Agent & System Concepts

> *We are all wanderers in the land of abstractions. We build machines that talk, think, plan and act; then we must watch over them, like shepherds watching a flock of particularly flighty goats. This book is your field guide.*

The research that inspired this glossary made a simple but powerful claim: **production agents are not chatbots**. When you move beyond toy demos and into applications that do real work — sending e-mails, updating CRMs, generating documents, orchestrating pipelines — you encounter a messy reality. An LLM-powered agent is part stochastic process and part distributed system; it uses tools, maintains memory, acts under permissions, causes side effects, and drifts over time[1]. Logs and metrics, the traditional tools of observability, can tell you that something happened; they cannot tell you whether it mattered[2]. The result is a new discipline: **LLM observability and agent runtime control**.

This glossary collects and explains the terms used in the research document and in the Neatlogs data model. Each concept is unpacked with the aim of making it accessible to engineers who are comfortable calling the OpenAI API but may be less familiar with observability, distributed systems, or governance. Along the way we will look at concrete examples, offer a few strong opinions, and occasionally wax poetic. Think of it as a bridge between the world of agent demos and the world of production uptime.

This chapter introduces the core abstractions at the heart of modern AI agents. If you have written a Python script that calls a large language model (LLM) API, you already know the basics of prompt in and completion out. Agents build on top of that primitive by giving the model access to tools, memory and autonomy. They also bring with them the messy realities of distributed systems — randomness, side effects, permissions and drift. We'll unpack each of these ideas and show how they relate to the problems of monitoring and control.

1. File citation: 700004768920402, L205-L220.
2. File citation: 494139611690319, L410-L435.

Agent

An **agent** is a software entity that uses an LLM to decide what to do next. It typically accepts a goal or task description, interacts with one or more tools (APIs, databases, file systems, search engines), maintains state between steps, and produces outputs with side effects (like sending an e-mail or updating a record). Agents differ from simple chatbots in that they have **autonomy**: they decide when to call a tool, how to synthesize results, and when to finish. A well-designed agent is constrained by a set of permissions (see below) and is evaluated based on the usefulness of its outcomes rather than the eloquence of its replies.

Multi-Agent System

A **multi-agent system** orchestrates several agents with distinct roles. For example, a *research* agent might gather information, a *writer* agent might draft a report, and a *critic* agent might review the draft. Each agent may run concurrently or sequentially, and they communicate via messages or shared state. Multi-agent architectures introduce complexity: you now have to reason about parent and child agents, hand-offs, and orchestration logic. They can unlock parallelism and specialization but often make tracing and error handling more difficult. When in doubt, start with a single agent and add more only when there is a clear separation of responsibilities.

Stochastic Distributed System

When the research says that production agents are *stochastic distributed systems*, it means two things. *Stochastic* refers to randomness: the LLM itself is a sampling process, and the environment (network latency, tool responses, user inputs) adds further randomness. A *distributed system* is any system made up of multiple components that communicate over a network. In such systems events occur concurrently and partial failures are common. Mathematically, a stochastic process is "a set of random variables that depicts how a system changes over time; it explains how a system's state varies in unforeseen or random ways"[3]. When you ask an agent to perform a task, its sequence of steps (the "trace") is such a stochastic process: every run is slightly different.

3. File citation: 135110979075677, L98-L105.

Tools

In agent land, a **tool** is any function external to the model that the agent can call. Examples include web search, database queries, e-mail APIs, file I/O, spreadsheets, code execution, and custom business functions. Tools are typed (they take structured input and return structured output), and agents must respect their contracts. A typed tool contract prevents hallucinated calls and ensures the agent passes valid parameters. From an implementation perspective, each tool call becomes a span in a trace (see the Observability chapter) and may be monitored for latency, cost and errors.

Memory

Agents maintain **memory** to carry context across turns. Memory can be as simple as a list of previous messages (chat history) or as elaborate as a vector store containing embedded documents. Memory allows an agent to refer back to earlier information, maintain long-term goals, and avoid forgetting instructions. It also introduces the possibility of **context drift**: as the memory grows, earlier context may be pushed out or become irrelevant, causing the agent to behave unpredictably.

