Model Context Intelligence

Intelligent Orchestration for the Multi-Model Era

Paul Puckett • December 2025

Executive Summary

Four independent forcing functions are converging on the same conclusion: organizations will operate across heterogeneous model tiers — from micro language models at the edge to frontier models in secure clouds — whether they plan for it or not.

Infrastructure risk. Frontier model inference depends on concentrated GPU supply chains: a single manufacturer, a single foundry, packaging capacity sold out through 2026. Taiwan Strait tensions threaten access to infrastructure that cannot be quickly replicated. Betting everything on frontier availability is a strategic vulnerability.

Economic crossover. Fine-tuning specialized models used to be prohibitively expensive. That has changed. The cost of training a domain-specific 3B-7B parameter model has dropped by orders of magnitude. For high-volume tasks, building a specialized model is now cheaper than renting frontier inference at scale — and unlike API spend, the model becomes an asset you own.

Accuracy crossover. Specialized models don't just match frontier on domain tasks — they often exceed it. A 3B model fine-tuned on millions of medical cases develops deeper pattern recognition than a 500B model trained on general web text. For bounded tasks, specialized models are frequently the accuracy-maximizing choice, not a compromise.

Compliance reality. Some environments cannot use frontier at all. Classified systems cannot send data to commercial APIs. Healthcare and financial institutions face data residency constraints. Edge deployments have no reliable connectivity. In these contexts, small and micro language models (SLMs and MLMs) aren't a fallback — they're the only path to AI capability.

Each forcing function independently justifies specialized model adoption. Together, they make heterogeneous model architectures an operational inevitability.

The question is not *whether* organizations will operate across model tiers. The question is *how* — deliberately, with intelligent orchestration, or chaotically, with fragmented tooling and no coherent strategy.

Today's multi-model deployments typically shuffle between frontier providers: GPT-4 for reasoning, Claude for writing, Gemini for multimodal. This is switching between frontier options, not architectural transformation. Multi-model across *tiers* is fundamentally different. You're routing a 50ms classification to a 125M parameter edge model, a code completion to a 3B fine-tuned model, and only escalating to frontier when the task demands it. When frontier models command $2.50-15 per million tokens and specialized SLMs run at $0.15-0.70, the cost differential can reach 20-100x depending on model choice — and self-hosted deployments push that further. The latency differential can be substantial. The compliance differential can be binary.

This heterogeneity demands orchestration intelligence: systematic rubrics for decomposition, compliance-gated routing, cross-boundary learning.

This paper introduces **Model Context Intelligence (MCI)**: an architectural pattern for multi-model orchestration. MCI synthesizes ideas from agent frameworks, model routing research, and enterprise patterns into opinionated positions on decomposition (SCALE rubric), routing (compliance gate then CLASSic optimization), context management, and cross-boundary learning. It complements existing SDKs like Microsoft Agent Framework, LangChain, and CrewAI, providing the decision logic they leave open.

1. The Forcing Functions

Four independent pressures are converging on the same conclusion: heterogeneous model architectures are inevitable.

1.1 Infrastructure Concentration Is a Strategic Risk

The current AI architecture concentrates risk at every layer.

Nvidia controls approximately 80% of the AI accelerator market. TSMC fabricates these chips. Advanced packaging capacity (CoWoS) is fully booked through 2025 and constrained through 2026. The April 2024 Taiwan earthquake caused only brief production pauses (TSMC's earthquake-resistant designs held), but it highlighted the concentration risk. The more pressing concern is geopolitical: Taiwan Strait tensions could disrupt access to capacity that cannot be quickly replicated elsewhere.

Most production applications route every request through frontier models requiring this concentrated infrastructure. When your calendar assistant, your code reviewer, your fraud detector, and your research agent all depend on the same GPU clusters, you've created correlated failure modes across your entire AI portfolio.

McKinsey projects $6.7 trillion in data center infrastructure investment needed by 2030 to meet compute demand, with AI workloads driving the majority of growth. But building infrastructure takes years. The gap between demand and capacity is structural, not temporary.

This is not a risk you can insure against. It is a risk you must architect around.

1.2 The Economics Have Crossed Over

Fine-tuning and training specialized models used to be prohibitively expensive — a frontier-lab luxury. That has changed.

The cost of fine-tuning a domain-specialized 3B-7B parameter model has dropped by orders of magnitude since 2022. Open base models (Llama, Mistral, Phi) reduce the need to train from scratch. LoRA and QLoRA enable fine-tuning on commodity GPUs — a single training job for an 8B model can cost as little as $30-500 depending on dataset size and compute choices. A small team can now produce a domain-specialized model for a fraction of what enterprise frontier API spend would cost for equivalent task volume.

Meanwhile, frontier inference costs add up. Stanford's 2025 AI Index reports that per-token costs dropped 280-fold between 2022 and 2024, yet enterprise AI budgets continue rising as usage growth outpaces efficiency gains. Frontier models still command $2.50-15 per million tokens for capable reasoning models, while specialized SLMs through inference providers run $0.15-0.70 per million — and self-hosted models approach near-zero marginal cost after hardware amortization.

The crossover has occurred: for high-volume, bounded tasks, *building* a specialized model is now cheaper than *renting* frontier inference. And unlike API spend, the specialized model is an asset you own, deploy where you choose, and improve over time.

1.3 Accuracy Favors Specialization

General-purpose models are jacks of all trades. On bounded domain tasks, specialized models increasingly beat them.

This is not surprising. A 3B parameter model fine-tuned on millions of medical case studies develops deeper pattern recognition for clinical reasoning than a 500B model trained on general web text. The smaller model has seen more relevant examples per parameter. Its weights are tuned for the task, not diluted across everything from poetry to Python.

Recent benchmarks illustrate the trend:

Model	Size	Domain / Task	Observation
Microsoft Phi-3-mini	3.8B	General reasoning	Competitive with Mixtral 8x7B and GPT-3.5 on many benchmarks
Fine-tuned Llama variants	7-8B	Text-to-SQL (Spider)	Approaching or matching GPT-4 accuracy with domain tuning
Granite-Code models	3-8B	Code generation	Strong HumanEval performance relative to size
MobileLLM	125M-350M	On-device classification	

Model	Size	Domain / Task	Observation
			Enables sub-200ms inference on mobile hardware
Domain fine-tuned SLMs	1-8B	Legal, medical, financial	Often outperform base models 10-40x larger on domain tasks

For tasks where you can define the domain and curate training data, frontier is often not the accuracy-maximizing choice — it's just the default.

1.4 Compliance Has No Frontier Option

Some environments cannot use frontier models at all.

Classified government systems cannot send data to commercial APIs. Healthcare organizations processing PHI need deployment configurations that meet specific HIPAA requirements. Financial institutions face data residency constraints that preclude certain cloud regions. Edge deployments — field offices, mobile platforms, disconnected operations — have no reliable connectivity to frontier APIs.

In these environments, SLMs and MLMs are not a compromise or a fallback. They are the only viable path to AI capability. A fine-tuned 7B model running in your enclave is infinitely more useful than a frontier model you cannot legally or physically access.

