



Politecnico di Torino & EURECOM

MSc Software Engineering (PoliTo) — MSc Data Science (EURECOM)

Academic Year 2025/2026 — Graduation Session September 2026

Internship at SAP Labs, Mougins, France — 16/02/2026 – 15/08/2026

Design and Development of a Multi-Agent LLM System for Dynamic Interface Generation and Integration with Business APIs

Supervisor:

MICHIARDI Pietro, EURECOM

DE RUSSIS Luigi, Politecnico di Torino

GARZA Paolo, Politecnico di Torino

Company Supervisor:

DEMONCHAUX Marine, SAP Labs

Candidate:

Alice Boccadifuoco

Rapport de stage confidentiel / Confidential thesis report

OUI / YES ☐ *NON / NO* ☒

ABSTRACT

Design and Development of a Multi-Agent LLM System for Dynamic Interface Generation

Alice Boccadifuoco

This thesis investigates the engineering challenges of deploying multi-agent LLM systems in enterprise environments, where the shared context window becomes a critical bottleneck. Through an industrial case study (the Joule multi-agent integration for the Unified Analytics Overview Capability of SAP Cloud ALM), four design strategies are applied: an API redesign achieving 75.6–85.6% token reduction on the `ProviderDefinitions` endpoint by avoiding the *Execution Trap*, empirically validated against a quantitative baseline benchmark; a hierarchical Router-Skill orchestration pattern enforcing context isolation and a migration of domain knowledge to a structured MCP server implementing progressive disclosure, both evaluated qualitatively against the baseline architecture; and a proposed dual-LLM compression architecture (Gemini 2.5 Flash for context summarization, GPT-4.1 for domain reasoning), of which the toolkit-response compression component was implemented and measured (61.5% reduction measured on a single payload). Within the scope of this case study, the results indicate that principled context management and clean architectural boundaries function as prerequisites, not optimizations, for reliable multi-agent systems in production enterprise environments.

RÉSUMÉ

Conception et Développement d'un Système Multi-Agent LLM pour la Génération Dynamique d'Interfaces

Alice Boccadifuoco

Cette thèse examine les défis posés par le déploiement de systèmes LLM multi-agents en environnement d'entreprise, où la fenêtre de contexte partagée constitue un goulot d'étranglement critique. À travers une étude de cas industrielle (l'intégration multi-agent Joule pour la Unified Analytics Overview Capability de SAP Cloud ALM), quatre stratégies de conception sont appliquées : une refonte de l'API atteignant 75,6–85,6% de réduction de tokens en évitant l'*Execution Trap*, validée empiriquement par rapport à une base de référence quantitative ; un patron d'orchestration Router-Skill hiérarchique garantissant l'isolation de contexte et une migration des connaissances métier vers un serveur MCP structuré mettant en œuvre la divulgation progressive, toutes deux évaluées qualitativement par rapport à l'architecture de référence ; et une architecture dual-LLM proposée (Gemini 2.5 Flash pour la compression, GPT-4.1 pour le raisonnement métier), dont la composante de compression des réponses des toolkits a été implémentée et mesurée (réduction de 61,5% mesurée sur une seule réponse). Dans le cadre de cette étude de cas, les résultats indiquent que la gestion rigoureuse du contexte et des frontières architecturales claires constituent des prérequis, et non des optimisations, pour des systèmes multi-agents fiables en environnement d'entreprise.

ACKNOWLEDGEMENTS

To do:

Contents

Abstract **1**

Résumé **2**

Acknowledgements **3**

List of Figures **6**

List of Tables **8**

I Foundations 9

1 Introduction **10**

2 Background and Related Work **15**

2.1 Multi-Agent System Design Patterns 15

2.2 Grounding Techniques for LLM Reliability 19

2.3 Context Window Economics 21

2.4 The Model Context Protocol 23

II Case Study: SAP Cloud ALM Unified Analytics Capability 26

3 System Context and Problem Statement **27**

3.1 SAP Cloud ALM 27

3.2 Joule: SAP's AI Assistant 28

3.3 The Multi-Agent Orchestration System 29

3.4 The Problem: Operational Inefficiency at Scale 31

3.4.1 Token Consumption at the Data Layer 31

3.4.2 Context Window Overload and the Execution Trap 32

3.4.3 Prompt Architecture and Context Bloat 33
3.5 Data Lifecycle Management 33
3.6 Research Objectives 34

4 Applying Hierarchical Orchestration: The Multi-Agent Architecture 36
4.1 The Initial Architecture 37
4.1.1 Tool Architecture of the Dashboard Agent 37
4.2 The Architectural Transition: CALM Orchestration 38
4.2.1 The Router-Skill Pattern in Production 39
4.2.2 Adapting the Agents to the Skill Abstraction 40
4.2.3 Context Isolation as an Architectural Guarantee 41
4.3 The Control Spectrum in Practice 41
4.3.1 Consistency with the Literature 42
4.4 Challenges and Resolutions 43
4.4.1 The Transition Complexity 43
4.4.2 Identifying the Right Abstraction Level 43
4.5 Data Layer Adjustments: Dashboard Versioning 44
4.6 Summary 45

5 Applying Context Window Economics: API and Data Layer Optimization 47
5.1 The Baseline: `MetricDefinitions` 48
5.1.1 The Context Cost at Scale 48
5.2 The Execution Trap 49
5.3 API Redesign: Maximizing Information Density 50
5.3.1 Structural Changes 50
5.3.2 Format Optimization: JSON vs. CSV 51
5.3.3 End-to-End Token Impact 52
5.4 Residual Variability and Limits 54
5.5 The Dual-LLM Architecture 55
5.5.1 Design Rationale 55
5.5.2 Compression Pipeline 56
5.5.3 Compression Results 57
5.5.4 Cost Efficiency 59

5.6 Performance Timing 59
5.7 Summary 60

6 Applying Grounding: MCP Integration and Prompt Architecture 62
6.1 The Grounding Problem in UAC 62
6.2 MCP Integration: Architecture 63
6.2.1 Tools: Structured Action Execution 64
6.2.2 Resources: Contextual Knowledge Injection 65
6.2.3 Prompts: Separated Expert Guidance 67
6.2.4 Maintainability and Reusability 67
6.3 Progressive Disclosure in Practice 69
6.3.1 Lazy Skill Loading 69
6.3.2 Resolving the Context Bloat Problem 70
6.4 The Hybrid Integration Challenge 71
6.5 The Skill Abstraction Boundary 72
6.6 Summary 73

7 Evaluation 75
7.1 Grounding Reliability 76
7.2 Context Isolation 77
7.3 Maintainability 78

8 Conclusions and Future Work 80
8.1 Summary 80
8.2 Limitations 83
8.3 Customer Outreach: A Dissemination Exercise 85
8.4 Future Directions 86

References 89

LIST OF FIGURES

2.1 The autonomy spectrum of agentic constructs. The key differentiator is who controls the execution flow, not the sophistication of the underlying model. Adapted from [1]. 17
2.2 Three orchestration patterns of practical relevance: tool-centric orchestration, hierarchical orchestration, and its Router-Skill specialization, which uses lightweight metadata routing and progressive disclosure. Adapted from [1]. 21

3.1 The SAP Cloud ALM Unified Analytics Overview Capability interface. The main panel displays an analytical dashboard composed of monitoring cards; the right panel shows the Joule AI assistant through which users interact with the system in natural language to create and modify dashboards. 29

4.1 Two-tier tool architecture of the dashboard agent, separating primary composition operations from secondary lookup capabilities. 38
4.2 The Router-Skill orchestration pattern as implemented in the CALM Orchestration infrastructure. The Router performs intent-based routing; the Skill executes in an isolated context; the MCP Server supplies tools, prompts, and resources on demand. 39

5.1 The dual-LLM compression pipeline. Gemini 2.5 Flash handles context compression; GPT-4.1 handles domain reasoning. Inputs are session history and toolkit responses; output is the classification and action result. 57

LIST OF TABLES

5.1 Token consumption for the `ProviderDefinitions` endpoint across response formats. Baseline is the legacy `getDefinition` endpoint. . . 52
5.2 End-to-end token consumption per query: new API implementation vs. DEV Bot baseline. Figures in thousands of tokens (tkn). Rows marked † reflect the residual Execution Trap case discussed in Section 5.4, not a regression introduced by the API redesign itself. 53
5.3 Token consumption for the `ProviderDefinitions` toolkit response before and after processing by the Gemini 2.5 Flash `compression_agent`. 58
5.4 Response time comparison (seconds) for selected queries. 60

Part I

Foundations

Chapter 1

Introduction

Software systems incorporating artificial intelligence have shifted away from isolated predictive models such as classifiers, rankers, and recommendation engines toward *agentic* systems: architectures in which large language models (LLMs) do not merely respond to a single prompt but plan, act, call external tools, and iteratively refine their outputs in pursuit of a goal, interleaving reasoning with action in the pattern demonstrated by ReAct [2] and surveyed more broadly for augmented language models by Mialon et al. [3]. This shift has been rapid and, in many respects, has outpaced the engineering discipline needed to deploy such systems reliably. Teams build multi-agent pipelines and discover, often in production, that system cost and reliability are determined far more by design decisions made early than by the capabilities of the underlying models: how agents are structured, how they share context, and how they retrieve domain knowledge.

Of these three decisions, this thesis treats context management as the more fundamental of the three, for a structural reason rather than a matter of emphasis: agent structure and knowledge retrieval are both implemented on top of the same finite resource, the context window, so a bad decision in either one still shows up as a context-window failure downstream. Chapter 2 (Section 2.3) develops this argument in full; here it motivates why the case study that follows is organized around token consumption as its central metric. The context window is central to these

challenges because every LLM-based agent perceives its world exclusively through a finite, shared token buffer. In a single-turn chatbot this constraint is manageable. In a multi-agent system handling complex, multi-step workflows over a growing conversation history, it becomes the primary engineering bottleneck: poorly designed APIs flood the context with redundant data, domain knowledge injected wholesale into every prompt adds further weight regardless of whether it is relevant to the request at hand, and agent structures with no mechanism for containing a reasoning failure let that failure propagate rather than staying contained [4]. Left unaddressed, these three problems compound with every additional turn of the conversation.

This thesis addresses these challenges through a concrete industrial case study rather than a synthetic benchmark. That choice of method is deliberate: the failure modes above are a function of sustained production load and real usage patterns, and a controlled benchmark constructed after the fact is not designed to reproduce the conditions that surfaced them in the first place. **SAP Cloud ALM** is an enterprise application lifecycle management platform serving organizations running large SAP landscapes. Its **Unified Analytics Overview Capability (UAC)** allows users to compose and manage operational monitoring dashboards: structured views over service health, exception rates, response times, and other operational metrics. The platform integrates with **Joule**, SAP's generative AI assistant, with the goal of enabling users to create and configure these dashboards through natural language alone.

The Joule integration for UAC is a multi-agent LLM system: an orchestrator (`unified_agent`) classifies and routes natural language requests to two domain-specific sub-agents (in the sense formalized in Chapter 2—reasoning modules that plan and call tools but whose lifecycle is owned by the orchestrator, not autonomous agents in their own right) that invoke backend APIs, reason over a metric catalog,

and compose valid dashboard configurations. When this work began, the system suffered from precisely the failure modes described above: a legacy metric discovery API returning payloads of approximately 219,000 tokens per query; domain knowledge statically embedded, in full, inside each sub-agent's own system prompt rather than externalized to a shared, queryable source, regardless of request complexity; and sub-agents that were self-contained, monolithic units—each bundling its reasoning logic together with its domain knowledge—rather than the isolated, independently loadable Skills the broader SAP platform's orchestration infrastructure was converging on.

Contributions

This thesis makes four technical contributions. The first three each resolve one of the three failure modes above, one apiece, though not in the order in which the failure modes were introduced: Contribution 1 addresses the oversized legacy API payload, Contribution 2 addresses the mismatch between the original agent structures and the platform's orchestration pattern, and Contribution 3 addresses the statically embedded domain knowledge. The fourth contribution is a complementary strategy that targets the same token-bottleneck problem as the first, at the session level rather than the endpoint level:

1. **API and data layer redesign.** The legacy `MetricDefinitions` endpoint was replaced by a restructured `ProviderDefinitions` endpoint that maximizes information density per LLM execution step (Section 2.3 defines this metric). This work identified and characterized the *Execution Trap*: the counter-intuitive phenomenon, empirically observed in this system and not previously named in the chunking/pagination literature this thesis is aware

of, whereby splitting API calls to reduce per-call payload size increases total token consumption by forcing additional reasoning steps (Chapter 5 presents the full empirical case). Token reductions of 75.6–85.6% were achieved at the endpoint level, and 48–73% end-to-end across the four representative discovery queries benchmarked in Chapter 5; the end-to-end range is wider than the endpoint-level range because the achievable reduction scales with how much redundant metadata the legacy endpoint happened to carry for each query's specific scope, not with the query's category alone.

2. **Hierarchical Router-Skill architecture.** The self-contained, monolithic sub-agents described above—each bundling its reasoning logic together with its domain knowledge—were redesigned as isolated, domain-specific Skills operating under the CALM Orchestration Agent, a platform-standard Router-Skill pattern that enforces context isolation and centralized control without the coordination overhead of fully decentralized topologies.

3. **MCP-based grounding and progressive disclosure.** Domain knowledge was migrated from static system prompts to a structured CALM MCP Server, decomposed into focused Tools, contextual Resources, and separated Domain Expert and API Efficiency Prompts. Full Skill definitions and domain resources are loaded strictly on demand, keeping upfront context overhead minimal.

4. **Dual-LLM compression architecture.** A proposed pipeline pairing Gemini 2.5 Flash for context compression with GPT-4.1 for domain reasoning. Of its two stages, toolkit-response compression was implemented and empirically validated, reducing the tested `ProviderDefinitions` response by 61.4%; session-history compression, designed around the sliding-window compaction

mechanism documented for Google ADK and referred to informally in this thesis as *tail-retention*, and its integration with the GPT-4.1 reasoning agent remain proposed future work.

A fifth, supporting contribution, a `status`/`instance_id` versioning schema for dashboards (Section 4.5), was also designed and implemented to support the draft/publish lifecycle required by enterprise users. This data-layer improvement is self-contained and not directly tied to the context-window economics narrative that organizes the four contributions above.

