Agent Skills for Context Engineering

Beispielprojekte

Vollständige Ausarbeitung aller 5 Beispielprojekte
mit Code-Implementierungen, Architektur-Diagrammen
und Skills-Mapping

Februar 2026

Inhaltsverzeichnis

Übersicht

Context Engineering beschreibt die Kunst, AI Agents mit dem richtigen Kontext zur richtigen Zeit zu versorgen – nicht zu viel, nicht zu wenig, in der richtigen Struktur. Während klassisches Prompt Engineering auf einzelne Interaktionen zielt, adressiert Context Engineering die systemischen Herausforderungen mehrstufiger Agent-Workflows: begrenzte Kontextfenster, Tool-Koordination, Memory-Management und Qualitätssicherung über viele Turns hinweg.

Dieses Dokument arbeitet fünf Beispielprojekte aus, die diese Prinzipien in produktionsnahen Systemen demonstrieren – vom persönlichen Knowledge Management über LLM-Evaluation bis hin zu Multi-Agent-Buchproduktion. Jedes Kapitel enthält die vollständige Architektur, Code-Implementierungen und ein Skills-Mapping, das theoretische Konzepte mit konkreten Design-Entscheidungen verbindet.

Kap.	Projekt	Kern-Innovation
1	Digital Brain Skill	Progressive Disclosure, Append-Only JSONL
2	LLM-as-Judge Skills	Type-safe Evaluation, Position Bias Mitigation
3	Book SFT Pipeline	Two-Tier Chunking, Prompt Diversity
4	X-to-Book System	Supervisor Pattern, Temporal Knowledge Graph
5	Interleaved Thinking	M2.1 Reasoning Trace Analysis, Auto-Optimization

1 Digital Brain Skill – Persönliches Operating System

Personal Operating System für Founders, Creators und Builder. Ein strukturiertes Knowledge Management System für AI-assistierte Produktivität, konzipiert als Granaria Code Plugin.

1.1 Architektur

Das System besteht aus sechs isolierten Modulen, die unabhängig voneinander funktionieren:

```
digital-brain/
  SKILL.md                    # Granaria Code kompatibel
  SKILLS-MAPPING.md           # Skills-Anwendung dokumentiert
  identity/                   # Personal Brand & Voice
    voice.md | brand.md | values.yaml | bio-variants.md
  content/                    # Content Creation Hub
    ideas.jsonl | posts.jsonl | calendar.md | engagement.jsonl
  knowledge/                  # Personal Knowledge Base
    bookmarks.jsonl | learning.yaml | competitors.md | research/
  network/                    # Relationship Management
    contacts.jsonl | interactions.jsonl | circles.yaml
  operations/                 # Produktivitaetssystem
    todos.md | goals.yaml | meetings.jsonl | metrics.jsonl
  agents/                     # Automation Scripts
    scripts/ (weekly_review.py, content_ideas.py, ...)
```

1.2 Design-Prinzipien

Progressive Disclosure: 3-Level Loading – `SKILL.md` → `MODULE.md` → Data Files. Nur das Minimum wird ins Kontextfenster geladen.

Append-Only Memory: JSONL mit Schema-First-Lines. Daten werden nie gelöscht, nur angehangen – perfekt für Agent-Workflows.

Module Isolation: Jede Domain (Identity, Content, Knowledge, Network, Operations, Agents) arbeitet unabhängig ohne Cross-Contamination.

Voice First: Vor jeder Content-Generierung wird `voice.md` gelesen, um konsistenten Ton sicherzustellen.