And as AI becomes essential to operations, "we can't use AI here" stops being acceptable. The pressure to bring capability inside the boundary — the enclave, the air gap, the edge device — will only grow.

1.5 Convergence

Each forcing function independently justifies specialized model adoption:

Forcing Function	Implication
Infrastructure risk	Reduce dependency on concentrated frontier supply chain
Economic crossover	Build specialized models instead of renting frontier at scale

Forcing Function	Implication
Accuracy crossover	Outperform frontier on domain tasks with right-sized models
Compliance reality	Deploy where frontier cannot go

Together, they make heterogeneous model architectures — spanning micro models at the edge, SLMs on-premises, and frontier in the cloud — not an optimization opportunity but an operational inevitability.

The question is not whether organizations will operate across model tiers. The question is whether they will do it deliberately, with intelligent orchestration, or chaotically, with fragmented tooling and no coherent routing strategy.

2. The Orchestration Gap

The specialized models exist. The integration protocols exist (MCP). The agent frameworks exist (LangChain, Microsoft Agent Framework, CrewAI). What's missing is the decision intelligence to coordinate them.

Multi-Model Today vs. Multi-Model Across Tiers

Today's multi-model deployments typically involve switching between frontier providers — GPT-4 for reasoning, Claude for writing, Gemini for multimodal. This is selecting among frontier options, not architectural transformation. The cost differentials are modest. The capability gaps are narrow. The compliance postures are similar. You're choosing between $15-30 per million token options based on vibes or vendor preference.

Multi-model across *tiers* is fundamentally different. You're not choosing between frontier options. You're routing:

- A 50ms classification to a 125M parameter model on the edge
- A code completion to a 3B fine-tuned model running on-premises
- A complex legal analysis to a 70B model in a FedRAMP environment
- A novel research synthesis to frontier — because this is the 5% of tasks that actually need it

The cost differential can reach 20-100x depending on model pairing. The latency differential can be an order of magnitude. The compliance differential can be binary — possible versus impossible.

This heterogeneity demands orchestration intelligence:

Decomposition: When should a complex task be broken into subtasks that can route to different model tiers? A contract review might decompose into clause extraction (SLM), risk classification (SLM), and complex liability analysis (frontier) — but only if the orchestrator understands task structure.

Routing: Which sub-agent — wrapping which model in which deployment — should handle this specific task? The answer depends on cost, latency, accuracy requirements, and compliance constraints, evaluated in that order of priority.

Context: How do you maintain state across a workflow that spans edge, on-prem, and cloud models with different security boundaries? The edge model's output becomes input to the cloud model, but the workflow state must persist coherently.

Learning: How do you improve routing decisions over time without leaking content across trust boundaries? A classified enclave can share that "decomposition pattern X works well for task type Y" without revealing what X or Y contained.

What Exists vs. What's Missing

Agent frameworks provide primitives for building multi-agent systems. They leave orchestration decisions to the implementer:

Framework	Provides	Leaves Open
LangChain	Tool chaining, agent patterns, memory abstractions	Decomposition rubrics, routing logic, compliance gates
Microsoft Agent Framework	Agent SDK, orchestration patterns, MCP support	Task-to-tier mapping, cross-boundary learning
CrewAI	Role-based coordination, task decomposition	Compliance-aware routing, performance optimization
AutoGen		

Framework	Provides	Leaves Open
	Multi-agent conversation, learning	Security boundary handling, production hardening

For environments where compliance is mandatory and auditability is required, "figure it out" isn't sufficient. You need systematic rubrics for decomposition, compliance-gated routing, and cross-boundary learning.

That decision intelligence is what MCI provides.

3. Introducing Model Context Intelligence

Model Context Intelligence is an architectural pattern for coordinating specialized sub-agents across heterogeneous model tiers — not a single product or implementation.

What MCI Is (and Isn't)

MCI is not a claim of invention. The component ideas exist across multiple frameworks and research efforts: LangChain pioneered tool chaining and agent patterns; AutoGen introduced multi-agent coordination with learning; CrewAI developed task decomposition and crew orchestration; vLLM's Semantic Router (now Signal-Decision) demonstrated learned model routing; Ray Serve and Kubernetes established patterns for distributed inference serving.

What MCI contributes is synthesis and opinion. We take specific positions on how these capabilities should combine for enterprise and mission-critical deployments: which rubrics for decomposition (SCALE) and routing (CLASSic), how context should persist, how learning should work across security boundaries, what the component boundaries should be. These are opinionated choices, informed by regulated-industry requirements, that are less explicitly codified in existing frameworks.

Why "Model Context Intelligence"?

The naming is deliberately parallel to **Model Context Protocol (MCP)**. Where MCP solves tool and data integration (giving models access to external capabilities), MCI solves orchestration intelligence: deciding which sub-agents handle which tasks, maintaining context across workflows, and learning from outcomes.

The term captures three essential dimensions:

Model: The architecture orchestrates across multiple models of different sizes, architectures, and specializations — from micro language models at the edge to frontier models in secure clouds.

Context: Intelligence is context-aware across multiple dimensions: conversation history, workflow state, user preferences, security constraints, resource budgets, and observed performance patterns.

Intelligence: The architecture exhibits adaptive intelligence through recursive learning. It observes its own execution, learns from outcomes, and continuously optimizes decomposition strategies, routing decisions, failure recovery policies, and synthesis approaches.

MCP and MCI are complementary layers in a complete AI architecture. MCP provides the foundation for tool access; MCI provides the intelligence for orchestration.

MCP as Multi-Layer Protocol

MCP operates across all five layers of the architecture, not just as a foundation for tool access:

Layer	MCP Role
Layer 1 (Foundation)	Sub-agents access tools, APIs, and data sources via MCP servers
Layer 2 (Model Tier)	Model endpoints can expose capabilities as MCP tools
Layer 3 (Sub-Agent)	Sub-agents can expose themselves as MCP servers for direct invocation
Layer 4 (Orchestration)	MCI orchestrator consumes MCP servers for routing context (compliance status, user preferences, security labels) and can expose itself as an MCP server for external callers
Layer 5 (Application)	Applications can interact with MCI through MCP rather than custom APIs

This multi-layer presence means MCP is the connective tissue of the architecture — not just how sub-agents access tools, but how components communicate, how external systems integrate, and how context flows. The orchestrator consuming MCP servers for compliance status means the compliance gate can query live policy systems rather than relying on stale configuration.

MCI provides the decision intelligence. MCP provides the protocol. Together they form a complete orchestration architecture where routing decisions are informed by real-time context from across the system.

---

4. Five-Layer Architecture

Before diving into components, here's how MCI fits into the overall stack:

Layer 5 - Application Interface: REST/GraphQL/Streaming APIs. Abstracts all orchestration complexity from applications.

Layer 4 - MCI Orchestration: Eight core components that decompose tasks, route to sub-agents, manage context, coordinate execution, and learn from outcomes. This is where intelligence lives.

