VANTIQ Developers Guide Series - Introduction to Intelligence in Vantiq Applications: Knowledge-based Rules, Predictive, Generative, and Agentic AI in Vantiq

Version 2.2

09/06/2026

Table of Contents

List of Figures

1. Introduction

Intelligent, real-time applications rarely rely on a single form of intelligence. Instead, they combine multiple complementary approaches, each suited to different kinds of decisions, uncertainty, and operational constraints. In Vantiq applications, intelligence spans a spectrum—from knowledge-based rules that encode what is already known, through predictive models that infer likely outcomes, to generative and agentic capabilities that reason, plan, and act independently.

At one end of this spectrum are **knowledge-based rules**, which capture explicit domain expertise, policies, and constraints. These rules provide speed, predictability, and explainability, making them essential for enforcing known conditions. These rules are implemented by a developer using existing domain knowledge.

Predictive AI extends intelligence beyond predefined logic by using models trained on historical data to discover hidden patterns, relationships, correlations, and to learn statistical relationships and complex patterns directly from data. Rather than relying on explicitly programmed rules, the model optimises its parameters to identify meaningful signals, suppress noise, and capture nonlinear dependencies within the data.

Predictive models enable applications to detect anomalies that are uncertain or are “shades of grey,” and to identify complex patterns that emerge across multiple signals—even when no explicit rule has been violated. Because these models are data-driven, they can be retrained with new information, continuously refining their understanding of patterns and relationships and improving performance over time.

Generative AI enables systems to synthesise information and generate meaningful outputs at runtime. Rather than selecting from predefined responses, generative models construct explanations, summaries, recommendations, and plans based on context, intent, and retrieved knowledge. This form of intelligence is especially valuable for human interaction, situational understanding, and reducing cognitive load in complex, information-rich environments.

Agentic AI refers to AI systems that don’t just synthesise information but act with a degree of autonomy to achieve goals—planning steps, making decisions, using tools, and adapting based on results. Generative AI serves as the agent’s reasoning and creation engine, generating plans, code, text, and analyses that guide each action. Many agentic AI systems are also built as multi-agent architectures, where multiple specialised agents collaborate—such as a planner, executor, critic, or domain expert—each with its own role and perspective. These agents communicate, delegate tasks, and cross-check one another, enabling greater scalability, robustness, and problem-solving capability than a single agent acting alone.

Each of these intelligence types has distinct strengths, costs, and trade-offs in terms of performance, determinism, explainability, and flexibility. Most real-world Vantiq applications combine several of them to achieve scalable, resilient, and situationally aware behaviour. The sections that follow explore each form of intelligence in more detail and describe how they are applied and integrated within the Vantiq platform.

The table below summarises these four forms of intelligence across the dimensions that matter most when choosing between them — and, in practice, when combining them.

	Knowledge-Based Rules	Predictive AI	Generative AI	Agentic AI
Best use case	Known, well-defined conditions that need fast, deterministic, auditable responses.	Detecting anomalies and complex multi-signal patterns that no explicit rule anticipates.	Synthesising context into human-readable explanations, summaries, recommendations, and plans at runtime.	Goal-driven tasks that require autonomous planning, decisions, tool use, and adaptation.
Typical Vantiq implementation pattern	Visual Event Handlers (VEH) and VAIL Rules bound to service events.	An external ML model (e.g. regression or DBSCAN clustering) or a computer-vision model served over the Open Inference Protocol, scored from a Visual Event Handler.	A large language model invoked through a Vantiq source, typically grounded with Retrieval-Augmented Generation (RAG) over curated knowledge.	Single-agent or multi-agent applications (e.g. planner, executor, critic) built from Vantiq agents and tools.
Strengths	Precise, explainable, low-latency, and easy to govern.	Learns from data, surfaces hidden and nonlinear patterns, handles uncertainty, and can be retrained over time.	Natural-language understanding and generation, multimodal, adaptable, and reduces cognitive load.	Autonomous multi-step reasoning and tool use; scalable and robust when agents collaborate.
Limitations	Limited to what is already known; rule sets grow complex and rigid and cannot discover new patterns.	Needs quality training data; probabilistic and less explainable; can drift and requires monitoring and retraining.	Can hallucinate; non-deterministic; adds latency and cost; needs grounding and guardrails.	Hardest to predict and control; emergent behaviour; greater oversight, cost, and governance overhead.
Example scenario	Raise an alert or shut a pump down when its pressure crosses a defined safety threshold.	Flag an emerging pump anomaly from subtle multivariate sensor drift before any threshold is crossed.	Turn a detected pump anomaly into a grounded, plain-language alert pushed to an engineer's mobile.	An agent diagnoses a pump anomaly, plans remediation steps, and coordinates actions across connected systems.

Table 1: Knowledge-based rules, predictive AI, generative AI, and Agentic AI at a glance

2. Knowledge-Based Rules: Encoding What Is Known

Knowledge-based rules represent the most explicit and reliable form of intelligence in an event-driven system. They encode domain expertise, policies, and constraints in a form that is predictable, explainable, and enforceable. Given the same inputs, these rules always produce the same outcome—making them essential for correctness, safety, and trust.

Rules are typically expressed as conditional logic or decision structures: if a known condition occurs, then a specific action or outcome must follow. This makes them especially well-suited for situations where behaviour is well understood, consequences are clear, and deviation is unacceptable.

In many systems, these rules are strengthened through the use of statistical functions—such as moving averages, rolling percentiles, or standard deviation—not to infer unknown behaviour, but to detect known and well-defined changes in signals over time. For example, a rule may trigger when a metric deviates beyond a predefined number of standard deviations from its baseline, or when a moving average crosses a fixed threshold.

2.1 Rules as Explicit System Knowledge

Knowledge-based rules capture what an organization already knows to be true about its domain. This includes:

- Business policies and procedures
- Regulatory and compliance requirements
- Safety constraints and operating limits
- Well-understood cause-and-effect relationships

Because this knowledge is explicitly encoded, the rules are transparent and auditable. Developers, operators, and regulators can inspect, validate, and reason about rule behaviour without needing to interpret predictive models or inferred logic.

2.2 Rules as Immediate Perception

In real-time, event-driven systems, rules serve as a form of immediate perception. They clearly recognize states at the moment an event occurs—for example, when a threshold is exceeded, a policy is violated, or a mandatory condition is met.

This makes rules ideal for:

- Detecting known error or failure conditions
- Enforcing hard safety or security boundaries
- Triggering mandatory actions or escalations
- Providing fast, deterministic responses with minimal latency

Unlike predictive models, rules do not estimate or infer; they assert certainty. When a rule fires, the system knows exactly why it happened.

2.3 Strengths and Limitations of Rules

Deterministic rules offer several critical strengths:

- Precision and reliability for known conditions
- Explainability, since the logic is explicit

- Low latency, making them suitable for real-time enforcement
- Governance and control, especially in regulated domains

However, rules are inherently limited by what is already known. They struggle with:

- Dependence on prior knowledge and anticipated conditions
- Inability to discover previously unknown patterns or relationships
- Rapid growth in complexity when modelling interactions across many signals
- Rigid interpretations of “normal” and “abnormal” that require manual adaptation
- Increasing maintenance overhead as rule sets expand and evolve

As systems grow in scale and complexity, attempting to encode all behaviour purely as rules can lead to escalating structural complexity, rule explosion, and increasing maintenance overhead. This becomes especially pronounced when rules must evaluate combinations of many interacting signals, thresholds, and contextual conditions. What begins as simple deterministic checks can evolve into deeply nested logic, cross-dependent rule chains, and overlapping condition sets that are difficult to reason about, test, and maintain. As the number of signals increases, the possible interaction paths grow exponentially, making it impractical to anticipate and explicitly encode every meaningful combination through static rules alone. Instead of relying solely on large collections of deterministic rules, complex discrete knowledge can be structured using approaches such as decision models, constraint solvers, knowledge graphs, or state machines. These methods reorganise knowledge into more scalable representations—such as structured decision logic, declarative constraints, explicit relationship models, state transitions, or higher-level derived concepts—so that complex interactions can be managed systematically without requiring an explosion of tightly coupled brittle rules.

2.4 Implementing Knowledge-Based Rules in Vantiq

Knowledge-based rules are implemented in Vantiq using either **Visual Event Handler (VEH)** or as **VAIL Rules**. The developer defines a Vantiq Service and either defines a public event-driven interface using an Inbound Event on the service or, for an integration-style service, associates Visual Event Handlers or VAIL Rules with a Source. Services can also define Private Events, and the process for implementing a Visual Event Handler or VAIL Rules for a private event is identical; the only difference is scope: private events can only be referenced within the service that implements them.

Visual Event Handlers are defined graphically using a low-code visual tool. Visual Event Handlers are implemented as a series of nodes with one-way connections between the nodes, never forming closed loops – AKA Directed Acyclic Graph (DAG). Each node is called a Task, and a Task is an instance of a template known as an Activity Pattern. There are ~18 predefined Activity Patterns, and it is possible to define custom activity patterns via the creation of VEH Components. These components can be shared with other developers via the platform's Catalog capabilities; we will not cover catalogs here. At the root of every VEH is one or more special Event Stream Tasks. Event Streams are tasks associated with either an Inbound Event (or Private Event) or associated with a Source (there are other styles of event streams, but we will not cover these here as they are not commonly used or recommended).

VAIL Rules are implemented in the VAIL Domain Specific Language (DSL). VAIL Rules are bound to the Service’s event-driven interface or to events being generated by Sources with a ‘WHEN EVENT OCCURS ON’ statement, which defines a trigger for when the rule will fire and encodes the required deterministic behaviour in the rule body. The ‘WHEN EVENT OCCURS ON’ syntax can also implement filtering logic, such as using a ‘WHERE’ clause. There are a

series of more advanced VAIL Rules clauses and correlation capabilities, but these are not supported by Service-based Rules, are rarely used, and won't be discussed here.

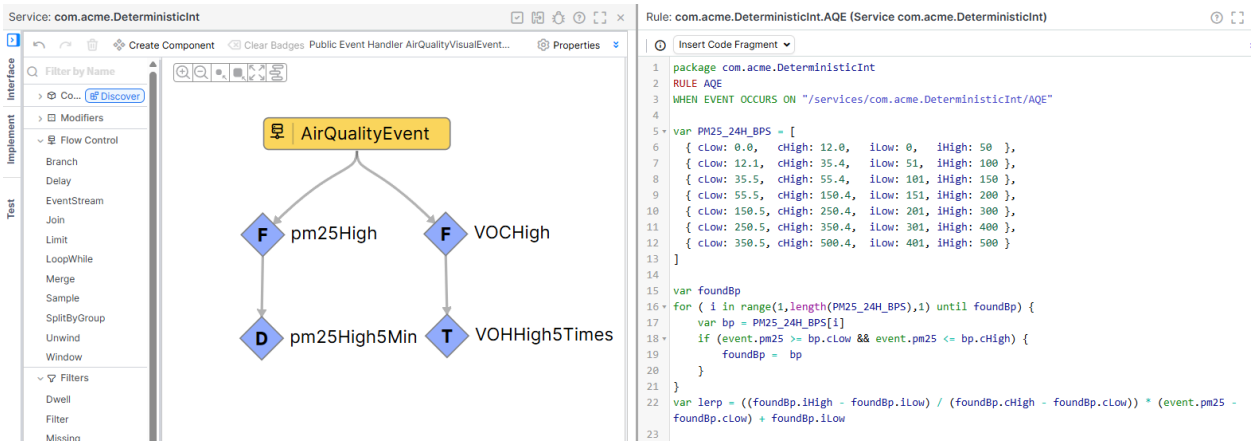


Figure 1 - Visual Event Handler and VAIL Rules

2.4.1 Implementing Visual Event Handlers

Visual Event Handlers are the recommended approach for implementing knowledge-based rules in Vantiq. Although the developer composes a rule graphically in the visual editor, the platform generates a full VAIL implementation behind the scenes—every saved handler becomes a set of VAIL rules, event streams, and procedures that the developer never has to write or maintain directly.

Each Task in a VEH is an instance of an **Activity Pattern**—a reusable, parameterised template. Activity Patterns are organised by purpose, and most knowledge-based rules can be expressed by composing a handful of them. The categories most relevant to rule implementation are:

- **Event ingestion:** Event Stream Tasks form the root of the handler and listen to an Inbound Event, a Source, or a Topic. Merge and Join combine independent streams; Split By Group partitions a stream by a key value.
- **Filtering and routing:** Filter restricts events using a VAIL condition; Branch creates conditional paths through the handler; Limit and Sample control event throughput.
- **Transformation and enrichment:** Transformation reshapes events using a procedure or a mapping; Enrich and Cached Enrich attach data drawn from persistent storage; Unwind expands array properties into separate events.
- **Time and state:** Window batches events over a duration; Threshold detects a value crossing a boundary; Dwell identifies a sustained condition; Missing fires when an expected event fails to arrive; Accumulate State carries state across a sequence of events.
- **Analytics:** Analytics and Compute Statistics calculate aggregates such as mean, median, percentile, and standard deviation—directly supporting the statistical functions used to strengthen rules.
- **Actions:** Save to Type, Publish to Service/Source/Topic, Procedure, VAIL, Notify, and Reply persist, route, or act on the events that satisfy the rule.

Implementation of a rule typically follows an iterative pattern. The developer begins by placing an Event Stream Task at the root of the handler, configured against the event the rule should respond to. Filtering, transformation, and enrichment Tasks are then added to focus the stream onto only the events relevant to the rule and to attach any contextual data it needs. The

conditional logic of the rule—the “if a known condition occurs” portion—is expressed using Filter, Branch, Threshold, Dwell, or Accumulate State. The action that should follow—the “then a specific action follows” portion—is implemented by connecting Notify, Save to Type, Publish, Procedure, or Reply Tasks to the appropriate output.

During development it is good practice to attach Log Stream Tasks at intermediate points to observe events flowing through each stage; these can be removed or disabled once the behaviour of the handler is validated. Starting with raw event logging and progressively adding filtering, transformation, and finally action Tasks lets the developer refine a rule rapidly without rewriting code.

When the event source feeding a VEH is configured as reliable, Vantiq automatically extends the same reliability guarantees to the handler. The platform records checkpoints as events flow through each Task, partitions checkpoint state by event UUID, and redelivers failed events from the appropriate resume point. Application errors—failures caused by the content of the event itself—do not trigger redelivery, while transient errors do. Stateless Tasks should therefore be implemented idempotently because they may execute more than once during recovery.

Visual Event Handlers are generally preferred over VAIL Rules for implementing knowledge-based rules because they are easier to read, review, and govern; they expose a pretested library of Activity Patterns that cover the constructs most rules need; and they automatically generate efficient, reliable VAIL. VAIL Rules remain the right choice when the logic is highly procedural, requires constructs not exposed in the visual library, or is more concisely expressed in code. The two approaches can also be combined—a VEH can invoke a VAIL procedure from within a Procedure Task to handle a specialised computation while keeping the overall control flow visual.

2.4.1.1 Event Correlation

In many rules, the conditions that drive a decision are not contained in a single event but emerge from the near-simultaneous occurrence of related events on two or more independent streams. The Join Activity Pattern is the mechanism by which a VEH expresses this kind of event correlation: it accepts events from multiple parent streams, evaluates a set of constraints that define when those events should be treated as a match, and produces a single combined event for downstream processing when the correlation succeeds.

A Join Task has two or more parent streams. The developer configures an ordered set of **constraints**—VAIL expressions that define when an event from one stream matches an event from another. The constraints are evaluated as a logical **AND**: every constraint must evaluate to true for the join to fire. There is no OR option—if any constraint fails, no combined event is produced for that candidate match. A typical constraint requires that the events share a common identifier, for example `engineSpeedStream.engineId == engineTempStream.engineId`. Constraints can refer to a single stream (to filter that stream’s events before correlation) or to events from two streams together (to express the correlation itself); the AND semantics apply uniformly across both kinds.

The **within duration** property defines the time window in which matching events must occur on all parent streams. If matching events arrive on every parent stream within that window, the Join emits a single combined event on its primary output containing the data from each contributing parent event. If a matching event fails to arrive within the window, the Join emits a partial event on its **timeout** secondary output, allowing the rule to react to the absence of correlation as well as to its presence.

The order in which the parent streams are evaluated by the Join matters, because it determines which stream paces the join: every join attempt begins with an event from the first stream in the join order and then waits for matching events on the remaining streams within the time window. The streams are always rendered left-to-right in the VEH editor, and that visual left-to-right order is the join order. By default, the join order matches the order in which the parent streams were added to the Join Task (first stream added is leftmost, then second, and so on). The developer can change this through the **join order** configuration on the Join Task; updating the join order also reorders the streams visually in the editor, so the visual layout always reflects the current join order. Choose the leftmost (first) stream to be the one whose events naturally define when a correlation should be evaluated.

This pacing behaviour has two practical consequences that are important to design around. First, each event that arrives on the leftmost stream produces at most one downstream event (either a combined join event or a timeout event); events arriving on the other parent streams do not trigger a join and do not produce output events. If two events arrive on the leftmost stream and only one matching event arrives on a right-hand stream within the **within duration** window, both leftmost events join against that single right-hand event and two combined output events are produced. Conversely, if only one event arrives on the leftmost stream and two matching events arrive on a right-hand stream, only one combined output event is produced, because there is no second leftmost event to drive a further join. The frequency of output events from a Join is therefore set by the frequency of events on the leftmost stream, not by the combined arrival rate of all parents.

Second, the **timeout** secondary output fires only when an event on the leftmost stream fails to find a match on every other parent stream within the window. Events that arrive on a right-hand stream and never find a matching leftmost event are silently discarded and never produce a timeout. A Join is therefore well-suited to expressing rules of the form “when X happens, expect Y within N seconds, and react if Y does not arrive,” where the trigger condition (X) is the leftmost stream and the expected confirmation (Y) is on a right-hand stream. It is not suitable when the rule needs to detect missing or unmatched events on a stream other than the leftmost; in those cases, re-order the parent streams using the join order property, or model the rule with a separate handler.

To make this concrete, consider a Join Task whose two parent streams are `TempEvent` (leftmost, and therefore the pacing stream) and `PressureEvent`, correlated on a shared `pumpId` with a **within duration** of one minute, see figure below.

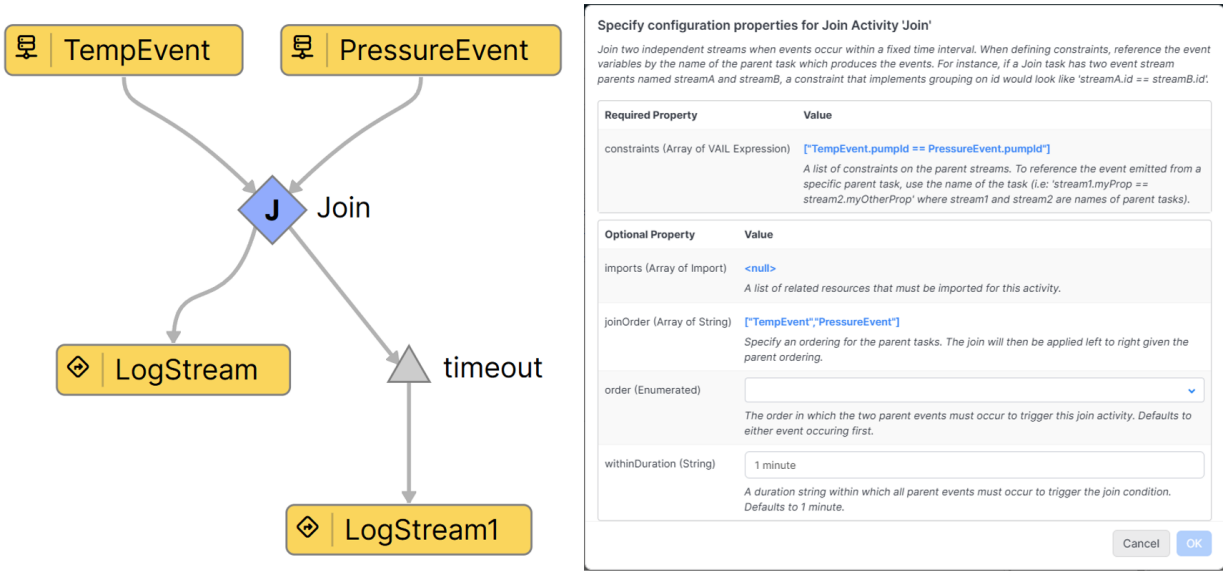


Figure 2: Join Example

The table below traces several arrival sequences through this Join and shows what it produces in each case.

Inbound event sequence (within 1 min, same pumpId)	**Output produced by the Join**	**Why**
TempEvent → PressureEvent	One combined event on the primary output: `{TempEvent={pumpId=1, temp=30}, PressureEvent={pumpId=1, pressure=2000}}`	The TempEvent opens a pending join attempt. The PressureEvent arrives inside the within duration window and satisfies the correlation, so the join fires once on the primary output.
TempEvent → PressureEvent → PressureEvent	One combined event on the primary output: `{TempEvent={pumpId=1, temp=35}, PressureEvent={pumpId=1, pressure=2500}}` The second PressureEvent produces no output.	The first PressureEvent satisfies the pending TempEvent and the join fires. After that there is no further leftmost event to drive another join, so the second PressureEvent has nothing to pair with and is silently discarded when its window expires.
PressureEvent → PressureEvent → TempEvent	One combined event on the primary output: `{TempEvent={pumpId=1, temp=40}, PressureEvent={pumpId=1, pressure=1000}}`	Neither PressureEvent can drive a join on its own — only the leftmost stream paces the Join. When the TempEvent arrives it correlates with a PressureEvent that is still inside the window and the join fires once. The remaining PressureEvent has no further leftmost event to pair with and is silently discarded.

Inbound event sequence (within 1 min, same pumpId)	Output produced by the Join	Why

TempEvent → TempEvent → PressureEvent	Two combined events on the primary output, both referencing the same PressureEvent:
	`{TempEvent={pumpId=1, temp=20}, PressureEvent={pumpId=1, pressure=5000}}`
	`{TempEvent={pumpId=1, temp=10}, PressureEvent={pumpId=1, pressure=5000}}`
		Each TempEvent is its own leftmost event and opens its own pending join attempt. Both attempts are still waiting when the PressureEvent arrives, and each finds a match within its window — so two combined events are produced against the same PressureEvent. The number of outputs is governed by the leftmost stream, not by the number of right-hand events.

