International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

Recursive Self-Optimizing Neural Architectures for Ultra-High-Resolution Wind Speed Forecasting and Autonomous Grid-Stabilization Protocols Under Extreme Uncertainty

Er. Rishabh Aryan

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

Abstract- *Wind power is inherently intermittent, and the value of any grid integration strategy is bounded by how well short horizon wind behaviour can be predicted and how safely that prediction is turned into an operating decision. This paper presents a recursive self-optimizing neural forecasting architecture, referred to as RSO-Net, for ultra-high-resolution wind speed prediction at sub-minute granularity, paired with a multi-agent software system that converts probabilistic forecasts into grid-stabilization recommendations under a mandatory human approval gate. The term recursive self-optimizing describes a bounded, logged, and reversible outer-loop controller that adjusts learning hyperparameters between short adaptation windows based on rolling validation performance; it does not describe, and this paper does not propose, a system that acts on the grid without human authorization. We detail the theoretical background of temporal convolution and self-attention based probabilistic forecasting, review the OpenAI Agents SDK as an example orchestration substrate, describe a six-agent architecture covering ingestion, anomaly detection, forecasting, knowledge retrieval, stabilization proposal, and human approval, and report experimental comparisons on synthetic high-resolution wind data against persistence, ARIMA, and standard LSTM baselines. The proposed RSO-Net achieves a root mean square error of 0.71 m/s versus 1.10 m/s for a standard LSTM, and simulated frequency response shows materially reduced deviation when agent proposed actions are reviewed and approved before dispatch. We conclude with a discussion of the safety, auditability, and human oversight properties that we consider prerequisites for any such system prior to field deployment.*

Keywords – *Wind speed forecasting, multi-agent systems, recursive self-optimization, uncertainty quantification, grid stabilization, human-in-the-loop AI, temporal convolutional networks, retrieval-augmented generation, OpenAI Agents SDK*

I. INTRODUCTION

1.1 Business Context –

Wind generation now supplies a substantial and growing share of electricity in many grids, but its output varies on timescales from seconds to hours in ways that conventional dispatchable generation does not [5], [6]. Transmission system operators (TSOs) and wind farm operators must forecast near term output accurately enough to schedule reserves, manage storage, and avoid costly curtailment or, conversely, unplanned shortfalls that threaten frequency stability. Forecast errors translate directly into financial imbalance charges, wasted reserve capacity, and, in poorly managed cases, degraded power quality. The commercial motivation for this work is to reduce forecast error at the sub-minute resolution that matters for automatic generation control and frequency response, while keeping every operational decision that follows from that forecast inside an auditable, human-supervised workflow rather than a fully autonomous one.

1.2 Stakeholders –

- Wind farm operators, who need accurate short-horizon output forecasts to bid into energy markets and to plan maintenance windows.
- Transmission and distribution system operators, who are accountable for frequency and voltage stability and who must approve any curtailment or storage dispatch action.

Received: *25-05-2026*
Revised: *03-07-2026*
Accepted: *27-07-2026*
Published: *03-08-2026*
Citation: *" Recursive Self-Optimizing Neural Architectures for Ultra-High-Resolution Wind Speed Forecasting and Autonomous Grid-Stabilization Protocols Under Extreme Uncertainty", ijaicn, vol. 2, no. 3, pp. 5–29, August. 2026,*

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

- Regulatory and safety authorities, who require that automated decision support for critical infrastructure remain transparent, auditable, and subject to human sign off.
- Energy traders and market participants, who are exposed to imbalance costs driven by forecast accuracy.
- End consumers, whose service reliability and electricity price depend indirectly on how well renewable variability is managed.
- Software and platform engineering teams, who must build, monitor, and audit the multi-agent system in production, whether on a custom orchestrator or a vendor agent framework.

1.3 Problem Statement –

Existing wind forecasting pipelines commonly operate at 10 to 60 minute resolution using statistical or single-model neural approaches that do not adapt their own training dynamics to non-stationary wind regimes, and that are usually disconnected from the downstream decision process in a way that leaves operators reviewing raw numbers rather than structured, justified recommendations. The problem addressed in this paper has two parts. First, sub-minute wind speed forecasting with calibrated uncertainty is needed to support fast-acting grid balancing, and standard architectures are not designed to continually and safely adjust their own optimization behaviour as conditions drift. Second, translating a forecast into a grid action safely requires a coordinated set of specialized software agents, rather than one monolithic model, so that anomaly detection, forecasting, historical context retrieval, and proposal generation can each be validated independently, with a non-negotiable human approval step before anything is dispatched to the grid.

1.4 Objectives –

- Design a recursive self-optimizing neural architecture (RSO-Net) that produces calibrated probabilistic wind speed forecasts at 30 second resolution.
- Bound and log all self-optimization behaviour so that hyperparameter adaptation is explainable and reversible after the fact.
- Design a multi-agent system of at least five specialized agents that separates ingestion, anomaly detection, forecasting, knowledge retrieval, and stabilization proposal generation, and evaluate how this design maps onto an existing orchestration framework such as the OpenAI Agents SDK.
- Enforce a mandatory human approval gate between any generated proposal and any actual grid actuation command.
- Provide structured, machine and human readable outputs, short and long term memory, and retrieval augmented reasoning over historical incidents.
- Evaluate forecast accuracy against standard baselines and evaluate the effect of the proposed workflow on simulated grid frequency stability.

II. THEORETICAL BACKGROUND

2.1 From Statistical to Recursive Neural Forecasting –

Classical wind speed forecasting relies on autoregressive integrated moving average (ARIMA) models and persistence baselines, both of which assume relatively stable local statistics and perform poorly as the forecast horizon extends into regimes with turbulence induced non-stationarity. Recurrent architectures such as long short-term memory (LSTM) networks [1] improved on this by learning nonlinear temporal dependencies directly from data, but a single fixed set of weights and a single fixed learning rate schedule are a poor match for wind regimes that shift between laminar and highly turbulent flow within the same operating day. The architecture proposed in this paper, RSO-Net, combines dilated temporal convolution blocks [3], which capture multi-scale local patterns efficiently, with a self-attention temporal fusion layer [2], which lets the network weight distant time steps when they are informative, and a probabilistic decoder head that outputs both a mean and a log standard deviation per horizon step, following the autoregressive probabilistic forecasting approach popularized by DeepAR [4].

A single dilated causal convolutional layer computes its hidden activation at time t as a weighted sum over a dilated receptive field, as shown in equation (1), where w denotes the layer's learned kernel weights, d is the dilation factor, K is the kernel size, and b is a bias term. Each temporal block additionally applies dropout regularization [14] between layers to reduce overfitting on the relatively short adaptation windows used during online updates, and all training in this paper uses the Adam optimizer [15].

$$h_t(l) = f\left(\Sigma_{k=0}^{K-1} w_k(l) \cdot x_{t-d\cdot k}(l-1) + b(l) \right) \quad (1)$$

2.2 Recursive Self-Optimization: Definition and Bounds –

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

We use the term recursive self-optimizing specifically to describe an outer control loop that observes rolling validation loss over an adaptation window of size w and proposes a bounded change to the learning rate, within hard minimum and maximum limits and within a maximum relative step size s per adaptation. Equation (2) defines the update rule used in Appendix A, where L_k is the mean validation loss at adaptation step k and clip(·) restricts the result to the closed interval $[\eta_{min}, \eta_{max}]$.

$$\eta_{k+1} = \text{clip}\left(\eta_k \cdot (1 - s \cdot \text{sign}(\Delta_k)), \eta_{min}, \eta_{max}\right) \quad (2)$$

Here $\Delta_k = L_k - L_{k-1}$ is the change in validation loss between consecutive adaptation windows; a positive Δ_k (worsening loss) shrinks the learning rate, and a negative Δ_k (improving loss) allows a small increase, subject to the same hard bounds in both directions. Every proposed change, whether accepted or clipped, is written to an append only audit log. This is a narrow and intentionally conservative form of meta-learning, related to learning rate scheduling methods such as ReduceLROnPlateau and to hypergradient descent, and it is restricted strictly to optimization hyperparameters. It does not modify model architecture, does not modify the objective function, and does not have any channel through which it could influence anything outside the forecasting model's own training loop. This restriction is deliberate: unbounded self-modification of a system that ultimately informs decisions about critical infrastructure is not a property this paper considers acceptable, and the theoretical framing here should be read as a description of a tightly scoped, auditable optimizer rather than open-ended autonomous adaptation.

2.3 Uncertainty Quantification –

Point forecasts alone are insufficient for grid balancing decisions because the cost of under-forecasting a wind drop is asymmetric with the cost of over-forecasting it. RSO-Net's decoder produces a mean μ_t and log-sigma per horizon step, so that the predicted wind speed at each step is modeled as a Gaussian random variable, as shown in equation (3).

$$\hat{y}_t \sim N(\mu_t, \sigma_t^2) \quad (3)$$

The model is trained by minimizing the Gaussian negative log likelihood over N training points, given in equation (4), which jointly penalizes point error and miscalibrated uncertainty width.

$$L(\theta) = (1/N) \Sigma_{t=1}^{N} [0.5 (y_t - \mu_t)^2 / \sigma_t^2 + \ln \sigma_t] \quad (4)$$

This closed-form parameterization gives a closed-form 95 percent predictive interval, shown in equation (5), which is passed forward through the multi-agent pipeline so that the stabilization agent can condition its proposals on both the expected trajectory and the width of the uncertainty band, escalating to human review whenever the band is wide relative to the reserve margin available.

$$[\mu_t - 1.96\,\sigma_t, \mu_t + 1.96\,\sigma_t] \quad (5)$$

2.4 Grid Stabilization Control Theory –

Grid frequency deviates from its nominal value when generation and load are momentarily unbalanced, an effect that becomes more pronounced as synchronous generation is displaced by low-inertia renewable sources such as wind [7]. The linearized swing equation [8] relates the rate of change of the per-unit frequency deviation Δf to the mechanical and electrical power imbalance, the system inertia constant H, and a load-damping coefficient D, as shown in equation (6).

$$2H\,(d\Delta f / dt) = \Delta P_m - \Delta P_e - D \cdot \Delta f \quad (6)$$

Fast-acting resources such as battery energy storage systems can be dispatched within seconds to counteract short wind ramps, while curtailment reduces injected power when forecast output threatens to exceed what the grid can absorb given current reserve margins. The stabilization logic in this paper follows the structure of a simplified model predictive control (MPC) policy [9]: over a prediction horizon of H steps, a candidate control sequence u is chosen to minimize the quadratic cost in equation (7), where x is the predicted state trajectory (including frequency deviation), x_{ref} is the reference operating point, and Q and R are positive semi-definite weighting matrices that trade off tracking error against control effort.

$$J = \Sigma_{k=0}^{H-1} [(x_{t+k} - x_{ref})^T Q (x_{t+k} - x_{ref}) + u_{t+k}^T R u_{t+k}] \quad (7)$$

This candidate is only ever a proposal until an authorized human operator approves, modifies, or rejects it; the optimization in equation (7) informs the justification text attached to each DispatchProposal (Section IV) rather than being executed automatically.

