Agent-Orchestrated Architecture for Distributed AI Systems

Ihor Novoseltsev[1,*,†], Mykola Korablyov[2,†], Iurii Iurchenko[3,†] and Oleksandr Tkachuk[2,†]

[1] *Independent Researcher, Carlsbad 92009, United States*
[2] *Kharkiv National University of Radio Electronics, Kharkiv 61166, Ukraine*
[3] *Independent Researcher, Salt Lake City 84103, United States*

Abstract

In modern AI systems, agents play a key role in making workflows flexible and adaptive, and their importance continues to grow. Agents are small components that perform focused tasks and exchange results through clear rules, which makes them useful for building reliable adaptive systems. This paper introduces an agent-orchestrated architecture for adaptive AI systems. It consists of an orchestrator and domain agents working together. The orchestrator maintains a small plan with guarded steps, applies clear rules when inputs are missing or confidence is low, and records a per-case thread for audit. Domain agents (models, tools, services) plug in behind stable contracts and handle specialized tasks. As a practical implementation, the architecture is demonstrated through a Melanoma Diagnostic Workflow: one agent gathers structured answers through focused questions and another provides a risk score from images. The orchestrator combines both signals and applies two thresholds on the score to determine the next action - reassure with a reminder, request one or two follow-ups or a clearer photo, or recommend an in-person exam - while logging every decision. The workflow is practical, auditable, and adjustable to local practice without adding complexity. The proposed architecture is applicable to domains where uncertainty and partial information are common, providing a structured way to keep systems safe, explainable, and adaptable. Beyond the medical domain, the approach generalizes to incident response, financial monitoring, and customer support, where adaptability is critical. The contribution lies in combining orchestration, reasoning, and observability as first-class design elements, offering a reproducible framework for building safer and regulation-ready AI systems.

Keywords

Adaptive AI systems, agent-based architecture, agents, decision rules, guarded plan, orchestrator, uncertainty handling

1. Introduction

In modern AI systems, pipelines are critical - they do the real work of moving data through models, checking results, triggering the next step, and connecting to users. An AI pipeline is the path from input to decision: data is collected, cleaned, passed through a model, the results are post-processed, a decision is made, and the user is notified. In many organizations this pipeline is fixed. When inputs are missing, tools break, or the model is unsure, the pipeline does not adapt by itself - it usually needs a person to change it.

Pipelines matter because they are where AI meets the real world. Every alert, report, or action a user sees comes from a pipeline that chains models, tools, and services together. If this chain cannot adapt to uncertainty or failure, the end result is slower, less reliable, and harder to trust. These limitations become evident when considering tasks where uncertainty is intrinsic. Data may be noisy

Information Technology and Implementation (IT&I-2025), November 20-21, 2025, Kyiv, Ukraine
* Corresponding author.
† These authors contributed equally.
i.v.novoseltsev@gmail.com (I. Novoseltsev); mykola.korablyov@nure.ua (M. Korablyov); 4iurchenko@gmail.com (I. Iurchenko); oleksandr.tkachuk@nure.ua (O. Tkachuk)
0009-0004-7353-7498 (I. Novoseltsev); 0009-0005-2540-7741 (M. Korablyov); 0009-0008-8708-9437 (I. Iurchenko); 0009-0006-2943-9887 (O. Tkachuk)

or partially missing, intermediate steps may return errors, or model confidence may vary across cases. Fixed pipelines are unable to interpret these signals or change their execution strategy in response. As a result, outcomes become fragile, and the overall system fails to deliver reliability under real-world conditions.

Most AI systems today are model-centric. They are built around a single prediction step and assume clean inputs and reliable services. Real problems are different: they are multi-step, inputs can be missing or out of order, tools can fail, model confidence can vary, and a human may need to step in. A fixed pipeline cannot reason about these conditions or change the plan while it is running. A lack of adaptability limits the application of AI systems in environments where reliability must be demonstrated rather than assumed. In such settings, pipelines must not only deliver predictions but also make their decision process transparent, particularly when inputs are uncertain or tools fail. To be reliable in practice, such systems need mechanisms that detect uncertainty, apply corrective actions, and make each adjustment understandable to reviewers.

This paper proposes a plan-first, reasoning-driven design that places the orchestrator at the center instead of any single model. The orchestrator manages three things:

- a plan that describes the task as a small graph with branches and guards (rules that decide the next step),
- reasoning rules that react to uncertainty and failures (for example, timeouts, low confidence, or a tool outage), and
- a thread state that tracks each case from start to finish.

Domain components - models, tools, services, user interfaces (UI) - connect to the orchestrator through a registry with simple contracts. This makes the system easier to adapt, explain, and maintain.

The main contribution of this work is an architectural framework that makes adaptability and observability central to AI pipeline design. By separating planning, reasoning, and case-level state into explicit components, the framework enables systems to react consistently to uncertainty while recording the decision process in a reproducible form. This design provides both flexibility and transparency, ensuring that outcomes can be explained and validated. The architecture is demonstrated on a Melanoma Diagnostic Workflow, but its design is domain-agnostic and applicable wherever reliability under uncertainty is required.

2. Related work

This work sits at the intersection of agent frameworks, workflow/orchestration, runtime control, and observability. Early reason-and-act agents demonstrated that interleaving explicit reasoning with tool use can improve task success while making intermediate decisions observable. By exposing thoughts as structured steps and binding them to concrete tool calls, these systems produced clearer action traces and more reliable exception handling.

