

Architecture

- Table of Contents

- Overview

- Service-Oriented Architecture

1. Core Services (/src/claude_mpm/services/core/)
2. Agent Services (/src/claude_mpm/services/agents/)
3. Communication Services (/src/claude_mpm/services/communication/)
4. Project Services (/src/claude_mpm/services/project/)
5. Infrastructure Services (/src/claude_mpm/services/infrastructure/)

- Core Systems

 - Project Structure

 - Dependency Injection

 - Interface-Based Design

- Three-Tier Agent System

 - Agent Tiers

 - Agent Registry

 - Agent Capabilities

- Hook System

 - Hook Types

 - Hook Registration

 - Hook Configuration

 - Performance

- Memory System

 - Architecture

 - Memory Updates

 - Memory Retrieval

 - Integration

- Communication Layer

 - Socket.IO Architecture

 - Dashboard Features

 - Event Emission

- Security Framework

 - Validation Layers

 - Filesystem Restrictions

 - Input Validation

- Performance Optimizations

 - Git Branch Caching

 - Lazy Loading

 - Non-Blocking Operations

 - Intelligent Caching

Architecture

System design and core concepts for Claude MPM.

Table of Contents

- [Overview](#)
- [Service-Oriented Architecture](#)
- [Core Systems](#)
- [Three-Tier Agent System](#)
- [Hook System](#)
- [Memory System](#)
- [Communication Layer](#)
- [Security Framework](#)
- [Performance Optimizations](#)

Overview

Claude MPM is built on a service-oriented architecture with clear separation of concerns, dependency injection, and interface-based contracts.

Core Principles: 1. **Service-Oriented:** Business logic in specialized service domains 2. **Interface-Based:** Well-defined contracts for components 3. **Dependency Injection:** Loose coupling through DI container 4. **Lazy Loading:** Deferred resource initialization 5. **Extensibility:** Hook system and plugin architecture 6. **Security First:** Input validation at all layers

Architecture Benefits: - **50-80% Performance Improvement:** Lazy loading and intelligent caching - **Enhanced Security:** Defense-in-depth with validation - **Better Testability:** Interface-based design enables easy mocking - **Improved Maintainability:** Clear separation of concerns - **Scalability:** Supports future growth and plugins

Service-Oriented Architecture

The service layer is organized into five domains:

1. Core Services (/src/claude_mpm/services/core/)

Purpose: Foundation services and interfaces

Key Components: - `interfaces.py`: Service contract definitions - `base.py`: Base service classes

Key Interfaces:

<code>IServiceContainer</code>	<code># Dependency injection</code>
<code>IRegistry</code>	<code># Agent discovery</code>
<code>IHealthMonitor</code>	<code># Service health</code>
<code>IConfigurationManager</code>	<code># Configuration</code>

2. Agent Services (/src/claude_mpm/services/agents/)

Purpose: Agent lifecycle and management

Key Components: - `deployment.py`: Agent deployment and lifecycle - `management.py`: Agent registry and services - `registry.py`: Agent discovery and loading

Capabilities: - Three-tier agent precedence (PROJECT > USER > SYSTEM) - Dynamic capabilities and schema validation - Agent versioning and compatibility - Hot-reloading and updates

3. Communication Services (`/src/claude_mpm/services/communication/`)

Purpose: Real-time communication and events

Key Components: - `socketio.py`: SocketIO server management - `websocket.py`: WebSocket connections

Features: - Real-time agent activity monitoring - File operation tracking - Session state synchronization - Multi-client connection pooling

4. Project Services (`/src/claude_mpm/services/project/`)

Purpose: Project analysis and workspace management

Key Components: - `analyzer.py`: Stack and structure analysis - `registry.py`: Project configuration

Capabilities: - Automatic technology stack detection - Architecture pattern recognition - Documentation discovery - Project-specific memory

5. Infrastructure Services (`/src/claude_mpm/services/infrastructure/`)

Purpose: Cross-cutting concerns

Key Components: - `logging.py`: Structured logging - `monitoring.py`: Health and metrics

Features: - Structured JSON logging - Performance metrics collection - Error handling and recovery - Circuit breaker patterns

Core Systems

Project Structure

```
claude-mpm/
├── .claude-mpm/      # Project-specific MPM directory
│   ├── agents/       # PROJECT tier (highest precedence)
│   ├── config/       # Project configuration
│   ├── hooks/        # Project hooks
│   └── memories/     # Agent memories
```