TempEvent only (no PressureEvent within 1 min)	One partial event on the **timeout** secondary output, containing the unmatched TempEvent. Nothing is produced on the primary output.	A leftmost TempEvent opened a pending join and its window expired without a matching PressureEvent. That is exactly the condition the **timeout** output is designed to surface, allowing the rule to react to the absence of correlation.
PressureEvent only (no TempEvent within 1 min)	No output on either the primary or the timeout output.	Events on a right-hand stream never initiate a join attempt. With no leftmost TempEvent to pace it, the PressureEvent is held until its window expires and is then silently discarded. The **timeout** output does not fire because it only surfaces unmatched leftmost events.

Typical use cases include:

- Correlating sensor readings of different types from the same physical asset (for example, engine speed and engine temperature) so that a rule can act on the combined operating state.
- Matching an order event against a corresponding shipment, payment, or fulfilment event within a service-level window.
- Confirming that a user or system action has been followed (or not followed) by a corresponding downstream event, with the timeout output handling the “not followed” case.
- Combining a primary event with contextual events from independent sources before applying a rule.

It is useful to contrast Join with the patterns it is most often confused with:

- **Merge** combines events from multiple streams into a single downstream stream without correlating them. Use Merge when the rule should treat events from several sources uniformly; use Join when the rule needs the events to be matched.
- **Enrich** and **Cached Enrich** attach data that lives in a persistent Type to an event. Use Enrich when the additional data is at rest in storage; use Join when the additional data is itself a transient event on another live stream.

2.4.1.2 Split By Group

Of the Activity Patterns available to a VEH, Split By Group is one of the most important and most frequently used. It is the mechanism by which a single VEH can process events from many distinct entities—machines, sensors, patients, vehicles, conversations, customers—while keeping the state and behaviour of each entity logically isolated from every other entity flowing through the same handler.

A Split By Group Task partitions a single inbound event stream into a set of independent sub-streams. The developer supplies a **group key expression**—a VAIL expression evaluated against each event that yields a value identifying which group the event belongs to. For example, an event containing readings from many devices might be grouped by `event.deviceId`, and a clinical event stream might be grouped by `event.patientId` or `event.bedNumber`. Every event that produces the same group key is routed into the same sub-stream; events with different keys flow into different sub-streams. From the developer's point of view, the rest of the handler is authored as if it were processing a single entity at a time, and the platform automatically scopes the downstream Task behaviour per group.

Per-group state isolation is the primary benefit of Split By Group. Tasks that maintain state—Window, Threshold, Dwell, Missing, Analytics, Compute Statistics, Linear Regression, K-Means, DBScan, and the other clustering and prediction Tasks—all become scoped to a single group when placed beneath a Split By Group. This means:

- A Window Task collects a separate batch of events for each group.
- A Missing Task fires independently for each group when an expected event from that group fails to arrive within the configured timeframe.
- A Threshold or Dwell Task fires independently for each group, based only on that group's events.
- An Analytics Task computes mean, percentile, or standard deviation independently per group.

Without Split By Group these Tasks would share a single state across every event the handler sees, which is rarely the desired behaviour in any real-world rule that operates across multiple entities. Conceptually, the developer designs the rule as though only one entity exists and Split By Group transparently replicates that behaviour for as many entities as the inbound stream contains. The following figures show two examples: one with a Window followed by an Analytics pattern and no Split By Group, and a second with a Split By Group before the Windows, to help illustrate what is happening behind the scenes.

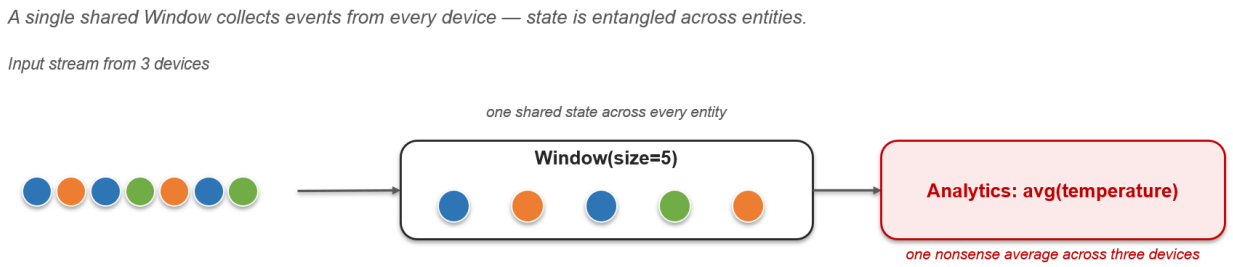


Figure 3: Windows & Analytics without SplitByGroup

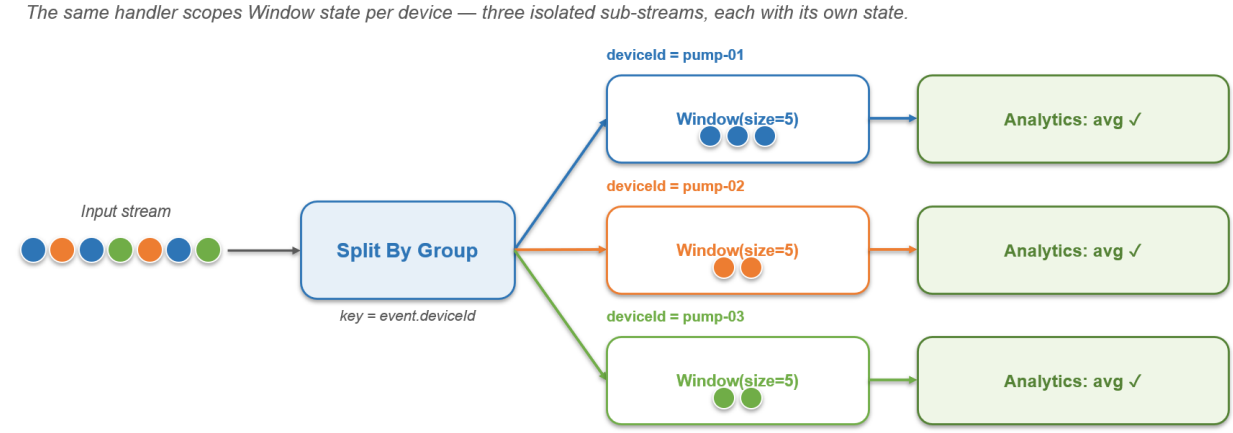


Figure 4: Windows & Analytics with SplitByGroup

Cached Enrich attaches data from a persistent Type to an incoming event, just as Enrich does, but it caches the record it retrieves and reuses it for subsequent events rather than querying storage each time. The cached association is refreshed automatically on a configurable interval, so Cached Enrich is well-suited to high-volume streams whose associated data changes relatively infrequently, and it typically performs better than a plain Enrich in that situation. Cached Enrich is most effective when placed after a Split By Group. Because Cached Enrich holds a single cached record at a time, scoping it to a single entity means the cached lookup is reused across the events belonging to that entity and refreshed only when the active entity changes. Placing Cached Enrich above a Split By Group would cause the cache to thrash as events from different entities arrive in interleaved order.

A small number of constraints apply when working with Split By Group:

- A Join Task cannot appear beneath a Split By Group. Correlation between independent streams must be performed either upstream of the Split By Group or in a separate handler.
- The group key expression should yield a stable, well-defined value for every event. Events whose key evaluates to null all flow into the same “null group,” which is rarely the intended behaviour and is usually a sign that an upstream filter or transformation is missing.

It is also important to recognise that Vantiq does **not** automatically manage or evict the state held for each group. Once a group has been seen by a Split By Group Task, the per-group state of every downstream stateful Task persists in memory for as long as the handler remains active. For handlers that operate over a small, bounded set of entities—a fixed roster of machines, patients, or vehicles—this is rarely a concern. For handlers that partition by a key with very high or unbounded cardinality—for example, a transient session identifier, a request UUID, or a per-

message correlation value—the accumulated per-group state can grow without bound and may eventually exhaust the memory available to the service hosting the handler. Developers should therefore be deliberate about choosing a group key whose cardinality is known and bounded, and where the natural cardinality is very large, should plan for explicit state management—through time-based eviction, periodic cleanup procedures, or restricting the handler to a finite, identifiable subset of entities.

As a matter of best practice, place Split By Group as early in the handler as practical—immediately after any filtering or transformation required to extract or compute the group key—so that the maximum amount of downstream logic benefits from per-group scoping. Choose a group key that reflects the natural unit of state for the rule: the equipment identifier for an equipment-monitoring rule, the patient identifier for a patient-monitoring rule, the user identifier for a per-user collaboration rule. When the group key represents a hierarchy (for example, region→site→device), Analytics Tasks downstream of the Split By Group can be configured with dimension hierarchies to compute aggregates at multiple levels simultaneously.

2.4.1.3 Temporal Operators

Many situations a rule needs to detect cannot be expressed in terms of a single event. They emerge from a sequence of events, from a sustained condition over time, or from the absence of an expected event. A temperature reading that exceeds a limit for more than thirty seconds, a moving average that crosses a control boundary, a process that fails to acknowledge a request within its service-level window—each of these requires the handler to reason across multiple events or across an extended time period rather than reacting to a single arrival.

Vantiq provides a family of Activity Patterns—collectively referred to here as **temporal operators**—specifically for this kind of reasoning. The most commonly used are:

- **Threshold** detects when a value crosses a configured boundary, producing an event the first time the crossing occurs. A typical use is “fire when the temperature first exceeds 80°C.” Threshold can be configured to fire on either direction of crossing and to apply hysteresis so that small oscillations around the boundary do not produce a flood of events.
- **Dwell** detects a *sustained* condition. It fires when a specified condition has held true continuously for at least a configured duration—the classic example being “fire when temperature has been above 80°C for thirty seconds,” distinguishing a transient spike from a sustained problem. Dwell is also widely used to detect when an asset has remained motionless, occupied, or in a particular state long enough to require attention.
- **Window** collects events into time- or count-based batches that can be processed as a group. A Window over thirty seconds yields a batch of every event that arrived during that window; a Window over the last twenty events yields a sliding batch of the most recent twenty. Window is the foundation for most aggregate calculations in a VEH—moving average, percentile, or standard deviation across a sliding range.
- **Missing** detects the *absence* of an expected event. Missing fires when an event that should have arrived within a configured timeframe has not. This is the natural way to express rules of the form “if no heartbeat is received within sixty seconds, raise an alarm” or “if a discharge confirmation does not arrive within the service-level window, escalate.”
- **Time Difference** measures the interval between sequential events on the stream. It is useful when the rule needs to react to events arriving unusually slowly or unusually quickly, or to compute the elapsed time between a triggering event and a corresponding completion event.

- **Rate** emits a periodic event reporting the frequency of events on its inbound stream. Rate is useful for detecting surges or drops in activity that signal anomalous behaviour.

All of these patterns are state-bearing: each carries some memory of previous events to evaluate the temporal condition. When the rule must reason about temporal behaviour on a per-entity basis (per machine, per patient, per session), the temporal operator should be placed beneath a Split By Group so that its state is partitioned by entity rather than shared across the entire inbound stream. Without that scoping, a Dwell intended to fire "when machine X has been above temperature for thirty seconds" would instead fire when the combined temperature stream across all machines satisfied the condition—almost certainly not what the rule intends.

2.4.1.4 Statistical Patterns and Lightweight Predictive Analytics

A subset of the Activity Patterns available in a VEH go beyond detecting known conditions and instead maintain statistical summaries of the inbound stream or fit a model to it. These patterns make a useful class of predictive analytics achievable directly inside the handler—without invoking an external machine-learning service—and let the developer make decisions on the resulting trends, fits, and clusters.

The main statistical patterns are:

- **Analytics** maintains one or more configured aggregates (mean, median, percentile, standard deviation, and so on) over the inbound stream. Optional dimension hierarchies allow the same aggregate to be calculated at multiple grouping levels simultaneously (for example, region→site→device).
- **Compute Statistics** is a focused variant that maintains the standard set of aggregates over a single property of the inbound event stream. It is the simplest way to track running statistics for one signal of interest.
- **Linear Regression** fits a linear model through the events in the inbound stream. Its get procedure returns the slope and intercept of the fitted line, Pearson's correlation coefficient r (a measure of how strongly the data follow a straight line, carrying the sign of the trend), and the mean squared error. Once fitted, the line can be used to predict the value of the dependent variable for any value of the independent variable.
- **Polynomial Fitter** fits a polynomial of a configured degree through the inbound events and exposes a prediction function. Useful when the underlying relationship is not adequately described by a straight line.
- **K-Means Cluster** assigns inbound events to one of a configured number of clusters and exposes the cluster centres. By comparing a new event's distance from the nearest centre, the handler can detect points that do not fit the established clusters—a lightweight form of anomaly detection.
- **DBScan** performs density-based clustering in which clusters emerge naturally from the data rather than being specified up front. Points that do not belong to any cluster are reported as outliers, providing an alternative anomaly-detection signal.

Together, these patterns enable two complementary forms of in-handler predictive analytics. The forecasting patterns (Linear Regression and Polynomial Fitter) allow the developer to ask a forward-looking question of the data: "Given the trend observed so far, what value is expected at time T?" The clustering patterns (K-Means and DBScan) let the developer ask a structural question: "Does this new event belong to a pattern we have already seen, or is it anomalous?" The aggregate patterns (Analytics and Compute Statistics) feed both forms by maintaining summary statistics that other Tasks (Filter, Branch, Threshold, Dwell) can react to—firing, for example, when a moving average crosses a control limit or when standard deviation exceeds a normal range.

An important access-pattern detail distinguishes these statistical patterns from most other Activity Patterns. **Compute Statistics**, **Analytics**, **Linear Regression**, **Polynomial Fitter**, **K-Means Cluster**, and **DBScan** do **not** emit their results as downstream events. Instead, each saved Task generates a corresponding **get procedure** that the developer must call to retrieve the current results—the running aggregates of Analytics and Compute Statistics, the slope, intercept, correlation coefficient, and mean squared error of Linear Regression, the coefficients and prediction function of Polynomial Fitter, the cluster centres of K-Means, and the cluster assignments and outliers identified by DBScan. Each Task continuously updates its internal state as events arrive, but no event flows on a primary output stream—the data is available only when explicitly queried. In practice, this means that any rule that depends on the current results must invoke the generated procedure (from a Procedure Task, a VAIL Task, or an external caller) to obtain them. The advantage of this design is that the latest results are always available on demand without flooding the downstream graph with recomputation events; the trade-off is that the developer must explicitly fetch the values rather than relying on the usual emit-and-react model.

2.4.1.4.1 Lightweight Analytics vs. Trainable Predictive AI

It is worth clarifying how these patterns relate to the Predictive AI described later in this guide. The essential difference is trainability. These lightweight patterns are not trained and do not learn: each one fits its result to the events currently in the stream or window — a slope, a set of cluster centres, a moving average — and recomputes from scratch as that data moves on. They hold no memory of past data beyond the current window, accumulate no experience, and do not improve with exposure. Run the same events through them tomorrow, and they behave exactly as they do today. The developer chooses which pattern to apply and sets the thresholds that act on its output; the pattern simply quantifies what is happening right now, which keeps it transparent and auditable, in keeping with the knowledge-based-rules philosophy of this chapter.

Full Predictive AI, covered in Predictive AI: Anticipating and Detecting Complex Patterns and Anomalies, is defined by the very capability these patterns lack: it learns from data through training. A predictive model is trained on large volumes of historical data and — crucially — can be retrained as new data arrives, continuously refining its understanding and adapting to conditions not present when it was first built. That ability to learn and adapt is what lets it discover unknown, nonlinear, and multivariate patterns, and to keep pace with a changing environment, rather than only describing the data immediately in front of it. In short, reach for these lightweight patterns when a transparent, fixed calculation over the live stream is enough; step up to a trained Predictive AI model when the system needs to learn from accumulated experience and adapt over time.

2.4.1.5 Predicting pump overheating – Linear Regression examples

A circulating pump in a chiller plant reports a temperature reading every second. The pump's normal operating temperature is around 55 °C, and the alarm threshold is 80 °C. A Threshold Task would fire only when the temperature reaches 80 °C—too late to prevent the alarm. Linear Regression, by contrast, lets the handler reason about the *trend* in the temperature curve and react to the trajectory *before* the alarm threshold is crossed.

The handler layout is a simple chain of Activity Patterns:

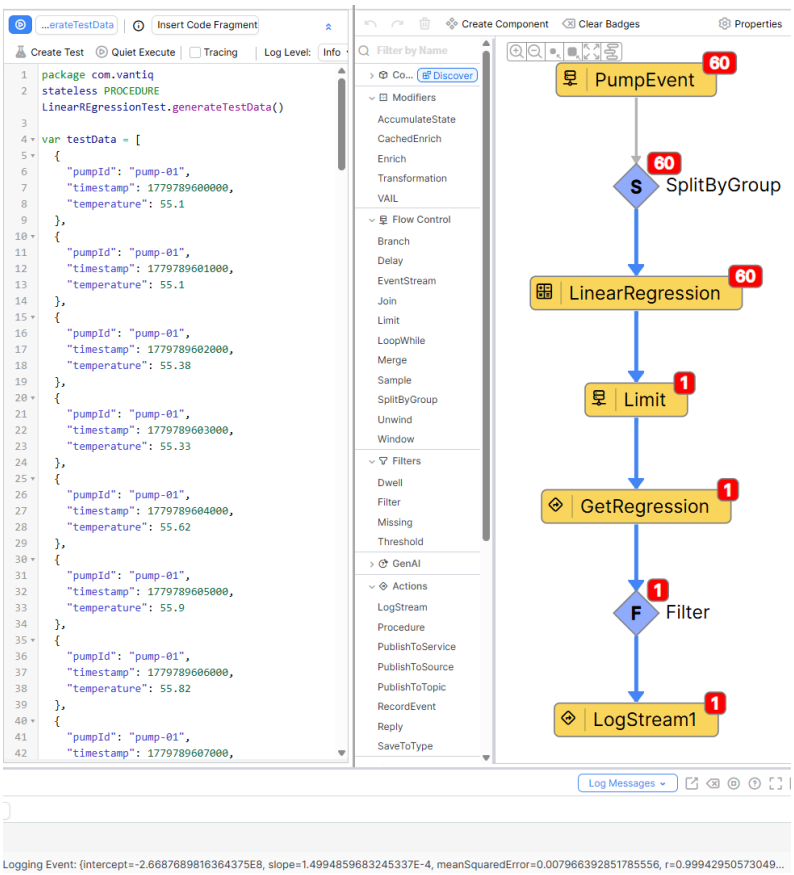


Figure 5: Linear Regression VEH Example

Linear Regression requires both x and y to be numeric, so the event provides `timestamp` as an epoch-milliseconds integer rather than as an ISO-8601 string. The Procedure Task calls the Linear Regression Task's generated *get* procedure, which returns the current slope, intercept, Pearson's correlation coefficient r, and mean squared error. Because x is expressed in milliseconds, the raw slope is in °C/ms (multiplying by 1000 gives °C/sec), and the intercept is the temperature extrapolated back to epoch 0—a large negative number that nonetheless cancels out cleanly in the prediction calculation shown below.

For this Linear Regression Task the get procedure returns the four fields below. The example values are taken from one run of the test data, and the right-hand column maps each field to the corresponding method on the underlying Apache Commons Math `SimpleRegression` class:

Field	Example value (this run)	Meaning (and SimpleRegression method)
`slope`	`1.4994859683245337E-4`	Slope of the fitted line. Because x is epoch milliseconds it is in °C/ms; ×1000 gives 0.1499 °C/sec.
`intercept`	`-2.6687689816364375E8`	Temperature extrapolated to x = 0 (epoch 1970), hence the large negative value; not directly meaningful with raw epoch-ms x.

`meanSquaredError`	`0.007966392851785556`	Sum of squared errors divided by the degrees of freedom (MSE); a small value indicates a tight fit.
`r`	`0.9994295057304937`	Pearson's correlation coefficient (not R^2); strength and sign of the linear trend. Square it for the coefficient of determination.

Calculating the prediction. The get procedure returns only the fitted parameters; the prediction itself is computed in the Procedure Task. The fitted line is `temperature = slope × tMs + intercept`. Solving that line for the time at which the temperature reaches the 80 °C alarm, then subtracting the current time, gives the predicted lead time in seconds:

```
slopePerSec = slope * 1000                    // 0.14995 °C/sec

alarmTimeMs = (80 - intercept) / slope        // epoch ms at which the line
reaches 80 °C

etaSeconds  = (alarmTimeMs - nowMs) / 1000    // nowMs = timestamp of the
latest event
```

With the values from the table above, `alarmTimeMs` ≈ 1,779,789,766,635; measured against the latest event timestamp of 1,779,789,659,000, this gives an `etaSeconds` of about 108 seconds. This is the one place where the raw epoch-based intercept is directly useful, because it anchors the line in absolute time. A simpler alternative computes the same quantity from the most recent temperature reading, `etaSeconds = (80 – currentTemp) / slopePerSec`, but that form mixes the noisy latest reading with the fitted slope, whereas the slope-and-intercept calculation relies entirely on the smoothed regression line.

The rule that fires the warning requires all three of the following conditions to be true:

- **slopePerSec** is greater than 0.1 °C/second – a clear upward trend, not noise.
- **r** (Pearson's correlation coefficient, returned directly by the get procedure) is greater than 0.7 – a strong, positive linear trend rather than noise. Because r carries the sign of the trend, a value above 0.7 also confirms the temperature is rising rather than falling.
- **etaSeconds** is less than 300 – the predicted time to alarm is under five minutes.

The first two conditions filter out noisy or spiky data where a linear trend is illusory. The third condition is the predictive payload: it converts a positive slope into an *actionable* lead time. The downstream consumer of the warning event can then act on the prediction—opening a bypass valve, paging the duty engineer, or throttling upstream load—well before the alarm threshold is actually crossed.

