Declarative Multi-Agent Plugin: Designing Agent Systems Without Code

Redefining LLM Agent Orchestration with Clean Architecture Principles

The Structural Challenge of Multi-Agent Systems

When LLM agents expand from single entities to collaborative multi-agent systems, complexity increases not linearly but combinatorially. The number of relationships to manage — role boundaries between agents, tool access permissions, execution order, escalation conditions — grows proportionally to the square of the number of agents, creating a fundamental structural problem.

Existing approaches such as LangChain, CrewAI, and AutoGen address this problem through Python/TypeScript SDK code. Agent definitions, tool connections, and orchestration logic are all intertwined within programming languages, forming a structure tightly coupled to specific SDKs. As a result, changing the runtime environment requires rewriting the entire system, and domain experts or non-developers are excluded from participating in agent system design.

This raises a fundamental question: is code truly necessary to define agent roles and relationships? Declaring "what an agent can do" and coding "how it operates" are inherently different activities.

The Essence of the Declarative Approach — "What" vs. "How"

Declarative programming is a paradigm that describes "the desired outcome" and delegates "how to achieve it" to the execution environment. SQL declares data conditions without specifying search algorithms; HTML/CSS declares document structure and style without dictating rendering engine implementation; Kubernetes YAML declares desired infrastructure state without prescribing container scheduling methods. Their common thread lies in the separation of execution environment (runtime) from declaration (specification).

The same principle can be applied to multi-agent systems. Agent roles, capabilities, and constraints are declared in Markdown and YAML, while execution is delegated to runtimes (Claude Code, Codex CLI, Gemini CLI, etc.). The core value of this approach can be summarized in two points. First, runtime neutrality — an abstract declaration like `tier: HIGH` can be interpreted as the optimal model in any runtime environment. Second, non-developer accessibility — anyone who knows Markdown and YAML can build agent systems, including domain experts.

Comparison	Traditional Frameworks	Declarative Approach
Agent Definition	Python/TypeScript SDK code	Markdown + YAML
Orchestration	Graph/function call chain code	Skill prompts (natural language)
Runtime Dependency	Tightly coupled to specific SDK	Runtime-neutral
Architecture Principles	None or framework-dependent	Clean Architecture
Tool Integration	Tool object creation in code	Abstract declaration (tools.yaml) → Gateway mapping
Tier Management	None	4-Tier (HEAVY/HIGH/MEDIUM/LOW) + automatic runtime mapping
Portability	Low (full rewrite required)	High (only replace runtime-mapping.yaml)

Architecture Design — Applying Clean Architecture to Agents

The core principles of Clean Architecture — unidirectional dependency and separation of concerns — are proven design philosophies that ensure long-term maintainability of software systems. The declarative multi-agent architecture applies these principles directly to agent systems, organized into three core layers.

Skills (Controller Layer)

The entry point for user requests and the orchestration layer. It classifies request intent and performs routing by delegating tasks to appropriate agents. Delegation skills (Core, Planning, Orchestrator) invoke agents, while direct skills (Setup, Utility) use the Gateway directly. The YAGNI principle is applied by skipping unnecessary layers depending on the execution path.

