The AI Agent

A Product Definition Across Every Surface

Part of the series *On the Design of AI Agents*
Prepared: April 26, 2026 | v0.10

John Dulay · Chief Product Officer

Overview

This document proposes a precise, product-level definition of what an AI agent actually is, how it behaves, and where it operates. It is written for product leaders who need to understand AI agents well enough to make good decisions about how they are designed, built, and deployed — not just what they are called.

The central thesis is this: an AI agent should be defined once and then deployed across every surface that needs it. Not a different agent per surface. One agent, with a defined identity, a defined set of capabilities, and a defined set of rules. Each surface draws on the portion of that definition that is appropriate for its context. A voice surface uses less of it than a research pipeline. A scheduled automation uses it differently than a live chat. But it is always the same agent.

The surface configuration determines how much of that capability is expressed in any given context. A voice surface runs a tight, fast loop. An async research pipeline runs a deep, multi-iteration loop. A scheduled automation runs without a user present at all. The agent is the same in all three cases. What changes is how the surface calls it, constrains it, and receives its output.

This framing has direct consequences for how you build, maintain, and scale agents. Build the agent once and it improves everywhere simultaneously. Update the Agent Charter and every surface reflects the change. Add a new surface and the agent is already capable — you are only writing the integration, not the intelligence.

This document organizes that framing into three distinct concepts:

Section	Description
The Agent	Agent Intelligence · Agent Charter — the structural components that define what the agent is
Agentic Loop	Perceive · Reason · Act · Observe · Repeat — the behavioral pattern that defines what the agent does
Agent Surfaces	Web Chat · Slack · Voice · API · Email · CLI · Browser · Automation — where the agent operates

These three concepts are kept deliberately separate. The Agent defines structure — what the agent is built from. The Agentic Loop defines behavior — how the agent operates. Agent Surfaces define an environment — where the agent exists in the world. Mixing these three has

been the source of most AI agent architecture confusion; keeping them separate is what makes the architecture maintainable, extensible, and clear.

The Agent

Before an agent can do anything, it has to exist as something. The Agent section defines what an AI agent is made of — the structural components that give it identity, capability, and purpose. Everything in this section is built once and reused everywhere. Nothing here is surface-specific.

The agent is defined by two structural components that together answer the fundamental question: what is this agent? The Agent Intelligence provides the raw cognitive capability. The Agent Charter directs that capability toward a specific purpose, with a specific identity, governed by specific rules.

The Core Distinction

The Agent Intelligence provides the capability to think. The Agent Charter governs how that thinking is expressed. Without an Agent Charter, the Agent Intelligence is capable of anything but committed to nothing.

Agent Intelligence

The Agent Intelligence is the large language model (LLM) at the foundation of every agent — the reasoning and language capability that interprets inputs, makes decisions, calls tools, and generates responses. It is stateless, generic, and interchangeable. It has no purpose, no values, and no identity of its own.

- **Model selection:** Choose based on task profile. Frontier models (Claude Opus, GPT-4o) excel at complex reasoning and long-context tasks. Smaller, faster models (Claude Haiku, GPT-4o mini) suit high-throughput, latency-sensitive surfaces like voice or chat.
- **Swappability:** Design the Agent Charter to be model-agnostic. Abstract the Agent Intelligence behind an interface so you can run A/B tests, fall back to an alternative model during an outage, or upgrade without rewriting the Agent Charter.
- **Multimodal support:** Modern frontier models accept text, images, audio, and documents as input. If your Agent Surfaces include voice, screenshots, or file uploads, ensure the Agent Intelligence supports the required modalities.
- **Cost and latency tradeoffs:** Model API cost is proportional to tokens processed. Per-surface model routing — using a lighter model for simple queries and escalating to a frontier model only when needed — can significantly reduce costs without degrading quality.

Agent Charter

The Agent Charter is the base specification that gives the Agent Intelligence a stable, purposeful identity. It is not a complete enumeration of everything the agent can do — you

cannot know that upfront. It is the floor: who this agent is, what it is here to do by default, and what it must never do regardless of context. Capability grows from this base through tools, skills, and discovery patterns defined within it.

The Agent Charter is built once and reused across all Agent Surfaces. Its eight components fall into three natural groups: identity (who the agent is), capability (what it can do), and governance (what it is permitted to do). Together they establish the agent's baseline — a stable foundation that grows through use, not a closed specification that must anticipate everything upfront.

- **Identity & Soul:** Who the agent is — persona, values, voice, communication style, and the constitutional rules that govern its character across every surface and interaction.
- **Skills:** Modular capability bundles that can be attached or detached at runtime, giving the agent domain-specific knowledge, tool access, and behavior without maintaining separate agents.
- **Tools & Tool Definitions:** The external actions the agent can take — read, write, compute, and agent-calling tools, exposed via MCP or custom integrations.
- **Knowledge Bases:** Curated, queryable information stores (vector, structured, or graph) that give the agent its domain expertise, accessed via RAG at query time.
- **Context Window Infrastructure:** The discipline of deciding what information occupies the Agent Intelligence's finite working memory at any given moment.
- **Memory & State:** Working, episodic, semantic, procedural, and persistent memory — spanning in-session state through cross-session long-term knowledge. The Charter defines what infrastructure exists for each type; the agent draws from all available types as judgment dictates.
- **Planning & Reasoning:** A library of reasoning patterns the agent can draw from — Chain of Thought, ReAct, Reflection, Plan-and-Execute, and others. The Charter does not pre-select a strategy. It provides reasoning guidance (when to prefer certain approaches in this domain) and makes the full library available. The agent selects and combines patterns as judgment dictates. New patterns are added to the library as they are discovered or defined — the agent benefits immediately without Charter changes.
- **Business Logic & Guardrails:** Domain-specific rules, escalation paths, compliance constraints, and access control policies that govern what the agent is permitted to do.

Agent Charter — Component Detail

1. Identity & Soul

Identity & Soul is the foundational layer of the Agent Charter — the component that makes this agent this agent rather than a generic assistant. It is loaded on every session and remains constant across every surface, every skill, and every interaction. While all other components may vary by context, Identity & Soul does not.

