VANTIQ Developers Guide Series – Designing Vantiq Applications

Version 2.0

30/03/2026

Table of Contents

List of Figures

1. Introduction

Designing real-time, event-driven, and situationally-aware applications in Vantiq begins with a clear understanding of domain requirements and culminates in the creation of a robust application design model. While the Vantiq Developers Guide – Introduction to Vantiq Development provides a comprehensive overview of Vantiq's event-driven architecture, service taxonomy, and development workflows, this document builds upon that foundation to focus specifically on the design phase of the application lifecycle.

In Vantiq, application design is not a linear or siloed activity—it is an integrated, iterative process that connects business needs to technical architecture. This process involves translating domain events, reactions, and commands (typically captured using the System Modeler) into a modular, loosely coupled system of services, clients, and external integrations via the Design Modeler.

Although a top-down approach is advocated, it is not mandated; it is possible to skip requirements and start modeling your application directly in the Design Model or even start developing your services and clients and then utilize the Design Model purely to connect your loosely coupled services and visualize your application.

This document will guide readers through the steps necessary to:

- Elicit and structure application requirements using simplified Event Storming techniques;
- Translate these requirements into a visual application design;
- Define event flows, service responsibilities, and interface boundaries;
- Prepare for implementation using Vantiq's model-driven tools.

Whether you begin from a clean slate or with an existing set of service components, this guide aims to help you structure your application design for scalability, adaptability, and clarity. The goal is not just to design for functionality, but to enable agility and intelligence at the architectural level, ensuring your Vantiq applications are as responsive and collaborative as the real-world environments they serve.

2. Requirements Definition

To build a real-time, event-driven collaborative application, it is essential to begin with a clear understanding of the system's **business-driven requirements**. These requirements are best captured using **Event Storming**, a lightweight, visual, and highly collaborative modelling technique that emphasizes business events and reactions.

Vantiq supports this process through its **System Modeler**, which enables stakeholders to capture and share a simplified Event Storming model. The goal is to collaboratively identify **what happens** in the domain (events), **how the system should react** (actions and commands), and **what the boundaries of functionality are** (stories). This becomes the foundation for system design without prematurely introducing implementation details.

2.1 What is Event Storming?

Event Storming is a **domain-driven discovery method** used to explore complex business processes and workflows through the lens of *events*—meaningful changes in state. Originally introduced by Alberto Brandolini (https://leanpub.com/introducing_eventstorming or https://www.oreilly.com/library/view/learning-domain-driven-design/9781098100124/), this approach encourages real-time collaboration between business experts, architects, and developers, using sticky notes or visual canvases to map out the system behaviour.

Key characteristics:

- **Collaborative**: Involves both technical and non-technical stakeholders.
- **Exploratory**: Reveals hidden logic, dependencies, and domain constraints.
- **Visual and Fast**: Quickly maps out end-to-end business flows.

2.2 Event Storming Concepts in the Vantiq System Modeler

In Vantiq, Event Storming is applied in a simplified but effective manner using four core concepts:

- **Events**
  Represent meaningful *business state changes* or occurrences. Events are immutable and describe facts about the system, e.g., TemperatureChanged, PumpFailed, or CustomerUpdated.
  Events are the **starting points** in the system's behaviour—they trigger all subsequent processing.

- **Reactions**
  Describe what should happen **in response to events**. An event may trigger one or more actions, which represent business logic or workflows. For example, a PumpFailed event might trigger actions such as NotifyTechnician and OpenRepairTicket.
  Actions define **domain reactions** and form the bridge between the event and the intended business behavior.

- **Commands**
  Represent explicit **requests to perform an operation**. They often result from actions and lead to future events. For example, DispatchTechnician might be a command issued in response to a PumpFailureIdentified action.
  Commands are usually **imperative**: they ask the system to *do something*.

- **Stories**
  A **story** is a collection of events, actions, and commands that describe a complete slice of business functionality or behaviour. Think of stories as *scenarios* or *use cases* that define what the system should do under specific circumstances.
  For example, a story might describe how the system detects a failing pump and coordinates human and system-based responses to mitigate the issue.
- **Notes**

  A **Note** is additional documentation that you can attach to the model or the events, reactions, and commands. Notes might be used to document business rules etc.

2.3 Modelling Guidelines

The purpose of the Event Storming model is to capture only what is necessary to describe **business behaviour**, without introducing implementation details. It should:

- Be understandable to both business and technical stakeholders.
- Focus on **real-world occurrences and required responses**.
- Avoid modelling design artifacts such as specific data structures, service endpoints, or user interfaces.
- Postpone decisions about system decomposition, routing, and storage to the **design phase**.

2.4 Example: Requirements Model for Predictive Pump Failure

The following example illustrates an Event Storming model representing a real-time collaborative scenario:

- **Events**:
  - TemperatureChanged
  - RPMChanged
- **Actions**:
  - AnalyzeSensorData — assess temperature and RPM patterns to detect anomalies.
  - DetectPumpFailure — determine whether failure conditions exist.
  - CollaborateWithFieldTechnician — notify and engage the appropriate human actor.

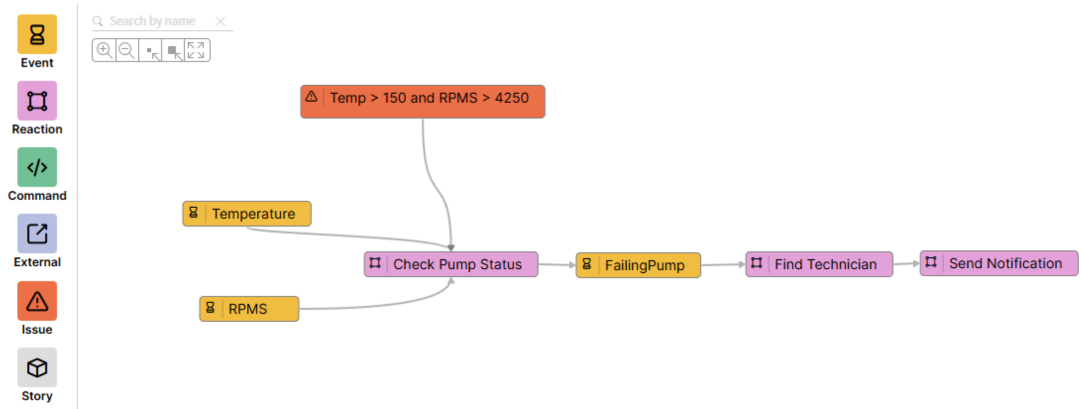

Figure 1 - Pump Requirements Model

This model identifies the key business scenario: detecting potential pump failures in real-time and coordinating resolution through human-in-the-loop collaboration.

