

Umsetzungsplan: Agentenkonfiguration Portierung zu claude-flow@alpha (V86)

Hauptaufgaben-Checkliste

1. **Migration der Agentendefinitionen** von Python-Constants zu JSON-Format
2. **Implementierung eines Config-Translators** für claude-flow v86 Format-Konvertierung
3. **Erweiterung des Hive-Launch-Prozesses** mit v86-spezifischer Konfigurationserstellung
4. **Anpassung der WSL-Bridge** für erweiterte Agenten-Parameter-Übergabe
5. **Validierung und Testing** der generierten Konfigurationen gegen v86-Schema

1. Architektur-Überblick

Neue Verzeichnisstruktur

```
src/
├── Agents_Configuration/      # NEU: Zentrale JSON-Konfigurationen
│   ├── agents/              # Individuelle Agentendefinitionen
│   │   ├── queen.json
│   │   ├── backend-dev.json
│   │   ├── frontend-dev.json
│   │   └── [64 weitere Agenten].json
│   ├── categories/          # Kategoriedefinitionen
│   │   ├── development.json
│   │   ├── testing.json
│   │   └── [14 weitere].json
│   ├── presets/             # Vordefinierte Agenten-Gruppen
│   │   ├── python-development.json
│   │   ├── full-stack.json
│   │   └── [weitere presets].json
│   └── schema/              # Validierungsschemas
│       ├── agent.schema.json
│       └── preset.schema.json
├── claude_flow_gui/
│   ├── converters/          # NEU: Format-Konverter
│   │   ├── __init__.py
│   │   ├── v86_converter.py  # claude-flow v86 Konverter
│   │   └── config_validator.py
│   └── mixins/
│       └── hive.py           # ANPASSEN: v86-Unterstützung
```

2. Detaillierte Implementierungsschritte

2.1 Migration der Agentendefinitionen zu JSON

Aktueller Zustand: Agenten in `constants.py` als Python-Dictionaries

Zielzustand: Strukturierte JSON-Dateien mit erweiterten Metadaten

Beispiel Agent-JSON (`src/Agents_Configuration/agents/backend-dev.json`):

```
json
{
  "id": "backend-dev",
  "name": "Backend Developer",
  "category": "development",
  "role": "primary",
  "color": "#4169E1",
  "icon": "🔧",
  "description": "Specializes in server-side logic, APIs, and database interactions",
  "capabilities": {
    "languages": ["python", "typescript", "rust", "go"],
    "frameworks": ["django", "fastapi", "express", "actix"],
    "databases": ["postgresql", "mongodb", "redis"],
    "tools": ["docker", "kubernetes", "terraform"]
  },
  "verification": {
    "methods": ["compile", "test", "lint", "security-scan"],
    "truthThreshold": 0.95,
    "maxFilesPerOperation": 10
  },
  "performance": {
    "priority": "high",
    "maxConcurrentTasks": 3,
    "timeout": 300000
  },
  "prompts": {
    "initialization": "You are a backend developer agent specializing in...",
    "taskTemplate": "Implement {task} following best practices for {language}..."
  }
}
```

Beispiel Preset-JSON (`src/Agents_Configuration/presets/python-development.json`):

```
json
```

```
{
  "id": "python-development",
  "name": "🐍 Python Development",
  "description": "Complete Python development team with testing and DevOps",
  "topology": "hierarchical",
  "agents": [
    {
      "id": "queen",
      "required": true,
      "model": "opus-4.1"
    },
    {
      "id": "backend-dev",
      "required": true,
      "model": "sonnet-4"
    },
    {
      "id": "tester",
      "required": true,
      "model": "sonnet-4"
    },
    {
      "id": "performance-tester",
      "required": false,
      "model": "sonnet-4"
    },
    {
      "id": "devops-engineer",
      "required": false,
      "model": "sonnet-4"
    }
  ],
  "orchestration": {
    "maxAgents": 8,
    "maxConcurrentAgents": 5,
    "strategy": "development",
    "faultTolerance": {
      "strategy": "retry-with-learning",
      "maxRetries": 3
    }
  },
  "hooks": {
    "preTask": ["validate-requirements", "setup-environment"],
    "postTask": ["run-tests", "update-memory"]
  }
}
```

```
}  
}
```

2.2 Config-Translator für claude-flow v86

Neue Datei: `src/claude_flow_gui/converters/v86_converter.py`

```
python
```

```

import json
import os
from pathlib import Path
from typing import Dict, List, Any
from datetime import datetime

class V86ConfigConverter:
    """Converts AI Coding Suite configs to claude-flow v86 format"""

    def __init__(self, agents_config_path: str = "src/Agents_Configuration"):
        self.config_path = Path(agents_config_path)
        self.agents = self._load_agents()
        self.presets = self._load_presets()

    def generate_v86_config(self,
                           selected_agents: List[str],
                           preset_name: str,
                           task: str,
                           project_path: str) -> Dict[str, Any]:
        """Generate complete v86 configuration structure"""

        # Load preset if available
        preset = self.presets.get(preset_name, {})

        # Build agent specifications with v86 requirements
        agent_specs = self._build_agent_specifications(selected_agents)

        config = {
            "name": f"{Path(project_path).name}-swarm",
            "version": "2.0.0-alpha.86",
            "orchestrator": {
                "maxAgents": len(selected_agents),
                "maxConcurrentAgents": min(8, len(selected_agents)),
                "defaultTopology": preset.get("topology", "hierarchical"),
                "strategy": preset.get("orchestration", {}).get("strategy", "development"),
                "memoryEnabled": True,
                "faultTolerance": {
                    "strategy": "retry-with-learning",
                    "maxRetries": 3,
                    "byzantineFaultTolerance": True,
                    "healthCheckInterval": 30000
                }
            },
            "agents": {
                "types": selected_agents,
                "spawning": {

```

```
        "autoSpawn": True,
        "maxAge": "2h",
        "healthCheck": True,
        "batchSize": min(5, len(selected_agents))
    },
    "specialization": agent_specs
},
"memory": {
    "backend": "sqlite",
    "persistentSessions": True,
    "database": ".swarm/memory.db",
    "tables": 12,
    "cacheSizeMB": 200,
    "compression": True,
    "distributedSync": True,
    "namespaces": ["default", "sparc", "neural", "coordination"],
    "retentionDays": 30
},
"neural": {
    "enabled": True,
    "models": 27,
    "wasmSimd": True,
    "training": {
        "patterns": ["coordination", "cognitive-analysis", "task-optimization"],
        "epochs": 50,
        "learningRate": 0.001,
        "batchSize": 32
    }
},
"hooks": self._generate_hooks_config(preset),
"performance": {
    "parallelExecution": True,
    "tokenOptimization": True,
    "batchProcessing": True,
    "timeout": 300000,
    "maxOutputSize": 500000,
    "tokenLimit": 100000
},
"security": {
    "monitoring": True,
    "cryptographicSigning": True,
    "auditTrail": True,
    "sandboxing": True
},
"telemetry": {
    "enabled": True,
    "tokenTracking": True,
```

```

        "costAnalysis": True,
        "realTimeMonitoring": True,
        "exportFormat": "json"
    },
    "task": task,
    "preset": preset_name,
    "metadata": {
        "created": datetime.now().isoformat(),
        "source": "AI Coding Suite",
        "version": "1.0.0"
    }
}