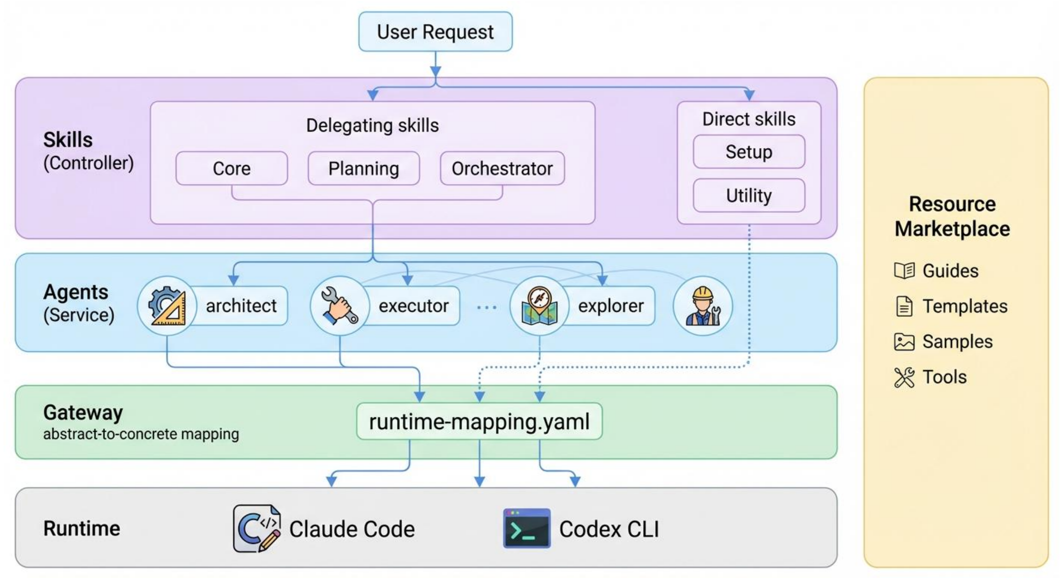
Agents (Service Layer)

Role-based autonomous execution units. Each agent consists of a pair: AGENT.md (goals and workflow prompts) and agentcard.yaml (identity, capabilities, and constraint declarations). Each agent focuses on a single specialized role, and tasks that cross role boundaries are delegated to other agents according to handoff rules.

Gateway (Infrastructure Layer)

The infrastructure layer that provides mapping tables between abstract declarations and concrete runtimes. runtime-mapping.yaml transforms agent abstract tiers into actual LLM models and abstract tools into actual tool implementations. When the runtime environment changes, only this mapping file needs to be replaced.

Beyond these three layers, a Resource Marketplace serves as a shared pool of guides, templates, samples, and tools that can be shared across plugins. The practical benefit of the unidirectional dependency structure (Skills → Agents → Gateway) lies in the independent replaceability of each layer. Adding or modifying agents does not affect the Gateway, and replacing the runtime does not require changes to Skills or Agents.



Declarative Multi-Agent Plugin Architecture — Clean Architecture-based layer structure

Operational Mechanism — Prompt Assembly and Tier-Based Execution

Prompt Assembly

When a delegation skill spawns an agent, the prompt is assembled in three layers. This ordering is designed to maximize the runtime's prefix cache hit rate.

Layer 1 — Shared Static (runtime-mapping).

Rules applied universally to all agents. It includes mapping information from the Gateway's runtime-mapping.yaml that transforms abstract declarations into concrete models and tools. Since this is identical across all agent invocations within a session, cache efficiency is highest.

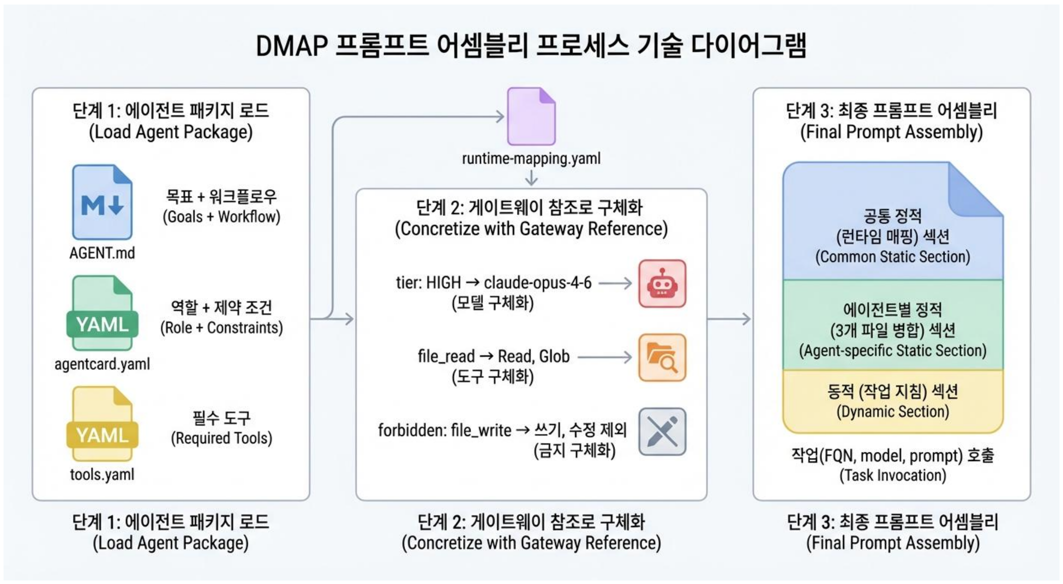
Layer 2 — Per-Agent Static (AGENT.md + agentcard.yaml + tools.yaml).