Permissions

Permissions specify what an agent is allowed to do. They can be coarse-grained (e.g. "can send e-mails") or fine-grained (e.g. "can update CRM records only in the `prospects` table"). In a governance context, each agent has an identity and an authorization scope. Permissions are enforced at runtime via a gateway that intercepts tool calls and checks them against policy. Without proper permissions, agents become dangerous: they may send e-mails to the wrong people, delete data, or exfiltrate sensitive information.

Side Effects

In programming, a **side effect** occurs when a function interacts with the outside world, such as writing to a file, making a network request, or mutating global state[4]. Pure functions compute a return value from their inputs and nothing else, while functions with side effects cause changes beyond their return values. Agents, by design, perform side effects: they send messages, create files, update databases.

Observability tooling must therefore capture not just the LLM's responses but the side effects that follow. A trace that ends with "success" but created no artifact is a phantom success.

Drift

Drift refers to changes over time that degrade performance. In the context of agents, drift can occur in several ways:

- **Data drift:** the distribution of input data (tool responses, user queries) shifts, making earlier assumptions invalid.
- **Model drift:** the underlying language model or tool models are updated, changing behavior without explicit coordination.
- **Context drift:** the agent's memory accumulates irrelevant information, causing it to lose focus.
- **Policy drift:** the permissions and governance rules evolve, but the agent still uses an outdated policy.

Drift is insidious because it is gradual; without monitoring, you may not notice until the agent's outputs are no longer useful. Detecting drift requires tracking outputs over time, replaying past runs with current models, and comparing results.

4. File citation: 9216522356914, L70-L83.

Observability Basics

Observability is the practice of instrumenting a system so you can understand what it is doing and why. In the agentic world this means capturing not just log lines but rich traces of agent behaviour, including tool calls, LLM prompts and completions, memory access, and side effects. Classical observability focuses on three signals: **metrics**, **logs** and **traces**[2]. Modern agent observability extends those concepts to include **spans**, **tokens**, **costs**, **errors**, and **providers**. This chapter introduces these terms and explains how they fit together.

Metrics

Metrics are numerical measurements of a system at a point in time. They include counts (number of requests), durations (latency), ratios (error rate), and gauges (CPU usage). Metrics are compact and easy to aggregate, which makes them ideal for dashboards and alert thresholds[5]. In agent observability, metrics might include the total number of tokens generated per run, the average latency of tool calls, or the cost of running a workflow. Metrics answer questions like "How many runs per minute are we handling?" or "Is the cost per successful task increasing?"

Logs

Logs are time-stamped records of discrete events. Each log entry captures what happened and when, often including a message, context fields, and a severity level. Logs are verbose and detailed; they are invaluable for debugging but can be overwhelming at scale[6]. In an agent system you might log every LLM prompt and completion, every tool request and response, and every unexpected error. Logs answer questions like "What did the agent ask the model?" or "Why did this tool call fail?"

Traces

Traces tell the story of a single request as it flows through a distributed system. A trace is composed of **spans**, each representing a unit of work such as an HTTP request, a database call, or an LLM invocation[1]. A trace has a unique ID shared by all of its spans, and one span is designated the **root**. Traces allow you to see how long each step took, in what order they occurred, and how they relate to one another. In the context of agents, a trace encompasses the entire run of an agent: from receiving

5. File citation: 494139611690319, L397-L409.

6. File citation: 494139611690319, L410-L417.

the task to producing the final output and all intermediate tool interactions. Traces answer questions like “What sequence of actions led to this result?”

Spans

A **span** is the smallest unit of a trace. According to the OpenTelemetry specification, a span includes an operation name, a trace ID, a span ID, a parent span ID (if it is not the root), a start timestamp and a duration[7]. Spans can also carry attributes (key-value pairs) and events. In agent observability we categorize spans by their role:

- **Agent Action:** a span representing a decision made by the agent (e.g. choosing a tool and arguments).
- **LLM Call:** a span for a single prompt/completion interaction with an LLM provider. Attributes include model name, token counts, and latency.
- **Tool Call:** a span for an external API call. Attributes include tool name, input payload, response, status code, latency and errors.
- **Retrieval:** a span for fetching knowledge from a vector store or database. Attributes include the number of records returned and the indices consulted.
- **Chain:** a span representing a call to another chain or sub-workflow. This is common when using frameworks like LangChain or PromptFlow.

The parent/child relationships between spans form a tree (often visualized as a waterfall). Understanding that tree is essential for diagnosing where latency and errors occur and for reconstructing the agent’s decision process.

