Claude-Flow Task-Completion Guide

Vollständige Abarbeitung von Teilaufgaben sicherstellen

Version 1.0 | Claude-Flow v2.0.0-alpha.86

Inhaltsverzeichnis

Problembeschreibung

Das Kernproblem

Bei der Ausführung komplexer Tasks mit Claude-Flow kann es vorkommen, dass nicht alle Teilaufgaben vollständig abgearbeitet werden. Dies äußert sich typischerweise in:

- **Unvollständige Implementierung**: Nur 60-80% der gewünschten Features werden umgesetzt
- **Vergessene Subtasks**: Kleinere, aber wichtige Aufgaben werden übersehen
- **Sequenzielle Abhängigkeiten**: Später Tasks werden übersprungen, wenn frühere zu lange dauern
- **Context-Verlust**: Bei langen Tasks geht der Überblick über alle Anforderungen verloren

Typische Szenarien

```bash
# Beispiel: Unvollständige Ausführung
User: "Build complete authentication system with registration, login, password reset, 2FA"
Result: ✅ Registration implementiert
        ✅ Login teilweise implementiert
        ❌ Password Reset vergessen
        ❌ 2FA nicht implementiert
```

Auswirkungen

- **Nacharbeit erforderlich**: Mehrere zusätzliche Sessions nötig
- **Inkonsistente Codebasis**: Teilweise implementierte Features
- **Zeit-Ineffizienz**: Wiederholtes Erklären des Kontexts
- **Frustration**: Erwartungen werden nicht erfüllt

Ursachenanalyse

1. Token-Limitierungen

Claude hat Context-Window-Beschränkungen, die bei langen Tasks zu Problemen führen:

```javascript
// Context-Window-Auslastung
const contextUsage = {
  codeGeneration: "40-50%",     // Generierter Code
  fileReading: "20-30%",        // Gelesene Dateien
  conversation: "15-20%",       // Bisheriger Dialog
  systemPrompts: "10-15%",      // System-Instruktionen
  remaining: "5-10%"            // Verfügbarer Puffer
}
```

2. Agenten-Fokus-Drift

Agenten können während der Ausführung vom ursprünglichen Ziel abweichen:

```
Start: "Build auth system with 4 features"
  ↓
Agent fokussiert auf Feature 1
  ↓
Deep-Dive in Optimierungen
  ↓
Feature 2 nur oberflächlich
  ↓
Feature 3-4 vergessen
```

3. Fehlende Struktur

Unstrukturierte Task-Beschreibungen führen zu unvollständiger Abarbeitung:

```bash
```

```
# Schlecht strukturiert
"Build auth system with user registration login password reset 2FA email verification rate limiting captcha"

# Gut strukturiert
"Build auth system:
1. User registration with email validation
2. JWT-based login
3. Password reset flow
4. 2FA with TOTP
Complete ALL 4 tasks in order."
```

4. Session-Unterbrechungen

Bei Unterbrechungen geht Kontext verloren:

- Memory wird nicht richtig persistiert
- Agenten starten ohne vorherigen Kontext
- Teilaufgaben-Status geht verloren

Lösungsstrategien

Strategie 1: Explizite Task-Strukturierung

Konzept

Teile komplexe Tasks in klare, atomare Einheiten mit expliziten Checkpoints.

Implementierung

bash

```
# Template für strukturierte Tasks
npx claude-flow@alpha swarm "
OBJECTIVE: Build complete authentication system

MANDATORY TASKS (must complete ALL):
□ Task 1: User registration endpoint with validation
□ Task 2: Login endpoint with JWT tokens
□ Task 3: Password reset with email flow
□ Task 4: 2FA implementation with TOTP

APPROACH:
1. Create TODO.md with all tasks
2. Implement each task sequentially
3. Mark completed in TODO.md
4. Validate all checkboxes are checked before finishing

IMPORTANT: Do NOT consider the task complete until ALL boxes are checked.
" --claude
```

Vorteile

- ✅ Klare Erfolgs-Kriterien
- ✅ Visueller Fortschritt
- ✅ Einfache Validierung

Nachteile

- ⚠️ Längere initiale Prompts
- ⚠️ Weniger Flexibilität

Strategie 2: Memory-basierte Kontinuität

Konzept

Nutze das SQLite Memory System für persistente Task-Verfolgung über Sessions.

Implementierung

```
bash
```

```bash
#!/bin/bash
# memory-based-completion.sh

PROJECT="auth-system"
NAMESPACE="auth-v1"

# Phase 1: Initialisierung mit Task-Breakdown
npx claude-flow@alpha hive-mind spawn "
1. Create task breakdown for authentication system
2. Store each subtask in memory with status='pending'
3. Create memory entries:
   - task_1: registration (pending)
   - task_2: login (pending)
   - task_3: password_reset (pending)
   - task_4: 2fa (pending)
" --namespace $NAMESPACE --claude

# Phase 2: Iterative Abarbeitung
for i in {1..4}; do
  echo "=== Processing Task $i ==="

  npx claude-flow@alpha swarm "
  1. Query memory for tasks with status='pending'
  2. Take the first pending task
  3. Implement it completely
  4. Update task status to 'completed' in memory
  5. Store implementation details in memory
  " --namespace $NAMESPACE --continue-session --claude

  # Status-Check
  npx claude-flow@alpha memory query "task_" --namespace $NAMESPACE

  # Kurze Pause zwischen Tasks
  sleep 2
done

# Phase 3: Validierung
npx claude-flow@alpha swarm "
1. Query all tasks from memory
2. Verify each has status='completed'
3. If any pending tasks remain, complete them now
4. Generate completion report
" --namespace $NAMESPACE --continue-session --claude
```