Layer 3 - Sub-Agent Layer: Specialized sub-agents implementing a standard SDK interface. Examples: context retrieval, calendar analysis, threat detection, code generation. Each sub-agent wraps one or more models with domain-specific logic.

Layer 2 - Heterogeneous Model Tier: Micro models (thousands to tens of millions of parameters, for edge and embedded), small models (hundreds of millions to a few billion, for local inference), and frontier models (tens of billions and up, for complex reasoning). Accessed via sub-agents, not directly.

Layer 1 - MCP Foundation: Tool and data integration layer. Sub-agents access external tools and data sources via MCP through the SDK.

Sub-Agents and Models are Decoupled

A critical architectural distinction: sub-agents and models are not the same thing. Sub-agents are deployment units that wrap models. The same model can power multiple sub-agents with entirely different characteristics.

Consider a Llama 70B model deployed four ways:

Sub-Agent	**Model**	**Deployment**	**Compliance**
Legal-Analysis-Gov	Llama 70B	Azure Government	FedRAMP High, IL4
Legal-Analysis-Commercial	Llama 70B	AWS US-East	SOC2
Legal-Analysis-EU	Llama 70B	AWS Frankfurt	GDPR
Legal-Analysis-Classified	Llama 70B	On-prem enclave	IL5

Same model. Same fine-tuning. Four different sub-agents with different compliance postures, data residency, and availability characteristics.

This decoupling matters because MCI routes to sub-agents, not models. The routing decision considers both deployment properties (compliance, residency, availability) and observed performance (cost, latency, accuracy). A sub-agent's performance reflects the combination of its underlying model and its deployment environment.

Sub-agents encapsulate capability, not just access. The wrapper ecosystem's frustration is that you can connect to dozens of models but lose critical capabilities when you leave the major providers. Grounding (RAG), tool use, structured outputs, function calling: these aren't standardized across models. A sub-agent solves this by encapsulating the model plus its capability bindings. The RAG pipeline, the tool definitions, the output schemas, the prompt templates that make a particular model work well for a particular task: all of that lives in the sub-agent. The orchestration layer routes to a sub-agent that can do "contract analysis with clause extraction." It doesn't know or care whether that sub-agent uses RAG, fine-tuning, or a 200-line system prompt to get there. This isolation means you can swap implementations, upgrade models, or change grounding strategies without touching orchestration logic.

Hybrid Coordination: Control Plane + Distributed Execution

MCI's architecture is deliberately hybrid: centralized decision-making with distributed execution.

Layer	Coordination Model	Rationale
Policy & Compliance	Centralized (Context Manager + Compliance Gate)	Non-negotiable. Gates, audit, intervention capability.
Task Routing	Centralized (Intelligent Router)	Requires global view of sub-agent capabilities, load, compliance status.
Task Execution	Distributed (Sub-agents do the work)	Parallelism, specialization, horizontal scale.
Agent Collaboration	Peer-to-peer (within authorized workflows)	Lightweight handoffs, clarification, intermediate artifacts.

This isn't a contradiction — it's separation of concerns. Agents can communicate directly during execution (share intermediate results, request clarification, hand off subtasks) without routing everything through a central coordinator. But they cannot *start* work or *complete* work without control plane authorization.

The control plane isn't a bottleneck — it's a checkpoint. And in compliance-constrained environments, checkpoints are features, not bugs.

Some architectures optimize for autonomous agent coordination, letting agents self-organize through messaging protocols, @mentions, and emergent collaboration. This works well for loosely coupled workflows where compliance is advisory. But **scaling self-management is incongruent with regulation and compliance.** When you need to prove that PHI never touched a non-HIPAA system, or that classified data never left the enclave, you need a control plane that enforced that constraint — not a log that shows it happened to work out.

5. The Eight Components of MCI

MCI comprises eight core components. Each has a specific role in the orchestration flow.

Component 1: Workflow Orchestrator (Go)

The Workflow Orchestrator is the entry point for every request. Its job: decide whether to decompose a task into subtasks, and if so, how.

Language choice: Go. Workflow orchestration engines like Temporal and Cadence are built in Go. The language offers strong concurrency, simple deployment, and adequate performance for decision logic.

Alternatives to consider: Python if the orchestrator relies heavily on LLM-based classification for SCALE assessment, since Python offers tighter integration with ML tooling. Java/Kotlin for enterprises with existing JVM infrastructure and teams.

This is not a trivial decision. Decomposition adds orchestration overhead. For simple tasks, sending directly to a capable model may be faster and cheaper. For complex tasks, decomposition enables parallel execution and right-sized model selection. For safety-critical tasks, decomposition may be too risky — or may require redundant validation paths.

The SCALE Rubric

To make this decision systematically, the Workflow Orchestrator applies the **SCALE** rubric:

Dimension	Question
Structure	Is this task naturally decomposable into meaningful subtasks?
Consequence	What happens if we fail?
Accuracy	What error tolerance exists?
Latency	What are the time constraints?
Experience	How proven is this pattern? (Volume + Confidence)

How SCALE Gets Applied

The Orchestrator assesses each dimension through a combination of classification, policy lookup, and historical data:

Step 1: Assess each dimension

Dimension	Assessment Method	Output
Structure	LLM classifier or pattern match against known task types	Atomic / Decomposable / Unknown
Consequence	Policy config + task metadata (e.g., domain tags, user role)	Advisory / Operational / Safety-Critical / Life-Safety
Accuracy	Derived from task type or explicit requirement in request	Approximate OK / Must Be Correct / Zero Defect
Latency	Explicit SLA in request or inferred from context	Real-time (<500ms) / Interactive (<5s) / Batch (>5s)
Experience	Lookup against decomposition template history	High (>1000 executions, >95% success) / Medium / Low / Novel

Step 2: Apply decision logic

The Orchestrator applies these assessments through a decision hierarchy where Consequence dominates.

For life-safety tasks, the Orchestrator only decomposes if there's a high-experience template with redundant validation built in. Otherwise, it routes the entire task to the highest-accuracy model available because decomposition risk is too high.

For atomic tasks (those that can't meaningfully be broken apart), the Orchestrator skips decomposition entirely.

For tasks with low experience and operational or higher consequence, the Orchestrator avoids decomposition due to insufficient confidence in the pattern.

For decomposable tasks with adequate experience, the Orchestrator applies the best-match template and parallelizes subtasks where latency requirements demand it and dependencies allow.

For unknown task structures, the Orchestrator routes to a frontier model for single-pass handling and logs the interaction for pattern discovery.

Step 3: Build execution plan

If decomposing, the Orchestrator: - Selects decomposition template (or synthesizes from similar patterns) - Identifies subtask dependencies (A must complete before B) - Determines parallelization opportunities - Passes plan to Sub-Agent Coordinator

Key Insight: Consequence as Override

When tasks have safety-of-navigation or loss-of-life implications, Consequence dominates all other factors. Accuracy becomes paramount. Cost becomes almost irrelevant. The Orchestrator may bypass decomposition entirely or mandate redundant validation paths regardless of efficiency.

