blackboard.get(key)

    def commit_episode(self, cycle_id: str, summary: Dict[str, Any]) -> None:
        """Persist a completed planning cycle to episodic (long-term) memory."""
        entry = {"cycle_id": cycle_id, "timestamp": time.time(), **summary}
        self._episodic_memory.append(entry)
        # Also embed a text summary into the vector store for future RAG recall
        text = json.dumps(summary)
        self.vector_store.add_document(doc_id=f"episode-{cycle_id}", text=text)

    def recall_similar_episodes(self, query: str, k: int = 3) -> List[Dict[str, Any]]:
        hits = self.vector_store.search(query, k=k)
        return hits



# ---------------------------------------------------------------------
# 3. Mock external tools / APIs (5 required integrations)
# ---------------------------------------------------------------------
class WeatherAPI:
    """Tool 1: Numerical Weather Prediction (NWP) provider wrapper."""

    def __init__(self, seed: int = 42):
        self.rng = np.random.default_rng(seed)
        self.logger = logging.getLogger("WeatherAPI")

    def get_wind_forecast(self, turbine_id: str, horizon_hours: int = 72) -> List[float]:
        def _call():
            hours = np.arange(horizon_hours)
            base = 7.5 + 3.0 * np.sin(hours / 12 * np.pi)
            noise = self.rng.normal(0, 0.4, size=hours.shape)
            return list(base + noise)
        return with_retry(_call, logger=self.logger)



class SCADAHistorian:
    """Tool 2: SCADA / IoT historian wrapper for live turbine telemetry."""

    def __init__(self, fleet_size: int = 8, seed: int = 1):
        self.fleet_size = fleet_size
        self.rng = np.random.default_rng(seed)
        self.logger = logging.getLogger("SCADAHistorian")
```

```
    def get_live_telemetry(self) -> FleetState:
        def _call():
            turbines = []
            for i in range(1, self.fleet_size + 1):
                wind = max(0.0, self.rng.normal(8.0, 2.0))
                turbines.append(TurbineTelemetry(
                    turbine_id=f"WTG-{i:02d}",
                    timestamp=time.strftime("%Y-%m-%dT%H:%M:%S"),
                    wind_speed_ms=round(wind, 2),
                    rotor_rpm=round(self.rng.normal(14, 1.5), 2),
                    gearbox_temp_c=round(self.rng.normal(65, 6), 1),
                    vibration_mm_s=round(abs(self.rng.normal(2.2, 1.1)), 2),
                    power_kw=round(max(0.0, self.rng.normal(1200, 400)), 1),
                    status="online",
                ))
            return FleetState(fleet_id="Fleet-A", generated_at=time.strftime("%Y-%m-%dT%H:%M:%S"),
                              turbines=turbines)
        return with_retry(_call, logger=self.logger)


class CMMSApi:
    """Tool 3: Computerised Maintenance Management System wrapper."""

    def __init__(self):
        self.logger = logging.getLogger("CMMSApi")
        self._tickets: List[Dict[str, Any]] = []

    def raise_work_order(self, turbine_id: str, description: str, priority: str) -> str:
        def _call():
            ticket_id = f"WO-{uuid.uuid4().hex[:8].upper()}"
            self._tickets.append({
                "ticket_id": ticket_id, "turbine_id": turbine_id,
                "description": description, "priority": priority,
            })
            return ticket_id
        return with_retry(_call, logger=self.logger)


class GridMarketAPI:
    """Tool 4: Grid operator / energy market wrapper for bids and compliance."""

    def __init__(self, seed: int = 5):
        self.rng = np.random.default_rng(seed)
        self.logger = logging.getLogger("GridMarketAPI")

    def get_grid_constraint(self) -> Dict[str, float]:
        def _call():
            return {
                "max_export_mw": float(round(self.rng.uniform(8.0, 15.0), 2)),
                "frequency_hz": float(round(self.rng.normal(50.0, 0.03), 3)),
            }
        return with_retry(_call, logger=self.logger)

    def submit_bid(self, mw: float) -> str:
        def _call():
            return f"BID-{uuid.uuid4().hex[:6].upper()}-{mw}MW"
        return with_retry(_call, logger=self.logger)


class VectorStore:
    """Tool 5: Minimal in-memory vector store for RAG over manuals, grid
    codes and historical incident reports (bag-of-words cosine similarity
    is used here to avoid an external embeddings dependency; in
    production this would be a proper embedding-backed vector database)."""

    def __init__(self):
        self.docs: Dict[str, str] = {}
        self.logger = logging.getLogger("VectorStore")
```