- **Persona:** The agent's role, name, and character — how it presents itself and relates to users.
- **Values:** The core principles the agent holds and will not compromise on, regardless of user instructions.

- **Voice & Style:** Communication tone, vocabulary, and formatting preferences that make the agent's responses recognizably consistent.
- **Constitutional Rules:** The non-negotiable behavioral boundaries — what the agent will never do, regardless of what any other component permits. These override everything else.
- **Dynamic Injection:** User-specific context, active session metadata, and current date/time injected at call time, personalizing behavior without changing identity.

Best Practice

Treat Identity & Soul as code. Version-control it, test changes against regression suites, and deploy it through a controlled pipeline. A change to Identity & Soul affects every surface simultaneously — never modify it casually in production.

2. Skills

Skills are modular, reusable capability bundles that can be attached to or detached from the Agent Charter at runtime. Each skill encapsulates a specific domain of knowledge, behavior, and tool access — enabling the same Agent Charter to serve a billing team, a sales team, and an engineering team without being rebuilt for each.

- **Instructions:** Domain-specific behavioral guidance appended to the system prompt (e.g., a Billing Support skill adds billing tone, policy, and escalation rules).
- **Tool Access:** The subset of tools this skill is permitted to call.
- **Knowledge Scope:** Pointers to the specific knowledge base sections or vector DB namespaces relevant to this skill.
- **Output Format Rules:** Skill-specific formatting preferences (e.g., a Code Review skill always responds with structured diff summaries).

Skill Composition Patterns

- **Single-skill agents:** Simpler to reason about; best for narrow, well-defined use cases.
- **Multi-skill agents:** Skills are selected dynamically based on user intent classification; requires a routing layer.
- **Skill inheritance:** A base skill defines shared behavior; specialized skills override or extend specific behaviors.

Why Skills Matter

Updating a shared skill propagates instantly to every agent that uses it. A change to the Billing skill updates the web chat agent, the Slack bot, and the voice agent simultaneously — without touching any of them individually. Skills are a library: the Charter makes them available, the agent activates what it needs during reasoning. No surface involvement. This propagation guarantee requires a skill registry: a versioned store from which skills are resolved at runtime. The registry must handle version pinning, deprecation, and resolution failures. It does not emerge automatically.

3. Tools & Tool Definitions

Tools are the agent's ability to act in the world — external functions the agent can call to retrieve information, execute operations, or interact with other systems. The Charter does not pre-register tools. Like skills and reasoning patterns, tools are discovered and selected by the agent at runtime as judgment dictates. The Charter defines two things: the trusted sources from which the agent is authorized to discover tools (MCP servers, registered APIs, internal service registries), and the tool schema that any discovered tool must conform to before the agent can use it. The agent discovers tools from trusted sources, validates them against the schema, and selects them during reasoning. The Charter author never needs to know what tools exist — only whose tools are trusted. Action-class prohibitions (never delete without confirmation, never send communications without user-scoped credentials, never write to production databases) belong in guardrails as hard stops, not in a pre-registered tool list.

- **Read tools:** Search, database queries, document retrieval, API GET requests. Low risk, generally always permitted.
- **Write tools:** Database mutations, API POST/PUT/DELETE, sending messages, creating records. Require explicit access control.
- **Compute tools:** Code execution, calculation engines, data transformation pipelines.
- **Agent-calling tools:** Tools that invoke other agents or sub-agents as part of a multi-agent architecture.

MCP — Model Context Protocol

The Model Context Protocol (MCP) has emerged as the dominant standard for connecting agents to external tools and data sources. Rather than custom integrations per tool, MCP defines a universal interface that agents use to discover and call any MCP-compatible server. By early 2026, MCP powers tens of millions of monthly SDK downloads and thousands of community servers, enabling near-instant integration with services ranging from CRMs to code repositories.

Tool Reliability

- Always define fallback behavior for tool failures — agents that crash on a failed API call are not production-ready.
- Use structured output schemas on tool responses so the agent can parse results consistently.
- Implement retry logic with exponential backoff for transient failures.
- Log every tool call and its result for observability and debugging.

4. Knowledge Bases

Knowledge bases are the curated, queryable information stores that give the agent its domain expertise. While Memory tracks dynamic conversational state, Knowledge Bases hold stable reference content — product documentation, policy manuals, support articles, or any corpus the agent needs to answer questions accurately.

Knowledge Base Types

Type	Best For
Vector / Embedding store	Unstructured text — docs, articles, support tickets, emails. Enables semantic similarity search.
Structured / SQL	Tabular data with known schemas — product catalogs, pricing tables, user records, inventory.
Knowledge graph	Highly relational data where connections matter — org charts, dependency maps, entity relationships.
Hybrid (vector + structured)	Most production systems. Vector search finds candidates; structured filters refine results.

RAG — Retrieval-Augmented Generation

RAG is the dominant pattern for connecting Knowledge Bases to the Agent Charter. Rather than baking all knowledge into the model's weights or the system prompt, the agent retrieves only the relevant passages at query time and injects them into the context window.

- **Query → Embed → Search → Inject:** The user's message is embedded, matched against the knowledge base index, and the top results are inserted into the context window before the Agent Intelligence generates a response.
- **Hybrid retrieval:** Combining semantic (vector) search with keyword (BM25) search reduces the risk of missing exact-match terms that semantic search alone would rank poorly.
- **Re-ranking:** A lightweight re-ranker model scores retrieved passages for relevance before injection, improving precision when initial retrieval returns noisy results.
- **Metadata filtering:** Attach structured metadata to every document (source, date, department, access level) and filter at retrieval time so the agent only sees content it is authorized to use.

Scope Note

This section covers how Knowledge Bases integrate with the Agent Charter. The design and maintenance of knowledge bases themselves — chunking strategies, embedding model selection, indexing pipelines, freshness guarantees, and governance — is covered in the companion document: Knowledge Infrastructure for AI Agents.

5. Context Window Management

The context window is the Agent Intelligence's working memory — the finite space that holds everything the model can see at any one time. Modern frontier models support context windows of 128K to 1M+ tokens, but managing what occupies that space is one of the most critical engineering disciplines in agent design. Context engineering has displaced prompt engineering as the defining skill in production agent development.