Memory-Schema für Task-Tracking

```javascript
// Beispiel Memory-Einträge
const taskMemory = {
  "task_1_registration": {
    status: "completed",
    implementedFiles: ["auth/register.py", "tests/test_register.py"],
    completedAt: "2024-08-18T10:30:00Z",
    validationPassed: true
  },
  "task_2_login": {
    status: "in_progress",
    implementedFiles: ["auth/login.py"],
    startedAt: "2024-08-18T10:45:00Z",
    blockers: ["JWT library not configured"]
  },
  "task_3_password_reset": {
    status: "pending",
    priority: "high",
    dependencies: ["task_2_login"]
  }
}
```

Vorteile

- ✅ Persistenz über Sessions
- ✅ Detailliertes Tracking
- ✅ Wiederaufnahme möglich

Nachteile

- ⚠️ Komplexeres Setup
- ⚠️ Memory-Management erforderlich

Strategie 3: Workflow-Orchestrierung

Konzept

Definiere Workflows in JSON mit automatischer Validierung und Fortsetzung.

Implementierung

```json
```

```
// workflows/complete-auth-system.json
{
  "version": "1.0",
  "workflow": {
    "name": "Complete Authentication System",
    "description": "Ensures all auth components are implemented",
    "settings": {
      "autoRetry": true,
      "maxRetries": 3,
      "continueOnError": false,
      "validateEachPhase": true
    },
    "phases": [
      {
        "id": "planning",
        "name": "Task Planning",
        "description": "Create detailed implementation plan",
        "tasks": [
          "Analyze requirements",
          "Create file structure",
          "Define API endpoints",
          "Create TODO.md"
        ],
        "validation": {
          "files": ["TODO.md", "docs/api-spec.md"],
          "required": true
        },
        "timeout": "10m"
      },
      {
        "id": "registration",
        "name": "User Registration",
        "description": "Complete registration system",
        "requires": ["planning"],
        "tasks": [
          "Create user model",
          "Implement registration endpoint",
          "Add email validation",
          "Write unit tests",
          "Test manually"
        ],
        "validation": {
          "files": [
            "models/user.py",
            "api/auth/register.py",
            "tests/test_registration.py"
```

```
      ],
      "tests": "pytest tests/test_registration.py",
      "required": true
    },
    "timeout": "20m"
  },
  {
    "id": "login",
    "name": "Login System",
    "description": "JWT-based authentication",
    "requires": ["registration"],
    "tasks": [
      "Implement login endpoint",
      "Generate JWT tokens",
      "Add refresh token logic",
      "Implement logout",
      "Write tests"
    ],
    "validation": {
      "files": [
        "api/auth/login.py",
        "api/auth/logout.py",
        "utils/jwt_handler.py",
        "tests/test_login.py"
      ],
      "tests": "pytest tests/test_login.py",
      "required": true
    },
    "timeout": "20m"
  },
  {
    "id": "password_reset",
    "name": "Password Reset",
    "description": "Email-based password reset",
    "requires": ["login"],
    "tasks": [
      "Create reset token model",
      "Implement forgot-password endpoint",
      "Implement reset-password endpoint",
      "Add email templates",
      "Write tests"
    ],
    "validation": {
      "files": [
        "api/auth/reset.py",
        "templates/reset_email.html",
        "tests/test_reset.py"
```

```
      ],
      "tests": "pytest tests/test_reset.py",
      "required": true
    },
    "timeout": "15m"
  },
  {
    "id": "two_factor",
    "name": "Two-Factor Authentication",
    "description": "TOTP-based 2FA",
    "requires": ["login"],
    "tasks": [
      "Add 2FA fields to user model",
      "Implement 2FA setup endpoint",
      "Implement 2FA verification",
      "Generate QR codes",
      "Write tests"
    ],
    "validation": {
      "files": [
        "api/auth/two_factor.py",
        "utils/totp_handler.py",
        "tests/test_2fa.py"
      ],
      "tests": "pytest tests/test_2fa.py",
      "required": true
    },
    "timeout": "20m"
  },
  {
    "id": "integration",
    "name": "Integration Testing",
    "description": "Full system validation",
    "requires": ["registration", "login", "password_reset", "two_factor"],
    "tasks": [
      "Run all unit tests",
      "Run integration tests",
      "Test complete user flow",
      "Generate test coverage report"
    ],
    "validation": {
      "tests": "pytest --cov=auth --cov-report=html",
      "minCoverage": 80,
      "required": true
    },
    "timeout": "10m"
  }
```

```
  ],
  "hooks": {
    "onPhaseStart": "echo 'Starting phase: ${PHASE_NAME}'",
    "onPhaseComplete": "echo 'Completed: ${PHASE_NAME}' >> completion.log",
    "onPhaseError": "echo 'Error in: ${PHASE_NAME}' >> error.log",
    "onWorkflowComplete": "./scripts/generate-report.sh"
  }
 }
}
```

Workflow-Ausführung

bash

```bash
#!/bin/bash
# execute-workflow.sh

WORKFLOW_FILE="workflows/complete-auth-system.json"

# Option 1: Automatische Ausführung
npx claude-flow@alpha workflow execute $WORKFLOW_FILE \
  --claude \
  --auto-continue \
  --verbose

# Option 2: Schritt-für-Schritt mit Validierung
npx claude-flow@alpha workflow execute $WORKFLOW_FILE \
  --phase planning \
  --validate-before-continue \
  --claude

# Nächste Phase
npx claude-flow@alpha workflow continue $WORKFLOW_FILE \
  --claude

# Status prüfen
npx claude-flow@alpha workflow status $WORKFLOW_FILE
```

Vorteile

- ✅ Vollautomatisierung möglich
- ✅ Klare Validierung
- ✅ Wiederverwendbar

Nachteile

- ⚠️ Initiale Erstellung aufwändig
- ⚠️ Weniger flexibel

Strategie 4: Validierungs-Driven Development

Konzept

Erstelle Validierungs-Checkpoints, die erfüllt sein müssen.