Tokens and Costs

LLMs are billed by **tokens**, which roughly correspond to chunks of text. Each call to an LLM consumes input tokens (prompt) and output tokens (completion). Observability systems count tokens at the span and trace level so you can measure **cost**, which is typically proportional to token usage. Costs may also include tool invocation charges. Tracking cost per successful output is critical for assessing ROI.

Latency

Latency measures how long an operation takes. In the context of spans, latency is the duration from the start to the end of the span. Observability systems compute latency percentiles (p50, p90, p95) to summarize performance. High latency may indicate slow LLM responses, slow tool endpoints, or network issues.

---

7. File citation: 700004768920402, L215-L236.

Errors

Errors capture failures during a span. An error may be an exception thrown by your code, a non-200 HTTP response from a tool, a timeout, or a malformed LLM response. Each error should include a type, message and stack trace where possible. Error metrics (count, rate) help you detect regressions and triage issues.

Provider and Model

Agents often use multiple **providers** (OpenAI, Anthropic, local LLMs) and **models** (e.g. `gpt-4o`, `claude-sonnet`, `embedding-model-v1`). Observability tooling captures which provider and model were used for each LLM span so you can compare performance and cost across vendors. Provider metadata also helps with governance: some models may not be approved for certain data types.

Differences Between Logs and Traces

It is common to confuse logs and traces. Logs describe isolated events; traces describe the path of a request across components. Logs are excellent for capturing errors and context, while traces show how those errors fit into the broader workflow. Honeycomb's article on observability puts it succinctly: logs are *time-stamped records generated by software applications, services or devices*[8], whereas traces *follow the path of a request as it traverses various components of a distributed system*[1]. A well-instrumented agent uses both: logs for detailed context and traces for the big picture.

8. File citation: 700004768920402, L166-L171.

Detection, Alerting & Analytics

Observing an agent is not enough; you also need to detect when something has gone wrong and alert the right people. Neatlogs introduces several abstractions to achieve this: **detections**, **alerts**, and **analytics**. This chapter explains these concepts and how they fit into an operational workflow.

Detections

A **detection** is a rule or classifier that inspects a trace or span and determines whether it exhibits some pattern of interest. Detections can be:

- **Conditional rules:** simple expressions over metrics or attributes (e.g. `cost > 0.05`, `error ≠ null`, `latency > 2.0s`).
- **Regex rules:** pattern matching on text (e.g. does the LLM output contain the word "hallucination"?).
- **Classification rules:** machine-learned models or heuristics that assign categories to outputs (e.g. sentiment analysis, toxicity detection, relevance detection).

Each detection produces a **result** for a trace or span. In Neatlogs, a detection result has a **state** — **positive**, **neutral** or **negative**. A positive detection means the pattern was found and is considered good (e.g. "the output contains the expected keyword"); negative means a bad pattern was found (e.g. "the output is hallucinated or irrelevant"); neutral means neither. Detections can run on the entire trace or on specific span types (only LLM calls, only tool calls, etc.).

Detections also have **thresholds** or **weights**. For example, a classifier might output a score between 0 and 1 indicating how hallucinatory the output is; you set a threshold (say 0.7) above which the detection is marked negative. Regex rules may count occurrences and trigger only after a certain number of matches.

When building your own detections, think about what matters for your application: factual correctness, relevance, tone, safety, compliance with policies, cost, and latency. Start with simple conditional rules (e.g. error count > 0) and gradually add more sophisticated classifiers as needed.

Alerts

Alerts notify humans or automated systems when something crosses a threshold. In Neatlogs, you can create alert rules based on metrics (cost, latency, error rate, token usage, detection count) or detection results. Alerts have:

- **Severity:** a qualitative level (info, warning, critical) indicating how urgent the issue is.
- **Window:** a time window over which the metric or detection is aggregated (e.g. last 10 minutes).
- **Threshold:** a numeric boundary (e.g. latency > 1s, error rate > 5%).
- **Cooldown:** a period after firing during which the alert will not fire again (to avoid spam).
- **Filters:** conditions on properties (workflow name, model, agent name) to scope the alert.
- **Channels:** where to send the alert (Slack channel, e-mail, incident management tool).