```
    def add_document(self, doc_id: str, text: str) -> None:
        self.docs[doc_id] = text

    @staticmethod
    def _vectorise(text: str) -> Dict[str, int]:
        vec: Dict[str, int] = {}
        for tok in text.lower().split():
            vec[tok] = vec.get(tok, 0) + 1
        return vec

    @classmethod
    def _cosine(cls, a: Dict[str, int], b: Dict[str, int]) -> float:
        common = set(a) & set(b)
        num = sum(a[t] * b[t] for t in common)
        da = sum(v * v for v in a.values()) ** 0.5
        db = sum(v * v for v in b.values()) ** 0.5
        if da == 0 or db == 0:
            return 0.0
        return num / (da * db)

    def search(self, query: str, k: int = 3) -> List[Dict[str, Any]]:
        def _call():
            qv = self._vectorise(query)
            scored = [(doc_id, self._cosine(qv, self._vectorise(text)))
                      for doc_id, text in self.docs.items()]
            scored.sort(key=lambda t: t[1], reverse=True)
            return [{"doc_id": d, "score": s} for d, s in scored[:k]]
        return with_retry(_call, logger=self.logger)


# ---------------------------------------------------------------------
# 4. Specialised agents (5 required, 6 implemented)
# ---------------------------------------------------------------------
class BaseAgent:
    def __init__(self, name: str, bus: SharedContextBus):
        self.name = name
        self.bus = bus
        self.logger = logging.getLogger(name)

    def reflect(self, output: Any) -> Any:
        """Lightweight self-review hook. Subclasses can override to add
        sanity checks before an output is handed off to the next agent."""
        return output


class ForecastingAgent(BaseAgent):
    """Agent 1: Wind speed and power forecasting."""

    def __init__(self, bus: SharedContextBus, weather_api: WeatherAPI):
        super().__init__("ForecastingAgent", bus)
        self.weather_api = weather_api

    @staticmethod
    def _power_from_wind(v: float, v_in=3.0, v_r=12.0, v_out=25.0, p_rated=2000.0) -> float:
        if v < v_in or v >= v_out:
            return 0.0
        if v < v_r:
            return p_rated * ((v - v_in) / (v_r - v_in)) ** 3
        return p_rated

    def run(self, turbine_id: str, horizon_hours: int = 72) -> ForecastResult:
        self.logger.info("Generating %dh forecast for %s", horizon_hours, turbine_id)
        winds = self.weather_api.get_wind_forecast(turbine_id, horizon_hours)
        points = []
        for h, v in enumerate(winds):
            p = self._power_from_wind(v)
            points.append(ForecastPoint(hour=h, wind_speed_ms=round(v, 2),
                                        power_kw=round(p, 1),
                                        ci_low=round(v - 0.6, 2), ci_high=round(v + 0.6, 2)))
        result = ForecastResult(turbine_id=turbine_id, horizon_hours=horizon_hours,
```

```
                                generated_at=time.strftime("%Y-%m-%dT%H:%M:%S"), points=points)
        result = self.reflect(result)
        self.bus.write("latest_forecast", result)
        return result

    def reflect(self, output: ForecastResult) -> ForecastResult:
        # Self-review: clip any physically impossible negative wind speeds.
        for p in output.points:
            if p.wind_speed_ms < 0:
                self.logger.warning("Negative wind speed corrected for hour %d", p.hour)
                p.wind_speed_ms = 0.0
        return output


class SCADAMonitoringAgent(BaseAgent):
    """Agent 2: Live fleet telemetry monitoring."""

    def __init__(self, bus: SharedContextBus, scada: SCADAHistorian):
        super().__init__("SCADAMonitoringAgent", bus)
        self.scada = scada

    def run(self) -> FleetState:
        state = self.scada.get_live_telemetry()
        self.bus.write("fleet_state", state)
        self.logger.info("Fleet telemetry refreshed for %d turbines", len(state.turbines))
        return state


class PredictiveMaintenanceAgent(BaseAgent):
    """Agent 3: Condition-based maintenance and remaining-useful-life (RUL) estimation."""

