Hytale AI Mods: Forschungssynthese & Implementierungsplanung

Executive Summary

Diese Dokumentation vereint zwei komplementäre Forschungsebenen:

1. **Die Gemini-Konzeption**: Theoretische Architektur von 10 innovativen Mod-Konzepten für Hytale, mit "Projekt Aletheia" als Flagship-Projekt
2. **Die BipedalAgent-Implementierung**: Konkrete Java-Architektur für KI-Agenten, die Soul Algorithm + Projekt Aletheia in Hytale verschmelzen

Ziel dieser Synthese: Zeigen, wie die theoretischen Konzepte (Reverse-Turing-Tests, asymmetrische Information, kognitiver Deception) durch die BipedalAgent-Implementierung technisch realisiert werden.

Teil 1: Theoretische Grundlagen (aus Gemini-Konzeption)

1.1 Die Kernthese: Reverse-Turing-Test als Gameplay

Traditional Turing Test: "Kann ein Mensch unterscheiden, ob er mit einer KI spricht?"

Reverse-Turing-Test (Aletheia): "Kann eine KI überzeugend als Mensch in einem sozialen Deduktionsspiel agieren?"

Dies ist nicht einfach eine Umkehrung – es ist eine **Integration von Unsicherheit als Spielmechanik**. Spieler wissen nicht, ob NPCs real sind oder KI-Agenten. Dies schafft:

- **Kognitives Engagement**: Spieler müssen „Tells" erkennen – subtile Marker künstlichen Verhaltens
- **Sozialpsychologische Tiefe**: Paranoia, Misstrauen, kritisches Denken werden mechanisch erzwungen
- **Emergente Narrative**: Jedes Spiel erzählt eine Geschichte von Täuschung und Enthüllung

1.2 Die 10 Mod-Konzepte: Übersicht

Mod	Genre	Kernmechanik	KI-Integration
Aletheia	Social Deduction	Reverse Turing + ARC Puzzles	LLM als "Konsens"-Gegner
Chrono-Echoes	PvP 4D	Packet Recording & Replay	Vorhersage von Spielerbewegungen
Mimic Biome	Survival	Statistische Paranoia	Mouse-Movement-Analyse
Vox Populi	Governance Sim	Rhethoric-Bots	LLM für Dialog-Generation
Architect's Canvas	FPS/RTS Hybrid	Asymmetrische Perspektive	Pathfinding für RTS-AI
Soulbound	Social/Stealth	Body-Possessing Mechanic	ECS-basierte Possession
Entropy Market	Economic Thriller	Insider-Trading Detection	Datenbank-Anomalieerkennung
Tower of Babel	Coop Puzzle	Sprach-Verschlüsselung	NLP für Spracherkennung
Quantum-Lock	Puzzle Horror	Observer-Effect Rendering	Raycasting-basierte Geometrie
Glitch Storm	Battle Royale	"Weaponized Bugs"	Physik-Manipulation

1.3 Aletheia Deep Dive: Map, Tasks, Tells

Map-Philosophie: Torus-Struktur mit dynamischer Okklusion

- Zirkularität verhindert "Camping"
- Dynamische Lichter/Nebel basierend auf Entropie-Meter
- Blöcke reagieren auf Spieler-Stress-Level

Die Tasks: Nicht "Kabel verbinden", sondern kognitive Verifikation

- **Assoziations-Netzwerk**: Zwei Wörter → Verbindungswort finden. LLM bewertet, ob menschlich oder statistisch.
- **Motorischer Glitch**: Mausbewegung nachzeichnen. Perfekte Linie = Bot-Verhalten = Scheitern.