By capturing **just enough** to express business behaviour, the Event Storming requirements model becomes the ideal bridge between stakeholder conversations and the more technical application design built in the **Design Modeler** and **Service Builder**.

3. Designing an Application

Designing a Vantiq application is a flexible, iterative process that can begin from multiple starting points depending on your project needs and team preferences. Some teams may begin with a formal System Model, using Event Storming to clarify business behaviour and identify system boundaries. Others may prefer to dive directly into designing services and clients using the Design Modeler and Service Builder. Both approaches are valid, and Vantiq supports this flexibility by enabling design to evolve as understanding deepens.

What matters most is how you separate responsibilities across the three fundamental artifacts: **services**, **clients, and external systems**.

3.1 Services, Clients & External Systems: Core Building Blocks of Vantiq Applications

At the heart of every Vantiq application is an interaction between services — event-driven logic components or AI Agents, clients — purpose-built interfaces for human interaction — and external systems. From a design perspective, external systems are essential as they define the connections between the Vantiq system and the external resources that can integrate into the application. External systems do not manifest as implementation artifacts in their own right but are implemented as services. These three artifacts serve different roles:

- **Services** encapsulate real-time processing, business logic, integration, and AI Agents. They are modular and loosely coupled, designed to respond to events and coordinate system behaviour.
- **Clients** are user-facing applications that support decision-making, task execution, and oversight. They connect directly to services to retrieve data, initiate workflows, and display real-time events.
- **External Systems** represent the connections between the Vantiq application and the outside world. These could be sources of events, external applications, or services that require integration. They will be implemented as services, loosely termed as integration services.

Whether you begin with a formal system model or prototype directly in the Design Modeler, structuring your application using clear, focused services and role-aligned clients is key to achieving a scalable, maintainable architecture.

3.2 Model-Driven or Design-First—Both Paths Converge

If you're starting from a System Model, the events, actions, and commands you've identified become valuable guides for shaping services and their interactions. The Design Modeler enables you to visually carry this structure forward, ensuring continuity from concept to implementation.

Alternatively, if you begin directly in the design phase—defining services and clients from the outset—the same principles apply. Each service should own a focused area of functionality, and each client should support a defined user role or task. As the design evolves, you can use modelling tools to reinforce clarity, reduce coupling, and validate behaviour.

Vantiq's tools are designed to support both top-down and bottom-up design approaches—allowing you to begin wherever your understanding is clearest and iterate toward a robust, event-driven architecture.

4. Designing Vantiq Services

In Vantiq, services are meant to be modular, adaptable, and independently deployable. Two key design principles help ensure your components are well-structured:

- Manageable Dependencies – Keeping components from becoming too reliant on each other.
- Clear Functional Focus – Ensuring each component does one specific job well.

These two design principles must also be weighed against a single overarching principle when designing systems: have only as many services and clients as needed, so not too many, and keep the system simple.

4.1 Managing Dependencies Between Services

When services rely too heavily on each other, they become harder to change, scale, or test independently. Vantiq enables you to define relationships between services using asynchronous events and declarative routing, minimizing direct dependency.

Here are common types of relationships, ranked from most flexible to most restrictive:

- **Independent Event Interaction – Routes (Best Practice)**
  Services should define inbound and outbound events, use routes to pass events between services, and avoid tightly bound interactions, such as a direct service-to-service publish-and-subscribe approach. Routes are defined within the Design Model, see diagram below. As routes are defined at the Design Model level, they can be changed without affecting the underlying services, and services can be imported from catalogs and incorporated into an application without change, and additional services can be added to an application without changing other services.

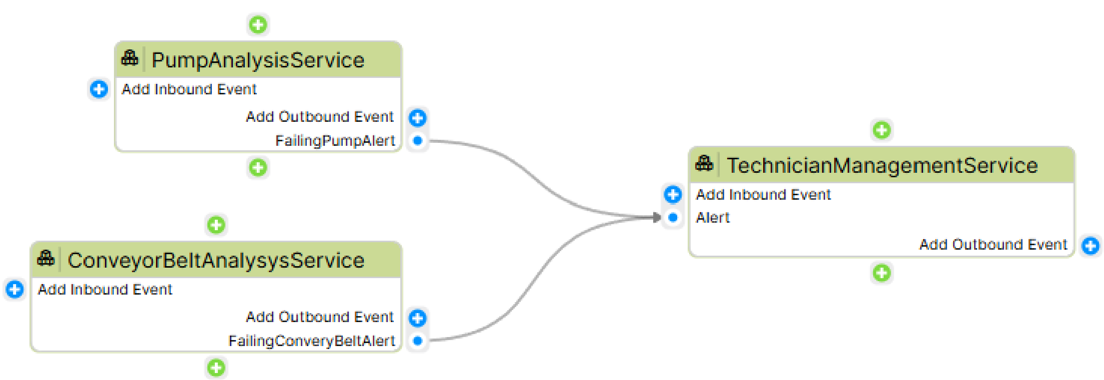

Figure 2 - Independent Event Interaction

In this example, the two analysis services publish outbound events that are both routed to the TechnicianManagementService. Additional analysis services could be added without changing any code.

Best Practice: Services are loosely coupled, and events are routed between services rather than tightly coupled via direct service to service invocations

- **Direct Event Interaction (Tightly Coupled)**
  One service publishes an event on another service. This creates a tightly bound relationship between the services. This limits both reuse and deployment options.

  Best Practice: Use this sparingly, only when tight coupling is necessary to break out functionality for good functional design.

- **Direct Invocation of Procedures (Tightly Coupled)**
  One service calls another using synchronous procedures. This creates a stronger runtime link and requires awareness of the target service's interface. Use this sparingly, only when real-time confirmation or a return value is essential.

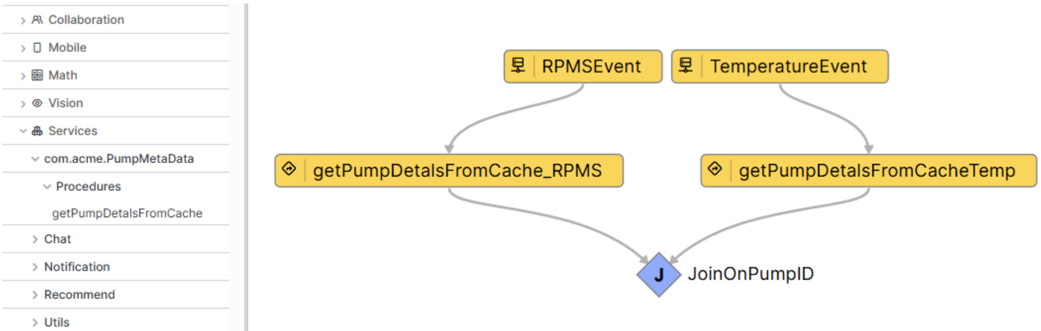

Figure 3 - Good Direct Invocation Example

This example is a Visual Event Handler within the PumpAnalysisService that is making direct invocations on the PumpDetailCache. This example shows tight coupling between these two services, but in this case, it has been done for good functional reasons; the PumpDetail service is managing pump metadata.