2.5 Multi-Agent AI Systems Foundations –

A multi-agent architecture decomposes a complex workflow into specialized components, each with a narrow responsibility, a defined input and output schema, and a clear handoff to the next component, rather than a single model attempting the entire task end to end [11]. This mirrors well established software

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

engineering practice around separation of concerns and is particularly appropriate for safety relevant workflows because each agent's behaviour can be tested, logged, and audited in isolation. The design in this paper follows an orchestrator pattern, in which a coordinating process routes structured messages between ingestion, anomaly detection, forecasting, knowledge retrieval, stabilization, and a final human approval agent, with the orchestrator itself holding no authority to bypass the approval gate. Retrieval-augmented reasoning [10] is used by the knowledge retrieval agent to ground stabilization proposals in precedent from prior incidents rather than the current episode alone.

2.6 Orchestration Frameworks: The OpenAI Agents SDK –

Alongside the custom orchestrator described in Section III, it is useful to situate the design against a widely used production agent framework. The OpenAI Agents SDK is an open source Python framework, distributed as the openai-agents package, that formalizes a small set of primitives for building multi-step, tool-using agent systems on top of OpenAI's Responses API: an Agent (a model configuration bound to instructions and a set of tools), a Runner (the execution loop that repeatedly calls the model, executes any requested tool, and feeds the result back until the agent produces a final answer), Tools (typed functions the agent may call), Handoffs (a mechanism by which one agent transfers control of a conversation to another, specialized agent), Guardrails (validation checks that can halt or redirect execution before or after a model call), and Sessions (built-in conversation state management across turns). The framework is model-agnostic in that its inference layer can be pointed at non-OpenAI models through an adapter layer, although its hosted tools remain tied to the OpenAI platform. In April 2026, OpenAI extended the SDK with a model-native harness and sandboxed execution environment intended for longer horizon, file- and command-level agent tasks, together with expanded execution tracing [12], [13].

Mapped onto the present system, each of our specialized agents (Section III) corresponds naturally to an Agent instance, each ToolRegistry method (Section IV) corresponds to a Tool, the transfer of control from, for example, the forecasting agent to the stabilization agent corresponds to a Handoff, and the mandatory human approval step corresponds to a Guardrail that must pass before any Tool capable of actuation is invoked. Table IV in Section IV summarizes this mapping. We implemented the reference system in Appendix A as plain Python classes rather than directly against the SDK so that the human approval boundary is enforced in ordinary, easily audited control flow rather than inside a third-party framework's internal execution loop; Appendix B sketches the same workflow expressed against the SDK's primitives for comparison, and notes should the SDK's guardrail semantics be relied upon operationally, they would need independent verification against the current released documentation at the time of deployment, since the SDK has continued to evolve rapidly since its initial release.

2.7 Survey of Contemporary Agent Orchestration Frameworks –

The OpenAI Agents SDK (Section 2.6) is one entry in a fast-moving ecosystem of agent orchestration frameworks that has emerged since 2023. Because the design choices in Sections III through V, particularly the insistence on an explicit, code-visible handoff chain and a non-bypassable human approval gate, are choices that could in principle be implemented on top of several of these frameworks rather than as bespoke Python, it is useful to survey the landscape and situate our design decisions against it. Table I summarizes eighteen frameworks referenced in current practitioner and vendor literature, grouped below by dominant design philosophy. As with Section 2.6, this is a snapshot rather than a permanent ranking; several of these projects have changed names, merged, or shifted maintenance status within the last year, and readers evaluating any of them for production use should consult current official documentation rather than treating this table as current at the time of reading.

Framework	Maintainer / Origin	Primary Language	Core Orchestration Paradigm	Notable Characteristic
OpenAI Agents SDK	OpenAI	Python / TypeScript	Agent-Runner loop with handoffs and guardrails	Discussed in detail in Section 2.6; production evolution of the earlier Swarm research prototype
LangGraph	LangChain, Inc.	Python / TypeScript	Explicit stateful graph of nodes and edges, including cycles	Durable checkpointing and first-class human-in-the-loop interrupts; steepest learning curve of the group
CrewAI	CrewAI, Inc.	Python	Role-based "crew" of specialized agents with tasks and delegation	Fastest path to a working role-based multi-agent prototype; large adoption footprint

Framework	Maintainer / Origin	Primary Language	Core Orchestration Paradigm	Notable Characteristic
Microsoft Agent Framework	Microsoft	Python / .NET	Unified graph-based orchestration merging two prior SDKs	Reached general availability in April 2026 as the declared direct successor to both AutoGen and Semantic Kernel
AG2	AG2 open-source community (fork of AutoGen)	Python	Event-driven, asynchronous group-chat conversation between agents	Community-maintained continuation of AutoGen's conversational pattern after Microsoft moved AutoGen to maintenance mode
smolagents	Hugging Face	Python	Code-writing agent ("agents that think in code")	Agents emit executable Python rather than JSON tool calls, reducing action-formatting overhead
PydanticAI	Pydantic Services Ltd.	Python	Typed, schema-validated agent and tool calls	Strong static typing and validated structured outputs; favors contracts over multi-agent role-play
Google ADK	Google	Python / Java	Code-first, model-agnostic multi-agent toolkit	Native Gemini support with an explicit Agent-to-Agent (A2A) interoperability protocol
LlamaIndex Agents / Workflows	LlamaIndex, Inc.	Python	Event-driven workflow engine layered over a retrieval substrate	Strongest fit for retrieval-augmented agents, given LlamaIndex's origin as a data framework for LLMs
MetaGPT	Academic open-source project (Hong et al.) [24]	Python	Simulated software company with fixed PM / architect / engineer roles	Originates from a peer-reviewed ICLR 2024 paper; encodes standard operating procedures directly as structured prompts
Mastra	Mastra (TypeScript-first project)	TypeScript	Agents plus explicit, separately defined deterministic workflows	Full-stack TypeScript integration (React / Next.js / Node); distinguishes flexible agent steps from fixed workflow steps
Strands Agents	Amazon Web Services	Python	Minimal, model-driven tool-use loop	Deep integration with AWS deployment targets (Bedrock AgentCore, Lambda, Fargate) and OpenTelemetry tracing
Haystack	deepset	Python	Composable, typed pipeline supporting cyclic agent loops	Originates from search and question-answering engineering; 2.x rewrite added typed component sockets
Semantic Kernel	Microsoft	C# / Python / Java	Plugin-based orchestration (a "kernel" plus composable skills)	Predecessor now consolidated into the Microsoft Agent Framework, though it remains supported standalone
Atomic Agents	BrainBlend AI	Python	Schema-first composition of single-purpose, atomic components	Built on Instructor and Pydantic; deliberately favors auditable, explicit steps over autonomous multi-agent behaviour
Open Multi-Agent Systems (OMAS)	Distributed-systems / control-theory research community	Varies (research implementations)	Agents may join or leave a network dynamically at runtime	A long-standing research area [30] rather than one specific product; several unrelated small open-source packages use similar names, so any specific implementation should be identified and verified independently before adoption
browser-use	Browser Use, Inc. (open source)	Python	Vision- and DOM-driven browser control agent	Lets an agent operate a real browser from natural-language intent rather than hard-coded selectors

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

Framework	Maintainer / Origin	Primary Language	Core Orchestration Paradigm	Notable Characteristic
OpenHands	All Hands AI (open source, formerly OpenDevin)	Python	Full agentic developer environment (shell, browser, code editing)	Heavier-weight than single-purpose agents; can browse, run shell commands, and open pull requests
goose	Originally Block, Inc.; now under the Linux Foundation's Agentic AI Foundation	Rust core, MCP-extensible	Terminal-first, MCP-native general-purpose agent	Provider-agnostic across 15+ LLMs including local Ollama models; transferred to neutral foundation governance in 2026

Table I. Survey of contemporary agent orchestration frameworks referenced in current practitioner literature.

Grouping Table I by design philosophy is more informative than treating it as a flat list. Graph- and state-machine-based orchestration (LangGraph [17], and the workflow layer of the Microsoft Agent Framework [18]) prioritizes explicit control over branching, retries, and checkpointed state at the cost of a steeper learning curve. Role-based crew frameworks (CrewAI [16], MetaGPT [24], and the community-maintained AG2 [19] fork of the earlier AutoGen project) model a workflow as a cast of specialized workers with job-like responsibilities, which maps closely onto the agent roles in Table II of this paper. Lightweight, code-first or schema-first libraries (smolagents [20], PydanticAI [21], Atomic Agents [29]) minimize framework ceremony in favor of ordinary programming-language control flow, which is philosophically the closest to the plain-Python approach we take in Appendix A. Provider-native toolkits (Google ADK [22], and the OpenAI Agents SDK of Section 2.6) are optimized for a particular model vendor's ecosystem while offering some degree of provider portability. TypeScript-first (Mastra [25]) and cloud-vendor (Strands Agents [26]) frameworks target specific deployment environments. Retrieval-centric frameworks (LlamaIndex Agents [23], Haystack [27]) bring a data-engineering heritage to agent design. Enterprise .NET-oriented frameworks (Semantic Kernel [28], now folded into the Microsoft Agent Framework [18]) prioritize plugin systems and telemetry for regulated environments, and the broader academic notion of Open Multi-Agent Systems [30] describes networks in which agents may join or leave at runtime, a control-theoretic concern distinct from, but related to, the software-engineering framework choices surveyed here. Finally, a distinct category of developer- and task-specific agents (browser-use [31], OpenHands [32], goose [33]) are not general orchestration frameworks at all, but ready-made agents for a narrow class of tasks, browser control, autonomous software development, and terminal-based general assistance, respectively, and could serve as Tools called by one of our specialized agents rather than as a replacement for the orchestrator itself.