This is why SCALE treats Consequence as a governing constraint, not just another weighted factor.

Component 2: Intelligent Router (Rust)

Once the Orchestrator has decided to decompose (or not), each task or subtask needs to be routed to a sub-agent. The Intelligent Router makes this decision.

Language choice: Rust. The Router sits on the critical path of every request. Industry precedent from vLLM Semantic Router and Envoy demonstrates that latency-critical routing benefits from Rust's performance characteristics and memory safety guarantees.

Alternatives to consider: Go is viable if latency requirements are relaxed (interactive rather than real-time). We do not recommend higher-level languages for this component; the Router's position on the critical path makes performance non-negotiable.

Routing to Sub-Agents, Not Models

Because sub-agents are deployment units that wrap models, routing evaluates the whole sub-agent, not the underlying model in isolation. A sub-agent's performance reflects the combination of its model capability and its deployment context.

The Router applies a two-phase selection process: compliance gating, then performance optimization.

Phase 1: Compliance Gate (Pass/Fail)

Compliance is a hard gate. Before any optimization, the Router filters to eligible sub-agents only.

Compliance encompasses all regulatory and policy requirements: Does the sub-agent meet required regulatory standards (HIPAA, PCI-DSS, FedRAMP, SOC2)? Can it handle the data's classification or sensitivity level? Does data stay within required jurisdictions? Does it have required certifications?

Any sub-agent that fails compliance requirements for the task is excluded. Only compliant sub-agents proceed to Phase 2.

The compliance gate is not just a filter — it's the architectural primitive that makes MCI viable for regulated environments. Without it, routing optimization could inadvertently select a faster, cheaper sub-agent that violates data residency requirements. The gate ensures that performance optimization only occurs within the set of compliant options.

This is why compliance is a gate (pass/fail) rather than a dimension in CLASSic scoring. You cannot trade compliance for performance. A sub-agent that's 50% faster but violates HIPAA isn't a performance win — it's a regulatory violation. The two-phase architecture makes this constraint structural, not policy-dependent.

Phase 2: CLASSic Performance Scoring

For compliant sub-agents, the Router applies **CLASSic** (Aisera, ICLR 2025) to evaluate observed performance:

Dimension	What It Measures
Cost	

Dimension	What It Measures
	Operational expenses: API usage, token consumption, infrastructure
Latency	End-to-end response times
Accuracy	Correctness for this task type
Stability	Consistency across diverse inputs and conditions
Security	Resilience against adversarial inputs, prompt injections, data leaks

These dimensions are measured at the sub-agent level as observed in production, not abstract model benchmarks. A sub-agent's CLASSic scores reflect the combination of its underlying model and its deployment environment.

Model Benchmarks vs. CLASSic: Design-Time vs. Runtime

Two levels of evaluation matter here:

Model benchmarks (MMLU, HumanEval, MATH, GPQA, SWE-Bench) are agent-agnostic tests of raw model capability. They answer questions like: Is this model good at code generation? How well does it reason about math? These benchmarks inform **design-time decisions** — specifically which model to wrap when building a sub-agent for a particular task type. A code-review sub-agent might wrap a model with strong HumanEval scores; a knowledge-QA sub-agent might prioritize MMLU performance.

CLASSic evaluates sub-agent performance across operational dimensions. It answers questions like: How does this sub-agent actually perform in production on cost, latency, accuracy, stability, and security? CLASSic informs **runtime routing decisions** — specifically which sub-agent to select for a given task.

Model benchmarks are inputs to sub-agent design. CLASSic is the basis for sub-agent selection. The Router operates on CLASSic scores observed in production, not raw model benchmarks.

Dimension	Scoring Method	Score Range
Cost	Observed cost per task type	$ (lower is better)

Dimension	Scoring Method	Score Range
Latency	Historical p95 latency for similar task types	ms (lower is better)
Accuracy	Historical success rate for this task type	0.0 - 1.0 (higher is better)
Stability	Variance in accuracy over trailing window	σ (lower is better)
Security	Historical resistance to adversarial probes	0.0 - 1.0 (higher is better)

Phase 3: Apply Workflow Weights

Different workflows prioritize differently. The Router applies weights from the workflow configuration, computing a weighted score across all five dimensions. The sub-agent with the highest combined score is selected.

Example weight profiles:

Workflow Type	Cost	Latency	Accuracy	Stability	Security
Real-time chat	0.1	0.4	0.25	0.1	0.15
Financial analysis	0.1	0.1	0.4	0.2	0.2
Bulk processing	0.4	0.1	0.3	0.1	0.1
Safety-critical	0.0	0.1	0.5	0.2	0.2

Hybrid Routing: Compliance as Gate, Performance as Gradient

The Router implements a hybrid approach: - **Compliance gate** enforces regulatory and policy requirements, auditable and explainable to regulators - **CLASSic optimization** improves performance based on observed outcomes, providing adaptive efficiency within compliant boundaries

Compliance is a gate. Performance is a gradient.

Component 3: Context Manager (Rust)

The Context Manager maintains all state across the workflow. Sub-agents are stateless by design — they receive context, do work, return results. The Context Manager owns the memory.

Language choice: Rust. Context is critical state. Corruption or race conditions in context management can cause cascading failures across the entire workflow. Rust's ownership model and compile-time guarantees provide correctness assurances that matter for this component.

Alternatives to consider: Go with careful concurrency design could work, but requires discipline that Rust enforces at compile time. If the Context Manager is primarily an application layer over a proven store like Redis or etcd, the language matters less because the storage layer handles correctness. We do not recommend dynamically-typed languages for this component.

What It Tracks

Context Type	Examples	Persistence
Conversation history	Full message log across turns	Indefinite (default)
Workflow state	Subtask completion, intermediate results, dependencies	Request-scoped
User preferences	Writing style, risk tolerance, formatting	Cross-session
Security constraints	Classification level, allowed operations	Policy-defined
Resource budgets	Token limits, cost ceiling, latency SLA	Request-scoped

Operational vs. Forensic Auditability

A critical distinction: MCI provides **operational auditability**, not merely forensic auditability.

Forensic auditability means you can reconstruct what happened after the fact — trace the message chain, analyze logs, piece together the workflow post-hoc. By the time you've reconstructed it, the action already happened. The data already moved. The decision already executed.

Operational auditability means you know what's happening right now, why it's happening, and you can intervene before execution. The Context Manager provides:

- **Pre-execution visibility:** What workflows are active, what state each is in, what agents are engaged
- **Real-time intervention:** Stop a workflow before a non-compliant action executes
- **Policy enforcement at decision time:** Compliance gates fire before routing, not after

Distributed coordination approaches (where agents self-organize through peer-to-peer messaging) can achieve forensic auditability — you can trace the @mention chain after the fact. But they struggle with operational auditability because there's no central point that knows the full workflow state in real time.

For regulated environments, forensic reconstruction isn't sufficient. You need a control plane that can answer "what is happening right now?" and "should this be allowed to proceed?" The Context Manager, combined with the compliance gate, provides that control plane.