1.3 File-Format-Konventionen

Format	Verwendungszweck
`.jsonl`	Append-only Logs – Agent-friendly, History wird bewahrt
`.yaml`	Strukturierte Konfiguration – Human-readable Hierarchien
`.md`	Narrative Inhalte – Editierbar, Rich Formatting
`.xml`	Komplexe Prompts – Klare Struktur für Agents

1.4 Automation: weekly_review.py

Das Review-Script analysiert automatisch Content-Performance, Netzwerk-Aktivität und Task-Fortschritt:

```
def load_jsonl(filepath):
    """Load JSONL, skip schema lines."""
    items = []
    with open(filepath, 'r') as f:
        for line in f:
            data = json.loads(line)
            if '_schema' not in data:
                items.append(data)
    return items

def analyze_content(week_start):
    posts = load_jsonl(BRAIN_ROOT / 'content' / 'posts.jsonl')
    week_posts = [p for p in posts
                  if p.get('published', '') >= week_start]
    return {'posts_published': len(week_posts), ...}
```

1.5 Automation: content_ideas.py

Generiert Content-Ideen basierend auf Top-Performing Content (Engagement-Score: likes + comments×2 + reposts×3), Recent Bookmarks und undeveloped Ideas (`status='raw'`):

```
def get_top_content(posts, n=5):
    for p in posts:
        eng = p.get('engagement', {})
        p['score'] = (eng.get('likes', 0)
                      + eng.get('comments', 0) * 2
                      + eng.get('reposts', 0) * 3)
    return sorted(posts, key=lambda x: x['score'],
```

```
                    reverse=True)[:n]
```

1.6 Usage-Beispiele

Content Creation: User fragt nach einem X-Thread über AI. Agent liest `voice.md` für Tone, prüft `brand.md` ob das Thema ein Content-Pillar ist, referenziert `posts.jsonl` für erfolgreiche Formate, dann generiert den Thread.

Meeting Prep: User sagt „Prepare me for my call with Sarah“. Agent sucht `contacts.jsonl`, holt History aus `interactions.jsonl`, prüft `todos.md` für offene Punkte, generiert ein Pre-Meeting Brief.

1.7 Skills-Mapping

Skill	Anwendung
`context-fundamentals`	Progressive Disclosure, Attention Budget Management
`context-optimization`	Module Separation, Just-in-Time Loading
`memory-systems`	JSONL Append-Only Logs, Structured Recall
`tool-design`	Self-contained Automation Scripts
`multi-agent-patterns`	Script-basierte Automation Agents
`evaluation`	Weekly Review Metrics

2 LLM-as-Judge Skills – Production-Ready Evaluation

TypeScript-Implementation von LLM-Evaluation-Tools basierend auf Eugene Yan's Research. Vercel AI SDK 6, 19 passing Tests. Drei Core Tools: Direct Score, Pairwise Compare, Rubric Generation.

2.1 Architektur

```
  Skills (MD docs) --> Prompts (templates) --> Tools (TS impl)
                                                      |
                                              EvaluatorAgent
                                  score() | compare() | generateRubric()
                                                      |
                                                OpenAI GPT API
```

2.2 Tool 1: Direct Score

Evaluiert eine einzelne Response gegen definierte Kriterien mit numerischen Scores. Einsatz: Factual Accuracy, Instruction Following, Content Quality, Compliance.

```typescript
// Zod Schema (gekuerzt)
const DirectScoreInputSchema = z.object({
  response: z.string(),
  prompt: z.string(),
  criteria: z.array(z.object({
    name: z.string(),
    description: z.string(),
    weight: z.number().min(0).max(1).default(1)
  })).min(1),
  rubric: z.object({
    scale: z.enum(['1-3', '1-5', '1-10']).default('1-5'),
    levelDescriptions: z.record(z.string()).optional()
  }).optional()
});
```

Jedes Kriterium wird gewichtet. Chain-of-Thought Justification ist Pflicht – der Judge muss Evidenz aus der Response extrahieren und Verbesserungsvorschläge liefern:

```
// Gewichtete Score-Berechnung
const totalWeight = criteria.reduce(
    (sum, c) => sum + c.weight, 0);
const weightedSum = scores.reduce((sum, s) => {
  const criterion = criteria.find(
      c => c.name === s.criterion);
  return sum + (s.score * (criterion?.weight || 1));
}, 0);
weightedScore = weightedSum / totalWeight;
```

2.3 Tool 2: Pairwise Compare

Vergleicht zwei Responses mit automatischer Position Bias Mitigation. Key Insight: LLMs bevorzugen systematisch die erste präsentierte Response.