Implementierung

bash

```bash
#!/bin/bash
# validation-driven-completion.sh

# Schritt 1: Task mit Validierungs-Anforderungen
npx claude-flow@alpha swarm "
Build authentication system.

SUCCESS CRITERIA (all must pass):
1. File exists: api/auth/register.py
2. File exists: api/auth/login.py
3. File exists: api/auth/reset.py
4. File exists: api/auth/2fa.py
5. Test passes: pytest tests/test_auth.py
6. Coverage > 80%: pytest --cov=auth
7. No linting errors: flake8 api/auth/
8. API docs exist: docs/api.md

Implement features until ALL criteria pass.
" --claude

# Schritt 2: Validierungs-Loop
while true; do
  # Validierung ausführen
  npx claude-flow@alpha swarm "
  Run validation checks:
  1. Check if all required files exist
  2. Run all tests
  3. Check code coverage
  4. Run linter

  If any validation fails:
  - Identify what's missing
  - Implement missing parts
  - Re-run validation

  If all validations pass:
  - Create SUCCESS.md with all check results
  - Exit
  " --continue-session --claude

  # Prüfen ob SUCCESS.md existiert
  if [ -f "SUCCESS.md" ]; then
    echo "✅ All validations passed!"
    break
  fi
```

```
  echo "⏳ Validation failed, continuing..."
  sleep 5
done
```

Validierungs-Klassen

```python
```

```python
# validation/auth_validator.py
class AuthSystemValidator:
    def __init__(self):
        self.checks = []

    def validate_all(self):
        results = {
            "files": self.check_required_files(),
            "tests": self.run_tests(),
            "coverage": self.check_coverage(),
            "linting": self.run_linting(),
            "api_docs": self.check_documentation(),
            "integration": self.test_integration()
        }

        self.generate_report(results)
        return all(results.values())

    def check_required_files(self):
        required = [
            "api/auth/register.py",
            "api/auth/login.py",
            "api/auth/reset.py",
            "api/auth/2fa.py",
            "models/user.py",
            "utils/jwt_handler.py"
        ]

        missing = [f for f in required if not os.path.exists(f)]

        if missing:
            print(f"❌ Missing files: {missing}")
            return False

        print("✅ All required files exist")
        return True

    def run_tests(self):
        result = subprocess.run(
            ["pytest", "tests/", "-v"],
            capture_output=True
        )
        return result.returncode == 0

    def check_coverage(self):
        result = subprocess.run(
```

```
        ["pytest", "--cov=auth", "--cov-report=term"],
        capture_output=True,
        text=True
    )

    # Parse coverage percentage
    for line in result.stdout.split('\n'):
        if 'TOTAL' in line:
            coverage = int(line.split()[-1].rstrip('%'))
            return coverage >= 80

    return False
```

Vorteile

- ✅ Objektive Erfolgs-Kriterien
- ✅ Automatisierte Überprüfung
- ✅ Qualitätssicherung

Nachteile

- ⚠️ Validierungs-Overhead
- ⚠️ Kann zu rigide sein

Strategie 5: Multi-Agent-Spezialisierung

Konzept

Nutze spezialisierte Agenten für verschiedene Teilaufgaben.

Implementierung

```bash
```

```
#!/bin/bash
# multi-agent-completion.sh

# Schritt 1: Coordinator-Agent für Überblick
npx claude-flow@alpha agent spawn coordinator \
  --name "AuthProjectManager" \
  --capabilities "planning,delegation,validation"

# Schritt 2: Spezialisierte Worker
npx claude-flow@alpha agent spawn coder \
  --name "RegistrationDev" \
  --capabilities "python,fastapi,validation"

npx claude-flow@alpha agent spawn coder \
  --name "AuthenticationDev" \
  --capabilities "jwt,security,cryptography"

npx claude-flow@alpha agent spawn coder \
  --name "EmailDev" \
  --capabilities "smtp,templates,async"

npx claude-flow@alpha agent spawn tester \
  --name "AuthTester" \
  --capabilities "pytest,coverage,integration"

# Schritt 3: Task-Zuweisung durch Coordinator
npx claude-flow@alpha task create "Build complete auth system" \
  --assign "AuthProjectManager" \
  --subtasks "planning,registration,login,reset,2fa,testing"

# Schritt 4: Orchestrierung
npx claude-flow@alpha hive-mind spawn "
As AuthProjectManager:
1. Break down auth system into subtasks
2. Assign registration to RegistrationDev
3. Assign login/JWT to AuthenticationDev
4. Assign password reset to EmailDev
5. Assign testing to AuthTester
6. Monitor progress of all agents
7. Ensure ALL subtasks are completed
8. Generate final report
" --agents 5 --claude
```