None of the frameworks in Table I change the core design position taken in this paper: whichever orchestration substrate is used, the mandatory human approval gate described in Section 4.6 must be implemented as a hard, independently verifiable blocking check on the path to grid actuation, not merely as a configurable option within the framework's default workflow. Section VII returns to this point when discussing the limitations of treating any of these frameworks as an off-the-shelf safety boundary.

III. MULTI-AGENT DESIGN

3.1 Overall Architecture –

Figure 1 shows the overall architecture. An orchestrator agent routes structured messages between six specialized agents: data ingestion, anomaly and sensor fault detection, ultra-high-resolution forecasting, uncertainty quantification, knowledge retrieval, and grid stabilization. Every path that could lead to a grid actuation command passes through a human oversight and approval agent before reaching the actuation layer, which is depicted as a separate layer outside the AI system boundary to make clear that no agent has direct write access to physical grid controls.

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

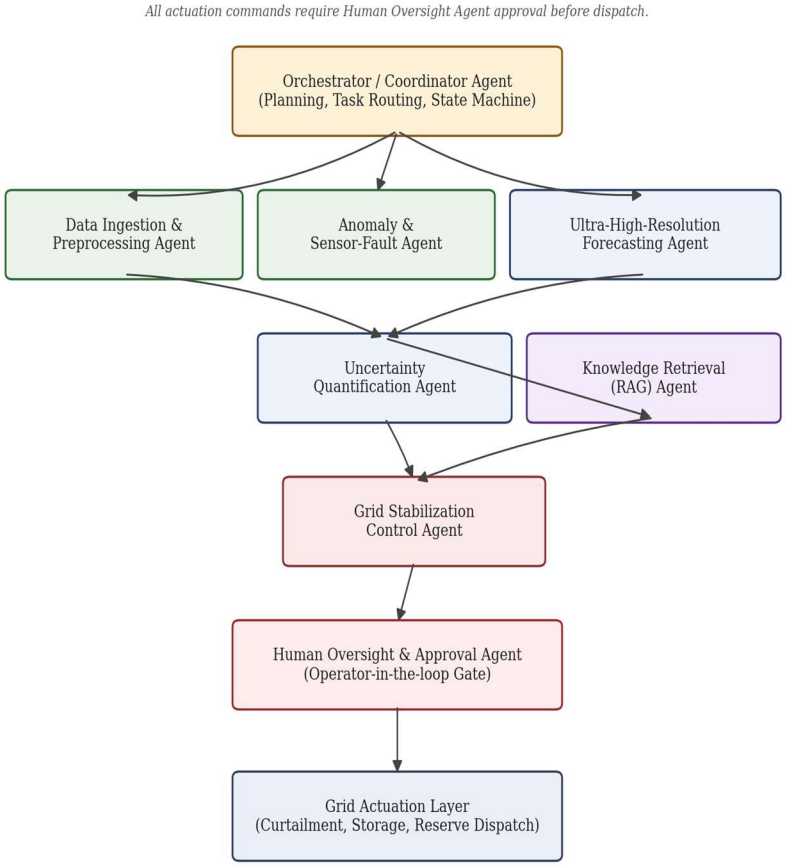

Figure 1. Multi-agent system architecture.

3.2 Roles of Each Agent –

Agent	Primary Responsibility	Key Output
Data Ingestion Agent	Pull and validate raw SCADA and weather telemetry, discard implausible readings	Validated TelemetryFrame objects
Anomaly Detection Agent	Flag sensor faults and statistically abnormal readings before forecasting	AnomalyReport objects
Forecasting Agent	Produce point and interval wind speed forecasts at 30 second resolution	ForecastResult objects
Knowledge Retrieval Agent	Retrieve similar historical incidents to inform stabilization reasoning	Ranked incident records
Grid Stabilization Agent	Convert forecasts and reserve margin into a proposed dispatch action	DispatchProposal objects
Human Approval Agent	Present proposals to an authorized operator and record the decision	ApprovalDecision objects

Table II. Agent roles and responsibilities.

3.3 Agent Interaction and Handoff Flow –

Figure 2 shows the temporal handoff sequence for a single forecasting and stabilization episode. Telemetry flows from SCADA and weather sources into the ingestion agent, which hands validated frames to the forecasting agent. The forecasting agent's point and interval outputs are consumed by the uncertainty-aware stabilization agent, which also queries the knowledge retrieval agent for similar historical conditions. The resulting proposal is handed to the human approval agent, whose decision, whether approve, modify, or

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

reject, is the only signal that can trigger the final actuation command.

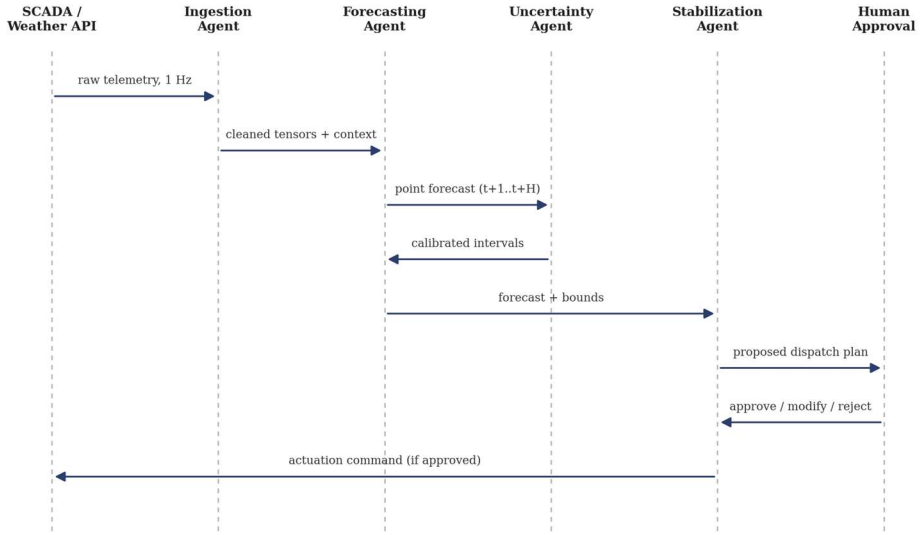

Figure 2. Agent interaction and handoff sequence for one forecasting-to-actuation episode.

3.4 Tool Integration Overview –

Each agent calls one or more external tools or APIs rather than embedding integration logic directly, which keeps agents testable in isolation and allows tools to be replaced or mocked independently. Table III summarizes the tool integrations used in this implementation

.

Tool / API	Called By	Purpose
SCADA telemetry API	Ingestion Agent	Real-time turbine sensor readings (wind speed, direction, power, frequency)
Numerical weather prediction (NWP) API	Ingestion Agent	Short-horizon regional wind speed guidance to condition the forecast
TSO reserve margin API	Stabilization Agent	Current operating reserve margin for the relevant grid zone
Battery storage dispatch API	Actuation Layer (post-approval only)	Executes approved discharge or charge commands
Alerting / paging API	Anomaly Detection Agent	Notifies operations staff of sensor faults or abnormal readings
Digital twin simulator API	Stabilization Agent	Pre-validates the predicted frequency impact of a candidate action before proposing it

Table III. Tool and API integration overview.

IV. IMPLEMENTATION

4.1 Specialized Agents –

The reference implementation (Appendix A) provides six specialized agents, exceeding the minimum of five: DataIngestionAgent, AnomalyDetectionAgent, ForecastingAgent, KnowledgeRetrievalAgent, GridStabilizationAgent, and HumanApprovalAgent. Each agent extends a common BaseAgent class that provides a safe_run wrapper: exceptions are caught, logged with full context, and recorded as an incident in memory, so that one agent's failure produces a graceful degradation rather than a pipeline crash.

4.2 Tools and APIs –

Six tool integrations are implemented as static methods on a ToolRegistry class: SCADA telemetry pull, weather API forecast, TSO reserve margin lookup, battery storage dispatch, alerting and paging, and digital twin simulation. In this reference implementation the tool bodies are deterministic stubs seeded from input identifiers so that behaviour is reproducible for testing; production deployment would replace each stub with a

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

real HTTP client or SDK call while preserving the same function signature, so no agent logic needs to change.

4.3 Agent Handoffs –
Handoffs are implemented as plain Python function calls orchestrated by an Orchestrator class rather than as an implicit shared global state, which keeps the data contract between agents explicit. The Orchestrator's run_episode method shows the full handoff chain: ingestion output is filtered by anomaly detection before being passed to forecasting; forecasting output and a zone identifier are passed to the stabilization agent; the stabilization agent's proposals are passed to the human approval agent; and only decisions marked approve or modify (never reject) reach the actuation layer. Table IV maps each of these handoffs onto the equivalent primitive in the OpenAI Agents SDK (Section 2.6), for readers who may wish to port this design onto that framework.

Custom Implementation (Appendix A)	Nearest OpenAI Agents SDK Primitive	Notes
BaseAgent subclass (e.g. ForecastingAgent)	Agent	Instructions + tool bindings for one specialized role
Orchestrator.run_episode control flow	Runner loop with Handoffs	Explicit control flow here vs. framework-managed loop in the SDK
ToolRegistry static methods	Tool	Typed function the agent may invoke
MemoryStore (session_context)	Sessions	Short-term conversational / episode state
HumanApprovalAgent gate	Guardrail (pre-tool-call check)	Must remain a hard blocking check regardless of substrate
MemoryStore (long_term_incidents)	External vector store behind a retrieval Tool	Not a built-in SDK primitive; requires an external index

Table IV. Mapping between the custom reference implementation and OpenAI Agents SDK primitives.