2.4.1.6 Detecting outbreak clusters from a stream of case reports: DBScan Example

A public health platform receives a stream of case reports for a contagious illness, such as measles. Each event is a single registered case, carrying the patient's location as latitude and longitude and the time the case was reported. The goal is to detect emerging geographic clusters—outbreak hotspots—as cases accumulate, so that contact tracing and resources can be directed to the right neighbourhoods. DBScan suits this far better than K-Means: the number of hotspots is unknown and changes over time, the affected areas are irregularly shaped, and

the many sporadic, unrelated cases scattered across the region must be treated as outliers rather than forced into the nearest cluster.

The Visual Event Handler is:

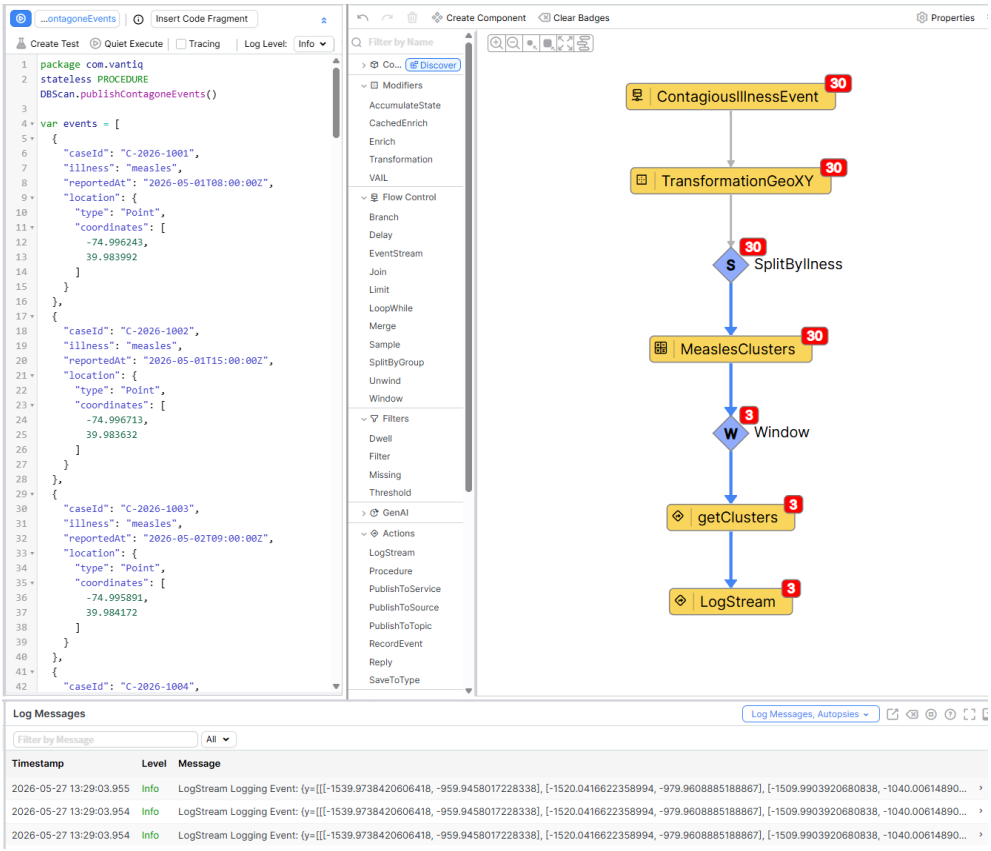
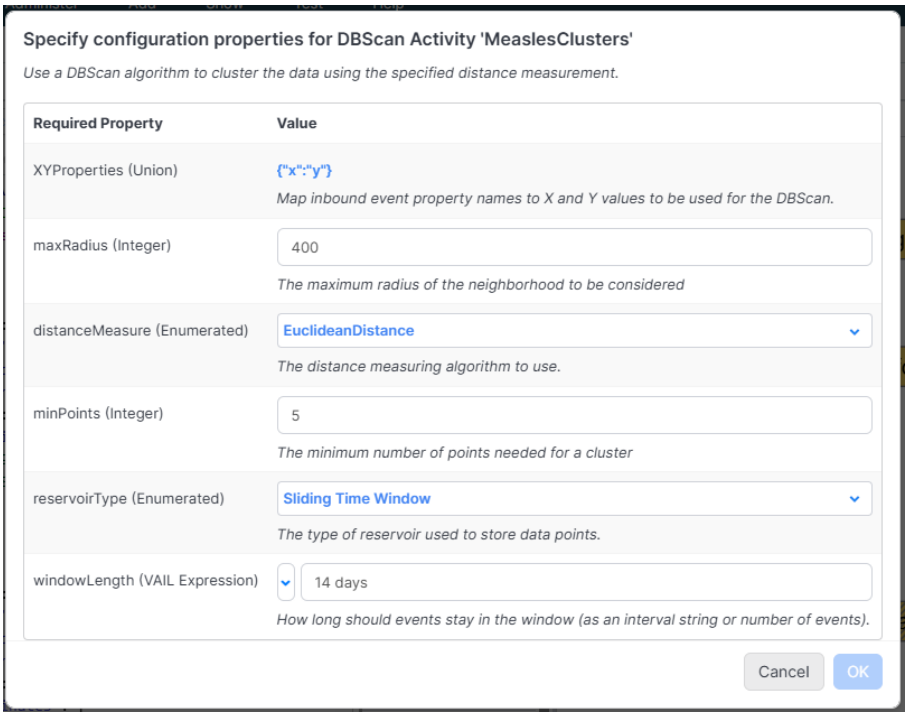


Figure 6: DBScan Example

The MeaslesCluster analysis task is configured as follows using a DBScan Activity Pattern.



This pattern expects Euclidean but the events are GeoJSON containing latitude and longitude which will result in distorted distances if used directly—a degree of longitude is shorter than a degree of latitude, and the ratio changes as you move north or south—so a Transformation (TransformGeoXY) Task first projects each case onto a local plane measured in metres about a regional reference point. Clustering then happens in metres, and maxRadius can be set as a real distance. The following is the VAIL transform procedure:

```
var R = 6371000                                    // Earth radius in metres
var DEG2RAD = 3.141592653589793 / 180              // convert degrees to radians
var lat0 = 40.0, lon0 = -75.0                      // regional reference point
var lon = event.location.coordinates[0]            // GeoJSON Point: [lon, lat]
var lat = event.location.coordinates[1]
event.x = (lon - lon0) * DEG2RAD * R * cos(lat0 * DEG2RAD)
event.y = (lat - lat0) * DEG2RAD * R
```

The `MeaslesDbScan` Task clusters on these projected coordinates. The DBScan Task automatically ingests every event on its inbound stream into its reservoir. Here, the reservoir is a Sliding Time Window, so a case contributes to clustering only while it is recent; once it falls outside the window, it ages out automatically and stops influencing the result. The Task is configured as follows, mapping onto the constructor of the Apache Commons Math `DBSCANClusterer(eps, minPts, measure)` class:

- **XYProperties = [x, y]** — the projected case coordinates, in metres.
- **reservoirType = Sliding Time Window, windowLength = 14 days** — keeps only cases reported in the last fortnight, roughly the period over which co-located measles cases indicate active transmission. (Choose Sliding Count to retain the last N cases instead.)

- **maxRadius (eps) = 400 m** — how close two cases must be to count as neighbours; the neighbourhood radius of a hotspot.
- **minPoints (minPts) = 5** — the minimum number of nearby cases for a hotspot to form. Isolated and sporadic cases fall below this floor and are reported as outliers rather than hotspots.
- **DistanceMeasure = Euclidean** — straight-line distance in the projected metric.

Evaluating the clusters after every single case would be wasteful and noisy. The above example uses a Window to only calculate the clusters every 10 events.

Replaying the accompanying sample data—30 measles cases reported over three weeks—the cluster check fires three times, once after each block of ten cases. Because the reservoir is a sliding 14-day window, the picture changes at each check:

Cluster check	Cases in window	Hotspots found (cluster size)	Outliers
After case 10 (day 8)	`10`	1 — Neighbourhood A (6 cases)	4 — first hotspot flagged
After case 20 (day 17)	`14`	1 — Neighbourhood B (6 cases)	8 — A has aged out; B is the new hotspot
After case 30 (day 22)	`20`	2 — Neighbourhoods B (6) and C (6)	8 — two simultaneous hotspots; A fully gone

The second check is the clearest illustration of the sliding window at work. The six Neighbourhood A cases that triggered the first alert were reported on days 1–3; by day 17 they have dropped out of the 14-day window, so although they remain in the database, they no longer count towards a current hotspot. Meanwhile Neighbourhood B has crossed the density threshold and is flagged. By the third check a second hotspot, Neighbourhood C, has also formed, and DBScan returns both—without being told that two clusters exist—while the ten scattered sporadic cases remain outliers throughout.

For that third check the get procedure returns two clusters. Each cluster carries its member count (`Cluster.getPoints().size()` on the underlying cluster), the member case coordinates (`Cluster.getPoints()`), and a computed centre: Neighbourhood B, 6 cases centred at { x: 1203, y: 817 } metres (40.00734, -74.98587), and Neighbourhood C, 6 cases centred at { x: -1498, y: -1010 } metres (39.99092, -75.01759). Cases that join no cluster—here the ten sporadic reports—are returned as outliers and raise no alert.

The accompanying `outbreak_case_events.json` file contains the 30 case reports used here, in arrival order. A single case has the following shape:

```
{
  "caseId": "C-2026-1001", "illness": "measles",
  "reportedAt": "2026-05-01T08:00:00Z",
  "location": { "type": "Point", "coordinates": [ -74.996243, 39.983992 ] }
}
```

Running the configured handler against the 30-case sample data produces three log events from the Procedure Task, one per fire of the count Window. Each log carries a y field—the

return value of the *get* procedure—which is a list of clusters, each cluster being a list of `[x, y]` points in the projected metric. Across the three invocations the cluster picture builds up as cases accumulate:

Check	Invocation time (2026-05-28)	Clusters returned	Cluster details (size, neighbourhood)
1	`11:04:52`	1	A — 6 cases, centred near (320, -1810) m
2	`11:05:17`	2	A — 6 cases; B — 6 cases, centred near (1204, 814) m
3	`11:05:40`	3	C — 6 cases, centred near (-1498, -1010) m; A — 6 cases; B — 8 cases

The third invocation captures all three clusters. Expanded for readability, its `y` array is:

```
y = [
  // Cluster 0 - Neighbourhood C (6 cases)
  [ [-1539.97,  -959.95], [-1520.04,  -979.96], [-1509.99, -1040.01],
    [-1489.97, -1009.98], [-1480.01, -1019.99], [-1450.02, -1050.01] ],
  // Cluster 1 - Neighbourhood A (6 cases)
  [ [  259.97, -1849.95], [  279.99, -1820.04], [  309.97, -1899.99],
    [  320.02, -1780.01], [  350.01, -1759.99], [  390.04, -1749.99] ],
  // Cluster 2 - Neighbourhood B (8 cases)
  [ [ 1150.02,   849.97], [ 1170.04,   780.03], [ 1180.00,   819.95],
    [ 1189.97,   869.99], [ 1209.99,   810.06], [ 1230.00,   790.04],
    [ 1239.97,   829.96], [ 1259.99,   760.02] ]
]
```

Each top-level element of `y` is a returned Cluster, exposing the case locations DBScan grouped together—equivalent to `Cluster.getPoints()` on the underlying Apache Commons Math object. The numbers are the projected `x, y` of each case in metres about the regional reference point set in the Transformation Task, which is why each cluster's points sit close together. The order of the clusters in `y` reflects DBScan's internal traversal, not a ranking; the ten sporadic cases joined no cluster and so do not appear in `y` at all—they remained outliers throughout the run.

The cluster count grows from one to two to three across the checks because new cases keep arriving and the existing neighbourhoods cross the density threshold one after another. After ten cases only Neighbourhood A is dense enough to meet `minPoints`; after twenty Neighbourhood B has also crossed it; after thirty Neighbourhood C has formed and B has grown to eight members. Each newly-detected hotspot is the trigger for the downstream Publish task to notify field teams.

A reader will note that this `1 → 2 → 3` progression differs from the progression table earlier in the section, which anticipated Neighbourhood A ageing out of the 14-day window between the second and third checks. The Sliding Time Window measures from the moment a case *arrives at the Task* rather than from the `reportedAt` field carried in the event, so when the sample data is replayed in a few seconds all thirty cases remain in-window and the early A cluster persists.

In a production deployment, where cases arrive across weeks of real time, the ageing behaviour described earlier will apply; to reproduce that behaviour in a test, either pace the replay at real time or switch the reservoir to a Sliding Count window so ageing depends on event count rather than wall-clock arrival.

For more sophisticated forms of prediction—deep learning, computer vision, multivariate time-series forecasting—Vantiq integrates with external machine-learning models, as described in the Predictive AI section that follows. The statistical patterns covered here are appropriate when the underlying behaviour is well-modelled by simple regressions, polynomials, or a small number of clusters, and when the additional infrastructure of an external ML pipeline is not warranted.

2.4.2 Implementing VAIL Rules

A VAIL Rule is a self-contained, text-based unit of event-driven logic. Where a Visual Event Handler is composed graphically, a VAIL Rule is written directly in the VAIL language: a single triggering condition followed by a block of statements that execute every time that condition is met. Inside a Service a Rule is bound either to the Service's event-driven interface or to a Source, exactly as a Visual Event Handler is.

Because a Rule is simply code, it is well suited to logic that is awkward to draw as a graph—iterating over a collection of variable length, building up an intermediate data structure, or calling a built-in procedure with arguments you compute on the fly. The examples that follow start from a simple filter and build up to ingesting computer-vision detections and grouping them into clusters.

2.4.2.1 The Shape of a VAIL Rule

Every Rule has the same basic structure: the RULE keyword and a name, a WHEN clause that defines the trigger, and a body of one or more VAIL statements. The trigger uses EVENT OCCURS ON together with an event path of the form /<resource>/<instance>[/<operation>]. An optional AS clause binds the incoming event to a name, and an optional WHERE clause filters which events actually fire the Rule:

```
RULE <ruleName>

WHEN EVENT OCCURS ON "/<resource>/<instance>[/<operation>]" [AS <alias>]
   [WHERE <condition>]


   // VAIL statements that run each time the Rule fires
```

Common event paths are /services/<serviceName>/<eventName> for a Service's inbound event interface, /types/<TypeName>/insert for data changes, /sources/<SourceName> for messages arriving from a Source, and /topics/<topic> for published events. When an alias is supplied the event payload is available as <alias>.value; without an alias it is available as event.value.

2.4.2.2 When to Use a VAIL Rule Instead of a Visual Event Handler

Visual Event Handlers remain the recommended way to implement most knowledge-based rules, and anything a Service-based Rule can do in its body a Visual Event Handler can do as well. The choice is therefore mainly about how the logic is best expressed and maintained.

Prefer a Visual Event Handler when:

- The flow maps naturally onto the built-in Activity Patterns—filtering, transformation, enrichment, calling a procedure, or invoking a model.
- The logic benefits from being documented visually and maintained by several developers.
- You need to correlate or join events across two or more independent streams, which is expressed with the Join Activity Pattern.

Prefer a VAIL Rule when:

- The logic is more concise as code—looping over a variable-length array, evaluating nested conditions, or assembling an intermediate structure before acting.
- You need to filter a high-volume stream efficiently—a WHERE clause on the trigger stops non-matching events from running the Rule at all, rather than filtering them after the fact.
- You want a compact definition that is easy to review as text in a code review or diff.
- You are calling a built-in procedure with arguments you have to compute, as in the clustering example below.
- A developer is simply more productive writing code than composing a graph.

Filtering deserves a closer look, because it is one case where a Rule can be genuinely more efficient than a Visual Event Handler rather than simply more concise. A WHERE clause on the WHEN trigger is evaluated as part of the triggering condition, so the Rule body runs only for events that match—events that fail the condition never trigger the Rule at all. The equivalent in a Visual Event Handler is an Event Stream task feeding a Filter task feeding the task that does the work; because the platform realises each of those tasks as its own underlying rule, the Event Stream and Filter still execute for every event, including the ones the Filter ultimately discards. When the goal is to act only on the small fraction of events that cross a threshold, a single Rule with a WHERE clause avoids that per-event overhead, which can matter at high event volumes.

One constraint applies specifically to Service-based Rules: they support a single triggering event—the WHEN EVENT OCCURS ON trigger and its WHERE filter—together with the full VAIL body, but they do not support the more advanced multi-event correlation clauses such as BEFORE, AFTER, WITHIN, and EXPECT. When event correlation is required inside a Service, use a Visual Event Handler and its Join Activity Pattern.

2.4.2.3 Example 1 – Filtering and Raising an Alert

The simplest Rules react to a single event, apply a condition, and take an action. Here, a Rule declared in the com.vantiq.PumpService package triggers on a TemperatureEvent arriving on that Service's inbound event interface, uses a WHERE clause to ignore readings within the safe range, and emits a HighTemperatureAlert outbound event for anything above the limit:

```
package com.vantiq.PumpService

RULE PumpTempRule

WHEN EVENT OCCURS ON "/services/com.vantiq.PumpService/TemperatureEvent"
   WHERE event.value.celsius > 80


   PUBLISH {
      pumpId:  event.value.pumpId,
      celsius: event.value.celsius,
```

```
      note:    "Temperature above safe operating limit"
   } TO SERVICE EVENT "com.vantiq.PumpService/HighTemperatureAlert"
```

Here the threshold lives in the WHERE clause, so the Rule fires only for readings above 80°C; the body never sees an in-range reading and has no need to re-test the limit. This is the trigger-level filtering described above—a case where a single Rule is both simpler and more efficient than the Event Stream, Filter and Procedure tasks the same behaviour would require in a Visual Event Handler.

2.4.2.4 Example 2 – Enriching an Event with a Lookup

Rules often need context that the triggering event does not carry. This Rule, declared in the com.vantiq.MaintenanceService package, fires when a maintenance request arrives on an MQTT Source, looks up the referenced asset, and—only for assets that are currently in service—emits a WorkAssignment outbound event enriched with the asset's owner:

```
package com.vantiq.MaintenanceService

RULE RouteMaintenanceRequest

WHEN EVENT OCCURS ON "/sources/MaintenanceRequestSource"
   var request = event.value
   // Retrieve the asset referenced by the request
   var asset = SELECT ONE * FROM Asset WHERE assetId == request.assetId
   if (asset != null && asset.status == "IN_SERVICE") {
      PUBLISH {
         assetId:  asset.assetId,
         owner:    asset.owner,
         priority: request.priority,
         message:  "New maintenance request for " + asset.name
      } TO SERVICE EVENT "com.vantiq.MaintenanceService/WorkAssignment"
   }
```

This Rule follows a common shape: a single trigger, a SELECT to fetch related data, a conditional test, and a PUBLISH to act on the result. Note that the trigger is the only part that changes when the same logic is driven by a Source rather than a Type—the body is identical.

2.4.2.5 Example 3 – Finding Clusters of People in YOLO Detections

A more powerful use of a Rule is to call one of Vantiq's built-in services. Object-detection models such as YOLO emit a frame for each processed image, containing an array of detected objects—each with a class label, a confidence score, and a bounding box. A common requirement is to detect when people are gathering not simply that several people are present, but that some of them are physically close together.

The MotionTracking.dbscanCluster(distance, minPoints, points) procedure does exactly this. It runs the DBSCAN density-based clustering algorithm over a list of points expressed as [[x1,y1],[x2,y2],…] and returns the list of clusters it finds, where each cluster is the list of points that belong to it. The distance argument defines the neighbourhood radius around a point, and minPoints is the minimum number of nearby points required to form a cluster.

The Rule below ingests YOLO frames via the com.vantiq.OccupancyService package's DetectionEvent inbound event, reduces each person detection to the centre point of its bounding box, and asks DBSCAN for clusters of at least three people within a 75-pixel radius. For every cluster it finds, it emits a PeopleClusterDetected outbound event:

```
package com.vantiq.OccupancyService

RULE DetectPeopleClusters

WHEN EVENT OCCURS ON "/services/com.vantiq.OccupancyService/DetectionEvent"
AS frame
   // Reduce each "person" detection to the centre of its bounding box
   var points = []
   for (obj in frame.value.detections) {
      if (obj.label == "person" && obj.confidence >= 0.5) {
         var x = obj.boundingBox.left + (obj.boundingBox.width  / 2)
         var y = obj.boundingBox.top  + (obj.boundingBox.height / 2)
         push(points, [x, y])
      }
   }
   // Only cluster when there are enough people to be meaningful
   if (points.size() >= 3) {
      // 75-pixel neighbourhood; at least 3 people to form a cluster
      var clusters = MotionTracking.dbscanCluster(75, 3, points)
      for (cluster in clusters) {
         PUBLISH {
            camera:      frame.value.cameraId,
            clusterSize: cluster.size(),
            members:     cluster,
            detectedAt:  now()
         } TO SERVICE EVENT
"com.vantiq.OccupancyService/PeopleClusterDetected"
      }
   }
```

Each entry in clusters is itself a list of [x,y] points, so cluster.size() is the number of people in that group and the points themselves can be passed downstream for visualisation or escalation. Tuning comes down to two numbers: increase distance to treat people who are further apart as part of the same group, and raise minPoints to ignore smaller gatherings. This is a good example of logic that is far more natural to write as a few lines of VAIL than to assemble from visual Activity Patterns.

3. Predictive AI: Anticipating and Detecting Complex Patterns and Anomalies

Predictive AI refers to the use of machine learning models trained on historical data to discover hidden patterns, relationships, and correlations, and to learn statistical relationships and complex patterns directly from data. These models enable applications to detect anomalies that are uncertain or are “shades of grey,” and to identify complex patterns that emerge across multiple signals—even when no explicit rule has been violated. Rather than relying on static rules that react to events as they occur, predictive AI enables systems to recognise early warning signals, identify deviations from expected patterns, and act proactively in more complex or subtle ways. This capability is especially important in real-time, event-driven systems, where early detection and foresight can significantly improve responsiveness, safety, and efficiency.