Lösung: Automatisches Position Swapping (A vs B, dann B vs A) mit Konsistenz-Check. Bei Inkonsistenz wird ein Tie gemeldet und der Confidence Score sinkt:

```
// Position Swap Logik
const resultAB = await evaluate(responseA, responseB);
const resultBA = await evaluate(responseB, responseA);
if (resultAB.winner !== resultBA.winner) {
  return { winner: 'tie', confidence: 0.5,
    note: 'Position bias detected' };
}
```

2.4 Tool 3: Rubric Generation

Erstellt domain-spezifische Evaluation Standards automatisch. Gibt eine Rubric mit Skala, Level-Beschreibungen und Kriterien zurück, die direkt an den Direct Score oder Pairwise Compare übergeben werden kann.

2.5 Research Insights (Eugene Yan)

2.6 Skills-Mapping

Insight	Implementierung
Direct Scoring best für objektive Kriterien	`directScore` Tool mit Rubric Support
Pairwise zuverlässiger für Präferenzen	`pairwiseCompare` mit Position Swapping
Position Bias beeinflusst Pairwise	Automatisches Position Swapping
Chain-of-Thought verbessert Zuverlässigkeit	Alle Evaluationen mit Justification + Evidence
Klare Rubrics reduzieren Varianz	`generateRubric` Tool

Skill	Anwendung
`advanced-evaluation`	LLM-as-Judge Patterns: Direct Scoring, Pairwise, Bias Mitigation
`tool-design`	Type-safe Tools mit Zod Schemas, strukturierte Fehlerbehandlung
`context-fundamentals`	Structured Output Design, klare Tool-Beschreibungen

3 Book SFT Pipeline – Author Style Transfer Training

Training kleiner Modelle (8B) für Author-Style-Transfer. Case Study Gertrude Stein: 70% Human Score auf Pangram bei nur $2 Gesamtkosten.

3.1 Core Algorithm: Segmentation

Problem: Bücher sind zu lang für einzelne Training Examples.

Lösung: Two-Tier Chunking mit Overlap. Kleinere Chunks (150–400 Wörter) erzeugen mehr Beispiele und besseren Style Transfer als größere.

```
def segment_text(text, min_words=150, max_words=400):
    paragraphs = [p.strip()
                  for p in text.split('\n\n') if p.strip()]
    chunks, buffer, buffer_words = [], [], 0

    for para in paragraphs:
        para_words = len(para.split())
        if (buffer_words + para_words > max_words
                and buffer_words >= min_words):
            chunks.append(Chunk(
                '\n\n'.join(buffer), buffer_words,
                len(chunks)))
            # Letzten Paragraph fuer Overlap behalten
            buffer = ([buffer[-1], para]
                      if buffer else [para])
            buffer_words = (len(buffer[-2].split())
                            + para_words)
        else:
            buffer.append(para)
            buffer_words += para_words
    return chunks
```

Key Insight: Overlap zwischen Chunks maximiert Training Examples ohne Redundanz. Der letzte Paragraph jedes Chunks wird als erster Paragraph des nächsten wiederverwendet.

3.2 Prompt Diversity – Verhindert Memorization

15+ verschiedene Prompt-Templates verhindern, dass das Modell spezifische Formulierungen auswendig lernt, und forcieren echtes Style Learning:

```
PROMPT_TEMPLATES = [
    "Write a passage in the style of {author}: {desc}",
    "Channel {author}'s voice to write about: {desc}",
    "In {author}'s distinctive prose style, describe: {desc}",
    "Write this scene as {author} would have: {desc}",
    "Using {author}'s rhythmic technique, write: {desc}",
]

def build_examples(chunk, instruction, author, variants=2):
    examples = []
    for i in range(variants):
        system = SYSTEM_PROMPTS[i % len(SYSTEM_PROMPTS)]
        template = PROMPT_TEMPLATES[
            (chunk.id + i) % len(PROMPT_TEMPLATES)]
        user = template.format(
            author=author, desc=instruction)
        examples.append(
            TrainingExample(system, user, chunk.text))
    return examples
```