Alerts help you catch regressions quickly. For example, you might set an alert when the number of negative hallucination detections per hour exceeds a threshold, or when the p95 latency of a tool spikes. Alerts should be actionable: if you can't define a clear remediation step, you probably don't need an alert.

Analytics

Analytics aggregate metrics, detections and other data across many traces to help you understand trends. Typical analytics include:

- **Cost analysis:** breakdown of cost by model, workflow, or tool.
- **Latency analysis:** p50/p90/p95 latency per model, tool or workflow.
- **Error analysis:** counts and rates of different error types over time.
- **Detection analysis:** frequency of positive/negative/neutral detections, broken down by detection type and workflow.
- **Tool analytics:** number of calls, error rate, retry rate, payload sizes, and trend direction for each tool.

Analytics support filtering on dimensions like workflow, model, tool, span type, detection type, and time range. Good analytics let you slice and dice your data to answer questions such as "Which tools are causing the most latency?", "Is the cost per successful output rising?", or "Which model is producing the highest rate of hallucinations?"

Triage and Suggestions

At the intersection of detections and analytics lies **triage**. When a detection fires or an alert triggers, you must decide what to do. Triage involves reviewing the

offending trace, reproducing the issue, and determining whether it is a one-off glitch or a systemic problem. Neatlogs' data model includes comment threads and suggestion boards to support this workflow. A **triage suggestion** might recommend increasing the temperature for a model, adding a guardrail to a tool call, or splitting a workflow into smaller steps. While these features were not fully captured in the schema, the intent is clear: use the signal from your observability stack to drive continuous improvement.

Governance & Control

Engineering does not stop at building; you must also control, govern and prove the behaviour of your agentic systems. This chapter discusses the concepts of runtime control, attestation, replay, evaluation, and policy enforcement. These ideas are crucial when deploying agents into regulated or high-stakes environments.

Runtime Control

Runtime control refers to the ability to intercept and influence an agent's actions while it is running. A runtime control plane acts as a gateway: every call to the LLM or a tool goes through it. The plane can enforce rate limits, check permissions, redact sensitive data, block forbidden actions, and inject additional context. Without runtime control you are limited to post-mortem analysis; with it you can prevent disasters before they happen.

Proof and Attestation

When agents interact with sensitive systems — bank accounts, medical records, personal data — you need proof of what they did. An **attestation** or **receipt** is a tamper-evident record of an agent action. It includes the agent's identity, the user's identity (if delegated), the time, the prompt hash, the tool call and its parameters, the policy version, and the result. The World Bank's guidance on tamper-proof logs notes that logs should be protected from unauthorized access and should be digitally signed so that any alteration is detectable[9]. Estonia's national system chains digitally signed logs so that it is "difficult to change their history"[10]. Agent attestation takes a similar approach: every record is cryptographically signed and linked to the previous record, forming an immutable ledger. This ledger is invaluable for audits, investigations and compliance reports.

Replay and Simulation

Replay is the ability to rerun an agent's trace under controlled conditions. You might replay a failed run to reproduce a bug, compare different model versions, test a new prompt or tool, or simulate what would have happened with updated policies. Replay requires capturing enough context — prompts, tool inputs, random

9. File citation: 809247269174362, L97-L118.

10. File citation: 809247269174362, L114-L116.

seeds, state snapshots — to faithfully reproduce the original execution. A strong observability stack stores this information automatically.

Evaluation

Evaluation goes beyond simple error counts and latency metrics. It asks: "Did the agent produce a useful outcome?" and "How good was this outcome?" Evaluations can be performed:

- **Automatically**, using heuristics or models to assess relevance, correctness, completeness or tone.
- **By humans**, via a feedback loop that labels outputs as good or bad.
- **By downstream effects**, such as conversion rates or task completion metrics.

Evaluations should be tied back to specific traces and can inform detection thresholds, alert rules and triage suggestions.

Governance

Governance encompasses policies, permissions, roles and responsibilities. In the context of agents it includes:

- **Identity:** each agent has an identity and may act on behalf of a user. Delegated identity allows you to track who initiated an action.
- **Permissions and scopes:** what resources the agent can access and what actions it can perform.
- **Policy versions:** the rules at the time of execution. When policies change you need to know which version applied to past actions.
- **Approval flows:** some actions require human approval. The runtime control plane can pause execution and wait for approval.
- **Role-based access control (RBAC):** different roles have different permissions. A developer may be allowed to replay traces but not to modify production policies.

