# Layered Agentic Systems

**Integrating Structured and Unstructured Agents in Layered Agentic Systems**

**Abstract**

This white paper describes a design pattern for architecting agentic systems using a layered approach that balances structured program logic with unstructured natural language processing. The approach draws from a conceptual model of a golf ball —an outer shell of listener-driven, user-facing agents surrounding an inner core of logic-driven, structured agents. The central mechanism for integration between these layers is the transmission and transformation of inputs and facts. The goal is to provide a reusable, modular pattern for building conversational and decision-capable AI workflows that are easy to extend, debug, and explain.

**Layered Architecture**

**Outer Layer: Clarifiers and Conflators**

The outer layer is composed of **user-facing agents** that:
• Interact with users via natural language
• Extract and clarify intent
• Conflate ambiguous input into structured facts
• Confirm meaning before delegation

These agents generally:
• Use listeners to invoke other agents
• Emit facts that are longterm or ephemeral
• Rarely require inputs unless for extraction or disambiguation

Example agent:

```
"clarify.request_type":
description: "Interprets and confirms the user's main reason for
calling."
listener: |
```

```
From what the user said, extract whether this is a crisis, a request for
resources, or a general inquiry.
Confirm the interpretation by restating it as a question.
facts:
request_type:
description: "Type of user request: resource, crisis, or general.
scope: local"
```

**Inner Layer: Decision and Action Agents**

The inner layer consists of **structured agents** that:
- Operate with clearly defined inputs
- Use deterministic logic or backend function calls
- Route, compute, or trigger system behavior

These agents typically:
- Use template or function:
- Read from run-local facts (set by outer agents)
- Return values into the same run context

Example agent:

```
"route.by_request_type":
description: Routes the call to the appropriate next step based on the
clarified request type.
template: |
{{- if eq .request_type "crisis" -}}
{{ .Get "emergency.checker" }}
{{- else if eq .request_type "resource" -}}
{{ .Get "info.navigator" }}
{{- else -}}
{{ .Get "engagement.listener" }}
{{- end }}
inputs:
request_type:
description: Type of user request.
scope: local
```

**Integration: Facts as the Glue**

The interface between outer and inner layers is based on:
• inputs: Declared expected fields for a structured agent
• facts: Populated outputs from listener or clarifier agents

All facts defined with scope: local are available within the current run context, making them accessible to downstream agents.

This allows:
• Seamless flow from AI interpretation to logic execution
• Natural language to be refined before structured decisions
• Agents to specialize while maintaining loose coupling

**Introspective Clarifiers: Outer Agents That Peek Ahead**

A powerful extension to this pattern is enabling outer-layer agents (e.g., clarifiers or listeners) to **introspect the inputs of downstream agents** they intend to prepare for.

This means:
• Before a clarifier runs, it can peek at the inputs of the next agent it will hand off to.
• It can dynamically tailor its prompt to guide the user toward answering those expected inputs.
• The prompt can even include the description fields from those inputs to help AI structure its language toward downstream needs.

This makes outer agents not just listeners, but *teleologically aware facilitators*. They shape interaction in a way that improves AI understanding and simplifies structured extraction in the next step.

This approach:
• Reduces mismatch between AI-generated facts and expected structured inputs
• Supports dynamic prompt generation based on agent spec
• Encourages a more fluid and intelligent handoff between AI and program logic

**Dynamic Entry Points: Changing the Start Agent**

Another important capability is the ability to **dynamically assign the start agent** — the agent that initially interacts with the user.

This enables:
• Flexible, context-sensitive entry flows
• Adaptive user experiences based on language, emotion, or inferred needs
• Personas or routing logic that tailor the entire conversation to the user profile

**How It Works**

The chat context tracks the current start agent. A template agent can change this by using:

```
{{ .Start "some.other.agent" }}
```

This sets a new start agent for the current session or conversation, allowing handoff from a general-purpose greeter to a specialized domain agent.

This pattern supports:
• Persona triage (e.g., senior support vs. crisis counseling)
• Human-like conversational flows where the first agent “hands off” after listening
• Greater modularity and reuse of downstream workflows

**Naming Convention for Layered Clarity**

To support readability and composability, we suggest the following prefixes:

| Layer | Prefix | Example |
|---|---|---|
| Outer (AI) | clarify. / listen. | clarify.name |
| Middle (Routing) | route. / branch. | route.by_request_type |
| Inner (Logic) | act. / handle. | act.escalate_to_operator |

This convention allows a reader or system trace to instantly recognize the agent’s role.

**Philosophical Model**

This model assumes a **dual-natured AI system**:
• The **yin**: fluid, interpretive, language-driven outer agents
• The **yang**: structured, logic-bound inner agents

The result is a hybrid architecture where:
• Outer agents *understand and clarify*
• Inner agents *decide and act*

Together, they allow an AI to feel natural to humans but remain reliable and debuggable to developers.

**Conclusion**

This layered agentic design provides a scalable, explainable, and modular way to build AI systems that speak with humans and reason like programs. By respecting the separation of unstructured conflation and structured execution—bridged through inputs and facts—developers can create agent workflows that are robust, testable, and extensible.