Three files containing each agent's role, workflow, and tool definitions are synthesized into a single prompt. AGENT.md provides goals and methods (WHY+HOW), agentcard.yaml provides identity and constraints (WHO+WHAT+WHEN), and tools.yaml provides abstract tool interfaces. The cache is reused for repeated invocations of the same agent.

Layer 3 — Dynamic (task instructions).

The specific task content delivered by the skill. This is the only layer that changes with every invocation, consisting of five items: task objective (TASK), expected deliverable (EXPECTED OUTCOME), mandatory requirements (MUST DO), prohibited actions (MUST NOT DO), and context (CONTEXT).

The Gateway's runtime-mapping.yaml performs three transformations during this assembly process. Model concretization maps an agent's abstract tier (e.g., `tier: HIGH`) to an actual model (e.g., `claude-opus-4-6`). Tool concretization maps abstract tools from tools.yaml to actual tools. Forbidden action concretization maps forbidden action categories from agentcard.yaml to actual tools. The final set of tools provided to the agent is (concretized tools) - (forbidden tools).


DMAP 프롬프트 어셈블리 프로세스 기술 다이어그램
단계 1: 에이전트 패키지 로드 (Load Agent Package)
AGENT.md
목표 + 워크플로우 (Goals + Workflow)
agentcard.yaml
역할 + 제약 조건 (Role + Constraints)
tools.yaml
필수 도구 (Required Tools)
runtime-mapping.yaml
단계 2: 게이트웨이 참조로 구체화 (Concretize with Gateway Reference)
tier: HIGH → claude-opus-4-6 (모델 구체화)
file_read → Read, Glob (도구 구체화)
forbidden: file_write → 쓰기, 수정 제외 (금지 구체화)
단계 3: 최종 프롬프트 어셈블리 (Final Prompt Assembly)
공통 정적 (런타임 매핑) 섹션 (Common Static Section)
에이전트별 정적 (3개 파일 병합) 섹션 (Agent-specific Static Section)
동적 (작업 지침) 섹션 (Dynamic Section)
작업(FQN, model, prompt) 호출 (Task Invocation)
단계 1: 에이전트 패키지 로드 (Load Agent Package)
단계 2: 게이트웨이 참조로 구체화 (Concretize with Gateway Reference)
단계 3: 최종 프롬프트 어셈블리 (Final Prompt Assembly)