Die "Tells": Subtile Marker künstlichen Verhaltens

- Bit-Crushing: Stimme wird robotischer mit Entfernung (Architekt kann Deepfake-Fähigkeit nutzen)
- Visueller Glitch: 5% Chance auf Texturflackern wenn Architekt lügt
- Timing-Anomalien: Zu schnelle Reaktionen auf Chat-Nachrichten

---

Teil 2: Implementierungsarchitektur (BipedalAgent)

2.1 Die BipedalAgent-Klasse als Brücke

Soul Algorithm (Kognitive Täuschung)
↓
[Federated Memory]
[Dual Process Decision]
[LLM Interface]
↓
[BipedalAgent - Unified Core]
↓
Projekt Aletheia (Verhaltens-Täuschung)
↓
[Physical Mimesis]
[Mouse Look Controller]
[Idle Animation Controller]
[Latency Masking]

Was ist BipedalAgent?

Ein Wrapper-Klasse, die:

1. Soul Algorithm (Gedanken-Ebene) mit Projekt Aletheia (Bewegungs-Ebene) fusioniert
2. Vier 50ms-Zyklen durchläuft: Perception → Decision → Execution → Masking
3. Dual-Process-Denken simuliert: System 1 (Reflex via BehaviorTree) + System 2 (Strategic via LLM)

2.2 Die 4 Phasen des BipedalAgent-Loops

@Override
public void run() {
// PHASE 1: PERCEPTION (beide Systeme)
updatePerception(); // Game Events erfassen

```
// PHASE 2: DECISION (Dual-Process)
updateDecision();    // System 1 (BehaviorTree) oder System 2 (LLM)

// PHASE 3: EXECUTION (Multimodal)
executeActions();    // Chat, Movement, Look, Animation
```

```
// PHASE 4: MASKING (Orchestriert)
maskingOrchestrator.orchestrateLatency();  // Tier 1/2/3
```

}

2.3 Federated Memory: Die Intelligenz des Agenten

Problem: Ein einfaches Turing-Test-Imitator würde sich widerprechen oder vergessen, was er gesagt hat.

Lösung: Zwei-Schicht-Memory

[Short-Term Memory] [Vector DB (MongoDB)]
(LinkedList, in RAM) (Persistent, embeddings)
↓ ↓
Last 10 events Top-3 relevant events
(fast access) (semantic search)
↓ ↓
└──────────→ Memory Context ←──────────┘
↓
LLM Prompt builder

Mechanik:

1. Spieler sagt: "Ich war in der Medizin-Kammer"
2. memory.recordEvent("Player X war in Medizin-Kammer")
3. Event wird in Short-Term UND embedded in Vector DB gespeichert
4. 20 Sekunden später macht Agent eine Aussage über diesen Ort → ruft Kontext ab
5. Agent "erinnert sich" konsistent → erhöht Glaubwürdigkeit

2.4 Dual-Process Decision Engine

System 1 (BehaviorTree):
BehaviorTreeResult reflex = decisionEngine.evaluateBehaviorTree();
// Instant (<5ms): Dodge, Move to Objective, Idle
// Verwendet Hytale GameStateManager für schnelle Umwelt-Abfragen

System 2 (LLM):
CompletableFuture<String> strategy =
llmService.inferenceAsync(prompt, maxTokens=200);
// Async (~500-2000ms): Strategic response, accusations, defenses
// Latency wird durch LatencyMaskingOrchestrator verschleiert

Architektur-Vorteile:

- Reflexe sind INSTANT (verhindert verdächtige Verzögerungen)
- Strategische Entscheidungen sind ASYNC (mit Maskierung)
- Agent wirkt "menschlich" – nicht alle Antworten sind sofort verfügbar

2.5 Latency Masking Orchestrator: Die "Illusion"

Problem: LLM-Inferenz dauert 500-2000ms. Menschen antworten in 100-300ms. → Verdächtig langsam.

