MASTER IMPLEMENTATION PLAN

EX-AI MCP Server - Complete System Restoration & Enhancement

Date: October 4, 2025
Branch: feat/auggie-mcp-optimization
Repository: /home/ubuntu/github_repos/EX-AI-MCP-Server
Status: 🔴 CRITICAL - System Partially Functional
Target: 🟢 PRODUCTION READY - Full Functionality Restored

SECTION 1: EXECUTIVE SUMMARY

What Went Wrong

The `feat/auggie-mcp-optimization` branch introduced critical architectural issues while attempting to optimize the system for autonomous operation:

Root Causes:
1. **Timeout Hierarchy Inversion:** Extended outer timeouts (600s daemon, 900s Kimi) prevent inner timeouts (25s tool, 90s expert) from ever triggering, creating a perception of system hanging
2. **Missing Progress Heartbeat:** Long-running operations provide no feedback, making users think the system is frozen
3. **Logging Path Divergence:** Workflow tools use different execution paths that bypass the logging infrastructure
4. **Configuration Chaos:** Multiple conflicting timeout values scattered across 15+ files with no co-ordination
5. **Expert Validation Bug:** Duplicate call issue caused 300+ second hangs, leading to temporary disabling of key feature

What Broke:
- ✅ Simple tools (chat, listmodels) → **WORKING**
- 🔴 Workflow tools (analyze, thinkdeep, debug, codereview) → **BROKEN** (hang without timeout)
- 🟡 Logging system → **PARTIAL** (works for simple tools only)
- 🔴 Expert validation → **DISABLED** (duplicate call bug)
- 🟡 Web search integration → **UNCLEAR** (no logging to verify)

What Improved:
- ✅ Code organization (bootstrap modules, mixin pattern)
- ✅ Critical bug fixes (server crash, schema validation, WebSocket shim)
- ✅ Documentation (77 new files, 23,906 lines)
- ✅ Test infrastructure (6/6 tests passing)

Why the System Broke

Architectural Missteps:

1. **Timeout Configuration Without Coordination**
   - Auggie config extended timeouts to support 30-60 minute sessions
   - But didn't implement progress heartbeat or graceful degradation
   - Result: Users wait 10 minutes for timeout instead of 25 seconds

2. **Refactoring Without Integration Testing**
   - Workflow tools refactored with different execution paths
   - Logging integration not tested after refactoring
   - Result: Silent failures, no visibility into execution

3. **Feature Disabling Without Root Cause Fix**
   - Expert validation disabled due to duplicate call bug
   - Bug not investigated or fixed
   - Result: Missing key differentiator of the system

4. **Configuration Proliferation**
   - Three different MCP configs (Auggie, Augment, Claude)
   - Each with different timeout values
   - No documentation of why differences exist
   - Result: Unpredictable behavior across clients

Fix Strategy

Three-Pronged Approach:

1. **Immediate Fixes (Week 1 - P0):**
   - Fix timeout hierarchy with coordinated values
   - Implement progress heartbeat (5-8 second intervals)
   - Unify logging infrastructure for all tools
   - Add graceful timeout handling with early termination

2. **High Priority Fixes (Week 2 - P1):**
   - Debug and fix expert validation duplicate call bug
   - Standardize timeout configurations across all clients
   - Implement graceful degradation for failures
   - Fix silent failure issues with proper error propagation

3. **Enhancement & Stabilization (Week 3 - P2):**
   - Integrate GLM and Kimi native web search following official APIs
   - Simplify continuation_id system
   - Update documentation to reflect actual state
   - Optimize WebSocket daemon stability

Implementation Philosophy:
- **Safety First:** Never break working functionality
- **Incremental:** One fix at a time with validation
- **Testable:** Every fix must have acceptance criteria
- **Documented:** Update docs as we go, not at the end

Expected Outcomes

After Week 1 (P0 Fixes):
- ✅ Workflow tools complete in reasonable time (60-120s)
- ✅ Users see progress updates every 5-8 seconds
- ✅ All tool executions logged correctly
- ✅ Timeouts trigger at expected intervals
- ✅ System feels responsive and reliable

After Week 2 (P1 Fixes):
- ✅ Expert validation re-enabled and working
- ✅ Consistent behavior across all three clients
- ✅ Graceful degradation when services fail
- ✅ Clear error messages for all failure modes
- ✅ Configuration is understandable and maintainable

After Week 3 (P2 Enhancements):
- ✅ Native web search working for GLM and Kimi
- ✅ Simplified continuation system
- ✅ Accurate documentation
- ✅ Stable WebSocket daemon
- ✅ Production-ready system

Final State:
- 🎯 All tools working correctly
- 🎯 Predictable performance (60-120s for workflow tools)
- 🎯 Comprehensive logging and monitoring
- 🎯 Graceful error handling
- 🎯 Clear documentation
- 🎯 VSCode Augment extension fully functional
- 🎯 Original vision from master task list achieved

SECTION 2: ARCHITECTURE CORRECTIONS

2.1 Timeout Hierarchy Redesign

Current Problem:

```
Tool Level:        25s (thinkdeep) / 90s (expert)
Daemon Level:      600s (EXAI_WS_CALL_TIMEOUT)
Shim Level:        600s (EXAI_SHIM_RPC_TIMEOUT)
Provider Level:    900s (KIMI_CHAT_TOOL_TIMEOUT_WEB_SECS)

Problem: Inner timeouts never trigger because outer timeouts are 10-24x longer!
```

Corrected Hierarchy:

```
Level 1 (Tool):           60s (simple) / 120s (workflow)
Level 2 (Expert):         90s (80% of tool timeout)
Level 3 (Daemon):         180s (1.5x tool timeout)
Level 4 (Shim):           240s (2x tool timeout)
Level 5 (Client):         300s (2.5x tool timeout)

Rule: Each outer timeout = 1.5x inner timeout (50% buffer)
```

Coordination Strategy:

1. **Tool-Level Timeout (Primary)**
   - Tools set their own timeout based on complexity
   - Simple tools: 60s
   - Workflow tools: 120s
   - Expert validation: 90s (within workflow timeout)

2. **Daemon-Level Timeout (Secondary)**
   - Daemon timeout = 1.5x max tool timeout
   - Catches tools that don't implement timeout properly
   - Provides graceful degradation

3. **Shim-Level Timeout (Tertiary)**
   - Shim timeout = 2x max tool timeout
   - Catches daemon failures
   - Last line of defense before client timeout

4. **Client-Level Timeout (Final)**
   - Client timeout = 2.5x max tool timeout
   - Prevents infinite hangs
   - User-facing timeout

Implementation Files:
- `config.py` - Central timeout configuration
- `src/daemon/ws_server.py` - Daemon timeout enforcement
- `scripts/run_ws_shim.py` - Shim timeout enforcement
- `Daemon/mcp-config.*.json` - Client-specific overrides
- `tools/workflow/base.py` - Tool timeout implementation
- `tools/workflows/thinkdeep.py` - Tool-specific timeouts
- `tools/workflow/expert_analysis.py` - Expert timeout

Configuration Schema:

```python
# config.py
class TimeoutConfig:
    # Tool timeouts
    SIMPLE_TOOL_TIMEOUT_SECS = 60
    WORKFLOW_TOOL_TIMEOUT_SECS = 120
    EXPERT_ANALYSIS_TIMEOUT_SECS = 90

    # Infrastructure timeouts (auto-calculated)
    DAEMON_TIMEOUT_SECS = WORKFLOW_TOOL_TIMEOUT_SECS * 1.5  # 180s
    SHIM_TIMEOUT_SECS = WORKFLOW_TOOL_TIMEOUT_SECS * 2.0    # 240s
    CLIENT_TIMEOUT_SECS = WORKFLOW_TOOL_TIMEOUT_SECS * 2.5  # 300s

    # Provider timeouts
    GLM_TIMEOUT_SECS = 90
    KIMI_TIMEOUT_SECS = 120
    KIMI_WEB_SEARCH_TIMEOUT_SECS = 150
```

2.2 Progress Heartbeat System Design

Purpose: Provide continuous feedback during long-running operations to prevent perception of hanging.

Requirements:
1. Heartbeat every 5-8 seconds during execution
2. Include progress information (step X of Y, current operation)
3. Non-blocking (doesn't slow down execution)
4. Graceful degradation if heartbeat fails

Architecture:

```python
class ProgressHeartbeat:
    """Manages progress heartbeat for long-running operations."""

    def __init__(self, interval_secs: float = 6.0):
        self.interval = interval_secs
        self.last_heartbeat = time.time()
        self.enabled = True

    async def send_heartbeat(self, message: str, metadata: dict = None):
        """Send progress heartbeat if interval elapsed."""
        now = time.time()
        if now - self.last_heartbeat >= self.interval:
            await self._emit_progress(message, metadata)
            self.last_heartbeat = now

    async def _emit_progress(self, message: str, metadata: dict):
        """Emit progress message to client."""
        # Send via WebSocket or logging
        pass
```

Integration Points:

1. **Workflow Tools** (`tools/workflow/base.py`)
   - Start heartbeat at beginning of execution
   - Update heartbeat at each step
   - Stop heartbeat at completion

2. **Expert Analysis** (`tools/workflow/expert_analysis.py`)
   - Heartbeat during expert validation
   - Show which expert is being consulted
   - Show progress through validation steps

3. **Provider Calls** (`src/providers/openai_compatible.py`)
   - Heartbeat during long API calls
   - Show streaming progress
   - Show retry attempts

Message Format:

```
{
  "type": "progress",
  "timestamp": 1696435200.123,
  "tool": "thinkdeep",
  "step": 2,
  "total_steps": 5,
  "message": "Analyzing code structure...",
  "elapsed_secs": 15.3,
  "estimated_remaining_secs": 45.0
}
```

Implementation Files:
- `utils/progress.py` - Progress heartbeat implementation (NEW)
- `tools/workflow/base.py` - Integration in workflow tools
- `tools/workflow/expert_analysis.py` - Integration in expert validation
- `src/providers/openai_compatible.py` - Integration in provider calls
- `src/daemon/ws_server.py` - WebSocket progress message routing

2.3 Logging Infrastructure Unification

Current Problem:
- Simple tools log correctly via `tools/simple/base.py`
- Workflow tools use different path via `tools/workflow/base.py`
- Workflow tool logs not appearing in `.logs/toolcalls.jsonl`
- No structured logging with request_id tracking

Unified Architecture:

```python
# utils/logging_unified.py (NEW)
class UnifiedLogger:
    """Unified logging for all tools."""

    def __init__(self, log_file: str = ".logs/toolcalls.jsonl"):
        self.log_file = log_file
        self.buffer = []

    def log_tool_start(self, tool_name: str, request_id: str, params: dict):
        """Log tool execution start."""
        entry = {
            "timestamp": time.time(),
            "event": "tool_start",
            "tool": tool_name,
            "request_id": request_id,
            "params": params
        }
        self._write_log(entry)

    def log_tool_progress(self, tool_name: str, request_id: str,
                          step: int, message: str):
        """Log tool execution progress."""
        entry = {
            "timestamp": time.time(),
            "event": "tool_progress",
            "tool": tool_name,
            "request_id": request_id,
            "step": step,
            "message": message
        }
        self._write_log(entry)

    def log_tool_complete(self, tool_name: str, request_id: str,
                          duration_s: float, result_preview: str):
        """Log tool execution completion."""
        entry = {
            "timestamp": time.time(),
            "event": "tool_complete",
            "tool": tool_name,
            "request_id": request_id,
            "duration_s": duration_s,
            "result_preview": result_preview[:200]
        }
        self._write_log(entry)

    def log_tool_error(self, tool_name: str, request_id: str,
                       error: str, traceback: str):
        """Log tool execution error."""
        entry = {
            "timestamp": time.time(),
            "event": "tool_error",
            "tool": tool_name,
            "request_id": request_id,
            "error": error,
            "traceback": traceback
        }
        self._write_log(entry)

    def _write_log(self, entry: dict):
        """Write log entry to file."""
        with open(self.log_file, "a") as f:
            f.write(json.dumps(entry) + "\n")
```

Integration Strategy:

1. **Create Unified Logger** (`utils/logging_unified.py`)
   - Single logging interface for all tools
   - Structured logging with request_id
   - Buffered writes for performance
   - Automatic log rotation

2. **Update Simple Tools** (`tools/simple/base.py`)
   - Replace existing logging with unified logger
   - Maintain backward compatibility
   - Add request_id tracking

3. **Update Workflow Tools** (`tools/workflow/base.py`)
   - Add unified logger integration
   - Log all execution steps
   - Log expert validation
   - Log progress updates

4. **Update Expert Analysis** (`tools/workflow/expert_analysis.py`)
   - Log expert validation start/complete
   - Log each expert consulted
   - Log validation results

Implementation Files:
- `utils/logging_unified.py` - Unified logger implementation (NEW)
- `tools/simple/base.py` - Integration in simple tools
- `tools/workflow/base.py` - Integration in workflow tools
- `tools/workflow/expert_analysis.py` - Integration in expert validation
- `src/daemon/ws_server.py` - Request ID generation and tracking

2.4 Error Handling and Graceful Degradation Strategy

Current Problem:
- Silent failures (errors not propagated)
- No graceful degradation (all-or-nothing)
- No circuit breaker (repeated failures not handled)
- No fallback strategies

Graceful Degradation Architecture:

```python
# utils/error_handling.py (NEW)
class GracefulDegradation:
    """Handles graceful degradation for failures."""

    def __init__(self):
        self.circuit_breakers = {}

    async def execute_with_fallback(
        self,
        primary_fn: Callable,
        fallback_fn: Callable = None,
        timeout_secs: float = 60.0,
        max_retries: int = 2
    ) -> Any:
        """Execute function with fallback and timeout."""

        # Check circuit breaker
        if self._is_circuit_open(primary_fn.__name__):
            if fallback_fn:
                return await fallback_fn()
            raise CircuitBreakerOpen(f"{primary_fn.__name__} circuit open")

        # Try primary function with retries
        for attempt in range(max_retries + 1):
            try:
                result = await asyncio.wait_for(
                    primary_fn(),
                    timeout=timeout_secs
                )
                self._record_success(primary_fn.__name__)
                return result

            except asyncio.TimeoutError:
                if attempt < max_retries:
                    await asyncio.sleep(2 ** attempt)  # Exponential backoff
                    continue

                # Timeout - try fallback
                self._record_failure(primary_fn.__name__)
                if fallback_fn:
                    return await fallback_fn()
                raise

            except Exception as e:
                self._record_failure(primary_fn.__name__)
                if attempt < max_retries:
                    await asyncio.sleep(2 ** attempt)
                    continue

                # Error - try fallback
                if fallback_fn:
                    return await fallback_fn()
                raise

    def _is_circuit_open(self, fn_name: str) -> bool:
        """Check if circuit breaker is open."""
        if fn_name not in self.circuit_breakers:
            return False
        cb = self.circuit_breakers[fn_name]
        return cb["failures"] >= 5 and time.time() - cb["last_failure"] < 300

    def _record_success(self, fn_name: str):
```

```
        """Record successful execution."""
        if fn_name in self.circuit_breakers:
            self.circuit_breakers[fn_name]["failures"] = 0

    def _record_failure(self, fn_name: str):
        """Record failed execution."""
        if fn_name not in self.circuit_breakers:
            self.circuit_breakers[fn_name] = {"failures": 0, "last_failure": 0}
        self.circuit_breakers[fn_name]["failures"] += 1
        self.circuit_breakers[fn_name]["last_failure"] = time.time()
```

Fallback Strategies:

1. **Expert Validation Fallback**
   - Primary: Full expert validation
   - Fallback: Skip expert validation, return without validation
   - Degraded: Return partial results with warning

2. **Web Search Fallback**
   - Primary: GLM native web search
   - Fallback: Kimi web search
   - Degraded: Skip web search, use cached knowledge

3. **Provider Fallback**
   - Primary: GLM-4.6
   - Fallback: GLM-4.5-flash
   - Degraded: Return error with retry suggestion

Implementation Files:
- `utils/error_handling.py` - Graceful degradation implementation (NEW)
- `tools/workflow/base.py` - Integration in workflow tools
- `tools/workflow/expert_analysis.py` - Expert validation fallback
- `src/providers/openai_compatible.py` - Provider fallback
- `tools/simple/base.py` - Web search fallback

2.5 Expert Validation Fix Approach

Current Problem:
- Expert validation calls analysis multiple times (duplicate calls)
- Causes 300+ second hangs
- Temporarily disabled (DEFAULT_USE_ASSISTANT_MODEL=false)
- Bug not investigated or fixed

Root Cause Analysis Strategy:

1. **Trace Execution Path**
   - Add detailed logging to expert_analysis.py
   - Track each call to expert validation
   - Identify where duplicate calls originate

2. **Check for Recursion**
   - Look for recursive calls in expert_analysis.py
   - Check if expert validation calls itself
   - Check if workflow tools call expert validation multiple times

3. **Check for Event Loops**
   - Look for event-driven triggers
   - Check if progress updates trigger re-validation
   - Check if continuation system triggers re-validation

Fix Strategy:

```python
# tools/workflow/expert_analysis.py
class ExpertAnalysis:
    """Expert validation with duplicate call prevention."""

    def __init__(self):
        self._validation_cache = {}  # Cache by request_id
        self._in_progress = set()    # Track in-progress validations

    async def validate_with_expert(
        self,
        request_id: str,
        content: str,
        context: dict
    ) -> dict:
        """Validate content with expert, preventing duplicates."""

        # Check cache
        cache_key = f"{request_id}:{hash(content)}"
        if cache_key in self._validation_cache:
            logger.info(f"Using cached expert validation for {request_id}")
            return self._validation_cache[cache_key]

        # Check if already in progress
        if cache_key in self._in_progress:
            logger.warning(f"Expert validation already in progress for {request_id}")
            # Wait for in-progress validation to complete
            while cache_key in self._in_progress:
                await asyncio.sleep(0.5)
            return self._validation_cache.get(cache_key)

        # Mark as in progress
        self._in_progress.add(cache_key)

        try:
            # Perform validation
            result = await self._perform_validation(content, context)

            # Cache result
            self._validation_cache[cache_key] = result

            return result

        finally:
            # Remove from in-progress
            self._in_progress.discard(cache_key)
```

Implementation Files:

- `tools/workflow/expert_analysis.py` - Duplicate call prevention
- `tools/workflow/base.py` - Expert validation integration
- `tools/workflow/conversation_integration.py` - Remove stub method (already done)
- `.env` - Re-enable expert validation (DEFAULT_USE_ASSISTANT_MODEL=true)