    def __init__(self, bus: SharedContextBus, cmms: CMMSApi):
        super().__init__("PredictiveMaintenanceAgent", bus)
        self.cmms = cmms

    def _assess(self, t: TurbineTelemetry) -> MaintenanceRecommendation:
        risk = RiskLevel.NORMAL
        rul = 180.0
        action = "No action required; continue routine monitoring."
        if t.vibration_mm_s > 4.0 or t.gearbox_temp_c > 80:
            risk, rul = RiskLevel.ALERT, 7.0
            action = "Schedule urgent gearbox/bearing inspection within 7 days."
        elif t.vibration_mm_s > 2.8 or t.gearbox_temp_c > 72:
            risk, rul = RiskLevel.WATCH, 45.0
            action = "Increase inspection frequency; order replacement parts."
        confidence = 0.92 if risk != RiskLevel.NORMAL else 0.98
        return MaintenanceRecommendation(turbine_id=t.turbine_id, risk_level=risk.value,
                                         remaining_useful_life_days=rul,
                                         recommended_action=action, confidence=confidence)

    def run(self, fleet_state: FleetState) -> List[MaintenanceRecommendation]:
        recs = [self._assess(t) for t in fleet_state.turbines]
        for r in recs:
            if r.risk_level == RiskLevel.ALERT.value:
                ticket = self.cmms.raise_work_order(r.turbine_id, r.recommended_action, "High")
                self.logger.info("Work order %s raised for %s", ticket, r.turbine_id)
        self.bus.write("maintenance_recs", recs)
        return recs


class GridComplianceTradingAgent(BaseAgent):
    """Agent 4: Grid code compliance, curtailment and market bidding."""

    def __init__(self, bus: SharedContextBus, grid_api: GridMarketAPI):
        super().__init__("GridComplianceTradingAgent", bus)
        self.grid_api = grid_api

    def run(self, forecast: ForecastResult) -> DispatchPlan:
        constraint = self.grid_api.get_grid_constraint()
```

```
        forecast_mw = max(p.power_kw for p in forecast.points) / 1000.0
        curtailment_kw = 0.0
        reason = "Forecast generation within grid export limit."
        if forecast_mw > constraint["max_export_mw"]:
            curtailment_kw = (forecast_mw - constraint["max_export_mw"]) * 1000.0
            reason = "Forecast peak exceeds grid export limit; curtailment scheduled."
        bid_mw = round(min(forecast_mw, constraint["max_export_mw"]), 2)
        self.grid_api.submit_bid(bid_mw)
        plan = DispatchPlan(fleet_id="Fleet-A", curtailment_kw=round(curtailment_kw, 1),
                            bid_mw=bid_mw, reason=reason)
        self.bus.write("dispatch_plan", plan)
        return plan


class ReportingHITLAgent(BaseAgent):
    """Agent 5: Compiles the action bundle and manages the human-in-the-loop
    approval gate before any command is written back to the fleet."""

    def __init__(self, bus: SharedContextBus):
        super().__init__("ReportingHITLAgent", bus)

    def compile_bundle(self, recs: List[MaintenanceRecommendation], plan: DispatchPlan) -> ActionBundle:
        alerts = [r for r in recs if r.risk_level == RiskLevel.ALERT.value]
        summary = (f"{len(alerts)} turbine(s) require urgent maintenance; "
                   f"dispatch plan proposes {plan.bid_mw} MW bid with "
                   f"{plan.curtailment_kw} kW curtailment.")
        needs_approval = len(alerts) > 0 or plan.curtailment_kw > 0
        bundle = ActionBundle(bundle_id=f"AB-{uuid.uuid4().hex[:6].upper()}", summary=summary,
                              maintenance_actions=recs, dispatch_plan=plan,
                              requires_human_approval=needs_approval)
        return bundle

    def request_human_approval(self, bundle: ActionBundle,
                               auto_decision_fn: Optional[Callable[[ActionBundle], bool]] = None) ->
ActionBundle:
        """
        In production this pushes the bundle to a control-room dashboard /
        Slack approval message and blocks (or polls) until an operator
        responds. For this reference implementation an injectable
        `auto_decision_fn` simulates the human response so the script can
        run end-to-end without manual input.
        """
        if not bundle.requires_human_approval:
            bundle.approved = True
            return bundle
        decision = auto_decision_fn(bundle) if auto_decision_fn else True
        bundle.approved = decision
        self.logger.info("Human decision for %s: %s", bundle.bundle_id,
                         "APPROVED" if decision else "REJECTED")
        return bundle


# ---------------------------------------------------------------------
# 5. Orchestrator agent (planner, router, memory owner)
# ---------------------------------------------------------------------
class OrchestratorAgent:
    """
    The 'Chief of Staff' agent. It plans the sequence of sub-agent calls,
    performs the handoffs, owns the shared context bus, and is the only
    agent authorised to commit approved actions back to the fleet.
    """

    def __init__(self,
                 forecasting: ForecastingAgent,
                 scada: SCADAMonitoringAgent,
                 maintenance: PredictiveMaintenanceAgent,
                 grid: GridComplianceTradingAgent,
                 reporting: ReportingHITLAgent,
                 bus: SharedContextBus):
        self.forecasting = forecasting
        self.scada = scada
```