└─ src/claude_mpm/	# Main package
└─ core/	# Core framework
└─ services/	# Service layer (5 domains)
└─ agents/	# USER tier agents
└─ hooks/	# Hook system
└─ cli/	# Command-line interface
└─ utils/	# Utilities
└─ docs/	# Documentation
└─ tests/	# Test suite
└─ scripts/	# Executable scripts

Key Guidelines: - Scripts: ALL in /scripts/, never in root - Tests: ALL in /tests/, never in root - Python modules: Always under /src/claude_mpm/ - Agent precedence: PROJECT > USER > SYSTEM

Dependency Injection

Service Container:

```
from claude_mpm.services.core.interfaces import IServiceContainer

# Register services
container.register(IAgentRegistry, AgentRegistryService,
                  singleton=True)
container.register(IHealthMonitor, HealthMonitorService,
                  singleton=True)

# Resolve dependencies
agent_registry = container.resolve(IAgentRegistry)
```

Service Lifecycle: 1. Registration: Services register interfaces 2. Resolution: Container resolves dependencies 3. Initialization: Services initialize with dependencies 4. Lifecycle: Container manages singletons

Best Practices: - Define clear interfaces - Use constructor injection - Prefer singleton for stateless services - Implement proper cleanup

Interface-Based Design

All major components implement explicit interfaces:

```
from abc import ABC, abstractmethod

class IAgentManager(ABC):
    @abstractmethod
    async def deploy_agent(self, agent_config: dict) -> bool:
        """Deploy an agent with configuration."""
        pass

    @abstractmethod
    async def list_agents(self) -> List[AgentInfo]:
```

```
"""List all available agents."""  
pass
```

Benefits: - Clear contracts between components - Easy mocking for tests - Better documentation - Interface segregation

Three-Tier Agent System

Agent precedence: **PROJECT > USER > SYSTEM**

Agent Tiers

PROJECT Tier (`.claude-mpm/agents/`): - Highest priority - Project-specific customizations - Overrides USER and SYSTEM

USER Tier (`~/claude-agents/` or `src/claude_mpm/agents/`): - Personal customizations - Shared across projects - Overrides SYSTEM

SYSTEM Tier (bundled): - Built-in default agents - Lowest priority - PM, Research, Engineer, QA, etc.

Agent Registry

Discovery Process: 1. Scan PROJECT tier (`.claude-mpm/agents/`) 2. Scan USER tier (`~/claude-agents/`) 3. Load SYSTEM tier (built-in) 4. Apply precedence rules 5. Register with metadata

Metadata Includes: - Agent name, version, tier - Capabilities and specializations - Model configuration - Instruction content

Agent Capabilities

Agents define capabilities in frontmatter:

```
----  
name: engineer  
model: claude-sonnet-4  
capabilities:  
  - code-implementation  
  - refactoring  
  - debugging  
specialization: engineering  
delegation: true  
----
```

Routing: PM agent routes tasks based on capabilities.

Hook System

Event-driven hooks for pre/post execution customization.

Hook Types

Pre-execution hooks: - Execute before agent invocation - Modify context or abort execution - Use cases: validation, logging, setup

Post-execution hooks: - Execute after agent completion - Process results or trigger actions - Use cases: cleanup, notifications, metrics

Hook Registration

```
from claudempm.hooks import HookRegistry

# Register pre-execution hook
@HookRegistry.register("pre_execution")
async def validate_input(context: HookContext) -> HookResult:
    if not context.is_valid():
        return HookResult(abort=True, reason="Invalid input")
    return HookResult(success=True)

# Register post-execution hook
@HookRegistry.register("post_execution")
async def log_completion(context: HookContext) -> HookResult:
    logger.info(f"Task completed: {context.task_id}")
    return HookResult(success=True)
```

Hook Configuration

In `.claude-mpm/config.yaml`:

```
hooks:
  pre_execution:
    enabled: true
    hooks:
      - validate_input
      - check_permissions
  post_execution:
    enabled: true
    hooks:
      - log_completion
      - update_metrics
```

Performance

v4.8.2+ Improvements: - 91% latency reduction (108ms → 10ms) - Non-blocking execution - Thread pool for HTTP calls - Intelligent caching

Memory System

Persistent project-specific knowledge using graph storage.