Best Practice: Use this when distributing the logic between two different services, making functional sense so as to avoid having a large service that becomes complex to manage and maintain.

- **Oversharing of Data Structures (Fragile Design)**
  Large, complex event payloads or data types are passed between services, even when each service only needs part of the data. This can make services fragile and performance-intensive. Instead, design lean, purpose-specific event types.

  Best Practice: *Remove unused event data as close to the source of the event as possible, structure your event such that sub-events can be easily passed to downstream services without the need to pass large events around.*

- **External System Reliance (Should Be Isolated)**
  Integration, whether inbound or outbound integration, is almost inevitable within a Vantiq application. Having the integration with external systems spread across multiple series of implementations where it is to be consumed leads to a fragile and complex application to maintain, especially as external systems are often being changed externally.

  Best Practice: Some services integrate with external APIs or platforms. These dependencies should be isolated in dedicated Integration Services that handle all interactions with third-party systems, so internal services remain unaffected by changes.

- **Shared Persistent State (Avoid if Possible)**
  When multiple services directly manipulate the same data storage (e.g., a shared table), changes in one place can break others.

  Best Practice: It's better to centralize data access in Data Services, where changes can be controlled and versioned more easily.

Favor *loose, event-based interactions* that maintain service independence. Avoid shared state, bloated payloads, or tight coupling unless absolutely justified by functional necessity.

4.2 Ensuring Focused Service Responsibilities

A well-structured service is focused: all of its logic supports a specific area of the application. This improves clarity, maintainability, and reuse.

Here are common levels of functional clarity ranked from best to worst in service design:

- **Focused Responsibility (Best Practice)**
  The service's event handlers and procedures work together to support a clearly defined purpose.

  Event Driven Focus: Event-driven systems should be designed so that event handlers consume related domain events and apply actions that support a single, well-defined business purpose. Handlers should exist to answer a business question or drive an operational decision, not simply to react to every event available.

  For example, in a manufacturing domain, a service responsible for monitoring the health of production equipment may consume both temperature and pressure events emitted by the same piece of equipment using one or more sensors. These events are evaluated together to determine whether the equipment is operating within acceptable limits, whether conditions are deteriorating, or whether intervention is required. Based on this evaluation, the service can trigger actions such as raising an alert, escalating an incident, or initiating maintenance, and then publish the following outbound events OperatingOutOfSpec or MaintenanceRequired.

  When multiple event types are consumed by the same handler, they should be related in a clear domain sense—describing the same asset, lifecycle stage, or operational concern. This focus keeps event handling logic cohesive, easier to evolve, and aligned with how the business understands and manages its systems.

  Domain-driven Focus: A common and effective way to design a focused service is to align it with ownership of a single business capability or area of responsibility. Rather than centering services on technical concerns, each service should represent something the business clearly understands and cares about.

  For example, a CustomerOnboardingService might own the complete onboarding experience for new customers. This includes collecting required information, verifying eligibility, tracking onboarding progress, enforcing business rules, triggering welcome communications, and answering questions about a customer's onboarding status. Every responsibility within the service exists to support this single business outcome: successfully onboarding a customer.

This kind of alignment ensures high cohesion and makes the service easier to test, scale, and evolve independently, because changes in onboarding policy, regulations, or customer experience are localized to one place.

Best Practice: Define services around a single, coherent business capability. Ensure that every operation, event, and data model supports the service’s central purpose. When a service mirrors how the business thinks about its work, it becomes easier to understand, adapt, and reuse across applications.

- **Shared Input, Diverging Logic (Caution)**
  A service may process the same incoming event in different ways, depending on the business context. This pattern is acceptable when the logic is related but should be reviewed to ensure the service remains understandable.

  *Best Practice**:** Use clear event type distinctions or context-based routing to manage divergent logic. Periodically refactor or split services if the branching logic grows complex or starts to obscure the core responsibility*

- **Unrelated Tasks in One Service (Avoid)**
  A service implements multiple unrelated responsibilities (e.g., email processing and inventory checks). This makes testing, reuse, and updates much harder. Separate into distinct services with clear names and responsibilities.

  Best Practice: Do not group logic based on developer ownership or convenience. Always separate unrelated concerns into their own additional services to enable better maintainability, testability, and deployment agility.

Strive for focused responsibility, exercise caution with loosely related logic, and avoid mixing unrelated concerns within the same service.

5. Designing AI Agents

5.1 Agent Definition

Agents are goal-oriented components within an application. They are typically invoked through textual requests aligned with their stated goals and use autonomous or semi-autonomous reasoning to achieve those goals based on the information provided.

Agents may also exhibit learning behaviour by capturing signals such as reasoning traces, execution outcomes, or feedback. These signals are used to adapt future behaviour, often through the integration of short- and long-term memory mechanisms.

Agents generally implement reasoning capabilities by delegating reasoning tasks to one or more invocations of a generative AI foundation model.

Within a Vantiq Agentic application, each agent is implemented as a Service and therefore inherits all service capabilities. An agent may be request/response driven, event-driven, or both. Agents may also use the persistent, fault-tolerant state of a Vantiq Service—such as Collaborations or semantic indexes—to retain context and support learning behaviours.