Thesis Structure

The thesis is organized in two parts. **Part I** provides the background. Chapter 2 surveys the four bodies of theory underpinning this work: multi-agent design patterns and the Router-Skill taxonomy, grounding techniques for LLM reliability, context window economics, and the Model Context Protocol as an architectural standard.

Part II presents the case study. Chapter 3 introduces SAP Cloud ALM, the UAC system, and the technical problems motivating the work. Chapter 4 documents the evolution of the multi-agent architecture toward the Router-Skill pattern, including the dashboard versioning data model. Chapter 5 presents the API redesign, the Execution Trap, and the dual-LLM compression pipeline. Chapter 6 covers the MCP integration and prompt architecture. Chapter 7 provides a qualitative architectural evaluation across grounding reliability, context isolation, and maintainability. Chapter 8 concludes with a summary of results, the customer outreach workshop, and directions for future work.

Chapter 2

Background and Related Work

This chapter establishes the conceptual vocabulary and technical context required for the case study in Part II. Four bodies of work are relevant: the taxonomy of agentic constructs and orchestration patterns, grounding techniques for LLM reliability, context window economics, and the Model Context Protocol as an architectural standard.

2.1 Multi-Agent System Design Patterns

Multi-agent systems have been studied since the 1990s, long before large language models existed. Early work established foundational concepts: the BDI (Belief-Desire-Intention) model of rational agency [5], the Contract Net protocol for task allocation across distributed agents [6], and agent-based software engineering as a distinct discipline [7]. Comprehensive surveys of the field [8, 9] identified coordination, communication, and control as the recurring core challenges—the same challenges that re-emerge in LLM-based multi-agent architectures today.

When large language models began to be composed into multi-agent pipelines, this classical vocabulary did not disappear; it reappeared under new names. HuggingGPT [10] orchestrated a set of specialized models through a central controller, and OpenAgents [11] treated tool use and multi-agent coordination as a platform concern, not a one-off implementation detail. The practitioner taxonomy adopted in

this thesis [1] is best read as a recent synthesis of this longer lineage, not a departure from it.

Other frameworks for composing LLMs into agentic systems—LangGraph, AutoGen, CrewAI—each define *agent* differently, and no consensus taxonomy has settled across the field yet. This thesis adopts Piskala's framing [1], which treats **control over execution**, not intelligence, as the critical design axis: the question that resolves most architectural debates is simply *who decides what happens next?* The framing is adopted here because it maps directly onto the Router-Skill pattern used in the case study (Chapter 4), not because it is the only viable taxonomy. It gives rise to a precise taxonomy of four constructs arranged along an autonomy spectrum.

- A **tool** is invoked with structured inputs, executes a bounded operation, and returns an output; it owns no control flow.
- A **skill** carries embedded procedural knowledge that an agent loads on demand into its own isolated context; it makes no decisions—it encodes procedures, not judgments.
- A **sub-agent** is a reasoning module invoked with a goal and constraints; it may plan and call tools, but its lifecycle is owned by a parent.
- An **agent** closes the control loop: it owns its goal, maintains state over time, and decides when to act and when to stop [1].

When an agent takes on the specific, top-level role of receiving a request and routing it to the appropriate downstream construct, this thesis calls that role the **orchestrator**—a name for a role an agent occupies, not a fifth construct on the spectrum above.

The practical design principle is to use the least autonomous abstraction that satisfies the requirement. A first diagnostic question separates the passive constructs from the active ones: if the component were replaced with a structured function call, would the observable behavior change? If not, a tool or skill suffices; if yes, a sub-agent or agent's added autonomy is doing real work. Two further questions resolve each pair. Between tool and skill: does the component need embedded procedural or domain knowledge to do its job, or only the caller-supplied inputs? A skill needs the former; a tool needs only the latter. Between sub-agent and agent: who decides when the work is finished—a parent that invoked it for a bounded goal, or the construct itself, persisting and re-evaluating its own goal over time? The former is a sub-agent, the latter an agent. The first of these three questions is the one applied explicitly in the case study (Chapter 4), where it resolves a skill-versus-sub-agent decision; the other two exist here to make the full four-way taxonomy operational, not because the case study needs them.

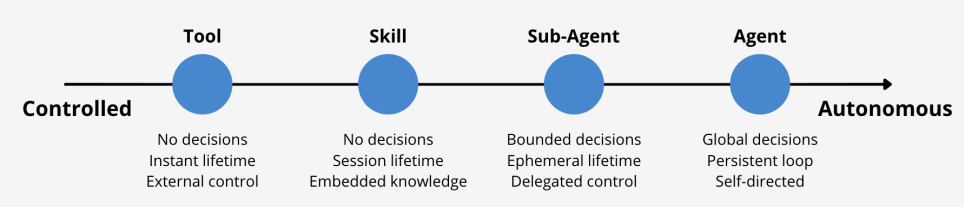


Figure 2.1: The autonomy spectrum of agentic constructs. The key differentiator is who controls the execution flow, not the sophistication of the underlying model. Adapted from [1].

This taxonomy gives rise to three orchestration patterns of practical relevance here: tool-centric orchestration, hierarchical orchestration, and a further specialization of the hierarchical form, the Router-Skill pattern.

- In **tool-centric** orchestration, a single reasoning loop invokes tools sequentially with no delegation—maximum predictability, suitable for compliance-

critical workflows.

- In **hierarchical** orchestration, a controlling orchestrator delegates bounded reasoning to domain-specific sub-agents; control remains centralized while cognition is modular. Piskala [1] identifies this as the sweet spot for most production enterprise systems, a claim made over the full space of orchestration patterns. Kim et al. [4] report a narrower, related finding that bears specifically on the hierarchical-versus-decentralized comparison—it says nothing about tool-centric or Router-Skill orchestration: in their benchmarks, fully **decentralized** multi-agent topologies degraded performance by 39–70% on sequential enterprise tasks compared to hierarchical baselines, with reasoning errors amplifying by a factor of 17.2× across independent agents. Neither source is peer-reviewed; both figures are reported here as motivating evidence for the hierarchical choice made in this case study, not as an established ranking across the three patterns above (Kim et al.'s decentralized topology is a comparison baseline within their study, not a fourth entry in this thesis's taxonomy of orchestration patterns).

- Central to this thesis, the **Router-Skill pattern** specializes hierarchical orchestration further: an orchestrator holds only lightweight (Piskala reports approximately 100-byte) metadata summaries of available skills and routes requests to isolated, domain-specific skills on demand. Full skill definitions are loaded strictly at activation time—an approach termed *progressive disclosure*. Piskala reports this reducing baseline context overhead from 15–20% to approximately 0.6% of the context window in his own benchmarks [1]; this thesis does not independently reproduce that measurement, and the figure should be read as motivating evidence for the pattern rather than a validated property of

this case study. In multi-skill workflows, the orchestrator coordinates between skills through structured output handoffs—each skill returns a typed result object, never its intermediate reasoning tokens—instead of sharing raw input, a design intended to prevent cross-domain reasoning contamination at context boundary points even when multiple skills are called during one session.

The choice among the three patterns is not stylistic. Tool-centric orchestration is the right default when every step must remain independently auditable; hierarchical orchestration is preferable once the task decomposes into semi-independent subdomains, as in the case study in Part II; the Router-Skill variant is worth its added infrastructure only once the number of subdomains is large enough that even lightweight orchestrator-side metadata becomes a scaling cost.

2.2 Grounding Techniques for LLM Reliability

The taxonomy above answers who decides what happens next; it says nothing about whether that decision is correct. An agent or skill at any point on the autonomy spectrum can still act on knowledge that is wrong, stale, or fabricated—the concern this section addresses.

LLMs generate outputs that are statistically plausible given their training distribution, not necessarily accurate with respect to the specific application context. *Grounding* is the engineering discipline of anchoring model outputs to verifiable external data [3]. The problem predates language models by decades: Harnad's symbol grounding problem [12] identified the same structural gap in the context of symbolic AI—a system that manipulates symbols with no causal connection to what those symbols denote cannot guarantee that its outputs track reality. Ungrounded LLM systems confabulate in the present-day form of this gap [13]: they reference

API fields that do not exist, produce status codes that were deprecated, and invent metric names that were never in the catalog.

Three grounding techniques are relevant to this thesis.

- **Prompt engineering** establishes the semantic frame before any user input arrives; effective prompts enforce domain separation (each reasoning component receives only knowledge relevant to its task) and apply progressive disclosure to avoid instruction degradation in long contexts [14].

- **Schema enforcement** via constrained decoding [15] provides structural grounding—guaranteeing that outputs conform to a JSON schema—but does not prevent semantically plausible yet factually incorrect content; it must be combined with semantic grounding.

- **Direct context injection** populates the prompt with authoritative, live system state assembled programmatically at request time. Unlike Retrieval-Augmented Generation (RAG) [16]—which searches a pre-indexed document corpus via semantic similarity and is suited to large static knowledge bases—context injection is appropriate when the grounding source is live, structured, session-specific state where exact correctness is required. The Unified Analytics Overview Capability (UAC) system examined in Part II relies exclusively on context injection via the MCP server: its domain knowledge—metric catalogs, validation rules, API schemas—is comparatively small, changes with the backend and not with document authorship, and must be exactly current rather than merely similar to a query. RAG is built for the opposite case, large and relatively static natural-language corpora, and is not applicable to its domain; Chapter 6 returns to this distinction with the system's actual data.

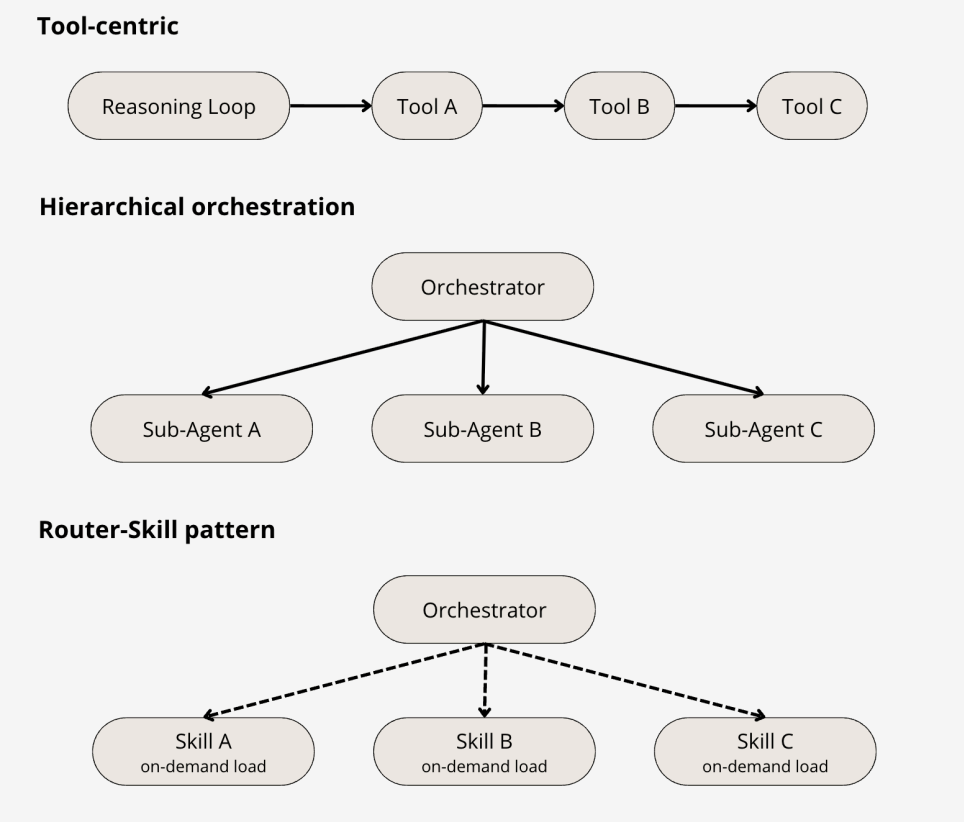


Figure 2.2: Three orchestration patterns of practical relevance: tool-centric orchestration, hierarchical orchestration, and its Router-Skill specialization, which uses lightweight metadata routing and progressive disclosure. Adapted from [1].

2.3 Context Window Economics

Grounding keeps an agent's reasoning tethered to correct knowledge; it does not address what that correctness costs. The three grounding techniques above interact with that cost differently: prompt engineering and direct context injection both add tokens to the context window before generation begins, while schema enforcement adds none to the prompt, instead constraining the output token distribution during decoding. The case study in Part II uses both prompt engineering and context injection, but not for the same purpose: prompt engineering frames the sub-agent's task and separates domains within the prompt, while context injection—as Section 2.2

states, the case study's sole *grounding* mechanism—is what supplies the live, authoritative knowledge. Both nonetheless consume context budget directly, which is why token economics is the second engineering constraint this thesis treats as a first-class design concern.

The context window is simultaneously a memory boundary, a reasoning budget, and a cost driver [14]. In multi-turn agentic systems its management is non-trivial: the full conversation history is re-sent to the model on every turn, so tool responses introduced early in a session—often thousands of tokens each—are re-transmitted on every subsequent turn regardless of continued relevance. Token consumption compounds with session length.

Two principles govern effective context management. First, **information density per execution step**, defined here as the number of tokens that materially inform the model's next decision divided by the total tokens retrieved for that reasoning step: the correct optimization target is not the size of individual API responses but this ratio, maximized per step rather than minimized per call. A counterintuitive failure mode identified in this work—the *Execution Trap*—arises when a large API payload is decomposed into multiple smaller calls to reduce per-call size. Because the model re-processes the full history at every step, the additional tool calls add more turns to the history than they save in per-call tokens, causing total session consumption to increase. The correct resolution is to maximize data density per call, not minimize call size; Chapter 5 documents the empirical case that surfaced this failure mode.

Second, **response format selection**: the same structured data serialized in different formats can carry substantially different token costs, depending on schema nesting depth and field-name verbosity. Elnashar et al. [17] compare several serialization formats for LLM prompts and report, qualitatively, that JSON is the

most verbose of the formats they test and that a hybrid CSV/prefix format is the most token-efficient; they do not report a specific multiplier, so none is repeated here. CSV eliminates per-row field-name repetition and achieves the highest density for homogeneous tabular data; JSON is preferable for hierarchical structures where nesting carries semantic meaning. Chapter 5 reports this thesis's own case-specific comparison between the two formats (Table 5.1). A complementary strategy is **hybrid model delegation**: routing token-heavy sub-tasks whose failure mode tolerates some quality loss—summarization, context compression—to a model that is cheaper and faster than the primary reasoning model, reserving that primary model for planning and composition, where a mistake is more consequential [18].

