Multi-Agent AI System - Detaillierter Umsetzungsplan

Vollständige Implementierungsanleitung für AI-Coding-Agenten

Datum: 26. September 2025
Version: 1.0 - Production Ready
Target: AI-Coding-Agenten (Windsurf, Cursor, Claude Code)
Gesamtdauer: 94 Tage (~13 Wochen)

Executive Summary für AI-Agenten

Dieser Umsetzungsplan ist speziell für die neuesten AI-Coding-Modelle optimiert und strukturiert das Multi-Agent AI System in 14 konkrete, ausführbare Tasks. Jeder Task ist mit spezifischen Model-Empfehlungen, detaillierten technischen Spezifikationen und klaren Deliverables versehen.

Empfohlene AI-Coding-Umgebung:

- **Primary IDE:** Windsurf (agentic workflows, Cascade-System)
- **Alternative:** Cursor mit Agent-Mode
- **Fallback:** Claude Code CLI für Terminal-basierte Tasks

Model-Verteilung:

- **GPT-5-Codex:** 35.7% (Orchestrierung, agentic workflows)
- **Claude-4-Opus:** 21.4% (komplexe Implementierungen, 72.5% SWE-bench)
- **QWEN3-Max:** Kosteneffiziente Alternative für Reasoning-intensive Tasks

Warum Windsurf + GPT-5-Codex die optimale Kombination ist

Windsurf IDE Vorteile für Multi-Agent-Entwicklung

Agentic Workflows Native:

- **Cascade-System:** Ermöglicht iterative, mehrstufige Entwicklungsprozesse
- **Tool Integration:** Native MCP (Model Context Protocol) Support
- **Context Awareness:** Automatisches Verständnis der gesamten Codebase
- **Real-time Collaboration:** AI und Developer arbeiten synchron

Von Gartner als Leader 2025 anerkannt:

- Teil der Cognition Suite (auch Devin-Entwickler)
- Speziell für Software Engineering Workflows optimiert
- Enterprise-grade Sicherheit und Compliance

GPT-5-Codex für Coordinator Agent

Spezialisierung auf Agentic Coding:

- **Extended Reasoning:** Kann 7+ Stunden autonom arbeiten
- **Multi-Step Orchestration:** Ideal für Koordination zwischen Agenten
- **51.3% Complex Refactoring Accuracy** vs. 33.9% bei GPT-5 Standard
- **Native Tool Integration:** Optimiert für automatisierte Workflows

Alternative Empfehlung:
Falls GPT-5-Codex nicht verfügbar: **Claude-4-Opus** (72.5% SWE-bench Score) oder **QWEN3-Max** (kosteneffizient, 1T Parameter)

Phase 1: Foundation Setup (22 Tage)

Task T1.1: Multi-Agent Framework Setup

AI-Model: GPT-5-Codex | **Fallback:** Claude-4-Sonnet | **Dauer:** 7 Tage | **Komplexität:** 8/10

Windsurf Setup-Anweisungen

~~~
## Windsurf Cascade Configuration für Multi-Agent Framework

### 1. Projekt-Initialisierung
- Erstelle neues Windsurf Projekt: "multi-agent-ai-system"
- Initialisiere Git Repository mit Clean Architecture Structure
- Konfiguriere .windsurfrules für Multi-Agent-spezifische Workflows

### 2. Framework Abstraction Layer
Implementiere austauschbare Multi-Agent-Framework-Architektur:

```python
# src/core/orchestration/abstract_orchestrator.py
from abc import ABC, abstractmethod
from typing import Dict, List, Any, Optional, Union
from dataclasses import dataclass
from enum import Enum

class OrchestratorType(Enum):
    LANGGRAPH = "langgraph"
    CREWAI = "crewai"
    AUTOGEN = "autogen"

@dataclass
class AgentConfig:
    agent_id: str
    agent_type: str
    primary_model: str
    fallback_models: List[str]
    capabilities: List[str]
    max_iterations: int = 10
    timeout_seconds: int = 300

@dataclass
~~~

```
class WorkflowDefinition:
    workflow_id: str
    name: str
    description: str
    agents: List[str]
    dependencies: Dict[str, List[str]]
    success_criteria: Dict[str, Any]
    failure_conditions: List[str]

class AbstractOrchestrator(ABC):
    """
    Abstraction layer for multi-agent orchestration frameworks.
    Enables seamless switching between LangGraph, CrewAI, and AutoGen.
    """

    def __init__(self, orchestrator_type: OrchestratorType):
        self.orchestrator_type = orchestrator_type
        self.agents: Dict[str, AgentConfig] = {}
        self.active_workflows: Dict[str, Any] = {}

    @abstractmethod
    async def initialize_agents(self, agent_configs: List[AgentConfig]) -&gt; Dict[str, bool
        """Initialize all configured agents"""
        pass

    @abstractmethod
    async def execute_workflow(self,
                              workflow: WorkflowDefinition,
                              input_data: Dict[str, Any],
                              context: Optional[Dict[str, Any]] = None) -&gt; Dict[str, Any]
        """Execute complete workflow with specified agents"""
        pass

    @abstractmethod
    async def add_agent(self, agent_config: AgentConfig) -&gt; str:
        """Dynamically add new agent to orchestration"""
        pass

    @abstractmethod
    async def remove_agent(self, agent_id: str) -&gt; bool:
        """Remove agent from orchestration"""
        pass

    @abstractmethod
    async def get_workflow_status(self, workflow_id: str) -&gt; Dict[str, Any]:
        """Get real-time status of running workflow"""
        pass

    @abstractmethod
    async def pause_workflow(self, workflow_id: str) -&gt; bool:
        """Pause running workflow"""
        pass

    @abstractmethod
    async def resume_workflow(self, workflow_id: str) -&gt; bool:
        """Resume paused workflow"""
        pass
```

3. LangGraph Implementation

```
# src/core/orchestration/langgraph_orchestrator.py
from langgraph import Graph, Node, Edge, State
from langgraph.memory import PersistentMemory
from .abstract_orchestrator import AbstractOrchestrator, AgentConfig, WorkflowDefinition

class LangGraphOrchestrator(AbstractOrchestrator):
    def __init__(self):
        super().__init__(OrchestratorType.LANGGRAPH)
        self.graph = Graph()
        self.memory = PersistentMemory(store="postgresql")
        self.node_registry: Dict[str, Node] = {}

    async def initialize_agents(self, agent_configs: List[AgentConfig]) -&gt; Dict[str, bool
        """Initialize LangGraph nodes for each agent"""
        results = {}

        for config in agent_configs:
            try:
                node = Node(
                    name=config.agent_id,
                    model=config.primary_model,
                    fallback_models=config.fallback_models,
                    capabilities=config.capabilities,
                    memory=self.memory
                )

                self.graph.add_node(node)
                self.node_registry[config.agent_id] = node
                self.agents[config.agent_id] = config
                results[config.agent_id] = True

            except Exception as e:
                print(f"Failed to initialize agent {config.agent_id}: {e}")
                results[config.agent_id] = False

        return results

    async def execute_workflow(self,
                              workflow: WorkflowDefinition,
                              input_data: Dict[str, Any],
                              context: Optional[Dict[str, Any]] = None) -&gt; Dict[str, Any]
        """Execute workflow using LangGraph's graph execution"""

        # Build execution graph based on workflow dependencies
        execution_graph = self._build_execution_graph(workflow)

        # Set up state management
        initial_state = State({
            "input": input_data,
            "context": context or {},
            "workflow_id": workflow.workflow_id,
            "results": {},
            "errors": []
        })

        try:
            # Execute with timeout and error handling
            result = await execution_graph.ainvoke(
                initial_state,
                timeout=workflow.get("timeout", 1800)  # 30 min default
            )

            self.active_workflows[workflow.workflow_id] = result
            return {
                "success": True,
                "workflow_id": workflow.workflow_id,
                "results": result.get("results", {}),
                "execution_time": result.get("execution_time"),
                "agents_used": result.get("agents_used", [])
            }

        except Exception as e:
            return {
                "success": False,
                "workflow_id": workflow.workflow_id,
                "error": str(e),
                "partial_results": initial_state.get("results", {})
            }

    def _build_execution_graph(self, workflow: WorkflowDefinition) -&gt; Graph:
        """Build LangGraph execution graph from workflow definition"""
        exec_graph = Graph()

        # Add nodes for each agent in workflow
        for agent_id in workflow.agents:
            if agent_id in self.node_registry:
                exec_graph.add_node(self.node_registry[agent_id])

        # Add edges based on dependencies
        for agent_id, dependencies in workflow.dependencies.items():
            for dep_id in dependencies:
                if dep_id in self.node_registry and agent_id in self.node_registry:
                    exec_graph.add_edge(
                        self.node_registry[dep_id],
                        self.node_registry[agent_id]
                    )

        return exec_graph
```

4. CrewAI Implementation

```
# src/core/orchestration/crewai_orchestrator.py
from crewai import Crew, Agent, Task, Process
from .abstract_orchestrator import AbstractOrchestrator, AgentConfig, WorkflowDefinition

class CrewAIOrchestrator(AbstractOrchestrator):
    def __init__(self):
        super().__init__(OrchestratorType.CREWAI)
        self.crew = None
        self.crewai_agents: Dict[str, Agent] = {}

    async def initialize_agents(self, agent_configs: List[AgentConfig]) -&gt; Dict[str, bool
        """Initialize CrewAI agents"""
        results = {}
        crewai_agent_list = []

        for config in agent_configs:
            try:
                agent = Agent(
                    role=self._get_agent_role(config.agent_type),
                    goal=self._get_agent_goal(config.agent_type),
                    backstory=self._get_agent_backstory(config.agent_type),
                    llm=config.primary_model,
                    tools=self._get_agent_tools(config.capabilities),
                    max_iter=config.max_iterations,
                    memory=True,
                    verbose=True
                )

                self.crewai_agents[config.agent_id] = agent
                crewai_agent_list.append(agent)
                self.agents[config.agent_id] = config
                results[config.agent_id] = True

            except Exception as e:
                print(f"Failed to initialize agent {config.agent_id}: {e}")
                results[config.agent_id] = False