Predictive models identify patterns, trends, and correlations across large and often high-velocity data streams—patterns that are difficult or impossible to capture with static rules alone. This includes subtle deviations, grey-area behaviours, and complex anomalies that do not violate any single threshold but emerge only when many signals are considered together. These models are continuously trained with new data, allowing system behaviour to evolve as conditions, environments, and usage patterns change.

Unlike rules, predictive AI can recognise early warning signals and weak indicators of future issues, even when individual metrics appear normal in isolation. This makes it particularly effective for detecting emerging risks, latent failures, or novel behaviours that have not been explicitly defined in advance.

Common predictive use cases include:

- Forecasting future states (e.g., demand, load, or system health)
- Estimating likelihood or risk (e.g., fraud probability, failure likelihood)
- Classifying situations based on learned patterns
- Detecting anomalies and emerging behaviours

3.1 Detecting What Knowledge-based Rules Cannot

A critical advantage of machine learning–based predictive AI is its ability to detect anomalies that are not explicitly defined in advance. Rule-based logic depends on known thresholds and conditions—*if X happens, do Y*. While effective for well-understood and repeatable scenarios, rules assume that abnormal behaviour is already known and describable.

Machine learning models take a different approach. They learn what *normal* looks like by observing historical and live data and can identify subtle deviations from that baseline—even when no individual rule has been violated. These anomalies may involve complex signal combinations, gradual behavioural drift, or rare edge cases that would be impractical or impossible to encode manually.

Examples include:

- Equipment behaving slightly differently across multiple sensors, even though no single threshold is exceeded
- Behavioural or usage patterns that are statistically unusual but do not violate predefined business rules
- Subtle changes in activity that indicate elevated risk before a clear incident occurs

In event-driven systems, ML-detected anomalies can be emitted as first-class events, enriched with confidence scores, severity estimates, or predicted outcomes. These events enable earlier intervention, smarter prioritisation, and more effective human-in-the-loop decision-making.

3.2 The Role of Computer Vision in Predictive AI

Computer vision (CV) is a powerful application of machine learning in real-time systems because it transforms raw visual input into structured, semantically meaningful information. Cameras and video streams are among the richest real-time data sources available, capturing contextual and environmental information that traditional sensors cannot.

Modern computer vision models can detect, classify, and track objects, people, and activities, producing high-level semantic events such as:

- Detect and classify objects
- Track movement and spatial relationships
- Estimate pose or activity
- Segment scenes into meaningful regions
- Assign confidence scores to detections

The output of a computer vision model is an interpretation of the current visual state — generally expressed as objects, labels, coordinates, bounding boxes, classifications, and associated confidence values.

Downstream processing of the results of CV event data can be used to detect situations of interest, detect temporal patterns, and infer behaviours

- Repeated detection of a person in a restricted area may contribute to a risk model
- Changes in motion patterns across frames may indicate abnormal behaviour
- Object absence or misplacement over time may trigger anomaly detection

Computer vision outputs are typically emitted as structured events enriched with confidence scores. These events can then feed rules, other statistical models, or higher-level reasoning systems to support anomaly detection, risk estimation, or goal-driven workflows.

3.3 Why Predictive AI Matters in Real-Time Systems

In real-time, situationally aware systems, speed alone is not enough—foresight matters. Predictive AI shifts systems from reactive behaviour (“something happened”) to anticipatory behaviour (“something is about to happen”). This enables:

- Earlier intervention, reducing risk and cost
- More intelligent prioritisation of actions and resources
- Fewer false positives through context-aware anomaly detection
- More effective collaboration by presenting likely outcomes, not just data

By combining rules for known conditions with machine learning models that detect unknown or emerging patterns, predictive AI delivers both **precision** and **adaptability**. Together—especially when augmented with computer vision—these capabilities form a critical foundation for intelligent, real-time, and situationally aware systems.

Predictive AI, including both traditional machine learning and computer vision–based models, are especially valuable when integrated with Vantiq because its outputs align naturally with Vantiq’s real-time, event-driven architecture. ML models can detect trends, risks, and anomalies—often before explicit thresholds are crossed—while computer vision models

transform raw visual streams into high-level semantic events that describe what is happening in the physical world.

3.4 Strengths and Limitations of Predictive AI

Predictive AI offers capabilities that go beyond knowledge-based rules by identifying patterns, trends, correlations, and weak signals in complex data. When applied appropriately, it significantly enhances situational awareness and proactive response in real-time systems.

Predictive models offer several critical strengths:

- Detection of subtle and complex patterns across many signals simultaneously
- Early warning capability, identifying risks before explicit thresholds are crossed
- Adaptability, since models can be retrained as environments and behaviours evolve
- Probabilistic reasoning, enabling prioritisation through confidence scores or risk estimates
- Improved anomaly detection, especially for grey-area or emerging behaviours

Unlike rules, which assert certainty based on predefined conditions, predictive models estimate statistical likelihood. This makes them particularly effective in environments where behaviour is dynamic, ambiguous, or too complex to describe exhaustively in advance.

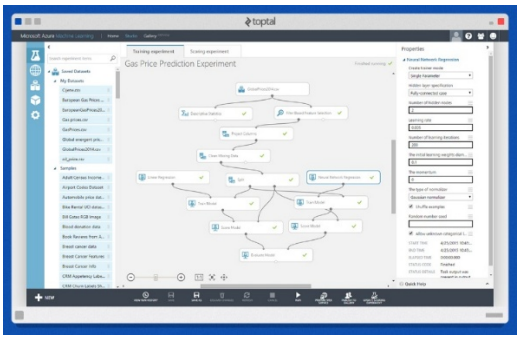
However, predictive AI also introduces important trade-offs and operational considerations:

- Dependence on training data quality, including risks of bias or incomplete representation
- Model drift, where performance degrades as real-world conditions change
- Retraining and lifecycle management overhead, including monitoring and version control
- Higher computational cost compared to simple rule evaluation
- Reduced explainability, particularly for complex models such as deep learning systems

It is important to clarify the role of determinism in predictive systems. At inference time, a trained model is mathematically deterministic: given the same input and fixed model parameters, it will produce the same output. However, unlike rule-based logic, predictive systems evolve over time. Models may be retrained, recalibrated, or adapted as new data becomes available, potentially changing their outputs for the same inputs.

3.5 Predictive AI and Vantiq

Predictive AI is implemented externally to Vantiq and can be integrated via either built-in sources, such as Azure ML Studio or Apache Camel's Deep Java Library component, or, more commonly, through REST APIs to a variety of ML platforms. These connections are then very easy to integrate into the event flow via synchronous procedure calls.



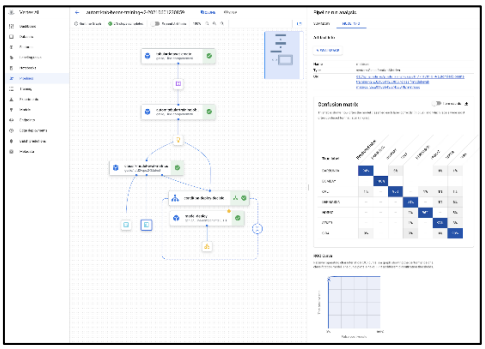


Figure 7: ML Studio & Google Vertex AI

The integration options described above are best understood through a worked example that continues the chiller-plant pump introduced in Section 2. The Linear Regression handler there reasoned about a single signal—temperature—and predicted when it would cross a fixed alarm threshold. Many real failure modes are not visible in any single signal: a pump can drift toward bearing wear or cavitation while its temperature, vibration, flow rate, motor current, and inlet and outlet pressures each remain comfortably inside their individual limits, and only the combination of those readings reveals that something is wrong. Detecting that kind of multivariate, "shades-of-grey" anomaly is beyond a single threshold or a single-variable regression and is a natural fit for an externally trained machine-learning model invoked from the event flow.

3.5.1 Azure ML Studio Example: Pump Anomaly Detector

3.5.1.1 The Model Side: Training a Multivariate Anomaly Detector in Azure Machine Learning

Rather than encoding what "too hot" or "too much vibration" means, the developer trains a model on what normal operation looks like across all of the pump's signals at once. Given a history of readings recorded while the pump was healthy, the model learns the shape of normal operation—the way the signals move together—and, at inference time, returns an anomaly score expressing how far a new reading departs from that learned normal. A reading whose individual values are all in range, but whose combination is unfamiliar produces a high score, which is precisely the case a threshold rule cannot catch.

A common and easily explained formulation reconstructs each reading from a compressed internal representation and measures the reconstruction error. Readings that resemble normal operation reconstruct accurately and score low; unfamiliar combinations reconstruct poorly and score high. That error is the anomaly score, and a threshold on the score—calibrated against known-normal data—is the tolerance that separates acceptable variation from a flagged anomaly. The same integration pattern applies to whatever technique produces the score, whether principal-component reconstruction, an isolation forest, a one-class support vector model, or an autoencoder.

In Azure Machine Learning the model can be assembled without code in the drag-and-drop Designer, where a training pipeline is built by connecting components on a canvas—typically a normal-operation dataset, a normalisation step, the chosen training component, a scoring component, and an evaluation step—or it can be authored directly with the Python SDK by teams that prefer code. Either way the result is a trained model registered in the workspace.

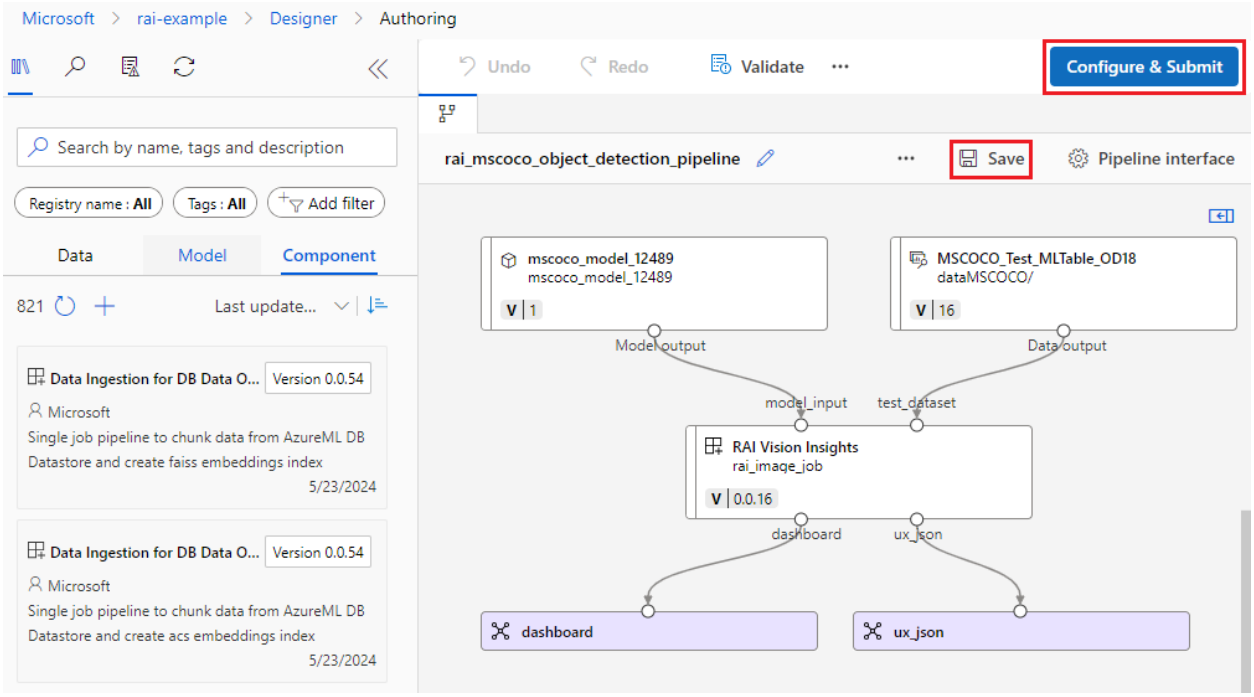


Figure 8: An Azure Machine Learning Designer training pipeline (illustrative)

Once trained and validated, the model is deployed as a managed online (real-time) endpoint. Azure exposes the endpoint as a REST service with a scoring URI and an authentication key and provides a test pane for sending a sample reading and inspecting the returned score before any external client is connected.

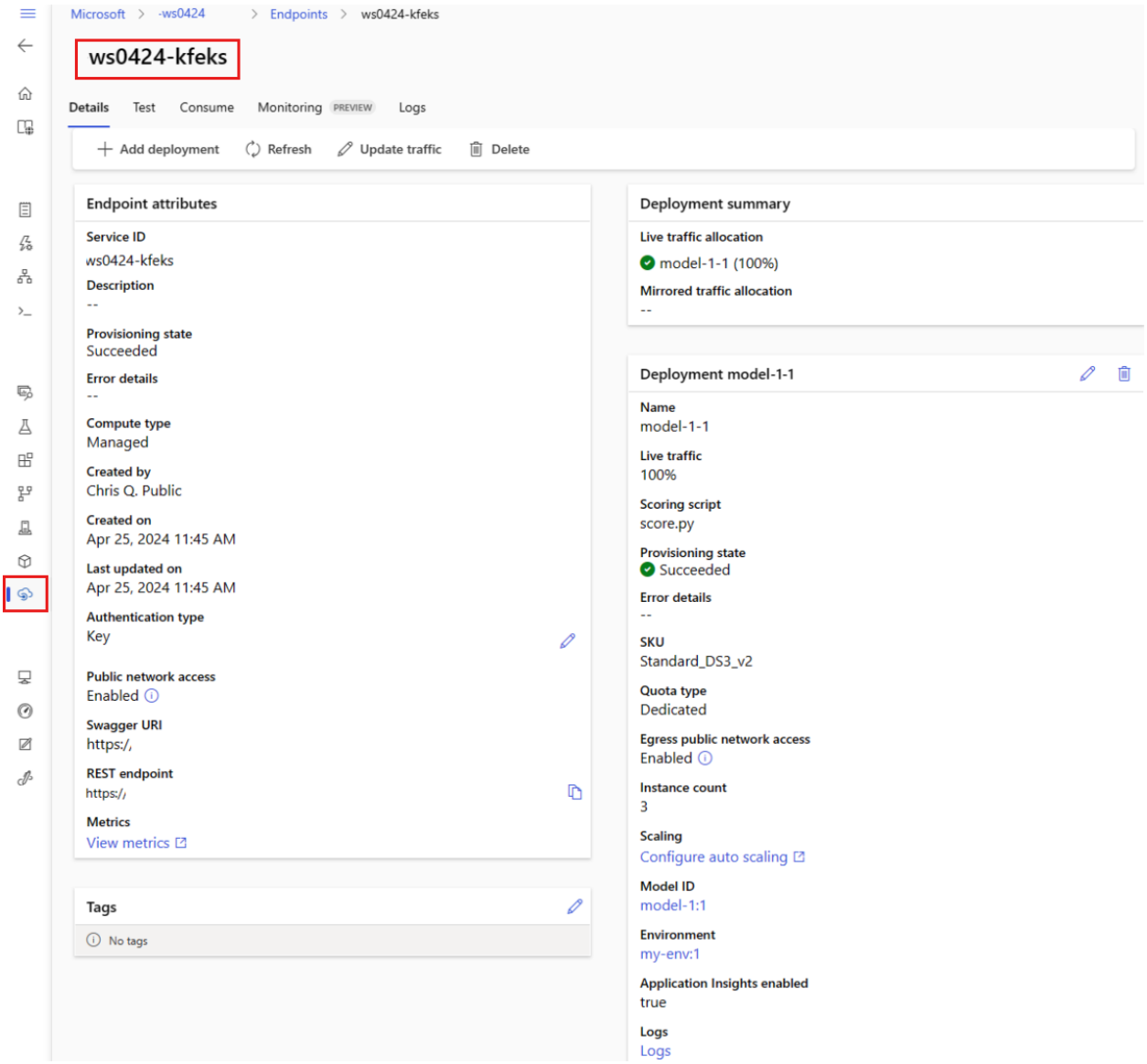


Figure 9: The deployed model exposed as a real-time scoring endpoint (illustrative)

The endpoint defines a simple contract. It accepts a JSON document containing one feature vector per reading—the pump's sensor values in the order the model was trained on—and returns the anomaly score together with, optionally, a flag indicating whether the score exceeded the model's calibrated tolerance. The exact field names are determined by the model's scoring script; a representative request is:

```
{
  "input_data": {
    "columns": [ "tempC", "vibration", "flowRate", "motorCurrent",
"inletPressure", "outletPressure" ],
    "data":     [ [ 58.2, 3.4, 118.0, 14.1, 2.05, 4.30 ] ]
  }
}
```

and the response, for example:

```
{ "anomaly_score": 0.87, "is_anomaly": true }
```

3.5.1.2 The Vantiq Side: Calling the Model over REST

On the Vantiq side, the deployed endpoint is reached like any other REST service. As noted above, Vantiq offers a built-in Microsoft MLStudio Analytics Source and, more commonly, a generic REMOTE source for calling REST APIs. The REMOTE source is used here because it works directly against the managed online endpoint: it is configured with the endpoint's scoring URI as its base address and the authentication key supplied as an Authorization: Bearer header, with Content-Type set to application/json. Because scoring is a request/response interaction, the handler calls the source synchronously and receives the score in line.

The handler mirrors the shape of the Linear Regression example. An Event Stream Task at the root receives each pump telemetry event; a Split By Group Task partitions the stream by pumpId so that every pump is scored independently; a Transformation Task assembles the feature vector in the exact column order the model expects; a Procedure Task calls the model and reads back the score; and a Branch Task compares the score against the tolerance, routing only the readings that exceed it to a Publish Task that emits a PumpAnomalyDetected event enriched with the score and a severity band.

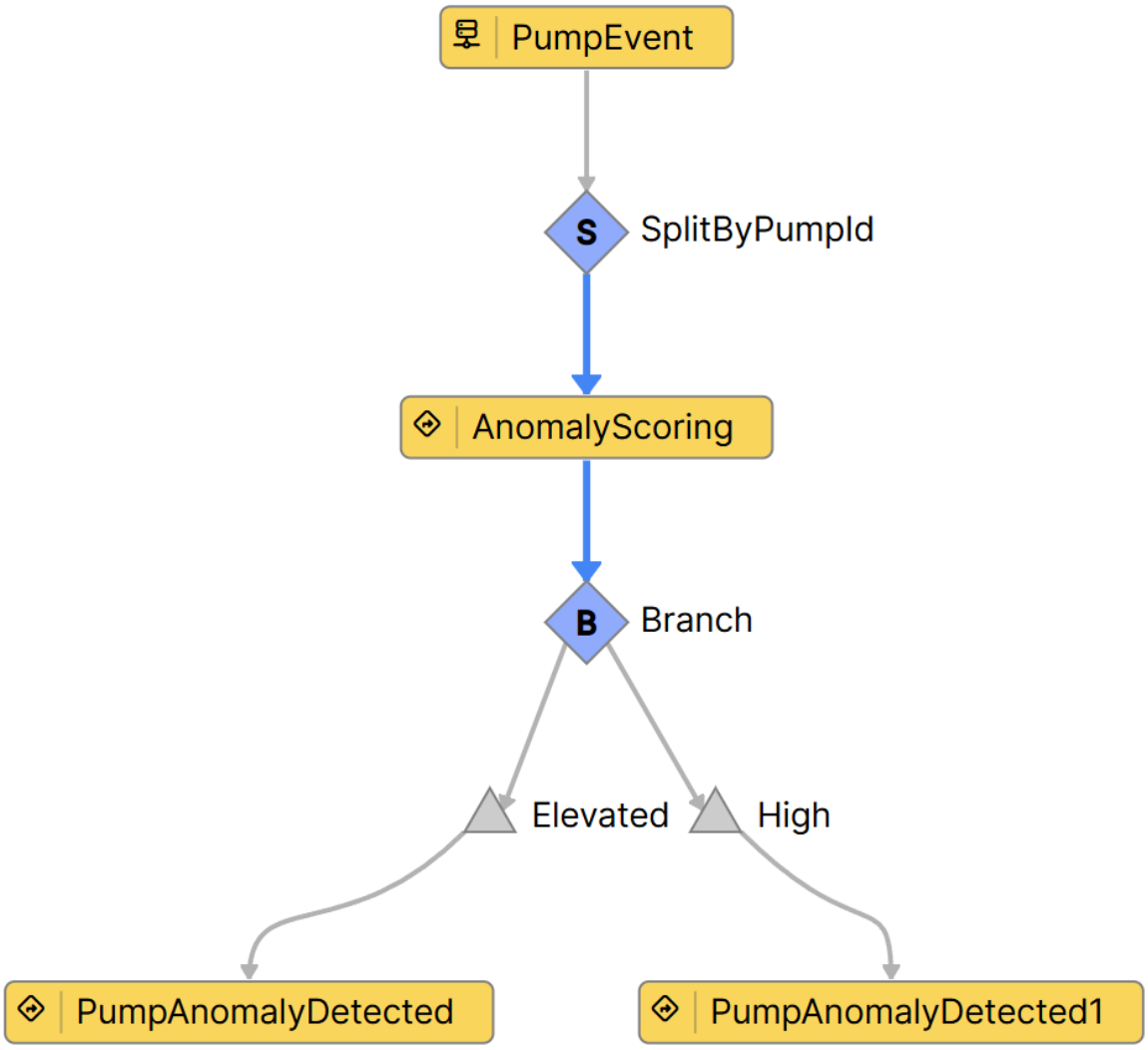


Figure 10: VEH that scores each pump reading against the Azure ML model

The Procedure Task calls the model with a single synchronous SELECT FROM SOURCE statement, passing the feature vector as the request body and reading the score from the response:

```
PROCEDURE scorePumpReading(event Object)
// Assemble the feature vector in the model's trained column order
var features = [ event.tempC, event.vibration, event.flowRate,
                 event.motorCurrent, event.inletPressure,
event.outletPressure ]
var request = { input_data: {
                   columns: [ "tempC", "vibration", "flowRate",
                              "motorCurrent", "inletPressure",
"outletPressure" ],
                   data:    [ features ] } }
```

```
// Synchronous request/response call to the Azure ML scoring endpoint
var response = SELECT FROM SOURCE PumpAnomalyModel WITH body = request
var score = response.anomaly_score
// The tolerance: scores above this threshold indicate a multivariate anomaly
if (score > 0.6) {
   var severity = "elevated"
   if (score > 0.85) { severity = "high" }
   PUBLISH {
      pumpId:     event.pumpId,
      score:      score,
      severity:   severity,
      reading:    event,
      detectedAt: now()
   } TO SERVICE EVENT "com.vantiq.PumpService/PumpAnomalyDetected"
}
```

Because the score arrives in the same synchronous call, the handler acts on it immediately—suppressing the noise of in-range readings and emitting only meaningful anomalies as first-class events enriched with a score and severity, exactly the pattern described earlier in this section. Downstream handlers, rules, or an operator dashboard consume PumpAnomalyDetected without needing to know that an external model produced it.