2.4 The Model Context Protocol

The Model Context Protocol (MCP), introduced by Anthropic in late 2024 [19], defines a standardized client-server interface through which language models interact with external systems. Domain knowledge, tool definitions, and API guidelines reside in the MCP server, not the agent prompt, loaded into context only when needed. This separates the agent's reasoning layer from the domain knowledge layer.

An MCP server exposes three component types.

- **Tools** are action-oriented operations that return structured JSON; they provide capability without embedded domain reasoning.
- **Resources** are contextual artefacts fetched by URI—API specifications, enumeration lists, schema definitions—appropriate for information too large or too domain-specific to embed statically in a system prompt.
- **Prompts** are reusable instruction templates encoding domain expertise; best

practice separates *Domain Expert Prompts* (workflow rules, business constraints) from *API Efficiency Prompts* (selective querying, over-fetch prevention) so each can be updated independently.

Within a hierarchical architecture, MCP servers sit below the skill layer. The orchestrator holds lightweight skill metadata; an activated skill discovers its tools and resources through the MCP server on demand. This realizes the three-stage progressive disclosure model at the architectural level: metadata at discovery, skill instructions at activation, MCP resources at execution. Domain knowledge thereby becomes a deployable, independently versioned artefact: for additive changes—adding a metric, updating a code list, revising a business rule—updating the MCP server requires no agent redeployment. Changes that break a resource's existing contract still require coordinated updates on the agent side, the same as any interface change in a client-server system. For existing systems, MCP adoption is additive—existing tool implementations can be wrapped as MCP tools and existing prompt content decomposed into the Tools/Resources/Prompts structure without discarding working implementations [19].

Individually, these four bodies of work address different failures: autonomy with no mechanism to contain it, reasoning with no check against fact, token growth with no budget enforcing a limit, and domain knowledge with no single authoritative home. In a production multi-agent system, though, these failures are not separable: an orchestration pattern chosen without regard for context cost risks violating the token budget once real usage load replaces the design-time assumptions that motivated the choice, and a grounding strategy that ignores the protocol boundary between agent and knowledge source risks re-creating the maintenance problem it was meant to solve. Part II applies all four together to a single industrial system; the failure modes introduced in Chapter 3 and the architectural responses

in Chapters 4–6 each draw on one or more of the four sections above.

Part II

Case Study: SAP Cloud ALM

Unified Analytics Capability

Chapter 3

System Context and Problem Statement

This chapter grounds the four theoretical frameworks of Part I (multi-agent design patterns, grounding techniques, context window economics, and the Model Context Protocol) in a concrete industrial system: the Unified Analytics Overview Capability (UAC) for SAP Cloud ALM, SAP's enterprise application lifecycle management platform. It describes the platform, the capability, the AI assistant that powers it, and the technical challenges that motivated the architectural decisions examined in the chapters that follow.

3.1 SAP Cloud ALM

SAP Cloud ALM is SAP's cloud-native application lifecycle management platform, designed to support the implementation, operation, and monitoring of SAP business applications across enterprise environments [20]. The platform serves two primary functional domains: **Operations**, which covers real-time monitoring of system health, service performance, and exception management; and **Projects**, which covers structured implementation workflows, task tracking, and deployment coordination. Within the Operations domain, a central challenge for enterprise users is the aggregation and interpretation of operational data across large, heterogeneous SAP landscapes. A single organization's SAP landscape can span many connected systems and services, each exposing its own catalog of metrics: telemetry such as

response times, exception rates, service availability, and user activity. Aggregated across every service into a single unfiltered listing, this catalog is large enough that the response reaches a size that becomes the central cost problem of this thesis (Section 3.4.1). Interpreting it and mapping it onto a monitoring intent requires knowledge the platform's raw catalog does not itself organize around a user's goals. The Unified Analytics Overview Capability (UAC) is SAP Cloud ALM's response to this challenge. UAC provides users with a unified, configurable analytics layer over the operational data of their SAP landscape. Its central artifact is the **analytical dashboard**: a structured composition of monitoring cards, each representing a focused view of a specific operational metric or set of metrics.

3.2 Joule: SAP's AI Assistant

SAP Cloud ALM integrates with **Joule**, SAP's generative AI assistant, which SAP positions as the AI layer across its broader product portfolio [21]; only its role within UAC is examined in this thesis. Within the context of UAC, Joule provides a natural language interface through which users can create and modify analytical dashboards without requiring deep knowledge of the underlying metric catalog or dashboard configuration schema. The practical goal requires closing the gap between free-form natural language and a structured, schema-constrained dashboard configuration: a user should be able to express an intent in natural language (for example, *"Create a dashboard showing response times and exception rates for my critical services over the last seven days"*) and receive a correctly configured, populated dashboard as a result. This requires the AI system to understand the user's intent, map that intent to available metrics in the catalog, construct the appropriate dashboard structure, and invoke the necessary backend APIs to persist the result. Joule's interaction

with UAC is mediated by a multi-agent orchestration system that decomposes the task across specialized agents, each responsible for a bounded scope of the overall process. The architecture of this system, and the challenges encountered in building it, constitute the primary subject of this thesis.

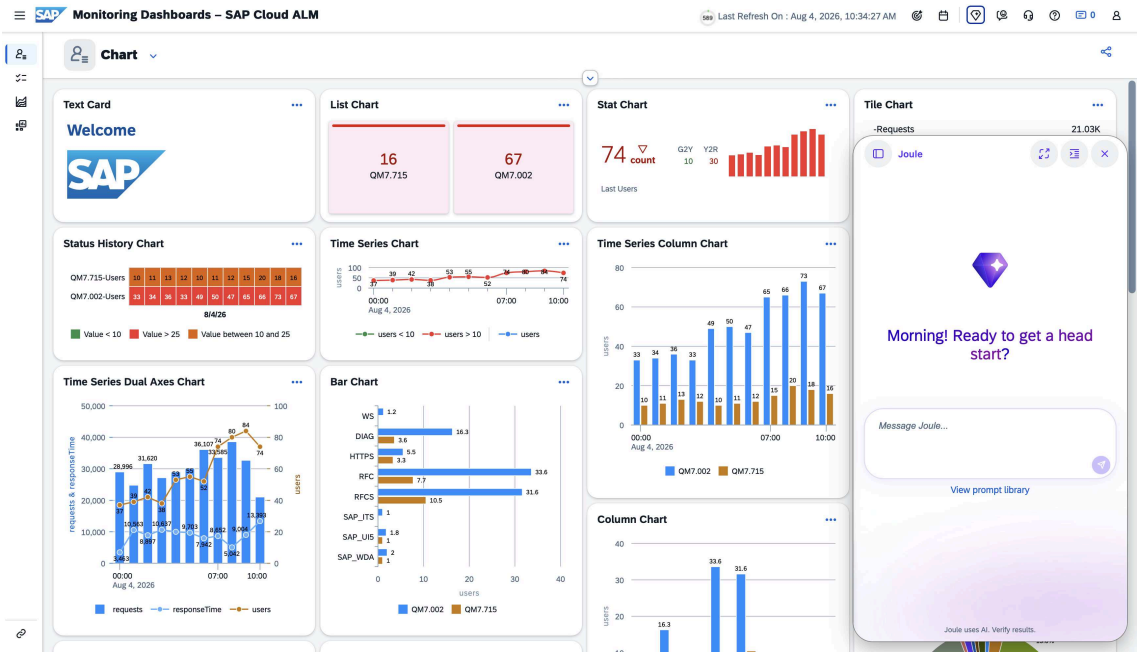


Figure 3.1: The SAP Cloud ALM Unified Analytics Overview Capability interface. The main panel displays an analytical dashboard composed of monitoring cards; the right panel shows the Joule AI assistant through which users interact with the system in natural language to create and modify dashboards.

3.3 The Multi-Agent Orchestration System

At the time this internship began, the Joule integration for UAC was structured around a central orchestrator, the `unified_agent`, responsible for classifying incoming user requests and routing them to the appropriate domain-specific sub-agent. In the taxonomy of Chapter 2, `unified_agent` is the only one of the three components that qualifies as an agent in the strict sense, since it owns the routing decision

and persists across the interaction; the two components it routes to are sub-agents (reasoning modules whose lifecycle it owns), and this thesis refers to them as such throughout. The two domain-specific sub-agents were:

- `joule_card_agent`: responsible for the creation and modification of individual monitoring cards (the dashboard's atomic unit, each configured with a time range, aggregation function, and display parameters) within the Operations domain.

- `dashboard_agent`: responsible for the composition of full dashboards: assembling cards into coherent analytical views and managing the dashboard lifecycle.

Each sub-agent invokes a set of backend tools (REST API endpoints exposed by the UAC backend) to discover metrics, retrieve catalog data, and persist configurations, using LLMs that translate natural language requests into structured API calls and compose valid dashboard configurations. This architecture reflects the hierarchical orchestration pattern discussed in Chapter 2: a controlling orchestrator delegates bounded reasoning tasks to domain-specific sub-agents, which in turn invoke deterministic tools, with control centralized in the orchestrator and cognition distributed across the sub-agents. The *control topology*, an orchestrator delegating bounded tasks to domain-scoped sub-agents, was sound on its own terms: none of the four problems in Section 3.4 required abandoning it. What was unsound was each sub-agent's *prompt content*: domain knowledge injected statically and in full, regardless of topology (Sections 3.4.1–3.4.3). Those problems would have occurred in a single-agent system just as easily, yet this chapter keeps the multi-agent frame because the eventual fix (Chapter 4) came bundled with a topology change too. A fifth problem sits at the topology level: partway through this work, SAP's orchestra-

tion infrastructure adopted the platform-standard Router-Skill pattern, which the existing agents, though functionally correct, were not built to conform to. Objective 1 (Section 3.6) addresses that mismatch, unrelated to any flaw in the original design.

3.4 The Problem: Operational Inefficiency at Scale

Three of the four problems below concern context window economics; the fourth concerns the data model. These inefficiencies were concrete enough to be measured directly: a multi-turn interaction could accumulate on the order of 600,000 tokens of re-processed metric data alone (Section 3.4.1; Chapter 5 quantifies the resulting cost and latency impact). At that volume, token cost and latency scale with conversation length in a way that threatens both per-session cost and production responsiveness, with the data-model concern (the dashboard lifecycle could not support a draft/publish workflow at all, independent of token cost) addressed separately in Section 3.5.

3.4.1 Token Consumption at the Data Layer

The `joule_card_agent`'s primary workflow requires discovering what operational metrics are available before it can compose a monitoring card. This relied on a legacy API endpoint, `MetricDefinitions`, that returned the full schema (filters, aggregations, parameters, provider metadata) for every available metric in one unfiltered response (Chapter 5 itemizes the response structure). In practice, this endpoint returned payloads of approximately 219,000 tokens for the query *"Show me all available metrics"*: the benchmark case measured in Chapter 5, though neither chapter documents the exact tokenizer used, so the figure should be read as an

order-of-magnitude estimate, without claiming precise reproducibility. Since LLMs re-process the entire conversation history on every turn, this payload was reintroduced at every reasoning step; for a multi-turn interaction, the cumulative cost translated directly into higher API costs and degraded latency; the redesign in Chapter 5 later reduced measured end-to-end response time on this same query from approximately 50 to 40 seconds.

3.4.2 Context Window Overload and the Execution Trap

An intuitive fix to the oversized `MetricDefinitions` payload (splitting it into a lightweight listing call and a separate on-demand full-detail call) was implemented and tested (exposed to the sub-agent as `list_available_metrics` and `discover_available_metrics`), but it revealed a deeper architectural constraint. Because the LLM re-processes the entire conversation history at every step, adding execution steps increases total context consumed, the opposite of the intended reduction: each additional call adds its own tokens plus the re-processing cost of everything before it. This phenomenon, splitting calls to shrink payloads while increasing total consumption, is termed here the **Execution Trap**, and is directly measurable: on queries routed through `discover_available_metrics`, the split-endpoint implementation consumed 405,548 and 225,285 tokens against DEV Bot baselines of 120,229 and 120,543, increases of 237% and 87% (Chapter 5 presents the full benchmark). The practical implication is that the optimization target is *total information retrieved per execution step*: maximize *data density*, the ratio of task-relevant tokens returned to total tokens consumed in context, including re-processing cost on subsequent turns, per step, over minimizing call size. Chapter 5 documents the comparison and the redesign that followed.

3.4.3 Prompt Architecture and Context Bloat

Beyond the data retrieval layer, the sub-agents' prompts themselves contributed to context inefficiency: domain knowledge (metric taxonomies, API usage guidelines, field validation rules) was embedded directly and statically in each sub-agent's system prompt, so every invocation loaded the full corpus of domain expertise upfront, regardless of the request's complexity. This follows structurally from the static-injection design: each new domain adds its own block of statically injected content, so prompt size scales with the number of supported domains by construction, with every block present regardless of which domain a request concerns. At the two domains implemented here (cards and dashboards) this was already measurable overhead, and nothing removes a block once added, so it does not plateau as domains grow.

3.5 Data Lifecycle Management

A further challenge, distinct from the token-efficiency concerns above, arose from dashboard lifecycle management. Enterprise users need to edit a dashboard privately, then either publish those changes or discard them and revert to the last published state: three explicit states and transitions (draft, published, discarded). The initial database schema for dashboards and cards used a single static primary key (`Id`) with no concept of versioning state. This design made it impossible to maintain multiple lifecycle states for the same logical entity: a published version and a draft under modification could not coexist. This limitation was also a structural, design-level safety gap: without a draft state, every sub-agent write applied directly to the published record, with no intermediate state to review and no path to recover the prior version.

3.6 Research Objectives

The five problems identified in this chapter, the topology mismatch (Section 3.3), the three context-window-economics problems (Section 3.4), and the data-lifecycle gap (Section 3.5), map onto four technical threads plus a supporting data-layer contribution. Objective 1 closes the topology mismatch. Of the three context-window problems, Token Consumption and the Execution Trap are closed jointly by Objectives 2 and 4; Prompt Architecture and Context Bloat is closed separately by Objective 3. The supporting objective closes the lifecycle gap.

1. **Hierarchical Architecture Evolution** (Chapter 4): Adapt the agent architecture to the Router-Skill orchestration pattern (defined in Section 2.1) introduced by the CALM Orchestration infrastructure.

2. **API and Data Layer Optimization** (Chapter 5): Redesign the backend data retrieval layer to maximize information density per LLM execution step, eliminating the Execution Trap and reducing token consumption.