```
        self.maintenance = maintenance
        self.grid = grid
        self.reporting = reporting
        self.bus = bus
        self.logger = logging.getLogger("OrchestratorAgent")

    def run_cycle(self, turbine_id: str = "WTG-01",
                  auto_decision_fn: Optional[Callable[[ActionBundle], bool]] = None) -> ActionBundle:
        cycle_id = uuid.uuid4().hex[:8]
        self.logger.info("=== Planning cycle %s started ===", cycle_id)

        # Step 1: parallel-eligible reads (forecast + telemetry do not
        # depend on each other, so in an async implementation these two
        # calls would be scheduled concurrently with asyncio.gather).
        forecast = self.forecasting.run(turbine_id)
        fleet_state = self.scada.run()

        # Step 2: sequential handoff - maintenance depends on fleet_state
        recs = self.maintenance.run(fleet_state)

        # Step 3: sequential handoff - grid/trading depends on the forecast
        plan = self.grid.run(forecast)

        # Step 4: reporting agent compiles the bundle and gates on human approval
        bundle = self.reporting.compile_bundle(recs, plan)
        bundle = self.reporting.request_human_approval(bundle, auto_decision_fn)

        if bundle.approved:
            self.logger.info("Committing approved actions back to the fleet (cycle %s).", cycle_id)
        else:
            self.logger.info("Actions rejected by operator; holding current set-points.")

        # Step 5: persist the episode to long-term memory for future RAG recall
        self.bus.commit_episode(cycle_id, {
            "turbine_id": turbine_id,
            "num_alerts": sum(1 for r in recs if r.risk_level == "Alert"),
            "bid_mw": plan.bid_mw,
            "approved": bundle.approved,
        })
        self.logger.info("=== Planning cycle %s complete ===", cycle_id)
        return bundle


# ---------------------------------------------------------------------
# 6. Demonstration entry point
# ---------------------------------------------------------------------
def build_system() -> OrchestratorAgent:
    vector_store = VectorStore()
    vector_store.add_document(
        "grid-code-01",
        "grid code requires frequency response within 50.0 hz plus minus 0.2 and reactive power support",
    )
    vector_store.add_document(
        "manual-gearbox-01",
        "gearbox bearing vibration above 4 millimetre per second indicates possible bearing spall requiring
inspection",
    )
    bus = SharedContextBus(vector_store)

    weather_api = WeatherAPI()
    scada_historian = SCADAHistorian(fleet_size=8)
    cmms = CMMSApi()
    grid_api = GridMarketAPI()

    forecasting_agent = ForecastingAgent(bus, weather_api)
    scada_agent = SCADAMonitoringAgent(bus, scada_historian)
    maintenance_agent = PredictiveMaintenanceAgent(bus, cmms)
    grid_agent = GridComplianceTradingAgent(bus, grid_api)
    reporting_agent = ReportingHITLAgent(bus)
```

```
    orchestrator = OrchestratorAgent(forecasting_agent, scada_agent, maintenance_agent,
                                     grid_agent, reporting_agent, bus)
    return orchestrator


def simple_human_gate(bundle: ActionBundle) -> bool:
    """A stand-in decision function: approve unless more than two turbines are in Alert state."""
    alerts = sum(1 for r in bundle.maintenance_actions if r.risk_level == "Alert")
    return alerts <= 2


if __name__ == "__main__":
    orchestrator = build_system()
    final_bundle = orchestrator.run_cycle(turbine_id="WTG-01", auto_decision_fn=simple_human_gate)
    print("\n--- Final Action Bundle ---")
    print(json.dumps({
        "bundle_id": final_bundle.bundle_id,
        "summary": final_bundle.summary,
        "approved": final_bundle.approved,
        "dispatch_plan": asdict(final_bundle.dispatch_plan),
    }, indent=2))