```

```

return config

```

```

def create_agent_configs(self, selected_agents: List[str], project_path: str):

```

```

    """Create individual agent configuration files for v86"""

```

```

    agents_dir = Path(project_path) / "agents"

```

```

    agents_dir.mkdir(exist_ok=True)

```

```

    for agent_id in selected_agents:

```

```

        agent_data = self.agents.get(agent_id, {})

```

```

        # Create v86-specific agent config

```

```

        agent_config = {
            "id": agent_id,
            "name": agent_data.get("name"),
            "role": agent_data.get("role", "worker"),
            "model": "sonnet-4" if agent_id != "queen" else "opus-4.1",
            "capabilities": agent_data.get("capabilities", {}),
            "verification": agent_data.get("verification", {
                "methods": ["compile", "test"],
                "truthThreshold": 0.85
            }),
            "prompts": agent_data.get("prompts", {}),
            "coordination": {
                "canDelegate": agent_id == "queen",
                "acceptsDelegation": True,
                "reportingInterval": 5000
            }
        }
    }

```

```

    config_path = agents_dir / f"{agent_id}.json"

```

```

    with open(config_path, "w") as f:

```

```

        json.dump(agent_config, f, indent=2)

```

```

def _build_agent_specifications(self, selected_agents: List[str]) -> Dict[str, Any]:

```

```
"""Build agent-specific specialization configs"""
```

```
specs = {}
```

```
for agent_id in selected_agents:
```

```
    agent = self.agents.get(agent_id, {})
```

```
    specs[agent_id] = {
```

```
        "verification": agent.get("verification", {}).get("methods", ["test"]),
```

```
        "truthThreshold": agent.get("verification", {}).get("truthThreshold", 0.85),
```

```
        "maxFilesPerOperation": agent.get("verification", {}).get("maxFilesPerOperation", 10)
```

```
    }
```

```
# Add language-specific settings for dev agents
```

```
if "capabilities" in agent and "languages" in agent["capabilities"]:
```

```
    specs[agent_id]["languages"] = agent["capabilities"]["languages"]
```

```
return specs
```

```
def _load_agents(self) -> Dict[str, Any]:
```

```
    """Load all agent definitions from JSON files"""
```

```
    agents = {}
```

```
    agents_dir = self.config_path / "agents"
```

```
    if agents_dir.exists():
```

```
        for agent_file in agents_dir.glob("*.json"):
```

```
            with open(agent_file, "r") as f:
```

```
                agent_data = json.load(f)
```

```
                agents[agent_data["id"]] = agent_data
```

```
    return agents
```

```
def _load_presets(self) -> Dict[str, Any]:
```

```
    """Load all preset definitions from JSON files"""
```

```
    presets = {}
```

```
    presets_dir = self.config_path / "presets"
```

```
    if presets_dir.exists():
```

```
        for preset_file in presets_dir.glob("*.json"):
```

```
            with open(preset_file, "r") as f:
```

```
                preset_data = json.load(f)
```

```
                presets[preset_data["id"]] = preset_data
```

```
    return presets
```

```
def _generate_hooks_config(self, preset: Dict[str, Any]) -> Dict[str, Any]:
```

```
    """Generate hooks configuration based on preset"""
```

```
    default_hooks = {
```



```

"enabled": True,
"types": [
    "pre-task", "post-task", "pre-edit", "post-edit",
    "pre-command", "post-command", "session-start",
    "session-end", "pre-search", "post-search",
    "pre-analysis", "post-analysis", "error-recovery", "notify"
],
"automation": {
    "agentAssignment": True,
    "performanceTracking": True,
    "errorRecovery": True,
    "autoFormat": True,
    "testOnSave": True
}
}

```

Merge with preset hooks if available

if "hooks" in preset:

 default_hooks.update(preset["hooks"])

return default_hooks

2.3 Erweiterte Hive-Launch Implementation

Anpassung: `src/claude_flow_gui/mixins/hive.py`

python

```

import json
from pathlib import Path
from datetime import datetime
from typing import List, Dict, Any

def launch_hive_mind_v86(self):
    """Launch Hive Mind with v86 configuration support"""

    # Import converter
    from ..converters.v86_converter import V86ConfigConverter

    # Get current configuration
    selected_agents = self.get_selected_agents()
    preset_name = self.agent_preset_combo.currentText()
    task = self.task_input.toPlainText()
    project_path = self.project_path_input.text()

    # Initialize converter
    converter = V86ConfigConverter()

    # Check for existing v86 config
    v86_config_path = Path(project_path) / "claude-flow.config.json"

    if v86_config_path.exists():
        # Update existing config
        with open(v86_config_path, "r") as f:
            existing_config = json.load(f)

        # Update task and agents
        existing_config["task"] = task
        existing_config["agents"]["types"] = selected_agents
        existing_config["metadata"]["updated"] = datetime.now().isoformat()

        config = existing_config
        print(f"✅ Updated existing v86 config: {v86_config_path}")
    else:
        # Generate new v86 configuration
        config = converter.generate_v86_config(
            selected_agents=selected_agents,
            preset_name=preset_name,
            task=task,
            project_path=project_path
        )
        print(f"✅ Generated new v86 config")

    # Save main config