Architecture

Storage: KuzuDB graph database (.claude-mpm/memory.db)

Memory Categories: - Project Architecture - Implementation Guidelines - Current Technical Context

Memory Updates

Agents store learnings via JSON response fields:

```
{
  "memory-update": {
    "Project Architecture": ["Uses FastAPI with async endpoints"],
    "Implementation Guidelines": ["Use Pydantic for validation"],
    "Current Technical Context": ["Auth uses JWT tokens"]
  }
}
```

Simplified format:

```
{
  "remember": [
    "API uses REST conventions",
    "Tests use pytest fixtures"
  ]
}
```

Memory Retrieval

Automatic: Prompts enhanced with relevant memories

Manual:

```
# Query memories
claude-mpm recall "authentication"
```

```
# View statistics
claude-mpm stats
```

Integration

MCP Tool: kuzu-memory provides memory operations - kuzu_enhance: Enhance prompts with context - kuzu_learn: Store new learnings - kuzu_recall: Query memories - kuzu_stats: View statistics

Communication Layer

Real-time communication via WebSocket and Socket.IO.

Socket.IO Architecture

Server: Flask-SocketIO server on port 8765

Events: - connect: Client connection - disconnect: Client disconnection - agent_started: Agent begins task - agent_completed: Agent finishes task - task_delegated: PM delegates to specialist - memory_updated: New learning stored - error_occurred: Error during execution

Dashboard Features

Real-time updates: - Active agents and tasks - Delegation flow visualization - System metrics (memory, latency) - Session information

Connection Management: - Multi-client support - Connection pooling - Automatic reconnection - Session correlation

Event Emission

```
from claudempm.services.communication import emit_event

# Emit agent started event
emit_event("agent_started", {
    "agent_id": "engineer",
    "task": "Implement authentication",
    "timestamp": datetime.now().isoformat()
})

# Emit task delegated event
emit_event("task_delegated", {
    "from_agent": "pm",
    "to_agent": "engineer",
    "task": "Implement feature",
    "reason": "Engineering expertise required"
})
```

Security Framework

Comprehensive input validation and sanitization.

Validation Layers

Layer 1: Input Validation - Type checking - Format validation - Range validation - Pattern matching

Layer 2: Sanitization - Path sanitization - Command sanitization - String escaping - SQL injection prevention

Layer 3: Authorization - Permission checks - Resource access control - Rate limiting

Filesystem Restrictions

Allowed Operations: - Read: Within project directory - Write: To `.claude-mpm/` and project files - Execute: Whitelisted commands only

Blocked Operations: - System file access - Parent directory traversal (`../`) - Absolute paths outside project - Dangerous commands (`rm -rf`, etc.)

Input Validation

```
from claude_mpm.security import validate_input, sanitize_path

# Validate file path
if not validate_input(file_path, input_type="path"):
    raise ValidationError("Invalid file path")

# Sanitize path
safe_path = sanitize_path(user_provided_path)
```

Performance Optimizations

v4.8.2+ Improvements:

Git Branch Caching

Before: Every git operation queries branch (108ms) **After:** 5-minute TTL cache (10ms)

Implementation:

```
@cache_with_ttl(ttl=300) # 5 minutes
def get_current_branch() -> str:
    return subprocess.check_output(
        ["git", "rev-parse", "--abbrev-ref", "HEAD"]
    ).decode().strip()
```

Lazy Loading

Strategy: Defer initialization until needed

Examples: - Agent loading: On-demand rather than startup - Service initialization: Lazy service resolution - Configuration: Load sections as accessed

Benefits: - Faster startup (50-80% improvement) - Reduced memory footprint - Better resource utilization

Non-Blocking Operations

HTTP Fallback: Thread pool for HTTP health checks

Before: Blocking HTTP calls **After:** Non-blocking with timeout

```
from concurrent.futures import ThreadPoolExecutor
```

```
executor = ThreadPoolExecutor(max_workers=4)
future = executor.submit(requests.get, url, timeout=5)
```

Intelligent Caching

Cache Strategy: - LRU cache for frequently accessed data - TTL cache for time-sensitive data - Size-limited cache to prevent memory issues

Cached Operations: - Git branch (5-minute TTL) - Agent metadata (until file change) - Configuration (until modification) - Project analysis (until structure change)

Next Steps: - Extending: See [extending.md](#) - API Reference: See [api-reference.md](#) - User Docs: See [../user/user-guide.md](#)