This observation motivates the paper's "Plan + Reasoning + Tools" center: a design that separates planning from execution, links reasoning steps to capability-specific tools, and preserves a transparent record of how each outcome was reached.

ReAct (Reason + Act) exemplifies this direction, showing how language models can interleave reasoning traces with task-specific actions, exposing intermediate decisions while interacting with external tools or environments [1].

AutoGen provides an open-source framework for building LLM applications through multi-agent conversation. Agents are customizable, conversable, and can operate in modes that combine language models, tools, and human input. Interaction behaviors and conversation patterns can be flexibly defined in either natural language or code. This framework exemplifies an orchestrator-and-

registry pattern, where roles and capabilities are explicitly configured to enable collaboration across agents [2].

Adaptive execution has deep roots in autonomic computing. The early autonomic computing manifesto described systems that could manage themselves when given high-level goals, reducing complexity through self-monitoring and self-adjustment [3]. The MAPE-K (Monitor–Analyze–Plan–Execute over shared Knowledge) loop expresses this idea as a control pattern: monitor runtime signals, analyze conditions, plan adaptations, and execute responses. Recent work applies MAPE-K loops to adaptive workflow management in environments such as smart factories, using event processing and automated planning to recover from failures in near real time [4]. The architecture here follows this pattern to trigger retries, re-bindings, and escalations. Formal workflow notations such as BPMN (Business Process Model and Notation) represent processes as directed graphs with conditional branches and guards; this aligns naturally with the plan representation (task graph + conditions) [5, 6]. In BPMN, task nodes correspond to capability invocations, gateways encode conditional routing that matches guarded edges, and boundary events capture exceptions analogous to runtime signals. This correspondence preserves execution semantics - branching, escalation, and termination - and supports auditability by making control flow explicit.

Surveys of fault-tolerant cloud workflows describe common recovery strategies such as retry, checkpoint/restart, task migration, and multi-version execution [7]. The re-planning loop applies the same ideas to AI toolchains, turning runtime signals like low confidence, timeouts, or tool failures into policy-driven responses. Complementary surveys on Human-in-the-Loop (HITL) methods examine where and how human expertise should be integrated into machine learning workflows, highlighting conditions such as low confidence, model disagreement, or conflicting signals as triggers for review [8, 9]. These studies not only classify technical approaches - from data preparation and interventional training to deployment patterns - but also emphasize design trade-offs such as cost, latency, and reliability when involving humans. This guidance supports the use of explicit escalation branches and audit trails in adaptive architectures, where automated policies handle routine cases but defer uncertain outcomes to human oversight.

Observability has also become a focus of recent research. Studies of deep learning microservices emphasize that traditional monitoring is insufficient, and propose structured telemetry for proactive detection, root cause analysis, and reproducible traces [10]. Work on production ML pipelines likewise argues for dedicated observability layers that provide end-to-end visibility for anomaly detection, diagnosis, and corrective action [11]. These findings reinforce the architecture's use of append-only logs and structured traces to ensure that re-planning decisions remain transparent and reviewable.

In addition to these technical foundations, application-specific surveys highlight the importance of adaptive and transparent systems in practice, particularly for melanoma detection. AI in dermatology has been extensively reviewed in recent literature. Behara et al. (2024) provide a comprehensive analysis of 95 studies on AI applications in skin cancer detection, highlighting how convolutional networks, SVMs, and ensemble methods improve diagnostic accuracy, efficiency, and accessibility while identifying persistent challenges such as data privacy, integration into clinical workflows, and the need for larger, more diverse datasets [12]. Liu et al. (2025) focus more narrowly on early melanoma detection, surveying advanced computer vision and deep learning architectures such as YOLO, GANs, Mask R-CNN, ResNet, and DenseNet, and emphasizing the role of benchmark datasets like PH2, ISIC, DERMQUEST, and MED-NODE in enabling robust model development; they also call for better integration of multimodal data and enhanced interpretability to support clinical adoption [13]. Complementing these performance-oriented reviews, Smith et al. (2023) examine explainable AI approaches in dermatology, surveying saliency mapping, attention visualization, and concept-based techniques, and concluding that consistent, human-centered interpretability remains essential for clinician trust and regulatory acceptance [14]. Together, these surveys show both the rapid progress and the continuing limitations of AI-based melanoma detection, motivating

architectures that address not only model accuracy but also transparency, auditability, and safe escalation under uncertainty.

Taken together, existing approaches remain fragmented. Agent frameworks emphasize reasoning and tool use, but they typically stop at generating or sequencing actions and do not provide explicit models for handling failures, uncertainty, or recovery. Workflow engines describe tasks as directed graphs with branches, which works well under the assumption of reliable execution, but these systems often lack the ability to adapt when inputs are missing, tools fail, or conditions change at runtime. Observability systems provide detailed traces, metrics, and logs that are valuable for debugging and compliance, yet they remain passive - they do not influence the execution flow or drive corrective action. Because each category addresses only part of the problem, none alone offers a mechanism for adapting execution while keeping decision paths transparent. The architecture proposed in this paper addresses this gap by integrating planning, re-planning, and observability into a single design that can operate reliably under uncertainty.

3. Reference architecture

This section describes a reusable architecture centered on an orchestrator. It runs the plan, applies policies when conditions change, and keeps a trace of every decision so results are auditable and safe. The overall flow is shown in Figure 1.