```

```
with open(v86_config_path, "w") as f:
    json.dump(config, f, indent=2)

# Create required directory structure
self._create_v86_structure(project_path)

# Generate individual agent configs
converter.create_agent_configs(selected_agents, project_path)

# Create MCP configuration
self._create_mcp_config(project_path)

# Create Claude settings
self._create_claude_settings(project_path, selected_agents)

# Create CLAUDE.md context file
self._create_claude_context(project_path, config)

# Launch via WSL with v86 command
self._execute_v86_command(project_path, task)

def _create_v86_structure(self, project_path: str):
    """Create v86 required directory structure"""
    dirs = [
        ".claude",
        ".claude/commands",
        ".claude/commands/analysis",
        ".claude/commands/automation",
        ".claude/commands/coordination",
        ".claude/commands/github",
        ".claude/commands/hooks",
        ".claude/commands/memory",
        ".claude/commands/workflows",
        ".mcp",
        ".hive-mind",
        ".hive-mind/sessions",
        ".swarm",
        "agents",
        "swarms",
        "swarms/development",
        "swarms/testing",
        "swarms/production",
        "workflows",
        "workflows/ci-cd",
        "workflows/deployment"
    ]
```

```
for dir_name in dirs:
    dir_path = Path(project_path) / dir_name
    dir_path.mkdir(parents=True, exist_ok=True)
    print(f"📁 Created: {dir_path}")
```

```
def _create_mcp_config(self, project_path: str):
    """Create MCP server configuration"""
    mcp_config = {
        "mcpServers": {
            "claude-flow": {
                "command": "npx",
                "args": ["-y", "claude-flow@alpha", "mcp", "start"],
                "env": {}
            },
            "ruv-swarm": {
                "command": "npx",
                "args": ["-y", "ruv-swarm@latest", "mcp", "start"],
                "env": {}
            }
        }
    }
```

```
mcp_path = Path(project_path) / ".mcp.json"
with open(mcp_path, "w") as f:
    json.dump(mcp_config, f, indent=2)
    print(f"✅ Created MCP config: {mcp_path}")
```

```
def _create_claude_settings(self, project_path: str, selected_agents: List[str]):
    """Create Claude settings file"""
    settings = {
        "model": "sonnet",
        "permissions": {
            "allow": [
                "Bash(mkdir:*)",
                "Bash(npm:*)",
                "Bash(node:*)",
                "Bash(npx:*)",
                "Write",
                "Edit",
                "MultiEdit"
            ],
            "deny": []
        },
        "hooks": {
            "PreToolUse": [
                {
                    "matcher": "Bash",
```

```

        "hooks": [
            {
                "type": "command",
                "command": "npx claude-flow@alpha hooks pre-command --command \"{}\" --validate-safety true"
            }
        ]
    },
],
"PostToolUse": [
    {
        "matcher": "Write|Edit|MultiEdit",
        "hooks": [
            {
                "type": "command",
                "command": "npx claude-flow@alpha hooks post-edit --file \"{}\" --memory-key \"swarm/{agent}/{step}\""
            }
        ]
    }
]
},
"env": {
    "BASH_DEFAULT_TIMEOUT_MS": "300000",
    "BASH_MAX_TIMEOUT_MS": "600000",
    "CLAUDE_FLOW_MAX_AGENTS": str(len(selected_agents)),
    "CLAUDE_FLOW_HOOKS_ENABLED": "true",
    "CLAUDE_FLOW_TELEMETRY_ENABLED": "true"
}
}

```

```
settings_path = Path(project_path) / ".claude" / "settings.json"
```

```
with open(settings_path, "w") as f:
```

```
    json.dump(settings, f, indent=2)
```

```
print(f"✅ Created Claude settings: {settings_path}")
```

```
def _create_claude_context(self, project_path: str, config: Dict[str, Any]):
```

```
    """Create CLAUDE.md context file for AI understanding"""
```

```
    context = f"""# Project Context for Claude
```

```
## Project Information
```

```
- **Name**: {config.get('name', 'Unknown')}
```

```
- **Version**: {config.get('version', '2.0.0-alpha.86')}
```

```
- **Created**: {config.get('metadata', {}).get('created', datetime.now().isoformat())}
```

```
- **Source**: AI Coding Suite
```

```
## Current Task
```

```
{config.get('task', 'No task specified')}
```

```
## Active Agents ({len(config.get('agents', {})).get('types', [])})
{'', 'join(config.get('agents', {})).get('types', [])})
```

Configuration

```
- **Topology**: {config.get('orchestrator', {}).get('defaultTopology', 'hierarchical')}
- **Strategy**: {config.get('orchestrator', {}).get('strategy', 'development')}
- **Max Concurrent Agents**: {config.get('orchestrator', {}).get('maxConcurrentAgents', 8)}
- **Memory Enabled**: {config.get('orchestrator', {}).get('memoryEnabled', True)}
- **Neural Models**: {config.get('neural', {}).get('models', 27)}
```

Capabilities

```
- Byzantine Fault Tolerance: ✓
- Persistent Memory: ✓
- Neural Processing: ✓
- Lifecycle Hooks: ✓
- Real-time Telemetry: ✓
```

Instructions

This project uses `claude-flow@alpha v86` for multi-agent orchestration. Each agent has specialized capabilities and veri

The system supports parallel execution, intelligent task distribution, and cross-session memory persistence through SQ

Use the following commands for orchestration:

```
- `npx claude-flow@alpha hive-mind spawn "<task>"` - Start new swarm
- `npx claude-flow@alpha swarm monitor` - Monitor active agents
- `npx claude-flow@alpha memory status` - Check memory usage
```

```
"""
```

```
context_path = Path(project_path) / "CLAUDE.md"
with open(context_path, "w") as f:
    f.write(context)
print(f"✓ Created context file: {context_path}")
```

```
def _execute_v86_command(self, project_path: str, task: str):
```

```
    """Execute v86 hive-mind command via WSL"""
```

```
    # Import WSL bridge
```

```
    from ..wsl_bridge import WSLBridge
```

```
    wsl = WSLBridge()
```

```
    # Build command
```

```
    command = [
        "npx", "claude-flow@alpha",
        "hive-mind", "spawn",
        f"{task}",
        "--config", f"./{Path(project_path).name}/claude-flow.config.json",
        "--claude",
```

```
--verbose",
--memory", "persistent",
--hooks", "enabled",
--telemetry", "enabled"
]

# Set environment variables
env_vars = {
    "CLAUDE_FLOW_DEBUG": "verbose",
    "CLAUDE_FLOW_MEMORY_PATH": f".swarm/memory.db",
    "CLAUDE_FLOW_HOOKS_ENABLED": "true",
    "CLAUDE_FLOW_TELEMETRY_ENABLED": "true"
}

# Execute
wsl.execute_command(
    command=" ".join(command),
    working_directory=project_path,
    environment=env_vars
)
```

2.4 WSL-Bridge Anpassungen

Anpassung: `src/claude_flow_gui/wsl_bridge.py`

```
python
```

```

import subprocess
import json
from pathlib import Path
from typing import Dict, List, Optional

class WSLBridge:
    """Enhanced WSL Bridge with v86 support"""

    def execute_v86_hive_command(self,
                                project_path: str,
                                task: str,
                                config_path: Optional[str] = None):
        """Execute claude-flow v86 hive-mind command"""

        # Determine config path
        if not config_path:
            config_path = f"{project_path}/claude-flow.config.json"

        # Build v86 command
        command_parts = [
            "wsl", "bash", "-c",
            f"cd {self._to_wsl_path(project_path)} && "
            f"npx claude-flow@alpha hive-mind spawn \"{task}\" "
            f"--config \"{config_path}\" "
            f"--claude --verbose "
            f"--memory persistent "
            f"--hooks enabled "
            f"--telemetry enabled"
        ]

        # Add environment variables
        env = os.environ.copy()
        env.update({
            "CLAUDE_FLOW_DEBUG": "verbose",
            "CLAUDE_FLOW_MEMORY_PATH": f"{project_path}/swarm/memory.db",
            "CLAUDE_FLOW_HOOKS_ENABLED": "true",
            "CLAUDE_FLOW_TELEMETRY_ENABLED": "true",
            "CLAUDE_FLOW_MAX_AGENTS": "12"
        })

        # Execute command
        try:
            result = subprocess.run(
                command_parts,
                env=env,
                capture_output=True,