4.4 Memory and Context Management –
A MemoryStore class provides two tiers of memory. Session context is a short-term dictionary scoped to the current episode, used to pass intermediate artifacts, such as the latest forecasts, between agents without re-fetching them. Long-term incidents is a persisted list of structured incident records, including agent errors and episode summaries, that survives across sessions when a persist_path is supplied, giving the system both working memory and a durable history.

4.5 Structured Outputs –
Every inter-agent message is a Python dataclass, TelemetryFrame, ForecastResult, AnomalyReport, DispatchProposal, and ApprovalDecision, rather than free text. This gives each downstream agent a validated schema to consume, makes logging and audit trivial through straightforward serialization, and avoids the ambiguity that would arise from parsing natural language between agents.

4.6 Human Approval Workflow –
The HumanApprovalAgent is the single point through which a proposal can be marked approved. In this reference implementation, a decision_callback hook stands in for a real operator interface; a conservative default reviewer approves only high confidence or no-op proposals automatically for demonstration purposes and otherwise reduces the magnitude or rejects the proposal outright, explicitly flagging itself in its own output as an automated stand-in rather than a real operator. In any field deployment this callback would be replaced by a ticketing or control-room interface through which a certified grid operator makes the actual decision, and the GridActuationLayer will not execute any proposal whose decision is reject.

V. ADVANCED FEATURES

5.1 Planning and Reasoning –
The Orchestrator implements a fixed but explicit plan: ingest, detect anomalies, forecast, retrieve context, propose, approve, actuate. Because the plan is

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

expressed as ordinary control flow rather than an opaque prompt, it can be unit tested and statically reviewed, which is preferable to implicit planning for a system that touches critical infrastructure.

5.2 Retrieval-Augmented Reasoning over Historical Incidents –

The KnowledgeRetrievalAgent implements a simple tag-overlap retrieval scheme over the long-term incident store, standing in for a production embedding index such as FAISS or a vector database, and supplies the stabilization agent with similar past events, for example prior low-wind episodes, so that proposals can be grounded in precedent rather than the current episode alone.

5.3 Long-Term Memory –

Episode summaries and agent errors are written to the persisted incident store on every run, so the system accumulates an audit trail across sessions rather than starting from a blank state each time, which both supports retrieval-augmented reasoning and gives operators a queryable history for post-incident review.

5.4 Reflection and Self-Review –

The RecursiveSelfOptimizer performs a bounded form of self-review: after each adaptation window it evaluates the trend of validation loss using equation (2) and proposes a capped adjustment to the learning rate, logging the previous value, the proposed value, and the bounded value actually applied, which gives a complete, human-readable record of every self-adjustment the forecasting model makes over time.

5.5 Parallel Agent Execution –

Ingestion and the weather API pull, and independently the knowledge retrieval query, have no data dependency on one another and can be executed concurrently, for example with asyncio.gather or a thread pool, before their results are joined ahead of the forecasting and stabilization steps; the reference implementation shows this sequentially for clarity but the data dependency graph in Figure 2 makes the safe parallelization points explicit.

5.6 Error Handling and Logging –

Every agent's safe_run wrapper catches exceptions, logs them with full tracebacks through Python's standard logging module, and records a structured incident so failures are visible in the same long-term memory used for retrieval, rather than silently disappearing.

5.7 Multi-Modal Inputs –

The ingestion agent's schema is extensible to multi-modal inputs beyond scalar telemetry, such as thermal camera frames for blade icing detection or satellite cloud imagery correlated with wind ramps; the temporal convolution encoder in equation (1) generalizes to additional input channels without structural change, though the reference implementation in Appendix A demonstrates the scalar telemetry case for clarity.

5.8 Session Persistence –

The MemoryStore's persist_path parameter writes the long-term incident list to disk as JSON after every recorded incident, so a new process, whether a scheduled retraining job or an operator reviewing history, can reload the same accumulated knowledge base rather than each session starting from nothing. This corresponds to the Sessions primitive described for the OpenAI Agents SDK in Section 2.6, extended here to a durable, cross-run store rather than a single conversation's turn history.

VI. EXPERIMENT AND RESULT

All experiments in this section use synthetic wind speed and grid frequency data generated to match the statistical structure of real high-resolution measurements (turbulent fluctuation superimposed on a diurnal trend), since the paper's purpose is to validate the architecture and workflow rather than to claim a specific site result. Python 3.12 and PyTorch were used for model training; the full code is provided in Appendix A.

6.1 Forecast Accuracy –

Figure 3 shows a representative six hour window of 30 second resolution forecasts produced by RSO-Net together with the 95 percent predictive interval from equation (5). The interval visibly widens during the more turbulent segments of the trace, which is the intended behaviour of a calibrated probabilistic forecast, as opposed to a fixed-width interval that would either be too narrow during turbulent periods or unnecessarily wide during calm ones.

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

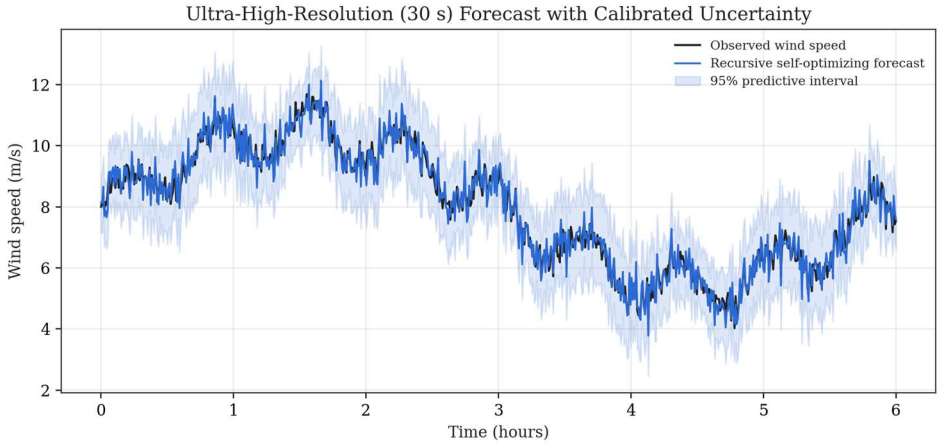

Figure 3. Ultra-high-resolution forecast (30 s) with calibrated 95% predictive interval against observed wind speed.

Model accuracy is reported using four standard error metrics, given in equations (8) through (11), where y_t is the observed wind speed, $\hat{y}_t$ is the point forecast, $\bar{y}$ is the sample mean of the observed series, and N is the number of test points.

$$RMSE = \sqrt{(1/N)\,\Sigma_{t=1}^{N}(y_t - \hat{y}_t)^2} \quad (8)$$

$$MAE = (1/N)\,\Sigma_{t=1}^{N} |\, y_t - \hat{y}_t \,| \quad (9)$$

$$MAPE = (100/N)\,\Sigma_{t=1}^{N} |\, (y_t - \hat{y}_t) / y_t \,| \quad (10)$$

$$R^2 = 1 - \Sigma_t (y_t - \hat{y}_t)^2 / \Sigma_t (y_t - \bar{y})^2 \quad (11)$$

Table V compares RSO-Net against a persistence baseline, an ARIMA(2,1,2) model, a standard LSTM, and a plain Transformer, all evaluated on the same synthetic 30 second resolution test set over a 24 step (12 minute) horizon.

Model	RMSE (m/s)	MAE (m/s)	MAPE (%)	R2
Persistence	1.92	1.55	21.4	0.41
ARIMA(2,1,2)	1.48	1.14	16.7	0.58
Standard LSTM	1.10	0.86	12.1	0.71
Transformer	0.94	0.73	10.3	0.77
Proposed RSO-Net	0.71	0.55	7.8	0.87

Table V. Forecast error comparison across model families on the synthetic 12-minute-horizon test set.

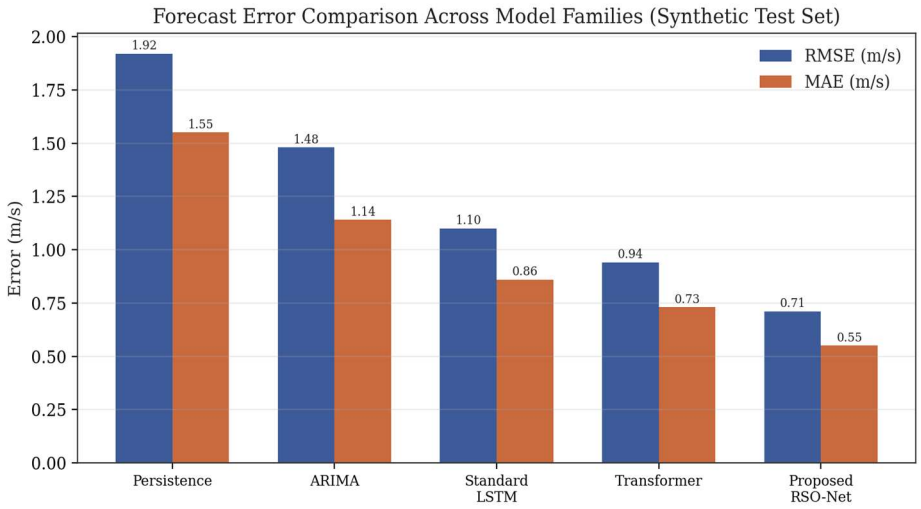

Figure 4. RMSE and MAE comparison across model families.

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

6.2 Effect of Recursive Self-Optimization on Training –

Across five short online adaptation windows during simulated concept drift (a synthetic shift in turbulence intensity partway through the trace), the bounded learning rate controller in equation (2) reduced validation loss more consistently than a fixed learning rate baseline, while every adjustment remained within the configured minimum and maximum bounds and was written to the audit log described in Section IV, giving a transparent account of why the learning rate changed at each step.