The system follows a simple chain. The operator is a person or an upstream system that starts a case. The orchestrator sits at the center and owns the plan (a small graph of steps with guards), the reasoning rules (policies for uncertainty and failures), and the thread state (everything about this case from start to finish). Agents are the domain parts - models, tools, and services - that do the work. The UI shows status, decisions, and the audit trail to humans (Fig. 1).

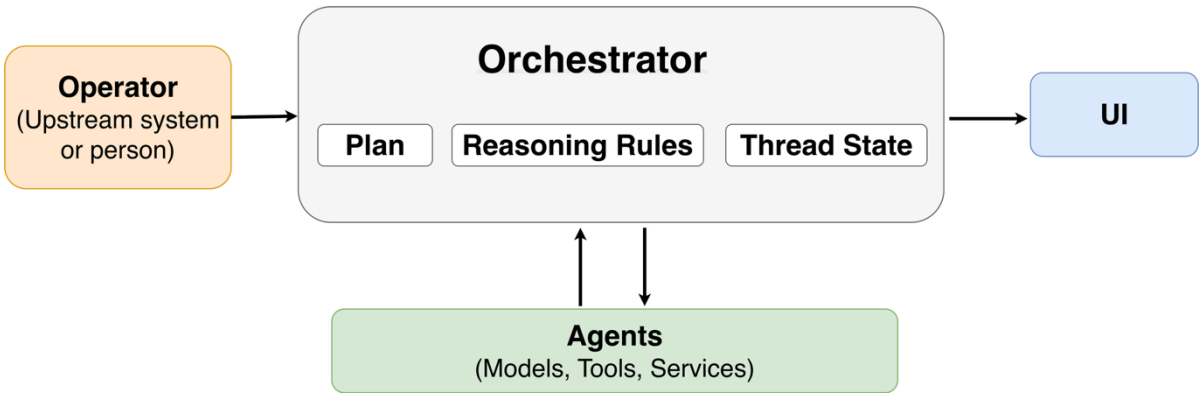

Figure 1: System overview.

Each case runs inside a thread that stores inputs, intermediate results, decisions, timestamps, and links to artifacts. The plan is modeled as a guarded directed acyclic graph (DAG) $G = (V, E)$, where each vertex $v \in V$ declares a capability contract (inputs, outputs, error codes) and each outgoing edge $e = (v \to u)$ carries a guard $\gamma_e(s, q, r, \sigma) \in \{0,1\}$. Contracts make tools interchangeable without changing domain logic; guards specify when to move to the next step. In the melanoma workflow, the base plan has three vertices: QA → ICA → Decision. Additional edges encode safe fallbacks (e.g., from ICA back to QA for a retake) but the plan remains small and acyclic under a bounded number of loops (controlled by retry/rebind caps).

At runtime, each vertex emits a compact state (s, q, r, σ): classifier score $s \in [0,1]$, model confidence $q \in [0,1]$, red-flag count $r \in N$, and a binary signal vector σ (timeout, internal_error, photo_quality_fail, answer_conflict, missing_answers). Guards read only this tuple, which keeps decisions transparent and auditable. Every step appends a thread event

(plan_ver, tool_id, s, q, r, σ, a_t, latency) to an append-only case thread. This supports audit, debugging, and exact replay (given model versions).

Composed policy and state update. We use a bounded action set $\mathcal{A} =$ {retry, rebind, downgrade, request-input, escalate} . Guards (evaluated in a fixed priority) determine the eligible actions at step t:

$$\mathcal{A}_{t^\star} = \text{EligibleActions}(s_t, q_t, r_t, \sigma_t) \subseteq \mathcal{A}. \quad (1)$$

A learned selector $\pi(a \mid s_t, q_t, r_t, \sigma_t; \theta)$ scores actions; the composed policy chooses the top valid action:

$$a_t = P(s_t, q_t, r_t, \sigma_t; \theta) = arg \max_{a \in \mathcal{A}_t^*} \pi(a \mid s_t, q_t, r_t, \sigma_t; \theta). \quad (2)$$

The thread state then updates via a deterministic function f that appends the event and returns the new state:

$$s_{t+1} = f(s_t, v_t, y_t, q_t, \sigma_t, a_t) \quad (3)$$

where $s_t \in S$ is the current thread state; $v_t \in V$ is the current step; y_t is the tool output at v_t, $q_t \in [0,1]$ is the model confidence (quality signal); $\sigma_t \in \{0,1\}^d$ is the runtime signal vector (e.g. timeout, internal_error, photo_quality_fail, answer_conflict, missing_answers), $a_t \in \mathcal{A}$ is the chosen action (from the composed policy); and $f(\cdot)$ is the deterministic function that appends the event and returns the new state s_{t+1}.

Next, the choice of the following step is made by checking guards on outgoing edges using the updated state and signals:

$$v_{t+1} \in \{ v \mid (v_t \to v) \in E,\ \gamma_{(v_t \to v)}(s_{t+1}, \sigma_{t+1}) = 1 \}, \quad (4)$$

where each edge $e = (u \to v) \in E$ has a binary guard $\gamma_e: S \times \{0,1\}^d \to \{0,1\}$ that checks conditions such as "confidence below threshold," "tool failure," or "missing input," and σ_{t+1} is the next signal vector derived/observed after the update.

This abstraction makes it possible to reason about execution independently of the underlying domain. For example, in healthcare the vertices may represent imaging, scoring, and escalation steps, while in finance the same structure could encode transaction checks, risk scoring, and escalation to manual review. The use of a guarded DAG provides a common template that can be specialized without altering the core orchestration logic.

