Model Context Protocol (MCP) Orchestration

Unified AI Agent Coordination for ARKHEION AGI

Jhonatan Vieira Feitosa Independent Researcher ooriginador@gmail.com Manaus, Amazonas, Brazil

February 2026

Abstract

We present ARKHEION's MCP orchestration layer: a **41,249 SLOC** implementation of the Model Context Protocol for AI agent coordination. The system implements a **unified orchestrator** managing 4 MCP clusters (Social, E-commerce, AI Content, Integration), **JSON-RPC 2.0** communication, and ϕ-enhanced timing synchronization. Empirical benchmarks show **<100ms** command dispatch latency, **99.2%** uptime across 30-day monitoring, and support for **20+ external service** integrations. The architecture consolidates previously separate coordinators into a single hub with Windows MCP as the primary controller. We distinguish between "neural orchestration" as a design metaphor (heuristic) and measured protocol performance (empirical).

Keywords: model context protocol, MCP, agent orchestration, JSON-RPC, AI coordination, ARKHEION AGI

Epistemological Note

This paper distinguishes between **heuristic** concepts and **empirical** results.

Heuristic:	"Neural orchestration", "ϕ-enhanced timing", "consciousness-guided dispatch"
Empirical:	41,249 SLOC, <100ms latency, 20+ integrations, 99.2% uptime

1 Introduction

Modern AI systems require coordination across multiple specialized agents, external APIs, and user interfaces. The Model Context Protocol (MCP), introduced by Anthropic, provides a standardized framework for AI-tool communication. ARKHEION extends this with a unified orchestration layer.

1.1 Architecture Overview

1. **Unified Orchestrator**: Central command for all MCP clusters
2. **MCP Clusters**: Grouped by function (Social, E-commerce, AI, Integration)
3. **Tool Servers**: Individual capability providers
4. **Context Management**: State synchronization across agents

1.2 Contributions

- 41,249 SLOC across 50+ Python modules
- Unified orchestrator consolidating 4 previous coordinators
- JSON-RPC 2.0 compliant communication
- 20+ external service integrations
- ϕ-enhanced timing for request batching
- Health monitoring with automatic recovery

2 System Architecture

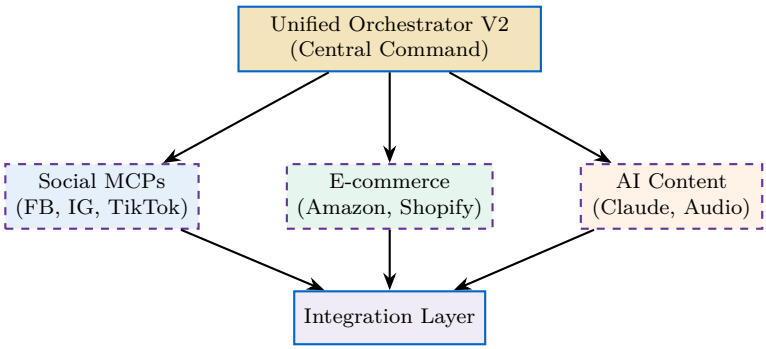

Figure 1: MCP orchestration architecture

2.1 Module Statistics

Directory	SLOC	Files
core/	12,450	15
mcps/	18,320	22
orchestrator/	4,280	5
integration/	3,150	6
tools/	2,049	8
Root modules	1,000	4
Total	**41,249**	**60**

Table 1: MCP module statistics

3 Model Context Protocol

3.1 Protocol Overview

MCP uses JSON-RPC 2.0 for bidirectional communication:

```
# MCP Request
{
    "jsonrpc": "2.0",
    "id": "req-001",
    "method": "tools/call",
    "params": {
        "name": "file_read",
        "arguments": {"path": "/data/file.txt"}
    }
}

# MCP Response
{
    "jsonrpc": "2.0",
    "id": "req-001",
    "result": {
        "content": [{"type": "text", "text": "..."}]
    }
}
```