Governance provides guardrails and accountability. Without it, agents are free to roam; with it, they become trustworthy members of your software ecosystem.

State Diff and Side-Effect Verification

An agent's success is measured not by whether it *said* it did something, but whether it actually changed the world. A **state diff** captures the difference between the world before and after a run: which database rows were modified, which files

were created, which tickets were closed. Verifying side effects means checking that the intended changes occurred and no unintended changes were made. Without side-effect verification you risk phantom success, where an agent claims victory but nothing useful happened.

Guardrails and Policy Enforcement

Guardrails are checks imposed on an agent's outputs and actions. They include schema validation (is the output JSON valid?), value constraints (is the date in the future?), safety filters (is the content toxic?), and permission checks (is this tool call allowed?). Guardrails can run before or after actions. **Pre-execution guardrails** stop dangerous actions before they happen; post-execution guardrails catch harmful outputs and request corrections. **Policy enforcement** is the application of guardrails at scale, defined by your governance policies and executed by the runtime control plane. A strong opinion: guardrails are not just nice-to-have; they are the difference between a toy and a production system.

Human-in-the-Loop

Agents may be autonomous, but humans remain in charge. **Human-in-the-loop** means that certain actions require a human to review and approve. For example, an agent can draft an e-mail but a human must click send; an agent can prepare a transfer but a human must authorise it. Observability tooling should support this flow by surfacing the context (trace, inputs, outputs) and allowing the human to make an informed decision.

Idempotency

Idempotency means that repeating an operation has the same effect as executing it once. When an agent calls a tool, the request should include an idempotency key so that retries do not cause duplicate side effects. Idempotency is a common pattern in distributed systems to ensure reliability in the face of retries and network failures.

Drift Monitoring and Long-Horizon Analysis

We introduced drift in the Systems chapter. **Drift monitoring** uses evaluations, analytics and state diffs over time to detect when an agent's behaviour changes. Long-horizon analysis examines sequences of traces, looking for subtle patterns: increasing latency, creeping cost, rising hallucination rates. Drift is inevitable; detecting it early is the only way to respond.

Multi-Agent Patterns

The research was clear: **multi-agent systems are overused**. Many flashy demos chain together five agents when a simple script would suffice. Still, there are legitimate patterns where multiple agents provide value. This chapter explores the common patterns, warns against over-engineering, and offers a few rules of thumb for when to add another agent.

Orchestrator

An **orchestrator** is a top-level agent responsible for managing other agents. It receives a task, decides which sub-agents to invoke, collects their outputs, and produces a final result. Orchestrators maintain the global state and handle retries and error handling. They are useful when sub-tasks are clearly separable (e.g. research versus writing) or when you want to parallelize work.

Builder–Reviewer Pair

In this pattern, one agent (the **builder**) produces a draft answer or artifact, and another agent (the **reviewer**) critiques it. The reviewer can ask the builder to make changes or can make corrections itself. This pair simulates the human writer/editor loop. It can improve quality, catch hallucinations and enforce style guidelines.

Specialist Agents

You might have one agent that specializes in data retrieval, another that writes code, and a third that summarizes results. Specialist agents are helpful when the tools, prompts or memory required for each task are distinct. For example, a retrieval agent might need access to a vector store and a summarization model; a coding agent might need a Python runtime and strict type contracts. Dividing responsibilities lets each agent have a smaller and more reliable prompt.

Digital Department

The **digital department** pattern imagines a whole team of agents: a manager, multiple workers, maybe even HR. This is the pattern the research cautioned against. Without clear separation of context and authority, multi-agent systems become fragile: messages get lost, context drifts, and costs explode. Only build a digital department if you have real parallel tasks or require persistent specialization.

Parallelism and Concurrency

Agents can run in parallel. For example, a research agent and a data-analysis agent can fetch information simultaneously, reducing overall latency. Parallelism is beneficial only when tasks do not depend on each other. When tasks are dependent, concurrency introduces complexity: how do you merge results, handle partial failures, or cancel in-flight tasks? Use concurrency with care.

Determinism vs Autonomy

One of the most contentious design choices is how much autonomy to give an agent. **Deterministic routing** means you decide ahead of time which tools to call and when; the agent simply fills in arguments. **Autonomous routing** means the agent decides its own plan. Deterministic systems are predictable and easier to debug; autonomous systems are flexible but harder to reason about. The research suggests using autonomy only when necessary; many tasks can be solved with well-structured prompts and simple conditional logic.