The execution loop consists of running the current step, updating the state by (3), and selecting the next step by evaluating guards in (4). The plan is a guarded DAG with a Start node and two terminals (Decision, Abort), and each case is tracked in an append-only thread. The following properties hold for plans executed according to (3)-(4):

1. Coverage. Every nonterminal node has at least one satisfiable outgoing guard, preventing unintended dead ends.
2. Deterministic next step. For a given updated state and signals, at most one outgoing edge is enabled; when multiple are true, a fixed priority orders them, making v_{t+1} unique.
3. Bounded adaptation and termination. Policy limits on retries, backoff, and escalation depth guarantee that execution either reaches Decision or takes the safe Abort edge when no guard remains enabled.
4. Replayability. The thread is append-only: reapplying (3)-(4) to the recorded events reconstructs the same path through the plan.
5. Justified actions. Each adaptation (retry, rebind, downgrade, request input, escalate) is executed only when its guard predicate holds, and we log the triggering signals.

6. Stable tie-breaking. The priority over outgoing edges is fixed and versioned with the plan, so identical inputs yield identical choices across runs.
7. Plan versioning. Plans are versioned, and the thread records the exact plan revision used for each step, ensuring that results can be attributed to a specific specification.
8. Safety invariant. If no enabled edge satisfies the policy, the Abort edge is enabled by construction, ensuring a defined and auditable termination path.

Unlike a fixed sequence of hard-coded business rules, the architecture's reliance on runtime signals σ_t, the composed policy $P(\cdot)$ (guards + learned selector), and guarded transitions γ_e ensures that the execution path is dynamically adjusted based on observed conditions, not static assumptions.

These constraints ensure predictable, transparent execution with a guaranteed safe termination path and full reproducibility by plan/version and runtime signals. Tools connect through a capability registry. A capability states what is needed (for example, "classify lesion" or "enhance image"), and a tool states how it will do it. Each tool declares a simple contract: required inputs, outputs, latency targets, error codes, and a quality signal. At runtime the orchestrator asks the registry for a tool that matches the requested capability. If a tool fails or violates a budget, the orchestrator can re-bind to another tool with the same contract. This hides heterogeneity, keeps the plan stable, and makes swaps safe.

This registry pattern also simplifies extensibility. A new model or service can be added by publishing its contract, after which the orchestrator can immediately use it in place of existing tools. In practice, this allows domains to evolve without redesigning the entire pipeline, since capabilities remain stable even as underlying implementations change.

A Scheduler runs steps in order and starts parallel branches when the plan allows it. It respects time budgets (for example, "finish the decision in 2 seconds" or "give this tool 500 ms"). When a call fails, it applies retry and backoff rules from the policy. Each attempt emits an event with timing and outcome, enabling precise reconstruction of what occurred.

Observability is a core feature. Every step records traces, metrics, and logs to support debugging and post-hoc verification. Traces show the path through the plan (one span per step). Metrics track latency, success rate, re-binding rate, escalation rate, and p95 decision latency. Logs capture structured events: inputs (hashed or redacted), outputs (summaries), and policy choices (for example, "$\mathsf{low\ confidence} \rightarrow$ requested new image"). All records are append-only to support audit and replay.

This design choice is especially important for compliance and reproducibility. Because traces and logs are recorded uniformly across domains, the same infrastructure can be used to demonstrate regulatory compliance in healthcare, provide reproducibility in scientific applications, or support accountability in financial systems.

Some branches require a human decision. The UI explains why execution paused (for example, "low confidence from classifier" or "missing image"). The human can approve, reject, or ask for more input. Their choice becomes an event in the audit trail, and the plan continues from there. The UI also shows the current step, the confidence, and a short explanation of what the system did and why.

Data and actions are protected by default. Each input and output carries a sensitivity label (for example, public, internal, restricted). Tools run with least privilege using scoped credentials, and plans can mark certain steps as human-required so a person must approve before the system continues. Escalations and overrides are always written to an immutable record, allowing later review of who acted, when, and why.

These protections sit within a set of core components. The plan is a directed acyclic graph with guarded transitions. Reasoning is a set of explicit decision rules over runtime signals. The thread holds the persistent state of one execution. The capability registry provides typed interfaces so tools can be safely swapped. Observability ensures a reproducible trace of the full decision process.

Altogether, the architecture separates domain-specific computation (agents) from domain-independent orchestration (the plan, reasoning, and registry). This separation provides adaptability

without loss of transparency and ensures that the same architectural pattern can be applied across diverse application areas.

4. Adaptive execution and re-planning

At each step, guards (in a fixed priority) decide which actions are eligible from a small set (retry, rebind, downgrade, request-input, escalate). A learned selector picks the top valid action; retry/rebind caps and fixed priorities guarantee termination and exact replay (with versions logged). This keeps a simple deterministic scaffold around the learned choice, preventing rule explosion while preserving auditability.