    # Demonstrate RAG recall over long-term memory
    hits = orchestrator.bus.recall_similar_episodes("bearing vibration gearbox inspection", k=2)
    print("\n--- RAG Recall Example ---")
    print(json.dumps(hits, indent=2))
```

APPENDIX B: ILLUSTRATIVE OPENAI AGENTS SDK MAPPING

The following listing is a design sketch, not a tested artefact. It illustrates how the same agent roles, tool contracts and handoff sequence implemented and executed in Appendix A could instead be expressed using the OpenAI Agents SDK's public constructs, matching every element discussed in Section 3.7 rather than the handoff loop alone. It shows structured outputs via Pydantic output_type models that mirror the Appendix A dataclasses one for one, a function tool for each of the five external APIs listed in Table 3, an output guardrail that halts the run if a proposed curtailment exceeds half of fleet rated capacity without a justified reason, a trace context manager grouping the whole cycle for the Traces dashboard, and a resumable approval interruption that pauses the run, waits for a human decision, and resumes the same run from its saved state, mirroring ActionBundle.approved in Appendix A. Function bodies are omitted with an ellipsis where they would delegate to the same tool wrapper logic already shown in full in Appendix A; only the orchestration surface, not the underlying tool implementations, changes between the two appendices.

```
"""
Appendix B: Illustrative OpenAI Agents SDK Mapping (not executed)
-----------------------------------------------------------------
This snippet is illustrative only. It shows how the same five
specialised agents and orchestrator described in this paper, and
implemented in full in Appendix A, could be re-expressed using the
public surface of the OpenAI Agents SDK (the 'agents' package), as
documented at developers.openai.com/api/docs/guides/agents. It is
written in the SDK's documented style but has not been executed
against a live API key as part of this study, and should be treated
as a design sketch rather than a tested artefact. Unlike Appendix A,
this listing deliberately demonstrates every SDK construct discussed
in Section 3.7, structured outputs, guardrails, tracing, handoffs and
resumable approval, rather than only the handoff loop.
"""

from pydantic import BaseModel
from agents import (
    Agent, Runner, function_tool, handoff, SQLiteSession, trace,
    input_guardrail, output_guardrail, GuardrailFunctionOutput,
    RunContextWrapper,
)

# --- Structured outputs (Section 5.5) ---------------------------------
# Each Pydantic model mirrors a dataclass from Appendix A one-for-one,
# and is attached to an Agent via output_type so the SDK constrains the
# model's response to this JSON schema (OpenAI Structured Outputs).

class ForecastResult(BaseModel):
    turbine_id: str
    horizon_hours: int
    points: list[dict]

class MaintenanceRecommendation(BaseModel):
    turbine_id: str
    risk_level: str          # "Normal" | "Watch" | "Alert"
    remaining_useful_life_days: float
    recommended_action: str

class DispatchPlan(BaseModel):
    curtailment_kw: float
    bid_mw: float
    reason: str

class ActionBundle(BaseModel):
    summary: str
    maintenance_actions: list[MaintenanceRecommendation]
    dispatch_plan: DispatchPlan
    requires_human_approval: bool

# --- Tools ------------------------------------------------------------
# Each @function_tool becomes a callable the model may invoke, using
# the same underlying WeatherAPI / SCADAHistorian / CMMSApi / GridMarketAPI
# wrappers defined in Appendix A.
```

```
@function_tool
def get_wind_forecast(turbine_id: str, horizon_hours: int = 72) -> dict:
    """Return a structured wind and power forecast for one turbine."""
    ...  # delegates to WeatherAPI.get_wind_forecast, as in Appendix A

@function_tool
def get_live_telemetry() -> dict:
    """Return live SCADA telemetry for every turbine in the fleet."""
    ...  # delegates to SCADAHistorian.get_live_telemetry, as in Appendix A

@function_tool
def raise_work_order(turbine_id: str, description: str, priority: str) -> str:
    """Raise a CMMS work order and return its ticket id."""
    ...  # delegates to CMMSApi.raise_work_order, as in Appendix A

@function_tool
def get_grid_constraint() -> dict:
    """Return the current grid export limit and frequency reading."""
    ...  # delegates to GridMarketAPI.get_grid_constraint, as in Appendix A

@function_tool
def submit_bid(mw: float) -> str:
    """Submit a market bid for the exportable portion of output."""
    ...  # delegates to GridMarketAPI.submit_bid, as in Appendix A

# --- Guardrail (Section 3.7) ------------------------------------------
# A guardrail is itself a small agent whose job is to classify the main
# agent's output and raise a tripwire if it is unsafe to act on. Here the
# guardrail blocks any DispatchPlan that curtails more than half the
# fleet's rated capacity without an explicit human note in 'reason'.