Testing Strategy:
1. Add detailed logging to track all expert validation calls
2. Test with debug tool (2-step workflow)
3. Verify expert validation called exactly once per step
4. Verify duration is 90-120 seconds (not 300+)
5. Verify expert_analysis contains real content (not null)

SECTION 3: CRITICAL FIXES (Week 1 - P0 Issues)

Fix #1: Timeout Hierarchy Coordination

Priority: P0 - CRITICAL
Impact: Workflow tools hang for 10 minutes instead of timing out at 25-90 seconds
Estimated Time: 2 days
Dependencies: None

Implementation Steps

Step 1: Create Central Timeout Configuration

File: `config.py`
Location: Lines 1-50 (add new TimeoutConfig class)

Code to Add:

```python
# config.py (add at top of file)
import os
from typing import Optional

class TimeoutConfig:
    """Centralized timeout configuration with coordinated hierarchy."""

    # Tool-level timeouts (primary)
    SIMPLE_TOOL_TIMEOUT_SECS = int(os.getenv("SIMPLE_TOOL_TIMEOUT_SECS", "60"))
    WORKFLOW_TOOL_TIMEOUT_SECS = int(os.getenv("WORKFLOW_TOOL_TIMEOUT_SECS", "120"))
    EXPERT_ANALYSIS_TIMEOUT_SECS = int(os.getenv("EXPERT_ANALYSIS_TIMEOUT_SECS", "90")
)

    # Infrastructure timeouts (auto-calculated with 50% buffer)
    @classmethod
    def get_daemon_timeout(cls) -> int:
        """Daemon timeout = 1.5x max tool timeout."""
        return int(cls.WORKFLOW_TOOL_TIMEOUT_SECS * 1.5)  # 180s

    @classmethod
    def get_shim_timeout(cls) -> int:
        """Shim timeout = 2x max tool timeout."""
        return int(cls.WORKFLOW_TOOL_TIMEOUT_SECS * 2.0)  # 240s

    @classmethod
    def get_client_timeout(cls) -> int:
        """Client timeout = 2.5x max tool timeout."""
        return int(cls.WORKFLOW_TOOL_TIMEOUT_SECS * 2.5)  # 300s

    # Provider timeouts
    GLM_TIMEOUT_SECS = int(os.getenv("GLM_TIMEOUT_SECS", "90"))
    KIMI_TIMEOUT_SECS = int(os.getenv("KIMI_TIMEOUT_SECS", "120"))
    KIMI_WEB_SEARCH_TIMEOUT_SECS = int(os.getenv("KIMI_WEB_SEARCH_TIMEOUT_SECS",
"150"))

    @classmethod
    def validate_hierarchy(cls) -> bool:
        """Validate timeout hierarchy is correct."""
        daemon = cls.get_daemon_timeout()
        shim = cls.get_shim_timeout()
        client = cls.get_client_timeout()

        # Check hierarchy: tool < daemon < shim < client
        if not (cls.WORKFLOW_TOOL_TIMEOUT_SECS < daemon < shim < client):
            raise ValueError(
                f"Invalid timeout hierarchy: "
                f"tool={cls.WORKFLOW_TOOL_TIMEOUT_SECS}, "
                f"daemon={daemon}, shim={shim}, client={client}"
            )
        return True

# Validate on import
TimeoutConfig.validate_hierarchy()
```

Step 2: Update Daemon Timeout

File: `src/daemon/ws_server.py`
Location: Line 89 (CALL_TIMEOUT definition)

Before:

```
CALL_TIMEOUT = int(os.getenv("EXAI_WS_CALL_TIMEOUT", "90"))
```

After:

```
from config import TimeoutConfig

CALL_TIMEOUT = TimeoutConfig.get_daemon_timeout()  # 180s (auto-calculated)
```

Step 3: Update Shim Timeout

File: `scripts/run_ws_shim.py`
Location: Line ~50 (RPC_TIMEOUT definition)

Before:

```
RPC_TIMEOUT = int(os.getenv("EXAI_SHIM_RPC_TIMEOUT", "600"))
```

After:

```
from config import TimeoutConfig

RPC_TIMEOUT = TimeoutConfig.get_shim_timeout()  # 240s (auto-calculated)
```

Step 4: Update Workflow Tool Timeout

File: `tools/workflow/base.py`
Location: Add timeout parameter to execute method

Code to Add:

```
from config import TimeoutConfig

class WorkflowTool:
    def __init__(self):
        self.timeout_secs = TimeoutConfig.WORKFLOW_TOOL_TIMEOUT_SECS

    async def execute(self, request: dict) -> dict:
        """Execute workflow with timeout."""
        try:
            result = await asyncio.wait_for(
                self._execute_workflow(request),
                timeout=self.timeout_secs
            )
            return result
        except asyncio.TimeoutError:
            return {
                "error": f"Workflow timed out after {self.timeout_secs}s",
                "partial_results": self._get_partial_results()
            }
```

Step 5: Update Expert Analysis Timeout

File: `tools/workflow/expert_analysis.py`
Location: Lines 115-125 (get_expert_timeout_secs method)

Before:

```python
def get_expert_timeout_secs(self, request=None) -> float:
    """Cap thinkdeep expert analysis to a shorter window."""
    import os
    try:
        return float(os.getenv("THINKDEEP_EXPERT_TIMEOUT_SECS", "25"))
    except Exception:
        return 25.0
```

After:

```python
from config import TimeoutConfig

def get_expert_timeout_secs(self, request=None) -> float:
    """Get expert analysis timeout from central config."""
    return float(TimeoutConfig.EXPERT_ANALYSIS_TIMEOUT_SECS)  # 90s
```

Step 6: Update MCP Configurations

File: `Daemon/mcp-config.auggie.json`
Location: Environment variables section

Before:

```json
{
  "EXAI_SHIM_RPC_TIMEOUT": "600",
  "EXAI_WS_CALL_TIMEOUT": "600",
  "KIMI_CHAT_TOOL_TIMEOUT_WEB_SECS": "900"
}
```

After:

```json
{
  "SIMPLE_TOOL_TIMEOUT_SECS": "60",
  "WORKFLOW_TOOL_TIMEOUT_SECS": "120",
  "EXPERT_ANALYSIS_TIMEOUT_SECS": "90",
  "GLM_TIMEOUT_SECS": "90",
  "KIMI_TIMEOUT_SECS": "120",
  "KIMI_WEB_SEARCH_TIMEOUT_SECS": "150"
}
```

Note: Daemon, shim, and client timeouts are auto-calculated from tool timeouts.

Step 7: Update Other MCP Configurations

Files:
- `Daemon/mcp-config.augmentcode.json`
- `Daemon/mcp-config.claude.json`

Apply same changes as Step 6.

Step 8: Update .env.example

File: `.env.example`
Location: Add timeout configuration section

Code to Add:

```
# Timeout Configuration (coordinated hierarchy)
# Tool timeouts (primary)
SIMPLE_TOOL_TIMEOUT_SECS=60
WORKFLOW_TOOL_TIMEOUT_SECS=120
EXPERT_ANALYSIS_TIMEOUT_SECS=90

# Provider timeouts
GLM_TIMEOUT_SECS=90
KIMI_TIMEOUT_SECS=120
KIMI_WEB_SEARCH_TIMEOUT_SECS=150

# Infrastructure timeouts (auto-calculated, do not set manually)
# DAEMON_TIMEOUT = WORKFLOW_TOOL_TIMEOUT * 1.5 = 180s
# SHIM_TIMEOUT = WORKFLOW_TOOL_TIMEOUT * 2.0 = 240s
# CLIENT_TIMEOUT = WORKFLOW_TOOL_TIMEOUT * 2.5 = 300s
```

Testing Instructions

Test 1: Verify Timeout Hierarchy

```
# Start Python interpreter
python3
>>> from config import TimeoutConfig
>>> TimeoutConfig.validate_hierarchy()
True
>>> print(f"Tool: {TimeoutConfig.WORKFLOW_TOOL_TIMEOUT_SECS}s")
>>> print(f"Daemon: {TimeoutConfig.get_daemon_timeout()}s")
>>> print(f"Shim: {TimeoutConfig.get_shim_timeout()}s")
>>> print(f"Client: {TimeoutConfig.get_client_timeout()}s")
```

Expected Output:

```
Tool: 120s
Daemon: 180s
Shim: 240s
Client: 300s
```

Test 2: Test Workflow Tool Timeout

```
# Test with thinkdeep tool
# Should timeout at 120s (not 600s)
time python3 -c "
from tools.workflows.thinkdeep import ThinkDeepTool
import asyncio

async def test():
    tool = ThinkDeepTool()
    # Simulate long-running operation
    await asyncio.sleep(200)  # Longer than timeout

asyncio.run(test())
"
```

Expected: Timeout after ~120 seconds with error message.

Test 3: Test Expert Validation Timeout

```
# Test expert validation timeout
# Should timeout at 90s (not 300s)
```

Acceptance Criteria:
- ✅ TimeoutConfig class validates hierarchy on import
- ✅ Daemon timeout = 180s (1.5x tool timeout)
- ✅ Shim timeout = 240s (2x tool timeout)
- ✅ Client timeout = 300s (2.5x tool timeout)
- ✅ Workflow tools timeout at 120s
- ✅ Expert validation timeouts at 90s
- ✅ All three MCP configs updated consistently
- ✅ Documentation updated in .env.example

Fix #2: Progress Heartbeat Implementation

Priority: P0 - CRITICAL
Impact: Users perceive system as hanging during long operations
Estimated Time: 2 days
Dependencies: Fix #1 (timeout hierarchy)

Implementation Steps

Step 1: Create Progress Heartbeat Module

File: `utils/progress.py` (NEW)
Location: Create new file

Code:

```python
"""Progress heartbeat system for long-running operations."""

import asyncio
import time
import logging
from typing import Optional, Dict, Any, Callable

logger = logging.getLogger(__name__)


class ProgressHeartbeat:
    """Manages progress heartbeat for long-running operations."""

    def __init__(
        self,
        interval_secs: float = 6.0,
        callback: Optional[Callable] = None
    ):
        """
        Initialize progress heartbeat.

        Args:
            interval_secs: Seconds between heartbeats (default 6s)
            callback: Optional callback function for progress updates
        """
        self.interval = interval_secs
        self.callback = callback
        self.last_heartbeat = time.time()
        self.enabled = True
        self.start_time = time.time()
        self.total_steps = 0
        self.current_step = 0

    def set_total_steps(self, total: int):
        """Set total number of steps for progress calculation."""
        self.total_steps = total

    def set_current_step(self, step: int):
        """Set current step number."""
        self.current_step = step

    async def send_heartbeat(
        self,
        message: str,
        metadata: Optional[Dict[str, Any]] = None
    ):
        """
        Send progress heartbeat if interval elapsed.

        Args:
            message: Progress message
            metadata: Optional metadata dict
        """
        if not self.enabled:
            return

        now = time.time()
        if now - self.last_heartbeat >= self.interval:
            await self._emit_progress(message, metadata)
            self.last_heartbeat = now

    async def force_heartbeat(
```

```
        self,
        message: str,
        metadata: Optional[Dict[str, Any]] = None
    ):
        """Force send heartbeat regardless of interval."""
        await self._emit_progress(message, metadata)
        self.last_heartbeat = time.time()

    async def _emit_progress(
        self,
        message: str,
        metadata: Optional[Dict[str, Any]]
    ):
        """Emit progress message."""
        elapsed = time.time() - self.start_time

        # Calculate estimated remaining time
        estimated_remaining = None
        if self.total_steps > 0 and self.current_step > 0:
            time_per_step = elapsed / self.current_step
            remaining_steps = self.total_steps - self.current_step
            estimated_remaining = time_per_step * remaining_steps

        progress_data = {
            "type": "progress",
            "timestamp": time.time(),
            "message": message,
            "elapsed_secs": round(elapsed, 1),
            "estimated_remaining_secs": round(estimated_remaining, 1) if estim-
ated_remaining else None,
            "step": self.current_step if self.current_step > 0 else None,
            "total_steps": self.total_steps if self.total_steps > 0 else None,
            "metadata": metadata or {}
        }

        # Log progress
        logger.info(f"Progress: {message} (elapsed: {elapsed:.1f}s)")

        # Call callback if provided
        if self.callback:
            try:
                await self.callback(progress_data)
            except Exception as e:
                logger.warning(f"Progress callback failed: {e}")

    def stop(self):
        """Stop sending heartbeats."""
        self.enabled = False

    async def __aenter__(self):
        """Context manager entry."""
        self.enabled = True
        self.start_time = time.time()
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        """Context manager exit."""
        self.stop()
        return False


class ProgressTracker:
    """Tracks progress across multiple operations."""
```

```python
    def __init__(self):
        self.operations = {}

    def create_heartbeat(
        self,
        operation_id: str,
        interval_secs: float = 6.0,
        callback: Optional[Callable] = None
    ) -> ProgressHeartbeat:
        """Create new progress heartbeat for operation."""
        heartbeat = ProgressHeartbeat(interval_secs, callback)
        self.operations[operation_id] = heartbeat
        return heartbeat

    def get_heartbeat(self, operation_id: str) -> Optional[ProgressHeartbeat]:
        """Get existing heartbeat for operation."""
        return self.operations.get(operation_id)

    def remove_heartbeat(self, operation_id: str):
        """Remove heartbeat for completed operation."""
        if operation_id in self.operations:
            self.operations[operation_id].stop()
            del self.operations[operation_id]


# Global progress tracker
_progress_tracker = ProgressTracker()


def get_progress_tracker() -> ProgressTracker:
    """Get global progress tracker instance."""
    return _progress_tracker
```

Step 2: Integrate in Workflow Tools

File: `tools/workflow/base.py`
Location: Add to WorkflowTool class

Code to Add:

```python
from utils.progress import ProgressHeartbeat

class WorkflowTool:
    async def execute(self, request: dict) -> dict:
        """Execute workflow with progress heartbeat."""

        # Create progress heartbeat
        async with ProgressHeartbeat(
            interval_secs=6.0,
            callback=self._send_progress_to_client
        ) as heartbeat:

            # Set total steps
            total_steps = self._calculate_total_steps(request)
            heartbeat.set_total_steps(total_steps)

            # Execute workflow
            results = []
            for step_num in range(1, total_steps + 1):
                heartbeat.set_current_step(step_num)

                # Send heartbeat before step
                await heartbeat.send_heartbeat(
                    f"Executing step {step_num} of {total_steps}...",
                    metadata={"step_name": self._get_step_name(step_num)}
                )

                # Execute step
                step_result = await self._execute_step(step_num, request)
                results.append(step_result)

                # Send heartbeat after step
                await heartbeat.force_heartbeat(
                    f"Completed step {step_num} of {total_steps}",
                    metadata={"step_result": step_result}
                )

            return {"results": results}

    async def _send_progress_to_client(self, progress_data: dict):
        """Send progress update to client via WebSocket."""
        # Implementation depends on WebSocket architecture
        # This will be connected to ws_server.py
        pass
```

Step 3: Integrate in Expert Analysis

File: `tools/workflow/expert_analysis.py`
Location: Add to validate_with_expert method

Code to Add:

```python
from utils.progress import ProgressHeartbeat

async def validate_with_expert(
    self,
    request_id: str,
    content: str,
    context: dict
) -> dict:
    """Validate content with expert, with progress updates."""

    async with ProgressHeartbeat(interval_secs=8.0) as heartbeat:
        # Start validation
        await heartbeat.force_heartbeat("Starting expert validation...")

        # Prepare validation request
        await heartbeat.send_heartbeat("Preparing validation request...")
        validation_request = self._prepare_request(content, context)

        # Call expert
        await heartbeat.send_heartbeat("Consulting expert model...")
        expert_response = await self._call_expert(validation_request)

        # Process response
        await heartbeat.send_heartbeat("Processing expert feedback...")
        result = self._process_response(expert_response)

        # Complete
        await heartbeat.force_heartbeat("Expert validation complete")

        return result
```

Step 4: Integrate in Provider Calls

File: `src/providers/openai_compatible.py`
Location: Add to streaming methods

Code to Add:

```python
from utils.progress import ProgressHeartbeat

async def generate_streaming(
    self,
    prompt: str,
    model: str,
    **kwargs
) -> AsyncIterator[str]:
    """Generate streaming response with progress updates."""

    async with ProgressHeartbeat(interval_secs=5.0) as heartbeat:
        await heartbeat.force_heartbeat("Starting streaming generation...")

        chunk_count = 0
        async for chunk in self._stream_chunks(prompt, model, **kwargs):
            chunk_count += 1

            # Send heartbeat every 5 seconds
            await heartbeat.send_heartbeat(
                f"Streaming... ({chunk_count} chunks received)",
                metadata={"chunk_count": chunk_count}
            )

            yield chunk

        await heartbeat.force_heartbeat(
            f"Streaming complete ({chunk_count} chunks total)"
        )
```

Step 5: Connect to WebSocket Server

File: `src/daemon/ws_server.py`
Location: Add progress message routing

Code to Add:

```python
async def send_progress_update(
    self,
    session_id: str,
    progress_data: dict
):
    """Send progress update to client."""
    if session_id in self.sessions:
        websocket = self.sessions[session_id]["websocket"]
        try:
            await websocket.send_json({
                "type": "progress",
                "data": progress_data
            })
        except Exception as e:
            logger.warning(f"Failed to send progress update: {e}")
```

Step 6: Update Client Handling

File: `scripts/run_ws_shim.py`
Location: Add progress message handling

Code to Add:

```python
async def handle_progress_message(self, message: dict):
    """Handle progress message from daemon."""
    progress_data = message.get("data", {})

    # Log progress
    logger.info(
        f"Progress: {progress_data.get('message')} "
        f"(elapsed: {progress_data.get('elapsed_secs')}s)"
    )

    # Optionally forward to MCP client
    # (depends on MCP protocol support for progress updates)
```

Testing Instructions

Test 1: Verify Heartbeat Timing

```python
import asyncio
from utils.progress import ProgressHeartbeat

async def test_heartbeat():
    messages = []

    async def callback(data):
        messages.append(data)

    async with ProgressHeartbeat(interval_secs=2.0, callback=callback) as hb:
        for i in range(10):
            await hb.send_heartbeat(f"Step {i}")
            await asyncio.sleep(1)

    # Should have ~5 messages (10 seconds / 2 second interval)
    print(f"Received {len(messages)} heartbeats")
    assert 4 <= len(messages) <= 6

asyncio.run(test_heartbeat())
```