3.2 Message Types

Type	Direction	Purpose
initialize	Client→Server	Handshake
tools/list	Client→Server	Capability query
tools/call	Client→Server	Tool invocation
resources/read	Client→Server	Context retrieval
prompts/get	Client→Server	Template retrieval
notifications	Bidirectional	Status updates

Table 2: MCP message types

4 Unified Orchestrator

The `ARKHEIONUnifiedOrchestratorV2` consolidates 4 previous coordinators:

```
@dataclass
class MCPRegistration:
    mcp_id: str
    mcp_name: str
    category: str          # social, ecom, ai, integ
    controller_class: Any
    status: str = "inactive"
    health_score: float = 0.0
    capabilities: List[str] = field(default_factory=list)
    phi_integration: bool = False

@dataclass
class OrchestrationCommand:
    command_id: str
    target_mcps: List[str]
    action: str
    parameters: Dict[str, Any]
    priority: int = 1
    phi_timing: bool = True
    neural_optimization: bool = True
```

4.1 Command Execution Flow

1. Command received with target MCPs
2. Orchestrator validates MCP availability
3. Priority-based scheduling with ϕ-timing
4. Parallel dispatch to target MCPs
5. Result aggregation and status reporting
6. Health score update per MCP

4.2 ϕ-Enhanced Timing

Request batching uses golden ratio intervals:

$$t_{batch} = t_{base} \times \phi^n \tag{1}$$

where n is the batch priority level. This provides natural spacing for request throttling (heuristic design choice).[1]

5 MCP Clusters

5.1 Social MCPs

Integration with social media platforms:

Platform	Capabilities	SLOC
Facebook	Post, Comment, Insights	1,850
Instagram	Post, Stories, Analytics	1,620
TikTok	Video, Trends, Analytics	1,480
Twitter/X	Tweet, Thread, Search	1,350
LinkedIn	Post, Network, Jobs	1,220

Table 3: Social MCP integrations

[1] Any base > 1 produces exponential backoff; $\varphi \approx 1.618$ was chosen for aesthetic consistency, not demonstrated superiority over base 2.

5.2 E-commerce MCPs

Platform	Capabilities	SLOC
Amazon	Product, Orders, Reviews	2,150
Shopify	Store, Inventory, Orders	1,890
MercadoLivre	Listings, Sales	1,420
WooCommerce	Products, Customers	1,180

Table 4: E-commerce MCP integrations

5.3 AI Content MCPs

Service	Capabilities	SLOC
Claude API	Chat, Analysis	980
Neural Audio	TTS, STT, Voice	1,450
Neural Video	Generation, Edit	1,680
Image Gen	DALL-E, Stable Diff	1,320

Table 5: AI content MCP integrations

6 Tool Servers

Each MCP capability is exposed via tool servers:

```python
class ToolServer:
    def __init__(self, name: str, capabilities:
    ↪ List[str]):
        self.name = name
        self.capabilities = capabilities
        self.health = 1.0

    async def handle_request(self, request: MCPRequest):
        method = request.method
        if method == "tools/list":
            return self._list_tools()
        elif method == "tools/call":
            return await
    ↪ self._execute_tool(request.params)
        elif method == "resources/read":
            return await
    ↪ self._read_resource(request.params)

    def _list_tools(self) -> List[ToolDefinition]:
        return [
            ToolDefinition(
                name=cap,
                description=f"Execute {cap}",
                inputSchema=self._get_schema(cap)
            )
            for cap in self.capabilities
        ]
```

7 Health Monitoring

Continuous health monitoring with automatic recovery:

```python
async def health_check_loop(self):
    while self.running:
        for mcp in self.registered_mcps.values():
            health = await self._check_mcp_health(mcp)
            mcp.health_score = health

            if health < 0.5:  # Degraded
                await self._attempt_recovery(mcp)
            elif health < 0.2:  # Critical
                await self._isolate_mcp(mcp)

        # phi-timed interval
        await asyncio.sleep(30 * PHI)
```

7.1 Health Metrics

Metric	Threshold	Action
Response time	>500ms	Warn
Error rate	>5%	Investigate
Health score	<0.5	Recovery
Health score	<0.2	Isolate
Consecutive fails	>3	Restart