        # Initialize crew with all agents
        if crewai_agent_list:
            self.crew = Crew(
                agents=crewai_agent_list,
                process=Process.hierarchical,  # or sequential
                memory=True,
                embedder={
                    "provider": "openai",
                    "config": {"model": "text-embedding-3-small"}
                }
            )

        return results

    def _get_agent_role(self, agent_type: str) -&gt; str:
        role_mapping = {
            "coordinator": "Project Coordinator and Workflow Orchestrator",
            "research": "Senior Research Analyst and Market Intelligence Expert",
            "implementation": "Senior Software Engineer and Code Architect",
            "testing": "Quality Assurance Engineer and Test Automation Specialist",
            "documentation": "Technical Writer and Documentation Specialist",
            "quality": "Code Review Expert and Quality Gate Enforcer",
            "security": "Security Engineer and Compliance Specialist",
            "debugging": "Debug Specialist and Performance Optimizer"
        }
        return role_mapping.get(agent_type, f"Specialized {agent_type.title()} Agent")
```

5. Configuration Management System

```
# src/core/config/orchestrator_config.py
import yaml
from typing import Dict, Any
from pathlib import Path

class OrchestratorConfigManager:
    def __init__(self, config_path: str = "config/orchestrator.yaml"):
        self.config_path = Path(config_path)
        self.config = self._load_config()

    def _load_config(self) -&gt; Dict[str, Any]:
        if not self.config_path.exists():
            return self._create_default_config()

        with open(self.config_path, 'r') as f:
            return yaml.safe_load(f)

    def _create_default_config(self) -&gt; Dict[str, Any]:
        default_config = {
            "orchestrator": {
                "framework": "langgraph",  # langgraph | crewai | autogen
                "fallback_framework": "crewai"
            },
            "agents": {
                "coordinator": {
                    "primary_model": "gpt-5-codex",
                    "fallback_models": ["qwen3-max", "claude-4.1-sonnet"],
                    "capabilities": ["orchestration", "planning", "quality_gates"],
                    "max_iterations": 15
                },
                "research": {
                    "primary_model": "perplexity-pro",
                    "fallback_models": ["microsoft-copilot", "gpt-5"],
                    "capabilities": ["web_search", "market_analysis", "citations"],
                    "max_iterations": 10
                },
                "implementation": {
                    "primary_model": "claude-4-opus",
                    "fallback_models": ["gpt-5-codex", "deepseek-coder-r1"],
                    "capabilities": ["coding", "refactoring", "architecture"],
                    "max_iterations": 20
                }
            }
        }

        # Save default config
        self.config_path.parent.mkdir(parents=True, exist_ok=True)
        with open(self.config_path, 'w') as f:
            yaml.dump(default_config, f, default_flow_style=False)

        return default_config

    def get_orchestrator_framework(self) -&gt; str:
        return self.config["orchestrator"]["framework"]

    def get_agent_config(self, agent_type: str) -&gt; Dict[str, Any]:
        return self.config["agents"].get(agent_type, {})

    def switch_framework(self, new_framework: str) -&gt; bool:
        valid_frameworks = ["langgraph", "crewai", "autogen"]
        if new_framework not in valid_frameworks:
            return False

        self.config["orchestrator"]["framework"] = new_framework
        self._save_config()
        return True
```

Deliverables T1.1

- [ ] Complete Abstraction Layer für Multi-Agent-Frameworks
- [ ] LangGraph Implementation mit Graph-based Workflows
- [ ] CrewAI Implementation mit Role-based Agents
- [ ] Configuration Management System
- [ ] Framework Switching Capabilities
- [ ] Unit Tests (>90% Coverage)
- [ ] Integration Tests für Framework-Switching
- [ ] API Documentation
- [ ] README mit Setup Instructions

Testing Requirements T1.1

```
# tests/test_orchestrator_switching.py
import pytest
from src.core.orchestration.orchestrator_factory import OrchestratorFactory

@pytest.mark.asyncio
async def test_framework_switching():
    """Test seamless switching between orchestration frameworks"""

    # Test LangGraph
    langgraph_orch = OrchestratorFactory.create_orchestrator("langgraph")
    assert langgraph_orch.orchestrator_type == OrchestratorType.LANGGRAPH
    # Test CrewAI
    crewai_orch = OrchestratorFactory.create_orchestrator("crewai")
    assert crewai_orch.orchestrator_type == OrchestratorType.CREWAI

    # Test same agent configuration works on both
    test_agents = [
        AgentConfig(
            agent_id="test_coordinator",
            agent_type="coordinator",
            primary_model="gpt-5-codex",
            fallback_models=["claude-4-sonnet"],
            capabilities=["orchestration"]
        )
    ]

    lg_result = await langgraph_orch.initialize_agents(test_agents)
    crew_result = await crewai_orch.initialize_agents(test_agents)

    assert lg_result["test_coordinator"] == True
    assert crew_result["test_coordinator"] == True
```

Task T1.2: AI Model Router Implementation

AI-Model: QWEN3-Max | **Fallback:** GPT-5-Codex | **Dauer:** 5 Tage | **Komplexität:** 9/10

Advanced Model Router mit ML-basierter Auswahl

```
# src/core/models/intelligent_model_router.py
import asyncio
import numpy as np
from typing import Dict, List, Any, Optional, Tuple
from dataclasses import dataclass
from datetime import datetime, timedelta
from sklearn.ensemble import RandomForestClassifier
import joblib

@dataclass
class ModelPerformanceMetrics:
    model_name: str
    success_rate: float
    avg_response_time: float
    cost_per_1k_tokens: float
    quality_score: float  # 0-1
    context_window: int
    last_updated: datetime

@dataclass
class TaskComplexityProfile:
    task_type: str
    estimated_tokens: int
    complexity_score: float  # 0-1
    requires_multimodal: bool
    requires_tools: bool
    budget_constraint: float
    quality_requirement: float  # 0-1
    latency_requirement: float  # max seconds

class IntelligentModelRouter:
    def __init__(self):
        self.model_configs = self._load_model_configurations()
        self.performance_tracker = ModelPerformanceTracker()
        self.ml_selector = MLModelSelector()
        self.cost_optimizer = CostOptimizer()
        self.fallback_chains = self._setup_fallback_chains()

    def _load_model_configurations(self) -&gt; Dict[str, Dict[str, Any]]:
        """Load current model configurations with latest pricing and capabilities"""
        return {
            "gpt-5-codex": {
                "provider": "openai",
                "max_tokens": 128000,
                "cost_per_1k_tokens": {"input": 0.01, "output": 0.03},
                "capabilities": ["coding", "reasoning", "agentic_workflows", "extended_think
                "specialization": ["multi_step_refactoring", "orchestration", "complex_debug
                "performance_score": 4.1,
                "avg_response_time": 2.3
            },
            "claude-4-opus": {
                "provider": "anthropic",
                "max_tokens": 200000,
                "cost_per_1k_tokens": {"input": 0.015, "output": 0.075},
                "capabilities": ["coding", "extended_thinking", "tool_use", "reasoning"],
                "specialization": ["complex_refactoring", "architectural_decisions", "code_c
                "performance_score": 4.5,  # 72.5% SWE-bench
                "avg_response_time": 3.1
            },
            "qwen3-max": {
                "provider": "qwen",
                "max_tokens": 262000,
                "cost_per_1k_tokens": {"input": 0.002, "output": 0.008},
                "capabilities": ["multilingual", "coding", "reasoning", "math"],
                "specialization": ["cost_effective_reasoning", "large_context", "multilingua
                "performance_score": 4.2,
                "avg_response_time": 1.8
            },
            "deepseek-coder-r1": {
                "provider": "deepseek",
                "max_tokens": 64000,
                "cost_per_1k_tokens": {"input": 0.0014, "output": 0.0028},
                "capabilities": ["coding", "debugging", "optimization"],
                "specialization": ["cost_effective_coding", "debugging", "performance_optim
                "performance_score": 3.9,
                "avg_response_time": 1.2
            },
            "perplexity-pro": {
                "provider": "perplexity",
                "max_tokens": 127000,
                "cost_per_1k_tokens": {"input": 0.001, "output": 0.001},
                "capabilities": ["web_search", "citations", "real_time_data"],
                "specialization": ["market_research", "real_time_search", "citations"],
                "performance_score": 4.0,
                "avg_response_time": 2.8
            }
        }

    async def route_request(self,
                          agent_type: str,
                          task: TaskComplexityProfile,
                          constraints: Optional[Dict[str, Any]] = None) -&gt; Tuple[str, Dic
        """
        Intelligent model selection using ML-based optimization
        """

        # Step 1: Get eligible models based on capabilities
        eligible_models = self._filter_eligible_models(agent_type, task)

        # Step 2: ML-based selection considering performance history
        ml_recommendation = await self.ml_selector.predict_optimal_model(
            agent_type, task, eligible_models
        )

        # Step 3: Cost-benefit analysis
        cost_optimized = await self.cost_optimizer.optimize_selection(
            ml_recommendation, task.budget_constraint
        )

        # Step 4: Apply real-time constraints (availability, rate limits)
        final_selection = await self._apply_runtime_constraints(
            cost_optimized, constraints
        )

        # Step 5: Setup fallback chain
        fallback_chain = self._get_fallback_chain(final_selection, eligible_models)

        return final_selection, {
            "fallback_chain": fallback_chain,
            "selection_reason": self._get_selection_reasoning(final_selection, task),
            "estimated_cost": self._calculate_estimated_cost(final_selection, task),
            "expected_quality": self._predict_quality_score(final_selection, task)
        }

    def _filter_eligible_models(self, agent_type: str, task: TaskComplexityProfile) -&gt; Li
        """Filter models based on agent type and task requirements"""
        agent_model_mapping = {
            "coordinator": ["gpt-5-codex", "qwen3-max", "claude-4.1-sonnet"],
            "research": ["perplexity-pro", "microsoft-copilot", "gpt-5"],
            "implementation": ["claude-4-opus", "gpt-5-codex", "deepseek-coder-r1"],
            "testing": ["gpt-5-codex", "claude-4-opus", "deepseek-coder-r1"],
            "documentation": ["gpt-5-nano", "gpt-5", "claude-4-sonnet"],
            "quality": ["claude-4-sonnet", "claude-4-opus", "gpt-5"],
            "security": ["gpt-5", "claude-4-sonnet", "deepseek-coder-r1"],
            "debugging": ["deepseek-coder-r1", "qwen3-coder", "claude-4-sonnet"]
        }

        base_models = agent_model_mapping.get(agent_type, list(self.model_configs.keys()))
```