3. **Grounding via MCP and Prompt Architecture** (Chapter 6): Offload domain knowledge from static system prompts to an MCP server (whose Tools, Resources, and Prompts components are defined in Section 2.4), applying progressive disclosure to achieve on-demand grounding without upfront context bloat.

4. **Context Compression via a Dual-LLM Pipeline** (also Chapter 5, alongside Objective 2, as a complementary strategy on the same token-bottleneck problem): Design a compression architecture pairing a lightweight model with the primary reasoning model to reduce redundant context across a session; implement and validate the toolkit-response compression stage.

A supporting objective, designing a dashboard versioning data model that supports the full draft/publish/discard lifecycle without destructively overwriting prior states (Section 4.5), addresses a related but architecturally distinct data-layer requirement.

Chapter 4

Applying Hierarchical Orchestration: The Multi-Agent Architecture

Chapter 2 established that hierarchical orchestration, a controlling orchestrator delegating bounded reasoning to domain-specific sub-agents, is what Piskala terms the *"sweet spot"* for most production enterprise AI systems [1]. This chapter examines how that pattern was applied and evolved within the SAP Cloud ALM Unified Analytics Capability. The architecture did not begin in its final form; it underwent a structural transition driven by changes in the broader SAP platform infrastructure, external to any shortcoming in the initial design. Documenting this evolution, including the constraints that forced it and the tradeoffs it entailed, is the primary contribution of this chapter.

A note on the evidence behind the claims that follow: `joule_card_agent` ran in production as a Skill, so claims about it reflect the running system; `dashboard_agent` was designed but not implemented, so claims about it remain design judgments, unconfirmed by measurement, and this chapter does not otherwise benchmark invocation counts, fault-inject, or measure coordination overhead. Both components keep their original `_agent` names despite being reclassified as Skills here; the suffix predates this thesis's taxonomy.

4.1 The Initial Architecture

As described in Chapter 3, the starting architecture organized the Joule integration around three hierarchically arranged components: the `unified_agent` as orchestrator, and two domain agents (`joule_card_agent` and `dashboard_agent`), each owning their respective tools and reasoning scope. This arrangement mapped cleanly onto the hierarchical orchestration pattern: the orchestrator owned routing decisions, the domain agents owned domain reasoning, and the backend tools owned deterministic execution. Control never decentralized.

From an architectural analysis standpoint, the design partially implemented progressive disclosure: the orchestrator held only lightweight metadata about downstream agents and was not required to reason about metric schemas or dashboard structure. However, the domain agents themselves still carried all their domain knowledge statically in their system prompts, a limitation that would drive the subsequent evolution described in this chapter.

4.1.1 Tool Architecture of the Dashboard Agent

The `dashboard_agent`'s tool set was organized into two tiers. Primary tools covered the main operations required for dashboard composition: retrieving the card catalog to understand what cards were available, creating a new dashboard entity, and assigning cards to it. Secondary tools, `findByName` and `findById`, provided lookup capabilities when the agent needed to resolve a named dashboard or card to its system identifier before operating on it.

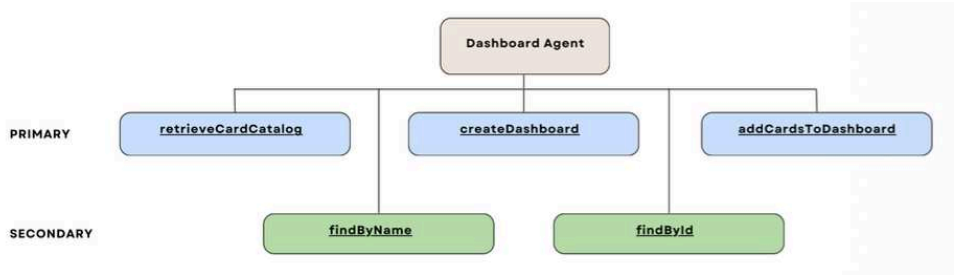


Figure 4.1: Two-tier tool architecture of the dashboard agent, separating primary composition operations from secondary lookup capabilities.

This two-tier structure is a practical expression of the context economy discussed in Chapter 2: the agent can initiate most workflows without exhausting secondary lookup tools, and only invokes them when the primary workflow requires disambiguation. By design, this structure is intended to minimize unnecessary tool invocations and keep the agent's reasoning steps proportional to task complexity.

Nothing in this initial design was diagnosed as broken; the transition documented in the rest of this chapter was driven purely by an external event in the broader SAP platform infrastructure, described next.

4.2 The Architectural Transition: CALM Orchestration

During the April development period, the underlying SAP platform infrastructure underwent an architectural shift. The legacy approach relied on monolithic agent structures where each agent contained both its reasoning logic and all the domain knowledge required to execute its tasks. The new platform introduced the **CALM Orchestration Agent**: a Java-based hierarchical orchestration system that formalized the Router-Skill pattern at the platform level.

4.2.1 The Router-Skill Pattern in Production

Under the new CALM Orchestration architecture, the system's topology changed. The CALM Orchestration Agent is a traffic controller: it does not attempt to execute any domain task directly. Instead, it routes user requests to isolated, domain-specific components called **Skills**, using a combination of keyword matching and LLM-based intent analysis; the fallback behavior for an ambiguous or unmatched request is not documented in this thesis's project material.

Each Skill is a self-contained unit of domain expertise: it has its own isolated context, its own set of tools, and its own connection to the MCP server that holds its domain knowledge. The Skills architecture maps directly onto the *skill* construct defined in Chapter 2: modules containing procedural knowledge that execute procedures without owning decision-making autonomy [1].

Figure 4.2 illustrates this topology: the Router Agent performs LLM-based routing, the selected Skill executes in an isolated context with its own tools, and the MCP Server provides the tools, prompts, and resources that ground the Skill's reasoning.

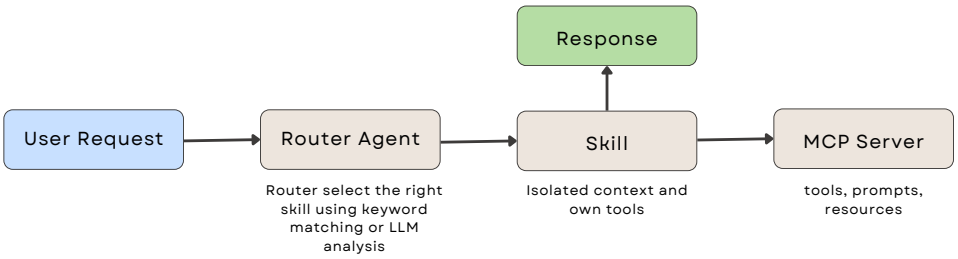


Figure 4.2: The Router-Skill orchestration pattern as implemented in the CALM Orchestration infrastructure. The Router performs intent-based routing; the Skill executes in an isolated context; the MCP Server supplies tools, prompts, and resources on demand.

4.2.2 Adapting the Agents to the Skill Abstraction

This architectural shift required redesigning both the `dashboard_agent` and `joule_card_agent` to conform to the Router-Skill pattern. As already noted, this redesign was fully implemented for `joule_card_agent`; for `dashboard_agent` only the architectural mapping (prompt schema, tool definitions, and state management strategy) was completed, with the code implementation itself pending the backend versioning work described in Section 4.5.

Both agents required the same three categories of change, carried through to implementation for `joule_card_agent` and to design only for `dashboard_agent`:

1. **Prompt schema overhaul**: Domain knowledge previously embedded in the system prompt (metric taxonomies, validation rules, API usage guidelines) was extracted and relocated to the MCP server. The agent's residual system prompt was reduced to execution scaffolding: how to call tools, how to structure responses, and what constraints to respect.

2. **Tool definition restructuring**: Tool definitions were updated to match the CALM Orchestration interface contract. This let the Orchestration Agent invoke each Skill without a translation layer, and it gave tool outputs a consistent structure for downstream consumption.

3. **State management redesign**: Because a Skill operates in an ephemeral, isolated context, it does not persist state between invocations; any state previously maintained across tool calls within the agent had to be managed explicitly within the scope of a single skill activation.

4.2.3 Context Isolation as an Architectural Guarantee

The transition to the Router-Skill pattern enforces context isolation as a structural property of the architecture itself. For `joule_card_agent`, fully implemented as a Skill, this is demonstrated in production: a Skill activated for metric card creation has no access to the context of any other Skill. For `dashboard_agent`, the same guarantee follows from its mapping above, but remains undemonstrated pending implementation. The CALM Orchestration Agent maintains the session-level context and coordinates inter-skill workflows, but each Skill reasons only within its own bounded window.

This isolation has the correctness benefit discussed in Chapter 2: reasoning errors within one skill cannot propagate directly into another. It also has an efficiency benefit: the context window of each Skill is sized for its specific domain task alone, decoupled from total system capability.

4.3 The Control Spectrum in Practice

The taxonomy of constructs developed in Chapter 2 (Tool, Skill, Sub-Agent, Agent) provides a useful lens for evaluating the final architecture. Figure 2.1 from that chapter maps these constructs along the control axis; the UAC implementation can be precisely located within this spectrum. The CALM Orchestration Agent operates closest to the *agent* end of the spectrum: it owns a persistent session loop, maintains state across multiple user turns, and makes global routing decisions. The `joule_card_agent`, fully adapted to the Skill abstraction as part of this work, is a *skill* in the strict sense: it embeds procedural knowledge, executes within a session lifetime, and makes no global decisions. The `dashboard_agent` maps to the same Skill pattern, not yet implemented. The backend API tools that the Skills invoke

are pure *tools*: no decisions, stateless single-invocation lifetime, external control. This layered structure illustrates a key principle from the theoretical literature: appropriate constructs at each layer of the hierarchy yield a system that is both powerful and controllable. The Orchestration Agent's autonomy is bounded by the isolation of the Skills below it; the Skills' reasoning is grounded by the determinism of the tools below them.

4.3.1 Consistency with the Literature

The CALM platform team set the Router-Skill pattern examined here; this thesis adapted two existing agents to it, without having chosen it. What follows shows consistency, short of validation, since the design process was only adapted here and never authored: the resulting architecture matches what [1] identifies as the optimal pattern for production enterprise systems (hierarchical orchestration with domain-specialized sub-agents) on the following properties:

- **Modularity**: each Skill is independently deployable and testable.
- **Specialization**: each Skill's context contains only domain knowledge relevant to its scope.
- **Centralized control**: the Orchestration Agent retains routing and session state; no Skill can unilaterally extend the system's behavior.
- **Bounded failure, by design**: an error within a Skill is intended to affect only that Skill's response, without corrupting the orchestrator's session state or cascading to sibling Skills.

Kim et al. [4] report that decentralized multi-agent topologies introduce coordination

overhead and error amplification; the SAP system avoids decentralized agent-to-agent communication by construction, ruling out the topology their finding concerns.

4.4 Challenges and Resolutions

4.4.1 The Transition Complexity

Transitioning the agent logic to the new Router-Skill architecture required more than a mechanical refactoring. The agent's prompt schemas, tool definitions, and state management had to be entirely overhauled to function as a Skill that relies on an external MCP server for its domain knowledge. Validating that the remodeled Skill communicated correctly with both the Orchestration Agent and the MCP server required a dedicated integration testing phase, and any system undergoing a similar migration should budget for one, since treating the redesign as complete once the code compiles would be premature.

4.4.2 Identifying the Right Abstraction Level

An early design question was whether the `dashboard_agent` should be implemented as a full *sub-agent* (with bounded but genuine reasoning autonomy) or as a *skill*. The decision in favor of the Skill abstraction was guided by the diagnostic test proposed in [1]: if the component were replaced with a structured function call, would the observable behavior change? For the dashboard composition workflow, the expected answer is no: given a correctly populated context (metric catalog, user intent, available tools), the composition process is judged deterministic enough that a skill's embedded procedural knowledge should suffice. Full sub-agent autonomy was rejected on the same reasoning: it would introduce non-determinism and coordination overhead this workflow does not need.

4.5 Data Layer Adjustments: Dashboard Versioning

The Router-Skill migration changes who is allowed to write to production data: `joule_card_agent` already writes as an autonomous Skill, and `dashboard_agent` will once implemented, so every write needs a safety net that does not depend on a person reviewing it first. This applies to both `Dashboards` and `Cards` because a card is not independent of its parent dashboard: a card edited while its dashboard is in draft must not surface in the published version until that draft is published, so the two tables face a single shared versioning problem. The original schema's single static primary key (`Id`) made it impossible for published and draft versions to coexist on either table, and without a draft state an AI agent could make irreversible changes to a live dashboard or card with no way for the user to review or discard them first.

The Adopted Solution

The adopted solution adds two fields to the `Dashboards` and `Cards` tables:

- `status`: an enumeration (`PUBLISHED`, `DRAFT`).
- `instance_id`: a field that groups all versions representing the same logical entity.

Creating a draft copies the published record under the same `instance_id` with `status = DRAFT`. Publishing atomically swaps the statuses; discarding simply deletes the draft record. During a draft editing session, the published record is never modified.

Evaluated Alternatives

Two more complex approaches were evaluated and rejected. Full history versioning (a new row per change, Git-style) provides maximum auditability but introduces version-chain query complexity, increasing the likelihood of agent errors when the LLM constructs dynamic API calls. Separate published/draft tables avoid version chains but require schema duplication and complicate entity identity management.

Design Rationale for AI Reliability

The `status`/`instance_id` approach was chosen because it is *the simplest model that satisfies the actual requirements.* The most common query, retrieving the published dashboard, requires only a plain indexed equality filter (`WHERE instance_id = ? AND status = 'PUBLISHED'`), with no subqueries or joins.

For an AI-managed system, schema simplicity functions as a reliability property: a straightforward schema should reduce malformed read and draft operations. Architectural scoping also limits the agent's tool set strictly to draft operations, making publish and discard safer by requiring explicit user invocation [1].

4.6 Summary

The architecture of the UAC Joule integration evolved from a bespoke hierarchical arrangement toward a platform-standard Router-Skill pattern, driven by the adoption of the CALM Orchestration infrastructure. This evolution reinforced the theoretical prescriptions of Chapter 2: the final system achieves modularity, specialization, and centralized control through a clean separation of routing intelligence (Orchestration Agent), domain expertise (Skills), and deterministic execution (Tools and MCP Server). The data-layer versioning design described in Section 4.5 extends

this principle to the persistence layer: simplicity in the data model directly supports reliability in agent-driven interactions.