Agent-Kommunikations-Matrix

```
javascript
```

```
// Agent-Interaktions-Konfiguration
const agentMatrix = {
  "AuthProjectManager": {
    role: "coordinator",
    communicatesWith: ["all"],
    responsibilities: [
      "task_breakdown",
      "assignment",
      "progress_tracking",
      "validation"
    ]
  },
  "RegistrationDev": {
    role: "worker",
    communicatesWith: ["AuthProjectManager", "AuthTester"],
    responsibilities: [
      "user_model",
      "registration_endpoint",
      "validation_logic"
    ]
  },
  "AuthenticationDev": {
    role: "worker",
    communicatesWith: ["AuthProjectManager", "AuthTester"],
    responsibilities: [
      "login_endpoint",
      "jwt_generation",
      "token_refresh",
      "2fa_implementation"
    ]
  },
  "EmailDev": {
    role: "worker",
    communicatesWith: ["AuthProjectManager", "RegistrationDev"],
    responsibilities: [
      "email_templates",
      "password_reset",
      "email_verification"
    ]
  },
  "AuthTester": {
    role: "validator",
    communicatesWith: ["all"],
    responsibilities: [
      "unit_tests",
      "integration_tests",
```

```
      "coverage_reports",
      "validation"
    ]
  }
}
```

Vorteile

- ✅ Parallelisierung möglich
- ✅ Spezialisierte Expertise
- ✅ Klare Verantwortlichkeiten

Nachteile

- ⚠️ Koordinations-Overhead
- ⚠️ Komplexere Fehlerbehandlung

Strategie 6: Checkpoint-basierte Fortsetzung

Konzept

Erstelle Checkpoints nach jeder Teilaufgabe für nahtlose Fortsetzung.

Implementierung

```bash
```

```bash
#!/bin/bash
# checkpoint-based-completion.sh

PROJECT="auth-system"
CHECKPOINT_DIR=".claude-flow/checkpoints"

# Checkpoint-Funktionen
create_checkpoint() {
  local name=$1
  local status=$2

  mkdir -p $CHECKPOINT_DIR

  # Status speichern
  echo "{
    \"name\": \"$name\",
    \"status\": \"$status\",
    \"timestamp\": \"$(date -Iseconds)\",
    \"files\": $(find . -name "*.py" -newer $CHECKPOINT_DIR/last.json 2>/dev/null | jq -R -s -c 'split("\n")[:-1]'),
    \"memory\": $(npx claude-flow@alpha memory export --namespace $PROJECT)
  }" > "$CHECKPOINT_DIR/${name}.json"

  # Als letzten Checkpoint markieren
  cp "$CHECKPOINT_DIR/${name}.json" "$CHECKPOINT_DIR/last.json"
}

restore_checkpoint() {
  local checkpoint=$1

  if [ -f "$CHECKPOINT_DIR/${checkpoint}.json" ]; then
    echo "Restoring checkpoint: $checkpoint"

    # Memory wiederherstellen
    local memory=$(jq -r '.memory' "$CHECKPOINT_DIR/${checkpoint}.json")
    npx claude-flow@alpha memory import --data "$memory" --namespace $PROJECT

    return 0
  else
    echo "Checkpoint not found: $checkpoint"
    return 1
  fi
}

# Hauptworkflow
echo "=== Starting Auth System Implementation ==="
```

```
# Task 1: Registration
npx claude-flow@alpha swarm "
Implement user registration:
1. Create user model
2. Registration endpoint
3. Email validation
4. Unit tests
Mark complete when done.
" --namespace $PROJECT --claude

create_checkpoint "registration" "completed"

# Task 2: Login
npx claude-flow@alpha swarm "
Implement login system:
1. Login endpoint
2. JWT generation
3. Token refresh
4. Logout endpoint
5. Tests
Continue from registration checkpoint.
" --namespace $PROJECT --continue-session --claude

create_checkpoint "login" "completed"

# Task 3: Password Reset
npx claude-flow@alpha swarm "
Implement password reset:
1. Forgot password endpoint
2. Reset token generation
3. Reset password endpoint
4. Email templates
5. Tests
Continue from login checkpoint.
" --namespace $PROJECT --continue-session --claude

create_checkpoint "password_reset" "completed"

# Task 4: 2FA
npx claude-flow@alpha swarm "
Implement 2FA:
1. TOTP setup
2. QR code generation
3. Verification endpoint
4. Backup codes
5. Tests
Continue from password_reset checkpoint.
```

```
" --namespace $PROJECT --continue-session --claude

create_checkpoint "2fa" "completed"

# Finale Validierung
npx claude-flow@alpha swarm "
Final validation:
1. Check all checkpoints completed
2. Run full test suite
3. Generate coverage report
4. Create completion summary
" --namespace $PROJECT --continue-session --claude

create_checkpoint "final" "validated"

echo "=== Auth System Complete ==="
```

Checkpoint-Recovery

bash

```
#!/bin/bash
# recover-from-checkpoint.sh

# Bei Fehler oder Unterbrechung
LAST_CHECKPOINT=$(ls -t .claude-flow/checkpoints/*.json | head -1 | xargs basename .json)

echo "Last checkpoint: $LAST_CHECKPOINT"

# Von letztem Checkpoint fortsetzen
restore_checkpoint $LAST_CHECKPOINT

npx claude-flow@alpha swarm "
Checkpoint recovered: $LAST_CHECKPOINT
Continue implementation from this point.
Check what's already done and complete remaining tasks.
" --namespace $PROJECT --continue-session --claude
```