Architectural Position: MCI Owns Memory

Default to indefinite persistence. Too many cycles with models are spent relearning things already known. If MCI is the orchestration layer, then MCI is where context lives — not the models. The models are stateless workers.

Regulated environments can configure retention policies (flush after request, TTL, scope limits) as overrides. But the architectural default assumes persistence because relearning is wasteful.

Representation: Declarative Structure, Learned Content

Structure is declarative. The categories of context (user preferences, workflow state, conversation history, constraints) are schema-defined — inspectable, queryable, auditable.

Content can be learned. Within those categories, content may use learned representations: summaries for long histories, embeddings for semantic retrieval, compression for efficiency.

You always know what kinds of context exist. How that context is internally represented can vary.

Component 4: Sub-Agent Coordinator (Go)

The Coordinator executes the plan created by the Orchestrator, invoking sub-agents in the correct order and handling failures.

Language choice: Go. Goroutines and channels are ideal for managing concurrent sub-agent invocations with clean cancellation and timeout handling. Go's concurrency model maps naturally to parallel and conditional execution patterns.

Alternatives to consider: Elixir/Erlang (OTP) if fault tolerance is the primary design constraint, since the "let it crash" supervision model excels at failure recovery. Java with virtual threads (Project Loom) for enterprises with JVM infrastructure. Rust with Tokio for deployments already using Rust elsewhere.

Execution Modes

Mode	When Used	Behavior
Sequential	Subtask B depends on output of A	Execute A, wait, execute B
Parallel	Subtasks A and B are independent	Execute simultaneously, reduce latency
Conditional	Subtask B only needed if A returns certain result	Evaluate A output, decide on B

Failure Handling

The Coordinator implements adaptive failure handling. When a sub-agent fails, the Coordinator first attempts retries if any remain, since the failure may be transient. If retries are exhausted, it checks whether an alternative sub-agent can handle the task and routes accordingly. If no alternative exists but the workflow can tolerate partial results, the Coordinator continues with a degraded response. If human escalation is configured, it queues for review. Only when all recovery options are exhausted does the workflow fail with a clear error.

Failure strategies are learned over time. The Adaptive Learning System observes which recovery approaches work for which failure types and updates Coordinator policies.

Component 5: Result Synthesizer (Go)

When a workflow involves multiple sub-agents, their outputs must be combined into a coherent response. The Synthesizer handles this.

Language choice: Go. Rule-based synthesis is straightforward business logic that doesn't require specialized language features. Using the same language as the Coordinator simplifies deployment and team expertise.

Alternatives to consider: Python if synthesis requires ML-assisted merging or semantic understanding beyond rule-based policies. In that case, the Synthesizer may call ML models for intelligent merging rather than implementing it directly.

Synthesis Challenges

- Different sub-agents may use different formats or terminology
- Outputs may partially overlap or conflict
- Confidence levels vary across sub-agents
- User expects unified response, not a list of fragments

Conflict Resolution

Conflicts are resolved via pre-configured MCI policies:

Conflict Type	Resolution Strategy
Factual disagreement	Prefer higher-confidence sub-agent; flag uncertainty
Format mismatch	Normalize to workflow-specified format
Partial overlap	Deduplicate, merge unique information
Missing subtask output	Note gap if critical, omit if optional

The Synthesizer does not make judgment calls — it follows policy. This keeps synthesis auditable and predictable.

Component 6: Response Validator (Go)

Before returning results, the Validator checks that outputs meet requirements.

Language choice: Go. Schema validation and pattern matching are well-supported, and using the same language as adjacent components simplifies the deployment footprint.

Alternatives to consider: This component is not typically standalone; validation logic often embeds in the API layer or Synthesizer. The language should match wherever it lives. If safety scanning requires ML-based content classification, that specific check may call out to a Python service.

Validation Checks

Check	Description	Action on Failure
Schema compliance	Output matches expected structure	Reject, retry synthesis
Completeness	All required fields present	Reject, identify missing subtask
Confidence threshold	Aggregated confidence meets minimum	Flag for review or reject
Safety scan	No prohibited content in output	Reject, log for review

---

Component 7: Resource Budget Manager (Go)

The Budget Manager tracks resource consumption against limits and enforces constraints.

Language choice: Go. Counter aggregation, threshold monitoring, and circuit breaker patterns are well-supported with low overhead. Go's simplicity keeps this component lightweight.

Alternatives to consider: This is often implemented as a library rather than a standalone service, embedded in the Coordinator or other components. In that case, use the host component's language. For distributed deployments, consider a sidecar pattern with the language matching your service mesh.

What It Tracks

Resource	Tracking Method	Constraint Action
Tokens	Sum across all sub-agent calls	Warn at 80%, hard stop at limit
Cost	Pricing × token usage	Warn at 80%, hard stop at limit
Latency	Wall-clock time from request start	Trigger early synthesis if approaching SLA
API calls	Count per external service	Rate limit, queue, or reject

Circuit Breakers

If a workflow is consuming resources at an unexpected rate (runaway decomposition, retry loops), the Budget Manager can trigger circuit breakers. When cost rate exceeds three times the expected rate, the workflow pauses, the operator is alerted, and the system awaits manual approval or timeout before proceeding.

Component 8: Adaptive Learning System (Python)

The Learning System observes outcomes and improves routing, decomposition, and coordination over time.

Language choice: Python. The ML ecosystem (PyTorch, scikit-learn, pandas, statistical libraries) lives in Python. There is no viable alternative for ML workloads at this level of sophistication.

Alternatives to consider: Scala/Spark for big data scale if telemetry volume exceeds what single-node Python can handle. Julia for numerical computing in specialized cases. We do not recommend Go or Rust for this component because the ML ecosystem gap is too significant.

Why Separate?

The Learning System operates on a **batch cadence**, not inline with requests. It ingests telemetry, analyzes patterns, and periodically updates policies. This separation keeps request latency predictable while enabling sophisticated ML. Go and Rust handle request-path performance; Python handles offline learning.

What It Learns

Learning Target	**Input Signals**	**Output**
Decomposition patterns	Task types, template success rates, failure modes	Updated template rankings, new template suggestions
Routing optimization	Sub-agent latency/accuracy/ cost per task type	Updated CLASSic weights, sub-agent rankings
Failure recovery	Failure types, recovery attempts, outcomes	Updated Coordinator retry/ fallback policies
Elicitation patterns	User clarification interactions, final task understanding	Improved workflow discovery questions

Cross-Enclave Learning: Patterns Only

In environments with security boundaries (defense, healthcare), learning operates on **patterns only, never content**:

- ✅ "Decomposition template X succeeded 94% of the time for task type Y"
- ✅ "Sub-agent A has 50ms lower latency than B for classification tasks"
- ❌ "User asked about [classified content]"
- ❌ "Sub-agent returned [PHI data]"

This approach is compliant by design. It offers different tradeoffs than federated learning or differential privacy: simpler to implement and audit, but less mathematically rigorous in its privacy guarantees. For environments where the primary concern is preventing content leakage across trust boundaries rather than statistical privacy, patterns-only learning may be sufficient.