Test 2: Test Workflow Tool Progress

```python
# Test with thinkdeep tool
# Should see progress updates every 6 seconds
```

Test 3: Test Expert Validation Progress

```python
# Test expert validation
# Should see progress updates every 8 seconds
```

Acceptance Criteria:
- ✅ ProgressHeartbeat class sends updates at configured interval
- ✅ Workflow tools send progress updates every 6 seconds
- ✅ Expert validation sends progress updates every 8 seconds
- ✅ Provider calls send progress updates every 5 seconds
- ✅ Progress messages include elapsed time and estimated remaining
- ✅ Progress messages logged correctly

- ✅ WebSocket server routes progress messages to clients
- ✅ No performance degradation from heartbeat system

Fix #3: Logging Infrastructure Unification

Priority: P0 - CRITICAL
Impact: Cannot debug workflow tool failures, no visibility into execution
Estimated Time: 2 days
Dependencies: None

Implementation Steps

Step 1: Create Unified Logger Module

File: `utils/logging_unified.py` (NEW)
Location: Create new file

Code:

```python
"""Unified logging infrastructure for all tools."""

import json
import time
import logging
import traceback
from pathlib import Path
from typing import Optional, Dict, Any
from datetime import datetime

logger = logging.getLogger(__name__)


class UnifiedLogger:
    """Unified logging for all tools with structured output."""

    def __init__(self, log_file: str = ".logs/toolcalls.jsonl"):
        """
        Initialize unified logger.

        Args:
            log_file: Path to log file (JSONL format)
        """
        self.log_file = Path(log_file)
        self.log_file.parent.mkdir(parents=True, exist_ok=True)
        self.buffer = []
        self.buffer_size = 10  # Flush after 10 entries

    def log_tool_start(
        self,
        tool_name: str,
        request_id: str,
        params: Dict[str, Any]
    ):
        """
        Log tool execution start.

        Args:
            tool_name: Name of the tool
            request_id: Unique request identifier
            params: Tool parameters
        """
        entry = {
            "timestamp": time.time(),
            "datetime": datetime.utcnow().isoformat(),
            "event": "tool_start",
            "tool": tool_name,
            "request_id": request_id,
            "params": self._sanitize_params(params)
        }
        self._write_log(entry)
        logger.info(f"Tool started: {tool_name} (request_id={request_id})")

    def log_tool_progress(
        self,
        tool_name: str,
        request_id: str,
        step: int,
        total_steps: int,
        message: str,
        metadata: Optional[Dict[str, Any]] = None
    ):
```

```
        """
        Log tool execution progress.

        Args:
            tool_name: Name of the tool
            request_id: Unique request identifier
            step: Current step number
            total_steps: Total number of steps
            message: Progress message
            metadata: Optional metadata
        """
        entry = {
            "timestamp": time.time(),
            "datetime": datetime.utcnow().isoformat(),
            "event": "tool_progress",
            "tool": tool_name,
            "request_id": request_id,
            "step": step,
            "total_steps": total_steps,
            "message": message,
            "metadata": metadata or {}
        }
        self._write_log(entry)
        logger.debug(
            f"Tool progress: {tool_name} step {step}/{total_steps} - {message}"
        )

    def log_tool_complete(
        self,
        tool_name: str,
        request_id: str,
        duration_s: float,
        result_preview: str,
        metadata: Optional[Dict[str, Any]] = None
    ):
        """
        Log tool execution completion.

        Args:
            tool_name: Name of the tool
            request_id: Unique request identifier
            duration_s: Execution duration in seconds
            result_preview: Preview of result (first 200 chars)
            metadata: Optional metadata
        """
        entry = {
            "timestamp": time.time(),
            "datetime": datetime.utcnow().isoformat(),
            "event": "tool_complete",
            "tool": tool_name,
            "request_id": request_id,
            "duration_s": round(duration_s, 3),
            "result_preview": result_preview[:200] if result_preview else None,
            "metadata": metadata or {}
        }
        self._write_log(entry)
        logger.info(
            f"Tool completed: {tool_name} in {duration_s:.2f}s "
            f"(request_id={request_id})"
        )

    def log_tool_error(
        self,
```

```
        tool_name: str,
        request_id: str,
        error: str,
        error_traceback: Optional[str] = None,
        metadata: Optional[Dict[str, Any]] = None
    ):
        """
        Log tool execution error.

        Args:
            tool_name: Name of the tool
            request_id: Unique request identifier
            error: Error message
            error_traceback: Optional traceback string
            metadata: Optional metadata
        """
        entry = {
            "timestamp": time.time(),
            "datetime": datetime.utcnow().isoformat(),
            "event": "tool_error",
            "tool": tool_name,
            "request_id": request_id,
            "error": error,
            "traceback": error_traceback,
            "metadata": metadata or {}
        }
        self._write_log(entry)
        logger.error(
            f"Tool error: {tool_name} - {error} (request_id={request_id})"
        )

    def log_expert_validation_start(
        self,
        tool_name: str,
        request_id: str,
        content_preview: str
    ):
        """Log expert validation start."""
        entry = {
            "timestamp": time.time(),
            "datetime": datetime.utcnow().isoformat(),
            "event": "expert_validation_start",
            "tool": tool_name,
            "request_id": request_id,
            "content_preview": content_preview[:100]
        }
        self._write_log(entry)
        logger.info(f"Expert validation started for {tool_name}")

    def log_expert_validation_complete(
        self,
        tool_name: str,
        request_id: str,
        duration_s: float,
        validation_result: str
    ):
        """Log expert validation completion."""
        entry = {
            "timestamp": time.time(),
            "datetime": datetime.utcnow().isoformat(),
            "event": "expert_validation_complete",
            "tool": tool_name,
            "request_id": request_id,
```

```
            "duration_s": round(duration_s, 3),
            "validation_result": validation_result[:200]
        }
        self._write_log(entry)
        logger.info(
            f"Expert validation completed for {tool_name} in {duration_s:.2f}s"
        )

    def _sanitize_params(self, params: Dict[str, Any]) -> Dict[str, Any]:
        """Sanitize parameters for logging (remove sensitive data)."""
        sanitized = {}
        for key, value in params.items():
            if key.lower() in ["api_key", "token", "password", "secret"]:
                sanitized[key] = "***REDACTED***"
            elif isinstance(value, str) and len(value) > 500:
                sanitized[key] = value[:500] + "...[truncated]"
            else:
                sanitized[key] = value
        return sanitized

    def _write_log(self, entry: Dict[str, Any]):
        """Write log entry to file."""
        self.buffer.append(entry)

        # Flush buffer if full
        if len(self.buffer) >= self.buffer_size:
            self._flush_buffer()

    def _flush_buffer(self):
        """Flush buffered log entries to file."""
        if not self.buffer:
            return

        try:
            with open(self.log_file, "a") as f:
                for entry in self.buffer:
                    f.write(json.dumps(entry) + "\n")
            self.buffer.clear()
        except Exception as e:
            logger.error(f"Failed to write logs: {e}")

    def flush(self):
        """Manually flush buffer."""
        self._flush_buffer()

    def __del__(self):
        """Flush buffer on deletion."""
        self._flush_buffer()


# Global unified logger instance
_unified_logger = UnifiedLogger()


def get_unified_logger() -> UnifiedLogger:
    """Get global unified logger instance."""
    return _unified_logger
```

Step 2: Update Simple Tools

File: `tools/simple/base.py`
Location: Update execute method to use unified logger

Before:

```
async def execute(self, request: dict) -> dict:
    # Existing logging code
    pass
```

After:

```
from utils.logging_unified import get_unified_logger

async def execute(self, request: dict) -> dict:
    """Execute simple tool with unified logging."""
    logger = get_unified_logger()
    request_id = request.get("request_id") or str(uuid.uuid4())
    start_time = time.time()

    # Log start
    logger.log_tool_start(
        tool_name=self.name,
        request_id=request_id,
        params=request
    )

    try:
        # Execute tool
        result = await self._execute_impl(request)

        # Log completion
        duration = time.time() - start_time
        logger.log_tool_complete(
            tool_name=self.name,
            request_id=request_id,
            duration_s=duration,
            result_preview=str(result)
        )

        return result

    except Exception as e:
        # Log error
        duration = time.time() - start_time
        logger.log_tool_error(
            tool_name=self.name,
            request_id=request_id,
            error=str(e),
            error_traceback=traceback.format_exc(),
            metadata={"duration_s": duration}
        )
        raise
```

Step 3: Update Workflow Tools

File: `tools/workflow/base.py`
Location: Update execute method to use unified logger

Code to Add:

```python
from utils.logging_unified import get_unified_logger

async def execute(self, request: dict) -> dict:
    """Execute workflow with unified logging."""
    logger = get_unified_logger()
    request_id = request.get("request_id") or str(uuid.uuid4())
    start_time = time.time()

    # Log start
    logger.log_tool_start(
        tool_name=self.name,
        request_id=request_id,
        params=request
    )

    try:
        # Execute workflow steps
        total_steps = self._calculate_total_steps(request)
        results = []

        for step_num in range(1, total_steps + 1):
            # Log progress
            logger.log_tool_progress(
                tool_name=self.name,
                request_id=request_id,
                step=step_num,
                total_steps=total_steps,
                message=f"Executing step {step_num}",
                metadata={"step_name": self._get_step_name(step_num)}
            )

            # Execute step
            step_result = await self._execute_step(step_num, request)
            results.append(step_result)

        # Log completion
        duration = time.time() - start_time
        logger.log_tool_complete(
            tool_name=self.name,
            request_id=request_id,
            duration_s=duration,
            result_preview=str(results),
            metadata={"total_steps": total_steps}
        )

        return {"results": results}

    except Exception as e:
        # Log error
        duration = time.time() - start_time
        logger.log_tool_error(
            tool_name=self.name,
            request_id=request_id,
            error=str(e),
            error_traceback=traceback.format_exc(),
            metadata={"duration_s": duration}
        )
        raise
```

Step 4: Update Expert Analysis

File: `tools/workflow/expert_analysis.py`
Location: Add logging to validate_with_expert method

Code to Add:

```
from utils.logging_unified import get_unified_logger

async def validate_with_expert(
    self,
    request_id: str,
    content: str,
    context: dict
) -> dict:
    """Validate content with expert, with logging."""
    logger = get_unified_logger()
    start_time = time.time()

    # Log start
    logger.log_expert_validation_start(
        tool_name=context.get("tool_name", "unknown"),
        request_id=request_id,
        content_preview=content
    )

    try:
        # Perform validation
        result = await self._perform_validation(content, context)

        # Log completion
        duration = time.time() - start_time
        logger.log_expert_validation_complete(
            tool_name=context.get("tool_name", "unknown"),
            request_id=request_id,
            duration_s=duration,
            validation_result=str(result)
        )

        return result

    except Exception as e:
        # Log error
        duration = time.time() - start_time
        logger.log_tool_error(
            tool_name=context.get("tool_name", "unknown"),
            request_id=request_id,
            error=f"Expert validation failed: {e}",
            error_traceback=traceback.format_exc(),
            metadata={"duration_s": duration}
        )
        raise
```

Step 5: Add Request ID Generation

File: `src/daemon/ws_server.py`
Location: Add request_id to all tool calls

Code to Add:

```python
import uuid

async def handle_tool_call(self, session_id: str, tool_name: str, params: dict):
    """Handle tool call with request_id."""
    # Generate request_id if not provided
    if "request_id" not in params:
        params["request_id"] = str(uuid.uuid4())

    # Execute tool
    result = await self._execute_tool(tool_name, params)

    return result
```

Testing Instructions

Test 1: Verify Simple Tool Logging

```bash
# Clear logs
rm .logs/toolcalls.jsonl

# Run simple tool
python3 -c "
from tools.simple.chat import ChatTool
import asyncio

async def test():
    tool = ChatTool()
    result = await tool.execute({'prompt': 'Hello'})

asyncio.run(test())
"

# Check logs
cat .logs/toolcalls.jsonl | jq .
```

Expected: 2 log entries (tool_start, tool_complete)

Test 2: Verify Workflow Tool Logging

```bash
# Clear logs
rm .logs/toolcalls.jsonl

# Run workflow tool
# Should see: tool_start, multiple tool_progress, tool_complete
```

Expected: Multiple log entries showing progress through workflow

Test 3: Verify Expert Validation Logging

```bash
# Clear logs
rm .logs/toolcalls.jsonl

# Run tool with expert validation
# Should see: tool_start, expert_validation_start, expert_validation_complete,
tool_complete
```

Acceptance Criteria:
- ✅ UnifiedLogger class created with all required methods
- ✅ Simple tools log correctly (start, complete, error)
- ✅ Workflow tools log correctly (start, progress, complete, error)
- ✅ Expert validation logs correctly (start, complete)
- ✅ All logs include request_id for tracking
- ✅ Logs written to .logs/toolcalls.jsonl in JSONL format
- ✅ Sensitive data sanitized in logs
- ✅ Log buffer flushes correctly
- ✅ No performance degradation from logging

SECTION 4: HIGH PRIORITY FIXES (Week 2 - P1 Issues)

Fix #4: Expert Validation Duplicate Call Bug

Priority: P1 - HIGH
Impact: Key feature disabled, quality degraded
Estimated Time: 3 days
Dependencies: Fix #3 (logging infrastructure)

Investigation Steps

Step 1: Add Detailed Logging

File: `tools/workflow/expert_analysis.py`
Location: Add logging to all methods

Code to Add:

```
import logging
logger = logging.getLogger(__name__)

class ExpertAnalysis:
    def __init__(self):
        self._call_count = {}  # Track calls by request_id

    async def validate_with_expert(self, request_id: str, content: str, context: dict)
:
        """Validate with detailed logging."""
        # Track call
        if request_id not in self._call_count:
            self._call_count[request_id] = 0
        self._call_count[request_id] += 1

        call_num = self._call_count[request_id]
        logger.warning(
            f"Expert validation called for request_id={request_id}, "
            f"call_num={call_num}, "
            f"stack_trace={''.join(traceback.format_stack())}"
        )

        # Rest of validation logic
        pass
```

Step 2: Run Test to Identify Duplicate Calls

Test Command:

```
# Enable debug logging
export LOG_LEVEL=DEBUG

# Run debug tool with 2 steps
python3 -c "
from tools.workflows.debug import DebugTool
import asyncio

async def test():
    tool = DebugTool()
    result = await tool.execute({
        'prompt': 'Debug this code',
        'max_steps': 2
    })

asyncio.run(test())
"

# Check logs for duplicate calls
grep "Expert validation called" .logs/toolcalls.jsonl
```

Expected Output: Should show if expert validation is called multiple times for same request_id.

Step 3: Analyze Call Stack

Review the stack traces from Step 2 to identify:
1. Where duplicate calls originate
2. If there's recursion
3. If there's an event loop triggering re-validation
4. If continuation system triggers re-validation

Step 4: Implement Fix Based on Root Cause

Scenario A: Recursion in Expert Analysis

File: `tools/workflow/expert_analysis.py`

Fix:

```python
class ExpertAnalysis:
    def __init__(self):
        self._validation_cache = {}
        self._in_progress = set()

    async def validate_with_expert(self, request_id: str, content: str, context: dict)
:
        """Validate with duplicate prevention."""
        # Create cache key
        cache_key = f"{request_id}:{hash(content)}"

        # Check cache
        if cache_key in self._validation_cache:
            logger.info(f"Using cached validation for {request_id}")
            return self._validation_cache[cache_key]

        # Check if in progress
        if cache_key in self._in_progress:
            logger.warning(f"Validation already in progress for {request_id}")
            # Wait for completion
            while cache_key in self._in_progress:
                await asyncio.sleep(0.5)
            return self._validation_cache.get(cache_key)

        # Mark as in progress
        self._in_progress.add(cache_key)

        try:
            # Perform validation
            result = await self._perform_validation(content, context)

            # Cache result
            self._validation_cache[cache_key] = result

            return result

        finally:
            # Remove from in progress
            self._in_progress.discard(cache_key)
```

Scenario B: Workflow Tool Calling Expert Multiple Times

File: `tools/workflow/base.py`

Fix:

```python
class WorkflowTool:
    def __init__(self):
        self._expert_validated_steps = set()

    async def _execute_step_with_expert(self, step_num: int, request: dict):
        """Execute step with expert validation (once per step)."""
        step_key = f"{request.get('request_id')}:{step_num}"

        # Check if already validated
        if step_key in self._expert_validated_steps:
            logger.info(f"Step {step_num} already validated, skipping")
            return await self._execute_step(step_num, request)

        # Execute step
        result = await self._execute_step(step_num, request)

        # Validate with expert (only once)
        if self.use_expert_validation:
            result = await self._validate_step_result(result, request)
            self._expert_validated_steps.add(step_key)

        return result
```

Scenario C: Event Loop Triggering Re-validation

File: `tools/workflow/base.py`

Fix:

```python
class WorkflowTool:
    async def execute(self, request: dict):
        """Execute workflow without event loop re-validation."""
        # Disable event-driven validation
        self._disable_event_validation()

        try:
            # Execute workflow
            result = await self._execute_workflow(request)
            return result
        finally:
            # Re-enable event validation
            self._enable_event_validation()
```

Step 5: Re-enable Expert Validation

File: `.env`
Location: Update DEFAULT_USE_ASSISTANT_MODEL

Before:

```
DEFAULT_USE_ASSISTANT_MODEL=false
```

After:

```
DEFAULT_USE_ASSISTANT_MODEL=true
```

Step 6: Test Fix

Test Command:

```
# Clear logs
rm .logs/toolcalls.jsonl

# Run debug tool with 2 steps
python3 -c "
from tools.workflows.debug import DebugTool
import asyncio

async def test():
    tool = DebugTool()
    result = await tool.execute({
        'prompt': 'Debug this code',
        'max_steps': 2
    })
    print(f'Duration: {result.get(\"duration_s\")}s')
    print(f'Expert analysis: {result.get(\"expert_analysis\")[:100]}')

asyncio.run(test())
"

# Check logs
grep "Expert validation called" .logs/toolcalls.jsonl | wc -l
```

Expected:
- Expert validation called exactly 2 times (once per step)
- Duration: 90-120 seconds (not 300+)
- Expert analysis contains real content (not null)

Testing Instructions

Test 1: Verify No Duplicate Calls

```
# Should see exactly 2 expert validation calls for 2-step workflow
grep "Expert validation called" .logs/toolcalls.jsonl
```

Test 2: Verify Duration

```
# Should complete in 90-120 seconds
time python3 -c "from tools.workflows.debug import DebugTool; ..."
```