3.5.2 A Computer-Vision Example: Detecting People with NVIDIA Triton over the Open Inference Protocol

Section 3.2 described how computer vision models convert raw video into structured semantic events with confidence scores. This example shows that pattern end-to-end, and illustrates a second, complementary integration shape. Where the anomaly detector above was a model Vantiq called synchronously for a single score, a computer-vision pipeline begins with a live video stream and sends each frame to a deep-learning model hosted on a dedicated inference server. The integration uses the Open Inference Protocol (OIP)—a standard HTTP/REST contract for model inference—so the same calling pattern works with the NVIDIA Triton Inference Server used here or any other OIP-compliant engine, and the model can be retrained or replaced without changing the Vantiq code.

Two sources are involved. A VIDEO source represents the camera and produces image frames; a REMOTE source represents the Triton server and carries the inference requests. In streaming mode, the VIDEO source is configured with the camera address and a polling interval that determines how often a frame is captured and delivered as an event:

```
{
   "ipCamera": "rtsp://entrance-camera.local:554/stream",
   "pollingInterval": "1 second",
   "requestDefaults": {
      "query": {
```

```
            "contentType": "image/jpeg",
            "resize": { "maxHeight": 640, "maxWidth": 640 }
        }
    }
}
```

The REMOTE source points at the Triton endpoint and enables the OIP binary-tensor extension, which transmits the image bytes directly rather than base64-encoding them into JSON:

```
{
    "requestDefaults": { "openInference": true },
    "uri": "http://triton.local:8000"
}
```

The handler is built from five tasks, an Event Stream Task at the root receives each frame event from the VIDEO source; a Procedure Task sends the frame to the model over OIP and returns the raw array of detections; an Unwind Activity Pattern splits that array so each detection flows downstream as its own event; a second Procedure Task converts each detection from the positional tensor row it arrives as into a named-field object; and a Filter Task keeps only the detections whose class is a person and whose confidence is 0.7 or greater.

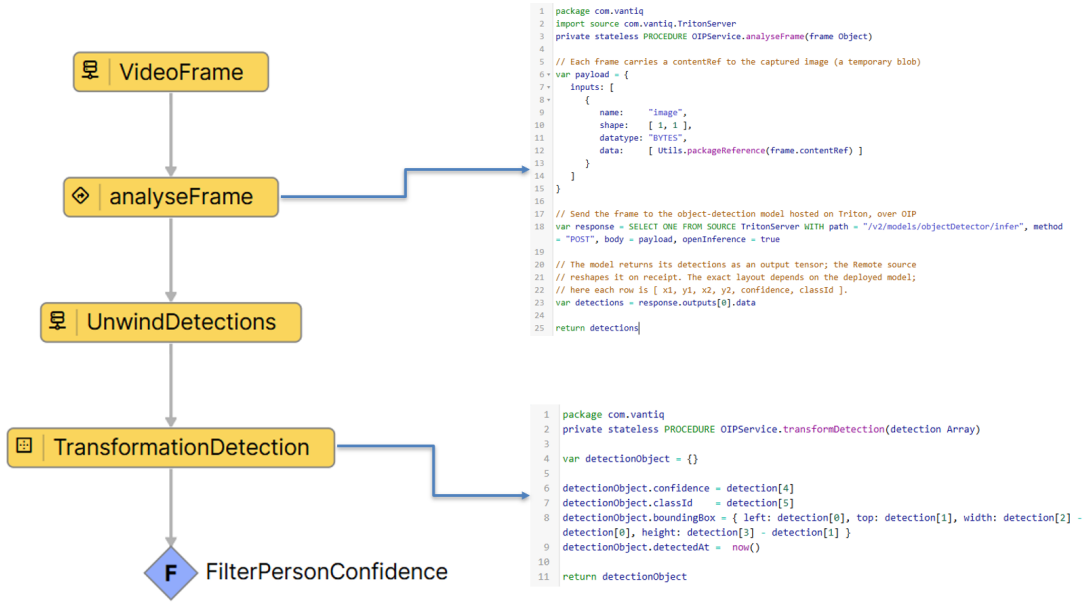


Figure 11: Computer-vision pipeline: video frames scored by an NVIDIA Triton model over the Open Inference Protocol

The first Procedure Task hands the image to the model. Every frame already carries a contentRef—a reference to the captured image, held briefly as a temporary blob—which is passed to Triton as a BYTES input tensor with Utils.packageReference, so the image never has to be decoded or re-encoded in VAIL:

```
private stateless PROCEDURE OIPService.analyseFrame(frame Object)
// Each frame carries a contentRef to the captured image (a temporary blob)
var payload = {
```

```
   inputs: [
      {
         name:     "image",
         shape:    [ 1, 1 ],
         datatype: "BYTES",
         data:     [ Utils.packageReference(frame.contentRef) ]
      }
   ]
}
// Send the frame to the object-detection model hosted on Triton, over OIP
var response = SELECT ONE FROM SOURCE TritonServer WITH path =
"/v2/models/objectDetector/infer", method = "POST", body = payload,
openInference = true
// The model returns its detections as an output tensor; the Remote source
// reshapes it on receipt. The exact layout depends on the deployed model;
// here each row is [ x1, y1, x2, y2, confidence, classId ].
var detections = response.outputs[0].data
return detections
```

The procedure returns the raw detections array exactly as the model produced it—each row a positional tensor of the form [x1, y1, x2, y2, confidence, classId]. The Unwind Activity Pattern downstream splits that array so each detection flows on as its own event, ready for the next stage to give it a more usable shape. Once converted and filtered, these are precisely the structured, confidence-scored events Section 3.2 describes, and they share the shape consumed by the people-clustering rule in Section 2—so the same detections can drive in-handler clustering, feed knowledge-based rules, or be summarised by higher-level reasoning, with no downstream consumer needing to know that an external GPU inference server produced them.

Because OIP is a standard contract, the same REMOTE source also reaches the server's management endpoints: a handler can confirm the server is live with a GET on /v2/health/live and that a model is loaded and ready with /v2/models/{model}/ready before sending traffic, and can read a model's input and output tensor metadata from /v2. Swapping the detector for a pose-estimation or segmentation model, or moving from Triton to a different OIP-compliant engine, changes only the model name in the request path.

The second Procedure Task converts each detection from the positional row it arrives as—[x1, y1, x2, y2, confidence, classId]—into a named-field object with classId, confidence, boundingBox, and detectedAt fields, so that the final Filter Task downstream can express its predicate in business terms—class is a person, confidence is at least 0.7—and forward only those events:

```
private stateless PROCEDURE OIPService.transformDetection(detection Array)
var detectionObject = {}
detectionObject.confidence = detection[4]
detectionObject.classId    = detection[5]
```

```
detectionObject.boundingBox = { left: detection[0], top: detection[1], width:
detection[2] - detection[0], height: detection[3] - detection[1] }
detectionObject.detectedAt =  now()


return detectionObject
```

4. Generative AI: Creating Meaningful Outputs from Context and Knowledge

Generative AI refers to a class of artificial intelligence systems that are capable of **producing new content or responses** based on context, intent, and learned knowledge. Unlike rules, which encode explicitly known behaviour, or predictive models, which estimate likely outcomes or detect anomalies, generative AI constructs outputs dynamically at runtime.

These outputs may include natural language text, summaries, explanations, structured data, recommendations, plans, and other generated artefacts. Rather than selecting from a predefined set of responses, generative AI synthesises information to produce responses tailored to the specific context in which it is invoked.

This capability makes generative AI particularly well-suited to environments where information is incomplete, unstructured, or ambiguous, and where human understanding and interaction are central.

4.1 Generating Rather Than Selecting

Traditional software systems—and many forms of AI—operate by choosing between known options: a rule fires, a threshold is crossed, or a model assigns a classification. Generative AI takes a fundamentally different approach. It **generates** responses rather than selecting them.

Given a combination of inputs such as:

- Current events and system state
- Historical data or prior interactions
- Domain knowledge and reference material
- User intent expressed in natural language

A generative model produces a response tailored to that context. Two similar situations may therefore result in different outputs, reflecting subtle differences in intent, timing, surrounding conditions or the inherent probabilistic nature of Generative models.

This ability to construct responses dynamically allows systems to move beyond rigid interaction patterns and toward more flexible, adaptive behaviour.

4.2 Natural Language Understanding and Generation

A defining capability of generative AI is its ability to both **understand and produce natural language**. This enables systems to interact with users in a way that more closely resembles human communication, rather than requiring structured queries, fixed commands, or complex user interfaces.

Generative AI can:

- Interpret user questions, instructions, or explanations expressed in plain language
- Extract intent, sentiment, and relevant entities from unstructured text
- Generate clear, context-aware responses, explanations, or summaries
- Translate between technical system representations and human-readable language

This makes generative AI a powerful interface layer for complex systems, allowing users to engage with sophisticated functionality without needing deep technical knowledge of the underlying implementation.

4.3 Multimodal Generative AI: Images, Video, and Audio

Modern generative AI systems are increasingly **multimodal**, meaning they can interpret and generate content across multiple types of data, not just text. In addition to natural language, generative models can work with images, video, and, in some cases, audio.

Multimodal generative AI can:

- Interpret images or video frames to understand scenes and detect behaviours beyond simply detecting objects
- Generate images or visual artefacts based on textual descriptions or contextual input
- Summarise or explain video content in natural language
- Extract meaning from visual or audio signals and combine it with other contextual information

For example, multimodal models can detect complex behaviour in a video, such as car crashes or fights. Object-detection models cannot detect such complex behavioural situations. These models also allow users to ask questions about an image, so they not only detect known objects but can also query and interrogate the image—how many people are in the frame, what models of car are involved in the crash, and so on.

4.4 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (**RAG**) is helpful to a large language model because it grounds the model's reasoning in relevant, authoritative information at the time of the request. Instead of relying solely on patterns learned during pretraining—which may be outdated, incomplete, overly general, or, in the case of proprietary knowledge, missing entirely—the model can retrieve current, domain-specific, or proprietary knowledge and incorporate it directly into its response.

This improves accuracy by reducing hallucinations, increases relevance by focusing the model on contextually appropriate information, and enables the use of data not included in the original training corpus. RAG also allows organisations to update knowledge sources without retraining the underlying model, making it far more practical for environments where information changes frequently. By coupling generative reasoning with targeted retrieval, RAG enhances factual grounding, contextual precision, and adaptability.

In a RAG-based approach, relevant information is retrieved at runtime from external sources—such as documents, knowledge bases, or vector databases—and provided to the generative model as contextual input. The model then uses this retrieved information, along with its general language understanding, to generate responses grounded in authoritative, up-to-date data.

Typical RAG workflows involve:

- Indexing domain knowledge (e.g., documents, manuals, incident reports, policies)
- Converting that knowledge into embeddings stored in a vector database
- Retrieving the most relevant information based on semantic similarity to the current query or situation
- Supplying the retrieved content as context to the generative model

This approach enables generative AI systems to produce responses that are not only fluent but also **accurate, contextual, and aligned with organisational knowledge**.

4.4.1 Benefits of RAG in Generative AI Systems

Retrieval-Augmented Generation provides several important benefits:

- **Improved Accuracy and Relevance**: Responses are grounded in real, domain-specific information rather than relying solely on the model's general training.
- **Up-to-Date Knowledge**: New or changing information can be incorporated simply by updating the knowledge source, without retraining models.
- **Explainability and Trust**: Generated outputs can reference or be traced back to retrieved source material, improving transparency and user confidence.
- **Scalability of Knowledge**: Large volumes of unstructured information can be made accessible through natural language queries without manual curation into rules or workflows.

RAG is especially effective in environments where users need clear, context-aware explanations or guidance derived from large and evolving knowledge bases.

4.5 Adaptability and Personalisation

Because generative AI responses are constructed dynamically, they can be adapted to the specific user, role, or situation. The same retrieved knowledge or interpreted visual content can be expressed differently depending on context.

For example, generative AI can:

- Adjust the level of detail based on the user's expertise
- Tailor explanations or recommendations to a specific role or responsibility
- Adapt tone and structure to suit operational, analytical, or instructional use
- Incorporate recent interaction or situational context into responses

This adaptability enables more personalised and relevant interactions without requiring separate logic paths or duplicated knowledge.

4.6 Generative AI as a Cognitive Load Reducer and Situational Awareness Enabler

In intelligent, event-driven systems, generative AI plays a critical role in **reducing cognitive load** for users operating in complex, information-rich environments. Rather than expecting users to manually navigate data, dashboards, documents, images, or alerts, generative AI shifts the burden of interpretation and synthesis onto the system itself.

This capability directly supports **situational awareness**, as described earlier in this document. Situational awareness requires not only access to real-time data, but also the ability to understand context—what is happening, why it is happening, and what it means right now. Generative AI contributes by transforming raw signals into coherent, context-aware understanding.

By interpreting unstructured information across text, images, video, and potentially audio, retrieving relevant knowledge at runtime, and synthesising context into clear, concise outputs, generative AI helps users focus on outcomes rather than process.

Generative AI reduces cognitive load and enhances situational awareness by:

- Transforming large volumes of raw data and events into concise explanations or summaries
- Connecting related signals across time, space, and systems into a single, coherent narrative
- Incorporating historical, environmental, and contextual factors into generated responses

- Presenting only the most relevant information for the current situation and user role Explaining *why* something matters, not just *what* happened

This allows users to quickly form an accurate mental model of the current situation without manually correlating events, metrics, documents, or visual inputs. Instead of forcing users to assemble situational understanding themselves, the system actively supports comprehension in real time.

These gains look different from one role to the next. A few examples:

- **Field service technician servicing a respirator** arrives to a plain-language summary of the unit's recent alarms, fault codes, and usage history, with the relevant manual steps, configuration, and likely cause already gathered, instead of scrolling through raw logs and searching several manuals on site.
- **ICU bedside nurse** receives a concise narrative of each patient's trend and significant events, with the latest labs, medications, and correlated vitals drawn together and explained, instead of reconstructing the picture from separate monitors, charts, and systems.
- **Plant control-room operator** reads a single situation summary that connects alarms across pumps, sensors, and cameras, surfaces only what is relevant to the current incident, and names the probable cause and affected assets, so triage begins from understanding rather than from a raw alarm list.

4.6.1 Gathering the missing context automatically

A summary is only as useful as the information behind it, and the most relevant facts are often not contained in the event that triggered it. A generative model, therefore, does not have to work only from the data carried by the triggering event. Modern models can be given a set of tools — read-only functions the model may call while it is composing its response — so that it can gather the additional real-time information it needs to produce a complete and accurate summary. Rather than the developer having to anticipate and pre-assemble every fact the model might require, the model itself determines what is missing and retrieves it on demand.

When asked to summarise an unfolding situation, the model can call one or more of these tools, fold the results into its reasoning, and produce a summary that reflects the state of the system at that moment — not just the single event that prompted it. These are information-gathering calls only: the model is reading the current state to inform its narrative, not changing that state or taking action on the user's behalf. (Allowing a model to act, rather than only to observe, is the step into agentic AI, discussed in the next section.)

Recognising that data is stale, knowing where the authoritative value resides, and fetching it is exactly the kind of manual correlation that overwhelms operators in information-rich environments. By letting the model gather that context itself, the user is presented with a single, current, self-contained explanation and is freed to focus on the decision rather than the data-gathering that precedes it.

4.7 Strengths and Limitations of Generative AI

Generative AI is a class of intelligent systems capable of dynamically producing new content, interpretations, and structured outputs at runtime. Rather than retrieving predefined responses, documents, or lexical search results, generative systems construct outputs based on context, intent, retrieved knowledge, and learned representations from large-scale training.

This capability makes generative AI particularly valuable in environments characterised by complexity, ambiguity, unstructured information, and heavy human interaction.

4.7.1 Strengths of Generative AI

Generative AI offers distinct strengths that expand the capabilities of intelligent systems.

- **Contextual Synthesis Across Diverse Inputs**: Generative models can combine multiple forms of input—structured data, unstructured documents, historical context, user prompts, and multimodal signals—into a coherent, context-aware response. This allows systems to interpret situations holistically rather than processing signals in isolation.
- **Dynamic Output Construction**: Outputs are generated at runtime and tailored to the specific context in which the model is invoked. Responses can adapt based on timing, user role, environmental conditions, or newly retrieved information. This enables flexible and highly contextual behaviour without requiring predefined response templates.
- **Natural Language Understanding and Interaction**: Generative AI enables systems to interpret and produce natural language, allowing users to interact conversationally rather than through rigid command structures or predefined interfaces. This reduces technical barriers and improves accessibility across roles and skill levels.
- **Reduction of Cognitive Load**: In complex operational environments, users are often overwhelmed by dashboards, alerts, logs, documents, and visual feeds. Generative AI can synthesise relevant information into concise explanations, summaries, and recommendations, accelerating comprehension and improving decision quality.
- **Personalisation and Role Adaptation**: Generated outputs can be adapted to different users or stakeholders. The same underlying situation may be explained differently to an executive, an operator, or a technical specialist, adjusting tone, detail, and focus dynamically.
- **Multimodal Reasoning**: Modern generative systems increasingly operate across text, images, and other modalities. They can interpret visual inputs, summarise multimedia content, and combine different data types into unified explanations.

Together, these strengths make generative AI especially effective in scenarios that require interpretation, explanation, planning support, and user-facing intelligence.

4.7.2 Limitations and Trade-Offs of Generative AI

Despite its expressive power, generative AI introduces important architectural and operational considerations.

- **Probabilistic Behaviour**: Generative models produce outputs based on probabilistic token selection. While configuration settings can reduce variability, outputs are inherently influenced by sampling parameters and context. This can introduce variation across runs and is inherently nondeterministic.
- **Hallucination Risk:** Generative systems may produce responses that are fluent but factually incorrect, incomplete, or fabricated if not properly grounded in reliable data sources. Without retrieval-based augmentation, generated content may appear authoritative while lacking factual accuracy.
- **Sensitivity to Prompt and Context Design**: Output quality depends heavily on the structure, clarity, and completeness of the prompt and contextual information provided. Poorly constructed prompts or insufficient grounding can degrade reliability.

- **Limited True Understanding**: Although generative AI can simulate reasoning and explanation, it does not possess intrinsic comprehension or causal understanding. Its outputs reflect learned statistical patterns rather than validated logical inference.
- **Explainability of Internal Mechanisms**: While outputs can be explained in natural language, the internal reasoning path of large-scale generative models is not directly transparent. This may limit traceability in environments requiring strict auditability.
- **Computational and Cost Overhead**: Large generative models require substantial computational resources relative to traditional application logic. Latency and cost must be managed carefully, particularly in high-throughput or real-time environments.
- **Governance and Data Security Considerations**: Deployments must address data privacy, secure handling of proprietary information, model access controls, and compliance requirements. When integrating external models, architectural safeguards are essential.

4.8 Generative AI and Vantiq

Vantiq provides a rich set of Generative AI resources and capabilities, including integration with a large set of Generative and Embedding models, support for Semantic Index and RAG through the integration with a vector database, unstructured content ingestion services, and built-in patterns such as submitPrompt and answerQuestion, allowing easy integration into the visual event flow. Vantiq also provides a rich low-code tool for constructing more advanced generative AI logic using the GenAI Builder, which can be integrated into the visual event flow or accessed programmatically. In addition to backend support for Generative AI, the integrated Client Builder provides built-in widgets that enable developers to include chat interfaces in their web and mobile applications.

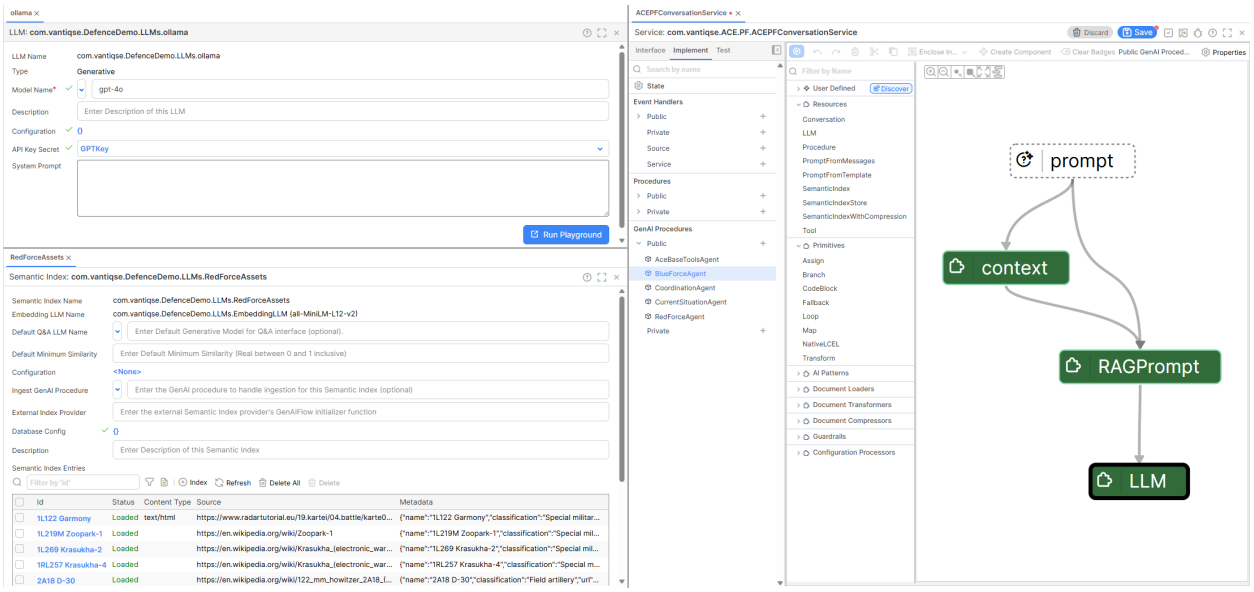


Figure 12: Generative AI & Vantiq

4.8.1 A Worked Example: Turning a Pump Anomaly into a Grounded, Plain-Language Alert

The pump example developed in Sections 2 and 3 already produces a high-level perception event: the Linear Regression handler warns that temperature is trending toward the alarm, and the externally trained model raises a PumpAnomalyDetected event carrying a score and

severity. What neither produces is an explanation a busy on-call engineer can act on without stopping to interpret telemetry and search the manuals. This is where generative AI fits: it adds an understanding-and-interaction layer on top of perception, turning the raw signals into a grounded recommendation and a concise, role-appropriate alert—the cognitive-load reduction described in Section 4.6.