Task Ledger and Handoff

A **task ledger** is a persistent record of tasks assigned, in progress, and completed. In multi-agent systems, the ledger prevents tasks from being lost or duplicated. **Handoff** refers to passing context from one agent to another. Clear handoff is essential: the receiving agent needs sufficient summary of what was done, the remaining objectives, and any relevant state. Without a good handoff, context drift occurs and agents may duplicate work or contradict each other.

Context Drift and Aggregation

When multiple agents work on the same task, their contexts can diverge. One agent may learn something that another agent never sees. **Aggregation** is the process of reconciling their outputs into a single consistent view. This might involve a summarization agent that synthesizes multiple research reports or a voting mechanism that chooses among competing answers. Without aggregation, the final output may be inconsistent or incoherent.

When to Use Multi-Agent Architectures

Use multiple agents when:

- You have tasks that are truly independent and can run in parallel.
- You need specialized capabilities that require different prompts, tools or environments.

- You want explicit review and critique cycles.

Avoid multiple agents when:

- The tasks can be accomplished with a simple sequence of tool calls.
- You are tempted to replicate human organizational structures without clear technical benefit.
- You lack the monitoring and governance infrastructure to handle the complexity.

Neatlogs Data Model

The Neatlogs data model is an embodiment of the concepts discussed in the previous chapters. It stores traces, spans, detections, alerts, analytics and other metadata in a structured way so that teams can inspect, search and debug agent runs. This chapter walks through the main entities and how they relate.

Trace

A **trace** represents a single run of a workflow or agent. It has a unique identifier and contains:

- **workflow:** the name of the workflow or pipeline being executed.
- **framework:** which agent framework was used (LangChain, CrewAI, custom, etc.).
- **root span:** the top level span, whose children represent the steps in the run.
- **status:** success, error, or partial success.
- **duration:** how long the run took from start to finish.
- **token usage and cost:** aggregated across all LLM spans.
- **detections:** results of any detection rules applied to the trace.
- **comments and votes:** feedback from users or team members.

Traces are searchable via filters on time, workflow, status, cost, model, tool, detection results and other attributes. Each trace can be expanded into a tree of spans for detailed analysis.

Span Types

Each trace consists of **spans**. Spans capture individual operations and are typed:

- **Agent Action:** an internal decision by the agent, such as choosing the next tool or deciding to end the run. Attributes include the agent name, the thought process (if captured), and the selected tool.
- **LLM:** a call to a language model. Attributes include provider, model, version, input prompt, output completion, token counts, latency, cost, and any errors.
- **Tool Call:** a call to an external API or tool. Attributes include tool name, description, input arguments, result, status, latency, error type and error message.
- **Retrieval:** a query against a vector store or database. Attributes include the indices queried, the query string or embedding, the number of results returned, and the sources.

- **Chain:** a nested call to another workflow or chain (e.g. using LangChain's `LCEL` or a custom sub-workflow). Spans of this type have their own child spans.

Each span has a parent span ID, forming a tree. The `depth` of a span indicates how many levels deep it is. Spans also have timestamps, durations, status (success, error), token counts (for LLMs), and error details.

Workflow

A **workflow** is the template for a run. It defines the sequence or graph of operations the agent should perform. In Neatlogs, workflows are identified by name and can be filtered on. You might have workflows like `research_and_write`, `customer_support_chat`, or `data_extraction`. Workflows can be versioned; the version is captured in the trace metadata.

Detections and Results

Detections are stored as separate objects linked to traces or spans. A detection object includes the rule ID, type (conditional, regex, classification), state (positive, neutral, negative), score or threshold used, and an optional explanation. Detection results are attached to traces so you can see at a glance which runs had issues.

Comments and Threads

Traces and spans can have **comment threads**. A comment thread has messages (with author and timestamp), resolved/unresolved status, and optional selected text (context highlighting). Comments allow engineers to discuss issues directly within the trace, tag colleagues, and document remediation steps. Emojis or reactions may be used for quick acknowledgment.

Alert Rules

Alert rules are top-level objects that define how and when notifications should be sent. A rule specifies the metric or detection to watch, the threshold, the time window, severity, cooldown period, and target channels (e.g. Slack). When a rule triggers, an alert event is generated and recorded. Alerts help bridge observability and operations.