Test 3: Verify Expert Analysis Content

```
# Expert analysis should contain real content
cat .logs/toolcalls.jsonl | jq '.expert_analysis'
```

Acceptance Criteria:
- ✅ Root cause identified and documented
- ✅ Fix implemented based on root cause
- ✅ Expert validation called exactly once per step
- ✅ Duration is 90-120 seconds (not 300+)
- ✅ Expert analysis contains real content (not null)

- ✅ No duplicate calls in logs
- ✅ Expert validation re-enabled in .env
- ✅ All workflow tools tested with expert validation

Fix #5: Standardize Timeout Configurations

Priority: P1 - HIGH
Impact: Unpredictable behavior across clients
Estimated Time: 2 days
Dependencies: Fix #1 (timeout hierarchy)

Implementation Steps

Step 1: Create Base Configuration Template

File: `Daemon/mcp-config.base.json` (NEW)
Location: Create new file

Code:

```json
{
  "mcpServers": {
    "exai": {
      "command": "python",
      "args": [
        "-u",
        "scripts/run_ws_shim.py"
      ],
      "env": {
        "PYTHONUNBUFFERED": "1",
        "PYTHONPATH": ".",

        "_comment_timeouts": "Coordinated timeout hierarchy (auto-calculated)",
        "SIMPLE_TOOL_TIMEOUT_SECS": "60",
        "WORKFLOW_TOOL_TIMEOUT_SECS": "120",
        "EXPERT_ANALYSIS_TIMEOUT_SECS": "90",

        "_comment_providers": "Provider timeouts",
        "GLM_TIMEOUT_SECS": "90",
        "KIMI_TIMEOUT_SECS": "120",
        "KIMI_WEB_SEARCH_TIMEOUT_SECS": "150",

        "_comment_concurrency": "Concurrency limits",
        "EXAI_WS_SESSION_MAX_INFLIGHT": "6",
        "EXAI_WS_GLOBAL_MAX_INFLIGHT": "16",
        "EXAI_WS_GLM_MAX_INFLIGHT": "8",
        "EXAI_WS_KIMI_MAX_INFLIGHT": "4",

        "_comment_session": "Session management",
        "EX_SESSION_SCOPE_STRICT": "false",
        "EX_SESSION_SCOPE_ALLOW_CROSS_SESSION": "true",

        "_comment_features": "Feature flags",
        "DEFAULT_USE_ASSISTANT_MODEL": "true",
        "EX_ALLOW_RELATIVE_PATHS": "true",
        "GLM_ENABLE_WEB_SEARCH": "true",
        "KIMI_ENABLE_INTERNET_SEARCH": "true"
      }
    }
  }
}
```

Step 2: Update Auggie Configuration

File: `Daemon/mcp-config.auggie.json`
Location: Replace entire file

Code:

```
{
  "mcpServers": {
    "exai": {
      "command": "python",
      "args": ["-u", "scripts/run_ws_shim.py"],
      "env": {
        "PYTHONUNBUFFERED": "1",
        "PYTHONPATH": ".",

        "_comment": "Auggie CLI Configuration - Optimized for autonomous operation",

        "_comment_timeouts": "Standard timeouts (same as base)",
        "SIMPLE_TOOL_TIMEOUT_SECS": "60",
        "WORKFLOW_TOOL_TIMEOUT_SECS": "120",
        "EXPERT_ANALYSIS_TIMEOUT_SECS": "90",
        "GLM_TIMEOUT_SECS": "90",
        "KIMI_TIMEOUT_SECS": "120",
        "KIMI_WEB_SEARCH_TIMEOUT_SECS": "150",

        "_comment_concurrency": "Auggie-specific: Higher GLM concurrency for heavy us-
age",
        "EXAI_WS_SESSION_MAX_INFLIGHT": "6",
        "EXAI_WS_GLOBAL_MAX_INFLIGHT": "16",
        "EXAI_WS_GLM_MAX_INFLIGHT": "10",
        "EXAI_WS_KIMI_MAX_INFLIGHT": "4",

        "_comment_session": "Auggie-specific: Flexible session management",
        "EX_SESSION_SCOPE_STRICT": "false",
        "EX_SESSION_SCOPE_ALLOW_CROSS_SESSION": "true",

        "_comment_features": "Standard features",
        "DEFAULT_USE_ASSISTANT_MODEL": "true",
        "EX_ALLOW_RELATIVE_PATHS": "true",
        "GLM_ENABLE_WEB_SEARCH": "true",
        "KIMI_ENABLE_INTERNET_SEARCH": "true"
      }
    }
  }
}
```

Step 3: Update VSCode Augment Configuration

File: `Daemon/mcp-config.augmentcode.json`
Location: Replace entire file

Code:

```
{
  "mcpServers": {
    "exai": {
      "command": "python",
      "args": ["-u", "scripts/run_ws_shim.py"],
      "env": {
        "PYTHONUNBUFFERED": "1",
        "PYTHONPATH": ".",

        "_comment": "VSCode Augment Configuration - Optimized for IDE integration",

        "_comment_timeouts": "Standard timeouts (same as base)",
        "SIMPLE_TOOL_TIMEOUT_SECS": "60",
        "WORKFLOW_TOOL_TIMEOUT_SECS": "120",
        "EXPERT_ANALYSIS_TIMEOUT_SECS": "90",
        "GLM_TIMEOUT_SECS": "90",
        "KIMI_TIMEOUT_SECS": "120",
        "KIMI_WEB_SEARCH_TIMEOUT_SECS": "150",

        "_comment_concurrency": "VSCode-specific: Balanced concurrency",
        "EXAI_WS_SESSION_MAX_INFLIGHT": "8",
        "EXAI_WS_GLOBAL_MAX_INFLIGHT": "20",
        "EXAI_WS_GLM_MAX_INFLIGHT": "8",
        "EXAI_WS_KIMI_MAX_INFLIGHT": "6",

        "_comment_session": "VSCode-specific: Strict session management",
        "EX_SESSION_SCOPE_STRICT": "true",
        "EX_SESSION_SCOPE_ALLOW_CROSS_SESSION": "false",

        "_comment_features": "Standard features",
        "DEFAULT_USE_ASSISTANT_MODEL": "true",
        "EX_ALLOW_RELATIVE_PATHS": "true",
        "GLM_ENABLE_WEB_SEARCH": "true",
        "KIMI_ENABLE_INTERNET_SEARCH": "true"
      }
    }
  }
}
```

Step 4: Update Claude Desktop Configuration

File: `Daemon/mcp-config.claude.json`
Location: Replace entire file

Code:

```
{
  "mcpServers": {
    "exai": {
      "command": "python",
      "args": ["-u", "scripts/run_ws_shim.py"],
      "env": {
        "PYTHONUNBUFFERED": "1",
        "PYTHONPATH": ".",

        "_comment": "Claude Desktop Configuration - Optimized for desktop app",

        "_comment_timeouts": "Standard timeouts (same as base)",
        "SIMPLE_TOOL_TIMEOUT_SECS": "60",
        "WORKFLOW_TOOL_TIMEOUT_SECS": "120",
        "EXPERT_ANALYSIS_TIMEOUT_SECS": "90",
        "GLM_TIMEOUT_SECS": "90",
        "KIMI_TIMEOUT_SECS": "120",
        "KIMI_WEB_SEARCH_TIMEOUT_SECS": "150",

        "_comment_concurrency": "Claude-specific: Conservative concurrency",
        "EXAI_WS_SESSION_MAX_INFLIGHT": "4",
        "EXAI_WS_GLOBAL_MAX_INFLIGHT": "12",
        "EXAI_WS_GLM_MAX_INFLIGHT": "6",
        "EXAI_WS_KIMI_MAX_INFLIGHT": "4",

        "_comment_session": "Claude-specific: Strict session management",
        "EX_SESSION_SCOPE_STRICT": "true",
        "EX_SESSION_SCOPE_ALLOW_CROSS_SESSION": "false",

        "_comment_features": "Standard features",
        "DEFAULT_USE_ASSISTANT_MODEL": "true",
        "EX_ALLOW_RELATIVE_PATHS": "true",
        "GLM_ENABLE_WEB_SEARCH": "true",
        "KIMI_ENABLE_INTERNET_SEARCH": "true"
      }
    }
  }
}
```

Step 5: Document Configuration Differences

File: `docs/configuration/mcp-configs.md` (NEW)
Location: Create new file

Code:

```markdown
# MCP Configuration Guide

## Overview

EX-AI MCP Server supports three client configurations:
1. **Auggie CLI** - Autonomous operation
2. **VSCode Augment** - IDE integration
3. **Claude Desktop** - Desktop app

## Standard Configuration (Base)

All configurations share these standard settings:

### Timeouts (Coordinated Hierarchy)
- `SIMPLE_TOOL_TIMEOUT_SECS`: 60s
- `WORKFLOW_TOOL_TIMEOUT_SECS`: 120s
- `EXPERT_ANALYSIS_TIMEOUT_SECS`: 90s
- `GLM_TIMEOUT_SECS`: 90s
- `KIMI_TIMEOUT_SECS`: 120s
- `KIMI_WEB_SEARCH_TIMEOUT_SECS`: 150s

Infrastructure timeouts are auto-calculated:
- Daemon: 180s (1.5x workflow timeout)
- Shim: 240s (2x workflow timeout)
- Client: 300s (2.5x workflow timeout)

### Features
- Expert validation: Enabled
- Relative paths: Enabled
- Web search: Enabled (GLM and Kimi)

## Client-Specific Differences

### Auggie CLI

**Optimized for:** Autonomous operation, long sessions

**Concurrency:**
- Session max: 6 (focused execution)
- Global max: 16
- GLM max: 10 (heavy GLM usage)
- Kimi max: 4

**Session Management:**
- Strict: false (flexible)
- Cross-session: true (continuity)

**Use Case:** 30-60 minute autonomous refactoring sessions

### VSCode Augment

**Optimized for:** IDE integration, responsive UI

**Concurrency:**
- Session max: 8 (balanced)
- Global max: 20
- GLM max: 8
- Kimi max: 6

**Session Management:**
- Strict: true (isolated)
- Cross-session: false (clean state)
```

```
**Use Case:** Interactive coding assistance in IDE

### Claude Desktop

**Optimized for:** Desktop app, conservative resource usage

**Concurrency:**
- Session max: 4 (conservative)
- Global max: 12
- GLM max: 6
- Kimi max: 4

**Session Management:**
- Strict: true (isolated)
- Cross-session: false (clean state)

**Use Case:** Desktop chat interface

## Customization

To customize configuration:

1. Copy `mcp-config.base.json` to `mcp-config.custom.json`
2. Modify values as needed
3. Update client to use custom config
4. Test thoroughly before deploying

## Validation

All configurations are validated on startup:
- Timeout hierarchy checked
- Concurrency limits validated
- Feature flags verified

Invalid configurations will fail fast with clear error messages.
```

Testing Instructions

Test 1: Verify Configuration Consistency

```
# Check all configs have same timeout values
grep "WORKFLOW_TOOL_TIMEOUT_SECS" Daemon/mcp-config.*.json
```

Expected: All configs show “120”

Test 2: Test Each Client

```
# Test Auggie CLI
# Test VSCode Augment
# Test Claude Desktop
# All should have consistent timeout behavior
```

Test 3: Verify Documentation

```
# Check documentation is accurate
cat docs/configuration/mcp-configs.md
```

Acceptance Criteria:
- ✅ Base configuration template created
- ✅ All three client configs updated consistently
- ✅ Timeout values standardized across all configs
- ✅ Only concurrency and session management differ between clients
- ✅ Configuration differences documented
- ✅ All configs tested and working
- ✅ Documentation accurate and complete

Fix #6: Graceful Degradation Implementation

Priority: P1 - HIGH
Impact: Silent failures, no fallback strategies
Estimated Time: 2 days
Dependencies: Fix #1 (timeout hierarchy), Fix #3 (logging)

Implementation Steps

Step 1: Create Graceful Degradation Module

File: `utils/error_handling.py` (NEW)
Location: Create new file

Code:

```python
"""Graceful degradation and error handling utilities."""

import asyncio
import time
import logging
from typing import Callable, Any, Optional, Dict
from functools import wraps

logger = logging.getLogger(__name__)


class CircuitBreakerOpen(Exception):
    """Raised when circuit breaker is open."""
    pass


class GracefulDegradation:
    """Handles graceful degradation for failures."""

    def __init__(self):
        self.circuit_breakers: Dict[str, Dict] = {}
        self.failure_threshold = 5
        self.recovery_timeout = 300  # 5 minutes

    async def execute_with_fallback(
        self,
        primary_fn: Callable,
        fallback_fn: Optional[Callable] = None,
        timeout_secs: float = 60.0,
        max_retries: int = 2,
        operation_name: str = None
    ) -> Any:
        """
        Execute function with fallback and timeout.

        Args:
            primary_fn: Primary function to execute
            fallback_fn: Optional fallback function
            timeout_secs: Timeout in seconds
            max_retries: Maximum retry attempts
            operation_name: Name for circuit breaker tracking

        Returns:
            Result from primary or fallback function

        Raises:
            CircuitBreakerOpen: If circuit breaker is open
            Exception: If both primary and fallback fail
        """
        op_name = operation_name or primary_fn.__name__

        # Check circuit breaker
        if self._is_circuit_open(op_name):
            logger.warning(f"Circuit breaker open for {op_name}")
            if fallback_fn:
                logger.info(f"Using fallback for {op_name}")
                return await self._execute_with_timeout(
                    fallback_fn, timeout_secs
                )
            raise CircuitBreakerOpen(f"{op_name} circuit breaker is open")

        # Try primary function with retries
```

```
        last_error = None
        for attempt in range(max_retries + 1):
            try:
                result = await self._execute_with_timeout(
                    primary_fn, timeout_secs
                )
                self._record_success(op_name)
                return result

            except asyncio.TimeoutError as e:
                last_error = e
                logger.warning(
                    f"{op_name} timed out (attempt {attempt + 1}/{max_retries + 1})"
                )
                if attempt < max_retries:
                    await asyncio.sleep(2 ** attempt)  # Exponential backoff
                    continue

            except Exception as e:
                last_error = e
                logger.warning(
                    f"{op_name} failed: {e} (attempt {attempt + 1}/{max_retries + 1})"
                )
                if attempt < max_retries:
                    await asyncio.sleep(2 ** attempt)
                    continue

        # All retries failed
        self._record_failure(op_name)

        # Try fallback
        if fallback_fn:
            logger.info(f"Primary failed, using fallback for {op_name}")
            try:
                return await self._execute_with_timeout(
                    fallback_fn, timeout_secs
                )
            except Exception as fallback_error:
                logger.error(f"Fallback also failed for {op_name}: {fallback_error}")
                raise

        # No fallback, raise last error
        raise last_error

    async def _execute_with_timeout(
        self,
        fn: Callable,
        timeout_secs: float
    ) -> Any:
        """Execute function with timeout."""
        if asyncio.iscoroutinefunction(fn):
            return await asyncio.wait_for(fn(), timeout=timeout_secs)
        else:
            return await asyncio.wait_for(
                asyncio.to_thread(fn), timeout=timeout_secs
            )

    def _is_circuit_open(self, operation_name: str) -> bool:
        """Check if circuit breaker is open."""
        if operation_name not in self.circuit_breakers:
            return False

        cb = self.circuit_breakers[operation_name]
```

```
        # Check if enough failures
        if cb["failures"]
< self.failure_threshold:
            return False

        # Check if recovery timeout elapsed
        time_since_failure = time.time() - cb["last_failure"]
        if time_since_failure >= self.recovery_timeout:
            # Reset circuit breaker
            logger.info(f"Circuit breaker recovered for {operation_name}")
            cb["failures"] = 0
            return False

        return True

    def _record_success(self, operation_name: str):
        """Record successful execution."""
        if operation_name in self.circuit_breakers:
            self.circuit_breakers[operation_name]["failures"] = 0
            logger.debug(f"Circuit breaker reset for {operation_name}")

    def _record_failure(self, operation_name: str):
        """Record failed execution."""
        if operation_name not in self.circuit_breakers:
            self.circuit_breakers[operation_name] = {
                "failures": 0,
                "last_failure": 0
            }

        self.circuit_breakers[operation_name]["failures"] += 1
        self.circuit_breakers[operation_name]["last_failure"] = time.time()

        failures = self.circuit_breakers[operation_name]["failures"]
        logger.warning(
            f"Circuit breaker failure recorded for {operation_name} "
            f"({failures}/{self.failure_threshold})"
        )

    def get_circuit_status(self, operation_name: str) -> Dict[str, Any]:
        """Get circuit breaker status."""
        if operation_name not in self.circuit_breakers:
            return {"status": "closed", "failures": 0}

        cb = self.circuit_breakers[operation_name]
        is_open = self._is_circuit_open(operation_name)

        return {
            "status": "open" if is_open else "closed",
            "failures": cb["failures"],
            "last_failure": cb["last_failure"],
            "time_until_recovery": max(
                0,
                self.recovery_timeout - (time.time() - cb["last_failure"])
            ) if is_open else 0
        }


# Global graceful degradation instance
_graceful_degradation = GracefulDegradation()


def get_graceful_degradation() -> GracefulDegradation:
```

```python
    """Get global graceful degradation instance."""
    return _graceful_degradation


def with_fallback(
    fallback_fn: Optional[Callable] = None,
    timeout_secs: float = 60.0,
    max_retries: int = 2
):
    """
    Decorator for graceful degradation with fallback.

    Usage:
        @with_fallback(fallback_fn=my_fallback, timeout_secs=30)
        async def my_function():
            # Primary implementation
            pass
    """
    def decorator(fn: Callable):
        @wraps(fn)
        async def wrapper(*args, **kwargs):
            gd = get_graceful_degradation()

            async def primary():
                return await fn(*args, **kwargs)

            async def fallback():
                if fallback_fn:
                    return await fallback_fn(*args, **kwargs)
                return None

            return await gd.execute_with_fallback(
                primary,
                fallback if fallback_fn else None,
                timeout_secs,
                max_retries,
                fn.__name__
            )

        return wrapper
    return decorator
```