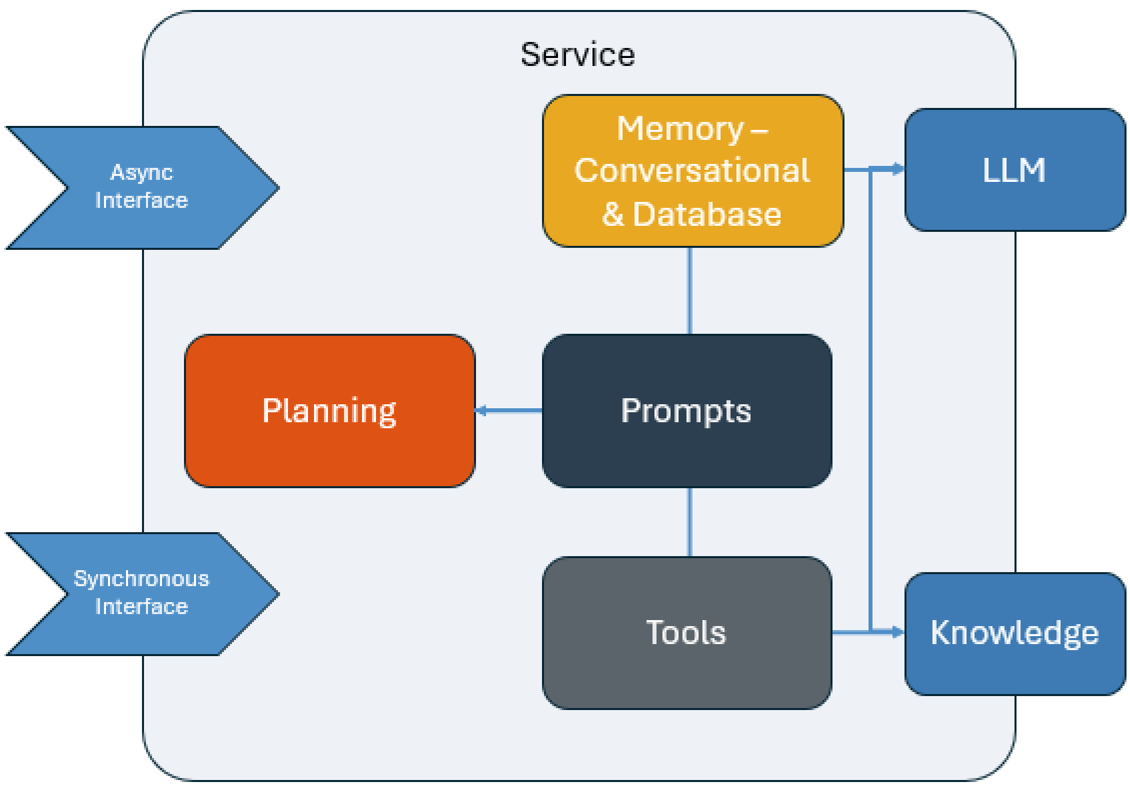

Figure 4 – Vantiq AI Agent

5.2 Agent Capabilities

Agents can implement a variety of capabilities depending on the problems they are intended to solve. Agents can implement a variety of capabilities depending on the problems they are intended to solve. Real-world agents often combine several capabilities such as planning, orchestration, execution, and reasoning. Some agents focus on reasoning about tasks, others on coordinating work, and others on executing concrete actions or tool calls. The capabilities described in this section are not mutually exclusive; real-world agentic systems often combine several of them.

These capabilities provide a mental model for understanding how responsibility, control flow, and intelligence are distributed within an agent-based application. By distinguishing between planning, orchestration, execution, and reasoning capabilities, architects can design systems that are more modular, scalable, and easier to evolve over time.

5.2.1 Reasoning Capabilities

5.2.1.1 Planning

Planning focuses on determining a sequence of steps required to achieve a goal. It reasons about actions, dependencies, constraints, and potential outcomes, often producing an explicit plan or strategy. Agents implementing planning capabilities are primarily concerned with task logic and goal achievement.

Planning capabilities answer questions like:

- What steps are needed to solve this problem?
- In what order should actions occur?
- What assumptions must hold for this to work?

Agents implementing planning capabilities may use one or more generative AI–based planning approaches to construct plans involving skills within a single agent or across multiple agents in a multi-agent system. The following are common planning approaches, though the list is not exhaustive.

- **Plan-Execute**: Plan–Execute generates a plan upfront and then executes it step by step while maintaining explicit control flow. Each step represents an action that may succeed, fail, or emit signals that can be observed. This enables monitoring of progress and allows execution to stop, retry, or escalate when issues occur. The approach provides structured and predictable execution but assumes relatively stable conditions.
- **Plan-Execute-Replan**: Plan–Execute–Replan extends Plan–Execute by monitoring execution and triggering replanning when failures, unexpected outcomes, or new information arise. 'Reflection' is commonly used to evaluate intermediate results and detect mismatches between expected and actual outcomes. The agent commits to a plan but remains willing to revise it as conditions change. This approach balances robustness with efficiency at the cost of increased complexity.
- **ReWOO (Reason Without Observation)**: ReWOO performs all reasoning and planning upfront without relying on intermediate observations during execution. The agent assumes that actions will succeed as planned and does not validate outcomes until completion. This makes the approach fast and cost-effective when tool calls or environment interactions are expensive. However, it is risky in dynamic or uncertain environments.
- **Tree of Thought (ToT)**: Tree of Thought expands reasoning into a search tree of intermediate thought states. At each stage, multiple possible next steps are generated, evaluated, and pruned. This enables deeper lookahead and backtracking compared to linear reasoning. The approach can improve solution quality on complex problems but incurs higher computational cost.

There are several other planning approaches and variations of these planning approaches, which we won't go into here, but it is worth stating that these are not an exhaustive list, and new approaches and implementations are being added regularly.

5.2.1.2 ReAct Reasoning Capability

Agents implementing the ReAct reasoning capability follow a tight reasoning-and-action loop, where the agent alternates between thinking about the current state and taking an action, often via tool calls. Rather than producing a complete plan upfront, the agent incrementally decides what to do next based on the most recent observations or results. This makes agents that implement the ReAct reasoning capabilities effective in dynamic or partially observable environments where assumptions frequently change.

ReAct is not an orchestration pattern and does not coordinate multiple agents. It describes the internal behaviour of a single agent, not how work is delegated, sequenced, or supervised across agents. In multi-agent systems, ReAct is commonly used within worker or planner agents that are themselves managed by higher-level orchestration logic.

5.2.2 Control Capabilities

5.2.2.1 Orchestration

Orchestration coordinates execution across agents in a multi-agent system or across tools in a single-agent system. Rather than reasoning deeply about domain tasks, it manages control flow, delegation, sequencing, and integration. Orchestration focuses on controlling how work is carried out rather than solving the domain problem itself. Its output consists of execution decisions—for example, selecting which agent or tool should perform a task, determining the order of steps, passing inputs between components, and deciding how to handle failures, retries, or low-quality results. These decisions define the flow of execution but do not produce the final business or domain outcome. In contrast, domain-level solutions are the concrete results generated by worker or execution capabilities, such as writing code, analysing data, generating reports, or answering user questions. The orchestrator acts as a coordinator that directs “who does what and when,” while leaving the actual problem-solving to specialised agents or tools

Agents implementing orchestration capabilities answer questions such as:

- Based on this user request, should I send it to the code generation agent, the documentation agent, or the debugging agent?
- Now that this step is complete, should I pass the output to the next stage in the pipeline or handle an error before continuing?
- What do we do if it fails or produces low-quality output?