Analytics Objects

Neatlogs stores aggregate analytics to power dashboards. Cost analytics group token usage and monetary cost by workflow, model or tool. Latency analytics compute percentiles for each span type. Error analytics classify errors and count their occurrence. Detection analytics count positive/negative/neutral detections. Tool analytics

aggregate call counts, error rates and retry rates. These objects allow quick retrieval of high-level statistics without scanning raw traces.

AI Providers

Neatlogs keeps track of **AI provider configurations**. An AI provider record includes the provider name (e.g. `openai`, `anthropic`), the base URL, the list of available models and deployments, and which model is the default. It also stores API keys for each deployment. *Warning: in the captured schema, the* `apiKey` *field was returned in plaintext. In production systems, API keys should be write-only and masked to prevent leakage.* AI provider metadata allows Neatlogs to compute costs, compare models, and manage provider health.

Unmodeled Concepts

The captured schema hinted at **evaluations**, **triage suggestions** and **drafts**, but the details were not fully present. Evals likely correspond to per-trace or per-span evaluations of output quality. Triage suggestions may provide recommended actions when detections fire (e.g. “review this tool call”, “adjust the system prompt”). Drafts might be in-progress versions of comments or evaluations. These pieces round out the workflow from detection to remediation.

Usage Examples

Understanding abstract terms is easier when you see them in action. This chapter presents a few concrete scenarios that tie together the concepts from the previous chapters. The examples are intentionally simple; feel free to expand them in your own experiments.

Example 1 – A Simple Research Agent

Suppose you build a research agent that takes a question, searches the web, summarises the results, and sends you an e-mail with the answer. Here's how the trace might look:

1. **Agent Action span (root):** the agent receives the question "What is the capital of France?"
2. **LLM span:** the agent calls an LLM to plan its steps. The prompt instructs the model to think about which tool to use; the completion suggests a web search.
3. **Tool Call span:** the agent invokes the `web_search` tool with the query. The tool returns a list of documents.
4. **Retrieval span:** the agent calls `vector_store.retrieve` to fetch relevant snippets from previously indexed documents.
5. **LLM span:** the agent calls the LLM again to summarise the retrieved snippets. The completion contains a concise answer.
6. **Tool Call span:** the agent uses the `send_email` tool to send the summary to your e-mail address.
7. **Agent Action span:** the agent decides it has completed the task and returns a success message.

For each span, Neatlogs records start and end times, inputs and outputs, token counts, latency, and any errors. If the web search fails, the tool call span will have an error, and the agent may decide to retry. Metrics aggregated over many such runs will tell you how often the agent succeeded, how much it cost, and how long it took.

Example 2 – Detection and Alert

Imagine you add a detection rule to flag hallucinations. You train a classifier that assigns a score between 0 (highly factual) and 1 (likely hallucinated). You set a threshold of 0.7; if the score exceeds this threshold the detection state is **negative**. One day a run produces a hallucination score of 0.85. Neatlogs attaches a detection result to the trace with state = negative and score = 0.85. You also create an alert

rule: if there are more than three negative hallucination detections in a five-minute window, send a Slack notification. The alert fires, and your team investigates. They discover the retrieval index was outdated; updating it reduces hallucinations. The detection and alert saved your users from bad answers.

Example 3 – Attestation and Replay

Your agent is authorised to update customer records in a CRM. When the agent writes to the CRM, the runtime control plane generates an attestation: it records the agent ID, user ID, time, prompt hash, tool call arguments (customer ID, new status), policy version and a digital signature. Six months later a customer disputes a change. You search the attestation ledger and find the exact record. You can prove that on June 1, 2025, your agent, acting on behalf of user *sks*, changed the status of customer 12345 to `prospect`. You also replay the trace to verify that the agent's logic still produces the same action. The combination of attestation and replay provides auditability and confidence.

Example 4 – Drift Detection

Over time you notice that your research agent's summaries are getting longer and more verbose. You compute the average number of words in the summary for each week and plot it. The graph shows a steady rise from 50 words to 200 words over three months. This is **drift**. You investigate and discover that a recent model update increased the maximum context window, causing the agent to include more context in its responses. By monitoring this trend you can adjust prompts or impose word limits before costs spiral.