Vorteile

- ✅ Robuste Wiederaufnahme
- ✅ Fortschritt gesichert
- ✅ Fehlertoleranz

Nachteile

- ⚠️ Zusätzlicher Storage
- ⚠️ Checkpoint-Management

Implementierungsmuster

Pattern 1: TODO-Driven Development

markdown

```
# TODO.md Template

## Authentication System Implementation

### Phase 1: Setup
- [ ] Project structure
- [ ] Dependencies installation
- [ ] Database configuration
- [ ] Environment variables

### Phase 2: Core Features
- [ ] User Registration
  - [ ] User model
  - [ ] Registration endpoint
  - [ ] Email validation
  - [ ] Unit tests
- [ ] Login System
  - [ ] Login endpoint
  - [ ] JWT generation
  - [ ] Token refresh
  - [ ] Logout
  - [ ] Unit tests
- [ ] Password Reset
  - [ ] Forgot password endpoint
  - [ ] Reset token model
  - [ ] Reset endpoint
  - [ ] Email templates
  - [ ] Unit tests
- [ ] Two-Factor Auth
  - [ ] TOTP implementation
  - [ ] QR code generation
  - [ ] Verification
  - [ ] Backup codes
  - [ ] Unit tests

### Phase 3: Integration
- [ ] Integration tests
- [ ] API documentation
- [ ] Performance tests
- [ ] Security audit

### Completion Criteria
- [ ] All unit tests passing
- [ ] Code coverage > 80%
- [ ] No linting errors
```

- [ ] API documentation complete
- [ ] Manual testing passed

Pattern 2: Progress Tracking

python

```python
# progress_tracker.py
import json
from datetime import datetime
from pathlib import Path

class TaskProgressTracker:
    def __init__(self, project_name):
        self.project_name = project_name
        self.progress_file = Path(f".claude-flow/progress/{project_name}.json")
        self.progress_file.parent.mkdir(parents=True, exist_ok=True)
        self.load_progress()

    def load_progress(self):
        if self.progress_file.exists():
            with open(self.progress_file) as f:
                self.progress = json.load(f)
        else:
            self.progress = {
                "project": self.project_name,
                "started": datetime.now().isoformat(),
                "tasks": {},
                "completed_count": 0,
                "total_count": 0
            }

    def add_task(self, task_id, description, dependencies=None):
        self.progress["tasks"][task_id] = {
            "description": description,
            "status": "pending",
            "dependencies": dependencies or [],
            "created": datetime.now().isoformat(),
            "started": None,
            "completed": None,
            "files": [],
            "tests": []
        }
        self.progress["total_count"] += 1
        self.save_progress()

    def start_task(self, task_id):
        if task_id in self.progress["tasks"]:
            self.progress["tasks"][task_id]["status"] = "in_progress"
            self.progress["tasks"][task_id]["started"] = datetime.now().isoformat()
            self.save_progress()

    def complete_task(self, task_id, files=None, tests=None):
```

```
        if task_id in self.progress["tasks"]:
            task = self.progress["tasks"][task_id]
            task["status"] = "completed"
            task["completed"] = datetime.now().isoformat()
            task["files"] = files or []
            task["tests"] = tests or []
            self.progress["completed_count"] += 1
            self.save_progress()

    def get_pending_tasks(self):
        return [
            (tid, task) for tid, task in self.progress["tasks"].items()
            if task["status"] == "pending"
        ]

    def get_next_task(self):
        pending = self.get_pending_tasks()

        # Finde Task ohne unerfüllte Dependencies
        for task_id, task in pending:
            deps = task.get("dependencies", [])
            if all(
                self.progress["tasks"].get(dep, {}).get("status") == "completed"
                for dep in deps
            ):
                return task_id, task

        return None, None

    def generate_report(self):
        report = f"""
# Progress Report: {self.project_name}

## Summary
- Total Tasks: {self.progress['total_count']}
- Completed: {self.progress['completed_count']}
- In Progress: {sum(1 for t in self.progress['tasks'].values() if t['status'] == 'in_progress')}
- Pending: {sum(1 for t in self.progress['tasks'].values() if t['status'] == 'pending')}
- Completion Rate: {self.progress['completed_count'] / max(self.progress['total_count'], 1) * 100:.1f}%

## Task Details
"""

        for task_id, task in self.progress["tasks"].items():
            status_emoji = {
                "completed": "✅",
                "in_progress": "🔄",
```

```
            "pending": " ⏳ "
        }.get(task["status"], " ❓ ")

        report += f"\n### {status_emoji} {task_id}: {task['description']}\n"
        report += f"- Status: {task['status']}\n"

        if task["started"]:
            report += f"- Started: {task['started']}\n"
        if task["completed"]:
            report += f"- Completed: {task['completed']}\n"
        if task["files"]:
            report += f"- Files: {', '.join(task['files'])}\n"
        if task["tests"]:
            report += f"- Tests: {', '.join(task['tests'])}\n"

    return report

def save_progress(self):
    with open(self.progress_file, 'w') as f:
        json.dump(self.progress, f, indent=2)
```