What Lives in the Context Window

Content Type	Typical Token Budget
Identity & Soul (static + dynamic)	500 – 4,000 tokens
Conversation history (recent turns)	2,000 – 20,000 tokens
Retrieved documents / RAG results	2,000 – 50,000 tokens
Tool call history (current session)	500 – 5,000 tokens
Injected user profile / preferences	200 – 1,000 tokens
Active task state / planning trace	500 – 5,000 tokens

Context Management Strategies

- **Sliding window:** Keep only the most recent N turns; discard older history. Simple but loses long-range context.
- **Summarization compression:** Periodically summarize older conversation segments into compact summaries before they scroll out of the window. Preserves key facts while freeing token budget.
- **Tiered memory (RAM/disk model):** Treat the context window as RAM. Hot information stays in-context; warm/cold information is archived to external memory and retrieved on demand.
- **Selective retrieval:** Rather than loading full conversation history, retrieve only the turns semantically relevant to the current query using vector search.
- **KV-cache optimization:** Structure prompts so that stable content (Identity & Soul, persistent user context) appears early and doesn't change between turns, maximizing cache hits and reducing cost.

Key Insight

Most agent failures in production are context failures, not model failures. The Agent Intelligence is usually capable — but it was given the wrong information, too much noise, or critical context scrolled out of the window. Systematic context engineering is what separates demo-quality agents from production-quality ones. The Charter defines the context window infrastructure: how large it is, what memory stores are available, what summarization tools exist. The agent manages its own context during reasoning — deciding what to retrieve, when to compress, and what to keep. Prescribing a fixed sliding window or token threshold in the Charter is the same mistake as pre-selecting a reasoning strategy: the agent has better information at runtime than the Charter author has at design time.

6. Memory & State

Memory transforms the agent from a stateless request-handler into a persistent collaborator. The Charter defines what memory infrastructure is available — which stores exist, their connection strings, and their capabilities. The agent decides which memory types to use and when. The Charter does not restrict access to specific memory types; restricting the agent's access to its own memory is the same mistake as restricting its reasoning strategies. Modern agent memory architectures draw on concepts from human cognition, organizing information by type, recency, and relevance.

Memory Type	Scope	Storage	Example Use
In-Context / Working	Current session only	Context window (RAM)	Ongoing conversation, active task state
Episodic	Specific past events	Vector DB (semantic search)	"Last time we discussed X, you preferred Y"
Semantic	General facts & knowledge	Vector DB / Knowledge Graph	Product catalog, company FAQs, user profile
Procedural	How to do things	Embedded in system prompt or skills	Workflow steps, tone guidelines, policies
Persistent / Long-term	Cross-session facts	Key-value or relational DB	User preferences, account history, prior decisions

Memory Storage Technologies

- **Vector databases (Pinecone, Weaviate, Chroma):** Store embeddings of text; support semantic similarity search. Best for episodic and semantic memory retrieval.
- **Knowledge graphs:** Store facts as interconnected nodes and relationships. Stronger than vector DBs at multi-hop reasoning ("who manages the team that owns project X?").
- **Relational / key-value stores:** Fast retrieval of structured facts (user preferences, account records). Best for persistent/procedural memory.
- **In-context (no external store):** Working memory for the current session. Zero latency; lost when the session ends.

Multi-Agent Memory Considerations

- Tag every memory entry with its source actor (user-stated vs. agent-inferred vs. system-generated).
- Use read/write access controls: not every agent should be able to write to shared long-term memory.
- Implement temporal reconciliation: when a new fact contradicts stored memory, preserve the historical record rather than silently overwriting it.

7. Planning & Reasoning

The Planning & Reasoning component defines the library of reasoning patterns available to the agent. These are not a menu to choose from upfront — they are tools the agent understands and can draw from during the Reason stage as judgment dictates. The agent may use a single pattern, combine several, or reason in ways that do not map cleanly to any named pattern. New

patterns are added to the library as they are discovered or defined; the agent benefits immediately without any Charter changes. The descriptions below explain what each pattern is good for, so the agent can make informed choices during reasoning.

- **ReAct (Reasoning + Acting):** The agent interleaves reasoning traces with tool calls in a loop: think → act → observe → think. Highly adaptive; best for dynamic, unpredictable tasks.
- **Plan-and-Execute:** The agent generates a full plan upfront, then executes each step sequentially without re-planning. More cost-predictable; less resilient to unexpected results mid-execution.
- **Reflection loops:** After completing a task, the agent evaluates its own output against the original goal and decides whether to revise. Adds self-correction without human intervention.
- **Human-in-the-loop checkpoints:** For high-stakes or irreversible actions, the agent pauses and requests explicit human approval before proceeding. Essential for production safety.

Scenario	Recommended Pattern
Simple Q&A, summarization	Direct LLM call — no agent needed
Multi-step research with variable sources	ReAct
Predictable workflow with known steps	Plan-and-Execute
Long-running task over hours/days	Plan-and-Execute + checkpointing
High-stakes actions (purchases, deletions)	Any pattern + human-in-the-loop
Creative generation with quality bar	ReAct + Reflection loop

8. Business Logic & Guardrails

Business Logic encodes the domain-specific rules, constraints, and decision trees that make the agent useful within a particular organization. Guardrails in this component define the hard floor — the things the agent must never do regardless of context. They are not an exhaustive rule list trying to anticipate every harmful scenario; that approach produces brittle, unmaintainable Charters. Instead, guardrails cover only true hard stops: regulatory prohibitions, irreversible actions, and explicit out-of-scope behaviors. Everything else is handled by the agent's Identity & Soul — a strong, well-defined character does more safety work than a long rule list, because it generalizes to situations no rule anticipated.

- **Escalation paths:** Conditions under which the agent hands off to a human (e.g., sentiment below threshold, high-value account, regulatory topic detected).
- **Access control policies:** Which users or roles can invoke which tools, access which data, or trigger which workflows.
- **Compliance guardrails:** Hard-coded prohibitions for regulated industries (finance, healthcare, legal) — actions the agent must never take regardless of user instructions.