Chapter 5

Applying Context Window Economics: API and Data Layer Optimization

Chapter 2 established that the context window is a finite, shared, and expensive resource, and that effective context management requires deliberate architectural decisions at the data retrieval layer. This chapter documents the application of those principles to the `joule_card_agent` within SAP Cloud ALM's Unified Analytics Capability, in two phases: redesigning the API layer to maximize information density per LLM execution step, then introducing a dual-LLM architecture to offload the remaining compression burden from the primary reasoning model. The empirical results—measured in token consumption across a benchmark of representative user queries—provide quantitative evidence for the claims made in the theoretical chapter.

The benchmark compares the new implementation against the **DEV Bot baseline**: the pre-existing legacy AI implementation of the `joule_card_agent` that relied on the original `MetricDefinitions` endpoint and static prompt injection, deployed in the development environment prior to this work. The performance analysis focuses exclusively on the `joule_card_agent` because it was the agent with the most critical API payload bottleneck—the metric discovery problem described in Section 5.1. The `dashboard_agent` operates on a card catalog whose payload characteristics are structurally different and whose Skill adaptation was designed but not

yet fully implemented at the time of benchmarking.

5.1 The Baseline: `MetricDefinitions`

The `joule_card_agent`'s metric discovery workflow relied on the legacy `MetricDefinitions` API endpoint. This endpoint was designed for general-purpose metric inspection by human developers, not for LLM consumption. Its response included, for each available metric, the full schema definition: all available filter dimensions, every supported aggregation function, detailed parameter descriptions, and provider metadata.

For a typical high-level metric listing query— *"Show me all available metrics"* — this endpoint returned approximately 219,000 tokens. The structure of the data was also suboptimal for LLM consumption: metrics were organized by filter type rather than by provider or domain, and redundant filter definitions appeared repeatedly across entries.

5.1.1 The Context Cost at Scale

The scale of this inefficiency becomes apparent when considering multi-turn interactions. A dashboard creation workflow typically involves: (1) a discovery query to understand available metrics, (2) one or more refinement queries as the user narrows their selection, and (3) card creation calls. Because LLMs re-process the entire conversation history on every turn, a 219,000-token metric payload introduced into the context at step 1 reappears in full at steps 2 and 3. Naively re-including that payload at each of the three turns gives an illustrative, back-of-envelope figure of $3 \times 219{,}000 \approx 600{,}000$ tokens from metric data alone—before accounting for system prompts, tool definitions, and reasoning steps. This estimate is not a

measured multi-turn benchmark—none of the queries reported later in this chapter (Table 5.2) are cumulative multi-turn totals—and is included only to motivate, qualitatively, why a single oversized payload becomes disproportionately costly once conversation history re-processing is taken into account.

5.2 The Execution Trap

The initial approach to reducing the metric payload was intuitive: split the `MetricDefinitions` endpoint into two smaller, more targeted endpoints, exposed to the agent as two separate tools. `list_available_metrics` would handle lightweight listing (metric names and short descriptions only); a second tool, `discover_available_metrics`, would handle full schema retrieval on demand, invoked only when the agent needed detailed configuration information for a specific metric.

This decomposition was implemented and tested. The results were counterproductive. Splitting the API into two calls required the agent to execute two sequential tool calls where previously it had executed one. Because each tool call adds an additional reasoning step, and because the LLM re-processes the entire conversation history at every step, the second call's context window includes not only its own prompt and response but also the full content of the first call. The net effect was that total token consumption *increased* relative to the single-call baseline. This is directly visible in the benchmark data (Table 5.2): for queries where the agent invoked `discover_available_metrics` rather than the lightweight `list_available_metrics` call, token consumption came in well above the DEV Bot baseline rather than below it—405.55k vs. 120.23k tokens for "Show details of each metric" (a 237% increase) and 225.29k vs. 120.54k tokens for "Retrieve all metrics with details" (an 87% increase), rather than the reduction seen on queries han-

dled through `list_available_metrics` alone. This two-endpoint split was retained in the deployed toolset as `discover_available_metrics`; Section 5.4 discusses its continued impact once the rest of the API was redesigned.

This experience surfaced a structural constraint with broad applicability, termed here the **Execution Trap**: in LLM systems that maintain full conversation history, reducing the payload size of individual tool calls by adding more calls does not reduce total token consumption—it increases it. The correct optimization target is *information density per execution step*—defined here as the ratio of task-relevant tokens returned by a tool call to the total tokens the call consumes in context (its own payload plus the re-processing cost it imposes on every subsequent turn)—not per-call response size. The goal is to retrieve the maximum useful information in the fewest possible tool calls.

5.3 API Redesign: Maximizing Information Density

The resolution was to keep a single endpoint but completely transform its response structure. The new `ProviderDefinitions` endpoint was designed with two objectives: (1) eliminate all data that the LLM does not need for metric selection, and (2) aggregate the remaining data into the densest possible representation.

5.3.1 Structural Changes

Three structural transformations distinguished the new endpoint from the legacy one:

1. **Provider-level aggregation**: metrics were grouped by provider (the SAP service exposing them) rather than by filter type. This organization aligns with how users think about their landscape—*"metrics for the Integration Suite"*

rather than *"metrics with a filter of type SERVICE_ID"*—and eliminates redundant filter definitions that appeared repeatedly in the legacy format.

2. **Static data removal**: provider metadata and filter data that were identical across all metrics from a given provider—constituting a large portion of the legacy payload—were removed. This data is not required for metric selection; it is required only at card configuration time.

3. **Separation of listing from detail**: the endpoint was designed to return, by default, only the information needed for the discovery step (metric names, descriptions, and provider association). Detailed schema information remained available through a separate `MetricDefinitionDetails` endpoint, invoked only when the agent had already identified the specific metric it intended to configure.

This design is a direct application of the *progressive disclosure* and *lazy loading* principles discussed in Chapter 2: information is loaded at the granularity required by the current execution step, not in anticipation of all possible future steps.

5.3.2 Format Optimization: JSON vs. CSV

Beyond structural changes, the response *format* was evaluated as an independent optimization variable. Chapter 2 noted that structured data formats differ substantially in token efficiency. The `ProviderDefinitions` endpoint was benchmarked in two variants: a restructured JSON format (v1) and a CSV format (v2).

Table 5.1 presents the results. These figures measure the token footprint of the `ProviderDefinitions` endpoint itself—the reduction comes entirely from the API restructuring described above, independently of any compression pipeline. The optimized JSON format (v1) reduced the endpoint's token footprint from the baseline

of 4,337 tokens to 1,056 tokens—a 75.6% reduction. The CSV format (v2) reduced it further to 623 tokens—an 85.6% reduction.

Table 5.1: Token consumption for the `ProviderDefinitions` endpoint across response formats. Baseline is the legacy `getDefinition` endpoint.

Endpoint / Format	**Tokens**	**Reduction vs. Baseline**
`getDefinition` (baseline)	4,337	0%
`getProviderDefinition` (v1 – JSON)	1,056	75.6%
`getProviderDefinition` (v2 – CSV)	623	85.6%

The CSV format achieves greater density by eliminating field-name repetition: in JSON, every record repeats every key; in CSV, keys appear once in the header row. For homogeneous tabular data—a list of metrics with consistent fields—this structural difference yields a meaningful token reduction. The tradeoff is that CSV requires the LLM to infer column semantics from the header rather than from field-level labels, which may introduce parsing fragility for ambiguous field names. For the `ProviderDefinitions` use case, where field names are short and unambiguous, this cost was judged acceptable. Nevertheless, CSV adds operational complexity: it is less widely used in the SAP backend infrastructure, harder to validate, and more fragile to changes in field order. Both formats were benchmarked, and the final format choice reflected an engineering compromise between token efficiency and operational maintainability rather than token efficiency alone.

5.3.3 End-to-End Token Impact

The token savings at the endpoint level compound significantly at the query level. Table 5.2 presents the end-to-end token consumption across a benchmark of ten representative user prompts, comparing the new API implementation against the legacy DEV Bot baseline. Two rows—marked †—show the new implementation

consuming *more* tokens than the baseline; this is not a regression introduced by the redesign but the residual Execution Trap case discussed in Section 5.4, reproduced here because `discover_available_metrics` remained available to the agent after the redesign.

Table 5.2: End-to-end token consumption per query: new API implementation vs. DEV Bot baseline. Figures in thousands of tokens (tkn). Rows marked † reflect the residual Execution Trap case discussed in Section 5.4, not a regression introduced by the API redesign itself.

Test Case	**New Implementation**	**DEV Bot Baseline**
Show me all available metrics	59.22	219.04
Show details of each metric†	405.55	120.23
Retrieve all metrics with details†	225.29	120.54
Retrieve details for the metric users	58.02	126.57
Create a card to show response time	93.48	183.91
What metrics can I put in my card?	143.96	305.06
I want to create a card, tell me available metrics	55.88	208.84
Show me all error-related metrics	62.04	120.15
Show response time for the last 7 days	125.03	181.96
Sum up all exceptions for my services	94.34	269.54

For the four queries served entirely through the lightweight `list_available_metrics` call—"Show me all available metrics," "What metrics can I put in my card?," "I want to create a card, tell me available metrics," and "Show me all error-related metrics"—the token reduction ranged from 48% to 73%, directly reflecting the endpoint-level improvements. The range is wide because the reduction scales with how much redundant filter and provider metadata the legacy `MetricDefinitions`

response carried for that particular query's scope, not with the query's category alone. For queries requiring both discovery and detail retrieval, the reduction was more variable still, reflecting the interaction between the two endpoint calls and the conversation history growth from multi-step reasoning.

5.4 Residual Variability and Limits

The benchmarking revealed one important qualification to the end-to-end results just presented. For queries that required the `discover_available_metrics` tool to return full schema data for all metrics simultaneously—rather than a lightweight listing—the new implementation consumed significantly *more* tokens than the baseline: up to a 237% increase in some cases (Table 5.2, "Show details of each metric," 405.55k vs. 120.23k tokens). This is the same `discover_available_metrics` tool identified in Section 5.2 as the source of the Execution Trap; the finding here is that it remained available to the agent, and therefore capable of triggering the same failure mode, even after the surrounding API was redesigned. Root cause analysis identified data over-fetching as the source: when the agent was instructed to retrieve full details for the entire metric catalog in a single step, the new endpoint returned a payload even larger than the legacy one, because the compression optimizations that worked for listing queries did not apply to exhaustive detail retrieval.

This finding applies the same principle established in Section 5.2 to a different variable: here it is the scope of a single call, rather than the number of calls, that determines whether the response matches the task or over-delivers relative to it. The correct resolution is selective detail retrieval—fetching full details for a specific metric only after it has been identified through listing, rather than in bulk—steered by prompt-level guidance (discussed in Chapter 6) and by API-level design that

discourages over-fetching as a usage pattern; neither the prompt nor the API design alone can be assumed to produce that behavior reliably.

5.5 The Dual-LLM Architecture

The API redesign addressed token consumption at the data retrieval layer. A parallel challenge existed at the context management layer: conversation history and toolkit response data accumulated across multi-turn interactions, even after structural API improvements. To address this, a dual-LLM compression architecture was proposed as a complementary design—intended to be layered on top of the API redesign rather than replacing it.

5.5.1 Design Rationale

The dual-LLM architecture addresses this challenge by introducing a dedicated compression stage before the primary reasoning stage. The design was inspired by the Google ADK (Agent Development Kit) pattern and organized around a clear division of labor:

- **Gemini 2.5 Flash** (compression agent): receives the accumulated session history and raw toolkit responses, and produces a condensed summary retaining only the information relevant to the current user intent.
- **GPT-4.1** (reasoning agent): receives the compressed context and the current user query, and produces the final response and tool invocations.

Neither label places these components on the Tool/Skill/Sub-Agent/Agent spectrum of Chapter 2: both are single-invocation pipeline stages with no persistent goal or control-flow ownership of their own, closer under that taxonomy's strict definitions

to a *tool* than to an *agent*. The informal "compression agent"/"reasoning agent" labels are kept because they are the names used in the project's own design material, not because this thesis classifies either stage as an agent in the taxonomic sense.

This separation exploits the different cost and capability profiles of available models. Gemini 2.5 Flash occupies the lower-cost, lower-intelligence region of the model capability-cost space—appropriate for summarization and data extraction tasks that do not require deep reasoning. GPT-4.1 occupies the higher-cost, higher-intelligence region—appropriate for the planning and composition tasks that constitute the agent's core function. By routing tasks to the model appropriate for their complexity, the architecture reduces cost without sacrificing reasoning quality [18].

5.5.2 Compression Pipeline

Figure 5.1 illustrates the two-phase pipeline as designed. In the **compression phase**, the session history and toolkit responses are processed by a Gemini 2.5 Flash compressor, producing a compressed context and compressed data artifacts. In the **classification and reasoning phase**, these compressed inputs—together with the current user query—are passed to the GPT-4.1 reasoning agent, which produces the final classification result and tool calls.

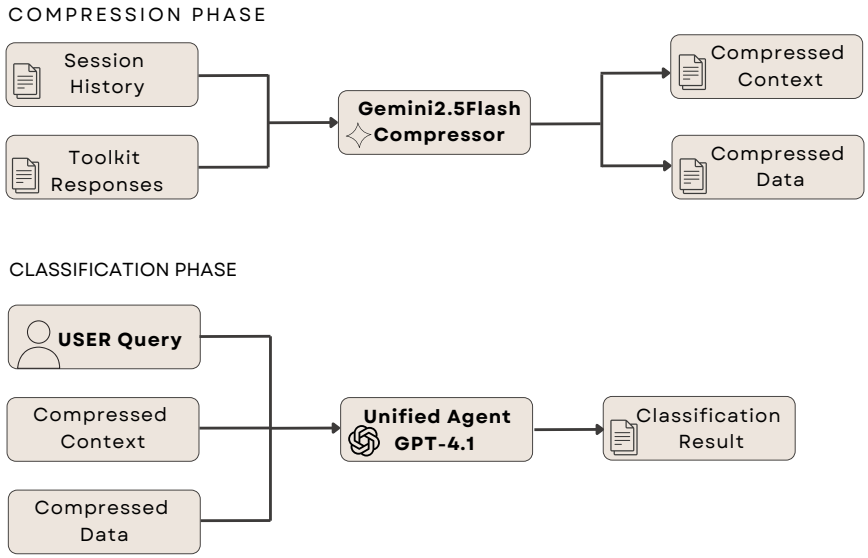


Figure 5.1: The dual-LLM compression pipeline. Gemini 2.5 Flash handles context compression; GPT-4.1 handles domain reasoning. Inputs are session history and toolkit responses; output is the classification and action result.