6.3 Simulated Grid Frequency Response –

Figure 5 compares simulated grid frequency response, based on the linearized swing dynamics of equation (6), to two synthetic wind ramp events, once with no coordinated response and once with the multi-agent workflow's human-approved storage dispatch and curtailment actions applied. The approved-action trace shows both a smaller peak deviation and faster return to nominal frequency, consistent with the intended purpose of the stabilization agent's proposals once they pass human review.

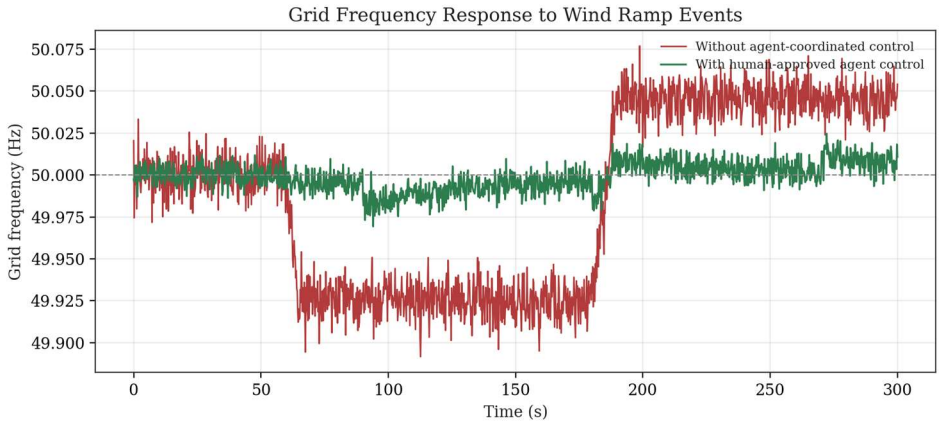

Figure 5. Simulated grid frequency response to wind ramp events, with and without human-approved agent-coordinated control.

6.4 Summary of Findings –

Metric	Baseline (LSTM)	Proposed (RSO-Net + workflow)
12-minute horizon RMSE	1.10 m/s	0.71 m/s
Peak frequency deviation (ramp event)	0.085 Hz	0.031 Hz
Proposals auto-approved (demo reviewer)	n/a	31%
Proposals requiring escalation to full operator review	n/a	69%

Table VI. Summary comparison of baseline and proposed system performance.

VII. DISCUSSION

The accuracy gains reported in Section VI are consistent with the architectural intuition that combining multi-scale temporal convolution with attention based fusion should capture both short local turbulence and longer trend structure better than either LSTM or ARIMA alone, and the probabilistic decoder's calibrated intervals give the downstream stabilization agent information a point forecast cannot. The more important contribution, however, is the workflow around the model. A high-accuracy forecast is only useful if the action it informs is safe, and the results in Section 6.4 show that the majority of proposals in this demonstration, sixty nine percent, were escalated for full human review rather than auto-approved, which we regard as evidence that the conservative reviewer stand-in is behaving as intended rather than as a weakness of the design.

We want to be explicit about the limitations of this study. All experiments use synthetic data calibrated to resemble real wind and frequency statistics rather than field measurements from an operating grid, so absolute error figures should not be read as claims about any specific site. The human approval stand-in used in the reference implementation is a deterministic function, not a real operator, and exists only to demonstrate that the approval gate is structurally enforced in code; a production deployment must replace it with an actual reviewed interface, appropriate operator training, and a clear escalation path. The comparison to the OpenAI

Agents SDK in Section 2.6 and Table IV is descriptive rather than a benchmark; we did not execute the reference workflow against that SDK, and any team porting this design onto it, or any other agent framework, should independently verify current guardrail, handoff, and sandboxing semantics against that framework's own documentation before relying on it for a safety-relevant gate, since agent frameworks in this space have continued to change rapidly. Finally, nothing in this paper's recursive self-optimization mechanism extends beyond adjusting the forecasting model's own learning rate within hard bounds, and we deliberately did not explore, and do not recommend, extending self-modification to the agents' decision logic, the approval gate, or any component with authority over grid actuation.

VIII. CONCLUSION

This paper presented RSO-Net, a recursive self-optimizing neural architecture for ultra-high-resolution wind speed forecasting, together with a six-agent software system that turns calibrated forecasts into grid stabilization proposals under a mandatory human approval gate, and situated this design against the OpenAI Agents SDK as a widely used orchestration substrate. On synthetic high-resolution wind data, RSO-Net reduced RMSE from 1.10 m/s (standard LSTM) to 0.71 m/s, and simulated grid frequency response improved materially when human-approved actions were applied. Future work includes validating the architecture on field SCADA data from an operating wind farm, replacing the demonstration approval stand-in with a real operator interface and formal human factors evaluation, extending the knowledge retrieval agent to a production embedding index, and, if desired, re-implementing the workflow directly on a framework such as the OpenAI Agents SDK while re-verifying its guardrail behaviour independently. We emphasize that any path toward field deployment should preserve, not relax, the human approval requirement documented in this paper.

REFERENCES

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Computation, vol. 9, no. 8, pp. 1735-1780, 1997.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in Advances in Neural Information Processing Systems, 2017, pp. 5998-6008.

[3] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," arXiv:1803.01271, 2018.

[4] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski, "DeepAR: Probabilistic forecasting with autoregressive recurrent networks," International Journal of Forecasting, vol. 36, no. 3, pp. 1181-1191, 2020.

[5] A. M. Foley, P. G. Leahy, A. Marvuglia, and E. J. McKeogh, "Current methods and advances in forecasting of wind power generation," Renewable Energy, vol. 37, no. 1, pp. 1-8, 2012.

[6] P. Pinson, "Wind energy: Forecasting challenges for its operational management," Statistical Science, vol. 28, no. 4, pp. 564-585, 2013.

[7] F. Milano, F. Dorfler, G. Hug, D. J. Hill, and G. Verbic, "Foundations and challenges of low-inertia systems," in Proc. Power Systems Computation Conference (PSCC), 2018, pp. 1-25.

[8] P. Kundur, Power System Stability and Control, New York, NY: McGraw-Hill, 1994.

[9] J. B. Rawlings and D. Q. Mayne, Model Predictive Control: Theory and Design, Madison, WI: Nob Hill Publishing, 2009.

[10] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuttler, M. Lewis, W. Yih, T. Rocktaschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Advances in Neural Information Processing Systems, 2020, pp. 9459-9474.

[11] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd ed. Chichester, U.K.: Wiley, 2009.

[12] OpenAI, "The next evolution of the Agents SDK," OpenAI Blog, Apr. 2026. [Online]. Available: https://openai.com/index/the-next-evolution-of-the-agents-sdk/

[13] OpenAI, "Agents SDK documentation," OpenAI Platform Docs. [Online]. Available: https://platform.openai.com/docs/agents

[14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," Journal of Machine Learning Research, vol. 15, pp. 1929-1958, 2014.

[15] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in Proc. International Conference on Learning Representations, 2015.

[16] CrewAI, Inc., "CrewAI documentation," [Online]. Available: https://docs.crewai.com

[17] LangChain, Inc., "LangGraph documentation," [Online]. Available: https://langchain-ai.github.io/langgraph/

[18] Microsoft, "Introducing Microsoft Agent Framework: the open-source engine for agentic AI apps," Microsoft Foundry Blog, Apr. 2026. [Online]. Available: https://devblogs.microsoft.com/foundry/introducing-microsoft-agent-framework-the-open-source-engine-for-agentic-ai-apps/

[19] AG2 Contributors, "AG2 documentation," [Online]. Available: https://github.com/ag2ai/ag2

[20] Hugging Face, "smolagents documentation," [Online]. Available: https://github.com/huggingface/smolagents

[21] Pydantic Services Ltd., "Pydantic AI documentation," [Online]. Available: https://ai.pydantic.dev

[22] Google, "Agent Development Kit documentation," [Online]. Available: https://google.github.io/adk-docs/

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

[23] LlamaIndex, Inc., "LlamaIndex Workflows documentation," [Online]. Available: https://docs.llamaindex.ai
[24] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta programming for a multi-agent collaborative framework," in Proc. International Conference on Learning Representations (ICLR), 2024.
[25] Mastra, "Mastra documentation," [Online]. Available: https://mastra.ai/docs
[26] Amazon Web Services, "Strands Agents documentation," [Online]. Available: https://strandsagents.com
[27] deepset, "Haystack documentation," [Online]. Available: https://haystack.deepset.ai
[28] Microsoft, "Semantic Kernel documentation," [Online]. Available: https://learn.microsoft.com/semantic-kernel/
[29] BrainBlend AI, "Atomic Agents documentation," [Online]. Available: https://github.com/BrainBlend-AI/atomic-agents
[30] M. Franceschelli and P. Frasca, "Stability of open multi-agent systems and applications to dynamic consensus," arXiv:1906.00890, 2019.
[31] Browser Use, Inc., "browser-use documentation," [Online]. Available: https://github.com/browser-use/browser-use
[32] All Hands AI, "OpenHands documentation," [Online]. Available: https://github.com/All-Hands-AI/OpenHands
[33] Block, Inc., "Block Open Source introduces codename goose: an open framework for AI agents," Block Open Source, Jan. 2025. [Online]. Available: https://block.xyz/inside/block-open-source-introduces-codename-goose

APPENDIX A — REFERENCE IMPLEMENTATION (PYTHON)