- **Decision trees:** Structured rules for routing, prioritization, and conditional behavior that are too deterministic to leave to model judgment alone.
- **Audit logging:** Every decision with business impact should be logged with enough context to reconstruct why the agent took the action it took.

The Agentic Loop

The Agent defines what the agent is. The Agentic Loop defines what it does. Once the Agent Charter is in place and the Agent Intelligence is available, the loop is what puts them to work — the repeating cycle through which the agent perceives its environment, reasons about what to do, acts in the world, and observes the results.

Every major AI company has converged on this same core pattern. The loop is what distinguishes an agent from a chatbot: a chatbot responds once and stops; an agent loops until the task is complete.

Stage	What Happens
Perceive	The agent receives input from the surface — user messages, API calls, scheduled triggers. Raw signals are normalized, authenticated, and structured into the standard message contract before reaching the Agent Charter.
Reason	The Agent Charter processes what it perceived. The Agent Intelligence drives planning, tool selection, memory retrieval, and decision-making governed by the Charter's identity, skills, knowledge, and business logic.
Act	The agent responds — delivering output back to the surface in the platform's native format. Tool calls execute, APIs are invoked, messages are sent. The agent does something in the world.
Observe	The results of action feed back into the agent. Conversation context is captured, memory is updated, outcomes are logged. The agent learns from what just happened before the next iteration begins.
Repeat	The loop runs continuously until the task is complete or a stopping condition is reached. Each iteration builds on the last — the agent adapts, corrects, and improves within the session.

Why the Loop Matters

The power of the Agentic Loop is not in any single stage — it is in the iteration. Each pass through the loop builds on the last. The agent adapts based on what it observed, corrects errors it detected, and progressively moves toward task completion in ways that no single-pass system can match.

Perceive

Perceive is where the Agentic Loop begins. The agent receives a signal from an Agent Surface — a user message, an API call, a scheduled trigger, a tool result — and prepares it for reasoning. Nothing reaches the Agent Charter without passing through the Perceive stage.

- **Inbound normalization:** Parse the surface-specific payload and map it to the standard message contract (user_id, session_id, message, surface, capability_flags, metadata).
- **Identity resolution:** Resolve the surface's identity tokens into a verified user_id. The Agent Charter never receives raw OAuth tokens or platform-specific session cookies.
- **Session management:** Establish or resume conversation threading. Retrieve relevant session context from memory to prepare for reasoning.
- **Capability flags:** Inspect the surface and inject the appropriate behavioral constraints — no_markdown: true for voice, max_tokens: 160 for SMS — so the Agent Charter tailors its output accordingly.
- **Multimodal grounding:** If the surface sends images, audio, or documents alongside text, these are prepared and structured for the Agent Intelligence.

Common Pitfall

Most agent failures are perception failures, not reasoning failures. If the agent receives poorly structured, stale, or incomplete context at the Perceive stage, no amount of reasoning capability will produce a good result. Investment in the Perceive stage pays the highest returns in production.

Reason

Reason is where the Agent Charter does its work. The Agent Intelligence processes the perceived input, governed entirely by the Agent Charter's identity, skills, knowledge, and business logic. This stage is internal to the agent — no surface interaction occurs during reasoning. The Reason stage is not a black box of LLM activity — it is a governed reasoning phase with defined entry conditions, an agent-directed internal structure, exit conditions, and an output contract. The agent draws from a library of reasoning patterns — Chain of Thought, ReAct, Reflection, Plan-and-Execute, and others — selecting and combining them as judgment dictates. No strategy is pre-configured. The stage interface is stable regardless of what reasoning happens inside it. Understanding Reason as a structured phase with defined interfaces on both sides is what makes it tractable as an engineering concern.

Entry conditions — the Reason stage begins with a prepared context package from Perceive: the normalized user message, resolved user identity, injected capability flags, and any user-scoped credentials resolved by the surface adapter. Memory retrieval, RAG, and skill activation happen inside the Reason stage — the agent decides what to fetch and when, because it has context the Perceive stage does not. Perceive's job is narrow: normalize the input, resolve identity, resolve credentials. It does not pre-fetch context the agent has not yet determined it needs.

Reasoning phase (agent-directed) — the agent determines how to reason based on the task at hand. It draws from a library of patterns — Chain of Thought for linear reasoning with no external calls, ReAct for interleaved tool use, Reflection for quality-sensitive drafting and revision, Plan-and-Execute for tasks with clear sequential dependencies, and others —

selecting, combining, or inventing approaches as judgment dictates. No strategy is pre-selected in the Charter. The Charter provides reasoning guidance: descriptive context about when certain approaches tend to work well in this domain. The agent uses this the same way it uses its knowledge base — as a resource to draw on, not a rule to follow. Strategies are tools in a kit, not a menu to choose from. New patterns are added to the library as they emerge; the agent has access to them immediately without Charter changes. The Reason stage interface is stable regardless of what reasoning happens inside it.

Exit conditions — the Reason stage exits when one of four conditions is met: (1) the agent has sufficient information to produce a response; (2) the reasoning budget is exhausted — the agent produces its best available response with what it has; (3) an unrecoverable error occurs during reasoning — surfaced to the outer loop's Observe stage; (4) a guardrail is triggered — reasoning halts immediately and the outer loop handles escalation. The Reason stage never exits silently. Every exit produces a typed result that the Act stage can respond to appropriately.

Output contract — the Reason stage produces a structured response object for the Act stage, not a raw generated string. At minimum it includes: the response content, the exit condition type (success / budget-exhausted / error / guardrail), the reasoning patterns used, the resource units consumed, total cost, and the reasoning trace. The reasoning trace is the internal record of how the answer was reached — its structure reflects whatever reasoning actually happened, which may involve multiple patterns in a single invocation. It is the primary artifact for debugging, behavioral diffs between Charter versions, and audit. Whether the trace is logged persistently, retained ephemerally, or discarded is a design decision with compliance and cost implications — it must be specified in the Charter, not left as an infrastructure default.

Act

Act is where reasoning becomes reality. The agent delivers its response, executes tool calls, and takes actions in the world. Act is the outbound half of the Agentic Loop — the agent's output traveling back through the surface to the user or system.