Two generative resources are defined: a generative LLM (for example, openai/gpt-4o or an Azure OpenAI deployment) and an embedding LLM for building a semantic index. A Semantic Index—call it PumpDocumentationIndex—is created over the plant's own pump documentation: manufacturer operating and troubleshooting manuals, maintenance standard operating procedures, and past incident and repair reports. Vantiq ingests this content (by upload, by URL, or from Vantiq Documents), splits it into overlapping chunks, embeds it, and stores the vectors for retrieval. This index grounds the model's answers in authoritative, organisation-specific knowledge rather than in its general training—the Retrieval-Augmented Generation (RAG) approach of Section 4.4.

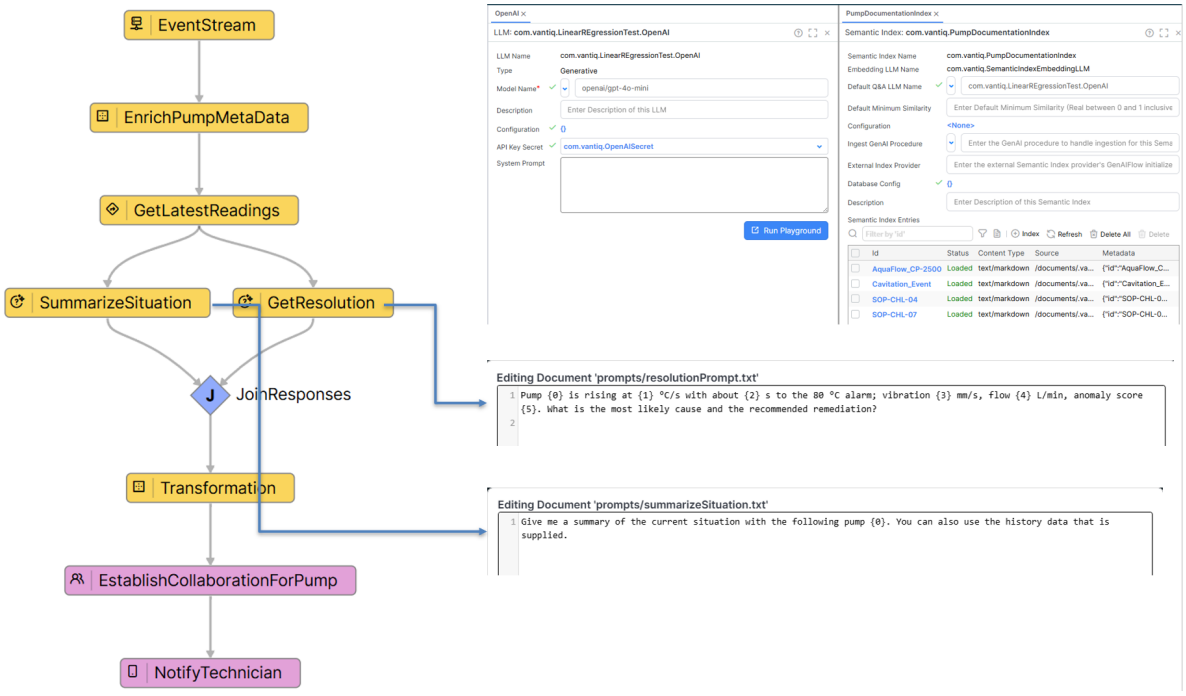


Figure 13: Generative AI worked example: a pump anomaly explained and pushed to the engineer's mobile

The Visual Event Handler uses an Answer Question task to direct an LLM to propose a resolution to the problem, drawing on the Semantic Index as its reference material so that the LLM is grounded in knowledge of the specific pump and the plant's current SOPs. A second generative pattern, Submit Prompt, then provides a concise summary of the current situation so that, rather than receiving raw data, the technician sees a grounded, narrative explanation.

The Answer Question task retrieves the most relevant manual and SOP chunks from the index and asks the qaLLM to synthesise an answer from them. Its result—held in indexResponse by default—carries three useful fields: answer, the synthesised remediation; metadata, the source passages used, so the recommendation can be traced back to a specific manual or past incident; and rephrasedQuestion. Because the answer is grounded in the indexed documents, it reflects the plant's actual equipment and history, and the citations make it auditable.

A Submit Prompt task then turns the situation into a short, human-readable alert. Its system prompt fixes the role and format—for example, "You are an assistant for plant maintenance

engineers; in three sentences state what is happening, why it matters, and the single recommended next action"—and its user prompt brings together the current pump state and the remediation produced in the previous step.

The generated summary is delivered to the on-call engineer's mobile by a Notify task—through the Vantiq mobile client built with the Client Builder, or a push or SMS notifier. Rather than a raw alarm and a telemetry chart, the engineer receives something like:

"Pump-01 is trending to overheat in about two minutes. The most likely cause is cavitation from low inlet pressure, matching incident INC-2024-118. Recommended action: open the bypass valve and inspect the inlet strainer."

The complete handler is a short, linear flow placed downstream of the perception tasks: an Event Stream Task receives the PumpAnomalyDetected event; an Enrich Task attaches the recent readings and pump metadata; an Answer Question Task produces the grounded remediation; a Submit Prompt Task turns the state and that remediation into the alert text; a Notify Task delivers it to the engineer's mobile; and a Save to Type Task records the answer and its source citations as an audit trail.

This single flow exercises several of the generative-AI capabilities described earlier: RAG grounding and traceability (Section 4.4), adaptation of the same facts to a specific role and format (Section 4.5), and the reduction of cognitive load by replacing raw data with a clear, actionable narrative (Section 4.6). It also completes the composition pattern of Section 6—knowledge-based rules and predictive models provide perception, generative AI provides understanding and communication, and the result is delivered to the point where the engineer can act on it. The same two steps can also be written programmatically, where finer control is needed, using SemanticSearch.answerQuestion for the grounded answer and io.vantiq.ai.LLM.submitPrompt for the summary, but the visual Answer Question and Submit Prompt patterns keep the flow readable and maintainable.

5. Agentic AI: Goal-Driven & Outcome-oriented

Agentic AI builds on reactive or assistive intelligence to **be goal-driven and outcome-oriented**. Rather than treating AI as something that produces answers, predictions, or generated content on demand, agentic AI focuses on carrying intent through to outcomes, especially in scenarios where the right response cannot be specified in advance and must be generated through reasoning and action.

An agentic system does not simply determine what should be done. It interprets a **goal**, constructs a **plan** to achieve it, and dynamically selects and sequences actions in response to evolving conditions. Rather than following a fixed workflow or predefined script, the system uses **generative AI** capabilities to reason about possible strategies, synthesise options, evaluate trade-offs, and decide among alternative courses of action. Planning is not a one-time activity: as actions are executed and new information becomes available, the system reassesses its assumptions, updates its understanding of the environment, and revises its plan accordingly.

5.1 What Is Agentic AI?

Agentic AI describes intelligent behaviour that exhibits the following characteristics:

- **Goal-Oriented Behaviour:** Agentic systems are driven by explicit goals and are designed to generate solutions to the problems presented as input, rather than simply applying predefined business logic or executing a fixed set of rules. Instead of mapping inputs to predetermined outputs, the system determines how to achieve the goal and dynamically constructs an approach appropriate to the situation.
- **Short-Term and Long-Term Memory:** Agentic systems rely on memory to sustain coherent behaviour across time rather than treating each interaction as independent. Short-term memory captures immediate context relevant to the current goal, such as recent events, intermediate results, pending actions, and transient observations, enabling the agent to reason coherently across multiple steps as situations evolve. Long-term memory captures enduring knowledge accumulated over time, including prior outcomes, historical interactions, learned preferences, recurring patterns, or experiential domain knowledge, allowing the agent to improve performance, avoid repeating past mistakes, and adapt behaviour based on what has previously worked or failed. Together, these memory forms enable agentic systems to reason with awareness of both recent context and accumulated experience, allowing behaviour to evolve rather than reset.
- **Autonomous Decision-Making Within Guardrails:** Agentic AI can make decisions and take action without direct human initiation at every step. That autonomy is bounded by predefined constraints such as policies, safety limits, approval requirements, or escalation rules.
- **Adaptive Execution:** Agentic systems monitor the outcomes of their actions and adjust behaviour accordingly. If an action fails, produces an unexpected result, or only partially achieves its intent, the system revises its approach rather than stopping.
- **Orchestrated Action:** Agentic AI coordinates activity across multiple services, systems, and collaborators. This may include invoking APIs, triggering workflows, interacting with external platforms, or engaging humans when needed.

In Vantiq, agentic AI is not a single feature or runtime primitive. It is an **architectural pattern** implemented using services, events, state, workflows, and AI capabilities working together.

5.2 Single-Agent and Multi-Agent Agentic Systems

An agentic system may be implemented using a single agent or multiple collaborating agents.

5.2.1 Single-Agent Systems

A single-agent agentic system owns an end-to-end objective and is responsible for all reasoning, decision-making, and coordination related to that goal. This approach is well-suited to focused domains where responsibility and authority are centralised.

For example:

> Consider an operational incident management agent responsible for resolving a system outage. When an anomaly event is detected, the agent adopts the goal of restoring service. Using short-term memory, it tracks current system state, recent alerts, actions already taken, and pending remediation steps. Using long-term memory, it recalls how similar incidents were resolved in the past, which actions were effective, and which failed. The agent gathers diagnostic data, proposes a remediation plan, executes corrective actions (such as restarting services or rerouting traffic), observes the results, and adapts its approach if the issue persists. If predefined guardrails are exceeded—such as potential customer impact or regulatory thresholds—the agent escalates to a human operator. Throughout the incident, the same agent remains responsible for driving the process from detection through resolution rather than handing off responsibility between steps.

This approach is well-suited to scenarios such as guided operational workflows, closed-loop optimisation, or incident response, where a single coherent objective must be pursued continuously.

5.2.2 Multi-Agent Systems

In more complex environments, agentic behaviour may be distributed across multiple specialised agents, each responsible for a specific role or aspect of the overall objective.

For example:

> Consider a remote monitoring system for a patient with Chronic Obstructive Pulmonary Disease (COPD), with the shared goal of preventing exacerbations and maintaining respiratory stability at home.
>
> A physiological monitoring agent continuously evaluates oxygen saturation, respiratory rate, spirometry readings, and activity levels. It detects threshold breaches using rules and identifies early deterioration patterns using predictive models. A medication management agent tracks inhaler usage, adherence, and rescue medication frequency. It recognises trends associated with increased instability and flags elevated risk based on past exacerbation patterns. A context agent monitors environmental and behavioural factors, such as air quality, temperature, exertion, and sleep quality, and identifies external stressors that may contribute to respiratory decline. A care coordination agent maintains the overall objective of patient stability. It aggregates risk signals from other agents and determines when to initiate interventions, such as patient outreach, telehealth consultations, or clinician notifications.
>
> Together, these specialised agents collaborate around a shared goal—moving beyond simple alerting to coordinated, proactive management. Rather than reacting only after a severe threshold is crossed, the system synthesises physiological signals, medication

behaviour, and environmental context to detect emerging risk earlier. Each agent contributes its domain expertise, while the coordination layer ensures interventions are timely, appropriately sequenced, and aligned with clinical guardrails. The result is a resilient, adaptive monitoring framework that reduces hospitalisation risk, supports clinicians with prioritised insight rather than raw data, and helps the patient maintain long-term respiratory stability at home.

5.3 Strengths and Limitations of Agentic AI

Agentic AI represents a form of intelligence focused on goal-directed behaviour. Rather than responding to isolated inputs or generating single responses, agentic systems adopt objectives, construct plans, execute coordinated actions, monitor outcomes, and adapt their behaviour as conditions evolve. When implemented effectively, agentic AI enables systems to carry intent through to measurable outcomes rather than stopping at recommendation or analysis.

5.3.1 Strengths of Agentic AI

Agentic AI offers several distinctive strengths that expand the system's capabilities beyond reactive behaviour.

- **Goal-Driven Execution**: Agentic systems operate with explicit objectives that persist over time. They are responsible not just for producing insight, but for progressing toward defined outcomes, even as intermediate steps and environmental conditions change.
- **Planning and Decomposition of Complex Tasks**: Agentic AI can break high-level goals into structured plans, sub-tasks, and ordered actions. This enables the management of multi-step workflows that would otherwise require manual coordination or rigid predefined orchestration.
- **Adaptive Behaviour**: Agentic systems monitor the effects of their actions and revise their approach when results deviate from expectations. If a chosen strategy fails or only partially succeeds, the system can reassess and pursue alternative actions rather than terminating execution.
- **Autonomous Decision-Making Within Guardrails**: Agentic systems can take action without requiring human initiation at every step. Autonomy is bounded by defined policies, safety constraints, escalation thresholds, and approval requirements, enabling controlled independence.

Together, these strengths make agentic AI particularly effective in complex operational environments where achieving outcomes requires coordination, adaptation, and continuity.

5.3.2 Limitations and Trade-Offs of Agentic AI

Despite its power, agentic AI introduces significant architectural, operational, and governance considerations.

- **Increased System Complexity**: Agentic systems require state management, memory handling, planning logic, orchestration capabilities, and monitoring mechanisms. This increases architectural complexity relative to stateless or single-step systems.
- **Risk of Unbounded Autonomy**: Without carefully designed guardrails, agentic systems may pursue goals in unintended ways. Clear constraints, escalation paths, and policy enforcement mechanisms are essential to prevent undesirable behaviour.
- **Planning Uncertainty**: Plans generated at runtime are subject to uncertainty and will almost certainly be different from one run to the next because these systems inherently utilise Generative AI to generate plans, etc.

- **Debugging and Traceability Challenges**: Multi-step reasoning, dynamic planning, and adaptation can make behaviour harder to trace and debug compared to static workflows. Observability and logging mechanisms must be designed explicitly.
- **Resource and Cost Overhead**: Heavy dependence on Generative AI, iterative reasoning, and repeated tool invocation may increase computational cost and system load, particularly in multi-agent architectures.
- **Governance and Accountability Considerations**: As systems take increasingly autonomous actions, organisations must define clear responsibility boundaries, approval models, and audit frameworks to ensure compliance and accountability.

5.4 Agentic AI and Vantiq

Vantiq Agentic AI implementation builds on the Generative AI capabilities and enables developers to define an AI Agent as a service-based resource with a set of associated Skills (representing the Agent's capabilities). The system also provides a set of routing and planning algorithms, including ReWoo (Reasoning Without Observation), Plan-Execute, and Reflection, to enable developers to build and orchestrate a group of agents into an advanced multi-agent application. These capabilities can be integrated into the event flows and situation awareness, as well as support human-in-the-loop and advanced web and mobile clients using the Client Builder.

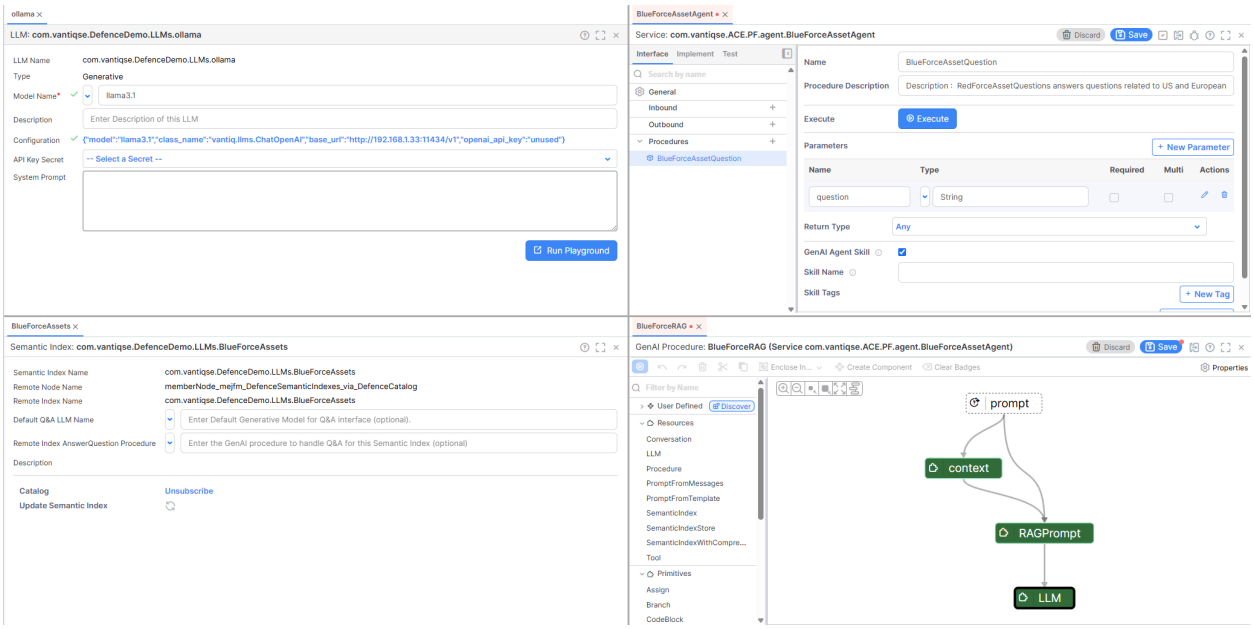


Figure 14: Agentic AI & Vantiq

5.4.1 Implementing a Single Agent Application in Vantiq

The pump example developed across Sections 2, 3, and 4 culminates in a grounded, plain-language alert delivered to the on-call engineer. What it does not do is take ownership of the outcome: even with a clear recommendation, a human must still interpret the situation, decide on a course of action, perform the corrective steps, monitor their effects, and revise the approach if the pump does not recover. A single-agent application places that responsibility on the agent itself. The agent adopts an explicit goal—to restore the pump to safe operation within

tolerance, or to escalate to a qualified technician—and remains responsible for that goal until it is achieved or formally handed off.

In Vantiq, an AI Agent is built as a Service with the GenAI Agent option enabled. Enabling this option causes the service to expose additional agentic capabilities. Rather than the inbound-event-and-procedure interface of an ordinary service, an AI Agent exposes a standard message-based interface: the agent receives a natural-language message and then decides how to fulfil the request by invoking one or more of its Skills. The strategy used to dispatch skills is configurable; here, we take the default. Every AI Agent also publishes an Agent Card that describes its capabilities and skills, but in this single-agent example, the card is not exercised — it is used primarily when integrating multiple agents. The worked example below is the PumpAgent2 service (package com.example.factory.agent).

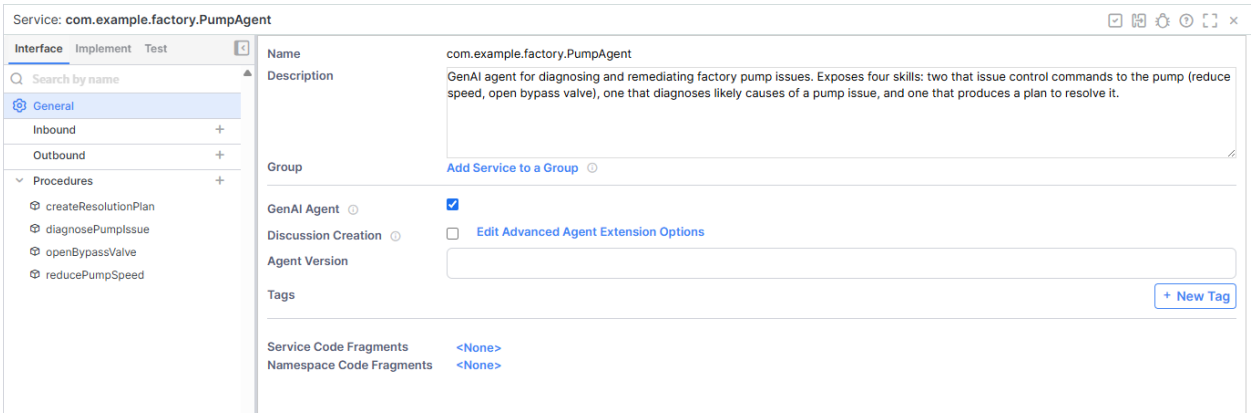


Figure 15: Agentic AI - PumpAgent2 example

For the pump agent, the six skills mirror the operational actions a technician would otherwise perform manually:

- `fetchRecentReadings(pumpId)` returns the most recent telemetry for the pump — temperature, pressure, vibration, and speed — from the real-time monitoring service, so the agent can re-read the situation between steps rather than relying on its conversation alone.
- `lookupRemediation(pumpId, symptoms)` grounds the agent's diagnosis in the plant's own knowledge: it forms a question from the observed symptoms and runs it through the generateRemediation GenAI Procedure, a Retrieval-Augmented Generation flow that retrieves the most relevant passages from the PumpKnowledgeIndex Semantic Index — the same grounding pattern introduced in Section 4.8.1 — and has the LLM synthesise citable guidance drawn from the plant's manuals, SOPs, and past incident reports.
- `openBypassValve(pumpId)` and `reducePumpSpeed(pumpId, speedPercent)` are the actuator skills; each delegates to the PumpControlService integration, which issues the command to the factory control system through a REST source and returns a structured acknowledgement the agent can reason about.
- `createWorkOrder(pumpId, summary, priority)` opens a maintenance work order through the CmmsService integration and returns the work-order acknowledgement.
- `escalateToTechnician(pumpId, proposedAction, context)` uses io.vantiq.ai.Agent.requestUserInput to bring a human into the loop when an action falls outside the agent's autonomy bounds, returning the technician's decision back into the plan.