```



```
text=True,  
timeout=300  
)
```

```
if result.returncode == 0:  
    print(f"✅ V86 Hive Mind launched successfully")  
    return result.stdout  
else:  
    print(f"❌ Error launching V86: {result.stderr}")  
    return None
```

```
except subprocess.TimeoutExpired:  
    print("⚠️ Command timed out after 5 minutes")  
    return None  
except Exception as e:  
    print(f"❌ Execution error: {e}")  
    return None
```

```
def validate_v86_installation(self) -> bool:  
    """Check if claude-flow@alpha v86 is available"""  
    try:  
        result = subprocess.run(  
            ["wsl", "bash", "-c", "npx claude-flow@alpha --version"],  
            capture_output=True,  
            text=True,  
            timeout=10  
        )
```

```
        if "2.0.0-alpha.86" in result.stdout:  
            print("✅ claude-flow@alpha v86 is installed")  
            return True  
        else:  
            print("⚠️ Different version detected:", result.stdout)  
            return False
```

```
    except Exception as e:  
        print(f"❌ Could not verify installation: {e}")  
        return False
```

```
def _to_wsl_path(self, windows_path: str) -> str:  
    """Convert Windows path to WSL path"""  
    path = Path(windows_path)  
    if path.drive:  
        # Convert C:\path\to\file to /mnt/c/path/to/file  
        drive = path.drive[0].lower()  
        path_str = str(path).replace(path.drive + "\\", "")
```

```
return f"/mnt/{drive}/{path_str.replace(chr(92), '/')}"  
return windows_path
```

3. Konfigurationsmigration und Loader

3.1 Migration Tool für bestehende Konfigurationen

Neue Datei: `src/claude_flow_gui/tools/migrate_configs.py`

```
python
```

```
#!/usr/bin/env python3
```

```
"""Migration tool for converting existing configs to v86 format"""
```

```
import json
```

```
import sys
```

```
from pathlib import Path
```

```
from typing import Dict, Any, List
```

```
from datetime import datetime
```

```
class ConfigMigrator:
```

```
    """Migrate AI Coding Suite configs to v86 format"""
```

```
    def __init__(self):
```

```
        self.migration_log = []
```

```
    def migrate_saved_config_to_v86(self,
```

```
        input_path: str,
```

```
        output_path: str = None) -> Dict[str, Any]:
```

```
    """Migrate existing saved config to v86 format"""
```

```
    print(f"📁 Reading config from: {input_path}")
```

```
    with open(input_path, "r") as f:
```

```
        old_config = json.load(f)
```

```
    # Extract old format data
```

```
    project_data = old_config.get("project", {})
```

```
    agents_data = old_config.get("agents", {})
```

```
    swarm_data = old_config.get("swarm", {})
```

```
    settings_data = old_config.get("settings", {})
```

```
    # Map to v86 format
```

```
    v86_config = {
```

```
        "name": project_data.get("name", "migrated-project"),
```

```
        "version": "2.0.0-alpha.86",
```

```
        "orchestrator": {
```

```
            "maxAgents": len(agents_data.get("selected", [])),
```

```
            "maxConcurrentAgents": min(8, len(agents_data.get("selected", []))),
```

```
            "defaultTopology": swarm_data.get("topology", "hierarchical"),
```

```
            "strategy": "development",
```

```
            "memoryEnabled": True,
```

```
            "faultTolerance": {
```

```
                "strategy": "retry-with-learning",
```

```
                "maxRetries": 3,
```

```
                "byzantineFaultTolerance": True,
```

```
                "healthCheckInterval": 30000
```

```

    }
},
"agents": {
    "types": agents_data.get("selected", []),
    "spawning": {
        "autoSpawn": True,
        "maxAge": "2h",
        "healthCheck": True,
        "batchSize": min(5, len(agents_data.get("selected", [])))
    },
    "specialization": self._migrate_agent_specs(agents_data)
},
"memory": {
    "backend": "sqlite",
    "persistentSessions": True,
    "database": ".swarm/memory.db",
    "tables": 12,
    "cacheSizeMB": int(settings_data.get("memorySize", "200MB").replace("MB", "")),
    "compression": True,
    "distributedSync": True,
    "namespaces": ["default", "sparc", "neural", "coordination"],
    "retentionDays": 30
},
"neural": {
    "enabled": True,
    "models": 27,
    "wasmSimd": True,
    "training": {
        "patterns": ["coordination", "cognitive-analysis", "task-optimization"],
        "epochs": 50,
        "learningRate": 0.001,
        "batchSize": 32
    }
},
"hooks": {
    "enabled": True,
    "types": [
        "pre-task", "post-task", "pre-edit", "post-edit",
        "pre-command", "post-command", "session-start",
        "session-end", "pre-search", "post-search",
        "pre-analysis", "post-analysis", "error-recovery", "notify"
    ],
    "automation": {
        "agentAssignment": True,
        "performanceTracking": True,
        "errorRecovery": True,
        "autoFormat": settings_data.get("autoFormat", True),

```

```

        "testOnSave": settings_data.get("testOnSave", False)
    }
},
"performance": {
    "parallelExecution": settings_data.get("parallelExecution", True),
    "tokenOptimization": True,
    "batchProcessing": True,
    "timeout": int(settings_data.get("timeout", "300000")),
    "maxOutputSize": 500000,
    "tokenLimit": 100000
},
"security": {
    "monitoring": True,
    "cryptographicSigning": True,
    "auditTrail": True,
    "sandboxing": True
},
"telemetry": {
    "enabled": True,
    "tokenTracking": True,
    "costAnalysis": True,
    "realTimeMonitoring": True,
    "exportFormat": "json"
},
"task": swarm_data.get("task", ""),
"preset": old_config.get("preset", "Custom"),
"metadata": {
    "created": datetime.now().isoformat(),
    "source": "AI Coding Suite",
    "version": "1.0.0",
    "migrated": True,
    "originalConfig": Path(input_path).name
}
}

```

Save migrated config

if not output_path:

output_path = input_path.replace(".json", "_v86.json")

with open(output_path, "w") as f:

json.dump(v86_config, f, indent=2)

print(f"✅ Migrated config saved to: {output_path}")