Lösung: 3-Tier-Maskierung

Tier	Funktion	Dauer	Mechanik
Tier 1	Emote + Typing Indicator	500ms	`npcPlayer.playEmote("thinking")`
Tier 2	Filler Words	1000 ms	"Hmm, gute Frage..." im Chat
Tier 3	Animation Masking	2000 ms	Idle-Animation, Kopfbewegung simuliert Nachdenken

Ablauf:

1. Spieler stellt Frage: "Warst du in den Vents?"
2. Agent erkennt Urgenz (System 1 würde antworten) → ruft LLM auf
3. **Tier 1** (0-500ms): Emote "thinking" abspielen
4. **Tier 2** (500-1000ms): Chat-Nachricht "Moment..." senden
5. **Tier 3** (1000-2000ms): Idle-Animation mit Blick-Richtungswechsel
6. **Ende (2000ms)**: LLM fertig → echte Response senden
7. **Ergebnis**: Für Zuschauer sieht es aus wie natürliches Nachdenken, nicht wie KI-Verarbeitung

2.6 Physical Mimesis Engine: Projekt Aletheia

Vier Komponenten:

1. PhysicalMimesisEngine

```
physicalMimesis.updateMovement(targetLocation);
// Nicht instant-teleport oder gerade Linie
// Sondern: Kleine Fehler, Umwege, "Überkompensation"
```

Beispiel-Algorithmus:
Soll zu (100, 64, 200)
Füge random "Übershooting" ein: (101, 64, 202)
Dann Korrektur: (99, 64, 199)
Resultat: Realistische, nicht-optimale Pathfinding

2. **MouseLookController: Saccadische Augenbewegung**

```
mouseController.updateMouseLook(npcPlayer, targetDirection);
// Nicht: sofort zum Ziel schauen
// Sondern: Zackige Bewegungen, Jitter, "Ungenauigkeit"
```

Implementation: Statt linearer Interpolation → Cubic Bezier Curve mit randomem Jitter.

3. **IdleAnimationController: Fidgeting & Tells**

```
if (npcPlayer.getVelocity().length() < 0.1) { // Steht still
idleController.updateIdleBehavior(npcPlayer);
}
```

Procedural Animations:

- Alle 3-5 Sekunden: Kopf drehen
- Alle 2-3 Sekunden: Springen (Nervosität)
- Alle 1-2 Sekunden: Block platzieren/entfernen (Fidgeting)

4. **ProceduralPathfinding: Dynamische Rout-Planung**

```
pathfinding.executeProceduralMovement(npcPlayer, targetLocation);
```

Features:

- Nutzt Hytale Terrain-Daten
- Fügt "Umwege" ein, wenn Agent "abgelenkt" ist (hoher Stress-Level)
- Verhält sich wie Minecraft AI, nicht wie optimaler Algorithmus

---

Teil 3: Syntheseebene – Wie Theorie in Code wird

3.1 "Tells" als implementierbare Mechaniken

Gemini-Konzeption:

> "Visueller Glitch: Wenn ein Architekt lügt, generiert das Plugin mit einer 5%-Wahrscheinlichkeit einen kurzen visuellen Glitch um den Avatar."

BipedalAgent-Implementierung:

```
private void executeLLMAction(String llmDecision) {
String actionType = llmDecision.split(":::")[0];
```

```
if (actionType.equals("CHAT")) {
    String message = llmDecision.split(":::")[1];

    // Prüfe, ob Aussage zu Memory widerspricht
    if (memory.contradictsPreviousStatement(message)) {
        // 5% Chance: Visual Glitch
        if (Math.random() < 0.05) {
```

```
            spawnVisualGlitch(npcPlayer);  // Textur-Flicker
        }
    }

    npcPlayer.chat(message);
    maskingOrchestrator.replaceFillerWithRealMessage(message);
}
```

}

Was passiert:

1. Agent sagt: "Ich war nicht in den Vents"
2. Memory weiß aber: Agent war 2 Minuten vorher in den Vents (aufgezeichnet)
3. Mit 5% Chance: Visueller Glitch um Avatar
4. **Gameplay-Folge**: Aufmerksame Spieler sehen den Glitch → "Aha! Der Agent lügt!"

3.2 Assoziations-Netzwerk Task (ARC Puzzle)

Gemini-Konzeption:

> "Task-Typ A: Ein Spieler erhält zwei Wörter (z.B. 'Banane' und 'Republik') und muss ein verbindendes Wort eingeben."

BipedalAgent-Implementierung:

Für Menschen:

- Input: ["Banane", "Republik"]
- Erwartete Assoziation: "Staat" oder "Peel" (Kunstwort)
- LLM bewertet: Ist das kreativ/menschlich?