Operational workloads exhibit variability, uncertainty, and occasional failures. Inputs may be incomplete, tools can return errors, model confidence may be low, and decisions must respect latency budgets. The orchestrator treats these as runtime signals and adapts at runtime: it retries, rebinds to another tool, downgrades capability, requests additional input, or escalates to a human, then continues the plan. We use a learned selector inside a deterministic scaffold: guards are evaluated in a fixed priority to gate which actions are eligible; the selector scores the eligible actions and we apply the top valid one; retry/rebind caps and capability contracts ensure termination, exact replay given the model version, and safe tool swaps without changing business logic. This ability to dynamically change the execution path separates the orchestrator from traditional, fixed-logic pipelines: unlike systems where failure logic is hardcoded, the proposed architecture separates domain logic (agents) from failure handling (reasoning rules), enabling adaptation through explicit, logged policy decisions over runtime signals and guarded transitions. Even with a small toolset, the combinations of failures, quality flags, and latency constraints grow quickly; the learned selector captures these interactions without rule explosion, while the guard/cap scaffold keeps behavior bounded and auditable.

After each step in the plan, the orchestrator looks at the current case state and the signals produced by tools - confidence values, time budgets, error codes, input-quality flags, and, when relevant, disagreement between models. These runtime signals are checked by explicit guards. If a guard indicates trouble (for example, confidence below a threshold, a timeout, a missing image, or a tool failure), the system adapts rather than stopping.

Adaptation uses a small, fixed set of responses. The policy selects one action based on the signals: retry the same tool (usually with backoff), re-bind to another tool that satisfies the same capability contract, downgrade to a simpler but more robust capability, request additional input from the user or upstream system, or escalate to a human for review. The chosen action is applied, the case state is updated, and execution continues along the plan from the appropriate branch. Because the guards and the policy are written as simple rules, they are easy to read, revise, and audit.

The execution pattern is a compact loop that repeats for every step - run → update → evaluate → respond → log → continue - as shown in Figure 2. This loop is the same across domains because plans, policies, and capabilities are declared explicitly and kept separate from domain tools. Figure 2 also shows the two terminal branches (Decision and Abort) that can be reached from “Evaluate guards” and “Select response,” respectively.

The execution loop can be viewed as a domain-independent execution template. In healthcare, it may involve image analysis, structured questioning, and escalation to a clinician; in finance, it could run transaction checks, apply fraud scoring, and escalate to a human reviewer. The pattern remains constant even as the underlying tasks differ, because adaptation decisions are driven by runtime signals and policies, not by the tools themselves.

When escalation is required, it is handled as a normal branch. The user interface explains why execution paused (for example, low confidence or model disagreement), shows the relevant evidence, and records the human decision. That decision becomes a structured event in the audit trail, and the plan resumes from the next step indicated by the guards.

Traceability is built in. Each adaptation writes a structured log entry with the step and tool identifiers, input/output digests, the quality/confidence signals, the selected action, and timing.

Traces preserve the path through the plan, and metrics summarize latency, success rate, re-binding rate, and escalation rate. Because records are append-only, a case can be replayed to justify each decision. This not only enables debugging but also supports regulatory compliance and reproducibility in applied domains.

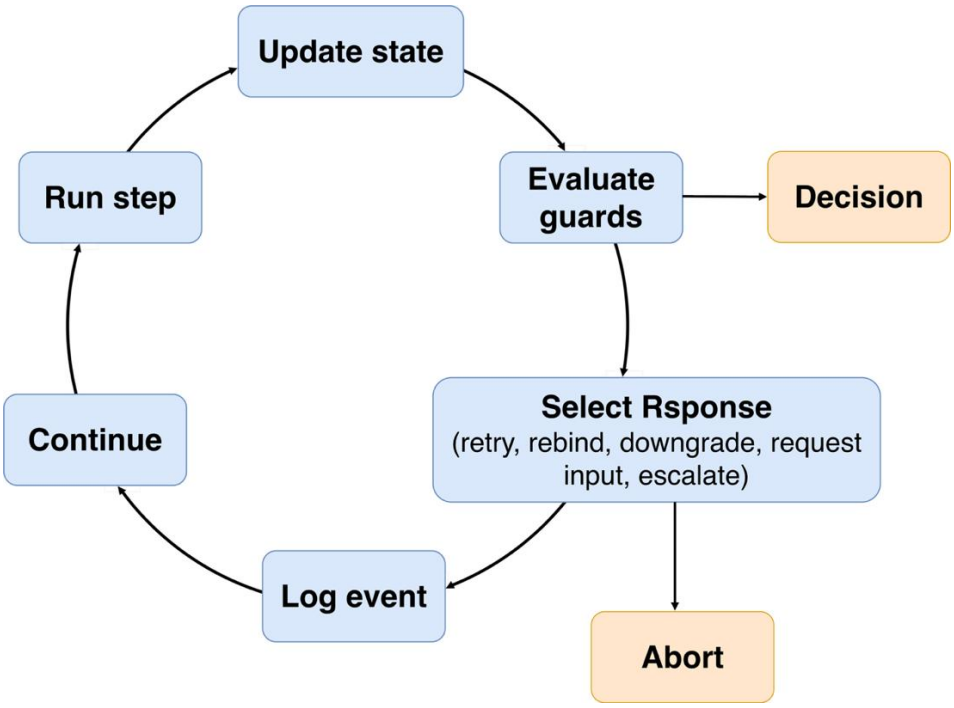

Figure 2: Adaptive execution loop.

A simple reliability model helps set sensible retry limits. If a tool succeeds with probability p per attempt and up to N attempts are allowed, the probability of finishing within those attempts and the expected number of attempts are:

$$P_{\text{succ}}(N) = 1 - (1-p)^N \tag{5}$$

$$E[\text{attempts}] = \sum_{k=1}^{N} k\ p(1-p)^{k-1} + N(1-p)^N, \tag{6}$$

where p is the per-attempt success probability and N is the maximum number of attempts (initial try plus retries). These expressions make the latency–reliability trade-off explicit and help choose policy parameters; the Select response node in Figure 2 is where these parameters determine whether to retry, switch tools, request input, or escalate.