---

6. Sub-Agent SDK

The Sub-Agent SDK is a core deliverable — the standard interface that enables ecosystem development.

What It Defines

Aspect	Specification
Interface contract	Input schema, output schema, capability declaration
Context access	Read-only access to relevant context slices
Telemetry emission	Required metrics, latency reporting, confidence scores
Invocation rules	Sub-agents cannot invoke other sub-agents; only Coordinator can
MCP integration	Standard patterns for accessing tools and data via MCP

Why This Matters

A well-defined SDK enables: - Domain experts to build sub-agents without understanding MCI internals - Sub-agent marketplace where components are reusable across MCI deployments - Clear contracts for testing, validation, and certification - Ecosystem growth independent of any single vendor

---

7. Workflow Discovery: Learning Decomposition Patterns

A critical question: where do decomposition templates come from?

Position: Hybrid with Active Elicitation

Workflow discovery is human-guided but machine-assisted. Rather than asking users to validate proposed decompositions once, the system engages in **iterative clarification** — asking the same underlying question in multiple formulations to reduce noise and surface true intent.

Example elicitation sequence for a task "analyze this contract":

1. "Does this require extracting specific clauses, or understanding overall risk?"
2. "If I found concerning terms, should I flag them or summarize the whole document?"
3. "Would a clause-by-clause breakdown be more useful than a risk summary?"

Each question probes the same underlying need (granularity of analysis) but from different angles. The pattern of responses reveals true intent more reliably than a single question.

How This Reduces Fragility

Single-question validation produces brittle patterns: - User says "yes, decompose into clauses" - Template hardcodes clause extraction - Different user with similar request needed risk summary - Template fails

Multi-formulation elicitation builds durable patterns: - System learns that "contract analysis" has two common intents - Template includes conditional logic based on elicited signals - Both use cases succeed

The Adaptive Learning System observes which elicitation sequences yield the most durable templates and refines the questioning approach itself.

---

8. Architectural Precedent: From Microservices to Control Planes

The pattern MCI follows is not new. It mirrors the evolution that transformed enterprise software over the past two decades.

Phase 1: Decomposition (Microservices)

In the 2000s, enterprises learned that monolithic applications couldn't scale. The solution was decomposition: break the monolith into specialized services, each doing one thing well, communicating through defined interfaces. This enabled independent scaling, independent deployment, technology diversity, and failure isolation.

Phase 2: Orchestration (Kubernetes)

Decomposition created a new problem: managing hundreds of services across thousands of containers. The solution was a control plane — Kubernetes — that handled scheduling, scaling, self-healing, and service discovery. The control plane abstracted infrastructure complexity, letting developers focus on services rather than servers.

AI is following the same path:

Era	Software	AI
Monolith	Single binary, uniform scaling	Single frontier model for everything
Decomposition	Microservices, specialized components	Specialized models, sub-agents
Control Plane	Kubernetes orchestrates containers	MCI orchestrates models

What MCI learns from Kubernetes:

- **Declarative intent.** Kubernetes manages desired state, not imperative commands. MCI manages task intent, not explicit model calls.
- **Scheduling intelligence.** Kubernetes places workloads based on resource requirements and constraints. MCI routes tasks based on SCALE and CLASSic assessments.
- **Self-healing.** Kubernetes restarts failed containers and reschedules workloads. MCI retries failed sub-agents and routes to alternatives.
- **Observability.** Kubernetes exposes metrics, logs, and traces. MCI emits telemetry for learning and debugging.
- **Abstraction.** Kubernetes abstracts infrastructure from applications. MCI abstracts sub-agent selection from applications.

Organizations that survived the microservices transition, and then the Kubernetes transition, know how to operate MCI systems. The patterns are familiar: decompose by capability, orchestrate through a control plane, observe everything, handle failures gracefully, optimize continuously.

9. Domain Vignettes: MCI Across Industries

The patterns MCI describes apply wherever organizations face constraints on cost, compliance, latency, or data boundaries. The following illustrative scenarios show how the same architectural decisions manifest in different contexts. These are not case studies — they are thought experiments demonstrating MCI's applicability across domains.

Vignette 1: Commercial SaaS (Multi-Tenant Cost Optimization)

A B2B SaaS platform provides AI-powered document analysis to thousands of customers. Their challenge: AI costs are eating margin, but customers expect instant results.

The Problem. Every document, from a two-page invoice to a 200-page contract, routes through the same frontier model. Simple classification tasks cost the same as complex legal analysis. Customers on the $50/month tier consume the same inference as enterprise customers paying $5,000/month.

MCI Applied.

The Workflow Orchestrator applies SCALE to incoming documents. Structure assessment reveals that most documents decompose naturally: extract metadata, classify type, route specialized analysis. Experience data shows that 73% of documents are routine types (invoices, receipts, simple agreements) where specialized sub-agents match frontier accuracy.

The Intelligent Router's compliance gate is straightforward here — all sub-agents meet SOC2, all data stays in the platform's cloud. CLASSic optimization focuses on Cost and Latency, with Accuracy thresholds per document type.

The Adaptive Learning System discovers that certain customer segments (legal, healthcare) have higher accuracy requirements. It learns to weight Accuracy higher for those tenant profiles without explicit configuration.

Illustrative Impact. In this scenario, inference costs could drop significantly (potentially 50-70%) while latency improves substantially. The margin unlocked funds further model specialization, creating a flywheel effect.

Key MCI Contribution. SCALE's Experience dimension prevented premature optimization — novel document types still route to frontier until pattern confidence builds.

Vignette 2: Regulated Commercial (Healthcare Claims Processing)

A healthcare payer processes millions of claims monthly. AI could accelerate adjudication, but HIPAA compliance and accuracy requirements create hard constraints.

The Problem. Claims contain PHI that cannot leave approved environments. Different claim types require different expertise (pharmacy, surgical, behavioral health). Incorrect adjudication creates regulatory exposure and patient harm. The payer operates in multiple states with varying regulations.

MCI Applied.

The compliance gate dominates routing decisions. Before any performance optimization, the Router filters sub-agents by: HIPAA certification status, state-specific regulatory compliance, PHI handling authorization, and data residency requirements. A claim from a California Medicaid patient routes only to sub-agents certified for that specific regulatory intersection.

SCALE's Consequence dimension governs decomposition. Pharmacy claims (high volume, well-understood) decompose freely. Complex surgical claims with potential fraud indicators route atomically to specialized sub-agents — the risk of decomposition error exceeds the efficiency gain.

The Context Manager maintains claim history across the member's lifetime, enabling pattern detection (potential fraud, care gaps) while enforcing strict access boundaries. The Adaptive Learning System shares patterns across the enterprise ("claims with characteristic X have 3x denial rate") without sharing any PHI.

Illustrative Impact. Adjudication time could drop from days to hours for routine claims. Compliance audit preparation simplifies dramatically because every routing decision is logged with compliance justification. Accuracy on complex claims improves through specialized sub-agents tuned for specific claim types.