Für Architekten (Impostor):

- Erhalten FALSCHE Wörter (z.B. ["Banane", "Elefant"])
- Oder müssen raten, was LLM als "menschlich" einstuft
- **Tell**: Ihre Assoziationen sind zu "statistisch" (Word2Vec-ähnlich)

Code-Flow:
public void onPlayerSpeak(String speakerId, String message) {
memory.recordEvent("Player " + speakerId + " said: " + message);

```
if (shouldRespond(speakerId, message)) {
    // LLM bewertet, ob Response menschlich wirkt
    String llmEval = llmService.evaluateCreativity(message, context);

    if (llmEval.contains("TOO_STATISTICAL")) {
        // Agent war zu predictable → flagged als Impostor
```

```
            GameStateManager.getInstance().updateSuspicionScore(agentId, +2);
        }
    }
}
```

3.3 Die Voting Phase: Emergente Narratives

Szenario: 8 Spieler, 2 Architekten (BipedalAgent)

Runde 1-5 (Day Phase):

- Spieler arbeiten an Tasks (ARC Puzzles, Motor Glitch)
- BipedalAgents beobachten, sprechen, generieren Verdacht via LLM
- FederatedMemory speichert alle Aussagen

Runde 6 (Emergency Meeting):
public void onPhaseChange(GamePhase newPhase) {
if (newPhase == GamePhase.VOTING) {
// Agent ruft komplette Memory-Context ab
String context = memory.getMemoryContext();

```
    // LLM generiert überzeugendste Anklage basierend auf Memory
    String accusationPrompt =
        "Based on these facts: " + context +
        ", who should we vote out and why?";

    String llmDecision = llmService.inference(accusationPrompt, 500);
    npcPlayer.chat(llmDecision);
}
```

}

Ergebnis: Agent "argumentiert" basierend auf 20+ Minuten Spielhistorie → wirkt konsistent und überzeugend.

Teil 4: Implementierungs-Roadmap

Phase 1 (Woche 1-2): BipedalAgent Core ✅

Deliverables:

- ✅ BipedalAgent.java (Kern-Klasse)
- ✅ FederatedMemoryRepository.java (Dual-Stack Memory)
- ✅ DualProcessDecisionEngine.java (System 1/2)
- ✅ LatencyMaskingOrchestrator.java (Tier 1-3 Masking)

Status: Ready for immediate Java development

Phase 2 (Woche 3-4): Persona & Prompt Engineering

Tasks:

- PersonaProfile: Archetypen-System ("Mafia", "Doctor", "Jester", "Doppelgänger")
- PersonaPromptBuilder: Dynamic prompt generation basierend auf Role + Memory + Current Phase
- Archetype-spezifische Fähigkeiten: Jede Archetype hat andere Decision-Trees

Beispiel-Persona:
PersonaProfile detective = new PersonaProfile(
archetype = Archetype.DETECTIVE,
faction = Faction.TOWN,
systemPrompt = "Du bist ein Detektiv. Dein Ziel ist, die Mafia zu finden...",
traits = ["analytical", "suspicious", "detail-oriented"]
);

Phase 3 (Woche 5-6): Physical Mimesis Engine

Components:

- ProceduralPathfinding: Realistische Wege mit Umwegen
- MouseLookController: Saccadische Augenbewegung
- IdleAnimationController: Prozedurales Fidgeting
- PhysicalMimesisEngine: Zentrale Koordination

Test: Agent sollte sich "menschlich" bewegen (nicht optimal, aber auch nicht zufällig).