3.3 Tinker Integration

Konvertierung in Tinker Datum Format für LoRA-Training. Key Insight: Weights 0 für Prompt-Tokens, 1 für Completion-Tokens. Das Modell lernt zu generieren, nicht den Prompt zu wiederholen:

```
def build_tinker_datum(example, tokenizer, renderer):
    model_input, weights = \
        renderer.build_supervised_example(messages)
    input_tokens = model_input.to_ints()
    target_tokens = input_tokens[1:]    # Next-Token Pred.
    weights = weights[1:]               # Weights ausrichten
    return {
        "model_input": input_tokens[:-1],
        "loss_fn_inputs": {
            "target_tokens": target_tokens,
            "weights": weights
        }
    }
```

3.4 Validation: Beweist Style vs. Content

Moderne Szenarien testen, ob das Modell Style oder Content gelernt hat. Wenn es Gertrude Steins Stil auf „barista making lattes“ anwenden kann, wurde Style transferiert:

```
MODERN_TEST_SCENARIOS = [
    "Write about a barista making lattes",
    "Describe two lovers communicating via text messages",
    "Write about someone anxious about climate change",
]

def validate_style_transfer(output, training_data_path):
    with open(training_data_path) as f:
        training_text = f.read()
    phrases = [output[i:i+50]
               for i in range(0, len(output)-50, 25)]
    exact_matches = sum(
        1 for p in phrases if p in training_text)
    return {
        'originality_score':
            1.0 - exact_matches / max(len(phrases), 1),
        'is_original': exact_matches < 3
    }
```

3.5 Skills-Mapping

Skill	Anwendung
`project-development`	Task-Model Fit Analyse, Pipeline-Architektur
`context-compression`	Intelligente Segmentierung (Two-Tier Chunking)
`multi-agent-patterns`	Staged Pipeline: acquire → prepare → process → parse
`evaluation`	Validation Methodology (Modern Scenario Testing)

4 X-to-Book System – Multi-Agent Architecture

Multi-Agent-System: Monitort X-Accounts, generiert täglich synthetisierte Bücher. Das umfangreichste Beispiel – demonstriert die Anwendung fast aller Skills auf ein Production System.

4.1 Architektur: Supervisor Pattern

Vier spezialisierte Agents werden von einem Orchestrator koordiniert. Entscheidung für Supervisor statt Peer-to-Peer: Buchproduktion hat klare sequentielle Phasen, die von zentraler Koordination profitieren.

```
ORCHESTRATOR (50k ctx) -- Routing + Checkpoints, keine Raw Data
    SCRAPER  (20k ctx) -- Per-Account, schreibt zu Filesystem
    ANALYZER (80k ctx) -- Per-Account, liest von Filesystem
    WRITER   (80k ctx) -- Per-Chapter, progressive Disclosure
    EDITOR   (60k ctx) -- Per-Chapter, Quality Gates
            |
    FILE SYSTEM STORAGE
      raw_data/{account}/{date}.json
      analysis/{account}/{date}.json
      drafts/{book_id}/chapter_{n}.md
            |
    TEMPORAL KNOWLEDGE GRAPH
      Entities: Account, Tweet, Theme, Book, Chapter
      Relationships: POSTED, DISCUSSES, AGREES_WITH, SOURCES
      (alle Relationships mit temporal validity periods)
```

4.2 Context Isolation – Kritische Design-Entscheidung

Problem (Supervisor Bottleneck): Orchestrator akkumuliert Context von allen Workers → Context explodiert.

Lösung: Raw Tweet Data passiert *nie* durch den Orchestrator Context. Scraper schreibt direkt ins Filesystem. Andere Agents lesen von dort. Orchestrator erhält nur Phase Summaries.

Jeder Agent hat ein explizites Token Budget: Orchestrator 50k (Routing only), Scraper 20k (ein Account), Writer 80k (ein Chapter). Progressive Disclosure: Book Outline lädt zuerst (lightweight), voller Chapter-Content erst wenn der Writer an diesem spezifischen Chapter arbeitet.