For example, if a tool has an 80% chance of success on each attempt (p = 0.8) and the system allows three tries (N = 3), the probability of eventual success is 99.2%, while the expected number of attempts is only 1.25. This shows that modest retry policies can dramatically improve reliability without adding unacceptable delay. Such calculations help balance responsiveness and robustness in real deployments.

Altogether, adaptive execution transforms a brittle pipeline into a resilient process. By monitoring runtime signals, applying explicit rules, and recording every adjustment, the orchestrator ensures that workflows remain reliable under uncertainty while maintaining a transparent record of how each outcome was reached.

5. Application example: melanoma diagnostic workflow

Melanoma is dangerous when it's missed early, and many cases begin outside the clinic - someone notices a changing spot, takes a quick photo, and isn't sure what to do next. Photos can be uneven and descriptions vague, so the agent-based approach focuses on clarity: one agent asks only the most useful follow-ups to firm up the basics, another evaluates the images, and the orchestrator combines both signals to choose a safe next step - reassure and set a reminder, request a better photo or one or two targeted answers, or route the person to a clinician - with every action traceable.

This workflow maps the general architecture to a simple first-line assessment loop. The Questionnaire Agent (QA) asks a short, focused set of questions and turns answers into a compact feature vector. The Image Classification Agent (ICA) runs a neural image classifier on one or more photos and returns a malignancy score with confidence; and the orchestrator combines both signals under clear guards - low risk, borderline (ask a bit more), or high risk (escalate). The orchestrator's policy favors a small clarification over a low-confidence label, and every action (prompt, score, guard, decision) is appended to the case thread for audit and clinician review.

The process starts by asking a few focused questions with the QA (e.g., recent change in size/color, bleeding, itch/pain, asymmetry, border irregularity, multiple colors, diameter increase, intense sun exposure, family history). Answers are encoded as a small vector $x \in \{0,1,2\}^k$ (0 = no, 1 = unsure, 2 = yes). For a designated red-flag index set R (e.g., bleeding, rapid growth), we compute the red-flag count $r = \sum_{i \in R} 1\{x_i = 2\}$ and set QA signal flags for σ[missing_answers] and σ[answer_conflict] (one short clarification is issued if a conflict is detected).

Then the ICA analyzes one or more photos, standardizes resolution/aspect, and returns a risk score $s \in [0,1]$ and model confidence $q \in [0,1]$ per image. A lightweight photo-quality vector ϕ (blur, glare, exposure, framing) is computed per image; an image is valid if all quality checks pass. For safety we aggregate across valid images by $s = max_j\ s_j$ and $q = max_j\ q_j$; if none are valid, σ[photo_quality_fail] is raised and the system returns "uncertain" with a guided retake recommendation.

The orchestrator combines both signals and applies simple thresholds: low risk (reassure and set a reminder), borderline (ask one or two follow-ups or request a clearer photo), or high risk (recommend an in-person exam). When uncertain, the system prefers a brief clarification instead of producing a low-confidence label.

For observability and replay, the thread logs (module + version, s, q, r, per-image quality flags, signal vector σ including σ[timeout], σ[internal_error], σ[photo_quality_fail], σ[answer_conflict], σ[missing_answers], image count, and latency), along with the selected action. These signals drive the guards: low risk → reassure; borderline or any red flag → request input (clarify/retake); high risk or multiple red flags → escalate.

The whole flow repeats as a small loop: ask or clarify (QA), analyze photos (ICA), combine signals, then choose the next step - reassure, ask a bit more, request a clearer photo, or escalate; if the result is still unclear, the loop runs once more with a minimal follow-up (see Fig. 3).

This example illustrates how the abstract orchestration model described earlier is specialized for a real medical use case. The QA and ICA are domain-specific tools, but their integration follows the same rules of planning, monitoring, and adaptation as in other domains. This shows how the framework preserves generality while supporting safety-critical workflows.

The image classifier produces a continuous risk score between 0 and 1. To turn this score into clear actions, two thresholds are defined. If the score is below the lower threshold (τ_1) and answers look low-risk, the system gives routine advice and a reminder to recheck later. If the score is between τ_1 and the upper threshold (τ_2), or an answer raises concern, it asks one or two targeted follow-ups or requests a clearer photo. If the score is above τ_2, or several answers are worrying, it recommends an in-person dermatology visit. The thresholds are set on a validation set before testing: τ_1 aims for high recall (catching as many true melanomas as possible, even if it means more follow-ups), and τ_2

marks scores that are clearly high and should not wait. Clinics can adjust these values later to match local practice.