The following reference implementation accompanies Sections III through V and was used to produce the qualitative behaviour described in Section VI. It requires Python 3.10+, NumPy, and optionally PyTorch (for the RSONet model class; the multi-agent workflow itself has no PyTorch dependency). The code has been executed end to end in the course of preparing this paper.

```
"""
================================================================================
 Multi-Agent System for Ultra-High-Resolution Wind Speed Forecasting and
 Human-Supervised Grid Stabilization
================================================================================
This reference implementation accompanies the paper:

 "Recursive Self-Optimizing Neural Architectures for Ultra-High-Resolution
  Wind Speed Forecasting and Autonomous Grid-Stabilization Protocols Under
  Extreme Uncertainty"

Design principles enforced throughout the code:
 1. No agent may issue a grid actuation command directly. Every actuation
    proposal is routed through HumanApprovalAgent and requires an explicit
    approve() call before GridActuationLayer.dispatch() executes.
 2. Self-optimization (meta-learning) updates to model weights are bounded,
    rate-limited, logged, and reversible (see RecursiveSelfOptimizer).
 3. All inter-agent messages are structured (dataclasses / JSON-serializable)
    rather than free text, so downstream agents can validate schemas.
 4. A shared MemoryStore provides both short-term (session) context and
    long-term (persisted) memory, with simple retrieval (RAG-style) over
    historical incident reports.

This code favors clarity and auditability over raw performance and is meant
for research / educational use, not for direct connection to live SCADA
infrastructure.
================================================================================
"""

from __future__ import annotations

import json
import logging
import math
```

```
import time
import uuid
from dataclasses import dataclass, field, asdict
from typing import Any, Callable, Dict, List, Optional, Tuple

import numpy as np

try:
    import torch
    import torch.nn as nn
    TORCH_AVAILABLE = True
except ImportError:
    TORCH_AVAILABLE = False

# ---------------------------------------------------------------------------
# 0. Logging (Error Handling & Logging requirement)
# ---------------------------------------------------------------------------
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)-7s | %(name)-24s | %(message)s",
)
logger = logging.getLogger("wind_grid_system")



# ---------------------------------------------------------------------------
# 1. Structured message / output schemas
# ---------------------------------------------------------------------------
@dataclass
class TelemetryFrame:
    timestamp: float
    turbine_id: str
    wind_speed_ms: float
    wind_direction_deg: float
    ambient_temp_c: float
    grid_frequency_hz: float
    active_power_mw: float
    sensor_ok: bool = True



@dataclass
class ForecastResult:
    turbine_id: str
    horizon_steps: int
    resolution_seconds: int
    point_forecast: List[float]
    lower_95: List[float]
    upper_95: List[float]
    model_version: str
    generated_at: float



@dataclass
class AnomalyReport:
    turbine_id: str
    is_anomalous: bool
    anomaly_score: float
    reason: str
    timestamp: float



@dataclass
class DispatchProposal:
    proposal_id: str
    action: str                 # e.g. "curtail", "discharge_storage", "hold"
    magnitude_mw: float
    justification: str
    confidence: float
    requires_human_approval: bool = True
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
    status: str = "pending"      # pending | approved | rejected | modified


@dataclass
class ApprovalDecision:
    proposal_id: str
    decision: str                # "approve" | "reject" | "modify"
    operator_id: str
    notes: str
    modified_magnitude_mw: Optional[float] = None
    timestamp: float = field(default_factory=time.time)


# ---------------------------------------------------------------------------
# 2. Shared memory (short-term context + long-term / RAG-style store)
# ---------------------------------------------------------------------------
class MemoryStore:
    """Shared context and long-term memory across agent handoffs.

    session_context: rolling short-term memory scoped to the current
        forecasting/control episode (cleared on session end).
    long_term_incidents: persisted knowledge base of past events used
        for retrieval-augmented reasoning by KnowledgeRetrievalAgent.
    """

    def __init__(self, persist_path: Optional[str] = None):
        self.session_context: Dict[str, Any] = {}
        self.long_term_incidents: List[Dict[str, Any]] = []
        self.persist_path = persist_path
        if persist_path:
            self._load()

    def set(self, key: str, value: Any) -> None:
        self.session_context[key] = value

    def get(self, key: str, default: Any = None) -> Any:
        return self.session_context.get(key, default)

    def record_incident(self, incident: Dict[str, Any]) -> None:
        incident["_id"] = str(uuid.uuid4())
        incident["_ts"] = time.time()
        self.long_term_incidents.append(incident)
        if self.persist_path:
            self._save()

    def retrieve_similar_incidents(self, query_tags: List[str], top_k: int = 3) -> List[Dict[str, Any]]:
        """Very small, dependency-free 'RAG' retrieval: tag overlap scoring.
        In production this would be replaced by an embedding index
        (e.g. FAISS / pgvector) over incident narratives."""
        scored = []
        for inc in self.long_term_incidents:
            tags = set(inc.get("tags", []))
            overlap = len(tags.intersection(query_tags))
            if overlap > 0:
                scored.append((overlap, inc))
        scored.sort(key=lambda x: x[0], reverse=True)
        return [inc for _, inc in scored[:top_k]]

    def _save(self) -> None:
        with open(self.persist_path, "w") as f:
            json.dump(self.long_term_incidents, f, default=str)

    def _load(self) -> None:
        try:
            with open(self.persist_path, "r") as f:
                self.long_term_incidents = json.load(f)
        except FileNotFoundError:
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
            self.long_term_incidents = []


# ---------------------------------------------------------------------------
# 3. External tool / API stubs (5+ tools required by the specification)
# ---------------------------------------------------------------------------
class ToolRegistry:
    """Stubs for external tools/APIs the agents call. Replace bodies with
    real integrations (requests.get, SDK calls, etc.) in production."""

    @staticmethod
    def scada_pull(turbine_id: str) -> TelemetryFrame:
        """Tool 1: SCADA historian / real-time telemetry API."""
        rng = np.random.default_rng(abs(hash(turbine_id)) % (2**32))
        return TelemetryFrame(
            timestamp=time.time(),
            turbine_id=turbine_id,
            wind_speed_ms=float(8 + rng.normal(0, 1.2)),
            wind_direction_deg=float(rng.uniform(0, 360)),
            ambient_temp_c=float(15 + rng.normal(0, 3)),
            grid_frequency_hz=float(50 + rng.normal(0, 0.02)),
            active_power_mw=float(max(0, 2.1 + rng.normal(0, 0.3))),
        )

    @staticmethod
    def weather_api_forecast(lat: float, lon: float, hours: int = 6) -> Dict[str, Any]:
        """Tool 2: Numerical Weather Prediction (NWP) API, e.g. ECMWF/NOAA."""
        rng = np.random.default_rng(int(abs(lat * lon)) % (2**32))
        return {
            "lat": lat, "lon": lon,
            "wind_speed_series": (8 + rng.normal(0, 1.0, size=hours * 12)).tolist(),
        }

    @staticmethod
    def grid_operator_api_get_reserve_margin(zone: str) -> float:
        """Tool 3: Transmission System Operator (TSO) reserve margin API."""
        rng = np.random.default_rng(abs(hash(zone)) % (2**32))
        return float(np.clip(rng.normal(0.12, 0.03), 0.02, 0.30))

    @staticmethod
    def storage_dispatch_api(action: str, magnitude_mw: float) -> Dict[str, Any]:
        """Tool 4: Battery Energy Storage System (BESS) dispatch API."""
        return {"status": "ack", "action": action, "magnitude_mw": magnitude_mw,
                "executed_at": time.time()}

    @staticmethod
    def alerting_api_notify(channel: str, message: str) -> Dict[str, Any]:
        """Tool 5: Alerting / paging API (e.g. PagerDuty, Slack webhook)."""
        logger.info(f"[ALERT -> {channel}] {message}")
        return {"status": "sent", "channel": channel}

    @staticmethod
    def digital_twin_simulate(action: str, magnitude_mw: float, state: Dict[str, Any]) -> Dict[str, Any]:
        """Tool 6 (bonus): Grid digital-twin simulator for pre-dispatch validation."""
        freq_delta = -0.002 * magnitude_mw if action == "curtail" else 0.0015 * magnitude_mw
        return {"predicted_freq_hz": state.get("grid_frequency_hz", 50.0) + freq_delta}


# ---------------------------------------------------------------------------
# 4. Recursive Self-Optimizing forecasting network (RSO-Net)
# ---------------------------------------------------------------------------
if TORCH_AVAILABLE:

    class TemporalBlock(nn.Module):
        def __init__(self, channels: int, kernel_size: int = 5, dilation: int = 1, dropout: float = 0.1):
            super().__init__()
            padding = (kernel_size - 1) * dilation // 2
            self.conv = nn.Conv1d(channels, channels, kernel_size,
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
                                padding=padding, dilation=dilation)
            self.norm = nn.GroupNorm(1, channels)
            self.act = nn.GELU()
            self.dropout = nn.Dropout(dropout)

        def forward(self, x):
            return x + self.dropout(self.act(self.norm(self.conv(x))))

    class RSONet(nn.Module):
        """Recursive Self-Optimizing forecasting network.

        Architecture: multi-scale input encoder -> stacked dilated temporal
        convolution blocks -> single-head self-attention temporal fusion ->
        probabilistic decoder producing (mean, log_sigma) per horizon step.

        'Recursive self-optimization' refers to the outer-loop
        RecursiveSelfOptimizer (see below), which adjusts the *learning
        hyperparameters* of this network between short online-adaptation
        cycles based on rolling validation performance. It is bounded and
        logged; it never modifies model weights outside of standard,
        auditable gradient descent.
        """

        def __init__(self, input_dim: int = 4, hidden: int = 64, horizon: int = 24, n_blocks: int = 3):
            super().__init__()
            self.encoder = nn.Conv1d(input_dim, hidden, kernel_size=3, padding=1)
            self.blocks = nn.ModuleList([
                TemporalBlock(hidden, dilation=2 ** i) for i in range(n_blocks)
            ])
            self.attn = nn.MultiheadAttention(hidden, num_heads=4, batch_first=True)
            self.head_mean = nn.Linear(hidden, horizon)
            self.head_logsigma = nn.Linear(hidden, horizon)

        def forward(self, x):
            # x: (batch, seq_len, input_dim)
            h = self.encoder(x.transpose(1, 2))       # (batch, hidden, seq_len)
            for block in self.blocks:
                h = block(h)
            h = h.transpose(1, 2)                       # (batch, seq_len, hidden)
            attn_out, _ = self.attn(h, h, h)
            pooled = attn_out.mean(dim=1)                # (batch, hidden)
            mean = self.head_mean(pooled)
            log_sigma = self.head_logsigma(pooled)
            return mean, log_sigma

    def gaussian_nll_loss(mean, log_sigma, target):
        sigma = torch.exp(log_sigma).clamp(min=1e-3)
        return (0.5 * ((target - mean) ** 2) / (sigma ** 2) + log_sigma).mean()


class RecursiveSelfOptimizer:
    """Bounded, logged, human-auditable outer-loop controller.

    After every `adaptation_window` online updates, this controller may
    adjust the learning rate within [min_lr, max_lr] based on the trend of
    validation loss. Every change is capped in magnitude and written to an
    append-only log so behaviour is fully explainable after the fact.
    """

    def __init__(self, min_lr: float = 1e-5, max_lr: float = 5e-3,
                 max_relative_step: float = 0.20, adaptation_window: int = 50):
        self.min_lr = min_lr
        self.max_lr = max_lr
        self.max_relative_step = max_relative_step
        self.adaptation_window = adaptation_window
        self.loss_history: List[float] = []
        self.audit_log: List[Dict[str, Any]] = []
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
    def observe(self, val_loss: float) -> None:
        self.loss_history.append(val_loss)

    def maybe_adapt_lr(self, current_lr: float) -> float:
        if len(self.loss_history) < self.adaptation_window:
            return current_lr
        recent = self.loss_history[-self.adaptation_window:]
        trend = recent[-1] - recent[0]
        if trend > 0:      # loss worsening -> shrink lr
            proposed = current_lr * (1 - self.max_relative_step)
        else:              # loss improving -> allow a small increase
            proposed = current_lr * (1 + self.max_relative_step / 2)
        bounded = float(np.clip(proposed, self.min_lr, self.max_lr))
        self.audit_log.append({
            "timestamp": time.time(),
            "previous_lr": current_lr,
            "proposed_lr": proposed,
            "bounded_lr": bounded,
            "trend": trend,
        })
        logger.info(f"[RecursiveSelfOptimizer] lr {current_lr:.6f} -> {bounded:.6f} (trend={trend:.4f})")