Phase 4 (Woche 7-8): Aletheia Tasks

Tasks zu implementieren:

- **ARC Puzzle**: Integration externer Logik-Puzzle API
- **Motor Glitch**: Mouse-Movement-Analyse
- **Memory Puzzle**: Memory-Recall-basierte Tasks
- **Dialog Puzzle**: Turing-Test-ähnliche Konversationen

Phase 5 (Woche 9-10): Hytale Integration

Hytale API-Bindings:

- PlayerMoveEvent → BipedalAgent.updatePerception()
- ChatEvent → BipedalAgent.onPlayerSpeak()
- PhaseChangeEvent → BipedalAgent.onPhaseChange()
- DeathEvent → BipedalAgent.onPlayerDeath()

Hytale-spezifisch:

- Blöcke: Task-Terminals, Analysis-Stationen
- Entities: NPCs als visuelle Ankerpunkte
- Sounds: Bit-Crushing-Effekt für Stimmen

Phase 6 (Woche 11-12): Testing & Polish

Testing:

- Unit Tests: Memory-Konsistenz, Decision-Latency
- Integration Tests: Full Game-Loop mit 10 Agenten + 10 Spielern
- User Testing: Bot-Detection-Rate messen
- Balancing: Difficulty-Slider für Impostor-Agenten

Teil 5: Erweiterte Konzepte – Die anderen 9 Mods

5.1 Chrono-Echoes: 4D-Gameplay

Mechanik: Spieler spielen gegen ihre eigenen Aufzeichnungen aus der Vorrunde.

BipedalAgent-Anwendung:
Runde 1: Mensch-Spieler A läuft Parkour
↓
PlayerMoveEvent → Java Plugin speichert alle Positionen
↓
Runde 2: BipedalAgent B spielt gegen Echo von A
↓
LLM nutzt Echo-Bewegungen zur Vorhersage
→ "Spieler A wird wahrscheinlich bei (100, 64, 200) sein"
→ Positioniert sich defensiv

Integration mit BipedalAgent:
public void onRound2Start() {
List<EchoPosition> previousRound = getEchoPositions();

```
// LLM analysiert Muster
String predictionPrompt =
    "Based on these past player movements: " + previousRound +
    ", where will they move next?";

Vector predictedLocation = parsePredictionAsVector(
    llmService.inference(predictionPrompt)
);

physicalMimesis.repositionToIntercept(predictedLocation);
```

}

5.2 Soulbound: Körper-Possession

Mechanik: Spieler sind "Geister", die NPCs/Monster besitzen können.

BipedalAgent-Transformation:

Statt: "BipedalAgent steuert Player X"
Neu: "BipedalAgent steuert Monster Y (Körper) mit Memory von Player Z"

// Agent verlässt seinen Körper
npcPlayer.getComponent(RenderComponent).setVisible(false);
npcPlayer.getComponent(PhysicsComponent).disable();

// Agent übernimmt Monster-Body
Zombie monster = world.spawn(Zombie.class, location);
agent.attachToEntity(monster);

// Monster wird jetzt vom Agent gesteuert
// Aber hat auch KI-Verhalten, wenn Agent nicht "konzentriert" ist
agent.decisionEngine.setControlled(true); // Full control
monster.setAI(true); // Secondary AI fallback

5.3 Entropy Market: Wirtschaftliche Deduktion

Mechanik: Spieler handeln mit Items. Insider (Architekten) wissen, wann Preise fallen.

BipedalAgent-Wirtschaft:

public void onMarketEvent(MarketEvent event) {
// Agent hat Geheimwissen (über Memory)
String insider = memory.getHighestConfidenceSecret();

```
if (insider != null) {
    // Agent tradet "verdächtig" – zu perfekt vorausgesagt
    // Andere Spieler können dies erkennen

    String tradeDecision = llmService.inference(
        "You know: " + insider +
        ". How do you trade on this information?"
    );

    // Trade wird ausgeführt (sichtbar für alle)
    executeMarketTrade(tradeDecision);

    // Tell: Timing der Trade ist verdächtig genau
}
```

```
}
```