The agent sits downstream of the perception and explanation layers built earlier. Unlike an ordinary service, an AI Agent is not driven by a typed inbound event; it is handed a message. A caller states the problem in natural language and lets the agent own the outcome, using io.vantiq.ai.A2A.messageSend to route the message to the named agent. A short fragment illustrates the invocation:

```
// Hand the incident to the agent as a message and let it own the outcome
var input = "Generate a plan to resolve the following issue: " + issue +
            " for pump " + pumpId
var msg   = io.vantiq.a2a.Message.forValue(input)
var res   =
io.vantiq.ai.A2A.messageSend("com.example.factory.agent.PumpAgent", msg)
```

Each skill is an ordinary procedure on the service, flagged as an agent skill so the framework can present it to the planner. Most skills are deliberately thin, delegating to the integration service that owns the external system. The reducePumpSpeed skill, for example, simply forwards the request to PumpControlService:

```
package com.example.factory.agent
STATELESS PROCEDURE PumpAgent.reducePumpSpeed(
        pumpId String REQUIRED DESCRIPTION "Identifier of the pump to slow
down",
        speedPercent Integer REQUIRED DESCRIPTION "New target speed as a
percentage of the pump's maximum (0-100)"
): Object
// Delegate to the control-system integration, which issues the command to
the
// factory control system and returns its acknowledgement. This integration
layer
// is also the natural place to enforce operating constraints before acting.
return
com.example.factory.integration.PumpControlService.reducePumpSpeed(pumpId,
speedPercent)
```

Once the message arrives, the agent constructs a plan rather than executing a fixed sequence. Planning and skill selection are handled by the agent-dispatch framework (io.vantiq.a2a.agentDispatch): the agent's backing LLM — openai/gpt-4o-mini in the sample — is presented with the message together with the descriptions of the available skills, and decides which skills to invoke and in what order — read the current state, consult the remediation, attempt the recommended action, re-read the state, then decide whether to continue or escalate. Vantiq supports several planning strategies for this:

- **ReWOO (Reasoning Without Observation)** suits a bounded, known skill set like this one, where the agent can lay out a single acyclic graph of tasks before executing it.
- **Plan-and-Execute** is available where the situation requires the agent to re-plan between steps.
- **Reflection** is used where the agent should critique its own intended action against its memory of recent outcomes before committing to it.

Throughout the incident, the agent maintains short-term memory in its conversation: the original problem, the readings observed, the actions already attempted, and the outcome of each. Because the GenAI Agent option is enabled together with its discussion extension, that conversation can be bound to a Collaboration instance, so the incident state survives restarts and is auditable after the fact, with every skill invocation recorded against the collaboration. When an action falls outside the agent's autonomy bounds — or an attempted remedy fails to bring the pump back inside tolerance within the agent's reassessment window — escalateToTechnician calls io.vantiq.ai.Agent.requestUserInput; the request propagates through the Conversation Widget to a qualified technician, whose response — approve, reject, or take over — is returned into the plan, and the agent continues, defers, or hands off accordingly.

This single-agent flow closes the loop introduced at the start of Section 5: perception is provided by the rules and predictive models of Sections 2 and 3, understanding and communication by the generative pattern of Section 4, and the agent now adds sustained goal ownership, adaptive execution, and bounded, human-supervised autonomy on top of them. The pump incident is no longer something an alert describes; it is something an agent resolves.

5.4.2 Implementing a Multi-Agent Application in Vantiq

Some objectives are too broad for a single agent. The COPD remote-monitoring scenario introduced in Section 5.2.2 illustrates this: respiratory stability at home depends on physiological signals, medication adherence, environmental and behavioural context, and clinical coordination, each of which constitutes a substantial body of knowledge in its own right. Concentrating it all in one agent produces something that is hard to test, hard to govern, and prone to losing focus. Distributing it across specialised agents—each owning a narrow role with its own skills and guardrails—yields a system that is easier to evolve and reason about, provided there is a clear pattern of collaboration.

In Vantiq, each specialised agent is built exactly as the single agent above: a Service with the GenAI Agent option enabled, its Skills implemented as procedures on the service. What makes the system multi-agent is the orchestration pattern between them. Every AI Agent automatically publishes an Agent Card describing its purpose and skills, and agents can be tagged so that a coordinator can find collaborators by role rather than by hard-coded name. A coordinating agent invokes another agent the same way any caller invokes an agent—by sending it a message with io.vantiq.ai.A2A.messageSend—so the called agent plans and executes against its own skills and conversation memory before returning its answer. The worked example below is four services in package com.example.copd.agent, with their external systems mocked by integration services in com.example.copd.integration, exactly as the pump agent's PumpControlService and CmmsService stand in for the factory's control system and CMMS.

The COPD application comprises three specialist agents and a coordinator around the shared goal of preventing exacerbations:

- **PhysiologicalAgent** owns the goal of detecting respiratory deterioration. Its skills evaluate the most recent SpO_2, respiratory rate, spirometry, and activity from the patient's PhysiologicalReading type (evaluateVitals), score deterioration risk by sending those features through a REMOTE source to the trained model of Section 3 (scoreDeteriorationRisk), and summarise the last 24 hours into a short, citable description (summarizeLast24h).
- **MedicationAgent** owns recognition of instability via medication behaviour. Its skills compute maintenance- and rescue-inhaler usage (computeInhalerUsage), report the adherence score (adherenceScore), and flag the rescue-medication trend against the patient's alert threshold (rescueMedicationTrend) from the MedicationRecord type.

- **ContextAgent** owns identifying external stressors. Its skills read air quality, indoor temperature, sleep quality, and exertion through a REMOTE source to the home gateway (queryEnvironment), and answer questions about local environmental advisories grounded against a Semantic Index (checkAdvisories).
- **CareCoordinationAgent** owns the overall goal to prevent exacerbation and maintain stability. It does not read sensors itself but plans across the other agents: discovering its collaborators by role (findSpecialists), asking each for an assessment (requestAssessment), and—weighing their responses against the patient's care plan—deciding when to scheduleTelehealth, notifyClinician, or escalateToCarer. The last uses io.vantiq.ai.Agent.requestUserInput to bring a family carer into the loop, exactly as the pump agent's escalateToTechnician brings in a human technician.

A guardrail policy is encoded as a skill the coordinator must consult before acting: checkGuardrail reads the patient's most recent SpO_2 and configured floor, and any plan that would withhold a clinician notification while SpO_2 is below that floor is automatically overridden to require one.

A short VAIL fragment illustrates how a caller hands the patient to the coordinator and lets it own the outcome—identical in shape to the single-agent invocation; only the agent and the message differ:

```
// Hand the patient to the coordinator as a message and let it own the
outcome

var input = "Assess patient " + patientId +

            " and decide whether to intervene"

var msg   = io.vantiq.a2a.Message.forValue(input)

var res   =
io.vantiq.ai.A2A.messageSend("com.example.copd.agent.CareCoordinationAgent",
msg)
```

Each of the coordinator's skills is an ordinary procedure on the service, flagged as an agent skill so the framework can present it to the planner. The requestAssessment skill confirms the named agent exists and carries the expected role, then dispatches an assessment request to it and feeds the answer back into the coordinator's plan:

```
package com.example.copd.agent

// One of the coordinator's skills: ask a specialist agent to assess the
patient

STATELESS PROCEDURE CareCoordinationAgent.requestAssessment(

        agentName String REQUIRED DESCRIPTION "Specialist agent to consult",

        patientId String REQUIRED DESCRIPTION "Patient to be assessed"):
Object

var result = null

// Confirm the named agent exists and carries the expected COPD role

var svc = SELECT ONE * FROM system.services

        WHERE isAgent == true AND name == agentName

var hasCopd = false

if (svc != null && svc.ars_properties != null && svc.ars_properties.tags !=
null) {
```

```
    for (t in svc.ars_properties.tags) {
        if (t == "copd") { hasCopd = true }
    }
}
if (svc == null || hasCopd == false) {
    result = { status: "unknown_agent", agent: agentName }
} else {
    // A2A dispatch — the called agent plans and executes using its own
    // skills, conversation memory, and guardrails
    var msg      = io.vantiq.a2a.Message.forValue(
            "Assess the current COPD deterioration risk for patient " +
patientId)
    var response = io.vantiq.ai.A2A.messageSend(agentName, msg)
    result = { agent: agentName, assessment: response, observedAt: now() }
}
return result
```

Most of the other skills are deliberately thin, delegating to the integration service that owns the external system—the coordinator's notifyClinician forwards to ClinicalService, just as the pump agent's reducePumpSpeed forwarded to PumpControlService. Grounding follows the same pattern introduced in Section 4.8.1: ContextAgent.checkAdvisories calls a Retrieval-Augmented Generation GenAI Flow (generateAdvisory) that retrieves the most relevant passages from the EnvAdvisoryIndex Semantic Index and has the LLM synthesise a citable answer —the multi-agent counterpart of the pump agent's lookupRemediation and its generateRemediation flow.

The coordinator's plan is constructed at runtime rather than executed as a fixed sequence. Vantiq supports several planning strategies for this:

- **Plan-and-Execute** suits this case, where the coordinator must re-plan between steps—asking each specialist for an assessment, observing the responses, and deciding whether to schedule telehealth, notify a clinician, or simply continue monitoring.
- **ReWOO** (Reasoning Without Observation) is available where the set of consultations is bounded and can be laid out as a single acyclic graph in advance.
- **Reflection** is useful where the coordinator should critique its own recommendation against the patient's care plan before committing to it.

Because the GenAI Agent option is enabled alongside its discussion extension, each agent's conversation can be bound to a Collaboration instance, ensuring the incident state survives restarts and is auditable after the fact. Every dispatch is recorded with the called agent's name, the inputs passed, and the outputs returned, so the end-to-end reasoning that led to a clinician notification is traceable from a single place—addressing the traceability concern raised in Section 5.3.2.

The result is a system that meets the COPD scenario's requirements without collapsing into a single, monolithic prompt. Each specialised agent remains testable in isolation against its own skills—PhysiologicalAgent.scoreDeteriorationRisk, for instance, returns a high-risk assessment for a deteriorating patient drawn entirely from that agent's data and model. The coordinator can be evolved—adding agents for new risk dimensions, swapping a planning algorithm, tightening

a guardrail—without reaching into the specialists. And the same composition pattern carries through the rest of this guide: rules and predictive models provide perception; generative reasoning produces grounded, role-appropriate explanations; and agents now provide sustained, coordinated, goal-directed execution across the entire patient experience.

6. Composing Intelligence: One Size Doesn't Fit All

Real-world applications rarely succeed by relying on a single form of intelligence. Different decisions require different trade-offs between speed, certainty, adaptability, explainability, and autonomy. As a result, effective intelligent systems are not built from one AI capability—they are composed of several, each applied where it provides the most value.

In Vantiq, intelligence should not be viewed as a monolithic feature, but as a coordinated set of complementary capabilities. The design questions are not which form of intelligence to use, but how to combine them to achieve reliable, scalable, and cost-effective outcomes.

6.1 Intelligence as Complementary Roles, Not Alternatives

Each intelligence type plays a distinct role within an integrated system architecture:

- **Knowledge-based rules** provide certainty and control. They encode explicit policies, constraints, and domain expertise. Rules define what must always hold true. They enforce boundaries, guarantee invariants, and trigger mandatory actions. Their strength lies in precision, performance, and auditability. Rules answer questions such as: *Is this condition allowed*? *Has a limit been exceeded*? *Must this action occur*?
- **Predictive AI** expands perception beyond what is explicitly encoded. By learning statistical relationships from historical data, predictive models identify patterns, correlations, and weak signals that would be impractical to define manually. They estimate likelihood, risk, and anomaly—answering questions such as: *Is this pattern statistically abnormal? How likely is failure?*
- **Generative AI** provides interpretation and contextual understanding. It synthesises structured data, unstructured knowledge, and the current state into explanations, summaries, and recommendations. It transforms signals into meaning and reduces cognitive load by answering: *What does this situation mean? Why does it matter?*
- **Agentic AI** introduces persistence and goal-directed execution. Rather than responding to isolated inputs, agentic systems adopt objectives and remain responsible for progressing toward outcomes. They plan, orchestrate actions, monitor results, and adapt under changing conditions.

When composed deliberately, these forms of intelligence create systems that are:

- Constrained by policy and domain knowledge (rules)
- Informed by statistical insight (predictive AI)
- Contextually understandable (generative AI)
- Capable of ensuring outcomes and goals are achieved (agentic AI)

Together, they form an integrated architecture of intelligent behaviour — moving from the encoding of domain knowledge and rules, through the identification of complex patterns and learned anomaly detection, to reasoning, and finally to planning and goal-oriented execution. No single form of intelligence can deliver all of these capabilities alone. It is their deliberate composition that enables scalable, resilient, and situationally aware real-time systems.

From Business Requirement to Architectural Choice

The composition of intelligence should be determined by the structure of the decision. Rather than asking which AI capability is most advanced, the design question is: What kind of decision is this, and what properties must it guarantee?

Different decision types require different architectural properties:

- **Well-understood, policy-bound decisions**
  When conditions are known in advance, and acceptable outcomes can be explicitly defined, knowledge-based rules provide precise and reliable enforcement. These rules encode domain expertise, regulatory requirements, safety constraints, and operational policies. They evaluate clearly defined conditions and trigger deterministic actions when those conditions are met. This approach is appropriate when the system must guarantee compliance, enforce invariants, and produce auditable, explainable outcomes. Predictive models may supplement this layer, but the core decision logic remains explicitly defined and verifiable.
- **Statistical risk assessment and complex pattern detection**
  When the goal is to identify non-obvious relationships, detect emerging behaviours, or recognise complex patterns in large or noisy data, predictive AI provides probabilistic assessment grounded in statistical modelling. Unlike knowledge-based rules, which evaluate explicitly defined conditions, predictive models learn patterns from historical data and generalise them to new situations.

  This capability is particularly important when signals are weak, multidimensional, or obscured by variability. Predictive AI can detect subtle correlations, classify complex states, estimate the likelihood of failure, and identify anomalies that would be impractical or impossible to define manually. The priority in these scenarios is early signal detection, pattern recognition under uncertainty, and quantified risk estimation rather than deterministic certainty.
- **Cognitive interpretation and human decision support**
  When decisions require synthesising heterogeneous information — structured data, unstructured content, historical context, and domain knowledge — generative AI provides semantic interpretation. Unlike predictive models, which estimate likelihood based on learned statistical structure, generative models reason across diverse inputs to construct coherent explanations, summaries, and recommendations.

  This capability is essential when the problem is not detecting a pattern but understanding its meaning. Generative AI integrates signals into narrative form, clarifies implications, compares alternative interpretations, and presents information in a way aligned to human roles or downstream systems. The priority in these scenarios is contextual coherence, semantic integration, and reduction of cognitive complexity.
- **Goal-driven, multi-step outcome execution**
  When an objective cannot be achieved through a fixed sequence of predefined steps, and intermediate results may alter the appropriate course of action, agentic AI provides dynamic goal-driven behaviour. Rather than executing a static workflow, the system adopts an explicit objective and remains responsible for achieving it.

  This capability is required when execution paths are conditional, branching, or evolving. The system must interpret changing conditions, select alternative strategies, coordinate services and collaborators, monitor intermediate outcomes, and revise its plan when assumptions no longer hold. The priority in these scenarios is sustained goal ownership, adaptive planning, and managing uncertainty.

6.1.1 A Common Composition Pattern

An intelligent Vantiq application could operate at all levels of intelligence using this pattern:

1. **Perception**

   a. Knowledge-based rules detect known conditions immediately.
   b. Predictive models identify complex anomalies, risks, or likely future states.
   c. Outputs are emitted as high-level events enriched with context and confidence.
2. **Understanding and Reasoning**
   a. Generative AI synthesises current events, historical context, and domain knowledge.
   b. Explanations, summaries, and candidate actions are generated for either humans or automated systems.
3. **Execution and Adaptation**
   a. Agentic AI takes responsibility for achieving a goal.
   b. It selects actions, orchestrates services and people, monitors outcomes, and adjusts behaviour as conditions change.
   c. Rules and predictive signals continue to feed the agent as the situation evolves.

This approach ensures that fast, deterministic logic is used where certainty is required, while more flexible and adaptive intelligence is applied only when needed—optimising both performance and cost.

6.1.2 Examples: Composing Intelligence in a Safety and Security Video Application

A safety and security application built on video cameras and computer vision does not rely on a single form of intelligence. Detecting and responding to situations such as crowd formation, fall-down incidents, or unattended luggage requires layered perception, interpretation, enforcement, and coordinated response. Each type of intelligence contributes a different capability, and operational effectiveness emerges from their deliberate composition.

The foundation of the system is computer vision. Video streams are transformed into structured semantic events such as person detected, object detected, posture classified, and people count. These outputs represent interpreted visual observations enriched with confidence scores. A person lying on the ground is not yet a medical emergency. A bag without a nearby individual is not automatically a threat. A group of people in proximity is not necessarily unsafe. Computer vision provides structured perception — awareness of what is present and occurring in the scene — but it does not determine intent or risk.

Knowledge-based rules can be used to define thresholds and determine when crowd density exceeds acceptable limits. Time-based conditions specify when a motionless posture becomes a welfare concern. Spatial and temporal rules define when luggage transitions from temporarily placed to unattended. In safety-critical environments, this layer guarantees that known constraints are enforced consistently and immediately.

Predictive AI extends beyond static thresholds through statistical modelling. For example, a clustering model can analyse spatial coordinates and movement trajectories of detected individuals to identify emergent group formations. Rather than counting people alone, the model identifies statistically significant clusters based on density, proximity, and movement convergence. This allows the system to distinguish between dispersed foot traffic and tightly grouped formations that may indicate a risk of congestion or abnormal behaviour. By learning what “normal” spatial distributions and movement patterns look like, the system can detect deviations that signal emerging risk before any explicit threshold is crossed. This provides probabilistic risk awareness grounded in statistical structure rather than fixed limits.

Generative AI adds contextual interpretation. Instead of presenting raw detections or statistical outputs, the system synthesises camera observations, clustering results, timing information, and

environmental context into coherent explanations. An operator may receive a summary stating that pedestrian movement near an exit is forming a high-density cluster with increasing inward flow. A supervisor may receive an explanation that a person has remained motionless beyond expected norms and is not receiving assistance. This layer transforms signals into situational understanding and reduces cognitive load.

Agentic AI introduces goal-driven behaviour. When congestion risk is identified, the system adopts the objective of restoring safe flow. It can notify staff, monitor density trends, and escalate if clustering intensifies. When a potential fall incident is detected, the system aims to ensure the individual's safety while minimising unnecessary disruption. Rather than executing a fixed escalation workflow, it evaluates multiple strategies based on context. It may analyse additional camera angles, assess nearby pedestrian behaviour to determine whether assistance is already being provided, correlate badge or identity data to understand whether the individual is staff or visitor, and weigh historical false-positive rates for similar detections in that location. The system maintains continuity of intent and adapts actions as the environment evolves.

In a safety and security video application, these layers function together. Computer vision provides perception. Knowledge-based rules enforce safety boundaries. Predictive AI models identify emerging spatial risk patterns. Generative AI explains the situation. Agentic AI ensures that identified risks are resolved. The strength of the system lies not in any single capability, but in the deliberate integration of perception, statistical insight, contextual explanation, and outcome-oriented execution.

6.1.3 Examples: Composing Intelligence in a Hospital Discharge Application

A hospital discharge process appears procedural on the surface, but in practice, it is dynamic, multi-disciplinary, and highly dependent on coordination across clinical, administrative, and logistical domains. Safely discharging a patient requires more than a checklist. It requires perception of state, enforcement of policy, detection of emerging risk, contextual understanding, and coordinated execution. No single form of intelligence can manage this end-to-end reliably.

The foundation of the system is structured clinical and operational data. Events such as physician discharge orders, medication reconciliation status, lab result availability, bed occupancy levels, transport requests, and insurance approvals represent the operational perception layer. These signals establish what has occurred and what state the patient and hospital currently occupy. However, raw status updates do not determine readiness or safety. They provide awareness, not resolution.

Knowledge-based rules enforce clinical and administrative requirements. A patient cannot be discharged without signed physician orders. Medication reconciliation must be completed. Required discharge documentation must be generated. Follow-up appointments must be scheduled for specific diagnoses. Insurance authorisation must be confirmed before certain services are initiated. These rules ensure regulatory compliance, patient safety standards, and billing correctness. They provide clear guardrails and prevent premature discharge when mandatory conditions are unmet.

Predictive AI adds a deeper statistical level of intelligence to the discharge process. For example, a clustering or classification model can analyse historical discharge timelines across departments to identify patterns associated with delays or readmissions. Patients with similar diagnosis codes, comorbidity clusters, age groups, or discharge destinations may historically exhibit elevated readmission risk. By grouping patients into statistically similar clusters, the system can identify when a current discharge resembles prior high-risk cases. It may also detect bottlenecks by clustering operational events, such as repeated delays in pharmacy fulfilment

during peak hours or transport wait times that trend above baseline. This allows the hospital to anticipate delays or elevated risk rather than discovering them after discharge occurs.

Generative AI introduces contextual interpretation and communication. Rather than presenting staff with fragmented status indicators, the system can synthesise patient state, pending tasks, risk scores, and historical patterns into a concise readiness summary. A care coordinator may receive a narrative explanation outlining the remaining prerequisites, the estimated time to discharge, and the identified risk factors. A physician may receive a summary of follow-up considerations tailored to the patient's profile. A discharge nurse may receive a structured explanation of medication changes and required patient education points. This reduces cognitive load and ensures that complex, cross-system information is presented coherently and role-appropriately.

Agentic AI introduces goal-driven coordination. Once a discharge order is issued, the system focuses on safely completing the discharge. It can monitor completion of required steps, notify the pharmacy when prescriptions are pending, schedule transport once clinical clearance is confirmed, escalate delays, and verify that follow-up appointments are scheduled. If a delay occurs — such as medication preparation exceeding expected duration — the system can adapt by notifying supervisors or reallocating resources. It maintains state throughout the discharge lifecycle and ensures the objective is not considered complete until all required conditions and safety checks are met.