Orchestration may be implemented using generative AI, semantic search, rules, or combinations thereof. The following are common orchestration approaches.

- **Router-Based Orchestration**: A router analyses the user request or current state and selects the most appropriate skill, or agent, to handle it. The routing decision is typically made using intent classification, rules, or an LLM-based scorer. Only the selected component runs, keeping execution efficient. This works well when tasks are clearly separable.
- **Hierarchical Orchestration**: A high-level “manager” agent decomposes the task into sub-tasks and assigns them to specialized workers. Each worker focuses on a narrow

responsibility, such as coding, searching, or testing. The manager integrates results and decides the next steps. The hierarchy itself does not require planning—it is a structural pattern that defines who decides what at each level. This structure scales well but requires careful design of roles and interfaces.

- **Sequential / Pipeline Orchestration**: Pipeline orchestration executes tasks in a fixed sequence, where the output of one stage becomes the input of the next. Each stage performs a specific transformation or validation step. This approach is simple and predictable but lacks flexibility when earlier assumptions are incorrect.
- **Supervisor–Worker**: In the Supervisor–Worker pattern, a supervisor agent oversees one or more worker agents that perform concrete tasks. The supervisor is responsible for assigning work, monitoring progress, and deciding when tasks are complete. ‘Reflection’ is commonly used by the supervisor to evaluate worker outputs, identify errors or gaps, and decide whether to accept results, request revisions, or retry tasks. This adds a quality-control loop without requiring workers to reason about the overall system

5.2.3 Execution Capabilities

5.2.3.1 Worker Execution Capabilities

The worker execution capability allows an agent to execute a specific task or sub-task assigned by a planner or orchestrator. It applies domain knowledge or skills—such as coding, analysis, writing, or testing—to produce a concrete result. Worker agents focus on task execution rather than control flow.

Worker agents operate within defined boundaries and report results back to a planner or orchestrator. They may use tools internally but do not decide when or why work is initiated.

5.2.3.2 Tool Using

The tool-using capability allows an agent to invoke and operate external tools or systems. It translates high-level instructions into concrete tool calls, manages inputs and outputs, and normalises results into a shared format. This capability encapsulates knowledge of tool interfaces, constraints, and failure modes.

Agents performing tool-using tasks are often orchestrated by planners or supervisors. They may perform lightweight validation or retries but do not make strategic decisions about task decomposition or sequencing.

5.2.4 Cross-Cutting

5.2.4.1 Learning Capabilities

Learning allows an agent to adapt its behaviour over time based on past experience and feedback. It captures signals such as successes, failures, user feedback, or execution outcomes and uses them to improve future decisions. Learning may influence planning strategies, orchestration choices, task execution, or tool usage.

Agents implementing learning capabilities often rely on semantic indexes as a form of long-term memory. These indexes store embeddings of past interactions, plans, results, or domain knowledge and enable similarity-based retrieval. While the semantic index provides storage and recall, the learning agent determines what to store, when to retrieve it, and how retrieved knowledge should influence future behaviour. In this way, semantic indexes support learning, but the agent itself is responsible for adaptation.

Learning as a Cross-Cutting Capability

Learning is rarely implemented as a standalone agent role. Instead, it is most often a cross-cutting capability such as planning, orchestration, worker execution, or tool-using capabilities. In these cases, learning enhances existing responsibilities rather than introducing new ones.

For example, an agent that implements a planning capabilities may learn which plans succeed most often in certain contexts, orchestration capabilities may learn improved routing or delegation strategies, and worker execution capabilities may learn how to improve output quality over time. Tool using capabilities may learn optimal invocation patterns or error-handling strategies. In all cases, learning leverages memory—often backed by semantic indexes—to inform future behaviour while preserving the original agent role and structure.

5.2.5 Agent Capability Stack

Agents are typically composed of multiple capabilities that operate at different levels of responsibility. Tool using and worker execution capabilities perform concrete actions. Orchestration capabilities coordinate work across tools or agents. Planning capabilities determine how a goal should be achieved. Learning capabilities operate across these layers, capturing experience and improving behaviour over time.

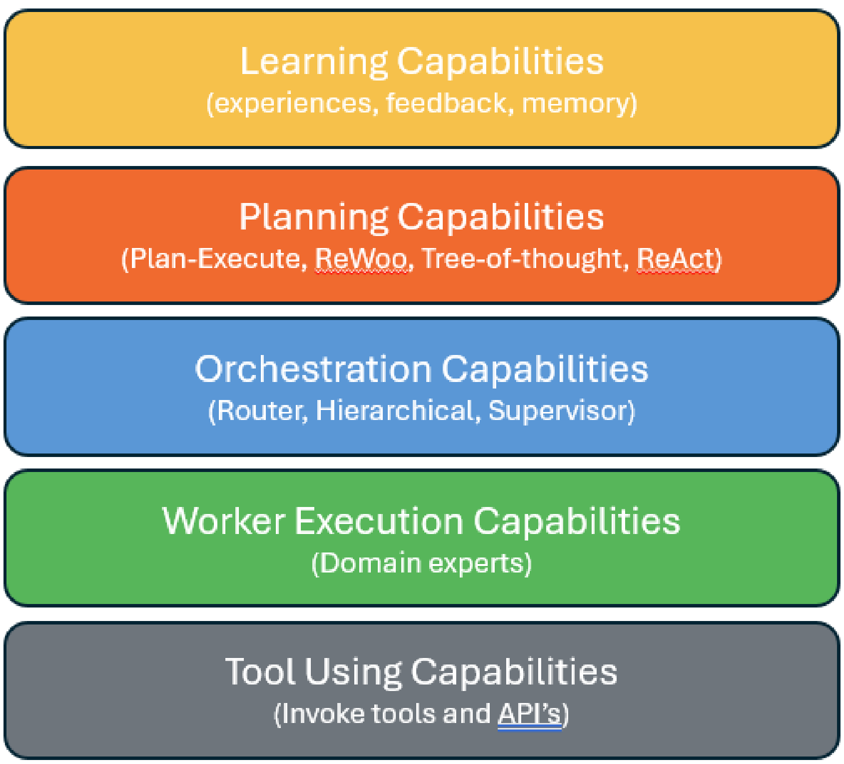

Figure 5 - Agent Capability Stack

Agents combine capabilities across layers rather than existing as a single type.