Step 2: Implement Expert Validation Fallback

File: `tools/workflow/expert_analysis.py`
Location: Add fallback logic

Code to Add:

```python
from utils.error_handling import get_graceful_degradation

class ExpertAnalysis:
    async def validate_with_expert_safe(
        self,
        request_id: str,
        content: str,
        context: dict
    ) -> dict:
        """Validate with expert with graceful degradation."""
        gd = get_graceful_degradation()

        async def primary():
            """Primary: Full expert validation."""
            return await self.validate_with_expert(
                request_id, content, context
            )

        async def fallback():
            """Fallback: Skip expert validation."""
            logger.warning(
                f"Expert validation failed for {request_id}, "
                "returning without validation"
            )
            return {
                "validated": False,
                "expert_analysis": None,
                "warning": "Expert validation unavailable",
                "content": content
            }

        try:
            result = await gd.execute_with_fallback(
                primary,
                fallback,
                timeout_secs=90.0,
                max_retries=1,
                operation_name="expert_validation"
            )
            return result

        except Exception as e:
            logger.error(f"Expert validation completely failed: {e}")
            return await fallback()
```

Step 3: Implement Web Search Fallback

File: `tools/simple/base.py`
Location: Add web search fallback

Code to Add:

```python
from utils.error_handling import get_graceful_degradation

class SimpleTool:
    async def execute_with_web_search_safe(
        self,
        request: dict
    ) -> dict:
        """Execute with web search and graceful degradation."""
        gd = get_graceful_degradation()

        async def primary_glm():
            """Primary: GLM native web search."""
            return await self._execute_with_glm_web_search(request)

        async def fallback_kimi():
            """Fallback: Kimi web search."""
            logger.info("GLM web search failed, trying Kimi")
            return await self._execute_with_kimi_web_search(request)

        async def fallback_no_search():
            """Final fallback: No web search."""
            logger.warning("All web search failed, using cached knowledge")
            return await self._execute_without_web_search(request)

        try:
            # Try GLM first
            result = await gd.execute_with_fallback(
                primary_glm,
                fallback_kimi,
                timeout_secs=150.0,
                max_retries=1,
                operation_name="glm_web_search"
            )
            return result

        except Exception as e:
            logger.warning(f"All web search providers failed: {e}")
            return await fallback_no_search()
```

Step 4: Implement Provider Fallback

File: `src/providers/openai_compatible.py`
Location: Add provider fallback

Code to Add:

```python
from utils.error_handling import get_graceful_degradation

class OpenAICompatibleProvider:
    async def generate_with_fallback(
        self,
        prompt: str,
        model: str,
        **kwargs
    ) -> str:
        """Generate with provider fallback."""
        gd = get_graceful_degradation()

        async def primary():
            """Primary: Requested model."""
            return await self.generate(prompt, model, **kwargs)

        async def fallback():
            """Fallback: GLM-4.5-flash (fast, cheap)."""
            logger.info(f"Model {model} failed, falling back to glm-4.5-flash")
            return await self.generate(prompt, "glm-4.5-flash", **kwargs)

        try:
            result = await gd.execute_with_fallback(
                primary,
                fallback,
                timeout_secs=120.0,
                max_retries=2,
                operation_name=f"provider_{model}"
            )
            return result

        except Exception as e:
            logger.error(f"All providers failed: {e}")
            raise
```

Step 5: Add Circuit Breaker Status Endpoint

File: `tools/diagnostics/status.py`
Location: Add circuit breaker status

Code to Add:

```python
from utils.error_handling import get_graceful_degradation

class StatusTool:
    async def get_circuit_breaker_status(self) -> dict:
        """Get circuit breaker status for all operations."""
        gd = get_graceful_degradation()

        operations = [
            "expert_validation",
            "glm_web_search",
            "kimi_web_search",
            "provider_glm-4.6",
            "provider_kimi-k2-0905-preview"
        ]

        status = {}
        for op in operations:
            status[op] = gd.get_circuit_status(op)

        return {
            "circuit_breakers": status,
            "timestamp": time.time()
        }
```

Testing Instructions

Test 1: Test Expert Validation Fallback

```
# Simulate expert validation failure
# Should fall back to no validation
```

Test 2: Test Web Search Fallback

```
# Simulate GLM web search failure
# Should fall back to Kimi web search
```

Test 3: Test Circuit Breaker

```
# Cause 5 consecutive failures
# Circuit breaker should open
# Subsequent calls should use fallback immediately
```

Test 4: Test Circuit Breaker Recovery

```
# Wait 5 minutes after circuit opens
# Circuit breaker should close
# Primary function should be tried again
```

Acceptance Criteria:
- ✅ GracefulDegradation class implemented
- ✅ Expert validation has fallback (skip validation)
- ✅ Web search has fallback (GLM → Kimi → no search)
- ✅ Provider has fallback (requested model → glm-4.5-flash)
- ✅ Circuit breaker opens after 5 failures

- ✅ Circuit breaker recovers after 5 minutes
- ✅ Circuit breaker status endpoint working
- ✅ All fallbacks tested and working
- ✅ No silent failures (all errors logged)

SECTION 5: GLM AND KIMI NATIVE WEB SEARCH INTEGRATION

Overview

Integrate native web search capabilities for both GLM and Kimi providers following their official API protocols. This ensures proper web search functionality without relying on external tools.

5.1 GLM Native Web Search Integration

Priority: P2 - MEDIUM
Impact: Proper web search functionality for GLM provider
Estimated Time: 2 days
Dependencies: None

GLM Web Search Specification

From glm.md documentation:
- GLM supports native web search via tools schema
- Web search tool is hidden from tool registry (internal function only)
- AI Manager (GLM-4.5-Flash) auto-triggers web search when `use_websearch=true`
- Uses `glm_web_search` function internally

Implementation Steps

Step 1: Verify GLM Web Search Tool Schema

File: `src/providers/capabilities.py`
Location: Lines 67-81 (verify existing schema)

Expected Schema:

```
GLM_WEB_SEARCH_TOOL = {
    "type": "function",
    "function": {
        "name": "glm_web_search",
        "description": "Search the web for current information",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "Search query"
                }
            },
            "required": ["query"]
        }
    }
}
```

Step 2: Verify Web Search Auto-Injection

File: `tools/simple/base.py`
Location: Lines 502-508 (verify existing auto-injection)

Expected Code:

```
def build_websearch_provider_kwargs(self, request: dict) -> dict:
    """Build provider kwargs with web search if requested."""
    kwargs = {}

    if request.get("use_websearch", False):
        # Auto-inject web search tool for GLM
        if self.provider_name == "glm":
            kwargs["tools"] = [GLM_WEB_SEARCH_TOOL]
            kwargs["tool_choice"] = "auto"

    return kwargs
```

Step 3: Add Web Search Logging

File: `src/providers/glm_chat.py`
Location: Add logging when web search is triggered

Code to Add:

```
import logging
logger = logging.getLogger(__name__)

class GLMChatProvider:
    async def generate(self, prompt: str, model: str, **kwargs):
        """Generate with web search logging."""

        # Check if web search is enabled
        tools = kwargs.get("tools", [])
        has_web_search = any(
            t.get("function", {}).get("name") == "glm_web_search"
            for t in tools
        )

        if has_web_search:
            logger.info(f"GLM web search enabled for model {model}")

        # Generate response
        response = await self._generate_impl(prompt, model, **kwargs)

        # Log if web search was used
        if has_web_search and response.get("tool_calls"):
            web_search_calls = [
                tc for tc in response["tool_calls"]
                if tc.get("function", {}).get("name") == "glm_web_search"
            ]
            if web_search_calls:
                logger.info(
                    f"GLM web search executed: "
                    f"{len(web_search_calls)} queries"
                )

        return response
```

Step 4: Add Web Search Metrics

File: `utils/metrics.py` (NEW)
Location: Create new file

Code:

```python
"""Metrics tracking for web search and other features."""

import time
from typing import Dict, Any
from collections import defaultdict


class MetricsTracker:
    """Track metrics for web search and other features."""

    def __init__(self):
        self.web_search_calls = defaultdict(int)
        self.web_search_duration = defaultdict(list)
        self.web_search_errors = defaultdict(int)

    def record_web_search(
        self,
        provider: str,
        duration_s: float,
        success: bool
    ):
        """Record web search execution."""
        self.web_search_calls[provider] += 1
        self.web_search_duration[provider].append(duration_s)

        if not success:
            self.web_search_errors[provider] += 1

    def get_web_search_stats(self) -> Dict[str, Any]:
        """Get web search statistics."""
        stats = {}

        for provider in self.web_search_calls:
            calls = self.web_search_calls[provider]
            durations = self.web_search_duration[provider]
            errors = self.web_search_errors[provider]

            stats[provider] = {
                "total_calls": calls,
                "avg_duration_s": sum(durations) / len(durations) if durations else 0,
                "error_rate": errors / calls if calls > 0 else 0,
                "success_rate": (calls - errors) / calls if calls > 0 else 0
            }

        return stats


# Global metrics tracker
_metrics_tracker = MetricsTracker()


def get_metrics_tracker() -> MetricsTracker:
    """Get global metrics tracker instance."""
    return _metrics_tracker
```

Step 5: Create Web Search Test

File: `tests/test_glm_web_search.py` (NEW)
Location: Create new file

Code:

```python
"""Tests for GLM native web search."""

import pytest
import asyncio
from tools.simple.chat import ChatTool


@pytest.mark.asyncio
async def test_glm_web_search_enabled():
    """Test GLM web search is enabled and working."""
    tool = ChatTool()

    result = await tool.execute({
        "prompt": "What are the latest developments in AI?",
        "use_websearch": True,
        "model": "glm-4.6"
    })

    # Verify result
    assert result is not None
    assert "content" in result
    assert len(result["content"]) > 0

    # Check logs for web search activation
    # (implementation depends on logging setup)


@pytest.mark.asyncio
async def test_glm_web_search_disabled():
    """Test GLM without web search."""
    tool = ChatTool()

    result = await tool.execute({
        "prompt": "What is Python?",
        "use_websearch": False,
        "model": "glm-4.6"
    })

    # Verify result
    assert result is not None
    assert "content" in result


@pytest.mark.asyncio
async def test_glm_web_search_metrics():
    """Test web search metrics tracking."""
    from utils.metrics import get_metrics_tracker

    tracker = get_metrics_tracker()
    tool = ChatTool()

    # Execute with web search
    await tool.execute({
        "prompt": "Latest AI news?",
        "use_websearch": True,
        "model": "glm-4.6"
    })

    # Check metrics
    stats = tracker.get_web_search_stats()
    assert "glm" in stats
    assert stats["glm"]["total_calls"] > 0
```

Testing Instructions

Test 1: Verify Web Search Schema

```
from src.providers.capabilities import GLM_WEB_SEARCH_TOOL
print(GLM_WEB_SEARCH_TOOL)
```

Test 2: Test Web Search Execution

```
python3 -c "
from tools.simple.chat import ChatTool
import asyncio

async def test():
    tool = ChatTool()
    result = await tool.execute({
        'prompt': 'What are the latest AI developments?',
        'use_websearch': True,
        'model': 'glm-4.6'
    })
    print(result)

asyncio.run(test())
"
```

Test 3: Check Web Search Logs

```
grep "GLM web search" .logs/toolcalls.jsonl
```

Test 4: Check Web Search Metrics

```
from utils.metrics import get_metrics_tracker
tracker = get_metrics_tracker()
print(tracker.get_web_search_stats())
```

Acceptance Criteria:
- ✅ GLM web search tool schema verified
- ✅ Web search auto-injection working
- ✅ Web search logging implemented
- ✅ Web search metrics tracking implemented
- ✅ Web search tests passing
- ✅ Web search verified working end-to-end

5.2 Kimi Native Web Search Integration

Priority: P2 - MEDIUM
Impact: Proper web search functionality for Kimi provider
Estimated Time: 2 days
Dependencies: None

Kimi Web Search Specification

From kimi.md documentation:
- Kimi uses `$web_search` builtin function (Moonshot API format)
- OpenAI-compatible structure
- Server-side execution (no client-side search needed)
- Auto tool choice

Implementation Steps

Step 1: Verify Kimi Web Search Tool Schema

File: `src/providers/capabilities.py`
Location: Lines 45-57 (verify existing schema)

Expected Schema:

```
KIMI_WEB_SEARCH_TOOL = {
    "type": "builtin_function",
    "function": {
        "name": "$web_search",
        "description": "Search the web for current information"
    }
}
```

Step 2: Verify Kimi Web Search Configuration

File: `.env`
Location: Verify environment variables

Expected Variables:

```
KIMI_ENABLE_INTERNET_SEARCH=true
KIMI_WEBSEARCH_SCHEMA=function
```

Step 3: Implement Kimi Web Search Auto-Injection

File: `tools/simple/base.py`
Location: Update build_websearch_provider_kwargs

Code to Add:

```
def build_websearch_provider_kwargs(self, request: dict) -> dict:
    """Build provider kwargs with web search if requested."""
    kwargs = {}

    if request.get("use_websearch", False):
        # Auto-inject web search tool
        if self.provider_name == "glm":
            kwargs["tools"] = [GLM_WEB_SEARCH_TOOL]
            kwargs["tool_choice"] = "auto"
        elif self.provider_name == "kimi":
            kwargs["tools"] = [KIMI_WEB_SEARCH_TOOL]
            kwargs["tool_choice"] = "auto"

    return kwargs
```

Step 4: Add Kimi Web Search Logging

File: `src/providers/kimi_chat.py`
Location: Add logging when web search is triggered

Code to Add:

```
import logging
logger = logging.getLogger(__name__)

class KimiChatProvider:
    async def generate(self, prompt: str, model: str, **kwargs):
        """Generate with web search logging."""

        # Check if web search is enabled
        tools = kwargs.get("tools", [])
        has_web_search = any(
            t.get("function", {}).get("name") == "$web_search"
            for t in tools
        )

        if has_web_search:
            logger.info(f"Kimi web search enabled for model {model}")

        # Generate response
        response = await self._generate_impl(prompt, model, **kwargs)

        # Log if web search was used
        if has_web_search and response.get("tool_calls"):
            web_search_calls = [
                tc for tc in response["tool_calls"]
                if tc.get("function", {}).get("name") == "$web_search"
            ]
            if web_search_calls:
                logger.info(
                    f"Kimi web search executed: "
                    f"{len(web_search_calls)} queries"
                )

        return response
```

Step 5: Create Kimi Web Search Test

File: `tests/test_kimi_web_search.py` (NEW)
Location: Create new file

Code:

```python
"""Tests for Kimi native web search."""

import pytest
import asyncio
from tools.simple.chat import ChatTool


@pytest.mark.asyncio
async def test_kimi_web_search_enabled():
    """Test Kimi web search is enabled and working."""
    tool = ChatTool()

    result = await tool.execute({
        "prompt": "What are the latest developments in AI?",
        "use_websearch": True,
        "model": "kimi-k2-0905-preview"
    })

    # Verify result
    assert result is not None
    assert "content" in result
    assert len(result["content"]) > 0


@pytest.mark.asyncio
async def test_kimi_web_search_disabled():
    """Test Kimi without web search."""
    tool = ChatTool()

    result = await tool.execute({
        "prompt": "What is Python?",
        "use_websearch": False,
        "model": "kimi-k2-0905-preview"
    })

    # Verify result
    assert result is not None
    assert "content" in result


@pytest.mark.asyncio
async def test_kimi_web_search_metrics():
    """Test web search metrics tracking."""
    from utils.metrics import get_metrics_tracker

    tracker = get_metrics_tracker()
    tool = ChatTool()

    # Execute with web search
    await tool.execute({
        "prompt": "Latest AI news?",
        "use_websearch": True,
        "model": "kimi-k2-0905-preview"
    })

    # Check metrics
    stats = tracker.get_web_search_stats()
    assert "kimi" in stats
    assert stats["kimi"]["total_calls"] > 0
```

Step 6: Document Web Search Integration

File: `docs/features/web-search.md` (NEW)
Location: Create new file

Code:

```
# Web Search Integration

## Overview

EX-AI MCP Server supports native web search for both GLM and Kimi providers.

## GLM Web Search

**Provider:** ZhipuAI/Z.ai
**Method:** Native web search via tools schema
**Function:** `glm_web_search` (hidden from tool registry)

### Usage

```python
result = await chat_tool.execute({
    "prompt": "What are the latest AI developments?",
    "use_websearch": True,
    "model": "glm-4.6"
})
```
```

Configuration

```
GLM_ENABLE_WEB_SEARCH=true
```

How It Works

1. User sets `use_websearch=true` in request
2. System auto-injects `glm_web_search` tool into tools array
3. GLM AI Manager decides when to trigger web search
4. Web search results integrated into response

Kimi Web Search

Provider: Moonshot
Method: Builtin `$web_search` function
Function: `$web_search` (OpenAI-compatible)

Usage

```
result = await chat_tool.execute({
    "prompt": "What are the latest AI developments?",
    "use_websearch": True,
    "model": "kimi-k2-0905-preview"
})
```

Configuration

```
KIMI_ENABLE_INTERNET_SEARCH=true
KIMI_WEBSEARCH_SCHEMA=function
```

How It Works

1. User sets `use_websearch=true` in request
2. System auto-injects `$web_search` builtin function
3. Kimi decides when to trigger web search
4. Web search executed server-side by Moonshot
5. Results integrated into response

Fallback Strategy

If web search fails, system gracefully degrades:

1. **Primary:** GLM web search
2. **Fallback:** Kimi web search
3. **Final:** No web search (cached knowledge)

Metrics

Web search metrics tracked:
- Total calls per provider
- Average duration
- Error rate
- Success rate

View metrics:

```
from utils.metrics import get_metrics_tracker
tracker = get_metrics_tracker()
print(tracker.get_web_search_stats())
```

Logging

Web search execution logged:
- When web search is enabled
- When web search is executed
- Number of queries
- Duration

Check logs:

```
grep "web search" .logs/toolcalls.jsonl
```

Testing

Run web search tests:

```
pytest tests/test_glm_web_search.py
pytest tests/test_kimi_web_search.py
```

Troubleshooting

Web search not working:
1. Check environment variables are set
2. Check logs for web search activation
3. Check metrics for error rate
4. Verify API keys are valid

Web search slow:
1. Check timeout configuration
2. Check network connectivity
3. Consider using fallback provider

Web search errors:
1. Check circuit breaker status
2. Check provider API status
3. Review error logs

```
#### Testing Instructions

**Test 1: Verify Kimi Web Search Schema**
```python
from src.providers.capabilities import KIMI_WEB_SEARCH_TOOL
print(KIMI_WEB_SEARCH_TOOL)
```

Test 2: Test Kimi Web Search Execution

```
python3 -c "
from tools.simple.chat import ChatTool
import asyncio

async def test():
    tool = ChatTool()
    result = await tool.execute({
        'prompt': 'What are the latest AI developments?',
        'use_websearch': True,
        'model': 'kimi-k2-0905-preview'
    })
    print(result)

asyncio.run(test())
"
```

Test 3: Check Kimi Web Search Logs
```

```
grep "Kimi web search" .logs/toolcalls.jsonl
```