Log migration

self.migration_log.append({

"input": input_path,

```
    "output": output_path,
    "timestamp": datetime.now().isoformat(),
    "agents_count": len(v86_config["agents"]["types"])
})
```

```
return v86_config
```

```
def _migrate_agent_specs(self, agents_data: Dict[str, Any]) -> Dict[str, Any]:
```

```
    """Migrate agent specifications to v86 format"""
```

```
    specs = {}
```

```
    # Default specifications for known agent types
```

```
    default_specs = {
```

```
        "queen": {
```

```
            "verification": ["orchestrate", "validate", "delegate"],
```

```
            "truthThreshold": 0.98,
```

```
            "maxFilesPerOperation": 20
```

```
        },
```

```
        "backend-dev": {
```

```
            "verification": ["compile", "test", "lint"],
```

```
            "truthThreshold": 0.95,
```

```
            "maxFilesPerOperation": 10,
```

```
            "languages": ["python", "typescript", "rust"]
```

```
        },
```

```
        "frontend-dev": {
```

```
            "verification": ["compile", "test", "lint", "build"],
```

```
            "truthThreshold": 0.92,
```

```
            "maxFilesPerOperation": 15,
```

```
            "languages": ["typescript", "javascript", "react"]
```

```
        },
```

```
        "tester": {
```

```
            "verification": ["unit-tests", "integration-tests", "coverage"],
```

```
            "truthThreshold": 0.90,
```

```
            "maxFilesPerOperation": 20
```

```
        }
```

```
    }
```

```
for agent in agents_data.get("selected", []):
```

```
    if agent in default_specs:
```

```
        specs[agent] = default_specs[agent]
```

```
    else:
```

```
        # Generic specification for unknown agents
```

```
        specs[agent] = {
```

```
            "verification": ["test", "validate"],
```

```
            "truthThreshold": 0.85,
```

```
            "maxFilesPerOperation": 10
```

```
        }
```

```

    return specs

def batch_migrate(self, config_dir: str):
    """Migrate all configs in a directory"""
    config_path = Path(config_dir)

    for config_file in config_path.glob("*.json"):
        if "_v86" not in str(config_file):
            print(f"\n🔄 Migrating: {config_file.name}")
            self.migrate_saved_config_to_v86(str(config_file))

    # Save migration log
    log_path = config_path / "migration_log.json"
    with open(log_path, "w") as f:
        json.dump(self.migration_log, f, indent=2)

    print(f"\n✅ Migration complete! Log saved to: {log_path}")

if __name__ == "__main__":
    migrator = ConfigMigrator()

    if len(sys.argv) < 2:
        print("Usage: python migrate_configs.py <input_config.json> [output_config.json]")
        print(" or: python migrate_configs.py --batch <config_directory>")
        sys.exit(1)

    if sys.argv[1] == "--batch":
        if len(sys.argv) < 3:
            print("Please specify a directory for batch migration")
            sys.exit(1)
        migrator.batch_migrate(sys.argv[2])
    else:
        output = sys.argv[2] if len(sys.argv) > 2 else None
        migrator.migrate_saved_config_to_v86(sys.argv[1], output)

```

4. Validierung und Testing

4.1 Schema Validierung

Neue Datei: `src/Agents_Configuration/schema/v86.schema.json`

json

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Claude Flow v86 Configuration",
  "type": "object",
  "required": ["name", "version", "orchestrator", "agents"],
  "properties": {
    "name": {
      "type": "string",
      "description": "Project or swarm name"
    },
    "version": {
      "type": "string",
      "pattern": "^2\\.0\\.0-alpha\\.\\.\\d+$",
      "description": "Claude Flow version"
    },
    "orchestrator": {
      "type": "object",
      "required": ["maxAgents", "defaultTopology"],
      "properties": {
        "maxAgents": {
          "type": "integer",
          "minimum": 1,
          "maximum": 15
        },
        "maxConcurrentAgents": {
          "type": "integer",
          "minimum": 1,
          "maximum": 12
        },
        "defaultTopology": {
          "type": "string",
          "enum": ["hierarchical", "mesh", "ring", "star", "sequential"]
        },
        "strategy": {
          "type": "string",
          "enum": ["development", "balanced", "parallel"]
        },
        "memoryEnabled": {
          "type": "boolean"
        },
        "faultTolerance": {
          "type": "object",
          "properties": {
            "strategy": {
              "type": "string"
            }
          }
        }
      }
    }
  }
}
```



```
"maxRetries": {
  "type": "integer",
  "minimum": 1,
  "maximum": 10
},
"byzantineFaultTolerance": {
  "type": "boolean"
},
"healthCheckInterval": {
  "type": "integer",
  "minimum": 1000
}
}
}
},
"agents": {
  "type": "object",
  "required": ["types"],
  "properties": {
    "types": {
      "type": "array",
      "items": {
        "type": "string"
      },
      "minItems": 1,
      "maxItems": 64
    },
    "spawning": {
      "type": "object",
      "properties": {
        "autoSpawn": {
          "type": "boolean"
        },
        "maxAge": {
          "type": "string",
          "pattern": "^\\d+[hms]$"
        }
      },
      "healthCheck": {
        "type": "boolean"
      },
      "batchSize": {
        "type": "integer",
        "minimum": 1,
        "maximum": 10
      }
    }
  }
}
```

```
},
"specialization": {
  "type": "object",
  "additionalProperties": {
    "type": "object",
    "properties": {
      "verification": {
        "type": "array",
        "items": {
          "type": "string"
        }
      },
    },
    "truthThreshold": {
      "type": "number",
      "minimum": 0,
      "maximum": 1
    },
    "maxFilesPerOperation": {
      "type": "integer",
      "minimum": 1
    },
    "languages": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  }
}
},
"memory": {
  "type": "object",
  "properties": {
    "backend": {
      "type": "string",
      "enum": ["sqlite", "postgresql", "memory"]
    },
    "persistentSessions": {
      "type": "boolean"
    },
    "database": {
      "type": "string"
    },
    "tables": {
      "type": "integer",
```

```
    "minimum": 1
  },
  "cacheSizeMB": {
    "type": "integer",
    "minimum": 10
  }
},
"neural": {
  "type": "object",
  "properties": {
    "enabled": {
      "type": "boolean"
    },
    "models": {
      "type": "integer",
      "minimum": 1,
      "maximum": 27
    },
    "wasmSimd": {
      "type": "boolean"
    }
  }
},
"task": {
  "type": "string"
},
"preset": {
  "type": "string"
}
}
```

4.2 Test Suite

Neue Datei: `tests/test_v86_converter.py`

python

```
import unittest
import json
import tempfile
import shutil
from pathlib import Path
from src.claude_flow_gui.converters.v86_converter import V86ConfigConverter
from src.claude_flow_gui.tools.migrate_configs import ConfigMigrator

class TestV86Converter(unittest.TestCase):

    def setUp(self):
        """Set up test environment"""
        self.test_dir = tempfile.mkdtemp()
        self.converter = V86ConfigConverter()
        self.migrator = ConfigMigrator()

    def tearDown(self):
        """Clean up test environment"""
        shutil.rmtree(self.test_dir)

    def test_generate_v86_config(self):
        """Test v86 config generation"""
        config = self.converter.generate_v86_config(
            selected_agents=["queen", "backend-dev", "tester"],
            preset_name="python-development",
            task="Build a REST API",
            project_path=self.test_dir
        )

        # Validate structure
        self.assertIn("version", config)
        self.assertEqual(config["version"], "2.0.0-alpha.86")
        self.assertIn("orchestrator", config)
        self.assertIn("agents", config)
        self.assertEqual(len(config["agents"]["types"]), 3)
        self.assertIn("memory", config)
        self.assertIn("neural", config)
        self.assertIn("hooks", config)

    def test_schema_compliance(self):
        """Test schema compliance of generated configs"""
        import jsonschema