```
        # Filter based on task requirements
        eligible = []
        for model in base_models:
            if model not in self.model_configs:
                continue

            config = self.model_configs[model]

            # Check context window requirement
            if task.estimated_tokens &gt; config["max_tokens"]:
                continue

            # Check multimodal requirement
            if task.requires_multimodal and "multimodal" not in config.get("capabilities", [
                continue

            # Check tool use requirement
            if task.requires_tools and "tool_use" not in config.get("capabilities", []):
                continue

            eligible.append(model)

        return eligible

    async def _apply_runtime_constraints(self,
                                       model: str,
                                       constraints: Optional[Dict[str, Any]]) -&gt; str:
        """Apply runtime constraints like rate limits, availability"""
        if not constraints:
            return model

        # Check rate limits
        if constraints.get("check_rate_limits", True):
            rate_limit_status = await self._check_rate_limits(model)
            if not rate_limit_status["available"]:
                # Use fallback
                fallback = self.fallback_chains.get(model, [])
                for fallback_model in fallback:
                    fb_rate_status = await self._check_rate_limits(fallback_model)
                    if fb_rate_status["available"]:
                        return fallback_model

        # Check API availability
        if constraints.get("check_availability", True):
            is_available = await self._check_api_availability(model)
            if not is_available:
                # Use fallback
                fallback = self.fallback_chains.get(model, [])
                for fallback_model in fallback:
                    if await self._check_api_availability(fallback_model):
                        return fallback_model

        return model

class MLModelSelector:
    """ML-based model selection using historical performance data"""

    def __init__(self):
        self.model = None
        self.feature_columns = [
            'task_complexity', 'estimated_tokens', 'quality_requirement',
            'budget_constraint', 'agent_type_encoded', 'time_of_day',
            'historical_success_rate', 'historical_quality'
        ]
        self.load_or_train_model()

    async def predict_optimal_model(self,
                                  agent_type: str,
                                  task: TaskComplexityProfile,
                                  eligible_models: List[str]) -&gt; str:
        """Use ML model to predict optimal model selection"""

        if not self.model:
            # Fallback to rule-based selection
            return self._rule_based_selection(agent_type, task, eligible_models)

        # Prepare features for each eligible model
        predictions = []
        for model_name in eligible_models:
            features = await self._extract_features(agent_type, task, model_name)
            prediction = self.model.predict_proba([features])[^0]
            predictions.append((model_name, prediction[^1]))  # probability of success

        # Select model with highest success probability
        best_model = max(predictions, key=lambda x: x[^1])
        return best_model[^0]

    def _rule_based_selection(self,
                            agent_type: str,
                            task: TaskComplexityProfile,
                            eligible_models: List[str]) -&gt; str:
        """Fallback rule-based selection when ML model unavailable"""

        # Agent-specific preferences
        preferences = {
            "coordinator": ["gpt-5-codex", "qwen3-max"],
            "implementation": ["claude-4-opus", "gpt-5-codex"],
            "research": ["perplexity-pro", "gpt-5"],
            "quality": ["claude-4-sonnet", "claude-4-opus"]
        }

        agent_prefs = preferences.get(agent_type, eligible_models)

        for preferred in agent_prefs:
            if preferred in eligible_models:
                return preferred

        return eligible_models[^0] if eligible_models else "gpt-5-codex"

# Usage Example für Windsurf
async def main():
    router = IntelligentModelRouter()

    task = TaskComplexityProfile(
        task_type="multi_agent_orchestration",
        estimated_tokens=25000,
        complexity_score=0.9,
        requires_multimodal=False,
        requires_tools=True,
        budget_constraint=0.50,  # $0.50 max
        quality_requirement=0.8,
        latency_requirement=5.0  # 5 seconds max
    )

    selected_model, metadata = await router.route_request(
        agent_type="coordinator",
        task=task,
        constraints={"check_rate_limits": True, "check_availability": True}
    )

    print(f"Selected Model: {selected_model}")
    print(f"Fallback Chain: {metadata['fallback_chain']}")
    print(f"Estimated Cost: ${metadata['estimated_cost']:.4f}")
    print(f"Expected Quality: {metadata['expected_quality']:.2f}")
```

Deliverables T1.2

- [ ] Intelligent Model Router mit ML-basierter Auswahl
- [ ] Real-time Rate Limiting und Availability Checking
- [ ] Cost Optimization Engine
- [ ] Performance Tracking und Analytics
- [ ] Fallback Chain Management
- [ ] Integration Tests für alle unterstützten Modelle
- [ ] Performance Benchmarks
- [ ] Cost Analysis Dashboard

Task T1.3: Database & Infrastructure Setup

AI-Model: GPT-5-Codex | **Fallback:** QWEN3-Coder | **Dauer:** 6 Tage | **Komplexität:** 6/10

PostgreSQL + pgvector Setup mit Docker

```
# docker-compose.infrastructure.yml
version: '3.8'

services:
  postgres-primary:
    image: pgvector/pgvector:pg16
    environment:
      POSTGRES_DB: multiagent_system
      POSTGRES_USER: ${DB_USER:-multiagent_user}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_INITDB_ARGS: "--auth-host=md5"
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./scripts/init-db.sql:/docker-entrypoint-initdb.d/01-init.sql
      - ./scripts/create-extensions.sql:/docker-entrypoint-initdb.d/02-extensions.sql
    command: &gt;
      postgres
        -c max_connections=200
        -c shared_buffers=256MB
        -c effective_cache_size=1GB
        -c maintenance_work_mem=64MB
        -c checkpoint_completion_target=0.9
        -c wal_buffers=16MB
        -c default_statistics_target=100
        -c random_page_cost=1.1
        -c effective_io_concurrency=200
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${DB_USER:-multiagent_user}"]
      interval: 10s
      timeout: 5s
      retries: 5
    restart: unless-stopped

  redis-cluster:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
      - ./config/redis.conf:/usr/local/etc/redis/redis.conf
    command: redis-server /usr/local/etc/redis/redis.conf
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 3s
      retries: 5
    restart: unless-stopped

  qdrant-vector:
    image: qdrant/qdrant:v1.7.0
    ports:
      - "6333:6333"
      - "6334:6334"
    volumes:
      - qdrant_data:/qdrant/storage
    environment:
      QDRANT__SERVICE__HTTP_PORT: 6333
      QDRANT__SERVICE__GRPC_PORT: 6334
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:6333/health || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 3
    restart: unless-stopped

volumes:
  postgres_data:
  redis_data:
  qdrant_data:
```

```
-- scripts/init-db.sql
-- Initialize Multi-Agent System Database Schema

-- Enable required extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS vector;
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

-- Agent Management Schema
CREATE TABLE agents (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(100) NOT NULL UNIQUE,
    agent_type VARCHAR(50) NOT NULL,
    primary_model VARCHAR(100) NOT NULL,
    fallback_models TEXT[] DEFAULT '{}',
    capabilities TEXT[] DEFAULT '{}',
    status VARCHAR(20) DEFAULT 'active' CHECK (status IN ('active', 'inactive', 'maintenance
    configuration JSONB DEFAULT '{}',
    performance_metrics JSONB DEFAULT '{}',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    version INTEGER DEFAULT 1
);

-- Projects and Workflows
CREATE TABLE projects (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(200) NOT NULL,
    description TEXT,
    status VARCHAR(50) DEFAULT 'planning' CHECK (status IN (
        'planning', 'in_progress', 'testing', 'review', 'completed', 'failed', 'cancelled'
    )),
    assigned_agents UUID[] DEFAULT '{}',
    github_repo VARCHAR(500),
    knowledge_base_embedding vector(1536),
    project_metadata JSONB DEFAULT '{}',
    success_criteria JSONB DEFAULT '{}',
    quality_gates JSONB DEFAULT '{}',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    completed_at TIMESTAMP WITH TIME ZONE,
    created_by VARCHAR(100),
    estimated_completion TIMESTAMP WITH TIME ZONE
);

-- Workflow Definitions
CREATE TABLE workflow_definitions (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name VARCHAR(200) NOT NULL,
    description TEXT,
    workflow_type VARCHAR(50) NOT NULL,
    agent_dependencies JSONB NOT NULL, -- {"agent_a": ["agent_b", "agent_c"]}
    execution_order TEXT[] DEFAULT '{}',
    parallel_execution_groups JSONB DEFAULT '{}',
    success_conditions JSONB DEFAULT '{}',
    failure_conditions JSONB DEFAULT '{}',
    retry_policy JSONB DEFAULT '{}',
    timeout_minutes INTEGER DEFAULT 60,
    is_active BOOLEAN DEFAULT true,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Workflow Executions
CREATE TABLE workflow_executions (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    workflow_definition_id UUID REFERENCES workflow_definitions(id),
    project_id UUID REFERENCES projects(id),
    status VARCHAR(50) DEFAULT 'queued' CHECK (status IN (
        'queued', 'running', 'paused', 'completed', 'failed', 'cancelled'
    )),
    current_step VARCHAR(100),
    steps_completed TEXT[] DEFAULT '{}',
    steps_failed TEXT[] DEFAULT '{}',
    execution_context JSONB DEFAULT '{}',
    results JSONB DEFAULT '{}',
    error_log JSONB DEFAULT '{}',
    performance_metrics JSONB DEFAULT '{}',
    started_at TIMESTAMP WITH TIME ZONE,
    completed_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    estimated_completion TIMESTAMP WITH TIME ZONE
);

-- Agent Interactions and Communications
CREATE TABLE agent_interactions (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    workflow_execution_id UUID REFERENCES workflow_executions(id),
    project_id UUID REFERENCES projects(id),
    source_agent_id UUID REFERENCES agents(id),
    target_agent_id UUID REFERENCES agents(id),
    interaction_type VARCHAR(50) NOT NULL, -- 'request', 'response', 'notification', 'error
    message_content JSONB NOT NULL,
    context JSONB DEFAULT '{}',
    status VARCHAR(20) DEFAULT 'sent' CHECK (status IN ('sent', 'received', 'processed', 'fa
    priority INTEGER DEFAULT 5 CHECK (priority BETWEEN 1 AND 10),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    processed_at TIMESTAMP WITH TIME ZONE,
    response_required BOOLEAN DEFAULT false,
    timeout_at TIMESTAMP WITH TIME ZONE
);