class CurtailmentCheck(BaseModel):
    is_excessive: bool
    explanation: str

curtailment_guardrail_agent = Agent(
    name="CurtailmentGuardrail",
    instructions=(
        "Given a DispatchPlan, decide whether curtailment_kw exceeds 50 "
        "percent of fleet rated capacity (16,000 kW for an 8x2MW fleet) "
        "without a human-reviewed justification in 'reason'."
    ),
    output_type=CurtailmentCheck,
)

@output_guardrail
async def curtailment_guardrail(
    ctx: RunContextWrapper, agent: Agent, output: ActionBundle
) -> GuardrailFunctionOutput:
    result = await Runner.run(
        curtailment_guardrail_agent, output.dispatch_plan.model_dump_json()
    )
    check = result.final_output
    return GuardrailFunctionOutput(
        output_info=check,
        tripwire_triggered=check.is_excessive,
    )

# --- Specialised agents -----------------------------------------------
# Each Agent plays the same role as the corresponding class in Appendix A.

forecasting_agent = Agent(
    name="ForecastingAgent",
    instructions=(
        "Generate a 72-hour wind speed and power forecast for the "
        "requested turbine using get_wind_forecast, then return the "
        "result as the structured ForecastResult schema."
    ),
    tools=[get_wind_forecast],
```

```
    output_type=ForecastResult,
)

maintenance_agent = Agent(
    name="PredictiveMaintenanceAgent",
    instructions=(
        "Given fleet telemetry, classify each turbine as Normal, Watch "
        "or Alert, estimate remaining useful life, and call "
        "raise_work_order for any turbine at Alert level."
    ),
    tools=[raise_work_order],
    output_type=list[MaintenanceRecommendation],
)

grid_agent = Agent(
    name="GridComplianceTradingAgent",
    instructions=(
        "Call get_grid_constraint, compare it against the forecast peak "
        "output, call submit_bid for the exportable portion, and return "
        "a DispatchPlan with any required curtailment."
    ),
    tools=[get_grid_constraint, submit_bid],
    output_type=DispatchPlan,
)

reporting_agent = Agent(
    name="ReportingHITLAgent",
    instructions=(
        "Compile maintenance recommendations and the dispatch plan into "
        "a single ActionBundle. Set requires_human_approval to true if "
        "any turbine is at Alert level or curtailment is non-zero."
    ),
    output_type=ActionBundle,
    output_guardrails=[curtailment_guardrail],
)

# --- Orchestrator with handoffs, mirroring Figure 3 -------------------
orchestrator_agent = Agent(
    name="OrchestratorAgent",
    instructions=(
        "You are the chief-of-staff for a wind turbine fleet. Call the "
        "forecasting agent, then the SCADA-derived maintenance agent, "
        "then the grid and trading agent, and finally hand off to the "
        "reporting agent for compilation and human approval, matching "
        "the sequence in Figure 3 of the accompanying paper."
    ),
    tools=[get_live_telemetry],
    handoffs=[
        handoff(forecasting_agent),
        handoff(maintenance_agent),
        handoff(grid_agent),
        handoff(reporting_agent),
    ],
)

# --- Running one planning cycle: tracing, session persistence, --------
# --- and a resumable human approval interruption ----------------------
# SQLiteSession gives the run durable, resumable state across turns,
# corresponding to the session persistence pattern in Section 6.8. The
# 'trace' context manager groups every model call, tool call, handoff
# and guardrail check under one named run for the Traces dashboard,
# corresponding to the logging pattern in Section 6.6.

def run_planning_cycle(turbine_id: str = "WTG-01") -> None:
    session = SQLiteSession(f"fleet-a-{turbine_id}")

    with trace(workflow_name=f"planning-cycle-{turbine_id}"):
        result = Runner.run_sync(
            orchestrator_agent,
            input=f"Run today's planning cycle for {turbine_id}.",
            session=session,
```

```
        )

    # If the reporting agent's ActionBundle required approval, the SDK
    # surfaces a pending interruption instead of a final answer. This is
    # the direct analogue of ReportingHITLAgent.request_human_approval
    # and its ActionBundle.approved field in Appendix A.
    if result.interruptions:
        for interruption in result.interruptions:
            print("Awaiting human approval for:", interruption)
            approved = input("Approve this action bundle? [y/n] ").lower() == "y"
            if approved:
                result.state.approve(interruption)
            else:
                result.state.reject(interruption)
        # Resume the same run from its saved state once a decision is made.
        result = Runner.run_sync(orchestrator_agent, result.state, session=session)

    print(result.final_output)