In a hospital discharge application, these layers work together continuously. Structured operational data provides awareness. Knowledge-based rules enforce compliance and safety constraints. Predictive models provide deep analysis of risk and bottlenecks. Generative reasoning synthesises complex context into an actionable understanding. Agentic coordination ensures that discharge is executed safely and efficiently from initiation through completion.

7. Design Guardrails: Governing Intelligent Applications

The preceding chapters showed how each form of intelligence expands what an application can do. With that capability comes the responsibility to keep behaviour safe, predictable, and accountable — especially as a system moves from deterministic rules toward predictive, generative, and agentic capabilities that infer, generate, and act with increasing autonomy. The guardrails below are not specific to any one intelligence type; they are cross-cutting design concerns that should be considered for every intelligent application. As a rule of thumb, the further an application moves along the spectrum from rules to agency, the more rigorously these guardrails need to be applied.

7.1 Grounding - RAG

Predictive and generative outputs should be anchored to verified, current data rather than allowed to operate in a vacuum. Techniques such as Retrieval-Augmented Generation (RAG) constrain a model to a curated body of knowledge and supply the context on which its output is based, reducing hallucination and keeping responses tied to the facts of the situation. A grounded output can cite the data it relied on; an ungrounded one cannot.

7.1.1 GenAI Flows & RAG

Implement grounding with the GenAI Builder's `RAG` component, which performs Retrieval-Augmented Generation against a semantic index. Set the task's `semanticIndex` property to the index to query, and tune retrieval with `minSimilarity` and `metadataFilter` to exclude weakly related or out-of-scope documents and the `limit` property to cap how many context documents are passed to the model (default 4). Source content is loaded into the index through the ingestion flow named in its `Ingest GenAI Procedure`, so you control how documents are pre-processed and chunked. Because the `RAG` task returns the question, the retrieved context, and the generated answer together, the application can display — and audit — the exact context each answer was grounded in.

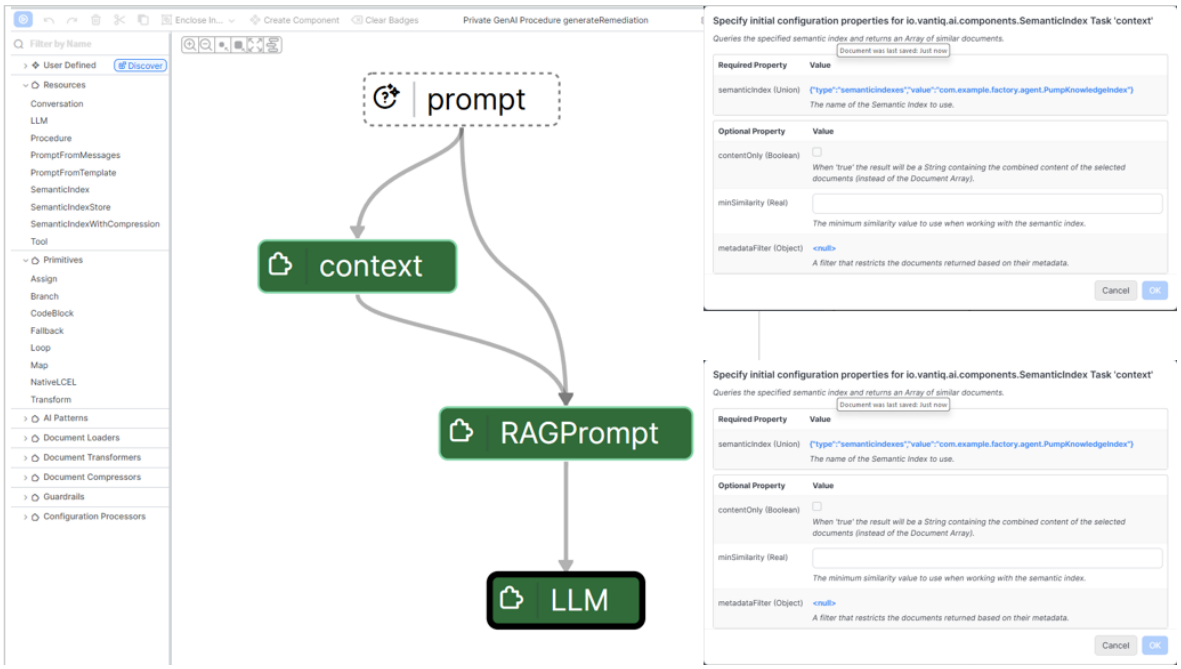


Figure 16: RAG in GenAI Flow

7.2 Content and conversation guard rails.

Beyond grounding and permissions, a generative system needs controls on what users may ask and on what the model is allowed to say: blocking unsafe, off-topic, or manipulative inputs, keeping responses on-policy and on-topic, and catching unsupported or sensitive content before it reaches a user. These input and output checks are a distinct layer of defence from the data a model is grounded in or the tools it is permitted to call.

7.2.1 GenAI Flows and Guardrails

Apply these checks with the GenAI Builder's `NeMoGuardrails` component, a protective layer that screens both the prompt going into an LLM and the response coming back out. It is configured with a generative model (`generativeLlm`) and an embedding model (`embeddingsLlm`), a `colang` document that defines the conversational flows and the canonical user and bot messages (CoLang v1), and a `yamlConfig` document holding the settings NeMo expects in its `config.yml`. The `inputRails` and `outputRails` properties list the applied to inputs and outputs to the LLM — including NeMo's built-in `self check input` and `self check output` rails — while `customModels` can route particular rails to different models and `actionsServer` points to a custom actions server. NeMo Guardrails itself is NVIDIA's open-source toolkit for wrapping an LLM in programmable "rails": input rails screen and sanitise user messages (jailbreak attempts, abuse, off-topic or out-of-scope requests, sensitive data); dialog rails keep the conversation on an approved path; retrieval rails act on the context returned by RAG; output rails fact-check, moderate, and strip unsupported or sensitive responses; and execution rails constrain the actions the model is allowed to trigger. Together, they let you express, as policy, exactly what the assistant will and will not do — reinforcing the

grounding, tool-permission, and human-approval controls described alongside it.

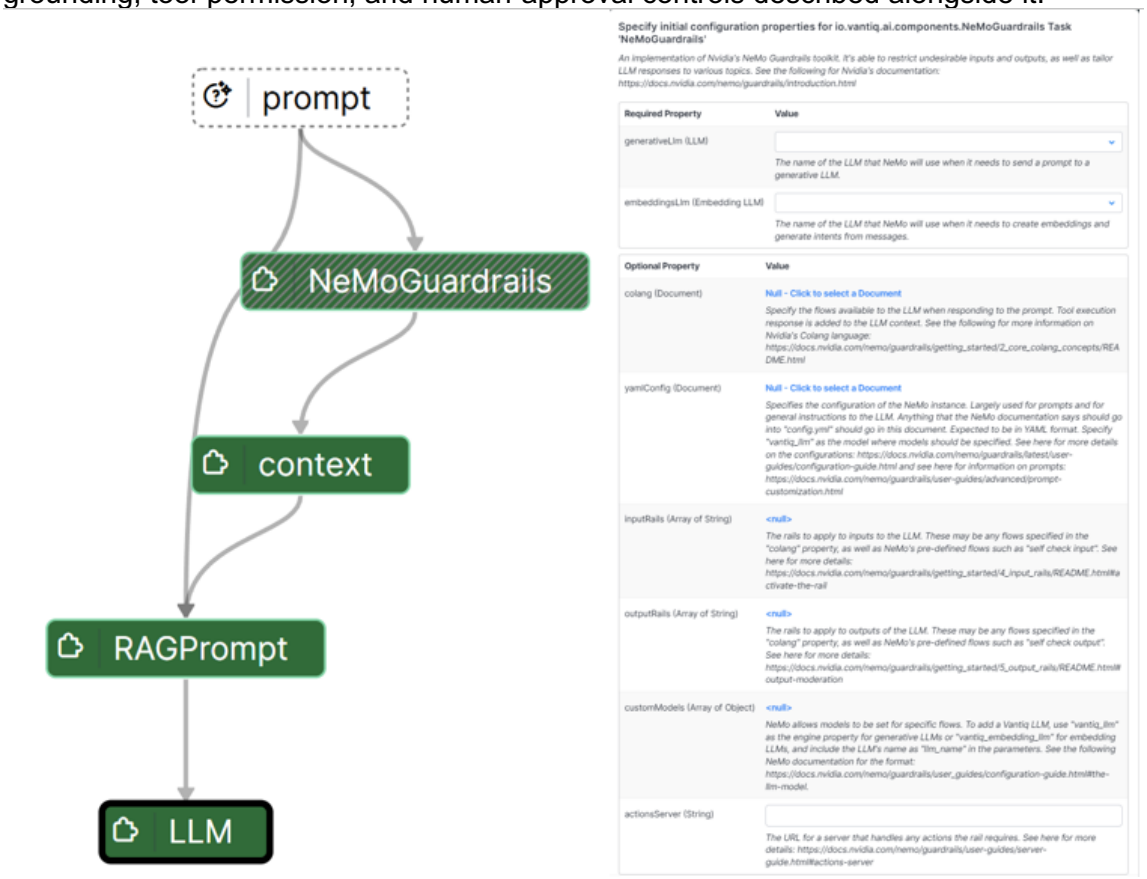


Figure 17: NeMo Guardrails

7.3 Auditability

Knowledge-based rules are inherently transparent — when a rule fires, the reason is explicit. That same traceability must be preserved as less deterministic intelligence is introduced. Inputs, model outputs, confidence scores, the context retrieved, and the actions ultimately taken should all be logged, so that any decision can be reconstructed and explained after the fact. Auditability is the foundation of both regulatory compliance and operational trust.

7.3.1 Auditing in GenAI Flows

In a GenAI flow, enable Task Auditing on the critical tasks. When enabled, Vantiq writes an Audit record each time the task executes, capturing the task's inputs and outputs together with a timestamp and the security context in use. Auditing is disabled by default and is turned on task by task — enable it on the decision-bearing tasks (the LLM call, the tool invocation, the action that changes state) rather than everywhere, so the trail stays meaningful and the volume manageable. Audit records are reviewed under the Administer menu in the IDE. A Visual Event Handler does not generate these audit records automatically; there, capture the equivalent trail programmatically — for example, a `Procedure` or `VAIL` task that records the inputs, outputs, and decisions taken at the points you care about — giving the same permanent, reconstructable record of what the system was told, what it produced, and the identity it ran as.

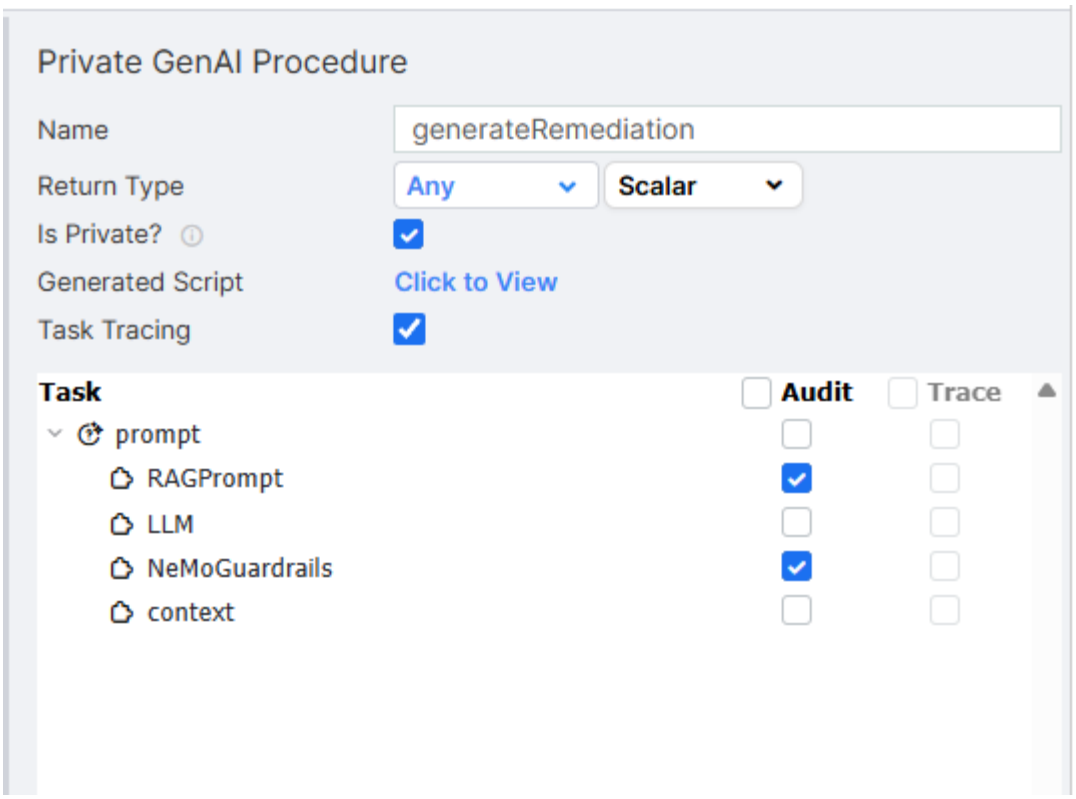


Figure 18: GenAI Flows and Audit Records

```
RULESET AuditPublishesToMySource
// Triggering condition is on any message arriving from a source named MySource
WHEN EVENT OCCURS ON "/sources/MySource" AS myVariable
// Construct an audit object with entity, op, ars_properties and message defined
var audit = {"entity": {resource: "sources", resourceId: "MySource"},
             "op": "publish",
             "ars_properties": { "importantProp": myVariable.importantProp },
             "message": "Message received from source MySource with value: " + myVariable.toString()}
// Manually insert the audit record
INSERT audits(audit)
```

Figure 19: VAIL Generated Audit Records

7.4 Human approval

Not every decision should be fully automated. High-impact, irreversible, or safety-critical actions should require explicit human sign-off, and low-confidence or ambiguous decisions should be escalated to a person rather than executed autonomously. Keeping a human in the loop at the right decision points is one of the most effective controls available, particularly for agentic systems that act on their own conclusions.

7.4.1 Request User Input

For agentic flows, bring a person into the loop with the built-in `io.vantiq.ai.Agent.requestUserInput` procedure. Calling it pauses the agent and routes a question back to the original requester, returning the user's answer as the result; in a multi-agent system, the framework automatically escalates the request up the call chain until it

reaches the user. Use it to require explicit sign-off before a high-impact action — for example, “Approve dispatching the field engineer?” The Discussion Widget will automatically surface these questions to the user in a Client Builder UI. It is also possible to subscribe to user requests for services when using non-Client Builder UIs.

7.5 Enforcing tool permissions in Vantiq

Two complementary mechanisms let you authorise individual tool calls in code, rather than trusting the model's judgement alone.

At the Tool task — the authorizer property. The GenAI Tool task (the component that lets an LLM call a function) exposes an authorizer property that names a procedure invoked to authorise the call before the underlying function executes. This provides a low-level, code-controlled checkpoint: the authorizer procedure inspects the requested tool and the arguments the model has proposed and decides whether the call may proceed. Because the decision is made in a procedure rather than in the prompt, it cannot be reasoned around or "prompt-injected" away by the model — it is enforced in code, can consult the current user, roles, namespace, or an external policy service, and can reject or escalate anything that falls outside policy. The model can request an action, but the action only runs once your own logic has approved it.

7.5.1 Inside a GenAI Agent's skills.

A Vantiq GenAI Agent exposes its capabilities as skills, and each skill is implemented as a procedure — a public procedure that the developer marks as a skill, which the agent's dispatcher selects by matching the procedure's description and parameters to the user's request. Because a skill is ordinary VAIL, authorisation can be enforced inside the skill procedure itself, before any privileged work is done, in one of two ways:

7.5.2 Automatic checks – In AI Agent Skills.

The Skills validate the request against their own rules first — the caller's identity, roles, the arguments' ranges and allowlists, and any business constraints — and return an error or refusal if the action is not permitted. The agent never reaches the privileged operation unless the check passes.

7.5.3 Human approval via requestUserInput.

For actions that should not be fully automated, the Skill calls `io.vantiq.ai.Agent.requestUserInput(...)` at the decision point. This pauses the Skill and routes a question back to the user — escalating up the call chain in a multi-agent system — blocks until they respond, and proceeds only if the user approves. It turns a high-impact skill, such as issuing a refund, dispatching a field engineer, or changing a clinical order, into an explicitly human-authorised step without leaving the agent's own logic.

Used together, the authorizer on a Tool task guards model-initiated function calls at the platform level, while skill-level checks and `requestUserInput` let each agent's capability enforce its own least-privilege and approval rules — so the model may request an action, but only vetted, and where appropriate, human-approved, code actually performs it.

7.6 Fallback behaviour.

Intelligent components can be unavailable, time out, or return uncertain or low-confidence results. The application should define safe default behaviour for these cases — degrading gracefully to deterministic rules, a conservative default, or human handling — rather than failing open or acting on an unreliable result. Well-designed fallbacks ensure the system remains safe even when its most adaptive components are not at their best.

7.6.1 Fallback in GenAI Flows

Two complementary mechanisms apply. In a GenAI flow, use the `Fallback` component: it runs a primary sub-flow and, if that fails, tries one or more alternative sub-flows in order until one succeeds, with the number of alternatives set by the `fallbackCount` property — for example, failing over from a primary model to an alternate, or from a generated answer to a safe canned response.

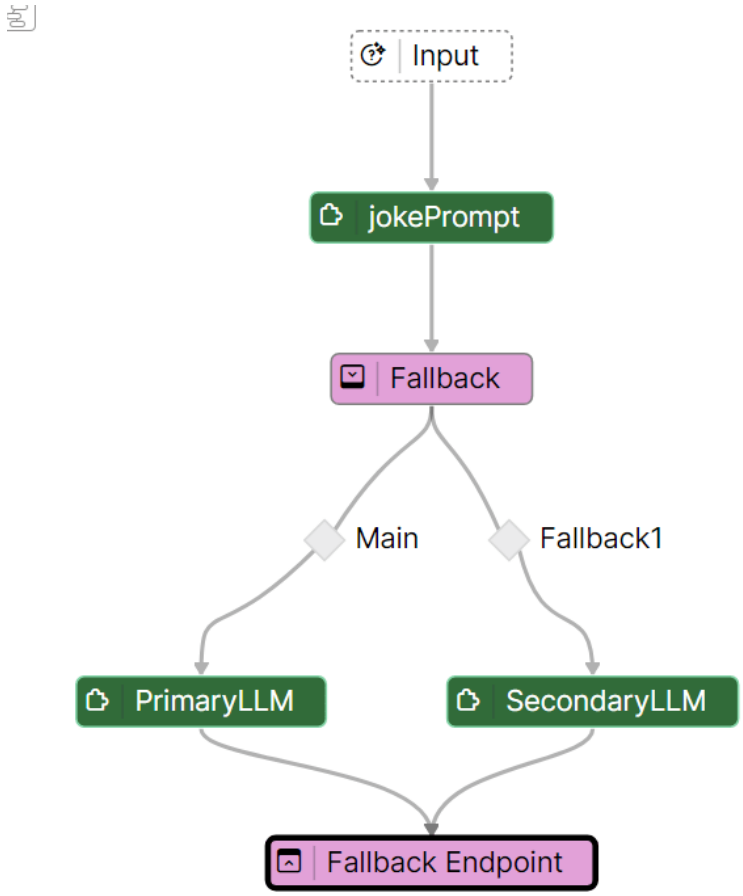


Figure 20: GenAI Flow - Fallback

7.6.2 Reliable delivery and retries in VEH

In a Visual Event Handler, reliable processing is provided by event redelivery: when a VEH source is configured with reliability guarantees, Vantiq checkpoints progress (partitioned by event UUID) and, after a failure, redelivers the event and resumes at the last incomplete task under at-least-once semantics. Because a task may therefore run more than once, design tasks

to be idempotent — stateful patterns such as `AccumulateState` and `Dwell` already ignore repeated state changes, while `Procedure` and `VAIL` tasks should be written to tolerate re-execution. Note that application errors are acknowledged immediately and are not redelivered, whereas transient errors allow the event to be retried.

7.7 Conclusion

These guardrails — grounding, content and conversation controls, auditability, human approval, enforced tool permissions, and fallback behaviour — are best understood not as a checklist to be satisfied once, but as a layered system of defence. Each closes off a different way an intelligent application can go wrong: grounding keeps outputs anchored to verified fact rather than invention; input and output rails keep a conversation safe and on-policy; auditability makes every decision reconstructable after the event; human approval holds back the actions that should never be fully automated; enforced tool permissions ensure a model can request privileged work but never perform it unilaterally; and fallback behaviour keeps the system safe when its most adaptive components are unavailable, slow, or unsure. No single control is sufficient on its own; their value lies in working together.

How much of this scaffolding a given application needs depends on where it sits on the spectrum from rules to agency. A deterministic, rules-based flow may need little more than sound auditability; a generative assistant adds grounding and content rails; and an agent that acts autonomously on its own conclusions warrants the full set, with human approval and code-enforced permissions at its most consequential decision points. The guiding principle is proportionality — the more freedom a component has to infer, generate, or act, the more rigorously these controls should be applied.

Vantiq provides natural enforcement points for each of these guardrails: rules and services impose policy and least-privilege boundaries; the event-driven architecture makes inputs and actions straightforward to log and monitor; GenAI flow components such as the RAG, `NeMoGuardrails`, and `Fallback` tasks build grounding, content rails, and graceful degradation directly into the model pipeline; and human tasks, together with `requestUserInput`, insert approval steps where they matter most. Critically, these controls are enforced in the design and in code rather than left to the model's own judgement, so they cannot be reasoned around or prompt-injected away. Treating them as part of the design from the outset — rather than as an afterthought bolted on once something has gone wrong — is what allows an intelligent application to be not only more capable, but genuinely trustworthy: safe, predictable, and accountable to the people who rely on it.