Prompt Assembly Process — Three-layer synthesis of static and dynamic layers

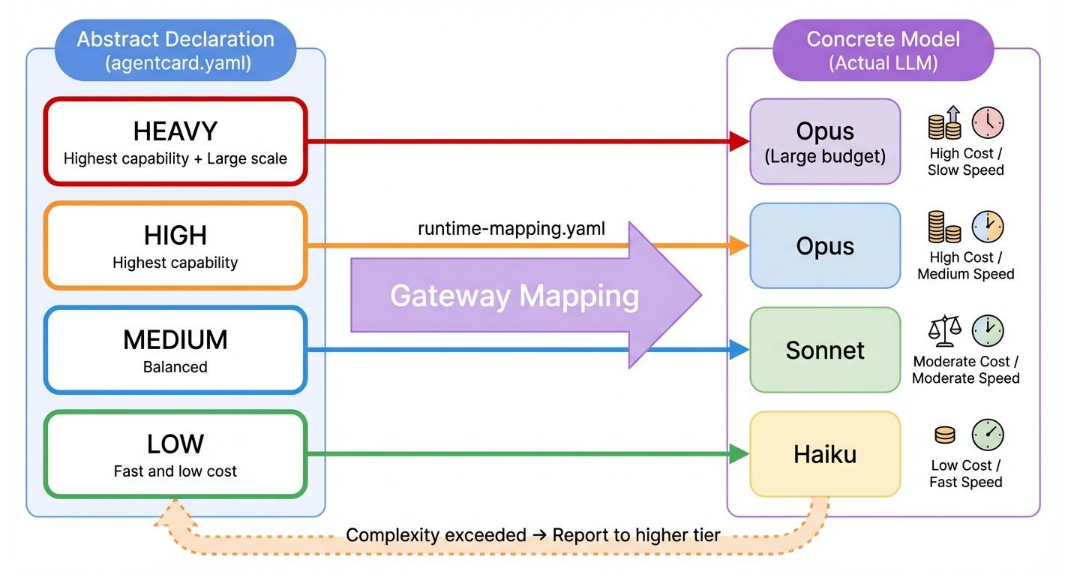
4-Tier Model Mapping

Agent capability requirements are declared across four abstract tiers: HEAVY, HIGH, MEDIUM, and LOW. Each tier is assigned an optimal model based on cost-capability trade-offs, and agents of the same role are separated into tier variants to ensure cost efficiency.

Tier	Characteristics	Suitable Tasks
HEAVY	Highest capability + large	Extended reasoning, large-scale multi-file tasks

	budget	
HIGH	Highest capability	Complex decision-making, deep analysis
MEDIUM	Balanced	Feature implementation, general analysis
LOW	Fast and low-cost	Single lookups, simple modifications

A noteworthy aspect is the escalation mechanism. When a LOW-tier agent recognizes its own limitations, it stops work and reports an escalation to a higher tier. This self-awareness-based step-by-step delegation is a key mechanism that prevents resource waste while ensuring quality. Tier variant agents inherit the base agent's configuration and only describe the parts that need overriding, minimizing duplication.



4-Tier Model Mapping — Transformation from abstract tier declaration to concrete LLM model

Execution Flow — Skill-Based Orchestration

Skill activation occurs through two pathways.