- **Outbound formatting:** Transform the Agent Charter's response into the surface's native format — plain text for SMS, markdown for Slack, SSML for voice, HTML for email.
- **Tool execution:** External tool calls are fired — API requests, database writes, message sends, code execution. These actions have real-world consequences.
- **Streaming:** For latency-sensitive surfaces (voice, live chat), responses are streamed token by token rather than waiting for full generation. Target under 3 seconds for synchronous surfaces.
- **Error handling:** Failed tool calls trigger fallback behavior, retries with exponential backoff, or graceful degradation — never silent failure.

Observe

Observe is where the loop feeds back into itself. The agent captures what happened — what it said, what tools returned, how the user responded — and uses that information to update its state before the next iteration begins. Observe is what transforms the agent from a stateless responder into a system that learns within and across sessions.

- **Conversation logging:** Every interaction across every surface is logged to a central observability platform for cross-surface analysis and regression testing. Note: conversation logging (infrastructure) and context capture (agent-internal, next bullet) are distinct operations with different consistency requirements. Logging may be async and eventually consistent. Context capture must be synchronous and complete before the next Perceive stage begins — conflating the two can introduce race conditions in high-throughput deployments.
- **Context capture:** The current turn is added to conversation history, making it available for retrieval in subsequent turns.
- **Memory updates:** New facts, user preferences, and decisions are written to the appropriate memory store — episodic, semantic, or persistent.
- **Tool result integration:** Tool outputs are fed back into the reasoning context, enabling the agent to evaluate results and decide whether to act again.
- **Cost and latency tracking:** Token usage, tool call costs, and response times are recorded per surface for operational monitoring.

Repeat

Repeat is not a stage so much as the nature of the loop itself. After Observe completes, the agent evaluates whether its goal has been met. If it has, the loop exits. If it hasn't, the loop begins again — with the new context, the updated memory, and the accumulated learning from the previous iteration feeding directly into the next Perceive stage.

Each iteration builds on the last. The agent does not start fresh — it starts informed. How loops start, pause, complete, and break is covered in full in the Loop Lifecycle section below.

Loop Lifecycle

Every Agentic Loop has a lifecycle — it starts, it may be interrupted, and it ends. Understanding how loops start, pause, complete, and break is essential for building production-grade agents. Unmanaged loops are one of the most common sources of runaway costs, degraded user experience, and agent failures in production.

Completion Conditions — Normal Exits

A loop completes normally when it reaches a defined end state. The agent exits cleanly and communicates the outcome to the user or system.

- **Task complete:** The agent evaluates its output against the original goal and determines the task is fully satisfied. This is the ideal exit — the loop did its job.
- **Null result:** The agent determines that no further progress is possible — the information doesn't exist, the tool returned nothing useful, or the goal is unachievable given current constraints. The agent exits and explains why.
- **User ends session:** The user explicitly terminates the interaction. The agent saves relevant context to memory before exiting so the session can be resumed later.
- **Human checkpoint reached:** Miscategorized here. A human checkpoint is not a completion condition — the task is unfinished and the loop state must be preserved. This

belongs exclusively under Interruption. Listing it here risks implementers calling session.close() instead of session.suspend(), causing context loss. Removed from this list; see Interruption below.

Break Conditions — Abnormal Exits

A loop breaks when something unexpected prevents it from continuing. Break conditions should always be handled explicitly — a loop that breaks silently is a production risk.

- **Maximum iteration limit hit:** The agent has looped more times than the defined budget allows. This is a safety ceiling, not an expected exit. When hit, the agent surfaces its current state and asks the user how to proceed rather than failing silently.
- **Tool failure cascade:** A critical tool is unavailable or returns unrecoverable errors across retries. The agent cannot complete the task without it and exits with a clear explanation of what failed and why.
- **Context window exhausted:** The loop has accumulated so much conversation history, retrieved content, and tool results that there is no room left in the context window to reason effectively. The agent must summarize, compress, or hand off before continuing.
- **Timeout:** The surface's latency threshold has been exceeded. Voice and live chat surfaces enforce hard timeouts — the agent must exit the loop and deliver whatever it has, even if the task is incomplete.
- **Safety trigger:** A guardrail in the Agent Charter has been activated — the agent has detected a request or output that violates its constitutional rules. The loop halts immediately and the violation is logged.
- **Output validation failure:** The agent's response fails a post-generation validation check — hallucination detected, schema mismatch, or prohibited content identified. The loop breaks rather than delivering a bad response.

Production Rule

Every break condition must produce an observable signal. Loops that break silently — with no log entry, no user message, no cost record — are invisible failures. Instrument every break condition at the point it occurs, not after the fact.

Interruption — Paused but Resumable

Not every loop exit is final. Some loops are interrupted — paused in a defined state, waiting for a condition to be met before resuming. Interruption is distinct from both completion and breaking: the task is not done, but the agent is not failing either.

- **Human-in-the-loop checkpoint:** The agent has reached a decision point that requires human approval before taking a consequential action — a purchase, a deletion, a message send. The loop pauses, surfaces the pending action to the user, and resumes only after explicit confirmation.
- **Async wait:** The agent is waiting for an external event to occur — an email reply, a webhook callback, a third-party API to complete a long-running job. The loop suspends with its state checkpointed and resumes automatically when the event arrives.

- **Session expiry with state preserved:** The user's session times out, but the agent's task state is saved to persistent memory. When the user returns — on the same surface or a different one — the loop resumes from where it left off with full context intact.

Interruption vs. Breaking

The distinction matters in practice: an interrupted loop has a defined path to resumption and its state is preserved. A broken loop has no path to resumption and must communicate clearly to the user what happened and what their options are. Confusing the two leads to lost context and frustrated users.

Loop Governance

Production agents require explicit governance over their loops. Without it, loops run unchecked — accumulating costs, consuming context, and making decisions without oversight.