        # Load schema
        schema_path = Path("src/Agents_Configuration/schema/v86.schema.json")
        if schema_path.exists():
            schema = json.load(schema_path.open())
            validator = jsonschema.Draft4Validator(schema)

            for config in self.converter.generate_configs():
                validator.validate(config)
```

```

        with open(schema_path) as f:
            schema = json.load(f)
    else:
        # Create minimal schema for testing
        schema = {
            "$schema": "http://json-schema.org/draft-07/schema#",
            "type": "object",
            "required": ["name", "version", "orchestrator", "agents"]
        }

    config = self.converter.generate_v86_config(
        selected_agents=["queen"],
        preset_name="minimal",
        task="Test",
        project_path=self.test_dir
    )

    # Should not raise exception
    try:
        jsonschema.validate(config, schema)
    except jsonschema.ValidationError as e:
        self.fail(f"Schema validation failed: {e}")

    def test_agent_config_creation(self):
        """Test individual agent config file creation"""
        agents = ["queen", "backend-dev"]
        self.converter.create_agent_configs(agents, self.test_dir)

        # Check if files were created
        agents_dir = Path(self.test_dir) / "agents"
        self.assertTrue(agents_dir.exists())

        for agent in agents:
            agent_file = agents_dir / f"{agent}.json"
            self.assertTrue(agent_file.exists())

            # Validate content
            with open(agent_file) as f:
                agent_config = json.load(f)
                self.assertIn("id", agent_config)
                self.assertEqual(agent_config["id"], agent)
                self.assertIn("model", agent_config)

    def test_migration_tool(self):
        """Test config migration from old to v86 format"""
        # Create old format config
        old_config = {

```

```
"project": {"name": "test-project"},
"agents": {
    "selected": ["queen", "backend-dev"],
    "queen_model": "opus-4.1",
    "worker_model": "sonnet-4"
},
"swarm": {
    "topology": "hierarchical",
    "task": "Test migration"
},
"settings": {
    "memorySize": "200MB",
    "parallelExecution": True
},
"preset": "python-development"
}
```

Save old config

```
old_path = Path(self.test_dir) / "old_config.json"
with open(old_path, "w") as f:
    json.dump(old_config, f)
```

Migrate

```
v86_config = self.migrator.migrate_saved_config_to_v86(str(old_path))
```

Validate migration

```
self.assertEqual(v86_config["version"], "2.0.0-alpha.86")
self.assertEqual(v86_config["agents"]["types"], ["queen", "backend-dev"])
self.assertEqual(v86_config["orchestrator"]["defaultTopology"], "hierarchical")
self.assertEqual(v86_config["memory"]["cacheSizeMB"], 200)
self.assertTrue(v86_config["metadata"]["migrated"])
```

def test_batch_migration(self):

"""Test batch migration of multiple configs"""

Create multiple old configs

for i **in** range(3):

```
    config = {
        "project": {"name": f"project-{i}"},
        "agents": {"selected": ["queen"]},
        "swarm": {"topology": "star", "task": f"Task {i}"},
        "settings": {},
        "preset": "minimal"
    }
```

```
    config_path = Path(self.test_dir) / f"config_{i}.json"
```

with open(config_path, "w") **as** f:

```
    json.dump(config, f)
```

```
# Batch migrate
self.migrator.batch_migrate(self.test_dir)

# Check migrated files
migrated_files = list(Path(self.test_dir).glob("*_v86.json"))
self.assertEqual(len(migrated_files), 3)

# Check migration log
log_path = Path(self.test_dir) / "migration_log.json"
self.assertTrue(log_path.exists())

if __name__ == "__main__":
    unittest.main()
```

5. UI-Anpassungen

5.1 Erweiterter Agent Configuration Tab

Anpassung: `src/claude_flow_gui/mixins/ui_tabs.py`

```
python
```

```

from PyQt6.QtWidgets import (QGroupBox, QCheckBox, QSpinBox, QComboBox,
                              QLabel, QGridLayout, QPushButton, QTextEdit)
from PyQt6.QtCore import Qt

def _create_v86_controls(self):
    """Create v86-specific control panel"""

    # V86 Features Group
    v86_group = QGroupBox("Claude Flow v86 Enterprise Settings")
    v86_group.setStyleSheet("""
        QGroupBox {
            font-weight: bold;
            border: 2px solid #3498db;
            border-radius: 5px;
            margin-top: 10px;
            padding-top: 10px;
        }
        QGroupBox::title {
            subcontrol-origin: margin;
            left: 10px;
            padding: 0 5px 0 5px;
        }
    """)

    v86_layout = QGridLayout()

    # Memory Settings
    memory_label = QLabel("Memory Backend:")
    self.v86_memory_backend = QComboBox()
    self.v86_memory_backend.addItems(["sqlite", "postgresql", "memory"])
    self.v86_memory_backend.setCurrentText("sqlite")

    self.v86_memory_enabled = QCheckBox("Enable Persistent Memory")
    self.v86_memory_enabled.setChecked(True)
    self.v86_memory_enabled.setToolTip(
        "Enables cross-session memory persistence using SQLite database"
    )

    memory_cache_label = QLabel("Cache Size (MB):")
    self.v86_memory_cache = QSpinBox()
    self.v86_memory_cache.setRange(10, 1000)
    self.v86_memory_cache.setValue(200)

    v86_layout.addWidget(memory_label, 0, 0)
    v86_layout.addWidget(self.v86_memory_backend, 0, 1)
    v86_layout.addWidget(self.v86_memory_enabled, 0, 2)

```



```
v86_layout.addWidget(memory_cache_label, 0, 3)
v86_layout.addWidget(self.v86_memory_cache, 0, 4)

# Neural Features
self.v86_neural_enabled = QCheckBox("Enable Neural Models (27 models)")
self.v86_neural_enabled.setChecked(True)
self.v86_neural_enabled.setToolTip(
    "Activates 27 cognitive models with WASM SIMD acceleration"
)

neural_models_label = QLabel("Active Models:")
self.v86_neural_models = QSpinBox()
self.v86_neural_models.setRange(1, 27)
self.v86_neural_models.setValue(27)

v86_layout.addWidget(self.v86_neural_enabled, 1, 0, 1, 2)
v86_layout.addWidget(neural_models_label, 1, 2)
v86_layout.addWidget(self.v86_neural_models, 1, 3)

# Fault Tolerance
self.v86_bft_enabled = QCheckBox("Byzantine Fault Tolerance")
self.v86_bft_enabled.setChecked(True)
self.v86_bft_enabled.setToolTip(
    "Enterprise-grade fault tolerance for mission-critical operations"
)

fault_strategy_label = QLabel("Recovery Strategy:")
self.v86_fault_strategy = QComboBox()
self.v86_fault_strategy.addItem(
    "retry-with-learning",
    "failover",
    "circuit-breaker"
)

v86_layout.addWidget(self.v86_bft_enabled, 2, 0, 1, 2)
v86_layout.addWidget(fault_strategy_label, 2, 2)
v86_layout.addWidget(self.v86_fault_strategy, 2, 3, 1, 2)