Direct Activation

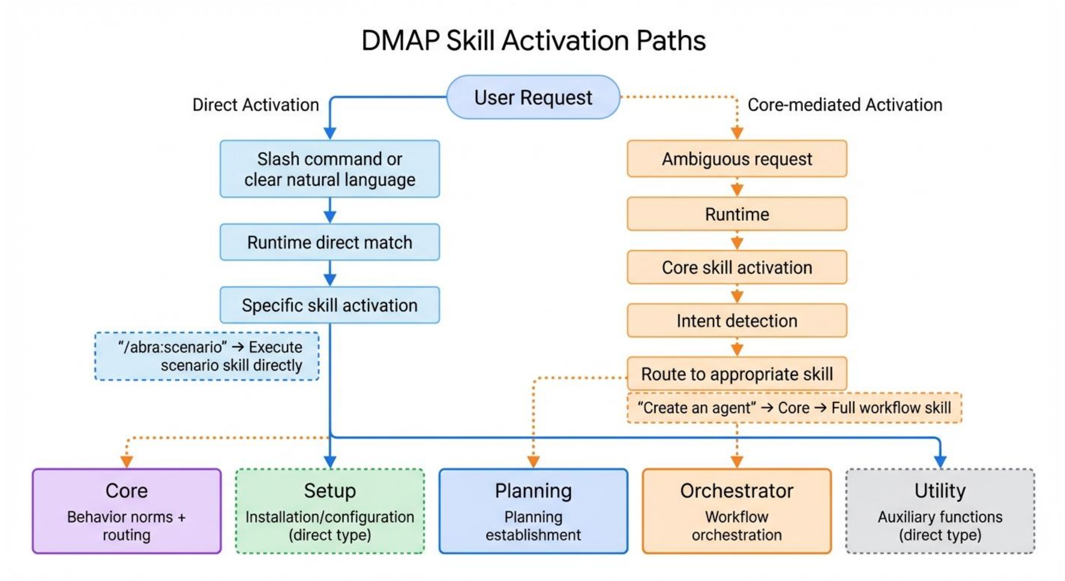
When the runtime detects an explicit slash command (`/plugin:skill`) or a natural language request that shows high similarity to a skill's metadata (frontmatter description), the corresponding skill is loaded immediately. This is a path where the user's intent is directly linked to a specific skill without intermediate layers.

Core-Mediated Activation

When a user's request is ambiguous or spans multiple skills, the runtime first activates the Core skill. The Core skill is limited to determining request intent and routing to the appropriate skill. Actual task execution is handled by the routed target skill (Orchestrator, Planning, etc.).

Execution paths diverge based on skill type. Delegation skills (Core, Planning, Orchestrator) invoke agents to delegate tasks, while direct skills (Setup, Utility) use Gateway tools directly. This dual-path design is the result of simultaneously applying Clean Architecture's separation of concerns and the YAGNI principle. Routing procedural tasks like installation or status checks through agents would generate unnecessary LLM calls, making it rational for direct skills to access the Gateway directly.

At session start, the runtime scans the `skills/` directory to automatically discover all skills. Skills become immediately available simply by placing them in the directory, with no separate registration procedure required.


DMAP Skill Activation Paths
User Request
Direct Activation
Core-mediated Activation
Slash command or clear natural language
Runtime direct match
Specific skill activation
"/abra:scenario" → Execute scenario skill directly
Ambiguous request
Runtime
Core skill activation
Intent detection
Route to appropriate skill
"Create an agent" → Core → Full workflow skill
Core
Behavior norms + routing
Setup
Installation/configuration (direct type)
Planning
Planning establishment
Orchestrator
Workflow orchestration
Utility
Auxiliary functions (direct type)

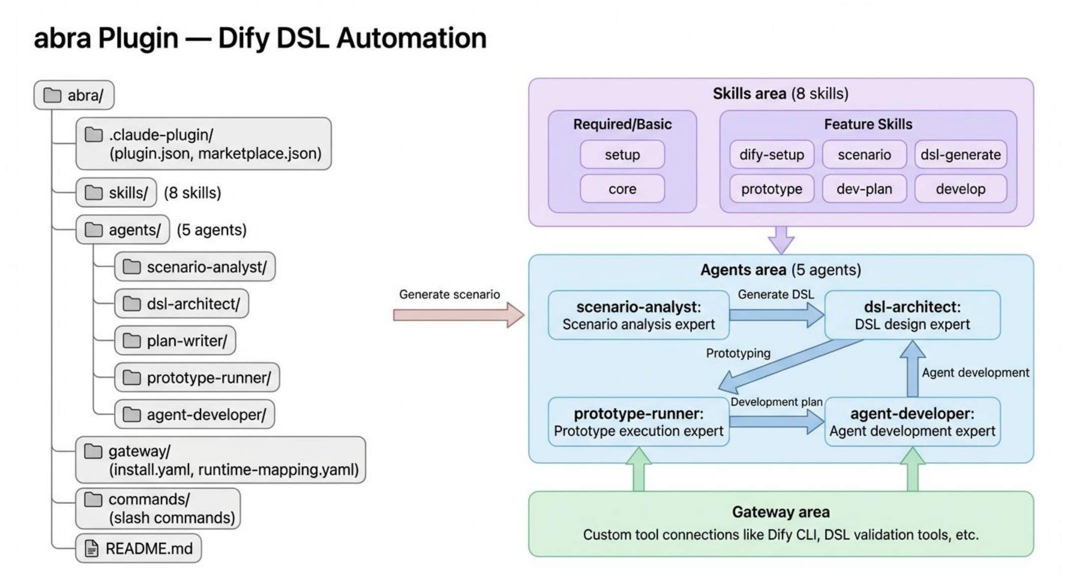
Skill Activation Pathways — Dual paths of direct activation and Core-mediated activation