Of these two stages, only toolkit-response compression was implemented and evaluated as part of this work, in the form of a dedicated `compression_agent` built on Gemini 2.5 Flash. Session-history compaction and its end-to-end integration with the GPT-4.1 reasoning agent were designed but not implemented in the final system; they are discussed here as a proposed architecture rather than a validated one.

5.5.3 Compression Results

The `compression_agent` was tested on live `ProviderDefinitions` responses—the same endpoint described earlier in this chapter (Table 5.1)—applying the compression stage on top of the already-optimized JSON payload. Table 5.3 presents the

measurement: for this response, the agent reduced the payload from 1,056 tokens (baseline) to 408 tokens—a 61.4% reduction—by retaining only the metric names and counts relevant to the current query and discarding structural metadata. This figure is an empirical measurement obtained from the implemented component, not an estimate.

Table 5.3: Token consumption for the `ProviderDefinitions` toolkit response before and after processing by the Gemini 2.5 Flash `compression_agent`.

Toolkit Response	**Tokens**	**Reduction vs. Baseline**
`ProviderDefinitions` (baseline)	1,056	0%
`compression_agent` output	408	61.4%

Beyond this single measurement, a 40–70% reduction for toolkit responses in general was used internally as a design target during the architecture's specification; the `ProviderDefinitions` measurement above falls within this range and is its only empirical instance within this thesis.

Session-history compression was designed around the sliding-window compaction mechanism documented in the official Google ADK context-compaction guide [22], which triggers periodic summarization at a configurable `compaction_interval` while retaining a trailing `overlap_size` of the most recent turns for continuity. This thesis refers to that retained trailing window informally as *tail-retention*—a paraphrase used here for readability, not a term used in Google's own documentation—to describe the effect the mechanism is meant to produce: recent turns stay available unsummarized to preserve pronoun resolution and immediate coherence, while older turns are folded into a summary once the configured interval is reached. This design was not empirically validated within this thesis; doing so would require completing the session-history compaction implementation and integrating it with the GPT-4.1 reasoning agent, which remains future work.

5.5.4 Cost Efficiency

Beyond token volume, routing the implemented toolkit-response compression stage to Gemini 2.5 Flash instead of GPT-4.1 realizes the cost advantage that motivates the dual-LLM design: compression and summarization tasks lean on fluency, instruction following, and accurate information extraction, none of which requires frontier-level reasoning capability. According to third-party model benchmarking data (Artificial Analysis[1]), GPT-4.1 is priced at $2.00 per million input tokens and $8.00 per million output tokens, against $0.30 per million input tokens and $2.50 per million output tokens for Gemini 2.5 Flash—roughly 6.7× cheaper on input and 3.2× cheaper on output. These are the currently listed rates rather than a snapshot retrieved specifically at benchmarking time, so the exact multiple may have shifted slightly since; the direction and order of magnitude of the gap are what motivate the model-tiering design. The same division of labor—cheaper model for compression, frontier model reserved for reasoning—applies to the remaining, not-yet-implemented stages of the architecture.

5.6 Performance Timing

Token consumption is not the only operational metric of relevance. Table 5.4 presents response time comparisons for the same benchmark queries: the new implementation was consistently equal to or faster than the DEV Bot baseline, with improvements of up to ten seconds on discovery-heavy queries, driven by reduced time processing large metric payloads and fewer sequential tool calls per query. Each figure reflects a single timed run per query, manually recorded and rounded to the nearest five seconds, with no repeated-trial variance collected—these values should

[1] `https://artificialanalysis.ai`, accessed August 2026

be read as indicative of direction and rough magnitude, not as statistically robust measurements.

Table 5.4: Response time comparison (seconds) for selected queries.

Test Case	**New Implementation**	**DEV Bot**
Show me all available metrics	40s	50s
Show details of each metric	30s	35s
Retrieve all metrics with details	25s	25s
Retrieve details for the metric users	30s	35s
Create a card to show response time	35s	35s
What metrics can I put in my card?	45s	50s
I want to create a card, tell me available metrics	35s	40s
Show me all error-related metrics	25s	25s
Show response time for the last 7 days	30s	30s
Sum up all exceptions for my services	30s	30s

5.7 Summary

The data retrieval and context management optimizations documented in this chapter produced substantial, measurable improvements in the token efficiency and response latency of the `joule_card_agent`. The primary contributions were:

- A redesigned `ProviderDefinitions` API endpoint achieving a 75.6% (JSON) to 85.6% (CSV) reduction in endpoint-level token consumption.
- End-to-end token reductions of 48–73% for queries served entirely through the lightweight listing call, against the baseline DEV Bot implementation.
- A proposed dual-LLM compression architecture (Gemini 2.5 Flash + GPT-4.1). Of its two stages, toolkit-response compression (target: 40–70% reduction) was implemented and validated, reducing the tested `ProviderDefinitions` response by 61.4% (1,056 to 408 tokens); session-history compaction, designed

around the sliding-window compaction mechanism documented for Google ADK [22] (referred to informally in this thesis as *tail-retention*), and its integration with the GPT-4.1 reasoning agent remain proposed future work.

- The identification and characterization of the Execution Trap: a counter-intuitive failure mode in which reducing per-call payload size by adding execution steps increases total token consumption.

This chapter treated context window economics as a primary engineering constraint on the `joule_card_agent` system specifically. The architectural decisions documented here—information density maximization, format optimization, lazy detail loading, and model tiering—were validated on this one system; whether they transfer directly to a system with different traffic patterns, backend constraints, or model choices is a plausible extrapolation this thesis makes but does not test.

Chapter 6

Applying Grounding: MCP Integration and Prompt Architecture

Chapter 2 established that grounding—tethering LLM reasoning to verifiable, external knowledge—is essential for reliable operation in knowledge-intensive domains, and described the Model Context Protocol as the emerging standard for externalizing that knowledge into structured, on-demand-accessible resources. This chapter examines how those principles were applied to the SAP Cloud ALM Unified Analytics Capability: how domain knowledge was migrated from static system prompts into an MCP server, and how the prompt architecture was redesigned to achieve grounding through progressive disclosure rather than upfront injection.

6.1 The Grounding Problem in UAC

Before the architectural changes described in this chapter, the `joule_card_agent` and `dashboard_agent` carried their domain knowledge directly within their system prompts. This included metric taxonomy descriptions, API usage guidelines, field validation rules (valid status values, supported aggregation functions), and workflow instructions for composing valid card and dashboard configurations.

This static injection pattern created two distinct problems. First, it was a source of context bloat: every agent invocation—regardless of the simplicity of the user request—loaded the full corpus of domain knowledge into the context window. A

user asking *"How many metrics are available?"* received the same upfront context injection as a user requesting a complex multi-metric dashboard, despite requiring far less domain knowledge to answer correctly.

Second, it was a maintenance problem. As the UAC metric catalog evolved and new API guidelines were issued, every update to domain knowledge required a corresponding update to the system prompt embedded within the agent code. There was no single source of truth for domain knowledge; it was distributed across prompt files with no systematic versioning or validation.

6.2 MCP Integration: Architecture

The transition to the CALM Orchestration architecture (Chapter 4) provided the structural opportunity to resolve both problems, though not by the same mechanism or on the same timeline. The maintenance problem is resolved by the single-source-of-truth structure discussed in Section 6.2.4; the bloat problem is resolved separately, by progressive disclosure, introduced only in Section 6.3—nothing in this section alone reduces upfront context injection. The new architecture introduced a dedicated **CALM MCP Server**: a platform-level MCP server that functions as the single source of truth for all domain knowledge consumed by the UAC Skills.

The MCP server was organized around the three primitive types defined by the MCP specification (Chapter 2): Tools, Resources, and Prompts. Tools and Resources are the constructs that progressive disclosure (Section 6.3) operates on to address the bloat problem; centralizing all three in one server, rather than distributing domain knowledge across per-agent prompt files, is what addresses the maintenance problem (Section 6.2.4). Each is described in turn below.

6.2.1 Tools: Structured Action Execution

Tools in the CALM MCP server are focused, action-oriented operations that return JSON data. Rather than general-purpose data retrieval functions, MCP tools are designed around specific, bounded operations that a Skill may need to perform: custom query functions that retrieve only the fields required for a given workflow, update functions that modify specific entity attributes without touching unrelated state, and validation functions that check configuration parameters against the current catalog before committing them.

This design principle—focused tools over general-purpose tools—directly limits the data surface area exposed to each LLM invocation. A Skill that needs to look up a metric's supported aggregation functions receives exactly those fields; it does not receive the metric's full schema.

On its face, this looks like the pattern Chapter 5 identified as the Execution Trap: narrow calls in place of one broad call. The two are not the same failure mode. The Execution Trap arose when a *single* information need was split into a *mandatory, fixed sequence* of calls—list, then detail—so that answering one question always cost two round trips, regardless of what the agent actually needed. MCP tools are narrow in a different sense: each tool is invoked *conditionally*, only when its specific, bounded need actually arises. The design allows, though does not guarantee, one tool call per sub-need rather than a fixed pipeline; no task-log analysis was performed to confirm how often that ratio holds in practice. What matters for data density is not "narrow calls vs. one broad call in the abstract," but tokens consumed per task, including re-processing, against the old monolithic-prompt baseline: a narrow tool returning only the fields a sub-task needs, invoked once, is denser by that measure than injecting the full metric schema on every invocation regardless of need.

The two-endpoint split that produced the Execution Trap failed a different test—it added a call that was *always* required in addition to the first, for information a single denser call could have supplied instead—the multi-call structure was mandatory, not conditional. This is the MCP-layer expression of the information density principle from Chapter 5, achieved here through conditional narrow calls rather than an unconditional sequence.

That argument holds for a single information need; it does not by itself bound the multi-need case Section 6.1 raised as motivating—a complex multi-metric dashboard request that may require N narrow tool calls, one per metric or validation check. Conditional-not-mandatory framing rules out an Execution-Trap-style fixed pipeline forced onto a single need; it does not establish that N narrow calls, summed, stay denser in aggregate than the old monolithic injection would have been. What can be said structurally is that per-call density does not degrade as N grows, since each call still returns only its sub-need's fields, while the old system paid the full injection cost once regardless of N. Whether the aggregate cost across N conditional calls stays below that fixed baseline for realistic N was not benchmarked for this multi-metric case specifically; Chapter 5's end-to-end measurements are the closest available evidence, but they measure the API layer's `ProviderDefinitions` calls, not this chapter's MCP tool calls, so they support the general principle without directly measuring this instance of it.

6.2.2 Resources: Contextual Knowledge Injection

Resources provide contextual information that the LLM needs at runtime but that changes on a materially slower cadence than conversation state: conversation state changes on every turn, whereas the resources below change only when the backend API surface or a code list is actually modified—an event tied to deployments, not

to user interaction. In the UAC MCP server, two categories of resources were implemented:

- **Auto-generated OpenAPI specifications**: the MCP server exposes API documentation describing the available backend endpoints, their parameter schemas, and their response structures. This documentation is generated automatically from the backend service definitions, removing the manual step—and the drift it invites—of hand-updating a prompt file whenever the API changes. The project material does not specify the regeneration trigger or the propagation lag between a backend change and an updated resource, so "always current" should be read as "current as of the last regeneration."

- **Code lists**: enumerated values required for field validation—valid status transitions, supported aggregation types, recognized metric providers—are exposed as JSON resources rather than being embedded in the system prompt. The LLM can retrieve these lists at the point of use, verifying its planned action against the authoritative enumeration before committing it.

Exposing validation data as resources rather than prompt text has a grounding benefit: the LLM retrieves the value the MCP server currently holds at runtime, rather than reasoning from a version embedded in a prompt file and never revisited—a practical application of the direct context injection technique discussed in Chapter 2. This is a genuine improvement over static embedding regardless of the exact regeneration cadence: a resource fetched at request time cannot be staler than the last regeneration, whereas an embedded prompt value has no update path short of an engineer editing it.

6.2.3 Prompts: Separated Expert Guidance

The most architecturally significant change was the decomposition of the monolithic system prompt into two distinct, separately loaded prompt types:

- **Domain Expert Prompts**: encode workflow guidelines, business rules, and domain best practices. These prompts tell the LLM *how* to approach a UAC task—what questions to ask the user, what order to perform operations, what to do when a metric is not found. They represent procedural expertise.

- **API Efficiency Prompts**: encode API usage constraints designed to protect system resources and prevent over-fetching. These prompts enforce rules such as: retrieve only the fields you need, do not call detail endpoints until discovery is complete, use selective field querying for list operations. They represent operational guardrails.

Separating these prompt types is intended to produce two unmeasured benefits: independent update schedules—business rules are expected to change on a different cadence than API efficiency constraints, though no change-frequency data confirms this in practice—and improved readability, since each type's purpose is identifiable at a glance rather than buried in an undifferentiated prompt block. The latter is a readability argument, not a measured reduction in audit or extension time, neither of which was tracked before or after the change.

6.2.4 Maintainability and Reusability

Apart from providing benefits like grounding and contextual efficiency, there are two more advantages of operating MCP architecture at enterprise level.

- **Maintainability**: domain knowledge is now gathered in one place rather than spread across several agent-prompt files requiring manual coordination. Whenever an API endpoint is updated, a metric category is added, or a validation rule changes, the update is made once, in the MCP server, rather than in every referencing agent prompt. Whether every consuming application picks up the change immediately or only on its next fetch depends on whether the resource is fetched fresh per call or cached client-side—a detail this thesis's project material does not specify, so "without redeployment of agents" should be read narrowly: no agent code needs to change, but propagation latency to already-running sessions is not established here. The OpenAPI specs used as Resources are generated directly from the backend service definitions, keeping them correlated with what the system actually exposes, independent of that propagation-latency question.

- **Reusability**: the knowledge base layer is architecturally independent of any single orchestration platform, in the sense that the MCP server's Tools, Resources, and Prompts encode no CALM-specific detail; only a thin skill definition layer translates between the MCP server and whatever orchestration infrastructure invokes it. This separation was exercised once, during the migration to the CALM Orchestration infrastructure described in Chapter 4: the domain knowledge accumulated in the MCP server did not need to be re-authored, only the agent-side interface layer was rewritten—a real, observed effect of that one migration, though no time or engineering-effort estimate was recorded for it. Whether the same would hold for a transfer to a different orchestration platform entirely remains an untested extrapolation, since only one such transition has been tried.