- **Iteration budgets:** Set a maximum iteration count per agent run, calibrated to the task type. Simple Q&A agents rarely need more than 3-5 iterations. Complex research agents may need 20+. Define the budget in the Agent Charter, not at the surface level. Precedence rule: the Charter defines the hard ceiling; capability_flags in the surface contract may only reduce the iteration budget, never increase it beyond the Charter maximum. A misconfigured or malicious surface integration cannot run an unbounded loop. This is the outer iteration budget — it governs full Perceive→Reason→Act→Observe cycles. A separate reasoning budget governs token consumption within a single Reason stage invocation — see Reasoning budget below.
- **Cost controls:** Set a maximum token spend per loop run. When the budget approaches, the agent should summarize rather than continue retrieving, and exit gracefully when the limit is reached.
- **Timeout budgets:** Define per-surface timeout thresholds in the capability flags injected during Perceive. A voice surface might enforce a 3-second hard timeout; an async pipeline might allow 60 seconds per iteration. As with iteration budgets, Charter-level timeouts are the ceiling; surface flags may only tighten them. When iteration budget and timeout budget conflict mid-loop, the more restrictive condition wins — the agent exits and surfaces its current state.
- **Reasoning budget:** Define a maximum token budget for the Reason stage. Because the agent may draw from multiple reasoning patterns in a single invocation — and no single strategy's unit (steps, cycles, plan steps) applies universally — tokens are the consistent measure across all approaches. This budget is separate from and in addition to the outer iteration budget. An outer budget of 5 iterations with no reasoning budget can still produce unbounded cost per iteration. The budget is defined in the Charter, follows the same ceiling rule (surfaces may only reduce it), and applies per Reason stage invocation. When exhausted, the agent exits with its best available answer — it does not loop back to Perceive. Tokens consumed must be reported in the Reason stage output contract so the outer Observe stage can track cumulative cost accurately.
- **Reasoning trace policy:** The reasoning trace — the internal record of how the Reason stage reached its answer — is the primary artifact for debugging and behavioral diffs between Charter versions. Its structure varies by strategy: a Chain of Thought trace is a linear reasoning sequence; a ReAct trace is a sequence of Thought-Action-Observation

records; a Reflection trace includes draft, critique, and revision rounds. The retention policy applies regardless of strategy. Whether the trace is logged persistently, retained ephemerally, or discarded is a design decision with compliance and cost implications. Regulated environments typically require persistent logging. Cost-sensitive deployments may discard after session end. The policy must be defined explicitly in the Charter alongside the reasoning budget — it cannot be left as an infrastructure default.

- **Loop tracing:** Every iteration of every loop should produce a trace entry — what stage the agent was in, what it decided, what tools it called, and what it observed. Traces are the primary debugging tool for production agent failures.
- **Escalation on repeated failure:** If the loop has broken on the same condition more than a defined number of times in a session, escalate to a human rather than retrying. Repeated breaks on the same condition are a signal the agent cannot resolve the problem autonomously.

Agent Surfaces

The Agent defines what the agent is. The Agentic Loop defines what it does. Agent Surfaces define where it does it — the channels, applications, and interfaces through which users and systems interact with the agent in the world.

Surfaces are external to the agent. The same Agent Intelligence, the same Agent Charter, and the same Agentic Loop operate across every surface. What changes per surface is not the agent — it is how the surface calls the agent, how much of the loop it allows to run, and how it receives and presents the agent's output.

This is the build-once principle in practice. A new surface does not require a new agent. It requires a surface integration — the Perceive and Act stages configured for that surface's specific protocol, format, and constraints — and a set of capability flags that tell the Agent Charter how to express itself in that context.

Agent Surface	Integration Method	Common Use Cases
Web Chat Widget	JavaScript SDK / iframe	Customer support, onboarding
Slack / Teams Bot	Events API + OAuth	Internal productivity, IT helpdesk
REST API	HTTP JSON endpoints	Mobile apps, third-party integrations
Email Responder	SMTP/IMAP or SendGrid	Support inbox, lead qualification
Voice (Phone)	Twilio / Deepgram	Call centers, scheduling bots
Browser Extension	Chrome Extension API	Research assistants, writing aids
CLI / Terminal	Node/Python CLI	Developer tooling, automation scripts
Automation Pipeline	Zapier / n8n / cron	Scheduled reports, data processing

Surface Characteristics

Every surface has a different tolerance for latency, a different expectation of response format, and a different relationship between the user and the agent. Understanding these differences is what drives the capability flag configuration that flows into the Agentic Loop at the Perceive stage.

- **Synchronous surfaces (Web Chat, Slack, Voice):** Require real-time responses, typically under 3 seconds. Streaming responses and lightweight model configurations are essential. Capability flags enforce no-markdown and token limits where required.
- **Asynchronous surfaces (Email, Automation Pipelines):** Tolerate longer processing times (10–60+ seconds). Enable richer reasoning, longer context retrieval, and more tool calls per request. The agent can run deeper loops here than it can on synchronous surfaces.
- **Programmatic surfaces (REST API, CLI):** Called by other software rather than humans directly. Require strict input/output contracts, versioned endpoints, and machine-readable error responses. The agent's output must be structured and predictable.
- **Scheduled surfaces (Automation Pipelines, Cron):** Agent-initiated rather than user-initiated. The Perceive stage receives a trigger rather than a user message — but all other loop stages run identically. The agent is equally capable; it just starts from a schedule rather than a conversation.

The Standard Message Contract

The Standard Message Contract is the formal interface between every Agent Surface and the Agentic Loop. It is the data schema that every surface integration must produce before the Perceive stage can begin. It normalizes the enormous variety of surface-specific payload formats into a single structure the Agent Charter understands.

- user_id — unique identifier for the user, resolved from the surface's auth mechanism
- session_id — conversation thread identifier, used to retrieve prior context from memory
- message — the user's input, normalized to plain text regardless of source format
- surface — the Agent Surface this message arrived from, used for routing and logging
- active_skills — removed in v0.08. Skill activation is entirely agent-internal. Surfaces have no awareness of skills. The agent reads the available skill library during reasoning and activates what the task requires. No surface field is needed or appropriate
- capability_flags — per-surface formatting and behavior constraints (no_markdown, max_tokens, etc.) Treat capability_flags as a versioned, typed schema with a defined registry — not a free-form metadata field. Surface flags may only reduce Charter-defined limits (e.g., lower the iteration ceiling); they cannot exceed them. Unknown flags must fail loudly, not silently.
- metadata — optional surface-specific context (channel name, user timezone, locale, etc.)