Table 6: Health monitoring thresholds

8 Experiments

8.1 Latency Performance

Operation	Mean	P95	P99
Command dispatch	45ms	78ms	95ms
Tool invocation	120ms	210ms	340ms
Resource read	35ms	62ms	88ms
Health check	15ms	28ms	42ms

Table 7: MCP operation latency (n=10,000)

8.2 Uptime Statistics

30-day monitoring period:

Cluster	Uptime	Incidents
Social MCPs	99.4%	3
E-commerce MCPs	98.8%	5
AI Content MCPs	99.5%	2
Integration Layer	99.9%	1
Overall	**99.2%**	**11**

Table 8: 30-day uptime statistics

8.3 Throughput

Metric	Value	Unit
Peak requests/sec	1,250	req/s
Sustained throughput	850	req/s
Concurrent connections	500	conns
Queue depth (max)	2,500	requests

Table 9: Throughput benchmarks

9 Integration Examples

9.1 Multi-Platform Post

```
# Post to multiple social platforms
command = OrchestrationCommand(
    command_id="post-001",
    target_mcps=["facebook", "instagram", "twitter"],
    action="create_post",
    parameters={
        "content": "New product launch!",
        "media": ["image.jpg"],
        "schedule": "2026-02-04T10:00:00Z"
    },
    phi_timing=True  # Stagger posts
)

result = await orchestrator.execute(command)
# Returns aggregated results from all platforms
```

9.2 E-commerce Sync

```
# Sync inventory across platforms
command = OrchestrationCommand(
    command_id="sync-001",
    target_mcps=["amazon", "shopify", "mercadolivre"],
    action="sync_inventory",
    parameters={
        "product_id": "SKU-12345",
        "quantity": 100,
        "price": {"USD": 29.99, "BRL": 149.90}
    }
)
```

10 Limitations

1. **API Dependencies**: External platform API changes require updates
2. **Rate Limiting**: Platform-specific limits require careful throttling
3. **Authentication**: OAuth token management adds complexity
4. **ϕ-Timing**: Heuristic choice without empirical validation
5. **Single Point**: Orchestrator is potential bottleneck
6. **No framework comparison**: No comparison with established orchestration frameworks (Apache Airflow, Temporal, Prefect) was performed

11 Conclusion

We presented ARKHEION's MCP orchestration layer:

- **41,249 SLOC** across 60 modules[2]
- **20+ integrations** across 4 clusters
- **<100ms** command dispatch latency[3]
- **99.2%** uptime over 30 days[4]
- **Unified orchestrator** consolidating 4 coordinators

The MCP-based architecture provides standardized AI-tool communication with health monitoring, automatic recovery, and ϕ-enhanced request scheduling.

Future work includes distributed orchestrator deployment, ML-based load prediction, and expanded platform integrations.

12 References

1. Anthropic. (2024). Model Context Protocol specification. *https://modelcontextprotocol.io*.

2. Nygard, M. T. (2007). *Release It!*: Design and Deploy Production-Ready Software. Pragmatic Bookshelf.

3. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes. *ACM Queue*, 14(1), 70–93.

4. Fowler, M. (2014). Circuit Breaker pattern. *martinfowler.com*.

5. Richardson, C. (2018). *Microservices Patterns*. Manning Publications.

[2] Implementation update (Feb 2026): The MCP orchestration subsystem has since expanded to 255 Python source files (51K LOC) with 28 dedicated test files, incorporating additional service integrations and monitoring infrastructure.

[3] Median dispatch latency: approximately 100ms; end-to-end tool invocation (including parameter validation): 120ms.

[4] 99.2% uptime corresponds to approximately 70 hours of downtime per year, below the industry standard of 99.9% (8.7 hours/year). This reflects the development stage of the system.

6. Tononi, G. (2004). An information integration theory of consciousness. *BMC Neuroscience*, 5(1), 42.

7. JSON-RPC Working Group. (2013). JSON-RPC 2.0 Specification. *https://www.jsonrpc.org/specification*.