# Hooks Configuration
self.v86_hooks_enabled = QCheckBox("Enable Lifecycle Hooks (14 types)")
self.v86_hooks_enabled.setChecked(True)
self.v86_hooks_enabled.setToolTip(
    "Enables all 14 lifecycle management hooks for automation"
)

# Hook selection
hooks_label = QLabel("Active Hooks:")
```

```

self.v86_hooks_list = QTextEdit()
self.v86_hooks_list.setMaximumHeight(60)
self.v86_hooks_list.setPlainText(
    "pre-task, post-task, pre-edit, post-edit, pre-command, "
    "post-command, session-start, session-end"
)

v86_layout.addWidget(self.v86_hooks_enabled, 3, 0, 1, 2)
v86_layout.addWidget(hooks_label, 4, 0)
v86_layout.addWidget(self.v86_hooks_list, 4, 1, 1, 4)

# Telemetry
self.v86_telemetry_enabled = QCheckBox("Enable Real-time Telemetry")
self.v86_telemetry_enabled.setChecked(True)

self.v86_token_tracking = QCheckBox("Token Usage Tracking")
self.v86_token_tracking.setChecked(True)

self.v86_cost_analysis = QCheckBox("Cost Analysis")
self.v86_cost_analysis.setChecked(True)

v86_layout.addWidget(self.v86_telemetry_enabled, 5, 0, 1, 2)
v86_layout.addWidget(self.v86_token_tracking, 5, 2, 1, 2)
v86_layout.addWidget(self.v86_cost_analysis, 5, 4)

# Migration Tools
migration_label = QLabel("Configuration Migration:")
self.migrate_button = QPushButton("Migrate Old Configs")
self.migrate_button.clicked.connect(self._migrate_configs)

self.validate_button = QPushButton("Validate v86 Config")
self.validate_button.clicked.connect(self._validate_v86_config)

v86_layout.addWidget(migration_label, 6, 0)
v86_layout.addWidget(self.migrate_button, 6, 1, 1, 2)
v86_layout.addWidget(self.validate_button, 6, 3, 1, 2)

v86_group.setLayout(v86_layout)

return v86_group

def _migrate_configs(self):
    """Trigger config migration"""
    from ..tools.migrate_configs import ConfigMigrator

# Get config directory
config_dir = Path(self.project_path_input.text()) / ".claude-flow" / "saved-configs"

```

```

if config_dir.exists():
    migrator = ConfigMigrator()
    migrator.batch_migrate(str(config_dir))

    self.show_message("Success",
                      f"Configs migrated successfully!\nCheck {config_dir} for v86 versions.")
else:
    self.show_message("Error", "Config directory not found!")

def _validate_v86_config(self):
    """Validate current v86 configuration"""
    import jsonschema

    # Generate current config
    from ..converters.v86_converter import V86ConfigConverter

    converter = V86ConfigConverter()
    config = converter.generate_v86_config(
        selected_agents=self.get_selected_agents(),
        preset_name=self.agent_preset_combo.currentText(),
        task=self.task_input.toPlainText(),
        project_path=self.project_path_input.text()
    )

    # Load schema
    schema_path = Path("src/Agents_Configuration/schema/v86.schema.json")

    try:
        if schema_path.exists():
            with open(schema_path) as f:
                schema = json.load(f)

            jsonschema.validate(config, schema)
            self.show_message("Success", "✅ Configuration is valid for v86!")
        else:
            self.show_message("Warning", "Schema file not found, skipping validation")

    except jsonschema.ValidationError as e:
        self.show_message("Validation Error", f"❌ Config validation failed:\n{str(e)}")

```

6. Dokumentation und Deployment

6.1 Benutzer-Dokumentation

Neue Datei: `docs/V86_INTEGRATION.md`

markdown

Claude Flow v86 Integration Guide

Übersicht

Die Integration von claude-flow@alpha v86 in die AI Coding Suite ermöglicht die nahtlose Nutzung von Enterprise-Gr

Quick Start

1. Konfiguration vorbereiten

1. ****Agenten auswählen****: Wählen Sie im Agent Configuration Tab die gewünschten Agenten
2. ****Preset wählen****: Nutzen Sie ein vordefiniertes Preset oder erstellen Sie eine Custom-Konfiguration
3. ****Task definieren****: Geben Sie eine klare Aufgabenbeschreibung ein
4. ****v86-Features konfigurieren****: Aktivieren Sie gewünschte Enterprise-Features

2. Hive Mind starten

Klicken Sie auf "Launch Hive Mind" um:

- v86-kompatible Konfigurationen zu generieren
- Erforderliche Verzeichnisstruktur zu erstellen
- Agenten-Definitionen zu exportieren
- WSL-Umgebung zu initialisieren

3. Generierte Dateien

Bei jedem Launch werden folgende Dateien erstellt/aktualisiert:

```
project-root/
├── claude-flow.config.json  # Haupt-v86-Konfiguration
├── .claude/
│   ├── settings.json      # Claude-spezifische Einstellungen
│   ├── mcp.json           # MCP-Server-Konfiguration
│   └── agents/
│       ├── queen.json     # Queen-Agent-Definition
│       ├── backend-dev.json  # Backend-Developer-Definition
│       └── [weitere].json  # Weitere Agenten-Definitionen
├── .swarm/
│   └── memory.db          # Persistente Memory-Datenbank
└── CLAUDE.md              # Kontext-Dokument für AI
```

Konfigurationspersistenz

Das System erkennt automatisch bestehende v86-Konfigurationen:

- Bei vorhandener `claude-flow.config.json` wird nur die Task aktualisiert
- Agenten-Setup bleibt über Sessions erhalten
- Memory-Datenbank speichert Cross-Session-Informationen

Migration bestehender Konfigurationen

Automatische Migration

```
```bash
```

```
python src/claude_flow_gui/tools/migrate_configs.py --batch .claude-flow/saved-configs/
```

## Einzelne Datei migrieren

```
bash
```

```
python src/claude_flow_gui/tools/migrate_configs.py old_config.json new_v86_config.json
```

## v86-spezifische Features

### Persistent Memory

- SQLite-basierte Speicherung
- 12 spezialisierte Tabellen
- Cross-Session-Datenpersistenz
- 200MB Standard-Cache

### Neural Processing

- 27 kognitive Modelle
- WASM SIMD-Beschleunigung
- Automatisches Training
- Mustererkennung

### Byzantine Fault Tolerance

- Enterprise-Grade Fehlertoleranz
- Automatische Wiederherstellung
- Gesundheitsüberwachung
- Retry-with-Learning Strategie