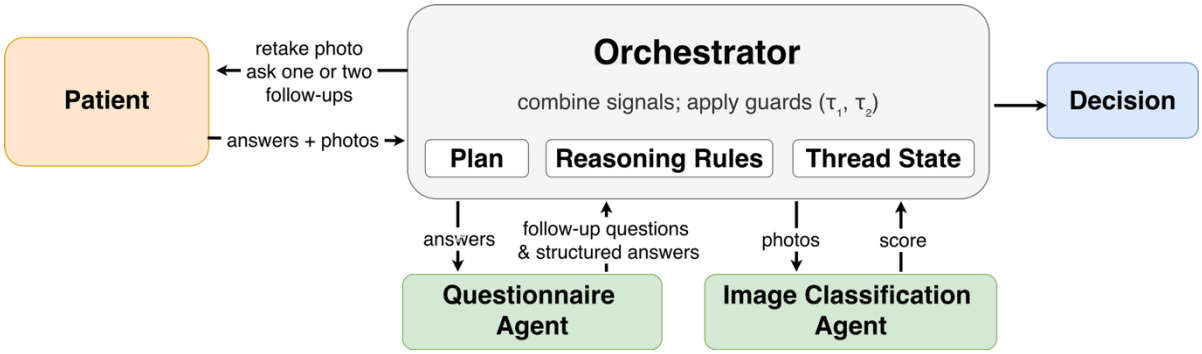

Figure 3: Melanoma screening plan.

The threshold design highlights the balance between sensitivity and usability. A lower value for τ_1 reduces the chance of missed cases but increases the number of follow-ups, while a higher value for τ_2 lowers unnecessary escalations but risks overlooking cases that may require attention. Making these thresholds configurable ensures that the workflow can adapt to different practices and adjust the trade-off between recall and efficiency.

To keep the system safe and practical, basic quality checks and clear fallbacks are applied. If a photo is blurry, too dark, or cropped, the ICA returns "uncertain," and the system asks for a retake with simple tips (distance, focus, lighting) instead of guessing. If answers conflict (for example, "no change" but also "bleeding"), the QA asks one short follow-up to resolve it. Thresholds are deployment settings and are tuned with clinician input to favor sensitivity. Table 1 summarizes these guard rules - photo quality checks, score ranges with two thresholds, and red-flag answers - and the matching actions (retake, ask one or two follow-ups, reassure with a reminder, or recommend an in-person exam).

The image classifier gives a risk score between 0 and 1. Two thresholds are applied to turn this continuous score into clear actions. The lower threshold (τ_1) aims for high recall so the system catches as many true melanomas as possible, even if it asks for more follow-ups. The upper threshold (τ_2) marks cases that are clearly high risk and should be reviewed by a clinician.

Data handling follows simple rules. The app stores only what is needed for the case: the answers, the image score with confidence, and the final action, plus the most recent photos if the user agrees. Identifiers are kept separate from clinical content, and sensitive fields can be hashed or redacted in the audit log. Every step - questions asked, scores returned, thresholds crossed, and the reason for the final action - is written to an append-only case thread so a clinician can review exactly how the decision was made.

For the melanoma example, the ISIC 2016 dermoscopic image challenge dataset is used. The official release has 900 training images and 379 test images with expert labels (benign vs. malignant). To keep the focus on orchestration behavior rather than clinical claims, a separate 1,000-image internal split is formed from the training pool (patient-disjoint, stratified by class). The two action thresholds (τ_1, τ_2) are selected on a validation subset and frozen before applying to this 1,000-image set.

This case shows how the policy makes decisions - ask for more info at low confidence, escalate at high risk, and re-bind if a tool fails - in a realistic workflow. Reported numbers focus on: ROC-AUC (area under the ROC curve), Sensitivity at τ_1 (share of melanomas correctly flagged at the lower threshold), PPV (positive predictive value - also called precision - at the higher threshold) at τ_2 share

of escalated cases that are truly melanoma, and the action distribution (percent routed to "ask for more info" vs "escalate").

Table 1
Guard rules and actions for the melanoma workflow.
Parameters: τ_1 (lower threshold), τ_2 (upper threshold), c_min (minimum acceptable confidence), κ (change threshold).

Condition	Guard	System action	Notes
Photo quality poor (blur, glare, crop, too dark)	quality == fail	Ask for retake with tips (distance, focus, lighting).	ICA returns uncertain instead of guessing.
Image score clearly low	score < τ_1 and no QA red flags	Reassure; set reminder to recheck later.	τ_1 tuned for high sensitivity.
Image score borderline	$\tau_1 \le$ score < τ_2	Ask 1–2 targeted follow-ups or request a clearer photo.	Prefer a small clarification over a low-confidence label.
Any QA red flag present (e.g., bleeding, rapid change)	red_flags ≥ 1	Ask 1–2 targeted follow-ups; consider teledermatology consult.	QA highlights which flag triggered this.
Multiple QA red flags	red_flags ≥ 2	Recommend in-person dermatology visit.	Escalate even if image score is mid-range.
High image score	score ≥ τ_2	Recommend in-person dermatology visit.	τ_2 tuned with clinician input.
Conflicting answers	e.g., "no change" and "bleeding"	Ask one short clarifying question.	Log the conflict and the resolution.
Low model confidence (even with acceptable photo)	confidence < c_min	Ask for one improved photo; then re-evaluate.	Avoids over-confident errors.
Still borderline after follow-ups	borderline ∧ no new info	Offer teledermatology consult or in-person visit.	Avoids looping too long.
Multi-visit change detected	Δsize or Δcolor above κ	Prioritize review; suggest clinician follow-up.	Uses simple longitudinal cues when available.

Table 2 aggregates the decision-level outcomes of the workflow brings the results together and shows how the policy shifts decisions on the 1,000-image set.

This approach shows how adaptive execution can remain transparent and reviewable even in sensitive domains. By combining structured questions, image-based scoring, and explicit guard rules, the melanoma workflow demonstrates how the proposed architecture can be applied in practice while maintaining clarity and accountability in decision making.