Pattern 3: Automated Continuation

bash

```bash
#!/bin/bash
# auto-continue.sh

MAX_ITERATIONS=10
ITERATION=0
PROJECT="auth-system"

# Fortsetzungs-Funktion
continue_if_incomplete() {
  local status=$(npx claude-flow@alpha swarm "
    Check TODO.md or task list.
    Return JSON: {\"incomplete_tasks\": number, \"tasks\": [list]}
  " --namespace $PROJECT --format json --claude)

  local incomplete=$(echo $status | jq -r '.incomplete_tasks')

  if [ "$incomplete" -gt 0 ]; then
    return 0  # Continue needed
  else
    return 1  # All complete
  fi
}

# Hauptloop
while [ $ITERATION -lt $MAX_ITERATIONS ]; do
  echo "=== Iteration $((ITERATION + 1)) ==="

  # Task ausführen
  npx claude-flow@alpha swarm "
    1. Check TODO.md for incomplete tasks
    2. Select next logical task
    3. Implement it completely
    4. Mark as complete in TODO.md
    5. Run tests for implemented feature
    6. Update progress report
  " --namespace $PROJECT --continue-session --claude

  # Prüfen ob weitere Tasks vorhanden
  if continue_if_incomplete; then
    echo "📋 Incomplete tasks remaining, continuing..."
    ITERATION=$((ITERATION + 1))
    sleep 5
  else
    echo "✅ All tasks completed!"
    break
  fi
```

```
done

if [ $ITERATION -eq $MAX_ITERATIONS ]; then
  echo "⚠️ Maximum iterations reached. Manual intervention required."
fi
```

Best Practices

1. Task-Strukturierung

DO ✅

```
bash

# Klare, nummerierte Aufgaben
npx claude-flow@alpha swarm "
Complete these tasks IN ORDER:
1. User registration with email validation
2. Login with JWT (depends on 1)
3. Password reset (depends on 2)
4. 2FA implementation (depends on 2)
IMPORTANT: Mark each complete before moving to next.
" --claude
```

DON'T ❌

```
bash

# Unstrukturierte Auflistung
npx claude-flow@alpha swarm "Build auth with registration login reset 2fa email jwt validation" --claude
```

2. Memory-Nutzung

DO ✅

```
bash

# Explizite Memory-Instruktionen
npx claude-flow@alpha swarm "
1. Store task list in memory as 'auth_tasks'
2. After each task, update status in memory
3. Before starting new task, check memory for progress
" --namespace auth --claude
```

DON'T ❌

```bash
# Keine Memory-Nutzung
npx claude-flow@alpha swarm "Build auth system" --claude
```

3. Validierung

DO ✅

```bash
# Eingebaute Validierung
npx claude-flow@alpha swarm "
Implement feature.
Then validate:
- All tests pass: pytest
- Coverage > 80%: pytest --cov
- No linting errors: flake8
If validation fails, fix and retry.
" --claude
```

DON'T ❌

```bash
# Keine Validierung
npx claude-flow@alpha swarm "Implement feature" --claude
```

4. Session-Management

DO ✅

```bash
# Konsistente Session-Nutzung
SESSION="auth-dev"
npx claude-flow@alpha hive-mind spawn "Task 1" --namespace $SESSION --claude
npx claude-flow@alpha swarm "Task 2" --namespace $SESSION --continue-session --claude
npx claude-flow@alpha swarm "Task 3" --namespace $SESSION --continue-session --claude
```

DON'T ❌

```bash
```

```
# Neue Sessions für jeden Task
npx claude-flow@alpha swarm "Task 1" --claude
npx claude-flow@alpha swarm "Task 2" --claude  # Kein Kontext von Task 1
```

5. Fehlerbehandlung

DO ✅

```
bash

# Robuste Fehlerbehandlung
npx claude-flow@alpha swarm "
Try to implement feature.
If error occurs:
1. Log error to errors.log
2. Store error context in memory
3. Attempt alternative approach
4. If still failing, document blocker
" --claude
```

DON'T ❌

```
bash

# Keine Fehlerbehandlung
npx claude-flow@alpha swarm "Implement feature" --claude
```

Workflow-Templates

Template 1: Microservice Development

```
bash
```

```
#!/bin/bash
# microservice-complete.sh

SERVICE_NAME="user-service"
NAMESPACE="microservice-$SERVICE_NAME"

# Workflow-Definition
cat > workflow.json << 'EOF'
{
  "service": "user-service",
  "tasks": [
    {
      "id": "api",
      "name": "API Endpoints",
      "subtasks": [
        "GET /users",
        "GET /users/{id}",
        "POST /users",
        "PUT /users/{id}",
        "DELETE /users/{id}"
      ]
    },
    {
      "id": "database",
      "name": "Database Layer",
      "subtasks": [
        "User model",
        "Migrations",
        "Seeders",
        "Indexes"
      ]
    },
    {
      "id": "business",
      "name": "Business Logic",
      "subtasks": [
        "Validation rules",
        "Business constraints",
        "Event handlers",
        "Notifications"
      ]
    },
    {
      "id": "tests",
      "name": "Testing",
      "subtasks": [
```

```
        "Unit tests",
        "Integration tests",
        "API tests",
        "Load tests"
      ]
    },
    {
      "id": "docs",
      "name": "Documentation",
      "subtasks": [
        "API documentation",
        "Database schema",
        "Business rules",
        "Deployment guide"
      ]
    }
  ]
}
EOF

# Ausführung
npx claude-flow@alpha swarm "
1. Read workflow.json
2. Create TODO.md from workflow
3. Implement each task completely
4. Check off completed items
5. Validate all subtasks are done
6. Generate completion report
" --namespace $NAMESPACE --claude

# Validierung
npx claude-flow@alpha swarm "
Validate microservice completeness:
- All endpoints implemented and tested
- Database fully configured
- Business logic covered by tests
- Documentation complete
Create VALIDATION_REPORT.md with results
" --namespace $NAMESPACE --continue-session --claude
```