Test 4: Compare GLM vs Kimi Web Search

```
# Run both and compare results
pytest tests/test_glm_web_search.py tests/test_kimi_web_search.py -v
```

Acceptance Criteria:
- ✅ Kimi web search tool schema verified
- ✅ Kimi web search auto-injection working
- ✅ Kimi web search logging implemented
- ✅ Kimi web search metrics tracking implemented
- ✅ Kimi web search tests passing
- ✅ Kimi web search verified working end-to-end
- ✅ Web search documentation complete
- ✅ Both GLM and Kimi web search working

SECTION 6: MEDIUM PRIORITY ENHANCEMENTS (Week 3 - P2 Issues)

Enhancement #1: Simplify Continuation ID System

Priority: P2 - MEDIUM
Impact: Confusing output format, verbose responses
Estimated Time: 2 days
Dependencies: None

Current Problem

Simple tools return continuation_id structure even for single-turn operations:

```
{
  "status": "continuation_available",
  "content": "Chat tool working...",
  "continuation_offer": {
    "continuation_id": "62d15167-479e-4f32-9464-88c7db08b734",
    "note": "You can continue this conversation for 19 more exchanges.",
    "remaining_turns": 19
  }
}
```

Issues:
1. Forced for all responses (not optional)
2. Metadata clutters content
3. Continuation offer appears when not requested
4. Format is verbose

Implementation Steps

Step 1: Make Continuation ID Optional

File: `tools/simple/base.py`
Location: Update response formatting

Before:

```
def format_response(self, content: str, metadata: dict) -> dict:
    """Format response with continuation_id."""
    return {
        "status": "continuation_available",
        "content": content,
        "metadata": metadata,
        "continuation_offer": self._create_continuation_offer()
    }
```

After:

```
def format_response(
    self,
    content: str,
    metadata: dict,
    include_continuation: bool = False
) -> dict:
    """Format response with optional continuation_id."""
    response = {
        "content": content
    }

    # Only include continuation if requested
    if include_continuation:
        response["continuation_id"] = self._get_or_create_continuation_id()
        response["continuation_info"] = {
            "remaining_turns": self._get_remaining_turns(),
            "expires_at": self._get_expiration_time()
        }

    # Include metadata separately (not in content)
    if metadata:
        response["_metadata"] = metadata

    return response
```

Step 2: Add Continuation Mode Parameter

File: `tools/simple/base.py`
Location: Update execute method

Code to Add:

```python
async def execute(self, request: dict) -> dict:
    """Execute tool with optional continuation mode."""
    # Check if continuation mode requested
    continuation_mode = request.get("continuation_mode", False)
    continuation_id = request.get("continuation_id")

    # Execute tool
    result = await self._execute_impl(request)

    # Format response
    response = self.format_response(
        content=result["content"],
        metadata=result.get("metadata"),
        include_continuation=continuation_mode or continuation_id is not None
    )

    return response
```

Step 3: Simplify Continuation Offer

File: `tools/simple/mixins/continuation_mixin.py`
Location: Update continuation offer format

Before:

```python
def _create_continuation_offer(self) -> dict:
    """Create continuation offer."""
    return {
        "continuation_id": str(uuid.uuid4()),
        "note": "You can continue this conversation for 19 more exchanges.",
        "remaining_turns": 19
    }
```

After:

```python
def _get_continuation_info(self) -> dict:
    """Get continuation information (simplified)."""
    return {
        "remaining_turns": self._get_remaining_turns(),
        "expires_in_secs": self._get_time_until_expiration()
    }
```

Step 4: Update Documentation

File: `docs/features/continuation.md` (NEW)
Location: Create new file

Code:

```
# Continuation System

## Overview

The continuation system allows multi-turn conversations with context preservation.

## Usage

### Single-Turn (Default)

```python
result = await chat_tool.execute({
    "prompt": "What is Python?"
})

# Response:
{
    "content": "Python is a programming language..."
}
```
```

Multi-Turn (Continuation Mode)

```
# First turn
result1 = await chat_tool.execute({
    "prompt": "What is Python?",
    "continuation_mode": True
})

# Response:
{
    "content": "Python is a programming language...",
    "continuation_id": "62d15167-479e-4f32-9464-88c7db08b734",
    "continuation_info": {
        "remaining_turns": 19,
        "expires_in_secs": 10800
    }
}

# Second turn
result2 = await chat_tool.execute({
    "prompt": "Tell me more about its features",
    "continuation_id": "62d15167-479e-4f32-9464-88c7db08b734"
})
```

Configuration

```
# Continuation expiration (default: 3 hours)
CONTINUATION_EXPIRATION_SECS=10800

# Maximum turns per continuation (default: 20)
CONTINUATION_MAX_TURNS=20
```

Best Practices

1. **Use continuation mode only when needed** - Don’t enable for single-turn operations

2. **Store continuation_id** - Save it for subsequent turns
3. **Check expiration** - Continuations expire after 3 hours
4. **Handle expiration gracefully** - Start new conversation if expired

Troubleshooting

Continuation expired:

```
{
    "error": "Conversation thread 'ctx-ce818efc' was not found or has expired",
    "suggestion": "Start a new conversation"
}
```

Solution: Start new conversation without continuation_id.

```
#### Testing Instructions

**Test 1: Single-Turn Without Continuation**
```python
result = await chat_tool.execute({"prompt": "Hello"})
assert "continuation_id" not in result
assert "content" in result
```
```

Test 2: Multi-Turn With Continuation

```
result1 = await chat_tool.execute({
    "prompt": "Hello",
    "continuation_mode": True
})
assert "continuation_id" in result1

result2 = await chat_tool.execute({
    "prompt": "Continue",
    "continuation_id": result1["continuation_id"]
})
assert "content" in result2
```

Test 3: Verify Simplified Format

```
result = await chat_tool.execute({
    "prompt": "Hello",
    "continuation_mode": True
})
assert "continuation_info" in result
assert "remaining_turns" in result["continuation_info"]
assert "expires_in_secs" in result["continuation_info"]
assert "note" not in result  # Verbose note removed
```

Acceptance Criteria:
- ✅ Continuation ID optional (not forced)
- ✅ Single-turn responses don't include continuation
- ✅ Multi-turn responses include continuation when requested
- ✅ Continuation format simplified

- ✅ Metadata moved to separate field
- ✅ Documentation updated
- ✅ All tests passing
- ✅ Backward compatibility maintained

Enhancement #2: Update Documentation

Priority: P2 - MEDIUM
Impact: Documentation references non-existent branches
Estimated Time: 1 day
Dependencies: None

Files to Update

1. BRANCH_COMPARISON_wave1-to-auggie-optimization.md
- Update to reference correct branch names
- Clarify which branch is the "working" baseline
- Document actual functionality differences

2. README.md
- Update installation instructions
- Update configuration examples
- Update timeout values
- Add troubleshooting section

3. docs/architecture/
- Update architecture diagrams
- Document timeout hierarchy
- Document progress heartbeat system
- Document logging infrastructure

4. docs/configuration/
- Document all environment variables
- Document MCP configurations
- Document client-specific differences

5. docs/features/
- Document web search integration
- Document continuation system
- Document graceful degradation
- Document circuit breakers

Implementation Steps

Step 1: Update Branch References

File: `BRANCH_COMPARISON_wave1-to-auggie-optimization.md`

Changes:
- Replace "wave1" with "docs/wave1-complete-audit"
- Add note about branch naming
- Document what worked in baseline

Step 2: Update README.md

File: `README.md`

Add sections:
- Quick Start
- Configuration
- Timeout Hierarchy
- Troubleshooting
- Contributing

Step 3: Create Architecture Documentation

File: `docs/architecture/timeout-hierarchy.md` (NEW)

Content:
- Diagram of timeout hierarchy
- Explanation of coordination
- Configuration examples
- Troubleshooting

Step 4: Create Configuration Guide

File: `docs/configuration/environment-variables.md` (NEW)

Content:
- Complete list of environment variables
- Default values
- Recommended values
- Client-specific overrides

Step 5: Create Troubleshooting Guide

File: `docs/troubleshooting.md` (NEW)

Content:
- Common issues and solutions
- Timeout issues
- Logging issues
- Web search issues
- Expert validation issues

Acceptance Criteria

- ✅ All branch references corrected
- ✅ README.md updated and accurate
- ✅ Architecture documentation complete
- ✅ Configuration documentation complete
- ✅ Troubleshooting guide complete
- ✅ All documentation reviewed and accurate
- ✅ No references to non-existent branches
- ✅ All examples tested and working

SECTION 7: WEBSOCKET DAEMON STABILITY

7.1 Connection Handling Improvements

Priority: P2 - MEDIUM
Impact: Better stability and error recovery
Estimated Time: 2 days

Implementation Steps

Step 1: Add Connection Health Checks

File: `src/daemon/ws_server.py`

Code to Add:

```
class WebSocketServer:
    async def health_check_loop(self):
        """Periodic health check for all connections."""
        while True:
            await asyncio.sleep(30)  # Check every 30 seconds

            for session_id, session in list(self.sessions.items()):
                try:
                    # Send ping
                    await session["websocket"].ping()
                except Exception as e:
                    logger.warning(f"Session {session_id} unhealthy: {e}")
                    await self.cleanup_session(session_id)
```

Step 2: Add Automatic Reconnection

File: `scripts/run_ws_shim.py`

Code to Add:

```
class WebSocketClient:
    async def connect_with_retry(self, max_retries: int = 3):
        """Connect with automatic retry."""
        for attempt in range(max_retries):
            try:
                await self.connect()
                return
            except Exception as e:
                if attempt < max_retries - 1:
                    wait_time = 2 ** attempt
                    logger.warning(
                        f"Connection failed (attempt {attempt + 1}), "
                        f"retrying in {wait_time}s..."
                    )
                    await asyncio.sleep(wait_time)
                else:
                    raise
```

Step 3: Add Connection Pooling

File: `src/daemon/ws_server.py`

Code to Add:

```python
class ConnectionPool:
    """Manage WebSocket connection pool."""

    def __init__(self, max_connections: int = 100):
        self.max_connections = max_connections
        self.active_connections = {}

    async def acquire(self, session_id: str, websocket):
        """Acquire connection from pool."""
        if len(self.active_connections) >= self.max_connections:
            raise Exception("Connection pool exhausted")

        self.active_connections[session_id] = {
            "websocket": websocket,
            "acquired_at": time.time()
        }

    async def release(self, session_id: str):
        """Release connection back to pool."""
        if session_id in self.active_connections:
            del self.active_connections[session_id]
```

7.2 Client-Specific Configuration Optimization

Priority: P2 - MEDIUM (VSCode Augment Priority)
Impact: Better performance for VSCode Augment extension
Estimated Time: 1 day

VSCode Augment Optimizations

File: `Daemon/mcp-config.augmentcode.json`

Optimizations:
1. Higher concurrency for IDE responsiveness
2. Strict session management for clean state
3. Faster timeouts for interactive use
4. Better error messages for IDE integration

Already implemented in Fix #5 (Standardize Timeout Configurations)

7.3 Error Recovery Mechanisms

Priority: P2 - MEDIUM
Impact: Better stability and reliability
Estimated Time: 1 day

Implementation Steps

Step 1: Add Error Recovery

File: `src/daemon/ws_server.py`

Code to Add:

```python
class WebSocketServer:
    async def handle_error_with_recovery(
        self,
        session_id: str,
        error: Exception
    ):
        """Handle error with recovery attempt."""
        logger.error(f"Error in session {session_id}: {error}")

        # Try to recover
        try:
            await self.recover_session(session_id)
            logger.info(f"Session {session_id} recovered")
        except Exception as recovery_error:
            logger.error(
                f"Recovery failed for session {session_id}: {recovery_error}"
            )
            await self.cleanup_session(session_id)

    async def recover_session(self, session_id: str):
        """Attempt to recover session."""
        if session_id in self.sessions:
            session = self.sessions[session_id]

            # Reset session state
            session["in_flight"] = 0
            session["last_activity"] = time.time()

            # Send recovery message
            await session["websocket"].send_json({
                "type": "recovery",
                "message": "Session recovered"
            })
```

7.4 Health Check Implementation

Priority: P2 - MEDIUM
Impact: Better monitoring and diagnostics
Estimated Time: 1 day

Implementation Steps

Step 1: Add Health Check Endpoint

File: `src/daemon/ws_server.py`

Code to Add:

```python
class WebSocketServer:
    async def handle_health_check(self, websocket):
        """Handle health check request."""
        health_status = {
            "status": "healthy",
            "timestamp": time.time(),
            "active_sessions": len(self.sessions),
            "total_requests": self.total_requests,
            "uptime_secs": time.time() - self.start_time,
            "circuit_breakers": self._get_circuit_breaker_status()
        }

        await websocket.send_json(health_status)

    def _get_circuit_breaker_status(self) -> dict:
        """Get circuit breaker status."""
        from utils.error_handling import get_graceful_degradation
        gd = get_graceful_degradation()

        operations = [
            "expert_validation",
            "glm_web_search",
            "kimi_web_search"
        ]

        status = {}
        for op in operations:
            status[op] = gd.get_circuit_status(op)

        return status
```

Step 2: Add Health Check Tool

File: `tools/diagnostics/health.py` (NEW)

Code:

```python
"""Health check tool for system diagnostics."""

import time
from tools.simple.base import SimpleTool


class HealthCheckTool(SimpleTool):
    """Tool for checking system health."""

    name = "health_check"
    description = "Check system health and status"

    async def execute(self, request: dict) -> dict:
        """Execute health check."""
        from utils.error_handling import get_graceful_degradation
        from utils.metrics import get_metrics_tracker

        gd = get_graceful_degradation()
        metrics = get_metrics_tracker()

        # Get circuit breaker status
        circuit_breakers = {}
        for op in ["expert_validation", "glm_web_search", "kimi_web_search"]:
            circuit_breakers[op] = gd.get_circuit_status(op)

        # Get web search metrics
        web_search_stats = metrics.get_web_search_stats()

        # Build health report
        health_report = {
            "status": "healthy",
            "timestamp": time.time(),
            "circuit_breakers": circuit_breakers,
            "web_search_stats": web_search_stats,
            "recommendations": self._get_recommendations(
                circuit_breakers,
                web_search_stats
            )
        }

        return health_report

    def _get_recommendations(
        self,
        circuit_breakers: dict,
        web_search_stats: dict
    ) -> list:
        """Get health recommendations."""
        recommendations = []

        # Check circuit breakers
        for op, status in circuit_breakers.items():
            if status["status"] == "open":
                recommendations.append(
                    f"Circuit breaker open for {op}. "
                    f"Will recover in {status['time_until_recovery']:.0f}s."
                )

        # Check web search error rates
        for provider, stats in web_search_stats.items():
            if stats["error_rate"] > 0.1:  # >10% error rate
                recommendations.append(
```

```
                    f"High error rate for {provider} web search: "
                    f"{stats['error_rate']:.1%}. Check API status."
                )

        if not recommendations:
            recommendations.append("All systems operating normally.")

        return recommendations
```

Acceptance Criteria

- ✅ Connection health checks implemented
- ✅ Automatic reconnection working
- ✅ Connection pooling implemented
- ✅ Error recovery mechanisms working
- ✅ Health check endpoint implemented
- ✅ Health check tool working
- ✅ VSCode Augment optimizations verified
- ✅ All stability improvements tested

SECTION 8: TOOL EXECUTION FLOW OPTIMIZATION

8.1 Standardize Execution Paths

Priority: P2 - MEDIUM
Impact: Consistent behavior across all tools
Estimated Time: 2 days

Current Problem

Simple tools and workflow tools use different execution paths:
- Simple tools: `tools/simple/base.py`
- Workflow tools: `tools/workflow/base.py`
- Different logging, timeout handling, error handling

Implementation Steps

Step 1: Create Base Tool Interface

File: `tools/base_tool_interface.py` (NEW)

Code:

```python
"""Base tool interface for all tools."""

from abc import ABC, abstractmethod
from typing import Dict, Any
import time
import uuid
import logging

logger = logging.getLogger(__name__)


class BaseToolInterface(ABC):
    """Base interface for all tools."""

    def __init__(self):
        self.name = self.__class__.__name__
        self.logger = None
        self.heartbeat = None

    @abstractmethod
    async def _execute_impl(self, request: Dict[str, Any]) -> Dict[str, Any]:
        """Implement tool-specific execution logic."""
        pass

    async def execute(self, request: Dict[str, Any]) -> Dict[str, Any]:
        """Execute tool with standard infrastructure."""
        from utils.logging_unified import get_unified_logger
        from utils.progress import ProgressHeartbeat

        # Setup
        self.logger = get_unified_logger()
        request_id = request.get("request_id") or str(uuid.uuid4())
        start_time = time.time()

        # Log start
        self.logger.log_tool_start(
            tool_name=self.name,
            request_id=request_id,
            params=request
        )

        # Create heartbeat
        async with ProgressHeartbeat(interval_secs=6.0) as heartbeat:
            self.heartbeat = heartbeat

            try:
                # Execute tool
                result = await self._execute_impl(request)

                # Log completion
                duration = time.time() - start_time
                self.logger.log_tool_complete(
                    tool_name=self.name,
                    request_id=request_id,
                    duration_s=duration,
                    result_preview=str(result)
                )

                return result

            except Exception as e:
                # Log error
```

```
                duration = time.time() - start_time
                self.logger.log_tool_error(
                    tool_name=self.name,
                    request_id=request_id,
                    error=str(e),
                    error_traceback=traceback.format_exc(),
                    metadata={"duration_s": duration}
                )
                raise
```

Step 2: Update Simple Tools to Use Interface

File: `tools/simple/base.py`

Changes:
- Inherit from BaseToolInterface
- Move execution logic to _execute_impl
- Remove duplicate logging/heartbeat code

Step 3: Update Workflow Tools to Use Interface

File: `tools/workflow/base.py`

Changes:
- Inherit from BaseToolInterface
- Move execution logic to _execute_impl
- Remove duplicate logging/heartbeat code

8.2 Improve Tool Response Formatting

Priority: P2 - MEDIUM
Impact: Consistent response format across all tools
Estimated Time: 1 day

Implementation Steps

Step 1: Create Standard Response Format

File: `tools/response_format.py` (NEW)

Code:

```python
"""Standard response format for all tools."""

from typing import Dict, Any, Optional
from datetime import datetime


class ToolResponse:
    """Standard tool response format."""

    def __init__(
        self,
        content: Any,
        metadata: Optional[Dict[str, Any]] = None,
        error: Optional[str] = None
    ):
        self.content = content
        self.metadata = metadata or {}
        self.error = error
        self.timestamp = datetime.utcnow().isoformat()

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary."""
        response = {
            "content": self.content,
            "timestamp": self.timestamp
        }

        if self.metadata:
            response["_metadata"] = self.metadata

        if self.error:
            response["error"] = self.error

        return response

    @classmethod
    def success(
        cls,
        content: Any,
        metadata: Optional[Dict[str, Any]] = None
    ) -> "ToolResponse":
        """Create success response."""
        return cls(content=content, metadata=metadata)

    @classmethod
    def error(
        cls,
        error: str,
        metadata: Optional[Dict[str, Any]] = None
    ) -> "ToolResponse":
        """Create error response."""
        return cls(content=None, metadata=metadata, error=error)
```

Step 2: Update All Tools to Use Standard Format

Update all tools to return ToolResponse objects.