Key MCI Contribution. Compliance-gate-then-optimize architecture means the system cannot accidentally route PHI to non-compliant sub-agents, regardless of performance pressure.

Vignette 3: Government (Multi-Classification Intelligence Analysis)

An intelligence organization processes information across classification levels. Analysts need AI assistance, but data cannot cross security boundaries.

The Problem. Information exists at Unclassified, Secret, and TS/SCI levels. Analysts working at higher levels need to incorporate lower-level information. AI models at each level have different capabilities based on available training data. Connectivity between enclaves is restricted or nonexistent. Edge deployments (field offices, mobile platforms) have intermittent connectivity and constrained compute.

MCI Applied.

MCI deploys hierarchically: local coordinators at edge locations, enclave coordinators at each classification level, and no coordinator that spans levels. Each enclave operates its own complete MCI stack with sub-agents certified for that classification.

The compliance gate is absolute. A task tagged TS/SCI routes only to sub-agents in the TS/SCI enclave. There is no "almost compliant." The Router at each level has visibility only into sub-agents at that level — cross-enclave routing doesn't exist.

Edge deployments run with micro models locally. When connectivity exists, the Coordinator syncs patterns (not content) with the regional coordinator. When connectivity drops, local MCI continues operating on cached patterns and local sub-agents.

The Adaptive Learning System demonstrates its patterns-only constraint most clearly here. The TS/SCI enclave learns that certain analytical patterns have high success rates. That pattern knowledge ("decomposition template X works well for task type Y") propagates to lower enclaves. The underlying content never moves.

Illustrative Impact. Analysts at each level get AI assistance appropriate to their environment. Edge analysts maintain capability during disconnected operations. The organization's analytical tradecraft improves globally through pattern sharing without any data spillage.

Key MCI Contribution. Cross-enclave learning with patterns-only constraints enables organizational learning that would otherwise require impossible data sharing.

The Common Thread

These vignettes differ in specifics but share architectural needs:

Need	SaaS	Healthcare	Government
Compliance gate	SOC2	HIPAA + state regs	Classification
Consequence sensitivity	Low (cost focus)	High (patient impact)	Extreme (national security)
Learning constraints	Tenant isolation	PHI boundaries	Classification boundaries
Edge requirements	Minimal	Regional data centers	Disconnected operations

MCI's value is providing a consistent framework for reasoning about these decisions. The compliance gate is always first. SCALE always governs decomposition. CLASSic always optimizes within constraints. The patterns-only learning constraint always enables cross-boundary improvement.

The specific thresholds, sub-agents, and policies differ. The architecture remains constant.

10. Architectural Positions Summary

This paper takes specific positions on open questions. These represent our considered judgment, but we invite challenge and refinement.

Question	Position	Rationale
Multi-tier necessity	Inevitable, not optional	Four forcing functions converge: infrastructure risk, economic crossover, accuracy crossover, compliance reality
Sub-agent/model relationship	Decoupled; same model can power multiple sub-agents with different deployment properties	Compliance is a deployment property, not a model property
Routing targets	Route to sub-agents, not models; CLASSic measures observed sub-agent performance	Sub-agents combine model capability with deployment context
Routing architecture	Compliance gate, then CLASSic optimization	Compliance is pass/fail; performance is a gradient
Cold-start routing	Start conservative, optimize as data accumulates	Effective first, then efficient
Workflow discovery	Hybrid with multi-formulation elicitation	Reduces fragility, captures nuance
Cross-enclave learning	Patterns only, never content	Compliant by design, auditable
Context persistence	Default indefinite; MCI owns memory	Relearning is wasteful
Context representation	Declarative structure, learned content	Debuggable yet flexible
Coordination model	Centralized control plane, distributed execution	Scaling self-management is incongruent with compliance
Auditability	Operational (real-time), not just forensic (post-hoc)	Regulated environments need intervention capability

11. Related Work and Positioning

MCI exists in a crowded field. This section clarifies what MCI contributes relative to existing work and how it complements rather than competes with available tools.

The Distinction: SDKs vs. Decision Frameworks

Most prior work in this space provides **SDKs and toolkits** — libraries that give developers primitives for building agent systems. MCI provides an **architectural pattern** with opinionated guidance on how to use those primitives in production environments with real constraints.

Use LangChain, Microsoft Agent Framework, or CrewAI to build your system. Use MCI's rubrics to make operational decisions within that system.

Agent Frameworks and SDKs

Microsoft Agent Framework (October 2025) unifies Semantic Kernel and AutoGen into a single SDK for building, deploying, and managing multi-agent systems. It provides the AIAgent abstraction, orchestration patterns (sequential, concurrent, group chat, handoff), MCP support, and enterprise features like observability and durable execution. MCI is designed to work with, not replace, Microsoft Agent Framework. Where MAF provides the "how to build," MCI contributes the "how to decide": which rubrics for decomposition (SCALE), which evaluation framework for routing (CLASSic), how to structure compliance gates, and how to enable cross-boundary learning.

LangChain (2022+) established foundational patterns for chaining LLM calls with tools, introducing abstractions for prompts, memory, and agents. Its influence on the field is substantial. MCI adopts similar composability principles but focuses on architectural patterns and decision rubrics rather than implementation primitives.

AutoGen (Microsoft Research, 2023+) pioneered multi-agent conversation patterns with support for human-in-the-loop and learned behaviors. Now unified into Microsoft Agent Framework, its research contributions inform MCI's thinking on multi-agent coordination.

CrewAI (2023+) developed role-based agent coordination with task decomposition patterns. MCI's Workflow Orchestrator serves a similar function but applies the SCALE rubric for systematic decomposition decisions rather than role-based heuristics.

LangGraph extends LangChain with graph-based workflow definitions. MCI's workflow patterns could be expressed in LangGraph; our contribution is the specific rubrics and policies rather than the graph abstraction itself.

Model Routing and Serving

vLLM Semantic Router (evolved to Signal-Decision, November 2025) demonstrated learned routing between models based on query characteristics. MCI adopts similar routing concepts but adds the compliance gate as a prerequisite phase and structures performance optimization around CLASSic dimensions.

Ray Serve provides distributed model serving with autoscaling. MCI operates at a higher abstraction layer — Ray Serve could be an implementation substrate for MCI sub-agents.

Semantic Kernel (Microsoft, 2023+) offered plugin-based orchestration with planner capabilities before its unification into Microsoft Agent Framework.

Distributed Agent Coordination

Emerging platforms like **AX Platform** explore peer-to-peer agent collaboration, where agents self-organize through messaging, @mentions, and emergent coordination patterns. This approach optimizes for autonomy and horizontal scale — agents communicate directly without central orchestration.

MCI takes a different position: **centralized control plane with distributed execution**. For loosely coupled workflows where compliance is advisory, emergent coordination works well. For regulated environments where you must prove data never crossed boundaries, you need a control plane that enforced constraints — not logs that show constraints happened to hold.

The distinction is operational vs. forensic auditability. Distributed coordination can reconstruct what happened. Centralized control planes know what's happening and can intervene. Both are valid architectures for different constraint profiles.