6.3 Progressive Disclosure in Practice

The MCP integration enables the application of progressive disclosure to the UAC prompt architecture. Rather than injecting the full contents of Domain Expert Prompts and API Efficiency Prompts at the start of every skill activation, the CALM Orchestration Agent starts with a lean baseline context. The detailed prompts are served by the MCP server on demand—loaded into context dynamically when the AI orchestrator explicitly activates a Skill to execute a specific task.

6.3.1 Lazy Skill Loading

At the orchestrator level, the CALM Orchestration Agent maintains only lightweight metadata summaries of available Skills—approximately 100 bytes per Skill, consistent with the order of magnitude Piskala reports for skill metadata (Section 2.1)—covering the Skill name, a one-sentence description, and activation keywords. No field-by-field byte accounting was recorded for this system specifically; illustratively, a short identifier (on the order of 15–20 bytes), a one-sentence description (on the order of 50–70 bytes), and a handful of keywords (on the order of 15–25 bytes) sum to roughly that order of magnitude, but this breakdown is offered only to show the figure is plausible against Piskala's reported number, not as an independently measured composition of this system's own Skill metadata. Full Skill definitions, including their embedded prompt references and tool declarations, are loaded strictly on demand when the router determines that a given Skill is needed for the current user request.

This implements the three-stage progressive disclosure model from Chapter 2:

1. **Discovery**: the orchestrator loads 100-byte Skill metadata summaries.
2. **Activation**: upon intent matching, the full Skill definition is loaded.

3. **Execution**: the Skill loads domain-specific MCP resources (API specs, code lists, expert prompts) only as needed during task execution.

The context efficiency gains compound across these stages: the orchestrator's context window is never polluted with Skill-level domain knowledge; the Skill's context window is never polluted with resource-level detail it has not yet requested.

6.3.2 Resolving the Context Bloat Problem

Progressive disclosure is the mechanism, promised in Section 6.2 above, that resolves the context bloat problem identified in Chapter 3. By construction, the system prompt of an active Skill is smaller than the old, static system prompt for the same domain: it contains execution scaffolding (how to call tools, how to format responses) but not the corpus of domain knowledge, which now arrives as MCP resources requested at the granularity the current execution step requires. This thesis does not report a direct before/after token count for a Skill's system prompt specifically (as distinct from the end-to-end, whole-query measurements in Chapter 5), so the reduction here is a structural consequence of the design—domain-knowledge content once present on every invocation is now present only when requested—rather than a quantified figure.

The context budget available for reasoning follows the same logic: a smaller system prompt leaves more of the window available for conversation history and tool responses. Whether this measurably reduces exposure to the "lost in the middle" degradation documented in [14]—where relevant information can fail to influence outputs when positioned far from the ends of a long context—was not tested here; no positional analysis or ablation was run, so the connection is offered as a plausible mechanism, not a demonstrated effect.

6.4 The Hybrid Integration Challenge

The Tools/Resources/Prompts architecture and the progressive-disclosure mechanism described above were not implemented against a blank slate. A practical complication arose during the MCP integration: the new SAP orchestration guidelines provided an idealized, "clean-slate" MCP architecture, but the existing codebase already had functional MCP implementations developed incrementally over preceding months. Blindly applying the new guidelines would have broken existing features that customers and the team depended on.

The resolution was a deliberate hybrid approach: rather than discarding prior work, specific architectural principles from the new guidelines were extracted and retrofitted into the existing codebase, selected against two concrete criteria—risk of breaking customer-relied-upon features, and cost of retrofit versus full replacement. The MCP server components were restructured into distinct Tools, Resources, and Prompts following the new separation of concerns, judged—through engineering assessment during the migration itself rather than a formal scored review—as low-risk to retrofit and high-value for the grounding and maintainability benefits described earlier in this chapter. The underlying implementation logic from the existing backend was preserved, since rewriting it risked breaking already-deployed customer-facing behavior for no corresponding benefit.

This pragmatic approach worked for this one migration; whether it generalizes to a principle for enterprise AI system evolution broadly is a claim this thesis does not have the evidence to make, since a single internship's case study is not a basis for inducing a general rule. What can be said at this thesis's scope is narrower: extracting the separation-of-concerns principle—Tools for actions, Resources for contextual data, Prompts for expert guidance—and applying it to the existing codebase let this

system gain the grounding and context efficiency benefits described above without the cost and risk of a full rewrite.

6.5 The Skill Abstraction Boundary

The MCP integration clarified the precise architectural boundary between the Skill and the MCP server, a boundary that has both a correctness dimension and an autonomy dimension.

From a correctness perspective, the Skill must be capable of completing its task using only the context provided in its activation plus the data available from MCP resources. It cannot rely on state from sibling Skills or on knowledge not accessible through the MCP interface. This constraint enforces the context isolation property described in Chapter 4 and is intended, by design, to bound the Skill's failure modes to its own context—a design intent rather than a tested guarantee, since no fault-injection test or production incident was used to verify containment. The failure modes this scoping is intended to cover are the ones reachable from within a Skill's own execution: an incorrect tool call (e.g., wrong parameters passed to an MCP Tool), a hallucinated parameter value not present in the retrieved Resource data, or a timeout on an MCP call the Skill was waiting on. It does not, by construction, cover failures whose cause lies outside a Skill's own context, such as the Orchestration Agent routing a request to the wrong Skill in the first place.

From an autonomy perspective, the boundary is where a distinction becomes practically meaningful. Piskala [1] distinguishes agentic constructs by their *control characteristics* rather than by perceived intelligence: a **Sub-Agent** retains autonomous decision-making authority over what to do next, while a **Skill** is a bounded unit of procedural knowledge that executes but does not decide—the decision of what

happens next remains with whatever invoked it. Under that framework, the UAC architecture positions its components unambiguously: the MCP server holds the domain knowledge; the Skill executes procedures using that knowledge but does not decide which procedure to run next; the Orchestration Agent makes that decision. This assignment is a design claim about how the system was built, not a property independently derived from or verified against Piskala's criteria; it does, however, mean the critical design axis the framework identifies—*who decides what happens next?*—has a single, unambiguous answer in this architecture: the Orchestration Agent decides, the Skill executes, the MCP server informs.

6.6 Summary

The MCP integration and prompt architecture redesign documented in this chapter transformed the UAC grounding strategy from static prompt injection to dynamic, on-demand knowledge delivery. The primary contributions were:

- Migration of domain knowledge from static system prompts to a structured CALM MCP Server, organized into focused Tools, contextual Resources, and separated Expert and Efficiency Prompts.
- Implementation of progressive disclosure at the orchestrator level (lazy Skill loading) and at the Skill level (on-demand MCP resource retrieval), reducing upfront context injection to the minimum required for each execution step.
- Resolution of the context bloat problem through prompt decomposition, creating a system where context consumption scales with task complexity rather than with total system capability.

- A pragmatic hybrid integration strategy that applied new architectural principles to an existing codebase without requiring a full rewrite.

The grounding architecture is designed to be more reliable and more efficient than the static-injection baseline it replaced, though the two claims hold at different strengths. The efficiency claim is structural: the context consumed at each step is sized to that step rather than to the full domain corpus, a consequence of progressive disclosure and prompt decomposition, even though no direct before/after token count was measured for a Skill's system prompt specifically (Section 6.3). The reliability claim is narrower: domain knowledge is drawn from a managed, versioned source rather than embedded in prompt files, and a Skill's failure modes are intended, by design, to stay contained within its own execution (Section 6.5)—a design intent, not a property confirmed through fault injection or production data, and one that excludes failures whose cause lies outside a Skill's own context, such as the Orchestration Agent routing a request to the wrong Skill. "More reliable," read together with the hedges above, means reliable within the failure modes the architecture was designed to contain, not an end-to-end guarantee for the system as a whole.

Chapter 7

Evaluation

Chapter 5 provided the thesis's primary quantitative evaluation: token reductions of 75.6–85.6% at the endpoint level and 48–73% end-to-end against the DEV Bot baseline, latency improvements of up to ten seconds on discovery-heavy queries, and a 61.5% reduction for the one implemented and tested stage of the dual-LLM compression architecture (its session-history stage was not empirically validated). Those results are not repeated here. This chapter provides a complementary *qualitative* evaluation of the architectural decisions made across Part II: grounding reliability, context isolation, and maintainability. All three are measurable in principle (a hallucination rate before and after the MCP migration, a count of errors contained versus propagated across Skill boundaries, a test-suite size or runtime comparison before and after the Router-Skill decomposition), but none of these were collected; the benchmarking documented in Chapter 5 was scoped to token consumption and latency, excluding these axes. What follows is therefore a description of the mechanism by which each property is intended to deliver its stated benefit. It is not an independent empirical measurement of whether it did. Where Chapters 4 and 6 already qualified a claim as unverified or design-only, that qualification is carried forward here, without being dropped.

7.1 Grounding Reliability

Before the MCP integration described in Chapter 6, domain knowledge was embedded statically in agent system prompts. This left the system exposed, by design, to a general failure mode well documented in the hallucination literature: model outputs degrading when the knowledge they reason from is stale or no longer matches the state of the world the model is meant to be reasoning about [13]. [13] surveys this failure mode for natural language generation broadly (summarization, dialogue, and related tasks), not for tool-use agents specifically; reading it as a claim about *tool-use decisions* degrading under stale knowledge is this thesis's own extension of that general mode to the UAC setting, offered without a documented case of it occurring in this system. Applied here, the extension points to a specific, plausible instance of the general mode: the agent could reason from prompt-embedded knowledge that had drifted from the actual state of the backend, referencing metric names that had been renamed, status values that had been deprecated, or API fields that no longer existed. Because the prompt was updated manually and asynchronously with backend changes, the gap between what the agent believed and what the system supported was, at minimum, possible by construction and plausible as a persistent source of incorrect tool calls. The MCP server addresses the precondition that made that failure possible, though not unconditionally, in the same conditional sense already established in Chapter 6: API specifications and code lists are served from the live backend at request time, no longer frozen at prompt-authoring time; the grounding problem moves from prompt maintenance to infrastructure maintenance without disappearing, since a stale MCP resource is exactly as capable of misleading the agent as a stale prompt was. "Grounded by construction" therefore holds only as long as the MCP server's generation pipeline is in fact kept current, an opera-

tional commitment this thesis does not measure. A secondary grounding benefit, developed in Chapter 6, comes from separating Domain Expert Prompts from API Efficiency Prompts: because the two are maintained as genuinely separate resources, no longer regions of one shared file, an edit to one cannot syntactically touch the other's content, removing one specific mechanism (though not every mechanism, such as a shift in overall context length) by which an unrelated edit could disrupt it. No incident count or before/after comparison was collected to confirm this in practice; the claim concerns the mechanism; it is not a measured outcome.

7.2 Context Isolation

The Router-Skill architecture enforces context isolation as a property of the architecture itself, to the same extent and with the same limitation already established in Chapter 4: demonstrated in production only for `joule_card_agent`; not yet for `dashboard_agent`. This chapter adds no evidence that changes that status. Two distinct isolation properties are worth separating here, since the Router-Skill migration only speaks to one of them. *Cross-Skill* isolation is what the migration changed: previously, `joule_card_agent` and `dashboard_agent` ran as sub-agents under a shared orchestrator, without the same formal isolation guarantee the Router-Skill pattern later introduced. Chapter 4 does not characterize that earlier arrangement's isolation properties one way or the other, so this is a comparison to what came later; it is not a claim that no isolation existed under the earlier design. Under the Router-Skill pattern, each fully-implemented Skill operates in a strictly isolated context, coordinated only through well-defined, structured handoffs managed by the CALM Orchestration Agent: demonstrated in production for `joule_card_agent` and a design intention, not yet demonstrated, for `dashboard_agent`. *Intra-Skill* isolation

is a separate question the migration does not address: `joule_card_agent`'s own workflow still requires a metric discovery phase before a card creation phase, both executed within that one Skill's own reasoning (Chapter 5), and a reasoning error in the first phase (for example, misidentifying a metric) can still propagate into the second, exactly as it could before the migration. The Router-Skill isolation guarantee bounds failures at the Skill boundary; it says nothing about failures between phases inside a single Skill's own multi-step reasoning, and this chapter makes no claim that it does. Within its actual scope (the Skill boundary), this isolation is intended to contain failures, by the same architectural argument Chapter 4 makes for bounded failure and connects to Kim et al. [4]'s finding on coordination overhead in decentralized topologies: no shared mutable state exists for an error to propagate through from one Skill's reasoning into a sibling's. No error rate, and no count of contained versus propagated failures, was collected to confirm this in practice: isolation removes a specific mechanism of propagation, but whether propagation actually decreased was not measured, here or in Chapter 4. This matches the error containment property described in Chapter 2, where context isolation bounds the blast radius of individual reasoning failures in hierarchical systems.

7.3 Maintainability

The architectural changes are intended to improve system maintainability along two axes; as with the properties above, no test-suite size, test-runtime, or bug-count comparison before and after the change was collected, so both points below describe an architectural argument; neither is a measured outcome. First, the Router-Skill decomposition makes each Skill independently testable in principle: a Skill can be validated by providing it with a fixed activation context and a set of MCP resources,

then verifying that it produces correct tool calls and structured outputs, whereas under the previous monolithic structure, testing required simulating the full multi-agent interaction from user request to final API call. Concretely, this bounds the test-input surface to one Skill's activation context and its declared MCP resources, well short of the full session state of a multi-turn, multi-agent interaction. The number of test cases such isolation would produce and their runtime were not measured, so no claim is made about either. Second, the MCP server is the single, versioned source of the Tools, Resources, and Prompts holding all domain knowledge consumed by the agents, in the sense already established in Chapter 6: a schema or catalog change is made once, in the MCP server, eliminating the earlier need to coordinate it across multiple agent prompt files, which is intended to prevent the class of consistency bug where a prompt file and the backend implementation it describes drift out of sync. No bug-tracker data confirms how often this class of bug occurred before the migration or whether it has occurred since; the claim is that the mechanism that produced it, multiple independently-editable copies of the same fact, no longer exists for MCP-sourced knowledge. That is distinct from a claim that the bug class has demonstrably disappeared. Together, these properties (runtime grounding, context isolation, and centralized knowledge management) describe what the architectural decisions documented in this thesis are designed to produce; this chapter has not measured them against the baseline system as a demonstrated improvement. It is an account of the mechanisms by which each property is intended to reduce a specific class of failure or maintenance cost relative to the static-prompt, monolithic-agent baseline described in Chapter 3, independent of the quantitative token savings reported in Chapter 5, which remain this thesis's only directly measured evaluation.