## Lifecycle Hooks

- 14 verschiedene Hook-Typen
- Automatisierte Workflows
- Pre/Post-Task-Verarbeitung
- Session-Management

## Kommandozeilen-Nutzung

### Direkter v86-Aufruf

```
bash

npx claude-flow@alpha hive-mind spawn "Your task" \
--config ./claude-flow.config.json \
--claude \
--verbose \
--memory persistent \
--hooks enabled \
--telemetry enabled
```

## Monitoring

```
bash

Aktive Agenten überwachen
npx claude-flow@alpha swarm monitor

Memory-Status prüfen
npx claude-flow@alpha memory status

Telemetrie anzeigen
npx claude-flow@alpha telemetry show
```

## Troubleshooting

### Problem: Konfiguration wird nicht gefunden

**Lösung:** Überprüfen Sie, ob der Projektpfad korrekt gesetzt ist und `.claude-flow/saved-configs/` existiert

### Problem: Agenten werden nicht geladen

**Lösung:** Validieren Sie die JSON-Dateien in `src/Agents_Configuration/agents/`

### Problem: Memory-Fehler

**Lösung:** Stellen Sie sicher, dass `.swarm/` Schreibrechte hat und genügend Speicherplatz vorhanden ist

## Problem: WSL-Befehle schlagen fehl

**Lösung:** Überprüfen Sie die WSL-Installation und ob `claude-flow@alpha v86` installiert ist:

```
bash
```

```
wsl bash -c "npx claude-flow@alpha --version"
```

## Performance-Optimierung

### Empfohlene Einstellungen

- **Kleine Projekte** (< 50 Dateien): 3-5 Agenten, hierarchische Topologie
- **Mittlere Projekte** (50-200 Dateien): 5-8 Agenten, mesh Topologie
- **Große Projekte** (> 200 Dateien): 8-12 Agenten, star Topologie

### Token-Optimierung

- Aktivieren Sie Token-Tracking für Kostenanalyse
- Nutzen Sie `maxFilesPerOperation` zur Begrenzung
- Setzen Sie angemessene `truthThreshold`-Werte

## Erweiterte Konfiguration

### Custom Agent Definition

```
json
```

```
{
 "id": "custom-agent",
 "name": "Custom Specialist",
 "category": "specialized",
 "capabilities": {
 "languages": ["python", "rust"],
 "tools": ["custom-tool-1", "custom-tool-2"]
 },
 "verification": {
 "methods": ["custom-verify"],
 "truthThreshold": 0.92,
 "maxFilesPerOperation": 15
 },
 "prompts": {
 "initialization": "You are a specialized agent for...",
 "taskTemplate": "Execute {task} with focus on..."
 }
}
```

## Custom Preset Creation

```
json

{
 "id": "custom-preset",
 "name": "🎯 Custom Team",
 "description": "Specialized team for specific tasks",
 "topology": "mesh",
 "agents": [
 {"id": "queen", "required": true, "model": "opus-4.1"},
 {"id": "custom-agent", "required": true, "model": "sonnet-4"}
],
 "orchestration": {
 "maxAgents": 6,
 "strategy": "parallel"
 }
}
```

## API-Referenz

### V86ConfigConverter

```
python
```



```
converter = V86ConfigConverter(agents_config_path="src/Agents_Configuration")
```

```
Generate v86 config
```

```
config = converter.generate_v86_config(
 selected_agents=["queen", "backend-dev"],
 preset_name="python-development",
 task="Build REST API",
 project_path="/path/to/project"
)
```

```
Create individual agent configs
```

```
converter.create_agent_configs(
 selected_agents=["queen", "backend-dev"],
 project_path="/path/to/project"
)
```

## ConfigMigrator

```
python
```

```
migrator = ConfigMigrator()
```

```
Migrate single config
```









```
v86_config = migrator.migrate_saved_config_to_v86(
 input_path="old_config.json",
 output_path="new_v86_config.json"
)
```




```
Batch migrate
```

```
migrator.batch_migrate(config_dir=".claude-flow/saved-configs")
```

## Changelog

### Version 1.0.0 (Initial v86 Integration)

-  Full v86 configuration support
-  64 agent definitions
-  87 MCP tools integration
-  Persistent memory with SQLite
-  27 neural models with WASM SIMD
-  Byzantine Fault Tolerance
-  14 lifecycle hooks
-  Real-time telemetry






-  Automatic migration tools
-  Schema validation
-  WSL integration

## Support




Bei Fragen oder Problemen:

- Öffnen Sie ein Issue im GitHub Repository
- Konsultieren Sie die [claude-flow Dokumentation](#)
- Nutzen Sie den Validate-Button zur Konfigurationsprüfung

## ## 7. Implementierungs-Timeline






| Phase | Aufgabe                                | Dauer  | Status                                                                            | Abhängigkeiten |
|-------|----------------------------------------|--------|-----------------------------------------------------------------------------------|----------------|
| ----- | -----                                  | -----  | -----                                                                             | -----          |
| **1** | JSON-Migration der Agentendefinitionen | 2 Tage |  | -              |
| **2** | V86 Converter Implementation           | 3 Tage |  | Phase 1        |
| **3** | Hive Launch Anpassungen                | 2 Tage |  | Phase 2        |
| **4** | WSL Bridge Updates                     | 1 Tag  |  | Phase 3        |
| **5** | Testing & Validierung                  | 2 Tage |  | Phase 4        |
| **6** | UI-Anpassungen                         | 1 Tag  |  | Phase 2        |
| **7** | Dokumentation                          | 1 Tag  |  | Phase 5        |

### ### Legende






-  Abgeschlossen
-  In Bearbeitung
-  Ausstehend

## ## 8. Kritische Erfolgsfaktoren

### ### Technische Anforderungen

-  v86-kompatible JSON-Generierung
-  Korrekte Verzeichnisstruktur
-  Schema-Validierung
-  WSL-Integration
-  Persistente Memory-Unterstützung

### ### Qualitätssicherung

-  Automatisierte Tests
-  Migration bestehender Configs
-  Fehlerbehandlung
-  Logging und Monitoring
-  Dokumentation

### ### Performance-Ziele

- Konfigurationsgenerierung < 1 Sekunde
- Migration von 100 Configs < 30 Sekunden
- Memory-Datenbank-Zugriff < 100ms
- WSL-Command-Ausführung < 5 Sekunden

## ## Zusammenfassung

Diese Implementierung ermöglicht die vollständige Integration von `claude-flow@alpha v86` in die AI Coding Suite. Die Agentenkonfigurationen werden nahtlos vom Python-basierten System in das v86-JSON-Format konvertiert, wobei alle Enterprise-Features wie Byzantine Fault Tolerance, persistente Memory und neuronale Modelle unterstützt werden.

Die Migration bestehender Konfigurationen erfolgt automatisiert, und das System erkennt und nutzt vorhandene v86-Konfigurationen intelligent. Durch die umfassende Dokumentation und Test-Suite ist eine reibungslose Integration und Wartung gewährleistet.