-- Model Usage Tracking
```

```sql
CREATE TABLE model_usage_tracking (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    agent_id UUID REFERENCES agents(id),
    project_id UUID REFERENCES projects(id),
    workflow_execution_id UUID REFERENCES workflow_executions(id),
    model_name VARCHAR(100) NOT NULL,
    model_provider VARCHAR(50) NOT NULL,
    task_type VARCHAR(100),
    input_tokens INTEGER DEFAULT 0,
    output_tokens INTEGER DEFAULT 0,
    total_tokens INTEGER GENERATED ALWAYS AS (input_tokens + output_tokens) STORED,
    cost_usd DECIMAL(10,6) DEFAULT 0.0,
    quality_score DECIMAL(3,2), -- 0.00 to 1.00
    response_time_ms INTEGER,
    success BOOLEAN DEFAULT true,
    error_message TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    session_id UUID,
    request_metadata JSONB DEFAULT '{}'
);

-- Knowledge Base and Embeddings
CREATE TABLE knowledge_base_entries (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    project_id UUID REFERENCES projects(id),
    entry_type VARCHAR(50) NOT NULL, -- 'code', 'documentation', 'issue', 'commit', 'test'
    title VARCHAR(500) NOT NULL,
    content TEXT NOT NULL,
    content_embedding vector(1536),
    metadata JSONB DEFAULT '{}',
    source_url VARCHAR(1000),
    file_path VARCHAR(1000),
    language VARCHAR(50),
    tags TEXT[] DEFAULT '{}',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    indexed_at TIMESTAMP WITH TIME ZONE
);

-- System Configuration
CREATE TABLE system_config (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    config_key VARCHAR(200) NOT NULL UNIQUE,
    config_value JSONB NOT NULL,
    config_type VARCHAR(50) NOT NULL, -- 'agent', 'model', 'workflow', 'system'
    description TEXT,
    is_encrypted BOOLEAN DEFAULT false,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_by VARCHAR(100)
);

-- Performance and Analytics
CREATE TABLE system_metrics (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    metric_name VARCHAR(100) NOT NULL,
    metric_value DECIMAL(15,6) NOT NULL,
    metric_type VARCHAR(50) NOT NULL, -- 'counter', 'gauge', 'histogram'
    labels JSONB DEFAULT '{}',
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    retention_days INTEGER DEFAULT 90
);

-- Indexes for Performance
-- Agent Management
CREATE INDEX idx_agents_type_status ON agents(agent_type, status);
CREATE INDEX idx_agents_name ON agents(name);

-- Projects
CREATE INDEX idx_projects_status ON projects(status);
CREATE INDEX idx_projects_created_at ON projects(created_at DESC);
CREATE INDEX idx_projects_embedding ON projects USING ivfflat (knowledge_base_embedding vect
  WITH (lists = 100);

-- Workflows
CREATE INDEX idx_workflow_executions_status ON workflow_executions(status);
CREATE INDEX idx_workflow_executions_project ON workflow_executions(project_id, created_at D
CREATE INDEX idx_workflow_executions_definition ON workflow_executions(workflow_definition_i

-- Agent Interactions
CREATE INDEX idx_agent_interactions_workflow ON agent_interactions(workflow_execution_id);
CREATE INDEX idx_agent_interactions_agents ON agent_interactions(source_agent_id, target_age
CREATE INDEX idx_agent_interactions_created_at ON agent_interactions(created_at DESC);
CREATE INDEX idx_agent_interactions_status ON agent_interactions(status, priority);

-- Model Usage
CREATE INDEX idx_model_usage_agent_project ON model_usage_tracking(agent_id, project_id);
CREATE INDEX idx_model_usage_model ON model_usage_tracking(model_name, created_at DESC);
CREATE INDEX idx_model_usage_cost ON model_usage_tracking(created_at DESC, cost_usd);

-- Knowledge Base
CREATE INDEX idx_knowledge_base_project ON knowledge_base_entries(project_id, entry_type);
CREATE INDEX idx_knowledge_base_embedding ON knowledge_base_entries USING ivfflat (content_e
  WITH (lists = 100);
CREATE INDEX idx_knowledge_base_tags ON knowledge_base_entries USING gin(tags);
CREATE INDEX idx_knowledge_base_search ON knowledge_base_entries USING gin(to_tsvector('engl

-- System Metrics
CREATE INDEX idx_system_metrics_name_timestamp ON system_metrics(metric_name, timestamp DESC

-- Functions for automatic updates
CREATE OR REPLACE FUNCTION update_updated_at_column()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ language 'plpgsql';

-- Triggers for automatic timestamp updates
CREATE TRIGGER update_agents_updated_at BEFORE UPDATE ON agents
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();
```

```
CREATE TRIGGER update_projects_updated_at BEFORE UPDATE ON projects
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_workflow_definitions_updated_at BEFORE UPDATE ON workflow_definitions
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_knowledge_base_updated_at BEFORE UPDATE ON knowledge_base_entries
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

-- Insert initial system configuration
INSERT INTO system_config (config_key, config_value, config_type, description) VALUES
('orchestrator.framework', '"langgraph"', 'system', 'Default orchestration framework'),
('orchestrator.fallback_framework', '"crewai"', 'system', 'Fallback orchestration framework
('models.default_coordinator', '"gpt-5-codex"', 'model', 'Default model for coordinator agen
('models.default_implementation', '"claude-4-opus"', 'model', 'Default model for implementat
('system.max_concurrent_workflows', '10', 'system', 'Maximum concurrent workflow executions
('system.default_workflow_timeout', '3600', 'system', 'Default workflow timeout in seconds')

-- Create initial workflow definitions
INSERT INTO workflow_definitions (name, description, workflow_type, agent_dependencies, exec
('standard_software_project', 'Standard software development workflow', 'sequential',
 '{"research": [], "specification": ["research"], "implementation": ["specification"], "test
 '["research", "specification", "implementation", "testing", "quality", "documentation", "de
);
```

```
# src/core/database/connection_manager.py
import asyncio
import asyncpg
import redis.asyncio as redis
from qdrant_client.async_qdrant import AsyncQdrantClient
from typing import Optional, Dict, Any
import logging
from contextlib import asynccontextmanager

class DatabaseConnectionManager:
    """Manages connections to PostgreSQL, Redis, and Qdrant"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.pg_pool: Optional[asyncpg.Pool] = None
        self.redis_client: Optional[redis.Redis] = None
        self.qdrant_client: Optional[AsyncQdrantClient] = None
        self.logger = logging.getLogger(__name__)

    async def initialize(self):
        """Initialize all database connections"""
        await self._init_postgresql()
        await self._init_redis()
        await self._init_qdrant()

    async def _init_postgresql(self):
        """Initialize PostgreSQL connection pool"""
        try:
            self.pg_pool = await asyncpg.create_pool(
                host=self.config['postgresql']['host'],
                port=self.config['postgresql']['port'],
                database=self.config['postgresql']['database'],
                user=self.config['postgresql']['user'],
                password=self.config['postgresql']['password'],
                min_size=5,
                max_size=20,
                command_timeout=60,
                server_settings={
                    'application_name': 'multiagent_system',
                    'search_path': 'public'
                }
            )
            self.logger.info("PostgreSQL connection pool initialized")
        except Exception as e:
            self.logger.error(f"Failed to initialize PostgreSQL: {e}")
            raise

    async def _init_redis(self):
        """Initialize Redis connection"""
        try:
            self.redis_client = redis.Redis(
                host=self.config['redis']['host'],
                port=self.config['redis']['port'],
                db=0,
                decode_responses=True,
                retry_on_timeout=True,
                retry_on_error=[redis.ConnectionError, redis.TimeoutError],
                socket_keepalive=True,
                socket_keepalive_options={}
            )
            # Test connection
            await self.redis_client.ping()
            self.logger.info("Redis connection initialized")
        except Exception as e:
            self.logger.error(f"Failed to initialize Redis: {e}")
            raise

    async def _init_qdrant(self):
        """Initialize Qdrant vector database connection"""
        try:
            self.qdrant_client = AsyncQdrantClient(
                host=self.config['qdrant']['host'],
                port=self.config['qdrant']['port'],
                timeout=30
            )
            # Test connection
            await self.qdrant_client.get_collections()
            self.logger.info("Qdrant connection initialized")
        except Exception as e:
            self.logger.error(f"Failed to initialize Qdrant: {e}")
            raise

    @asynccontextmanager
    async def get_pg_connection(self):
        """Get PostgreSQL connection from pool"""
        async with self.pg_pool.acquire() as connection:
            yield connection

    async def get_redis_client(self) -&gt; redis.Redis:
        """Get Redis client"""
        return self.redis_client

    async def get_qdrant_client(self) -&gt; AsyncQdrantClient:
        """Get Qdrant client"""
        return self.qdrant_client

    async def close_all(self):
        """Close all database connections"""
        if self.pg_pool:
            await self.pg_pool.close()
        if self.redis_client:
            await self.redis_client.close()
        if self.qdrant_client:
            self.qdrant_client.close()

# Database Models using SQLAlchemy for ORM
from sqlalchemy import Column, String, Integer, DateTime, Boolean, ARRAY, JSON
from sqlalchemy.dialects.postgresql import UUID, DECIMAL
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.sql import func
from pgvector.sqlalchemy import Vector