Template 2: Frontend Component Library

bash

```bash
#!/bin/bash
# component-library-complete.sh

LIBRARY="ui-components"
NAMESPACE="lib-$LIBRARY"

# Component-Liste
COMPONENTS=(
  "Button"
  "Input"
  "Select"
  "Modal"
  "Table"
  "Card"
  "Navigation"
  "Footer"
)

# Für jede Komponente
for COMPONENT in "${COMPONENTS[@]}"; do
  echo "=== Building Component: $COMPONENT ==="

  npx claude-flow@alpha swarm "
  Build complete $COMPONENT component:

  Required Files:
  1. components/$COMPONENT/$COMPONENT.tsx - Main component
  2. components/$COMPONENT/$COMPONENT.styles.ts - Styles
  3. components/$COMPONENT/$COMPONENT.types.ts - TypeScript types
  4. components/$COMPONENT/$COMPONENT.test.tsx - Tests
  5. components/$COMPONENT/$COMPONENT.stories.tsx - Storybook
  6. components/$COMPONENT/README.md - Documentation

  Requirements:
  - Fully typed with TypeScript
  - Accessible (ARIA labels)
  - Responsive design
  - Dark mode support
  - 100% test coverage
  - Storybook stories with all variants

  Validate ALL files exist and requirements met before completing.
  " --namespace $NAMESPACE --continue-session --claude

  # Checkpoint nach jeder Komponente
  echo "{
```

```
  \"component\": \"$COMPONENT\",
  \"completed\": \"$(date -Iseconds)\"
 }" >> ".claude-flow/checkpoints/$LIBRARY.json"
done

# Finale Integration
npx claude-flow@alpha swarm "
Complete component library:
1. Create index.ts exporting all components
2. Generate library documentation
3. Create usage examples
4. Run full test suite
5. Build library bundle
6. Generate completion report
" --namespace $NAMESPACE --continue-session --claude
```

Template 3: Data Pipeline

bash

```
#!/bin/bash
# data-pipeline-complete.sh

PIPELINE="etl-pipeline"
NAMESPACE="pipeline-$PIPELINE"

# Pipeline-Stages definieren
npx claude-flow@alpha swarm "
Create complete ETL pipeline with these stages:

EXTRACTION STAGE:
□ Database connector (PostgreSQL)
□ API connector (REST)
□ File reader (CSV, JSON, XML)
□ Error handling for failed connections
□ Retry logic with exponential backoff
□ Unit tests for each connector

TRANSFORMATION STAGE:
□ Data validation
□ Data cleaning (nulls, duplicates)
□ Type conversion
□ Business rule application
□ Data enrichment
□ Aggregation functions
□ Unit tests for each transformation

LOADING STAGE:
□ Target database connector
□ Batch processing
□ Transaction management
□ Error recovery
□ Audit logging
□ Performance optimization
□ Integration tests

ORCHESTRATION:
□ Scheduler setup (Airflow/Celery)
□ Dependency management
□ Monitoring dashboard
□ Alert configuration
□ Documentation
□ End-to-end tests

Mark each item as complete.
Do not finish until ALL boxes are checked.
```

```
" --namespace $NAMESPACE --claude

# Validierung mit Metriken
npx claude-flow@alpha swarm "
Validate pipeline completeness:
1. Run extraction test with sample data
2. Verify transformation accuracy
3. Check loading performance
4. Test error scenarios
5. Measure throughput
6. Generate metrics report

Required metrics:
- Extraction success rate > 99%
- Transformation accuracy > 99.9%
- Loading speed > 1000 records/second
- Error recovery success > 95%
" --namespace $NAMESPACE --continue-session --claude
```

Troubleshooting

Problem: Agent vergisst Teilaufgaben

Symptome

- Nur erste 2-3 von 5+ Aufgaben werden implementiert
- Agent wechselt Fokus ohne Fertigstellung

Lösungen

```
bash
```

```
# Lösung 1: Explizite Erinnerungen
npx claude-flow@alpha swarm "
CRITICAL: You have 5 tasks. Current progress:
Task 1: ✅ Complete
Task 2: ✅ Complete
Task 3: ⏳ In Progress
Task 4: ❌ Not Started
Task 5: ❌ Not Started

Continue with Task 3. Do NOT finish until all 5 are complete.
" --continue-session --claude

# Lösung 2: Checkpoint-Validierung
npx claude-flow@alpha swarm "
Before considering work complete:
1. Count total required tasks
2. Count completed tasks
3. If completed < total, continue working
4. Only finish when completed == total
" --continue-session --claude
```

Problem: Context-Window-Überschreitung

Symptome

- Fehler bei langen Tasks
- Unvollständige Generierung
- Token-Limit-Warnungen

Lösungen

bash

```
# Lösung 1: Task-Splitting
# Statt einem großen Task
npx claude-flow@alpha swarm "Build complete app with 20 features" --claude

# Besser: Aufteilen
for i in {1..20}; do
  npx claude-flow@alpha swarm "
  Implement feature $i of 20.
  Keep implementation focused and concise.
  " --namespace app --continue-session --claude
done

# Lösung 2: Code-Kompression
npx claude-flow@alpha swarm "
After implementing each feature:
1. Remove unnecessary comments
2. Compress verbose code
3. Extract common functions
4. Clear unused imports
" --continue-session --claude
```

Problem: Inkonsistente Implementierung

Symptome

- Verschiedene Patterns in verschiedenen Teilen
- Inkompatible Interfaces
- Style-Inkonsistenzen