Implementation Case Study — DMAP and the abra Plugin

The practical implementation of the declarative architecture described above is DMAP (Declarative Multi-Agent Plugin). DMAP is an open-source plugin architecture standard that defines multi-agent systems using only Markdown and YAML without any code. The DMAP Builder is a tool that automatically generates plugins according to this standard.

The abra plugin serves as an application case for DMAP. abra is a plugin that automatically generates Dify workflow DSL from a single natural language command and performs prototyping. Five agents (scenario-analyst, dsl-architect, plan-writer, prototype-runner, agent-developer) each handle their specialized roles, while eight skills orchestrate the workflow from scenario analysis to code development. The entire structure is composed solely of Markdown and YAML — no programming code is used in any aspect, including agent role definitions, tool connections, or execution ordering.

This implementation demonstrates that the declarative approach is not merely a theoretical possibility but a viable, working system. The entire project is publicly available on GitHub (https://github.com/cna-bootcamp/gen-dmap).


abra Plugin — Dify DSL Automation
abra/
.claude-plugin/ (plugin.json, marketplace.json)
skills/ (8 skills)
agents/ (5 agents)
scenario-analyst/
dsl-architect/
plan-writer/
prototype-runner/
agent-developer/
gateway/ (install.yaml, runtime-mapping.yaml)
commands/ (slash commands)
README.md
Skills area (8 skills)
Required/Basic
setup
core
Feature Skills
dify-setup
scenario
dsl-generate
prototype
dev-plan
develop
Agents area (5 agents)
Generate scenario
scenario-analyst: Scenario analysis expert
Generate DSL
dsl-architect: DSL design expert
Prototyping
Agent development
prototype-runner: Prototype execution expert
Development plan
agent-developer: Agent development expert
Gateway area
Custom tool connections like Dify CLI, DSL validation tools, etc.

abra Plugin Structure — Declarative composition of 5 agents and 8 skills

Outlook — The Future of the Declarative Agent Ecosystem

The impact of the declarative approach on the agent development ecosystem is noteworthy across three dimensions.

First, it changes the collaboration model between developers and non-developers. When agent system design shifts from code to declarations, a division of labor becomes possible where domain experts directly define agent roles and constraints while developers handle infrastructure mapping. This expands the range of participants in agent system design.

Second, runtime portability opens the possibility of agent reuse. If a single plugin can operate across various runtimes such as Claude Code, Codex CLI, and Gemini CLI simply by swapping runtime-mapping.yaml, the agent ecosystem can evolve in a direction free from platform lock-in.

Third, there is the direction of agent system standardization. By formalizing agent roles, capabilities, constraints, and handoffs on top of universal formats like Markdown and YAML, a foundation is established for systematic verification, sharing, and composition of agent packages.

The paradigm shift from code to declaration ultimately separates the concerns of "what to build" from "how to execute it." This suggests that the principle software engineering has repeatedly proven — that raising the level of abstraction makes system complexity manageable — holds true in the domain of LLM agents as well.

> **NOTE:** DMAP standard documents and samples are available on GitHub: https://github.com/cna-bootcamp/gen-dmap