Evaluation Frameworks

CLASSic (Aisera, ICLR 2025) provides the Cost, Latency, Accuracy, Stability, Security rubric that MCI adopts for sub-agent evaluation. We apply CLASSic with specific scoring methods and workflow weight profiles but did not create the framework. Our contribution is positioning CLASSic as a runtime optimization layer that operates after compliance gating — not as a complete routing solution.

Model benchmarks (MMLU, HumanEval, MATH, GPQA, SWE-Bench) inform sub-agent design decisions. MCI distinguishes design-time model selection (informed by benchmarks) from runtime sub-agent routing (informed by CLASSic).

Research on Model Routing

Leeroo Orchestration of Experts demonstrated training an orchestrator on benchmark performance for intelligent model routing. This represents static learned routing — pre-computing which model handles which query type based on benchmark runs. MCI's adaptive learning operates at runtime with production feedback, learning across multiple dimensions (not just accuracy) and adapting to observed performance.

What MCI Contributes

Given this context, MCI's contribution is synthesis and opinion for production environments:

Original contributions: - **SCALE rubric** for decomposition decisions (Structure, Consequence, Accuracy, Latency, Experience) with Consequence as governing override - **Two-phase routing architecture:** compliance gate (pass/fail) before CLASSic optimization (gradient) - **Cross-enclave learning constraint:** patterns propagate, content never crosses boundaries - **Sub-agent/model decoupling rationale:** compliance is a deployment property, enabling the same model in multiple sub-agents with different compliance postures - **Hybrid coordination model:** centralized control plane for policy/routing, distributed execution for scale, peer collaboration within authorized workflows - **Four forcing functions framework:** infrastructure risk, economic crossover, accuracy crossover, compliance reality as converging pressures toward multi-tier architectures

Architectural positions: - Specific language recommendations per component (Go for orchestration, Rust for routing, Python for learning) - Integration model with MCP as multi-layer connective tissue - Context persistence defaults (indefinite, owned by MCI) - Cold-start strategy (conservative first, optimize as data accumulates) - Operational auditability as requirement, not just forensic reconstruction

What we deliberately don't provide: - An SDK (use Microsoft Agent Framework, LangChain, or others) - A runtime (deploy on Kubernetes, Ray Serve, or cloud platforms) - Model recommendations (benchmarks evolve too quickly)

MCI is opinionated where existing frameworks are flexible. For environments where "it depends" isn't acceptable — where auditability, compliance, and mission-criticality constrain choices — MCI provides concrete positions to adopt, adapt, or argue against.

---

12. Call for Collaboration

This paper represents architectural thinking, not a finished implementation. The positions taken here require refinement, challenge, and production validation.

What We're Looking For

Domain expertise. The healthcare and government vignettes reflect general patterns, not deep operational knowledge. Practitioners in these domains can identify where MCI's assumptions break down.

Implementation experience. Teams building multi-model systems encounter constraints and patterns not captured here. Production feedback sharpens architectural guidance.

Research partnerships. The Adaptive Learning System's patterns-only constraint needs formal analysis. What privacy guarantees does it actually provide? How does it compare to differential privacy or federated learning for specific threat models?

Benchmark development. CLASSic provides dimensions; specific measurement methodologies for each dimension in different contexts remain underdeveloped.

How to Contribute

This paper is published at github.com/pbpuckett3/mci-pattern-wp. Issues, pull requests, and forks are welcome.

Conclusion

Four forcing functions — infrastructure risk, economic crossover, accuracy crossover, and compliance reality — are converging on the same conclusion: heterogeneous model architectures are inevitable. Organizations will operate across model tiers, from micro models at the edge to frontier models in secure clouds, whether they plan for it or not.

The specialized models exist. The integration protocols exist. The agent frameworks exist. What's been missing is a coherent framework for combining them in production environments where compliance is mandatory, auditability is required, and "it depends" isn't an acceptable architecture.

MCI provides that framework: when to decompose (SCALE), how to route (compliance gate then CLASSic), what to track (Context Manager), how to learn (patterns only across boundaries). These are opinionated positions, not universal truths. They reflect the constraints of regulated industries and mission-critical deployments.

The question is no longer whether multi-tier model architectures work. It is whether organizations will adopt them deliberately — with intelligent orchestration — or be forced into them reactively, with fragmented tooling and no coherent strategy.

MCI is a pattern for the former. Not a product. Not an SDK. A framework for building systems that work when the constraints are real.

References

Industry Analysis - Stanford HAI. "Artificial Intelligence Index Report 2025." Stanford University, April 2025. - McKinsey Global Institute. "AI Infrastructure: The Engine of the AI Era." McKinsey & Company, 2024. - "TSMC Earthquake Response and Recovery." TSMC Quarterly Investor Briefing, Q2 2024.

Model Research - Abdin et al. "Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone." Microsoft Research, April 2024. - Liu et al. "MobileLLM: Optimizing Sub-billion Parameter Language Models for On-Device Use Cases." Meta AI Research, February 2024. - Dubey et al. "The Llama 3 Herd of Models." Meta AI, July 2024. - Mishra et al. "Granite Code Models: A Family of Open Foundation Models for Code Intelligence." IBM Research, May 2024.

Agent Frameworks - Microsoft. "Microsoft Agent Framework Documentation." Microsoft Learn, 2025. - Chase, H. "LangChain Documentation." LangChain Inc., 2022-2025. - Wu, Q., et al. "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation." Microsoft Research, September 2023. - Moura, J. "CrewAI Documentation." CrewAI, 2023-2025.

Routing and Orchestration - vLLM Project. "Semantic Router / Signal-Decision Documentation." vLLM, 2024-2025. - Kwon, W., et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention." SOSP 2023. - Aisera. "CLASSic: A Framework for LLM Performance Evaluation." ICLR 2025. - Leeroo AI. "Orchestration of Experts: Learning to Route for Mixture of LLMs." 2024.

Distributed Agent Coordination - AX Platform. "The Agentic Experience." ax-platform.com, 2025. - AX Platform. "AX: MCP-Native Collaboration for AI Agents." acflow.substack.com, 2025.

Protocol Standards - Anthropic. "Model Context Protocol Specification." Anthropic, November 2024.

Architectural Precedent - Burns, B., et al. "Kubernetes: Up and Running." O'Reilly Media, Third Edition, 2022. - Newman, S. "Building Microservices." O'Reilly Media, Second Edition, 2021.

About the Author

Paul Puckett is Chief Technology Officer at Clarity Innovations and founder and CEO of Relentless Pursuits Consulting Group. He previously served as Director of the U.S. Army's Enterprise Cloud Management Agency (ECMA), where he managed the Army's $800M+ cloud portfolio and led the transformation that took the Army from the slowest to the fastest cloud adopter in the federal government globally.

His work spans defense technology, enterprise architecture, and AI systems design, with particular focus on mission-critical deployments where compliance and auditability are non-negotiable constraints.

GitHub: github.com/pbpuckett3