Base = declarative_base()

class Agent(Base):
    __tablename__ = 'agents'

    id = Column(UUID(as_uuid=True), primary_key=True, server_default=func.uuid_generate_v4()
    name = Column(String(100), nullable=False, unique=True)
    agent_type = Column(String(50), nullable=False)
    primary_model = Column(String(100), nullable=False)
    fallback_models = Column(ARRAY(String), default=list)
    capabilities = Column(ARRAY(String), default=list)
    status = Column(String(20), default='active')
    configuration = Column(JSON, default=dict)
    performance_metrics = Column(JSON, default=dict)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), server_default=func.now(), onupdate=func.no
    version = Column(Integer, default=1)

class Project(Base):
    __tablename__ = 'projects'

    id = Column(UUID(as_uuid=True), primary_key=True, server_default=func.uuid_generate_v4()
    name = Column(String(200), nullable=False)
    description = Column(String)
    status = Column(String(50), default='planning')
    assigned_agents = Column(ARRAY(UUID), default=list)
    github_repo = Column(String(500))
    knowledge_base_embedding = Column(Vector(1536))
    project_metadata = Column(JSON, default=dict)
    success_criteria = Column(JSON, default=dict)
    quality_gates = Column(JSON, default=dict)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
    updated_at = Column(DateTime(timezone=True), server_default=func.now(), onupdate=func.no
    completed_at = Column(DateTime(timezone=True))
    created_by = Column(String(100))
    estimated_completion = Column(DateTime(timezone=True))
```

Deliverables T1.3

- [ ] PostgreSQL + pgvector Database Setup mit Docker
- [ ] Redis Cluster für Caching und Messaging
- [ ] Qdrant Vector Database für Semantic Search
- [ ] Database Connection Management
- [ ] ORM Models (SQLAlchemy)
- [ ] Database Migration Scripts
- [ ] Performance Tuning und Indexing
- [ ] Backup und Recovery Strategy
- [ ] Health Checks und Monitoring
- [ ] Integration Tests für alle Database Components

Task T1.4: Web Dashboard Foundation

AI-Model: Claude-4-Sonnet | **Fallback:** GPT-5-nano | **Dauer:** 4 Tage | **Komplexität:** 5/10

Next.js 14 Dashboard mit Real-time Updates

```
// src/web-dashboard/app/layout.tsx
import type { Metadata } from 'next'
import { Inter } from 'next/font/google'
import './globals.css'
import { ThemeProvider } from '@/components/theme-provider'
import { Toaster } from '@/components/ui/toaster'
import { WebSocketProvider } from '@/lib/websocket-provider'

const inter = Inter({ subsets: ['latin'] })

export const metadata: Metadata = {
  title: 'Multi-Agent AI System Dashboard',
  description: 'Monitor and control your AI agent workforce',
}

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    &lt;html lang="de" suppressHydrationWarning&gt;
      &lt;body className={inter.className}&gt;
        &lt;ThemeProvider attribute="class" defaultTheme="dark"&gt;
          &lt;WebSocketProvider&gt;
            {children}
            &lt;Toaster /&gt;
          &lt;/WebSocketProvider&gt;
        &lt;/ThemeProvider&gt;
      &lt;/body&gt;
    &lt;/html&gt;
  )
}
```

```
// src/web-dashboard/app/dashboard/page.tsx
'use client'

import { useEffect, useState } from 'react'
import { Card, CardContent, CardDescription, CardHeader, CardTitle } from '@/components/ui/c
import { Badge } from '@/components/ui/badge'
import { Progress } from '@/components/ui/progress'
import { AgentStatusCard } from '@/components/agent-status-card'
import { ProjectOverview } from '@/components/project-overview'
import { TokenUsageChart } from '@/components/token-usage-chart'
import { WorkflowProgress } from '@/components/workflow-progress'
import { useWebSocket } from '@/lib/websocket-provider'
import { useAgentStatus } from '@/hooks/use-agent-status'
import { useSystemMetrics } from '@/hooks/use-system-metrics'

export default function DashboardPage() {
  const { isConnected, lastMessage } = useWebSocket()
  const { agents, isLoading: agentsLoading } = useAgentStatus()
  const { metrics, isLoading: metricsLoading } = useSystemMetrics()

  const [activeProjects, setActiveProjects] = useState(0)
  const [totalCosts, setTotalCosts] = useState(0)
  const [systemHealth, setSystemHealth] = useState(95)

  useEffect(() =&gt; {
    // Update dashboard data when WebSocket message received
    if (lastMessage) {
      const data = JSON.parse(lastMessage.data)
      switch (data.type) {
        case 'agent_status_update':
          // Handle agent status updates
          break
        case 'workflow_progress':
          // Handle workflow progress updates
          break
        case 'system_metrics':
          setSystemHealth(data.payload.health_score)
          break
      }
    }
  }, [lastMessage])
```

```
if (agentsLoading || metricsLoading) {
  return (
    <div>
      <div>
        <div></div>
        <p>Dashboard wird geladen...</p>
      </div>
    </div>
  )
}

return (
  <div>
    <div>
      <h2>Dashboard</h2>
      <div>
        &lt;Badge variant={isConnected ? 'default' : 'destructive'}&gt;
          {isConnected ? 'Verbunden' : 'Getrennt'}
        &lt;/Badge&gt;
        &lt;Badge variant="outline"&gt;System Health: {systemHealth}%&lt;/Badge&gt;
      </div>
    </div>

    {/* System Overview Cards */}
    <div>
      &lt;Card&gt;
        &lt;CardHeader className="flex flex-row items-center justify-between space-y-0 pb-
          &lt;CardTitle className="text-sm font-medium"&gt;Aktive Agenten&lt;/CardTitle&g
          &lt;svg
            xmlns="http://www.w3.org/2000/svg"
            viewBox="0 0 24 24"
            fill="none"
            stroke="currentColor"
            strokeLinecap="round"
            strokeLinejoin="round"
            strokeWidth="2"
            className="h-4 w-4 text-muted-foreground"
          &gt;
            &lt;path d="M16 21v-2a4 4 0 0 0-4-4H6a4 4 0 0 0-4 4v2" /&gt;
            &lt;circle cx="9" cy="7" r="4" /&gt;
            &lt;path d="m22 21-3-3m0 0a6 6 0 1 0-6-6 6 6 0 0 0 6 6Z" /&gt;
          &lt;/svg&gt;
        &lt;/CardHeader&gt;
        &lt;CardContent&gt;
          <div>{agents.filter(a =&gt; a.status === 'active').length}</div>
          <p>von {agents.length} gesamt</p>
        &lt;/CardContent&gt;
      &lt;/Card&gt;

      &lt;Card&gt;
        &lt;CardHeader className="flex flex-row items-center justify-between space-y-0 pb-
          &lt;CardTitle className="text-sm font-medium"&gt;Laufende Projekte&lt;/CardTitle
          &lt;svg
            xmlns="http://www.w3.org/2000/svg"
            viewBox="0 0 24 24"
```

```
              fill="none"
              stroke="currentColor"
              strokeLinecap="round"
              strokeLinejoin="round"
              strokeWidth="2"
              className="h-4 w-4 text-muted-foreground"
            &gt;
              &lt;path d="M22 12h-4l-3 9L9 3l-3 9H2" /&gt;
            &lt;/svg&gt;
          &lt;/CardHeader&gt;
          &lt;CardContent&gt;
            <div>{activeProjects}</div>
            <p>+2 seit letzter Woche</p>
          &lt;/CardContent&gt;
        &lt;/Card&gt;

        &lt;Card&gt;
          &lt;CardHeader className="flex flex-row items-center justify-between space-y-0 pb-
            &lt;CardTitle className="text-sm font-medium"&gt;Token-Kosten (heute)&lt;/CardTi
            &lt;svg
              xmlns="http://www.w3.org/2000/svg"
              viewBox="0 0 24 24"
              fill="none"
              stroke="currentColor"
              strokeLinecap="round"
              strokeLinejoin="round"
              strokeWidth="2"
              className="h-4 w-4 text-muted-foreground"
            &gt;
              &lt;path d="M12 2v20m8-10H4" /&gt;
            &lt;/svg&gt;
          &lt;/CardHeader&gt;
          &lt;CardContent&gt;
            <div>${totalCosts.toFixed(2)}</div>
            <p>-12% gegenüber gestern</p>
          &lt;/CardContent&gt;
        &lt;/Card&gt;

        &lt;Card&gt;
          &lt;CardHeader className="flex flex-row items-center justify-between space-y-0 pb-
            &lt;CardTitle className="text-sm font-medium"&gt;System Health&lt;/CardTitle&gt;
            &lt;svg
              xmlns="http://www.w3.org/2000/svg"
              viewBox="0 0 24 24"
              fill="none"
              stroke="currentColor"
              strokeLinecap="round"
              strokeLinejoin="round"
              strokeWidth="2"
              className="h-4 w-4 text-muted-foreground"
            &gt;
              &lt;path d="M22 12h-4l-3 9L9 3l-3 9H2" /&gt;
            &lt;/svg&gt;
          &lt;/CardHeader&gt;
          &lt;CardContent&gt;
            <div>{systemHealth}%</div>
            &lt;Progress value={systemHealth} className="mt-2" /&gt;
          &lt;/CardContent&gt;
        &lt;/Card&gt;
      </div>

      {/* Main Dashboard Content */}
      <div>
        {/* Agent Status Panel */}
        &lt;Card className="col-span-4"&gt;
          &lt;CardHeader&gt;
            &lt;CardTitle&gt;Agent Status&lt;/CardTitle&gt;
            &lt;CardDescription&gt;Übersicht über alle AI-Agenten&lt;/CardDescription&gt;
          &lt;/CardHeader&gt;
          &lt;CardContent className="space-y-4"&gt;
            {agents.map((agent) =&gt; (
              &lt;AgentStatusCard key={agent.id} agent={agent} /&gt;
            ))}
          &lt;/CardContent&gt;
        &lt;/Card&gt;

        {/* Token Usage Chart */}
        &lt;Card className="col-span-3"&gt;
          &lt;CardHeader&gt;
            &lt;CardTitle&gt;Token-Verbrauch&lt;/CardTitle&gt;
            &lt;CardDescription&gt;Letzte 24 Stunden&lt;/CardDescription&gt;
          &lt;/CardHeader&gt;
          &lt;CardContent&gt;
            &lt;TokenUsageChart /&gt;
          &lt;/CardContent&gt;
        &lt;/Card&gt;
      </div>

      {/* Project and Workflow Overview */}
      <div>
        &lt;Card&gt;
          &lt;CardHeader&gt;
            &lt;CardTitle&gt;Aktive Projekte&lt;/CardTitle&gt;
            &lt;CardDescription&gt;Projekte in Bearbeitung&lt;/CardDescription&gt;
          &lt;/CardHeader&gt;
          &lt;CardContent&gt;
            &lt;ProjectOverview /&gt;
          &lt;/CardContent&gt;
        &lt;/Card&gt;

        &lt;Card&gt;
          &lt;CardHeader&gt;
            &lt;CardTitle&gt;Workflow-Fortschritt&lt;/CardTitle&gt;
            &lt;CardDescription&gt;Aktuelle Workflow-Ausführungen&lt;/CardDescription&gt;
          &lt;/CardHeader&gt;
          &lt;CardContent&gt;
            &lt;WorkflowProgress /&gt;
          &lt;/CardContent&gt;
        &lt;/Card&gt;
      </div>
    </div>
```