- **Simple Agent**
  - Reasoning: ReAct
  - Execution: Tool Use

- **Planner Agent**
  - Reasoning: Planning
  - Control: Orchestration
  - Execution: Worker tasks
- **Complex multi-agent systems**
  - Planner agent:
    Planning + orchestration
  - Worker agents:
    Worker execution + tool use
  - Supervisor agent:
    Orchestration + reflection
  - Learning:
    Cross-cutting across all agents

5.3 Designing Agentic Systems: Multi vs Single Agents

5.3.1 When to Use a Single Agent

Use a single agent when the problem is cohesive, bounded, and low-coordination.

A single agent works well when:

- The task can be solved with one primary skill, a group of related skills, or a reasoning style
- Steps are mostly sequential and easy to reason about
- Latency and cost matter more than modularity
- You don't need strong isolation between responsibilities

Typical examples:

- Answering questions or generating content
- Simple workflows with a small number of tool calls
- ReAct-style agents that reason and act locally

5.3.2 When to Use a Multi-Agent System

Use a multi-agent system when the problem requires **specialization, parallelism, or strong separation of concerns**.

Multi-agent systems make sense when:

- The task naturally decomposes into **distinct roles or skills**
- Different parts of the problem benefit from **different prompts, tools, or policies**
- You need **parallel execution** to reduce end-to-end latency
- The problem can be **decomposed into smaller**, specialised responsibilities, reducing overall complexity and resulting in a more **robust and maintainable system**.
- **Distributed Agentic systems**, integrating existing agents into another agentic application

Typical examples:

- Complex, multi-step business processes

- IDE copilots with planning, coding, testing, and review
- Systems that require supervision, validation, or retries
- Long-running or event-driven workflows

Multi-agent systems organise complex problems into specialised components, with each agent responsible for a clearly defined role. By decomposing the problem in this way, complexity is distributed rather than concentrated, making individual agents easier to design, test, and evolve. This separation of responsibilities improves scalability by allowing components to be developed and scaled independently, enhances control through explicit orchestration and coordination, and increases system resilience by isolating failures and enabling targeted recovery or substitution of individual agents.

5.4 Designing AI Agents

5.4.1 Agent Responsibility Assignment

AI Agents encapsulate autonomous or semi-autonomous reasoning capabilities within an application. Like services, agents must be designed to be loosely coupled, coherent, testable, and easy to evolve. Two core principles guide effective agent design:

- Ensuring each agent has a focused responsibility
- Managing dependencies between agents and systems

Failure to follow these principles leads to brittle systems, unpredictable behaviour, and difficulty scaling or governing agent behaviour.

Each AI Agent should have a single, clearly defined responsibility. Agents that attempt to perform multiple unrelated tasks become difficult to understand, test, and govern.

The following patterns are ranked from best to worst.

- **Single-Purpose Agent (Best Practice)**
  The agent performs one well-defined function with clear inputs, constraints, and outputs.

  Best Practice: It should be to define the agent's responsibility in one sentence. All the Agent's skills or capabilities should be related. Limit the agent's available tools to only what is required. An agent may use multiple services when they support a single coherent responsibility, but cross-service integration logic should preferably be encapsulated behind shared tools or services so the agent remains focused on its primary role.

- **Conditional Behaviour Within a Single Domain (Use Carefully)**
  Allow conditional behaviour within an agent when it supports a single, coherent business responsibility. This is often desirable, as it enables the agent to encapsulate a complete function with multiple valid outcomes. However, as conditional logic grows, ensure it does not expand into multiple distinct responsibilities such as coordination, integration, or reporting. When an agent begins to combine different types of decisions or accumulate complex branching across unrelated concerns, it becomes harder to reason about, test, and evolve. In these cases, consider separating those concerns into specialised agents or services while keeping the original agent focused on its core responsibility.

For instance, a Document Review Agent that evaluates a document and decides to approve, reject, or request a revision is well-scoped, as all outcomes fall within the same business function. However, if the same agent also determines which downstream systems to call, performs external compliance checks, and sends notifications, it is no longer just reviewing documents—it is coordinating and integrating across multiple concerns. At this point, separating these responsibilities can simplify the design and improve maintainability.

Best Practice: If this behaviour starts to increase, think about breaking the capabilities into sub-agents and making this agent an explicit orchestrator/router

- **Multi-Purpose or “Generalist” Agent (Avoid)**
  A single agent performs unrelated responsibilities such as classification, data access, approvals, messaging, and reporting. These agents are difficult to test and scale. As new functionality/capabilities are added, the agents grow larger and more complex.

  Best Practice: Split agents by business capability or decision responsibility. It is preferable to have many small agents with clear contracts over one large agent. Resist grouping tasks for convenience rather than clarity.

5.4.2 Designing Multi-Agent Systems

Multi-agent systems should minimise coupling between agents and centralise orchestration logic. Contract-based message interaction, explicit orchestration, and controlled context sharing improve maintainability, observability, and reliability. External system integrations should be isolated behind shared tools or services, keeping agents focused on reasoning and decision-making.

Within multi-agent systems, the following patterns are ranked from most desirable to least desirable.

- **Contracted Message-Based Interaction (Best Practice)**
  Agents communicate exclusively through well-defined message contracts. Each agent consumes an explicit input schema and produces a structured output schema, without knowledge of downstream consumers
  - Agents are decoupled from each other’s internal logic, prompts, and tools
  - Agents can be replaced, retrained, or reconfigured independently
  - Message contracts provide natural boundaries for testing, replay, and auditing
  - The Vantiq Multi Agent framework provides this contract-based interaction, and if using multiple agents or the standalone GenAI Services, rely on this contract rather than implementing your own

  Best Practice: Utilise GenAI Services and the defined contract that is provided by the multiple agent framework – GenAI Agent Description, Skills, etc. If using standard services, then: Define explicit message types with clear semantic meaning; Keep message payloads minimal and purpose-driven

- **Orchestrator and Collaboration of multiple Agents (Strong Practice)**
  A coordinating workflow or orchestration agent manages execution order, routing, retries, and termination. Agents do not invoke one another directly.
  - Dependency relationships are centralized and visible, and all logic related to orchestration is centralized

- If using the GenAI Services, then use the Agent Tags to group related agents together. If not using the GenAI Service, then use some other mechanism of dynamically identifying groups of Agents

Best Practice: Place sequencing, branching, and error handling in the orchestrator. Treat orchestrator logic as part of the Agent implementation rather than expected agent behaviour.