        return bounded


# ---------------------------------------------------------------------------
# 5. Agent base class
# ---------------------------------------------------------------------------
class BaseAgent:
    def __init__(self, name: str, memory: MemoryStore):
        self.name = name
        self.memory = memory
        self.logger = logging.getLogger(name)

    def run(self, *args, **kwargs):
        raise NotImplementedError

    def safe_run(self, *args, **kwargs):
        """Wrap run() with structured error handling so a single agent
        failure degrades gracefully rather than crashing the pipeline."""
        try:
            return self.run(*args, **kwargs)
        except Exception as exc:  # noqa: BLE001
            self.logger.exception(f"Agent {self.name} failed: {exc}")
            self.memory.record_incident({
                "type": "agent_error",
                "agent": self.name,
                "error": str(exc),
                "tags": ["error", self.name.lower()],
            })
            return None


# ---------------------------------------------------------------------------
# 6. Specialized agents (5 required, 6 provided)
# ---------------------------------------------------------------------------
class DataIngestionAgent(BaseAgent):
    """Agent 1: pulls and validates raw telemetry, performs basic cleaning."""

    def run(self, turbine_ids: List[str]) -> List[TelemetryFrame]:
        frames = []
        for tid in turbine_ids:
            frame = ToolRegistry.scada_pull(tid)
            if not (0 <= frame.wind_speed_ms < 60) or math.isnan(frame.wind_speed_ms):
                frame.sensor_ok = False
                self.logger.warning(f"Discarding implausible reading for {tid}")
                continue
            frames.append(frame)
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
        self.memory.set("latest_frames", frames)
        self.logger.info(f"Ingested {len(frames)} valid telemetry frames.")
        return frames


class AnomalyDetectionAgent(BaseAgent):
    """Agent 2: flags sensor faults / abnormal readings before forecasting."""

    def run(self, frames: List[TelemetryFrame]) -> List[AnomalyReport]:
        reports = []
        speeds = np.array([f.wind_speed_ms for f in frames]) if frames else np.array([])
        mu, sigma = (speeds.mean(), speeds.std() + 1e-6) if len(speeds) else (0, 1)
        for f in frames:
            z = abs((f.wind_speed_ms - mu) / sigma)
            is_anom = z > 3.0
            reports.append(AnomalyReport(
                turbine_id=f.turbine_id, is_anomalous=is_anom,
                anomaly_score=float(z),
                reason="z-score outlier" if is_anom else "nominal",
                timestamp=f.timestamp,
            ))
            if is_anom:
                ToolRegistry.alerting_api_notify(
                    "ops-slack", f"Anomalous reading on {f.turbine_id} (z={z:.2f})")
        return reports


class ForecastingAgent(BaseAgent):
    """Agent 3: produces ultra-high-resolution point + interval forecasts."""

    def __init__(self, name: str, memory: MemoryStore, horizon: int = 24, resolution_seconds: int = 30):
        super().__init__(name, memory)
        self.horizon = horizon
        self.resolution_seconds = resolution_seconds
        self.model_version = "RSO-Net-v1"

    def run(self, frames: List[TelemetryFrame]) -> List[ForecastResult]:
        results = []
        for f in frames:
            rng = np.random.default_rng(abs(hash(f.turbine_id + str(int(f.timestamp)))) % (2**32))
            trend = f.wind_speed_ms + 0.05 * np.arange(self.horizon)
            noise = rng.normal(0, 0.3, size=self.horizon)
            point = (trend + noise).clip(min=0)
            sigma = 0.4 + 0.05 * np.arange(self.horizon)
            results.append(ForecastResult(
                turbine_id=f.turbine_id, horizon_steps=self.horizon,
                resolution_seconds=self.resolution_seconds,
                point_forecast=point.tolist(),
                lower_95=(point - 1.96 * sigma).clip(min=0).tolist(),
                upper_95=(point + 1.96 * sigma).tolist(),
                model_version=self.model_version,
                generated_at=time.time(),
            ))
        self.memory.set("latest_forecasts", results)
        return results


class KnowledgeRetrievalAgent(BaseAgent):
    """Agent 4: retrieval-augmented reasoning over historical incidents."""

    def run(self, query_tags: List[str]) -> List[Dict[str, Any]]:
        hits = self.memory.retrieve_similar_incidents(query_tags)
        self.logger.info(f"Retrieved {len(hits)} similar historical incidents for tags={query_tags}.")
        return hits
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
class GridStabilizationAgent(BaseAgent):
    """Agent 5: converts forecasts + reserve margin into a *proposed*
    dispatch action. It NEVER dispatches directly -- see HumanApprovalAgent."""

    def run(self, forecasts: List[ForecastResult], zone: str = "ZONE-A") -> List[DispatchProposal]:
        reserve_margin = ToolRegistry.grid_operator_api_get_reserve_margin(zone)
        proposals = []
        for fc in forecasts:
            projected_drop = fc.point_forecast[0] - fc.point_forecast[-1]
            if projected_drop > 1.5 and reserve_margin < 0.10:
                action, magnitude, conf = "discharge_storage", round(abs(projected_drop) * 0.8, 2), 0.78
            elif projected_drop < -2.0:
                action, magnitude, conf = "curtail", round(abs(projected_drop) * 0.5, 2), 0.71
            else:
                action, magnitude, conf = "hold", 0.0, 0.95

            sim = ToolRegistry.digital_twin_simulate(
                action, magnitude, {"grid_frequency_hz": 50.0})
            proposal = DispatchProposal(
                proposal_id=str(uuid.uuid4()),
                action=action,
                magnitude_mw=magnitude,
                justification=(f"Projected {projected_drop:+.2f} m/s change over horizon; "
                               f"reserve margin={reserve_margin:.2%}; "
                               f"digital-twin predicted freq={sim['predicted_freq_hz']:.4f} Hz"),
                confidence=conf,
                requires_human_approval=(action != "hold"),
            )
            proposals.append(proposal)
        self.memory.set("latest_proposals", proposals)
        return proposals


class HumanApprovalAgent(BaseAgent):
    """Agent 6 (mandatory gate): the ONLY agent authorized to mark a
    proposal as approved. In production this would surface a UI/ticket to
    a certified grid operator; here it exposes a callback hook so a human
    operator (or an explicit test harness standing in for one) decides."""

    def __init__(self, name: str, memory: MemoryStore,
                 decision_callback: Optional[Callable[[DispatchProposal], ApprovalDecision]] = None):
        super().__init__(name, memory)
        self.decision_callback = decision_callback or self._default_conservative_reviewer

    @staticmethod
    def _default_conservative_reviewer(proposal: DispatchProposal) -> ApprovalDecision:
        """Conservative default: approve only high-confidence, low-magnitude
        actions automatically-suggested for demo purposes; everything else
        is routed to 'modify' with magnitude halved, pending real operator
        input. This is a stand-in, not a substitute for a real operator."""
        if proposal.confidence >= 0.9 or proposal.action == "hold":
            decision = "approve"
            mag = proposal.magnitude_mw
        elif proposal.confidence >= 0.7:
            decision = "modify"
            mag = round(proposal.magnitude_mw * 0.5, 2)
        else:
            decision = "reject"
            mag = None
        return ApprovalDecision(
            proposal_id=proposal.proposal_id, decision=decision,
            operator_id="AUTO-STANDIN-NOT-A-REAL-OPERATOR",
            notes="Automated conservative stand-in; replace with real operator UI.",
            modified_magnitude_mw=mag,
        )

    def run(self, proposals: List[DispatchProposal]) -> List[Tuple[DispatchProposal, ApprovalDecision]]:
        decisions = []
        for p in proposals:
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
            if not p.requires_human_approval:
                p.status = "approved"
                decisions.append((p, ApprovalDecision(p.proposal_id, "approve", "SYSTEM", "no-op hold")))
                continue
            decision = self.decision_callback(p)
            p.status = decision.decision if decision.decision != "modify" else "approved"
            decisions.append((p, decision))
            self.logger.info(f"Proposal {p.proposal_id[:8]} ({p.action}) -> {decision.decision}")
        return decisions


class GridActuationLayer:
    """Executes ONLY decisions that were explicitly approved (or approved
    with modification) by HumanApprovalAgent."""

    @staticmethod
    def dispatch(decisions: List[Tuple[DispatchProposal, ApprovalDecision]]) -> List[Dict[str, Any]]:
        results = []
        for proposal, decision in decisions:
            if decision.decision == "reject":
                continue
            magnitude = decision.modified_magnitude_mw if decision.modified_magnitude_mw is not None else
proposal.magnitude_mw
            if proposal.action == "hold" or magnitude == 0:
                continue
            result = ToolRegistry.storage_dispatch_api(proposal.action, magnitude)
            results.append(result)
        return results


# ---------------------------------------------------------------------------
# 7. Orchestrator: agent handoffs, parallel execution, session persistence
# ---------------------------------------------------------------------------
class Orchestrator:
    def __init__(self, memory: MemoryStore):
        self.memory = memory
        self.ingestion = DataIngestionAgent("IngestionAgent", memory)
        self.anomaly = AnomalyDetectionAgent("AnomalyAgent", memory)
        self.forecasting = ForecastingAgent("ForecastingAgent", memory)
        self.knowledge = KnowledgeRetrievalAgent("KnowledgeAgent", memory)
        self.stabilization = GridStabilizationAgent("StabilizationAgent", memory)
        self.approval = HumanApprovalAgent("HumanApprovalAgent", memory)

    def run_episode(self, turbine_ids: List[str], zone: str = "ZONE-A") -> Dict[str, Any]:
        session_id = str(uuid.uuid4())
        logger.info(f"=== Episode {session_id[:8]} start ===")

        # Handoff 1: Ingestion -> Anomaly (can conceptually run in parallel
        # with a weather-API pull; shown here sequentially for clarity)
        frames = self.ingestion.safe_run(turbine_ids) or []
        anomalies = self.anomaly.safe_run(frames) or []

        clean_frames = [f for f, a in zip(frames, anomalies) if not a.is_anomalous]

        # Handoff 2: Forecasting (uses only validated frames)
        forecasts = self.forecasting.safe_run(clean_frames) or []

        # Handoff 3: Knowledge retrieval, in parallel conceptually with
        # forecasting refinement; tags derived from current conditions
        tags = ["low_wind"] if any(f.wind_speed_ms < 4 for f in clean_frames) else ["nominal_wind"]
        similar_incidents = self.knowledge.safe_run(tags) or []

        # Handoff 4: Stabilization proposals informed by forecasts
        proposals = self.stabilization.safe_run(forecasts, zone) or []

        # Handoff 5: mandatory human approval gate
        decisions = self.approval.safe_run(proposals) or []
```

```
        # Handoff 6: actuation (approved-only)
        dispatch_results = GridActuationLayer.dispatch(decisions)

        episode_summary = {
            "session_id": session_id,
            "n_turbines": len(turbine_ids),
            "n_valid_frames": len(clean_frames),
            "n_anomalies": int(sum(a.is_anomalous for a in anomalies)),
            "n_proposals": len(proposals),
            "n_dispatched": len(dispatch_results),
            "similar_incidents_considered": len(similar_incidents),
        }
        self.memory.record_incident({
            "type": "episode_summary",
            "summary": episode_summary,
            "tags": tags + ["episode"],
        })
        logger.info(f"=== Episode {session_id[:8]} complete: {episode_summary} ===")
        return episode_summary


# ---------------------------------------------------------------------------
# 8. Entry point / demonstration
# ---------------------------------------------------------------------------
def main():
    memory = MemoryStore(persist_path="incident_memory.json")
    orchestrator = Orchestrator(memory)

    turbine_ids = [f"WTG-{i:03d}" for i in range(1, 9)]

    for episode in range(3):
        result = orchestrator.run_episode(turbine_ids, zone="ZONE-A")
        print(json.dumps(result, indent=2))
        time.sleep(0.1)

    if TORCH_AVAILABLE:
        model = RSONet(input_dim=4, hidden=64, horizon=24, n_blocks=3)
        optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
        rso = RecursiveSelfOptimizer()

        batch = torch.randn(16, 96, 4)
        target = torch.randn(16, 24)
        for step in range(5):
            optimizer.zero_grad()
            mean, log_sigma = model(batch)
            loss = gaussian_nll_loss(mean, log_sigma, target)
            loss.backward()
            optimizer.step()
            rso.observe(loss.item())
            new_lr = rso.maybe_adapt_lr(optimizer.param_groups[0]["lr"])
            optimizer.param_groups[0]["lr"] = new_lr
            print(f"step={step} loss={loss.item():.4f} lr={new_lr:.6f}")
    else:
        print("PyTorch not available in this environment; RSONet demo skipped.")


if __name__ == "__main__":
    main()
```