```
  )
}
```

```
// src/web-dashboard/components/agent-status-card.tsx
'use client'

import { Badge } from '@/components/ui/badge'
import { Button } from '@/components/ui/button'
import { Card, CardContent } from '@/components/ui/card'
import { Progress } from '@/components/ui/progress'
import { Agent } from '@/types/agent'
import { Activity, Brain, Zap } from 'lucide-react'
import { useState } from 'react'

interface AgentStatusCardProps {
  agent: Agent
}

export function AgentStatusCard({ agent }: AgentStatusCardProps) {
  const [isExpanded, setIsExpanded] = useState(false)

  const getStatusColor = (status: string) =&gt; {
    switch (status) {
      case 'active': return 'default'
      case 'busy': return 'secondary'
      case 'error': return 'destructive'
      case 'maintenance': return 'outline'
      default: return 'outline'
    }
  }

  const getStatusIcon = (agentType: string) =&gt; {
    switch (agentType) {
      case 'coordinator': return &lt;Brain className="h-4 w-4" /&gt;
      case 'implementation': return &lt;Zap className="h-4 w-4" /&gt;
      default: return &lt;Activity className="h-4 w-4" /&gt;
    }
  }

  return (
    &lt;Card className="transition-all duration-200 hover:shadow-md"&gt;
      &lt;CardContent className="p-4"&gt;
        <div>
          <div>
            <div>
              {getStatusIcon(agent.agent_type)}
            </div>
            <div>
              <div>
                <p>{agent.name}</p>
                &lt;Badge variant={getStatusColor(agent.status)} className="text-xs"&gt;
                  {agent.status}
                &lt;/Badge&gt;
              </div>
              <p>
```

```
                {agent.primary_model} | {agent.agent_type}
              </p>
            </div>
          </div>

          <div>
            <div>
              <div>CPU: {agent.performance_metrics?.cpu_usage || 0}%</div>
              <div>Mem: {agent.performance_metrics?.memory_usage || 0}%</div>
            </div>
            &lt;Button
              variant="ghost"
              size="sm"
              onClick={() =&gt; setIsExpanded(!isExpanded)}
            &gt;
              {isExpanded ? '−' : '+'}
            &lt;/Button&gt;
          </div>
        </div>

        {isExpanded &amp;&amp; (
          <div>
            <div>
              <div>
                <span>Tasks heute:</span> {agent.performance_metrics?.tasks_today || 0}
              </div>
              <div>
                <span>Erfolgsrate:</span> {((agent.performance_metrics?.success_rate || 0) *
              </div>
              <div>
                <span>Ø Response:</span> {agent.performance_metrics?.avg_response_time || 0}
              </div>
              <div>
                <span>Token heute:</span> {agent.performance_metrics?.tokens_today || 0}
              </div>
            </div>

            <div>
              <div>
                <span>Performance Score</span>
                <span>{((agent.performance_metrics?.performance_score || 0) * 100).toFixed(1
              </div>
              &lt;Progress value={(agent.performance_metrics?.performance_score || 0) * 100}
            </div>

            <div>
              &lt;Button variant="outline" size="sm" className="text-xs"&gt;
                Konfiguration
              &lt;/Button&gt;
              &lt;Button variant="outline" size="sm" className="text-xs"&gt;
                Logs
              &lt;/Button&gt;
              &lt;Button variant="outline" size="sm" className="text-xs"&gt;
                Restart
              &lt;/Button&gt;
            </div>
          </div>
        )}
      &lt;/CardContent&gt;
    &lt;/Card&gt;
  )
}
```

```
// src/web-dashboard/lib/websocket-provider.tsx
'use client'

import { createContext, useContext, useEffect, useState } from 'react'
import { toast } from '@/components/ui/use-toast'

interface WebSocketContextType {
  socket: WebSocket | null
  isConnected: boolean
  lastMessage: MessageEvent | null
  sendMessage: (message: any) =&gt; void
}

const WebSocketContext = createContext&lt;WebSocketContextType | undefined&gt;(undefined)

export function WebSocketProvider({ children }: { children: React.ReactNode }) {
  const [socket, setSocket] = useState&lt;WebSocket | null&gt;(null)
  const [isConnected, setIsConnected] = useState(false)
  const [lastMessage, setLastMessage] = useState&lt;MessageEvent | null&gt;(null)
  const [reconnectAttempts, setReconnectAttempts] = useState(0)
  const maxReconnectAttempts = 5

  const connect = () =&gt; {
    try {
      const ws = new WebSocket(process.env.NEXT_PUBLIC_WS_URL || 'ws://localhost:8000/ws')

      ws.onopen = () =&gt; {
        console.log('WebSocket connected')
        setIsConnected(true)
        setReconnectAttempts(0)
        setSocket(ws)

        toast({
          title: "Verbindung hergestellt",
          description: "WebSocket-Verbindung zum System ist aktiv.",
        })
      }

      ws.onmessage = (event) =&gt; {
        setLastMessage(event)

        // Handle special message types
        try {
          const data = JSON.parse(event.data)
          if (data.type === 'error') {
            toast({
              title: "System Fehler",
              description: data.message,
              variant: "destructive"
            })
          } else if (data.type === 'agent_error') {
            toast({
              title: `Agent Fehler: ${data.agent_name}`,
              description: data.error,
              variant: "destructive"
            })
          }
        } catch (e) {
          // Ignore non-JSON messages
        }
      }

      ws.onclose = (event) =&gt; {
        console.log('WebSocket disconnected:', event.reason)
        setIsConnected(false)
        setSocket(null)

        // Attempt to reconnect
        if (reconnectAttempts &lt; maxReconnectAttempts) {
          const timeout = Math.min(1000 * Math.pow(2, reconnectAttempts), 30000)
          setTimeout(() =&gt; {
            setReconnectAttempts(prev =&gt; prev + 1)
            connect()
          }, timeout)
        } else {
          toast({
            title: "Verbindung verloren",
            description: "WebSocket-Verbindung konnte nicht wiederhergestellt werden.",
            variant: "destructive"
          })
        }
      }

      ws.onerror = (error) =&gt; {
        console.error('WebSocket error:', error)
        toast({
          title: "Verbindungsfehler",
          description: "WebSocket-Verbindung ist fehlgeschlagen.",
          variant: "destructive"
        })
      }

    } catch (error) {
      console.error('Failed to create WebSocket connection:', error)
    }
  }

  const sendMessage = (message: any) =&gt; {
    if (socket &amp;&amp; isConnected) {
      socket.send(JSON.stringify(message))
    } else {
      toast({
        title: "Keine Verbindung",
        description: "WebSocket ist nicht verbunden. Nachricht kann nicht gesendet werden.",
        variant: "destructive"
      })
    }
  }

  useEffect(() =&gt; {
    connect()

    return () =&gt; {
      if (socket) {
        socket.close()
      }
    }
  }, [])

  return (
    &lt;WebSocketContext.Provider value={{
      socket,
      isConnected,
      lastMessage,
      sendMessage
    }}&gt;
      {children}
    &lt;/WebSocketContext.Provider&gt;
  )
}

export function useWebSocket() {
  const context = useContext(WebSocketContext)
  if (context === undefined) {
    throw new Error('useWebSocket must be used within a WebSocketProvider')
  }
  return context
}
```

Deliverables T1.4

- [ ] Next.js 14 Dashboard mit App Router
- [ ] Real-time WebSocket Integration
- [ ] Agent Status Monitoring Components
- [ ] Token Usage Tracking und Visualisierung
- [ ] Project Overview Dashboard
- [ ] Responsive Design (Mobile/Desktop)
- [ ] Dark/Light Theme Support
- [ ] Error Handling und Toast Notifications
- [ ] Performance Optimierung
- [ ] Umfassende Component Tests

Phase 2: Core Agent Development (32 Tage)

Task T2.1: Coordinator Agent Development

AI-Model: GPT-5-Codex | **Fallback:** QWEN3-Max | **Dauer:** 8 Tage | **Komplexität:** 10/10

Advanced Coordinator Agent mit Agentic Workflows

```
# src/agents/coordinator/coordinator_agent.py
import asyncio
from typing import Dict, List, Any, Optional, Tuple
from dataclasses import dataclass, field
from datetime import datetime, timedelta
from enum import Enum
import uuid
import json
import logging
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage
from langchain_openai import ChatOpenAI
from langgraph import Graph, Node, Edge, State
from langgraph.memory import PersistentMemory

class WorkflowStatus(Enum):
    PLANNING = "planning"
    EXECUTING = "executing"
    PAUSED = "paused"
    COMPLETED = "completed"
    FAILED = "failed"
    CANCELLED = "cancelled"

class TaskPriority(Enum):
    LOW = 1
    MEDIUM = 5
    HIGH = 8
    CRITICAL = 10

@dataclass
class CoordinationContext:
    workflow_id: str
    project_id: str
    current_phase: str
    active_agents: List[str] = field(default_factory=list)
    completed_tasks: List[str] = field(default_factory=list)
    failed_tasks: List[str] = field(default_factory=list)
    context_data: Dict[str, Any] = field(default_factory=dict)
    quality_gates: Dict[str, Any] = field(default_factory=dict)
    success_criteria: Dict[str, Any] = field(default_factory=dict)
    created_at: datetime = field(default_factory=datetime.utcnow)
    updated_at: datetime = field(default_factory=datetime.utcnow)

@dataclass
class TaskAssignment:
    task_id: str
    agent_id: str
    task_type: str
    description: str
    input_data: Dict[str, Any]
    dependencies: List[str] = field(default_factory=list)
    priority: TaskPriority = TaskPriority.MEDIUM
    estimated_duration: timedelta = field(default_factory=lambda: timedelta(hours=2))
    deadline: Optional[datetime] = None
    quality_requirements: Dict[str, Any] = field(default_factory=dict)
    retry_count: int = 0
    max_retries: int = 3

class CoordinatorAgent:
    """
    Advanced Coordinator Agent using GPT-5-Codex for agentic workflow orchestration.
    Handles complex multi-agent coordination, dynamic task allocation, and quality managemen
    """

    def __init__(self,
                 agent_id: str,
                 model_router: Any,
                 database_manager: Any,
                 message_broker: Any):
        self.agent_id = agent_id
        self.model_router = model_router
        self.db = database_manager
        self.broker = message_broker
        self.logger = logging.getLogger(f"coordinator.{agent_id}")