- **Direct Agent Invocation (Use with Caution)**
  One agent directly invokes another agent synchronously as part of its reasoning flow. There are two levels of direct invocation: through the multi-agent framework, e.g., messageSend, and the direct invocation of a skill or procedure. The two approaches are both forms of tight binding, making testing and observation in isolation harder. There are occasions when this approach is appropriate, such as integrating narrow, stable utility agents (e.g., redaction, validation)

  Best Practice: Keep invocation interfaces narrow and stable. If using the multi-agent framework, the use of messageSend rather than direct skill invocation or procedure invocation is desirable, as it provides a level of abstraction and also binds the two agents together, but not the two skills. Apply strict timeouts and fallback behaviour.

- **Oversharing Context Between Agents (Discouraged)**
  Agents pass large context blobs, full conversation histories, or raw documents to downstream agents "just in case". This creates hidden and implicit dependencies, increases, and makes downstream behaviour sensitive to unrelated prompt changes, and also potentially increases cost (token counts), latency, and unpredictability.

  Best Practice: Pass summaries, extracted facts, and evidence references instead of raw data.

- **Shared Mutable Memory Across Agents (Avoid)**
  Multiple agents directly read and write to the same long-term memory or shared data structures. This could lead to race conditions and inconsistent state, loss of provenance and accountability, and difficult debugging and auditing.

  Best Practice: Each Agent should maintain its own memory, both long- & short-term, and utilize write-through capabilities such as Collaborations and Conversations. Also, not all conversational memory needs to be persisted, so this should not be the default action.

- **Fragmented External System Dependencies/Tools (Avoid)**
  Multiple agents independently integrate with external systems (databases, CRMs, ticketing systems, messaging platforms). This increased maintenance when external APIs change leads to fragmented observability, making tracing and debugging complex.

  Best Practice: Isolate external integrations behind shared tools or services. This allows for the provision of consistent schemas, retry behaviour, and logging. Keep agents focused on reasoning and decision-making, not integration mechanics.

6. Design Services from Event Storming

Event Storming provides a powerful way to explore and visualize how a real-time, collaborative application behaves from a business perspective. It captures the flow of events—things that happen—and the actions and commands that respond to those events. Once you've mapped out this behaviour, the next step is turning that model into a set of modular, maintainable services. That's where the principles of **functional focus** and **managing dependencies** come into play.

These two principles are essential for moving from an exploratory model to a well-structured, production-ready service architecture. They help ensure that each part of your system is logically organized, independently manageable, and able to evolve without unnecessary complexity.

6.1 Functional Focus: Building Services with a Clear Purpose

Functional focus is about giving each service a specific, well-defined job. It ensures that the logic inside a service supports a single area of business responsibility, and nothing more. When a service is functionally focused, it's easier to understand, test, reuse, and maintain.

Your Event Storming model naturally reveals these focused areas of behaviour. For example, you might identify a group of events like TemperatureChanged, RPMChanged, and PumpFailureDetected, and a series of actions like AnalyzeSensorData or DetectAnomaly. These clearly relate to real-time monitoring and diagnostics—so it makes sense to create a service that owns this responsibility, such as RefrigerationPumpMonitoringService.

By keeping all related logic together and excluding unrelated functionality, you avoid creating "God services" that try to do too much. Instead, each service becomes a self-contained module that handles one business concept end-to-end.

6.2 Managing Dependencies: Designing for Independence

While functional focus tells you what to group together, **managing dependencies** is about how those services interact with each other. Poorly managed dependencies create hidden linkages that make systems fragile and hard to change. Instead, the goal is to reduce the need for services to know about each other in detail.

In a well-structured Vantiq application, services communicate using events routed through the **Design Model**, not through direct procedure calls or shared databases. This kind of design minimizes dependencies—services react to what's happening in the system without needing to be aware of who triggered it or what will happen next.

For instance, after a RefrigerationPumpMonitoringService detects a potential failure, it can emit a PumpFailureDetected event. A separate IncidentResponseService can then react by notifying a technician or creating a repair order. These two services work together, but they aren't directly connected—they're simply listening and responding to events. This separation of concerns makes the system easier to scale, debug, and evolve.

There are times when direct communication is necessary, such as when one service needs to validate information synchronously (e.g., "Is this technician available now?"). But this should be the exception, not the norm. When dependencies are well managed, changes in one service are less likely to break another.

It's also important to avoid hidden or indirect dependencies, such as passing overly large data payloads between services or having multiple services update the same data source. Instead, use **Integration Services** to manage connections to external systems and **Data Services** to centralize access to persistent data. This keeps the core of your system clean and isolated.

6.3 From Model to Modular Services

As you move from an Event Storming model to a Vantiq-based application design, use **functional focus** to define what each service *should do*, and **dependency management** to define *how it should interact with others*.

Ask yourself:

- Do the actions and logic inside this service support a single business concept?
- Can this service be developed, tested, and deployed on its own?
- Does it rely on other services in ways that could cause future complexity?
- Is communication between services handled through well-scoped events rather than tight integrations?

By using these two principles together, you can design services that are modular, resilient, and aligned with the real-time, event-driven nature of your application. Vantiq's modeling tools—especially the **System Modeler** and **Design Modeler**—make it easy to capture this structure visually and implement it cleanly.

7. Identifying Vantiq Clients

In Vantiq, client applications play a central role in bridging real-time, event-driven systems with human users. However, the design principles that guide client applications are distinctly different from those used to design services. Rather than reflecting the internal structure of the system, Vantiq clients should be organized around real-world user tasks and decision flows.

7.1 Clients Are Defined by Functional Intent

Clients should be organized by what users need to accomplish, not by how backend functionality is implemented.

A single client may depend on multiple services, and a single service may support many clients. What defines the boundary of a client is the workflow it supports, not the system modules it connects to.

For example:

- A Field Technician App exists to help a technician receive job alerts, view details, and complete tasks.
- A Supervisor Dashboard serves to oversee operations, track escalations, and manage task assignments.
- An AI Assistant Panel supports decision-making by summarizing cases or suggesting resolutions.

These are not reflections of service groupings; they are reflections of user purpose.

7.2 Clients Use Explicit, Direct Integration with Services

Clients in Vantiq do not benefit from the loose event routing mechanisms available to services. Instead, they must explicitly bind to service interfaces using:

- Procedure calls to initiate actions or retrieve data
- Event subscriptions to receive real-time updates

This means there is no abstraction layer between the client and the services it depends on. If a client needs to receive events, the corresponding service must publish them as public outbound events. If the client needs to trigger behavior, the service must expose that behavior as a public procedure.

Because of this direct integration model, services intended for client consumption must expose stable, well-documented interfaces tailored to the client’s needs, rather than internal logic or implementation details.