Chapter 8

Conclusions and Future Work

8.1 Summary

This thesis addressed a cluster of interconnected failures in a production multi-agent LLM system: token consumption at the data retrieval layer that reached approximately 219,000 tokens for a single metric-listing call (Section 5.1) and an estimated 600,000 tokens across a three-turn interaction (Chapter 3); context bloat from a Skill's full domain-expertise corpus loading on every invocation, in place of the roughly 100-byte metadata summaries the redesigned architecture now loads at discovery time (Chapter 6); architectural mismatch between the initial two-agent design and the platform-mandated Router-Skill pattern (Chapter 4); and a data model keyed on a single static `Id` that could not represent a draft and a published version of the same dashboard at once (Chapter 3). Each problem had a theoretical counterpart in the literature on context window economics, grounding, and multi-agent design patterns. This thesis's contribution is to show how those frameworks apply in a real, constrained industrial system, and, in one case, where a naive reading of them needs refinement: the Execution Trap (Chapter 5) shows that splitting a large call into smaller ones can increase total token consumption rather than reduce it. The refinement proposed here is that the correct target is information density—the ratio of task-relevant tokens returned to total tokens consumed, including re-processing cost—per execution step, not the size of any individual call.

The remaining contributions below apply existing frameworks to this system; they do not revise them.

- The **API redesign** resolved the most expensive failure mode. Restructuring the `ProviderDefinitions` endpoint—grouping metrics by provider, removing redundant filter metadata, and separating lightweight listing from full schema retrieval—cut the primary discovery payload by 75.6–85.6%, depending on response format (two endpoint-level measurements against one legacy baseline, not a statistical sample; Table 5.1). More importantly, the work identified the *Execution Trap*: splitting API calls to shrink individual payloads increases total consumption by adding reasoning steps. The redesign's own listing/detail split avoids this because detail retrieval is invoked conditionally, only after the agent has identified a specific metric, not as a second call attached to every listing request; Chapter 5 documents the one case where the detail tool was invoked in bulk and reproduced the same failure. The underlying mechanism—re-processing cost compounding with each additional round trip—should generalize to any LLM architecture that retains full, unsummarized history; this thesis measured it in one production system only, and did not test architectures using summarization, truncation, or retrieval-augmented context. Within that scope, the argument is to maximize information density per execution step, not minimize per-call response size.

- The **Router-Skill architecture** resolved the mismatch between the initial monolithic agents and the CALM Orchestration platform, enforcing context isolation as a hard property: each Skill reasons within its own bounded window, with no shared state between domains. This is demonstrated in production for the fully-implemented `joule_card_agent` Skill, and designed but not

yet running for `dashboard_agent` (both retain their legacy `_agent`-suffixed names; Chapter 4). The containment is meant to limit token consumption and failure propagation by removing the shared-state path through which one Skill's error could reach another, though no failure-rate measurement confirms this in production. The final architecture is consistent with the theoretical prescription of hierarchical orchestration for enterprise systems [1]—a narrower claim than validating the framework, which would need comparison against an architecture that did not follow it. What this thesis does show is that the prescription was realizable under real constraints, including a mid-project platform migration and legacy code the team could not discard (Chapter 6).

- The **MCP server integration** completed the grounding strategy by replacing static prompt injection with runtime knowledge retrieval. Domain Expert Prompts and API Efficiency Prompts, separated by concern and served on demand, are intended to keep the agents reasoning from current knowledge rather than a frozen snapshot; whether the underlying generation pipeline is itself kept current with the backend is not measured here (see Limitations). Progressive disclosure—from 100-byte Skill metadata through full Skill activation to on-demand MCP resource retrieval—implements the three-stage loading model of Chapter 2 at the platform level.

- The **dual-LLM compression architecture** (Gemini 2.5 Flash + GPT-4.1) is a proposed pipeline whose toolkit-response compression stage was implemented and measured on one `ProviderDefinitions` response, reducing it by 61.4% (1,056 to 408 tokens)—shown to work on this one payload, not across varying payload types or repeated trials. Session-history compaction, designed around the sliding-window compaction mechanism documented for

Google ADK [22] and referred to informally in this thesis as *tail-retention*, and its integration with the reasoning stage remain unimplemented future work. "Compression" and "reasoning" are informal labels for these two single-invocation pipeline stages, not a claim that either is an *agent* under this thesis's taxonomy.

- As a secondary contribution, the **dashboard versioning schema** solved the lifecycle-management requirement with the simplest data model that met the actual requirements: a `status/instance_id` approach supporting draft/publish/discard workflows while staying straightforward for LLM-driven interaction. The design weighed this against full-history versioning and separate-table approaches (Chapter 3) on engineering-overhead and auditability grounds, not comparative failure data, since none was collected for approaches that were not built. Simplicity reduced the surface for the kind of implementation error Chapter 3 discusses (an agent misreading an edit as a replace); whether that holds as a general property of AI-managed data systems is a hypothesis this thesis raises but does not test.

8.2 Limitations

The results above are reported without the qualifications attached to them earlier in this thesis; this section collects those qualifications in one place, since a conclusions chapter that restates only the confirming results is not a complete account of the work.

Scope of implementation. Not everything designed here was built and run. The `dashboard_agent`'s Router-Skill migration was designed but not implemented, so its context-isolation guarantee is a design property, not a demonstrated one

(Chapter 4). The dual-LLM architecture's session-history compaction stage was designed but not implemented or integrated with the reasoning stage (Chapter 5).

Scope of measurement. Several claims here are structural arguments about what a design change makes possible, not measurements of whether that possibility was realized. No hallucination rate, error-propagation count, or test-suite comparison was collected for the grounding, isolation, and maintainability properties discussed in Chapter 7; the connection drawn there to Kim et al.'s [4] finding on decentralized-topology coordination overhead is likewise unmeasured on this system. The "lost in the middle" connection in Chapter 6 is a plausible mechanism, not a tested effect—no positional analysis or ablation was run. The timing benchmark in Chapter 5 is one recorded run per query, rounded to the nearest five seconds, with no repeated-trial variance; neither this thesis nor the chapter it summarizes specifies the tokenizer behind the token-count figures reported throughout.

Open engineering questions. Whether an MCP resource is fetched fresh per call or cached, and what propagation latency that implies for an already-running session, is not established anywhere in this thesis's project material (Chapter 6)—directly relevant to how strong a claim can be made about domain knowledge staying current.

Trade-offs not evaluated. This thesis does not report whether context isolation costs anything in cross-domain reasoning quality, or whether the compression stage introduces latency or fidelity loss large enough to matter for specific query types. Both are plausible costs of the architectural choices made here, and neither was measured.

8.3 Customer Outreach: A Dissemination Exercise

The preceding sections describe this thesis's engineering contributions, evaluated against the production UAC system. This section is different: a dissemination activity that reused the same conceptual model—the MCP protocol's Tools, Resources, and Prompts primitives and its progressive-disclosure strategy (Section 2.4; a different sense of "Tools" than the Tool/Skill/Sub-Agent/Agent taxonomy of Section 2.1)—in an unrelated context, for an unrelated audience, with no shared measurement against the core chapters' benchmarks. It is included because it happened during this internship and because the conceptual reuse is worth recording; it does not extend or confirm the thesis's empirical results.

The exercise was a customer-facing workshop for SAP Cloud ALM customers wishing to build their own AI-powered extensions. It translated the Tools/Resources/Prompts model of Chapter 2 into a hands-on *vibe coding* exercise: participants prompted an AI coding assistant (Cline), through structured natural language alone, to build a functional Python MCP server exposing five SAP Cloud ALM tools (OAuth 2.0 authentication—a bounded authorization exchange rather than a single call, but counted as one of the five following the workshop's own naming—plus project listing, task listing, task creation, and task update) and a companion FastAPI client. The pedagogical intent was to shift participants from writing code to writing precise prompts, echoing the same progressive disclosure and context management concepts used in production; the parallel is conceptual only, and the workshop did not share tools, codebase, or benchmarked infrastructure with the rest of this thesis.

The feedback gathered from participants during the session was positive: most completed the exercise and reported satisfaction with and interest in the experience.

The workshop was delivered successfully and produced a reusable cross-platform guide. The next iteration will extend tool coverage to operational metrics and deployment tracking, closer to the UAC capabilities described in this thesis.

8.4 Future Directions

The four directions below follow from the limitations above, ordered from most to least tied to this thesis's core empirical findings: the first two stress-test the production system's own architecture; the latter two are an adjacent optimization and a dissemination extension.

Formal context budget management. The Execution Trap and the over-fetching failure mode (Chapter 5) both stem from the absence of explicit context budget enforcement in the agent's reasoning loop. A promising direction is a runtime context budget manager that tracks tokens consumed per step against a defined budget and takes a specified action once a threshold is reached. This raises open design questions deliberately, since they are what make it a research direction rather than an implementation ticket: what quantity should define "approaching" a threshold—a fixed token count, a fraction of the context window, or something workload-specific; what value is appropriate for this workload; and what the agent should do on crossing it, from a hard stop to a forced compression pass using the compression stage already implemented in Chapter 5. Resolving these would make budget-aware behavior a system property, not something left to prompt-level guardrails alone.

Dashboard versioning at scale. The current implementation assumes a given dashboard has at most one active draft at any time, a validation rule enforced in the service layer. As usage scales, concurrent editing may arise: two users creating

drafts of the same dashboard, or an AI-initiated edit colliding with a user-initiated one. This thesis does not define what "realistic concurrent load" means for this workload—no target request rate, editor count, or timeframe is established—so a first step is establishing that baseline from production usage data before evaluating the draft/publish workflow's locking and conflict-resolution semantics against it. Only then can a concurrency model—optimistic locking, pessimistic locking, or last-write-wins with explicit conflict notification—be chosen on empirical grounds.

Adaptive model routing. The dual-LLM architecture in Chapter 5 uses a static assignment: Gemini 2.5 Flash for compression, GPT-4.1 for reasoning. A natural extension is dynamic routing, selecting each model based on the estimated complexity of the current request and the remaining context budget—though "complexity" would first need to be operationalized, whether as a token-count threshold, a task-type classification, or an estimated reasoning-step count, none of which this thesis defines. RouteLLM [18] addresses a related but distinct problem, routing single-turn queries between general-purpose models of different cost tiers rather than between two fixed pipeline roles; it is a source of routing techniques to adapt, not a framework already shown to apply here. The benchmark data collected in this work offers a starting point for training such a policy once the complexity metric is defined.

Workshop curriculum expansion. The workshop in Section 8.3 covers SAP Cloud ALM Project and Task management APIs. The most immediate extension is incorporating tools from the Operations domain—operational metrics, service health, deployment tracking—bringing the curriculum closer to the UAC capabilities at the core of this thesis. A second extension is testing the vibe coding pedagogy with a larger, more diverse cohort, gathering structured feedback on which prompt

patterns needed the most manual iteration to refine the curriculum's prompt engineering guidance—the participant data Section 8.3 notes was not collected for the first iteration.

Taken together, the first two directions extend this thesis's own measured system under conditions it was not tested against; the latter two extend it into adjacent problems that reuse its concepts without depending on its specific measurements. Each requires defining a metric before it can be evaluated as more than a proposal, and this section has tried to name which metric is missing in each case.

REFERENCES

[1] D. B. Piskala, "Agent, sub-agent, skill, or tool? A practitioner's guide to extending agentic AI systems," TechRxiv preprint, https://doi.org/10.36227/techrxiv.177204917.78786098/v1, 2026.

[2] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *International Conference on Learning Representations*, 2023.

[3] G. Mialon, R. Dessì, M. Lomeli, C. Nalmpantis, R. Pasunuru, R. Raileanu, B. Rozière, T. Schick, J. Dwivedi-Yu, A. Celikyilmaz *et al.*, "Augmented language models: a survey," *Transactions on Machine Learning Research*, 2023.

[4] Y. Kim *et al.*, "Towards a science of scaling agent systems," arXiv preprint arXiv:2512.08296, 2025.

[5] A. S. Rao and M. P. Georgeff, "BDI agents: From theory to practice," in *Proceedings of the First International Conference on Multi-Agent Systems*, 1995, pp. 312–319.

[6] R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver," in *IEEE Transactions on Computers*, vol. 29, no. 12, 1980, pp. 1104–1113.

[7] N. R. Jennings, "On agent-based software engineering," *Artificial Intelligence*, vol. 117, no. 2, pp. 277–296, 2000.

[8] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[9] G. Weiss, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[10] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "HuggingGPT: Solving AI tasks with ChatGPT and its friends in HuggingFace," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[11] T. Xie, F. Zhou, Z. Cheng, P. Shi, L. Weng, Y. Liu, T. J. Hua, J. Zhao, Q. Liu, C. Liu *et al.*, "OpenAgents: An open platform for language agents in the wild," in *Proceedings of the Conference on Language Modeling*, 2024.

[12] S. Harnad, "The symbol grounding problem," *Physica D: Nonlinear Phenomena*, vol. 42, no. 1–3, pp. 335–346, 1990.

[13] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[14] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," in *Transactions of the Association for Computational Linguistics*, vol. 12, 2024, pp. 157–173.

[15] B. T. Willard and R. Louf, "Efficient guided generation for large language models," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[16] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.

[17] A. Elnashar, J. White, and D. C. Schmidt, "Enhancing structured data generation with GPT-4o: Evaluating prompt efficiency across prompt styles," *Frontiers in Artificial Intelligence*, 2025.

[18] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica, "RouteLLM: Learning to route LLMs with preference data," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

[19] Anthropic, "Introducing the model context protocol," https://www.anthropic.com/news/model-context-protocol, 2024.

[20] SAP, "SAP cloud ALM — SAP help portal," https://help.sap.com/docs/r/product/CloudALM/latest/en-US, 2026, accessed August 2026.

[21] ——, "SAP announces Joule, a new generative AI assistant," https://news.sap.com/2023/09/joule-new-generative-ai-assistant/, 2023.

[22] Google, “Context compaction — Agent Development Kit (ADK) documentation,” https://google.github.io/adk-docs/context/compaction/, 2026, accessed August 2026.