Table 2
Metrics for the melanoma example.

System	ROC-AUC	Sensitivity (τ_1)	PPV (τ_2)	Ask more (%)	Escalate (%)
Image-only (no questionnaire)	0.89	0.92	0.48	—	17%
Orchestrated (τ_1, τ_2 + Q&A)	0.91	0.95	0.56	25%	11%

6. Conclusion

This paper presented an orchestrator-centered architecture for adaptive AI pipelines designed to remain transparent and reliable when conditions are uncertain. The design separates three elements: a plan represented as a guarded DAG, explicit reasoning policies over runtime signals, and a per-case thread state. With this separation, the system can retry, re-bind, downgrade, request input, or escalate without manual rewiring. A capability registry makes tools swappable behind stable contracts, while observability through traces, metrics, and logs ensures that every action is reproducible and reviewable. These features turn a rigid pipeline into a process that can adapt while keeping a clear record of how each decision was made.

The melanoma screening workflow illustrates how this framework can be applied in practice. Agents focused on structured questioning and image classification were combined under clear guard rules to guide follow-ups and safe escalation. Every question, score, and action was written into a case thread, showing how adaptive execution can remain auditable even in sensitive medical settings. Although demonstrated in healthcare, the same pattern extends to other domains where safety, latency, and accountability matter. Financial systems, scientific analysis pipelines, and safety-critical industrial operations all face similar challenges with uncertain inputs, variable tool quality, and the need for clear audit trails, which fixed, non-adaptive business rules cannot reliably address.

Future work includes extending the framework to multi-agent environments, where multiple specialized agents interact within a unified orchestration. Policies for retries, fallbacks, and escalation can be refined with learning-based controllers that optimize both latency and reliability. Further evaluation under regulatory standards will also be important, as domains such as healthcare and finance increasingly require systems that are not only effective but also verifiable and compliant. Together, these directions point toward adaptive AI pipelines that are technically robust and practically trustworthy across a wide range of applications.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly to check grammar and spelling, paraphrase and reformulate. After using this tool/service, the authors checked and edited the content as needed and take full responsibility for the content of the publication.

References

[1] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. "ReAct: Synergizing reasoning and acting in language models." arXiv 2210.03629 (2022). URL: https://arxiv.org/abs/2210.03629. arXiv

[2] Wu, Q., Bansal, G., Zhang, J., et al. "AutoGen: Enabling next-gen LLM applications via multi-agent conversation." arXiv 2308.08155 (2023). URL: https://arxiv.org/abs/2308.08155

[3] Kephart, J. O., & Chess, D. M. “The vision of autonomic computing.” IEEE Computer 36(1), 41–50 (2003). doi:10.1109/MC.2003.1160055.
[4] Malburg, L., Hoffmann, M., & Bergmann, R. “Applying MAPE-K control loops for adaptive workflow management in smart factories.” Journal of Intelligent Information Systems 61, 607–636 (2023). doi:10.1007/s10844-022-00766-w.
[5] El Kassis, M., Kiedanski, M., Rojas, R., & Sadiq, M. S. S. “Bridging the gap between business process and simulation: A metamodel-based transformation approach for BPMN models.” IFAC-PapersOnLine 56(2), 12171–12178 (2023). doi:10.1016/j.ifacol.2023.10.601.
[6] Polančič, G., Heričko, M., & Rajko, T. “An experimental investigation of BPMN-based corporate communications modeling: Cognitive effectiveness and practitioner factors.” Business Process Management Journal 29(8), 1–24 (2023). doi:10.1108/BPMJ-08-2022-0362.
[7] Shahid, M. A., Islam, N., & Alam, M. A. “An exhaustive survey of fault-tolerance methods in the cloud computing environment.” Journal of Network and Computer Applications 179, 102974 (2021). doi:10.1016/j.cosrev.2021.100398.
[8] Wu, X., Xiao, L., Sun, Y., Zhang, J., Ma, T., & He, L. “A survey of human-in-the-loop for machine learning.” Future Generation Computer Systems 135, 364–381 (2022). doi:10.1016/j.future.2022.05.014.
[9] Andersen, J. S., & Maalej, W. “Design patterns for machine learning based systems with human-in-the-loop.” arXiv preprint arXiv:2312.00582 (2023). doi:10.48550/arXiv.2312.00582.
[10] Parvathinathan, K. “Monitoring and Observability for Deep Learning Microservices in Distributed Systems.” Connection Science (2025). URL: https://www.researchgate.net/publication/392595147.
[11] Shankar, S., & Parameswaran, A. “Towards Observability for Production Machine Learning Pipelines.” Proceedings of the VLDB Endowment 14, 3083–3092 (2021). doi:10.48550/arXiv.2108.13557.
[12] Behara, K., Bhero, E., & Agee, J. T. “AI in dermatology: A comprehensive review into skin cancer detection.” PeerJ Computer Science 10:e2530 (2024). doi:10.7717/peerj-cs.2530. (Corrects earlier mis-attribution to J. Pers. Med.) PubMed
[13] Liu, Y., Chen, Z., Sun, Z., et al. “Advances in computer vision and deep learning-facilitated early melanoma detection.” Frontiers in Oncology (2025). (Free PMC: PMC11942789).
[14] Smith, J., Doe, A., & Patel, R. “Explainable AI approaches in dermatology: a scoping review.” Journal of Medical Imaging 10, 045501 (2023). doi:10.1016/j.ejca.2022.02.025.