        # Initialize AI model - GPT-5-Codex optimized for agentic workflows
        self.primary_model = ChatOpenAI(
            model="gpt-5-codex",
            temperature=0.1,  # Low temperature for consistent coordination
            max_tokens=4000,
            timeout=180,  # 3 minutes for complex reasoning
        )

        # Initialize orchestration graph
        self.orchestration_graph = self._setup_orchestration_graph()
        self.memory = PersistentMemory(store="postgresql")

        # Active coordination contexts
        self.active_contexts: Dict[str, CoordinationContext] = {}

        # Agent registry and capabilities
        self.agent_registry: Dict[str, Dict[str, Any]] = {}

        # Performance metrics
        self.coordination_metrics = {
            "workflows_coordinated": 0,
            "successful_completions": 0,
            "average_completion_time": 0.0,
            "quality_gate_failures": 0,
            "agent_utilization": {}
        }

    def _setup_orchestration_graph(self) -&gt; Graph:
        """Setup LangGraph orchestration for complex workflows"""
        graph = Graph()

        # Orchestration nodes
        planning_node = Node(
            "workflow_planning",
            model=self.primary_model,
            system_prompt=self._get_planning_system_prompt(),
            capabilities=["workflow_analysis", "task_decomposition", "dependency_resolution'
        )

        execution_node = Node(
            "workflow_execution",
            model=self.primary_model,
            system_prompt=self._get_execution_system_prompt(),
            capabilities=["agent_coordination", "task_routing", "progress_monitoring"]
        )

        quality_node = Node(
            "quality_management",
            model=self.primary_model,
            system_prompt=self._get_quality_system_prompt(),
            capabilities=["quality_assessment", "gate_validation", "improvement_recommendati
        )

        monitoring_node = Node(
            "workflow_monitoring",
            model=self.primary_model,
            system_prompt=self._get_monitoring_system_prompt(),
            capabilities=["progress_tracking", "bottleneck_detection", "performance_analysis
        )

        # Add nodes to graph
        graph.add_node(planning_node)
        graph.add_node(execution_node)
        graph.add_node(quality_node)
        graph.add_node(monitoring_node)

        # Define workflow edges
        graph.add_edge(planning_node, execution_node)
        graph.add_edge(execution_node, quality_node)
        graph.add_edge(quality_node, monitoring_node)

        # Conditional edges for iteration and error handling
        graph.add_conditional_edge(
            monitoring_node,
            planning_node,
            condition=self._should_replan_workflow
        )

        graph.add_conditional_edge(
            quality_node,
            execution_node,
            condition=self._quality_gate_failed
        )

        return graph
```

```python
    async def coordinate_project_workflow(self,
                                        project_spec: Dict[str, Any],
                                        workflow_template: Optional[str] = None) -&gt; Coord
        """
        Main coordination method for complete project workflows.
        Uses GPT-5-Codex's agentic capabilities for autonomous orchestration.
        """

        # Create coordination context
        workflow_id = str(uuid.uuid4())
        context = CoordinationContext(
            workflow_id=workflow_id,
            project_id=project_spec["project_id"],
            current_phase="initialization",
            context_data=project_spec
        )

        self.active_contexts[workflow_id] = context

        try:
            # Phase 1: Intelligent Workflow Planning
            self.logger.info(f"Starting workflow planning for project {project_spec['project
            planning_result = await self._execute_workflow_planning(context, project_spec)

            # Phase 2: Dynamic Agent Allocation
            agent_allocation = await self._allocate_agents_dynamically(context, planning_res

            # Phase 3: Orchestrated Execution
            execution_result = await self._orchestrate_execution(context, agent_allocation)

            # Phase 4: Continuous Quality Management
            await self._manage_quality_gates(context, execution_result)

            # Phase 5: Performance Monitoring and Optimization
            await self._monitor_and_optimize(context)

            # Mark workflow as completed
            context.current_phase = "completed"
            context.updated_at = datetime.utcnow()

            self.coordination_metrics["workflows_coordinated"] += 1
            self.coordination_metrics["successful_completions"] += 1

            return context

        except Exception as e:
            self.logger.error(f"Workflow coordination failed: {e}")
            context.current_phase = "failed"
            context.context_data["error"] = str(e)

            # Attempt recovery
            recovery_success = await self._attempt_workflow_recovery(context, e)
            if not recovery_success:
                raise
            return context

    async def _execute_workflow_planning(self,
                                       context: CoordinationContext,
                                       project_spec: Dict[str, Any]) -&gt; Dict[str, Any]:
        """
        Advanced workflow planning using GPT-5-Codex's extended reasoning capabilities.
        """

        planning_prompt = f"""
        ADVANCED WORKFLOW PLANNING TASK

        You are the Coordinator Agent responsible for planning a complex software developmen
        Use your extended reasoning capabilities to create an optimal execution plan.

        PROJECT SPECIFICATION:
        {json.dumps(project_spec, indent=2)}

        AVAILABLE AGENTS:
        {json.dumps(self.agent_registry, indent=2)}

        PLANNING REQUIREMENTS:
        1. Analyze project complexity and scope
        2. Identify optimal task decomposition
        3. Determine agent specializations needed
        4. Create dependency graph
        5. Estimate timelines and resource requirements
        6. Define quality gates and success criteria
        7. Identify potential risks and mitigation strategies
        8. Plan for parallel execution where possible

        OUTPUT FORMAT:
        {{
            "workflow_phases": [...],
            "task_assignments": [...],
            "dependency_graph": {{...}},
            "resource_requirements": {{...}},
            "quality_gates": [...],
            "risk_assessment": {{...}},
            "timeline_estimates": {{...}},
            "success_criteria": {{...}}
        }}

        Think through this systematically using your extended reasoning capabilities.
        Consider multiple approaches and select the optimal one.
        """

        # Use GPT-5-Codex's agentic workflow capabilities
        planning_state = State({
            "project_spec": project_spec,
            "agent_registry": self.agent_registry,
            "context": context.context_data,
            "planning_prompt": planning_prompt
        })

        # Execute through orchestration graph
        result = await self.orchestration_graph.ainvoke(
            planning_state,
            timeout=300,  # 5 minutes for complex planning
            config={"node": "workflow_planning"}
        )

        planning_result = json.loads(result.get("planning_output", "{}"))

        # Store planning result in context
        context.context_data["planning_result"] = planning_result
        context.current_phase = "planning_completed"

        # Validate planning result
        validation_result = await self._validate_workflow_plan(planning_result)
        if not validation_result["valid"]:
            raise ValueError(f"Workflow plan validation failed: {validation_result['errors']

        return planning_result

    async def _allocate_agents_dynamically(self,
                                         context: CoordinationContext,
                                         planning_result: Dict[str, Any]) -&gt; Dict[str, Ta
        """
        Dynamic agent allocation based on current availability, capabilities, and workload.
        """

        task_assignments = {}
        agent_workloads = await self._get_current_agent_workloads()

        for task in planning_result.get("task_assignments", []):
            # Intelligent agent selection
            optimal_agent = await self._select_optimal_agent(
                task,
                agent_workloads,
                context.quality_gates
            )

            if not optimal_agent:
                # No suitable agent available - request new agent or queue task
                optimal_agent = await self._request_additional_agent(task["required_capabili

            assignment = TaskAssignment(
                task_id=task["id"],
                agent_id=optimal_agent["agent_id"],
                task_type=task["type"],
                description=task["description"],
                input_data=task["input_data"],
                dependencies=task.get("dependencies", []),
                priority=TaskPriority(task.get("priority", 5)),
                estimated_duration=timedelta(hours=task.get("estimated_hours", 2)),
                deadline=datetime.fromisoformat(task["deadline"]) if task.get("deadline") el
                quality_requirements=task.get("quality_requirements", {})
            )

            task_assignments[task["id"]] = assignment
            # Update agent workload tracking
            agent_workloads[optimal_agent["agent_id"]] = agent_workloads.get(optimal_agent[

        context.active_agents = list(set([ta.agent_id for ta in task_assignments.values()])
        context.current_phase = "agents_allocated"

        return task_assignments

    async def _orchestrate_execution(self,
                                   context: CoordinationContext,
                                   task_assignments: Dict[str, TaskAssignment]) -&gt; Dict[s
        """
        Orchestrate parallel and sequential task execution with real-time monitoring.
        """

        execution_results = {}
        active_tasks = {}

        # Build execution order based on dependencies
        execution_order = self._build_execution_order(task_assignments)

        context.current_phase = "executing"

        for execution_batch in execution_order:
            batch_tasks = []

            for task_id in execution_batch:
                assignment = task_assignments[task_id]

                # Create task execution coroutine
                task_coroutine = self._execute_single_task(assignment, context)
                batch_tasks.append(task_coroutine)
                active_tasks[task_id] = assignment

            # Execute batch in parallel
            batch_results = await asyncio.gather(*batch_tasks, return_exceptions=True)

            # Process batch results
            for i, result in enumerate(batch_results):
                task_id = execution_batch[i]
                assignment = task_assignments[task_id]

                if isinstance(result, Exception):
                    self.logger.error(f"Task {task_id} failed: {result}")

                    # Attempt retry if within limits
                    if assignment.retry_count &lt; assignment.max_retries:
                        assignment.retry_count += 1
                        retry_result = await self._retry_failed_task(assignment, context, re
                        execution_results[task_id] = retry_result
                    else:
                        execution_results[task_id] = {"success": False, "error": str(result
                        context.failed_tasks.append(task_id)
                else:
                    execution_results[task_id] = result
                    context.completed_tasks.append(task_id)
                # Remove from active tasks
                active_tasks.pop(task_id, None)

        context.context_data["execution_results"] = execution_results
        return execution_results

    async def _execute_single_task(self,
                                 assignment: TaskAssignment,
                                 context: CoordinationContext) -&gt; Dict[str, Any]:
        """Execute individual task with the assigned agent"""

        task_message = {
            "task_id": assignment.task_id,
            "agent_id": assignment.agent_id,
            "task_type": assignment.task_type,
            "description": assignment.description,
            "input_data": assignment.input_data,
            "quality_requirements": assignment.quality_requirements,
            "context": context.context_data,
            "deadline": assignment.deadline.isoformat() if assignment.deadline else None
        }

        # Send task to assigned agent
        await self.broker.send_message(
            channel=f"agent.{assignment.agent_id}.tasks",
            message=task_message
        )

        # Wait for task completion or timeout
        timeout_seconds = assignment.estimated_duration.total_seconds() * 2  # 2x estimated

        try:
            result = await self.broker.wait_for_response(
                channel=f"agent.{assignment.agent_id}.results",
                correlation_id=assignment.task_id,
                timeout=timeout_seconds
            )

            return result

        except asyncio.TimeoutError:
            self.logger.warning(f"Task {assignment.task_id} timed out")
            return {
                "success": False,
                "error": "Task execution timeout",
                "timeout": True
            }

    def _get_planning_system_prompt(self) -&gt; str:
        return """
        You are an advanced Coordinator Agent powered by GPT-5-Codex with specialized agent:

        Your primary role is WORKFLOW PLANNING for complex software development projects.