Why the Contract Matters

The Standard Message Contract is what makes the build-once architecture possible. As long as every surface integration produces a valid contract, the Agent Charter never needs

to know what surface it is talking to. Surfaces are interchangeable. The agent is not.

Surface Configuration and Loop Depth

One of the most important surface decisions is how deeply the Agentic Loop is allowed to run. Not every surface benefits from — or can support — a full multi-iteration loop. The capability flags injected at Perceive are the mechanism for controlling this.

- **Loop depth:** Set the maximum iteration count per surface. Voice and live chat: 1–3 iterations. Slack and web chat: 3–10. Async pipelines and research surfaces: 10–30+.
- **Tool access per surface:** Not every tool should be available on every surface. A voice surface probably should not be invoking database write tools. Capability flags can restrict tool access to what is appropriate for the context.
- **Model selection per surface:** Lighter, faster models for synchronous surfaces. Frontier models for complex async tasks. The Agent Charter defines the routing policy (e.g., “use lightweight model when latency_sensitive: true”); the surface provides the flag. Model selection belongs to the Charter, not the surface — so that upgrading or swapping models requires a Charter change, not edits across every surface integration.
- **Output format per surface:** SSML for voice. Markdown for Slack. HTML for email. Plain text for SMS. The Agent Charter generates content; the surface configuration governs how it is formatted and delivered.

Multi-Surface Consistency

Because all surfaces share the same Agent Charter, the agent's knowledge, identity, and decisions are consistent everywhere. A user who contacts support via chat and follows up by email encounters the same agent — with the same values, the same knowledge, and continuity of context. The surface changes. The agent does not.

Key Benefits

Building a single agent and deploying it across multiple surfaces delivers significant advantages over building surface-specific agents:

Single Source of Truth	Update the Agent Charter once — all surfaces reflect changes instantly.
Consistent Behavior	Same identity, same logic, same guardrails across every surface.
Faster Iteration	Test changes in one environment; deploy to all surfaces with confidence.

Cost Efficiency	One Agent Intelligence, one Agent Charter, one maintenance cycle.
Scalability	Add new Agent Surfaces without rebuilding the agent from scratch.

Authentication

Authentication in a multi-surface agent is not a single problem — it is three distinct layers, each with a different owner and a different mechanism. Conflating them is the most common source of security failures in production agent deployments.

Layer 1: Surface-to-Agent (Inbound)

This layer verifies who the user is before a message reaches the Agent Charter. Each surface has its own mechanism — Slack uses OAuth tokens, web chat uses session cookies or JWTs, voice uses webhook signatures, REST API uses bearer tokens or API keys. The surface adapter (Perceive stage) is responsible for validating these credentials and resolving them to a verified user identity. The Agent Charter must never see a raw token. It receives only a resolved user_id and verified context. If the adapter cannot resolve identity, the request does not reach the loop.

Layer 2: Agent-to-Tools (Outbound)

When the agent calls a tool, it needs credentials for that tool. This is the most complex authentication layer and the one most teams get wrong. There are two distinct patterns, and the choice between them is a design decision made per tool in the Charter — not an implementation afterthought.

Service-scoped credentials — the agent acts as itself, using a service account with fixed credentials. Appropriate for tools where user identity does not affect access or results: looking up product data, checking inventory, querying a shared knowledge base. Credentials are stored in the environment and injected at tool registration time. They never appear in the Agent Charter or the loop trace.

User-scoped credentials — the agent acts on behalf of the user, using the user's own OAuth token for the tool. Required whenever the tool result or permission depends on who is asking: reading a user's calendar, sending email as the user, accessing files the user owns. These tokens are not stored in the Charter. They are resolved by the surface adapter during Perceive — the adapter completes any necessary OAuth flow, obtains a scoped token for the user, and injects it into the session context via the Standard Message Contract. The Agent Charter accesses it there when calling the tool. The Charter never initiates an OAuth flow and never stores user tokens.

Layer 3: Agent Identity

When multiple agents exist — or when agents call other agents — the agents themselves need verifiable identities so they can authenticate to shared services and to each other. This is an emerging area addressed by A2A (Agent-to-Agent) protocols, where agents publish signed capability descriptions and authenticate inter-agent calls via standardized mechanisms. The

design principle is the same as Layer 1: identity is resolved before any capability is exercised, and the agent receiving a request never acts on an unverified caller. This layer is documented here as a design concern; its full specification belongs in a future document in this series covering multi-agent networks. See the series index for the current scope of that work.

Standard Message Contract — Authentication Fields

The Standard Message Contract carries the output of Layer 1 and Layer 2 resolution into the loop. The existing user_id field carries the resolved identity from Layer 1. The contract's metadata field should be extended to carry user-scoped credential tokens resolved by the surface adapter for Layer 2 — keyed by tool name, scoped to the session, and never logged in plaintext. The Agent Charter reads these from session context when invoking user-scoped tools. This keeps credential handling at the surface boundary where it belongs, and keeps the Charter stateless with respect to auth.

Agent Management

A deployed agent requires four distinct management surfaces, each with a different owner, a different cadence, and a different kind of change it is responsible for. Conflating them — treating the Charter as a config file, or using the gateway to patch behavior — produces agents that are hard to reason about and impossible to audit.

The Charter as Behavioral Source of Truth

The Agent Charter is the single artifact that defines what the agent is and how it behaves. Every behavioral change — refined identity, new skill, updated guardrail, new tool registration — goes through the Charter. It should be version-controlled like code: reviewed, tagged, and deployed through a defined process. Editing the Charter in place without versioning is the equivalent of deploying untested code directly to production.

Charter versioning must produce behavioral diffs — a description of how the agent will behave differently after the change, not merely which configuration fields changed. A config diff tells you that the system prompt was edited; a behavioral diff tells you that the agent will now decline requests it previously accepted, or will call a new tool in circumstances it previously handled with a static response. This distinction is critical for review, for regression testing, and for audit trails in regulated environments. It is a gap in every current agent framework and represents a first-class design requirement, not an implementation nice-to-have.