7.3 Maintain Narrow, Focused Service Usage Within Clients

A well-structured Vantiq client should use only the specific subset of service interfaces necessary for its intended purpose. This avoids unnecessary complexity, improves testability, and strengthens the clarity of the client’s role in the system.

For instance, a mobile client used by a technician in the field should:

- Subscribe only to task assignment events
- Invoke only those procedures related to task completion or issue reporting

It should not subscribe to system-wide events, nor should it attempt to access administrative services used by dispatchers or supervisors.

This is similar to the principle of least privilege in security: a client should have access only to what it needs to do its job, and nothing more.

7.4 Clients Must Be Context-Aware and Role-Specific

The form, behavior, and scope of a client should be informed by its real-world usage context:

- Is it used in the field or in an office?
- Is it designed for quick input or deep analysis?
- Does it require offline capabilities?
- What level of decision-making authority does the user have?

These considerations shape everything from the UI layout to the event types it subscribes to.

For example, a mobile technician app is likely optimized for clarity, speed, and low data consumption. It may show only the most recent or highest-priority task. Meanwhile, a supervisory control panel may require filtering, sorting, historical review, and rich visualizations of ongoing system activity.

These are not differences in underlying data, they are differences in user needs and responsibilities.

7.5 Shared Logic and UI Components Should Be Reused, Not Duplicated

While each client is functionally distinct, it is important to maintain consistency and efficiency across the client suite. This is achieved through the use of shared utilities and reusable UI components:

- Common data formatting, unit conversions, or ID generation logic can be centralized in utility services.
- Shared UI widgets (maps, charts, status indicators) can be reused across clients to ensure visual and functional consistency.
- JavaScript libraries can provide helper functions that simplify client-side logic without duplication.

This enables client developers to focus on functional UX design while benefiting from tested and consistent foundational behaviors.

7.6 The Service-Client Relationship Is Purpose-Driven

In summary, the relationship between services and clients in Vantiq is not structural, but intentional and purpose-driven. Clients are built to support human actions in context. Services are built to encapsulate behavior, data, or external integration. The connection between them exists only where function demands it.

Clients:

- Are aligned with user roles and real-world tasks
- Integrate directly with services using explicit procedures and event subscriptions
- Depend on clear, public service interfaces—not internal logic
- Must be simple, focused, and minimal in scope
- Benefit from reusable utility services and shared components

By maintaining this alignment, Vantiq applications remain adaptable, modular, and human-centered, scalable not just in infrastructure, but in usability and maintainability.

8. Design Principles and Best Practices

Vantiq applications are built to be **real-time**, **event-driven**, **situationally aware**, and **intelligently collaborative**. To achieve these goals, developers must adopt sound architectural practices that emphasize **modularity**, **resilience**, **maintainability**, and **clarity of purpose**. The following design principles synthesize lessons from Vantiq's service model, client architecture, and development methodology into a set of practical guidelines for scalable and sustainable application design.

1. **Functional Responsibility and Service Cohesion**
    a. **Single Responsibility Rule:** Each service/agent should encapsulate a distinct domain concern. Services should be **focused and cohesive**, managing a clearly bounded set of business logic or system tasks.
    b. **Avoid "God Components":** Resist the temptation to accumulate unrelated logic into a single monolithic service or generalist agent. Overloaded components reduce clarity, reusability, and testability.
    c. **Traceability to Domain Concepts:** Service/agent functionality should directly reflect the real-world concepts modelled in Event Storming (e.g., MonitoringService, DispatchService)

2. **Agent-Oriented Design Principles**
    a. **Goal-Driven, Single-Responsibility Agents:** Agents should be defined by the goal or decision they are responsible for achieving. Each agent must have a clear, focused responsibility aligned to a single domain concern, avoiding the accumulation of unrelated capabilities. When responsibilities begin to diverge, decompose into specialised agents rather than expanding a single generalist agent.
    b. **Independent Agents with Explicit Interaction Boundaries:** Agents should operate as independent components that interact through well-defined boundaries. They should not depend on the internal logic, prompts, or tools of other agents, and direct invocation should be used sparingly. Structured, decoupled interaction enables agents to be replaced or evolved without impacting others.
    c. **Purposeful Agent Decomposition:** A multi-agent system should be introduced only when the problem requires specialization, separation of concerns, or coordination across distinct responsibilities. A single agent is appropriate for cohesive, bounded problems, while multiple agents should be used when clear role separation emerges.

3. **Loosely Coupled Architecture**
    a. **Event-First Design:** Prefer asynchronous **event-based communication** between service/agent over synchronous calls. Events enable decoupled, scalable interactions and improve system resiliency.
    b. **Use the Design Model to Route Events:** Abstract service connections through visual event routes. This promotes maintainability and allows dynamic reconfiguration without code changes.
    c. **Minimize Synchronous Dependencies:** Use procedure calls only when immediate responses or validations are required (e.g., authentication, data lookup).

4. **Manageable Service Interactions**

a. **Lean Event Payloads:** Events should carry only the necessary data for downstream consumers. Avoid bloated payloads that create tight coupling and reduce performance.
b. **Avoid Shared State:** Never allow multiple services to modify the same persistent storage directly. Encapsulate shared data access in **dedicated Data Services**.
c. **Isolate External Systems:** Use **Integration Services** to interact with third-party systems (e.g., Kafka, REST APIs). Keep these dependencies out of business logic services.

5. **Client-Driven by Role and Intent**
    a. **Task-Oriented Design:** Vantiq clients should be centered on **real-world user goals**, not backend structure. Define clients by their purpose (e.g., FieldTechnicianApp, SupervisorDashboard).
    b. **Minimalist Integration Scope:** A client should call **only the services it requires**. Apply the principle of least privilege to maintain clarity and security.
    c. **Explicit Contracts:** Services that support clients must expose **well-defined public procedures and outbound events**. Never expose internal-only logic to clients.

6. **Reusability and Shared Components**
    a. **Utility Services:** Centralize shared logic (e.g., ID generation, formatting, calculations) in reusable utility services.
    b. **Shared UI Components:** Use common widgets (e.g., maps, charts, filters) across clients to ensure visual consistency and avoid duplication.
    c. **Common Libraries:** Maintain shared JavaScript code or themes to simplify maintenance and promote best practices across clients.

7. **Modelling Continuity and Traceability**
    a. **Start from Event Storming (when possible):** Use the System Modeler to identify domain events, reactions, and commands before designing services.
    b. **Maintain Alignment:** Ensure that the logic and flow defined in the Design Model remain faithful to the original requirements model.
    c. **Iterative Refinement:** Vantiq supports both **top-down and bottom-up** workflows—embrace continuous evolution and back-mapping between models and implementation.