4.3 Temporal Knowledge Graph

Kein einfacher Vector Store, sondern ein temporaler Knowledge Graph. Begründung: Das System muss Fragen beantworten wie „Was war @accounts Position zu AI Regulation im Januar?“ oder „Welche Accounts stimmen bei Crypto überein?“. Das erfordert Relationship Traversal und temporal Validity, die Vector Stores nicht bieten können.

4.4 Tool Consolidation

Drei konsolidierte Tools statt 15+ narrow Tools. Statt separate Tools für `fetch_timeline`, `fetch_thread`, `fetch_engagement`, `search_tweets` gibt es ein `x_data_tool` mit Action-Parameter. Jedes Tool hat explizite Usage Triggers, Parameter-Dokumentation und Error Recovery Guidance.

4.5 Evaluation: 5 gewichtete Dimensionen

Dimension (Gewicht)	Prüfkriterium
Source Accuracy (30%)	Zitate verifiziert gegen Original-Tweets
Thematic Coherence (25%)	Narrativer Fluss und Kohärenz
Completeness (20%)	Theme Coverage vollständig
Insight Quality (15%)	Synthese über bloßes Restatement hinaus
Readability (10%)	Prosa-Qualität

Bücher unter Score 0.7 oder mit Source Accuracy unter 0.8 werden automatisch für Human Review flagged.

4.6 Skills-Mapping

Pattern	Skill-Quelle	Anwendung
Context Isolation via Sub-Agents	multi-agent-patterns	Jeder Agent hat sauberen Context für seine Phase
Filesystem als Koordination	multi-agent-patterns	Vermeidet Context Bloat durch Shared State
Progressive Disclosure	context-fundamentals	Chapter-Content lädt nur bei Bedarf
Temporal Knowledge Graph	memory-systems	Trackt evolvierende Positionen über Zeit
Observation Masking	context-optimization	Raw Tweets nie im Orchestrator Context
Tool Consolidation	tool-design	3 Tools statt 15+
Multi-dim. Evaluation	evaluation	5 gewichtete Qualitätsdimensionen

5 Interleaved Thinking – Reasoning Trace Optimizer

Debug und Optimierung von AI Agents durch Analyse von Reasoning Traces. Nutzt MiniMax M2.1's Interleaved Thinking für tiefen Einblick in Agent Decision-Making.

5.1 Warum M2.1? Interleaved vs. Traditional Thinking

```
Traditional:  Think --> Act --> Act --> Act --> Done
              ^
              (Reasoning nur am Anfang)

M2.1:         Think --> Act --> Think --> Act --> Think --> Act --> Done
              ^               ^                ^
              (Continuous Reasoning zwischen jedem Tool Call)
```

Long Tasks erfordern Fokus über viele Turns. Tool Outputs bringen unerwartete Informationen. Debugging braucht Sichtbarkeit in Decision-Making, nicht nur Outputs. Das `thinking` Block (Anthropic SDK) exponiert dieses Reasoning für die Analyse.

5.2 Komponente 1: TraceCapture

Wraps M2.1 API, um alle Thinking Blocks, Tool Calls und Responses während der Agent-Ausführung zu erfassen:

```python
class TraceCapture:
    def __init__(self, api_key=None,
                 base_url="https://api.minimax.io/anthropic",
                 model="MiniMax-M2.1"):
        self.client = anthropic.Anthropic(
            api_key=api_key, base_url=base_url)

    def run(self, task, system_prompt, tools,
            tool_executor, max_turns=10) -> ReasoningTrace:
        """Execute task and capture full reasoning trace.
        Returns ReasoningTrace with all thinking blocks,
        tool calls, and responses."""
```

```
        ...
```

5.3 Komponente 2: TraceAnalyzer – Pattern Detection

Analysiert Reasoning Traces auf bekannte Failure Modes:

Pattern	Erkennungsmerkmale
context_degradation	Wiederholte Fragen, Widersprüche, vergessene Details
tool_confusion	Falsche Tool-Wahl, falsche Parameter, Fehlinterpretation
instruction_drift	Graduelles Abweichen von Instruktionen/Persona
hallucination	Erfundene Fakten, fabricated Tool Results
goal_abandonment	Topic Drift, Aufgeben, Zielwechsel ohne Grund
circular_reasoning	Gleiche Queries wiederholt, Looping ohne Fortschritt
premature_conclusion	Zu frühes Beenden, unvollständige Antworten
missing_validation	Kein Cross-Checking, keine Fehlerbehandlung

Jedes erkannte Pattern enthält: **Evidence** (Excerpts aus Thinking Blocks), **Severity** (Critical/High/Medium/Low), **Suggestion** (konkreter Verbesserungsvorschlag für den Prompt) und **Confidence** Score.

5.4 Komponente 3: PromptOptimizer

Generiert verbesserte Prompts basierend auf der Analyse. Iteriert bis zur Konvergenz:

```
Execute Agent --> Capture Traces --> Analyze Patterns --> Optimize
      ^                                                      |
      |______________________________________________________|
```

5.5 Komponente 4: OptimizationLoop

Automatisierter Capture → Analyze → Improve → Re-Run Zyklus:

```python
from reasoning_trace_optimizer import (
    OptimizationLoop, LoopConfig)

config = LoopConfig(
    max_iterations=5,
    min_score_threshold=80.0)

loop = OptimizationLoop(config=config)
result = loop.run(
    task="Analyze this codebase and suggest improvements",
    initial_prompt="You are a code reviewer.",
    tools=[read_file_tool, search_tool],
```

```
    tool_executor=execute_tool)

print(f"Improved: {result.initial_score}"
      f" -> {result.final_score}")
print(f"Final prompt:\n{result.final_prompt}")
```

5.6 Komponente 5: SkillGenerator

Konvertiert Learnings automatisch in teilbare Agent Skills. Macht die Optimierungs-Ergebnisse wiederverwendbar für andere Projekte und Agents.

5.7 Skills-Mapping

Skill	Anwendung
`context-fundamentals`	Verständnis von Attention-Mechanik, Context Degradation
`context-optimization`	Analyse von Compaction-Bedarf, Observation Masking
`tool-design`	Erkennung von Tool Confusion, Verbesserung von Tool Descriptions
`multi-agent-patterns`	Identifikation von Koordinationsproblemen
`evaluation`	Quality Scoring, Pattern Detection

Zusammenfassung: Skills-Integration

Jedes Beispiel demonstriert die praktische Anwendung unterschiedlicher Skill-Kombinationen. Zusammen decken sie das gesamte Spektrum des Context Engineering ab:

Beispiel	Primäre Skills	Key Innovation
Digital Brain	context-fundamentals, memory-systems	Progressive Disclosure (3-Level), Append-Only JSONL
LLM-as-Judge	advanced-evaluation, tool-design	Type-safe Evaluation, Position Bias Mitigation
Book SFT Pipeline	project-development, context-compression	Two-Tier Chunking, Prompt Diversity
X-to-Book System	multi-agent-patterns, memory-systems	Supervisor Pattern, Temporal Knowledge Graph
Interleaved Thinking	context-fundamentals, evaluation	M2.1 Reasoning Trace Analysis, Auto-Optimization

Universelle Design-Prinzipien

1. **Progressive Disclosure** – Nur laden, was gerade gebraucht wird. `SKILL.md` → `MODULE.md` → Data Files.
2. **Platform Agnostic** – Übertragbare Prinzipien, kein Vendor Lock-in. Granaria Code, Cursor, beliebige AI Assistants.
3. **Konzeptuelle Fundierung + Praktische Beispiele** – Python-Pseudocode ohne spezifische Dependencies.
4. **File System as Memory** – Just-in-Time Context Loading. Filesystem als Koordinationsmechanismus zwischen Agents.
5. **Evaluation First** – Multi-dimensionale Qualitätsbewertung in jedem Projekt.
6. **Skills Traceability** – `SKILLS-MAPPING.md` in jedem Projekt dokumentiert, welche Architektur-Entscheidung von welchem Skill inspiriert wurde.