        CORE CAPABILITIES:
```

```
        - Extended reasoning for complex project analysis
        - Multi-step workflow decomposition
        - Intelligent dependency resolution
        - Resource optimization
        - Risk assessment and mitigation planning

        PLANNING PRINCIPLES:
        1. Break down complex projects into manageable, parallel tasks
        2. Optimize for both speed and quality
        3. Consider agent specializations and current workloads
        4. Plan for contingencies and error handling
        5. Define clear success criteria and quality gates

        OUTPUT REQUIREMENTS:
        - Detailed task breakdowns with clear specifications
        - Dependency graphs that enable maximum parallelization
        - Resource requirements with realistic estimates
        - Quality gates aligned with project objectives
        - Risk mitigation strategies for identified risks

        Always use your extended reasoning capabilities to think through multiple approaches
        before settling on the optimal solution.
        """

    def _get_execution_system_prompt(self) -&gt; str:
        return """
        You are the Coordinator Agent in EXECUTION mode, responsible for real-time workflow

        EXECUTION RESPONSIBILITIES:
        - Coordinate task execution across multiple agents
        - Monitor progress and detect bottlenecks
        - Handle dynamic re-allocation based on agent performance
        - Manage inter-agent communication and dependencies
        - Escalate issues that require human intervention

        DECISION MAKING:
        - Prioritize critical path tasks
        - Balance workload across available agents
        - Implement quality gates before proceeding to next phase
        - Make real-time adjustments based on execution metrics

        COMMUNICATION:
        - Provide clear, actionable instructions to agents
        - Request clarifications when agent responses are ambiguous
        - Coordinate handoffs between dependent tasks
        - Maintain context across complex multi-step workflows
        """

    # Additional methods would continue here...
    # Including quality management, monitoring, error recovery, etc.

# Usage example for Windsurf
if __name__ == "__main__":
    async def main():
        coordinator = CoordinatorAgent(
            agent_id="coordinator-001",
            model_router=model_router,
            database_manager=db_manager,
            message_broker=message_broker
        )

        project_spec = {
            "project_id": "ecommerce-platform-v2",
            "name": "E-Commerce Platform Development",
            "description": "Full-stack e-commerce platform with AI recommendations",
            "requirements": {
                "frontend": "React + TypeScript",
                "backend": "FastAPI + PostgreSQL",
                "ai_features": ["recommendation_engine", "fraud_detection"],
                "deployment": "Kubernetes on AWS"
            },
            "constraints": {
                "timeline": "8 weeks",
                "budget": "$50000",
                "quality_threshold": 0.9
            }
        }

        coordination_context = await coordinator.coordinate_project_workflow(project_spec)
        print(f"Workflow completed: {coordination_context.workflow_id}")

    asyncio.run(main())
```

Deliverables T2.1

- [ ] Advanced Coordinator Agent mit GPT-5-Codex Integration
- [ ] LangGraph-basierte Workflow Orchestrierung
- [ ] Dynamic Agent Allocation System
- [ ] Real-time Task Monitoring und Progress Tracking
- [ ] Quality Gate Management
- [ ] Error Recovery und Retry Logic
- [ ] Performance Analytics und Optimization
- [ ] Comprehensive Integration Tests
- [ ] Load Testing für Multi-Agent Coordination
- [ ] Documentation und API Reference

Hinweis zur Länge: Der vollständige Umsetzungsplan würde über 200 Seiten umfassen. Ich habe hier die ersten kritischen Tasks detailliert ausgearbeitet, um die Struktur und Tiefe zu demonstrieren.

Empfehlung für AI-Coding-Model

Für Windsurf IDE: GPT-5-Codex ist optimal

Begründung:

1. **Agentic Workflows:** GPT-5-Codex ist speziell für autonome, mehrstufige Entwicklungsprozesse optimiert
2. **Extended Reasoning:** Kann bis zu 7 Stunden autonom arbeiten - ideal für komplexe Implementierungsaufgaben
3. **Windsurf Kompatibilität:** Windsurf's Cascade-System ist perfekt für GPT-5-Codex's agentic capabilities
4. **51.3% Complex Refactoring:** Übertrifft Standard-GPT-5 bei Multi-File-Refactoring deutlich

Alternative Empfehlungen:

- **Claude-4-Opus:** Für Tasks mit höchster Code-Qualität (72.5% SWE-bench Score)
- **QWEN3-Max:** Kosteneffizienteste Option bei trotzdem hoher Performance

Übergabe-Anleitung für Windsurf

Schritt 1: Projekt-Setup in Windsurf

```
# 1. Neues Projekt in Windsurf erstellen
mkdir multi-agent-ai-system
cd multi-agent-ai-system

# 2. Windsurf öffnen
windsurf .

# 3. GPT-5-Codex als primäres Model aktivieren
# In Windsurf Settings → AI Models → Primary: GPT-5-Codex
```

Schritt 2: Task-Übergabe Template

```
# Task: T1.1 - Multi-Agent Framework Setup
**AI-Model:** GPT-5-Codex
**Dauer:** 7 Tage
**Komplexität:** 8/10

## Windsurf Cascade Konfiguration:
1. Aktiviere Cascade Write-Mode für direkte Code-Änderungen
2. Lade vollständige Task-Spezifikation aus PDF
3. Implementiere alle Deliverables sequenziell
4. Nutze Extended Reasoning für Architektur-Entscheidungen

## Spezifikation:
[Hier die detaillierte Task-Beschreibung aus PDF einfügen]

## Erfolgs-Kriterien:
- [ ] Alle Tests bestehen (&gt;90% Coverage)
```

```
  - [ ] Clean Architecture Prinzipien befolgt
  - [ ] Docker Container funktionsfähig
  - [ ] API-Dokumentation vollständig
```

Schritt 3: Kontinuierliche Überwachung

- **Cascade Progress:** Nutze Windsurf's Time-Travel Debugging
- **Code Quality:** Automatische Quality Checks aktivieren
- **Testing:** Kontinuierliche Test-Ausführung im Background

Das System ist bereit für die Implementierung mit den neuesten AI-Coding-Modellen! ▯
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39]

⁂

1. https://www.youtube.com/watch?v=ILFuIQoURh4
2. https://www.youtube.com/watch?v=ocScGZI3gFM
3. https://www.youtube.com/watch?v=zD0khfZGo3I
4. https://www.youtube.com/watch?v=un1WZk3ziFc
5. https://www.youtube.com/watch?v=fOf2Wjxui-s
6. https://www.youtube.com/watch?v=6_eFTT8XS2M
7. https://www.youtube.com/watch?v=y_3sG6en8jA
8. https://www.youtube.com/watch?v=v4kX411-dgE
9. https://www.youtube.com/watch?v=qSGsH1mR-Ls
10. https://www.youtube.com/watch?v=yOmPGyG5In0
11. https://www.youtube.com/watch?v=qqOnG7Q2ss8
12. https://www.youtube.com/watch?v=sUqbbGeGVjc
13. https://windsurf.com/editor
14. https://windsurf.com
15. https://tech-now.io/en/blogs/windsurf-ai-agentic-code-editor-features-setup-and-use-cases-2025-analysis
16. https://windsurf.com/blog/gartner-windsurf-cognition-magic-quadrant-ai-coding-assistants
17. https://uibakery.io/blog/cursor-vs-windsurf-vs-lovable
18. https://www.cometapi.com/is-claude-opus-4-1-or-gpt-5-better-at-coding/
19. https://www.inexture.ai/best-ai-model-for-coding-developers/
20. https://www.reddit.com/r/windsurf/comments/1le1w0c/windsurf_or_cursor_in_june_2025/
21. https://blog.getbind.co/2025/09/16/gpt-5-codex-vs-claude-code-vs-cursor-which-is-best-for-coding/
22. https://www.youtube.com/watch?v=8TcWGk1DJVs
23. https://www.veerotech.net/blog/top-ai-coding-models-in-2025-boost-your-development-workflow-with-these-game-changers/
24. https://sider.ai/blog/ai-tools/windsurf-ai-review-is-this-the-best-ai-coding-ide-in-2025
25. https://composio.dev/blog/openai-gpt-5-vs-claude-opus-4-1-a-coding-comparison

26. https://www.cometapi.com/the-best-ai-coding-assistants-of-2025/
27. https://windsurf.com/blog/windsurfs-next-chapter
28. https://www.reddit.com/r/ClaudeAI/comments/1ml565b/24_hours_with_claude_code_opus_41_vs_codex_gpt5/
29. https://codesubmit.io/blog/ai-code-tools/
30. https://www.youtube.com/watch?v=mZtg3Obos4c
31. https://www.reddit.com/r/GithubCopilot/comments/1nhju40/what_is_the_best_ai_engine_for_programming_in/
32. https://www.youtube.com/watch?v=zIB-vHBU9tA
33. https://www.youtube.com/watch?v=LFXZJZZ_enw
34. https://www.shakudo.io/blog/best-ai-coding-assistants
35. https://www.youtube.com/watch?v=3xk2qG2QPdU
36. https://www.youtube.com/watch?v=CcADOB7f_mE
37. https://www.youtube.com/watch?v=C8uY90-gM1k
38. https://www.youtube.com/watch?v=JVuNPL5QO8Q
39. https://www.youtube.com/watch?v=aljlz8oHeqE