Lösungen

bash

```
# Lösung: Style-Guide etablieren
npx claude-flow@alpha swarm "
First, create STANDARDS.md with:
- Code style rules
- Naming conventions
- File structure
- Pattern library

Then implement all features following STANDARDS.md.
Refer to standards for every implementation.
" --claude

# Nachträgliche Vereinheitlichung
npx claude-flow@alpha swarm "
Refactor all code to follow STANDARDS.md:
1. Check each file against standards
2. Refactor non-compliant code
3. Ensure consistency across all modules
4. Run linter to verify
" --continue-session --claude
```

Problem: Fehlende Dependencies

Symptome

- Import-Fehler
- Undefined Functions
- Missing Modules

Lösungen

bash

```
# Lösung: Dependency-Check
npx claude-flow@alpha swarm "
Before implementing:
1. List all required dependencies
2. Add to requirements.txt or package.json
3. Install all dependencies
4. Verify imports work

Then proceed with implementation.
" --claude

# Automatische Dependency-Resolution
npx claude-flow@alpha swarm "
If import error occurs:
1. Identify missing package
2. Add to dependencies
3. Install package
4. Retry import
5. Document in README
" --continue-session --claude
```

Checklisten

Pre-Task Checklist

- [ ] **Task-Definition klar strukturiert?**
  - Nummerierte Liste
  - Klare Abhängigkeiten
  - Messbare Erfolgs-Kriterien
- [ ] **Memory-Strategy definiert?**
  - Namespace festgelegt
  - Progress-Tracking geplant
  - Checkpoint-Strategie
- [ ] **Validierung geplant?**
  - Test-Anforderungen definiert
  - Coverage-Ziele gesetzt
  - Validierungs-Checkpoints
- [ ] **Ressourcen vorbereitet?**
  - Dependencies installiert
  - Projekt-Struktur erstellt
  - Konfiguration vollständig

During-Task Checklist

- [ ] **Progress monitoren**
  - `hive-mind status` regelmäßig prüfen
  - Memory-Einträge verifizieren
  - Checkpoints erstellen
- [ ] **Bei Problemen eingreifen**
  - Explizite Fortsetzungs-Anweisungen
  - Kontext-Erinnerungen
  - Task-Refokussierung
- [ ] **Session-Kontinuität sichern**
  - `--continue-session` verwenden
  - Namespace konsistent halten
  - Memory exportieren

Post-Task Checklist

- [ ] **Vollständigkeit validieren**
  - Alle Teilaufgaben abgeschlossen?
  - Tests erfolgreich?
  - Dokumentation komplett?
- [ ] **Quality Assurance**
  - Code-Review durchgeführt?
  - Linting-Fehler behoben?
  - Performance akzeptabel?
- [ ] **Dokumentation**
  - README aktualisiert?
  - API-Docs generiert?
  - Changelog gepflegt?
- [ ] **Backup & Archivierung**
  - Memory exportiert?
  - Code committed?
  - Checkpoints gesichert?

Emergency Recovery Checklist

Wenn Task-Ausführung fehlschlägt:

- [ ] **Status Assessment**

```bash
npx claude-flow@alpha hive-mind status
npx claude-flow@alpha memory query --recent
```

- [ ] **Checkpoint Recovery**

```bash
ls -la .claude-flow/checkpoints/
# Letzten funktionierenden Checkpoint identifizieren
```

- [ ] **Memory Export**

```bash
npx claude-flow@alpha memory export --output emergency-backup.json
```

- [ ] **Sanfte Wiederaufnahme**

```bash
npx claude-flow@alpha swarm "
Check current state.
Identify what's complete and what's missing.
Continue from last successful point.
" --continue-session --claude
```

- [ ] **Aggressive Wiederaufnahme** (wenn sanft fehlschlägt)

```bash
npx claude-flow@alpha init --force
npx claude-flow@alpha memory import emergency-backup.json
npx claude-flow@alpha hive-mind spawn "Continue task" --restore --claude
```

Zusammenfassung

Die vollständige Abarbeitung von Teilaufgaben in Claude-Flow erfordert:

1. **Strukturierte Planung**: Klare Task-Definition mit messbaren Zielen
2. **Persistente Verfolgung**: Memory und Checkpoints für Kontinuität
3. **Aktive Validierung**: Regelmäßige Überprüfung des Fortschritts
4. **Robuste Fortsetzung**: Mechanismen für Wiederaufnahme
5. **Qualitätssicherung**: Eingebaute Tests und Validierung

Empfohlene Standard-Strategie

Für die meisten Projekte empfiehlt sich eine Kombination aus:

- **TODO-Driven Development** für Übersichtlichkeit
- **Memory-basierte Kontinuität** für Persistenz
- **Checkpoint-System** für Robustheit
- **Validierungs-Loops** für Qualität

Quick-Start-Command

```bash
# Optimaler Start für vollständige Task-Completion
npx claude-flow@alpha hive-mind spawn "
PROJECT: [Your Project Name]

APPROACH:
1. Create TODO.md with ALL required tasks
2. Store task list in memory
3. Implement each task completely
4. Mark done in TODO.md and memory
5. Create checkpoint after each task
6. Validate ALL tasks complete before finishing

CRITICAL: Do NOT finish until TODO.md shows 100% completion
" --namespace your-project --claude --verbose
```

Dokumentation Version 1.0 Erstellt für Claude-Flow v2.0.0-alpha.86 Stand: August 2025