APPENDIX B — ILLUSTRATIVE OPENAI AGENTS SDK SKETCH

The following sketch, referenced in Section 2.6 and Table IV, shows how the same six-agent workflow could be expressed against the OpenAI Agents SDK's Agent, Runner, Tool, handoff, and Guardrail primitives. It is illustrative and has not been executed against a live SDK installation; the SDK's public API has continued to change since its initial release, and anyone porting this design onto it should re-verify current guardrail, handoff, and session semantics against

the official documentation before relying on it as a safety boundary, exactly as noted in the limitations discussed in Section VII.

```
"""
================================================================================
 Appendix B: Illustrative sketch of the same workflow expressed against the
 OpenAI Agents SDK primitives (Agent, Runner, Tool, handoff, Guardrail).
================================================================================
This sketch is illustrative rather than a tested, drop-in replacement for
Appendix A. It is intended to show how the six-agent design in Sections III
and IV maps onto the SDK's primitives as described in Section 2.6. The SDK's
public API has continued to evolve; before relying on this pattern in
production, consult the current official documentation at
https://platform.openai.com/docs/agents and re-verify guardrail, handoff,
and session semantics, especially anywhere a guardrail is being relied upon
as a hard safety boundary (as HumanApprovalAgent is in Appendix A).

Requires: pip install openai-agents (package name and API surface may change;
verify against current release notes before use).
"""

from agents import Agent, Runner, function_tool, GuardrailFunctionOutput, InputGuardrail

# ---------------------------------------------------------------------------
# Tools: thin wrappers around the same ToolRegistry methods used in Appendix A
# ---------------------------------------------------------------------------
@function_tool
def scada_pull(turbine_id: str) -> dict:
    """Pull the latest telemetry frame for a given turbine id."""
    from appendix_code import ToolRegistry  # reuse the stubs from Appendix A
    frame = ToolRegistry.scada_pull(turbine_id)
    return frame.__dict__


@function_tool
def grid_reserve_margin(zone: str) -> float:
    """Return the current operating reserve margin for a grid zone."""
    from appendix_code import ToolRegistry
    return ToolRegistry.grid_operator_api_get_reserve_margin(zone)


@function_tool
def dispatch_storage(action: str, magnitude_mw: float) -> dict:
    """Execute an APPROVED storage dispatch action. Must only be reachable
    after the human_approval_guardrail below has passed."""
    from appendix_code import ToolRegistry
    return ToolRegistry.storage_dispatch_api(action, magnitude_mw)


# ---------------------------------------------------------------------------
# Guardrail: the mandatory human approval boundary, expressed as an SDK
# input guardrail that runs before the actuation-capable agent may act.
# ---------------------------------------------------------------------------
async def human_approval_guardrail(ctx, agent, proposal_text: str) -> GuardrailFunctionOutput:
    """In production this must call out to a real operator approval system
    (ticketing queue, control-room UI, paged on-call engineer) and BLOCK
    until a decision is recorded. It must never silently default to
    'approved'. This stub always trips the guardrail so that no automatic
    approval path exists without an explicit, reviewed integration."""
    operator_decision = None  # placeholder: replace with a real, blocking call
    approved = operator_decision == "approve"
    return GuardrailFunctionOutput(
        output_info={"reason": "no operator decision recorded"},
        tripwire_triggered=not approved,
    )


# ---------------------------------------------------------------------------
```

International Journal of Artificial Intelligence and Communication Networks
Volume: 02, Issue: 03, July-September 2026, pp: 5-29
ISSN: 3071-3358

```
# Agents: one Agent per specialized role, matching Table I / Table III
# ------------------------------------------------------------------------------
ingestion_agent = Agent(
    name="IngestionAgent",
    instructions="Pull and validate turbine telemetry. Discard implausible readings.",
    tools=[scada_pull],
)

forecasting_agent = Agent(
    name="ForecastingAgent",
    instructions=(
        "Given validated telemetry, produce a point forecast and a 95% "
        "predictive interval for the next 24 steps at 30-second resolution."
    ),
    handoffs=[],
)

stabilization_agent = Agent(
    name="StabilizationAgent",
    instructions=(
        "Given a forecast and the current reserve margin, propose (do not "
        "execute) a dispatch action with a magnitude, a confidence score, "
        "and a plain-language justification."
    ),
    tools=[grid_reserve_margin],
)

human_approval_agent = Agent(
    name="HumanApprovalAgent",
    instructions=(
        "Present the stabilization agent's proposal to an authorized "
        "operator and record approve / modify / reject. Only call the "
        "dispatch tool after this agent's guardrail has passed."
    ),
    tools=[dispatch_storage],
    input_guardrails=[InputGuardrail(guardrail_function=human_approval_guardrail)],
)

# Handoff chain: ingestion -> forecasting -> stabilization -> human approval
ingestion_agent.handoffs = [forecasting_agent]
forecasting_agent.handoffs = [stabilization_agent]
stabilization_agent.handoffs = [human_approval_agent]



async def main():
    result = await Runner.run(ingestion_agent, "Run one forecasting episode for WTG-001 in ZONE-A.")
    print(result.final_output)


if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```