8.3 Enhance Tool Metadata and Status Reporting

Priority: P2 - MEDIUM
Impact: Better visibility into tool execution
Estimated Time: 1 day

Implementation Steps

Step 1: Add Tool Metadata

File: `tools/base_tool_interface.py`

Code to Add:

```
class BaseToolInterface(ABC):
    def get_metadata(self) -> Dict[str, Any]:
        """Get tool metadata."""
        return {
            "name": self.name,
            "description": self.description,
            "version": self.version,
            "timeout_secs": self.timeout_secs,
            "supports_streaming": self.supports_streaming,
            "supports_continuation": self.supports_continuation
        }
```

Step 2: Add Status Reporting

File: `tools/diagnostics/tool_status.py` (NEW)

Code:

```
"""Tool status reporting."""

from tools.base_tool_interface import BaseToolInterface


class ToolStatusReporter:
    """Report status of all tools."""

    def __init__(self):
        self.tools = {}

    def register_tool(self, tool: BaseToolInterface):
        """Register tool for status reporting."""
        self.tools[tool.name] = tool

    def get_all_tool_status(self) -> Dict[str, Any]:
        """Get status of all tools."""
        status = {}

        for name, tool in self.tools.items():
            status[name] = {
                "metadata": tool.get_metadata(),
                "health": "healthy",  # Could add health checks
                "last_execution": None  # Could track last execution
            }

        return status
```

Acceptance Criteria

- ✅ BaseToolInterface created
- ✅ All tools use standard execution path
- ✅ Standard response format implemented
- ✅ Tool metadata enhanced
- ✅ Status reporting implemented
- ✅ Consistent behavior across all tools
- ✅ All tests passing

SECTION 9: TESTING AND VALIDATION PROCEDURES

9.1 Unit Tests for Critical Components

Priority: P1 - HIGH
Estimated Time: 3 days

Test Files to Create

1. test_timeout_hierarchy.py

```python
"""Tests for timeout hierarchy."""

import pytest
import asyncio
from config import TimeoutConfig


def test_timeout_hierarchy_validation():
    """Test timeout hierarchy is valid."""
    assert TimeoutConfig.validate_hierarchy()


def test_timeout_values():
    """Test timeout values are correct."""
    assert TimeoutConfig.WORKFLOW_TOOL_TIMEOUT_SECS == 120
    assert TimeoutConfig.get_daemon_timeout() == 180
    assert TimeoutConfig.get_shim_timeout() == 240
    assert TimeoutConfig.get_client_timeout() == 300


@pytest.mark.asyncio
async def test_tool_timeout():
    """Test tool respects timeout."""
    from tools.workflows.thinkdeep import ThinkDeepTool

    tool = ThinkDeepTool()

    # Should timeout at 120s
    with pytest.raises(asyncio.TimeoutError):
        await asyncio.wait_for(
            tool.execute({"prompt": "test"}),
            timeout=125  # Slightly longer than tool timeout
        )
```

2. test_progress_heartbeat.py

```python
"""Tests for progress heartbeat."""

import pytest
import asyncio
from utils.progress import ProgressHeartbeat


@pytest.mark.asyncio
async def test_heartbeat_timing():
    """Test heartbeat sends at correct interval."""
    messages = []

    async def callback(data):
        messages.append(data)

    async with ProgressHeartbeat(interval_secs=2.0, callback=callback) as hb:
        for i in range(10):
            await hb.send_heartbeat(f"Step {i}")
            await asyncio.sleep(1)

    # Should have ~5 messages (10 seconds / 2 second interval)
    assert 4 <= len(messages) <= 6


@pytest.mark.asyncio
async def test_heartbeat_progress_calculation():
    """Test heartbeat calculates progress correctly."""
    messages = []

    async def callback(data):
        messages.append(data)

    async with ProgressHeartbeat(interval_secs=1.0, callback=callback) as hb:
        hb.set_total_steps(5)

        for i in range(1, 6):
            hb.set_current_step(i)
            await hb.force_heartbeat(f"Step {i}")
            await asyncio.sleep(0.5)

    # Check progress calculation
    assert len(messages) == 5
    assert messages[0]["step"] == 1
    assert messages[4]["step"] == 5
```

3. test_logging_unified.py

```python
"""Tests for unified logging."""

import pytest
import json
from pathlib import Path
from utils.logging_unified import UnifiedLogger


def test_unified_logger_creation():
    """Test unified logger creates log file."""
    log_file = ".logs/test_toolcalls.jsonl"
    logger = UnifiedLogger(log_file)

    logger.log_tool_start("test_tool", "req123", {"param": "value"})
    logger.flush()

    # Check log file exists
    assert Path(log_file).exists()

    # Check log entry
    with open(log_file) as f:
        entry = json.loads(f.readline())
        assert entry["event"] == "tool_start"
        assert entry["tool"] == "test_tool"
        assert entry["request_id"] == "req123"


def test_unified_logger_sanitization():
    """Test logger sanitizes sensitive data."""
    logger = UnifiedLogger(".logs/test_toolcalls.jsonl")

    params = {
        "api_key": "secret123",
        "normal_param": "value"
    }

    sanitized = logger._sanitize_params(params)

    assert sanitized["api_key"] == "***REDACTED***"
    assert sanitized["normal_param"] == "value"
```

4. test_graceful_degradation.py

```python
"""Tests for graceful degradation."""

import pytest
import asyncio
from utils.error_handling import GracefulDegradation, CircuitBreakerOpen


@pytest.mark.asyncio
async def test_fallback_on_failure():
    """Test fallback is used when primary fails."""
    gd = GracefulDegradation()

    async def primary():
        raise Exception("Primary failed")

    async def fallback():
        return "fallback_result"

    result = await gd.execute_with_fallback(
        primary,
        fallback,
        timeout_secs=5.0,
        max_retries=0
    )

    assert result == "fallback_result"


@pytest.mark.asyncio
async def test_circuit_breaker_opens():
    """Test circuit breaker opens after failures."""
    gd = GracefulDegradation()

    async def failing_fn():
        raise Exception("Always fails")

    # Cause 5 failures
    for i in range(5):
        try:
            await gd.execute_with_fallback(
                failing_fn,
                None,
                timeout_secs=1.0,
                max_retries=0,
                operation_name="test_op"
            )
        except:
            pass

    # Circuit breaker should be open
    status = gd.get_circuit_status("test_op")
    assert status["status"] == "open"

    # Next call should raise CircuitBreakerOpen
    with pytest.raises(CircuitBreakerOpen):
        await gd.execute_with_fallback(
            failing_fn,
            None,
            timeout_secs=1.0,
            max_retries=0,
            operation_name="test_op"
        )
```

9.2 Integration Tests for Tool Execution

Priority: P1 - HIGH
Estimated Time: 3 days

Test Files to Create

1. test_simple_tools_integration.py

```python
"""Integration tests for simple tools."""

import pytest
from tools.simple.chat import ChatTool


@pytest.mark.asyncio
async def test_chat_tool_execution():
    """Test chat tool executes correctly."""
    tool = ChatTool()

    result = await tool.execute({
        "prompt": "What is Python?",
        "model": "glm-4.6"
    })

    assert result is not None
    assert "content" in result
    assert len(result["content"]) > 0


@pytest.mark.asyncio
async def test_chat_tool_with_web_search():
    """Test chat tool with web search."""
    tool = ChatTool()

    result = await tool.execute({
        "prompt": "What are the latest AI developments?",
        "use_websearch": True,
        "model": "glm-4.6"
    })

    assert result is not None
    assert "content" in result
```

2. test_workflow_tools_integration.py

```python
"""Integration tests for workflow tools."""

import pytest
from tools.workflows.debug import DebugTool


@pytest.mark.asyncio
async def test_debug_tool_execution():
    """Test debug tool executes correctly."""
    tool = DebugTool()

    result = await tool.execute({
        "prompt": "Debug this code: def add(a, b): return a + b",
        "max_steps": 2
    })

    assert result is not None
    assert "results" in result
    assert len(result["results"]) == 2


@pytest.mark.asyncio
async def test_debug_tool_with_expert_validation():
    """Test debug tool with expert validation."""
    tool = DebugTool()

    result = await tool.execute({
        "prompt": "Debug this code",
        "max_steps": 2,
        "use_expert_validation": True
    })

    assert result is not None
    # Check expert validation was performed
    # (implementation depends on response format)
```

9.3 End-to-End Tests for Each Client

Priority: P1 - HIGH
Estimated Time: 2 days

Test Scenarios

1. VSCode Augment Extension
- Test tool execution from VSCode
- Test timeout behavior
- Test progress updates
- Test error handling

2. Auggie CLI
- Test autonomous operation
- Test long-running sessions
- Test continuation system
- Test expert validation

3. Claude Desktop
- Test desktop app integration
- Test conversation flow

- Test web search
- Test file operations

9.4 Performance Benchmarks

Priority: P2 - MEDIUM
Estimated Time: 2 days

Benchmarks to Create

1. Tool Execution Time

```
"""Benchmark tool execution time."""

import time
import asyncio
from tools.simple.chat import ChatTool


async def benchmark_chat_tool():
    """Benchmark chat tool execution."""
    tool = ChatTool()

    # Warm up
    await tool.execute({"prompt": "Hello"})

    # Benchmark
    times = []
    for i in range(10):
        start = time.time()
        await tool.execute({"prompt": f"Test {i}"})
        times.append(time.time() - start)

    avg_time = sum(times) / len(times)
    print(f"Average execution time: {avg_time:.2f}s")

    assert avg_time < 30  # Should complete in <30s


asyncio.run(benchmark_chat_tool())
```

2. Concurrent Tool Execution

```python
"""Benchmark concurrent tool execution."""

import asyncio
from tools.simple.chat import ChatTool


async def benchmark_concurrent_execution():
    """Benchmark concurrent tool execution."""
    tool = ChatTool()

    # Execute 10 tools concurrently
    tasks = [
        tool.execute({"prompt": f"Test {i}"})
        for i in range(10)
    ]

    start = time.time()
    results = await asyncio.gather(*tasks)
    duration = time.time() - start

    print(f"10 concurrent executions: {duration:.2f}s")

    assert duration < 60  # Should complete in <60s
    assert len(results) == 10


asyncio.run(benchmark_concurrent_execution())
```

9.5 Regression Test Suite

Priority: P1 - HIGH
Estimated Time: 2 days

Regression Tests

1. Test All Fixes Don’t Break Existing Functionality

```python
"""Regression tests for all fixes."""

import pytest


@pytest.mark.regression
class TestTimeoutHierarchyRegression:
    """Regression tests for timeout hierarchy fix."""

    def test_simple_tools_still_work(self):
        """Test simple tools still work after timeout fix."""
        pass

    def test_workflow_tools_still_work(self):
        """Test workflow tools still work after timeout fix."""
        pass


@pytest.mark.regression
class TestLoggingRegression:
    """Regression tests for logging fix."""

    def test_simple_tool_logging_still_works(self):
        """Test simple tool logging still works."""
        pass

    def test_workflow_tool_logging_now_works(self):
        """Test workflow tool logging now works."""
        pass
```

2. Test Backward Compatibility

```python
"""Backward compatibility tests."""

import pytest


@pytest.mark.compatibility
class TestBackwardCompatibility:
    """Test backward compatibility."""

    def test_old_request_format_still_works(self):
        """Test old request format still works."""
        pass

    def test_old_response_format_still_works(self):
        """Test old response format still works."""
        pass
```

Acceptance Criteria

- ✅ All unit tests passing
- ✅ All integration tests passing
- ✅ All end-to-end tests passing
- ✅ Performance benchmarks meet targets
- ✅ Regression tests passing
- ✅ No functionality broken by fixes

- ✅ Backward compatibility maintained

SECTION 10: IMPLEMENTATION SEQUENCE

Week 1: Critical Fixes (P0)

Days 1-2: Timeout Hierarchy
- Monday: Implement timeout hierarchy coordination
- Tuesday: Test and validate timeout hierarchy

Days 3-4: Progress Heartbeat
- Wednesday: Implement progress heartbeat system
- Thursday: Integrate heartbeat in all tools

Day 5: Logging Unification
- Friday: Implement unified logging infrastructure

Validation Checkpoint:
- ✅ Workflow tools complete in 60-120s
- ✅ Progress updates every 5-8 seconds
- ✅ All tools logging correctly
- ✅ Timeouts trigger at expected intervals

Week 2: High Priority Fixes (P1)

Days 6-8: Expert Validation
- Monday: Investigate duplicate call bug
- Tuesday: Implement fix
- Wednesday: Test and re-enable expert validation

Days 9-10: Configuration Standardization
- Thursday: Standardize timeout configurations
- Friday: Test all three clients

Validation Checkpoint:
- ✅ Expert validation working correctly
- ✅ Consistent behavior across all clients
- ✅ Graceful degradation implemented
- ✅ Clear error messages

Week 3: Enhancements (P2)

Days 11-12: Web Search Integration
- Monday: Implement GLM web search
- Tuesday: Implement Kimi web search

Days 13-14: System Optimization
- Wednesday: Simplify continuation system
- Thursday: Optimize WebSocket daemon

Day 15: Documentation
- Friday: Update all documentation

Validation Checkpoint:
- ✅ Web search working for both providers
- ✅ Continuation system simplified
- ✅ WebSocket daemon stable
- ✅ Documentation accurate and complete

Dependencies Between Fixes

Critical Path:
1. Timeout Hierarchy (Fix #1) → Must be done first
2. Progress Heartbeat (Fix #2) → Depends on #1
3. Logging Unification (Fix #3) → Independent
4. Expert Validation (Fix #4) → Depends on #3
5. Configuration Standardization (Fix #5) → Depends on #1
6. Graceful Degradation (Fix #6) → Depends on #1, #3

Parallel Work:
- Fixes #1, #3 can be done in parallel
- Fixes #2, #5 depend on #1
- Fixes #4, #6 depend on #3

Checkpoints for Validation

After Week 1:
- Run all workflow tools
- Verify timeout behavior
- Check progress updates
- Review logs

After Week 2:
- Test expert validation
- Test all three clients
- Verify graceful degradation
- Check error handling

After Week 3:
- Test web search
- Test continuation system
- Verify WebSocket stability
- Review documentation

Rollback Procedures

If Issues Arise:

1. **Timeout Hierarchy Issues**
   - Rollback: Revert config.py changes
   - Restore: Previous timeout values
   - Impact: System returns to previous state

2. **Progress Heartbeat Issues**
   - Rollback: Disable heartbeat in tools
   - Restore: Remove heartbeat integration
   - Impact: No progress updates, but tools still work

3. **Logging Issues**
   - Rollback: Revert to old logging
   - Restore: Previous logging code
   - Impact: Logging works as before

4. **Expert Validation Issues**
   - Rollback: Disable expert validation
   - Restore: Set DEFAULT_USE_ASSISTANT_MODEL=false
   - Impact: No expert validation, but tools work

Rollback Strategy:
- Keep previous version in separate branch
- Test rollback procedure before starting
- Document rollback steps for each fix
- Have rollback scripts ready

SECTION 11: CONFIGURATION MANAGEMENT

11.1 Environment Variable Cleanup

Current State: 50+ environment variables scattered across multiple files

Target State: Organized, documented, validated environment variables

Recommended Structure

File: `.env.example`

Content:

```
# ==============================================================================
# EX-AI MCP SERVER CONFIGURATION
# ==============================================================================

# ------------------------------------------------------------------------------
# TIMEOUT CONFIGURATION (Coordinated Hierarchy)
# ------------------------------------------------------------------------------
# Tool timeouts (primary)
SIMPLE_TOOL_TIMEOUT_SECS=60
WORKFLOW_TOOL_TIMEOUT_SECS=120
EXPERT_ANALYSIS_TIMEOUT_SECS=90

# Provider timeouts
GLM_TIMEOUT_SECS=90
KIMI_TIMEOUT_SECS=120
KIMI_WEB_SEARCH_TIMEOUT_SECS=150

# Infrastructure timeouts (auto-calculated, do not set manually)
# DAEMON_TIMEOUT = WORKFLOW_TOOL_TIMEOUT * 1.5 = 180s
# SHIM_TIMEOUT = WORKFLOW_TOOL_TIMEOUT * 2.0 = 240s
# CLIENT_TIMEOUT = WORKFLOW_TOOL_TIMEOUT * 2.5 = 300s

# ------------------------------------------------------------------------------
# PROVIDER CONFIGURATION
# ------------------------------------------------------------------------------
# GLM (ZhipuAI/Z.ai)
GLM_API_KEY=your_glm_api_key_here
GLM_BASE_URL=https://api.z.ai/v1
GLM_DEFAULT_MODEL=glm-4.6
GLM_TEMPERATURE=0.6
GLM_MAX_TOKENS=65536
GLM_STREAM_ENABLED=true
GLM_ENABLE_WEB_SEARCH=true

# Kimi (Moonshot)
KIMI_API_KEY=your_kimi_api_key_here
KIMI_BASE_URL=https://api.moonshot.ai/v1
KIMI_DEFAULT_MODEL=kimi-k2-0905-preview
KIMI_TEMPERATURE=0.5
KIMI_MAX_TOKENS=32768
KIMI_ENABLE_INTERNET_SEARCH=true
KIMI_WEBSEARCH_SCHEMA=function
KIMI_FILES_MAX_SIZE_MB=20

# ------------------------------------------------------------------------------
# CONCURRENCY CONFIGURATION
# ------------------------------------------------------------------------------
EXAI_WS_SESSION_MAX_INFLIGHT=6
EXAI_WS_GLOBAL_MAX_INFLIGHT=16
EXAI_WS_GLM_MAX_INFLIGHT=8
EXAI_WS_KIMI_MAX_INFLIGHT=4

# ------------------------------------------------------------------------------
# SESSION MANAGEMENT
# ------------------------------------------------------------------------------
EX_SESSION_SCOPE_STRICT=false
EX_SESSION_SCOPE_ALLOW_CROSS_SESSION=true
CONTINUATION_EXPIRATION_SECS=10800
CONTINUATION_MAX_TURNS=20

# ------------------------------------------------------------------------------
# FEATURE FLAGS
```

```
# ------------------------------------------------------------------------------
DEFAULT_USE_ASSISTANT_MODEL=true
EX_ALLOW_RELATIVE_PATHS=true

# ------------------------------------------------------------------------------
# LOGGING CONFIGURATION
# ------------------------------------------------------------------------------
LOG_LEVEL=INFO
LOG_FILE=.logs/toolcalls.jsonl
LOG_ROTATION_SIZE_MB=100
LOG_RETENTION_DAYS=30

# ------------------------------------------------------------------------------
# CIRCUIT BREAKER CONFIGURATION
# ------------------------------------------------------------------------------
CIRCUIT_BREAKER_FAILURE_THRESHOLD=5
CIRCUIT_BREAKER_RECOVERY_TIMEOUT_SECS=300

# ------------------------------------------------------------------------------
# METRICS CONFIGURATION
# ------------------------------------------------------------------------------
METRICS_ENABLED=true
METRICS_FILE=.logs/metrics.jsonl
```