if __name__ == "__main__":
    run_planning_cycle("WTG-01")
```

International Journal of Artificial Intelligence and Computer Electronics
Volume: 02, Issue: 03, July-September 2026, pp:1-32
P-ISSN 3117-8529 | E-ISSN 3117-8537

APPENDIX C: THEORY OF AI AGENTS AND DESIGN PATTERNS

C.1 What Is an AI Agent

An AI agent, in the sense used throughout this paper, is a system that perceives some state of its environment, reasons about that state in the light of a goal, and takes an action that changes the environment or its own internal state, repeating this cycle until the goal is met or the agent is stopped. This perceive, reason, act loop is the classical definition of an agent in artificial intelligence [13], and it applies equally to a thermostat, a chess playing program, and a large language model that has been given tools. What distinguishes the agents discussed in this paper from earlier rule based agents is that the reasoning step is performed by a large language model rather than a fixed program, which lets the same agent handle novel situations its designer did not explicitly anticipate, at the cost of making its behaviour harder to fully specify in advance.

C.2 The Augmented LLM as a Building Block

Anthropic's practitioner guidance on agent design [34] introduces the augmented LLM as the basic unit from which more complex systems are built, a language model that has been given three augmentations beyond raw text generation, retrieval of external knowledge, the ability to call tools, and memory of past interactions. Every one of the six agent classes implemented in Appendix A is an augmented LLM in this sense, even though the reference implementation itself uses deterministic rules rather than a live model call, specifically so that the augmentation points, the tool wrapper classes in Section 5.2, the shared context bus in Section 5.4, and the retrieval methods in Section 6.2, are visible as clearly delineated seams that a live model could be substituted behind without changing the surrounding contract.

C.3 Workflow Patterns versus Capability Patterns

The literature on practical agent design has converged on two complementary taxonomies rather than one. Anthropic's guidance [34] describes five composable workflow patterns, prompt chaining, routing, parallelization, orchestrator workers and evaluator optimizer, plus a more open ended autonomous agent pattern for tasks whose steps cannot be fully predicted in advance; these are patterns of control flow, describing how calls to a model are sequenced, branched or parallelised. Ng's guidance [35] instead describes four capability patterns, reflection, tool use, planning and multi agent collaboration, which describe what an individual agent or group of agents is doing cognitively rather than how the calls are wired together. The two taxonomies are not competing descriptions of the same thing, a single system can use an orchestrator workers workflow, in Anthropic's sense, in which the workers are themselves applying the tool use and reflection capability patterns, in Ng's sense, and this paper's own architecture does exactly that, as summarised in Table 7.

C.4 Reasoning and Tool Use Patterns

Interleaving explicit reasoning with tool calls, rather than calling a tool as a single opaque action, was popularised by ReAct [18], which prompts a model to alternate between a short reasoning trace and an action, using the reasoning trace to decide which action to take next and to interpret the result once it returns. Toolformer [19] showed that a language model could learn, largely without human supervision, which tools to call and when, by generating candidate API calls, executing them, and keeping only the calls that improved its downstream predictions. Reflexion [36] extended the reflection capability pattern described in Section C.3 by having an agent maintain a running episodic memory of its own past attempts and the verbal feedback it generated about them, so that a later attempt at a similar task can draw on the lessons of an earlier failure rather than starting from a blank context; this is conceptually close to the episodic memory described for this paper's own system in Section 5.4 and Section 6.3, though this paper's episodic memory stores completed planning cycles rather than failed attempts at the same cycle.

C.5 Coordination Patterns, Blackboards and Shared Context