The API Gateway as Operational Control Plane

The API gateway sits between surface adapters and the agent core. It is the operational management layer — responsible for what the agent does in production without touching what the agent is. Its concerns are: rate limits and quota enforcement per surface, cost caps and token budget monitoring, traffic routing between Charter versions for A/B testing or staged rollouts, kill switches and circuit breakers for surfaces or the agent as a whole, and request logging for audit and compliance. When something goes wrong in production, the gateway is where you intervene. Changes here do not alter the agent's behavior — they alter when and how it is reached.

The Observability Platform as Feedback Loop

Every loop iteration across every surface feeds the observability platform. This is not a monitoring concern — it is how the Charter evolves. Because the Charter is a base that grows through use rather than a complete upfront specification, the observability platform is the mechanism by which real usage informs what needs to be added. What the agent fails at reveals missing tools. Where it goes off-piste reveals underspecified identity. Where users drop off reveals latency or format mismatches. Charter updates in production should be driven by what the observability platform shows, not by speculation. This closes the loop between design, deployment, and iteration.

The Skill Registry as Capability Lifecycle

Skills require their own management surface — a versioned registry that handles the full capability lifecycle. The registry is responsible for four things: publishing and versioning new skill releases, deprecation routing (redirecting requests for an old skill version to its successor, with a defined grace period), resolution failure handling (what happens when a session starts with an active_skills reference that no longer exists), and cross-agent dependency tracking (which agents are pinned to which skill versions). The registry is effectively a package manager for agent capability. It must be designed explicitly, with governance policies for who can publish, what a release requires, and how deprecation is communicated to dependent agents. It does not emerge automatically from the architecture.

Considerations & Trade-offs

Surface-Specific Limitations

Not every capability in the Agent Charter will be appropriate for every surface. Voice surfaces cannot render markdown or tables. Email surfaces may need HTML formatting. The capability_flags field in the Standard Message Contract is how the Perceive stage communicates these constraints to the Agent Charter, allowing it to tailor its output format per surface without changing its core behavior.

Latency Requirements

Different surfaces have different latency tolerances. Voice and live chat require sub-3-second responses. Email and async pipelines can tolerate 10–60 seconds. Context window size directly affects latency — larger context means more tokens to process. Consider lighter skill configurations and smaller Agent Intelligence models for latency-sensitive surfaces, while reserving full capability for asynchronous surfaces where quality matters more than speed.

Authentication & Identity

Each surface may have a different authentication mechanism. The Perceive stage is responsible for resolving user identity and injecting verified user context into the Standard Message Contract before the Reason stage begins. The Agent Charter should never handle raw authentication tokens directly.

Cost Management

A single Agent Charter called from many surfaces can rapidly multiply Agent Intelligence API costs. Implement caching for common queries, optimize KV-cache usage by structuring prompts with stable content (Identity & Soul) first, set per-surface token limits, and monitor usage by

surface through the API gateway. Memory retrieval costs (vector DB queries) should also be tracked alongside model API costs.

Summary

The Position

Most organizations are building too many agents and not enough of the right one. They build a chatbot for the website, a bot for Slack, a different automation for email — each with its own identity, its own logic, its own maintenance burden. They call all of it "AI" and wonder why it doesn't feel coherent.

The alternative is to build one agent — built on a strong base and designed to grow — and let every surface call it. The Agent Intelligence provides the reasoning. The Agent Charter provides the identity, the knowledge, the skills, and the rules — a base that grows. The Agentic Loop drives the behavior. The surfaces are just where the agent shows up.

Influences & References

This document draws on a number of external ideas and sources. The following acknowledges those influences directly. Where a specific design decision in this document traces to a specific source, it is named. Where the influence is broader — a tradition or body of thinking rather than a single work — it is described briefly.

Direct Influences

OpenClaw / SOUL.md — Carl Vellotti (learnopenclaw.com). The Identity & Soul component, the principle that specificity in persona produces dramatically better agent behavior than vague instructions, and the framing of character as the primary safety mechanism all trace directly to OpenClaw's SOUL.md design. The observation that "the alpha comes from the systems around the model" is the same claim as "the Charter is the floor, not the ceiling."

ReAct: Synergizing Reasoning and Acting in Language Models — Yao et al., 2022 (Google / Princeton). The Thought–Action–Observation cycle, the interleaving of reasoning and tool use, and the inner loop structure of the Reason stage.

Constitutional AI: Harmlessness from AI Feedback — Bai et al., 2022 (Anthropic). The principle that a well-defined identity generalizes to situations no rule anticipated, and that guardrails work better as principles than as exhaustive lists.

Model Context Protocol (MCP) — Anthropic, 2024. Tool discovery over pre-registration. The trusted-sources model in this document — where the Charter defines which sources the agent is authorized to discover tools from rather than pre-registering every tool — is a direct application of MCP's design philosophy.

Chain-of-Thought Prompting Elicits Reasoning in Large Language Models — Wei et al., 2022 (Google). The Chain of Thought reasoning pattern and its role as one mode among many in the reasoning library.

Reflexion: Language Agents with Verbal Reinforcement Learning — Shinn et al., 2023. The Reflection pattern — draft, critique, revise — and its position as a distinct reasoning approach suited to quality-sensitive tasks.

LLM Powered Autonomous Agents — Lilian Weng, 2023 (Lil'Log). The most comprehensive survey of the agent design space at the time this document was written. Informed the memory taxonomy, the planning pattern landscape, and the framing of agents as systems rather than prompts.

Framework Influences

The following frameworks shaped the architecture described here — either as direct influences or as instructive contrasts. LangGraph (LangChain) for its state machine and loop model. Semantic Kernel (Microsoft) for its plugin and skill architecture. OpenAI Assistants API for its session and thread management model. CrewAI for its identity-heavy, role-first agent design. AutoGPT for the instructive failure case: what happens when an agent has no stable base, no consistent identity, and almost everything is runtime.

Background Lineage

The memory type taxonomy — working, episodic, semantic, procedural, persistent — draws on cognitive architecture research, particularly Anderson et al.'s ACT-R and Laird et al.'s SOAR. The adapter pattern and surface separation draw on microservices architecture patterns. The Charter-as-base principle has intellectual ancestors in the Unix philosophy of small composable tools and in Domain-Driven Design's bounded context model, though neither is cited directly.