11.2 Client-Specific Configuration Templates

Already implemented in Fix #5 (Standardize Timeout Configurations)

11.3 Default Values and Best Practices

Best Practices:

1. **Timeout Values**
   - Simple tools: 60s (sufficient for most operations)
   - Workflow tools: 120s (allows multi-step execution)
   - Expert validation: 90s (within workflow timeout)

2. **Concurrency Limits**
   - Session max: 6-8 (balanced execution)
   - Global max: 16-20 (prevent resource exhaustion)
   - Provider max: 4-8 (respect API limits)

3. **Session Management**
   - Strict mode: false for Auggie (flexibility)
   - Strict mode: true for VSCode/Claude (isolation)
   - Cross-session: true for Auggie (continuity)
   - Cross-session: false for VSCode/Claude (clean state)

4. **Feature Flags**
   - Expert validation: true (key feature)
   - Relative paths: true (user convenience)
   - Web search: true (enhanced capabilities)

11.4 Configuration Validation

File: `utils/config_validation.py` (NEW)

Code:

```python
"""Configuration validation utilities."""

import os
import logging
from typing import Dict, Any, List

logger = logging.getLogger(__name__)


class ConfigValidator:
    """Validate configuration values."""

    def __init__(self):
        self.errors = []
        self.warnings = []

    def validate_all(self) -> bool:
        """Validate all configuration."""
        self.validate_timeouts()
        self.validate_api_keys()
        self.validate_concurrency()
        self.validate_features()

        if self.errors:
            for error in self.errors:
                logger.error(f"Configuration error: {error}")
            return False

        if self.warnings:
            for warning in self.warnings:
                logger.warning(f"Configuration warning: {warning}")

        return True

    def validate_timeouts(self):
        """Validate timeout configuration."""
        from config import TimeoutConfig

        try:
            TimeoutConfig.validate_hierarchy()
        except ValueError as e:
            self.errors.append(f"Invalid timeout hierarchy: {e}")

    def validate_api_keys(self):
        """Validate API keys are set."""
        required_keys = ["GLM_API_KEY", "KIMI_API_KEY"]

        for key in required_keys:
            if not os.getenv(key):
                self.errors.append(f"Missing required API key: {key}")

    def validate_concurrency(self):
        """Validate concurrency limits."""
        session_max = int(os.getenv("EXAI_WS_SESSION_MAX_INFLIGHT", "6"))
        global_max = int(os.getenv("EXAI_WS_GLOBAL_MAX_INFLIGHT", "16"))

        if session_max > global_max:
            self.errors.append(
                f"Session max ({session_max}) cannot exceed "
                f"global max ({global_max})"
            )
```

```python
    def validate_features(self):
        """Validate feature flags."""
        # Add feature flag validation
        pass


def validate_configuration() -> bool:
    """Validate configuration on startup."""
    validator = ConfigValidator()
    return validator.validate_all()
```

Integration:

File: `server.py`

Code to Add:

```python
from utils.config_validation import validate_configuration

# Validate configuration on startup
if not validate_configuration():
    logger.error("Configuration validation failed")
    sys.exit(1)
```

Acceptance Criteria

- ✅ Environment variables organized and documented
- ✅ Client-specific templates created
- ✅ Default values documented
- ✅ Best practices documented
- ✅ Configuration validation implemented
- ✅ Validation runs on startup
- ✅ Clear error messages for invalid configuration

SECTION 12: DOCUMENTATION UPDATES

12.1 Files That Need Documentation Updates

Priority Files:

1. **README.md** - Main project documentation
2. **BRANCH_COMPARISON_wave1-to-auggie-optimization.md** - Branch comparison
3. **docs/architecture/** - Architecture documentation
4. **docs/configuration/** - Configuration guides
5. **docs/features/** - Feature documentation
6. **docs/troubleshooting.md** - Troubleshooting guide

12.2 README Improvements

File: `README.md`

Sections to Add/Update:

1. **Quick Start**
   - Installation steps
   - Basic configuration
   - First tool execution

2. **Configuration**
   - Environment variables
   - Timeout hierarchy
   - Client-specific configs

3. **Architecture**
   - System overview
   - Component diagram
   - Data flow

4. **Features**
   - Tool ecosystem
   - Web search integration
   - Expert validation
   - Continuation system

5. **Troubleshooting**
   - Common issues
   - Timeout problems
   - Logging issues
   - Web search issues

6. **Contributing**
   - Development setup
   - Testing procedures
   - Pull request process

12.3 API Documentation Corrections

Files to Create:

1. **docs/api/tools.md** - Tool API reference
2. **docs/api/providers.md** - Provider API reference
3. **docs/api/websocket.md** - WebSocket protocol
4. **docs/api/mcp.md** - MCP protocol implementation

12.4 Architecture Diagrams

Diagrams to Create:

1. **System Architecture**

```
┌─────────────────────────────────────────────────────────────┐
│                      MCP Client Layer                       │
│  (VSCode Augment, Auggie CLI, Claude Desktop)               │
└─────────────────────────────────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────┐
│              scripts/run_ws_shim.py                         │
│  - Timeout: 240s (2x tool timeout)                          │
│  - MCP protocol translation                                 │
└─────────────────────────────────────────────────────────────┘
                          │ WebSocket (ws://127.0.0.1:8765)
                          ▼
┌─────────────────────────────────────────────────────────────┐
│          src/daemon/ws_server.py                            │
│  - Timeout: 180s (1.5x tool timeout)                        │
│  - Session management                                       │
│  - Progress heartbeat routing                               │
└─────────────────────────────────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────┐
│                      server.py                              │
│  - Tool registry and routing                                │
│  - Model provider integration                               │
│  - Request handling                                         │
└─────────────────────────────────────────────────────────────┘
                          │
          ┌───────────────┴───────────────┐
          ▼                               ▼
┌──────────────────────┐      ┌──────────────────────┐
│  Simple Tools        │      │  Workflow Tools      │
│  (60s timeout)       │      │   (120s timeout)     │
│  - chat              │      │   - analyze          │
│  - listmodels        │      │   - thinkdeep        │
│  ✅ WORKING          │      │   - debug            │
│                      │      │   - codereview       │
│                      │      │   ✅ FIXED           │
└──────────────────────┘      └──────────────────────┘
```

1. **Timeout Hierarchy**

```
Client Timeout (300s)
    │
    ├─ Shim Timeout (240s)
    │   │
    │   ├─ Daemon Timeout (180s)
    │   │   │
    │   │   ├─ Workflow Tool Timeout (120s)
    │   │   │   │
    │   │   │   └─ Expert Validation Timeout (90s)
    │   │   │
    │   │   └─ Simple Tool Timeout (60s)
    │   │
    │   └─ Provider Timeouts
    │       ├─ GLM: 90s
    │       ├─ Kimi: 120s
    │       └─ Kimi Web Search: 150s
    │
    └─ Rule: Each outer timeout = 1.5x inner timeout
```

1. **Progress Heartbeat Flow**

```
Tool Execution Start
    │
    ├─ Create ProgressHeartbeat (6s interval)
    │
    ├─ Execute Step 1
    │   ├─ Send heartbeat (elapsed: 6s)
    │   └─ Complete step
    │
    ├─ Execute Step 2
    │   ├─ Send heartbeat (elapsed: 12s)
    │   └─ Complete step
    │
    ├─ Execute Step 3
    │   ├─ Send heartbeat (elapsed: 18s)
    │   └─ Complete step
    │
    └─ Tool Execution Complete
```

1. **Graceful Degradation Flow**

```
Request
    │
    ├─ Try Primary Function
    │   ├─ Success → Return result
    │   └─ Failure → Record failure
    │
    ├─ Check Circuit Breaker
    │   ├─ Open → Use fallback immediately
    │   └─ Closed → Retry primary
    │
    ├─ Retry Primary (with backoff)
    │   ├─ Success → Return result
    │   └─ Failure → Try fallback
    │
    ├─ Try Fallback Function
    │   ├─ Success → Return result
    │   └─ Failure → Return error
    │
    └─ Update Circuit Breaker
        ├─ 5 failures → Open circuit
        └─ 5 minutes → Close circuit
```

Acceptance Criteria

- ✅ README.md updated and comprehensive
- ✅ All branch references corrected
- ✅ API documentation complete
- ✅ Architecture diagrams created
- ✅ Configuration guides complete
- ✅ Troubleshooting guide complete
- ✅ All documentation reviewed and accurate
- ✅ Examples tested and working

APPENDIX A: FILE REFERENCE

Critical Files by Priority

P0 - Critical:
- `config.py` - Central timeout configuration
- `src/daemon/ws_server.py` - WebSocket daemon
- `scripts/run_ws_shim.py` - WebSocket shim
- `tools/workflow/base.py` - Workflow tool base
- `tools/workflow/expert_analysis.py` - Expert validation
- `utils/progress.py` - Progress heartbeat (NEW)
- `utils/logging_unified.py` - Unified logging (NEW)

P1 - High Priority:
- `Daemon/mcp-config.auggie.json` - Auggie configuration
- `Daemon/mcp-config.augmentcode.json` - VSCode configuration
- `Daemon/mcp-config.claude.json` - Claude configuration
- `utils/error_handling.py` - Graceful degradation (NEW)
- `.env` - Environment variables

P2 - Medium Priority:
- `src/providers/capabilities.py` - Web search schemas
- `tools/simple/base.py` - Simple tool base
- `src/providers/glm_chat.py` - GLM provider
- `src/providers/kimi_chat.py` - Kimi provider
- `utils/metrics.py` - Metrics tracking (NEW)

New Files to Create

1. `utils/progress.py` - Progress heartbeat system
2. `utils/logging_unified.py` - Unified logging
3. `utils/error_handling.py` - Graceful degradation
4. `utils/metrics.py` - Metrics tracking
5. `utils/config_validation.py` - Configuration validation
6. `tools/base_tool_interface.py` - Base tool interface
7. `tools/response_format.py` - Standard response format
8. `tools/diagnostics/health.py` - Health check tool
9. `Daemon/mcp-config.base.json` - Base configuration template
10. `docs/configuration/mcp-configs.md` - Configuration guide
11. `docs/features/web-search.md` - Web search documentation
12. `docs/features/continuation.md` - Continuation documentation
13. `docs/troubleshooting.md` - Troubleshooting guide
14. `tests/test_timeout_hierarchy.py` - Timeout tests
15. `tests/test_progress_heartbeat.py` - Heartbeat tests
16. `tests/test_logging_unified.py` - Logging tests
17. `tests/test_graceful_degradation.py` - Degradation tests
18. `tests/test_glm_web_search.py` - GLM web search tests
19. `tests/test_kimi_web_search.py` - Kimi web search tests

APPENDIX B: ACCEPTANCE CRITERIA CHECKLIST

Week 1 (P0 - Critical Fixes)

Timeout Hierarchy:
- [ ] TimeoutConfig class validates hierarchy on import
- [ ] Daemon timeout = 180s (1.5x tool timeout)
- [ ] Shim timeout = 240s (2x tool timeout)
- [ ] Client timeout = 300s (2.5x tool timeout)
- [ ] Workflow tools timeout at 120s
- [ ] Expert validation timeouts at 90s
- [ ] All three MCP configs updated consistently
- [ ] Documentation updated in .env.example

Progress Heartbeat:
- [ ] ProgressHeartbeat class sends updates at configured interval
- [ ] Workflow tools send progress updates every 6 seconds
- [ ] Expert validation sends progress updates every 8 seconds
- [ ] Provider calls send progress updates every 5 seconds

- [ ] Progress messages include elapsed time and estimated remaining
- [ ] Progress messages logged correctly
- [ ] WebSocket server routes progress messages to clients
- [ ] No performance degradation from heartbeat system

Logging Unification:
- [ ] UnifiedLogger class created with all required methods
- [ ] Simple tools log correctly (start, complete, error)
- [ ] Workflow tools log correctly (start, progress, complete, error)
- [ ] Expert validation logs correctly (start, complete)
- [ ] All logs include request_id for tracking
- [ ] Logs written to .logs/toolcalls.jsonl in JSONL format
- [ ] Sensitive data sanitized in logs
- [ ] Log buffer flushes correctly
- [ ] No performance degradation from logging

Week 2 (P1 - High Priority Fixes)

Expert Validation:
- [ ] Root cause identified and documented
- [ ] Fix implemented based on root cause
- [ ] Expert validation called exactly once per step
- [ ] Duration is 90-120 seconds (not 300+)
- [ ] Expert analysis contains real content (not null)
- [ ] No duplicate calls in logs
- [ ] Expert validation re-enabled in .env
- [ ] All workflow tools tested with expert validation

Configuration Standardization:
- [ ] Base configuration template created
- [ ] All three client configs updated consistently
- [ ] Timeout values standardized across all configs
- [ ] Only concurrency and session management differ between clients
- [ ] Configuration differences documented
- [ ] All configs tested and working
- [ ] Documentation accurate and complete

Graceful Degradation:
- [ ] GracefulDegradation class implemented
- [ ] Expert validation has fallback (skip validation)
- [ ] Web search has fallback (GLM → Kimi → no search)
- [ ] Provider has fallback (requested model → glm-4.5-flash)
- [ ] Circuit breaker opens after 5 failures
- [ ] Circuit breaker recovers after 5 minutes
- [ ] Circuit breaker status endpoint working
- [ ] All fallbacks tested and working
- [ ] No silent failures (all errors logged)

Week 3 (P2 - Enhancements)

Web Search Integration:
- [ ] GLM web search tool schema verified
- [ ] GLM web search auto-injection working

- [ ] GLM web search logging implemented
- [ ] Kimi web search tool schema verified
- [ ] Kimi web search auto-injection working
- [ ] Kimi web search logging implemented
- [ ] Web search metrics tracking implemented
- [ ] Web search tests passing
- [ ] Both GLM and Kimi web search working
- [ ] Web search documentation complete

Continuation System:
- [ ] Continuation ID optional (not forced)
- [ ] Single-turn responses don't include continuation
- [ ] Multi-turn responses include continuation when requested
- [ ] Continuation format simplified
- [ ] Metadata moved to separate field
- [ ] Documentation updated
- [ ] All tests passing
- [ ] Backward compatibility maintained

Documentation:
- [ ] All branch references corrected
- [ ] README.md updated and comprehensive
- [ ] API documentation complete
- [ ] Architecture diagrams created
- [ ] Configuration guides complete
- [ ] Troubleshooting guide complete
- [ ] All documentation reviewed and accurate
- [ ] Examples tested and working

WebSocket Daemon:
- [ ] Connection health checks implemented
- [ ] Automatic reconnection working
- [ ] Connection pooling implemented
- [ ] Error recovery mechanisms working
- [ ] Health check endpoint implemented
- [ ] Health check tool working
- [ ] VSCode Augment optimizations verified
- [ ] All stability improvements tested

Testing and Validation

Unit Tests:
- [ ] All unit tests passing
- [ ] Timeout hierarchy tests passing
- [ ] Progress heartbeat tests passing
- [ ] Logging tests passing
- [ ] Graceful degradation tests passing

Integration Tests:
- [ ] All integration tests passing
- [ ] Simple tools integration tests passing

- [ ] Workflow tools integration tests passing
- [ ] Web search integration tests passing

End-to-End Tests:
- [ ] VSCode Augment tests passing
- [ ] Auggie CLI tests passing
- [ ] Claude Desktop tests passing

Performance:
- [ ] Performance benchmarks meet targets
- [ ] No performance degradation from fixes
- [ ] Concurrent execution working correctly

Regression:
- [ ] Regression tests passing
- [ ] No functionality broken by fixes
- [ ] Backward compatibility maintained

APPENDIX C: CONTACT AND SUPPORT

For Questions or Issues

During Implementation:
- Review diagnosis report: `/home/ubuntu/diagnosis_report.md`
- Review this plan: `/home/ubuntu/master_implementation_plan.md`
- Check documentation: `docs/`
- Review test results: `tests/`

After Implementation:
- Create GitHub issues for bugs
- Submit pull requests for improvements
- Update documentation as needed

CONCLUSION

This master implementation plan provides a comprehensive, actionable roadmap for fixing all identified issues and completing the EX-AI MCP Server system according to the original vision.

Key Success Factors:
1. Follow the implementation sequence (Week 1 → Week 2 → Week 3)
2. Validate each fix before moving to the next
3. Test thoroughly at each checkpoint
4. Document as you go
5. Maintain backward compatibility
6. Keep rollback procedures ready

Expected Timeline:
- Week 1: Critical fixes (P0)
- Week 2: High priority fixes (P1)

- Week 3: Enhancements (P2)
- Total: 3 weeks to production-ready system

Final State:
- ✅ All tools working correctly
- ✅ Predictable performance (60-120s for workflow tools)
- ✅ Comprehensive logging and monitoring
- ✅ Graceful error handling
- ✅ Clear documentation
- ✅ VSCode Augment extension fully functional
- ✅ Original vision from master task list achieved

This plan is ready for an AI coder to execute step-by-step, with clear instructions, acceptance criteria, and validation procedures for each fix.

Document Version: 1.0
Created: October 4, 2025
Last Updated: October 4, 2025
Status: Ready for Implementation