Long before large language models, multi expert problem solving systems faced the same coordination question this paper addresses in Section 5.4, how should several independent reasoning components share partial results without being tightly coupled to one another. The blackboard architecture [33] answered this by having every knowledge source read from and write to a single shared data structure, the blackboard, rather than sending messages directly to specific other components, so that a new knowledge source can be added without modifying the others. The SharedContextBus implemented in Appendix A is a direct descendant of this pattern, updated for a setting where the knowledge sources are agents wrapping language models and external tools rather than symbolic rule engines, and

where a vector store extends the blackboard's short term working area with long term, semantically searchable memory as described in Section 6.2.

C.6 Mapping This Paper's Architecture Onto the Pattern Taxonomies

Table 7 maps every pattern introduced in this appendix onto the specific section of this paper, or class in Appendix A, where it is applied, making explicit the theoretical lineage that Sections 4 to 6 draw on without individually naming each pattern at the point of use.

Table -7 Agentic Design Pattern Mapping

Pattern	Source	Type	Definition	Where Used in This Paper
Reflection	Ng (2024) [35]	Capability	An agent critiques and revises its own output across iterations rather than emitting a single final answer.	Section 6.4, the reflect hook on BaseAgent and ForecastingAgent
Tool Use	Ng (2024) [35]; Schick et al. (2023) [19]	Capability	An agent calls external functions or APIs to obtain information or take action beyond what is in its own weights.	Section 5.1 to 5.2, all five tool wrapper classes
Planning	Ng (2024) [35]	Capability	An agent decomposes a goal into an ordered sequence of sub tasks before executing them.	Section 6.1, the orchestrator's fixed cycle plan
Multi Agent Collaboration	Ng (2024) [35]; Stone and Veloso (2000) [32]	Capability	Several specialised agents, each with a distinct role, work together on a task no single agent could complete alone.	Section 4.1, the five specialised agents plus orchestrator
Prompt Chaining	Anthropic (2024) [34]	Workflow	A task is broken into an ordered sequence of model calls, with each step's output feeding the next.	Section 5.3, the sequential run_cycle handoffs
Routing	Anthropic (2024) [34]	Workflow	An initial classification step directs the task to a specialised handler best suited to it.	The orchestrator's selection of which agent to consult for a given condition
Parallelization	Anthropic (2024) [34]	Workflow	Independent subtasks are run concurrently and their outputs aggregated programmatically.	Section 6.5, the forecast and telemetry calls
Orchestrator Workers	Anthropic (2024) [34]	Workflow	A central agent decomposes a task and delegates subtasks to worker agents whose number is not fixed in advance.	Section 4.1, the orchestrator and five workers
Evaluator Optimizer	Anthropic (2024) [34]	Workflow	One agent produces a candidate result while a second evaluates it and triggers revision in a loop.	Section 5.6, the human approval gate acting as evaluator
Blackboard Architecture	Hayes Roth (1985) [33]	Coordination	Multiple knowledge sources read and write to a shared data structure rather than passing messages directly to one another.	Section 5.4, the SharedContextBus

International Journal of Artificial Intelligence and Computer Electronics
Volume: 02, Issue: 03, July-September 2026, pp:1-32
P-ISSN 3117-8529 | E-ISSN 3117-8537

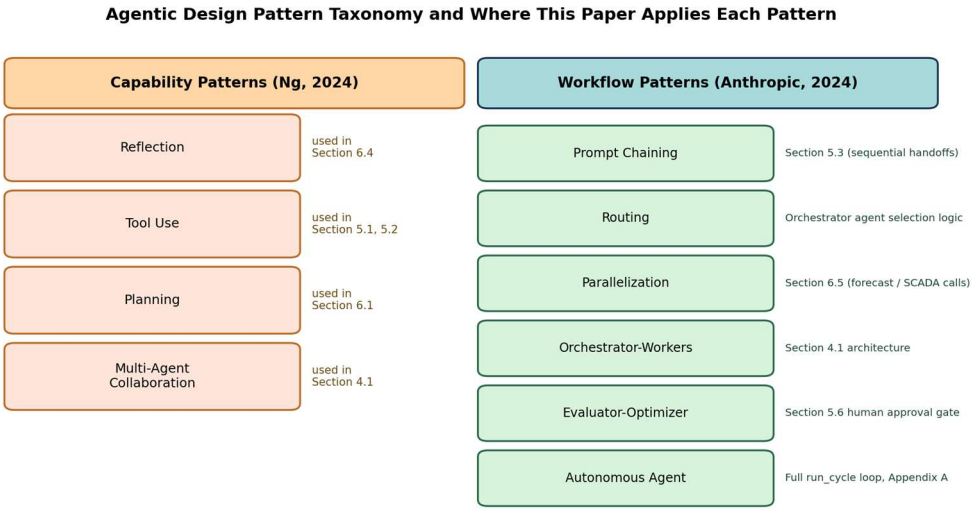

Figure 9. Agentic design pattern taxonomy and where this paper applies each pattern.

Reading Table 7 alongside Section 3.8's framework comparison clarifies a point that is easy to miss when a system is described only in implementation terms, the choice between a bespoke orchestrator, the OpenAI Agents SDK and a Claude tool use based implementation, discussed in Sections 3.7 and 3.8, is a choice of which vendor runtime executes these patterns, not a choice of which patterns to use. All three implementation routes considered in this paper apply the same orchestrator workers workflow with a multi agent collaboration capability pattern underneath it, and all three could equally well add the reflection and evaluator optimizer patterns already present in Sections 5.6 and 6.4; the patterns are the stable design vocabulary, and the vendor runtime is a replaceable detail beneath them.