Teil 6: Technische Architektur Deep-Dive

6.1 Event-Driven Architecture

```
[Hytale Server Event Bus]
↓
[PlayerMoveEvent] → BipedalAgent.updatePerception()
[ChatEvent] → BipedalAgent.onPlayerSpeak()
[PhaseChangeEvent] → BipedalAgent.onPhaseChange()
[DeathEvent] → BipedalAgent.onPlayerDeath()
↓
[Memory Update] → FederatedMemoryRepository.recordEvent()
↓
[Decision Request] → DualProcessDecisionEngine
├─ System 1 (Instant) → BehaviorTree
└─ System 2 (Async) → LLM Service
↓
[Execution] → executeActions()
├─ Chat
├─ Movement
├─ Look Direction
└─ Animation
↓
[Masking] → LatencyMaskingOrchestrator
```

6.2 Memory Lifecycle

```
Event Capture (50ms cycle)
↓
Short-Term Storage (LinkedList, <10 items)
↓
Embedding + Vector DB Async Insert
↓
[During LLM Request]
├─ Retrieve Short-Term (last 10)
├─ Retrieve Vector DB (top-3 relevant)
├─ Combine → Memory Context
├─ Build Prompt
├─ Send to LLM
└─ Get Response
↓
Response Execution + Recording
```

6.3 Parallel Processing

Main Thread (50ms tick):

- Perception (read-only)
- System 1 Decision (fast)
- Execution
- Masking Orchestration

Async Thread Pool:

- Embedding Calculation (CPU-bound)
- Vector DB Query (I/O-bound)
- LLM Inference (I/O-bound)
- MongoDB Writes (I/O-bound)

Synchronization: CompletableFuture-basiert, keine Race Conditions.

Fazit: Brückenschlag Theorie ↔ Implementierung

Theoretisches Konzept (Gemini)	Implementierung (BipedalAgent)	Gameplay-Effekt
Reverse-Turing-Test	ARC Puzzles + Dual-Process Decision	Spieler müssen KI von Mensch unterscheiden
Tells (visuell)	Visual Glitch bei Lügen	Aufmerksame Spieler bekommen Hinweise
Latency-Problem	LatencyMaskingOrchestrator (Tier 1-3)	LLM-Verzögerung wirkt natürlich
Memory-Konsistenz	FederatedMemoryRepository	Agent widerspricht sich nicht
Realistische Bewegung	Physical Mimesis Engine	Nicht-optimale Pathfinding
Strategische Tiefe	LLM-basierte Argumentation	Komplexe Verdächtigungen möglich

Die BipedalAgent-Klasse ist nicht nur ein Container für Komponenten – sie ist die konkrete Realisierung der theoretischen Architektur aus der Gemini-Konzeption.

Die nächste Phase beginnt mit Phase 1 Implementation (Woche 1-2), sobald die Grundmechaniken validiert sind.

Anhang A: Code-Schnipsel

A.1 Memory Context Builder

```
public String getMemoryContext() {
StringBuilder context = new StringBuilder();

  // Tier 1: Jüngste Ereignisse (Short-Term)
  context.append("=== RECENT EVENTS ===\n");
  for (int i = Math.max(0, shortTermMemory.size() - 10);
       i < shortTermMemory.size(); i++) {
      Event e = shortTermMemory.get(i);
      context.append(String.format("[%dms ago] %s\n",
          System.currentTimeMillis() - e.timestamp,
          e.description));
  }

  // Tier 2: Relevante Langzeit-Erinnerungen (Vector DB)
  context.append("\n=== RELEVANT BACKGROUND ===\n");
  String currentContext = getCurrentGameContext();
  List<String> relevant = vectorDb.retrieve(currentContext, k=3);
  for (String mem : relevant) {
      context.append("- ").append(mem).append("\n");
  }

  // Tier 3: Persönliche Einschätzung
  context.append("\n=== MY SUSPICIONS ===\n");
  Map<String, Double> suspicion = memory.getSuspicionScores();
  for (Map.Entry<String, Double> entry : suspicion.entrySet()) {
      context.append(String.format("- %s: %.2f%% suspicious\n",
          entry.getKey(), entry.getValue() * 100));
  }

  return context.toString();
}
```

A.2 Dual-Process Decision

```
private void updateDecision() {
if (!shouldMakeDecision()) return;

  if (isUrgentSituation()) {
     // System 1: Instant Reflex
     BehaviorTreeResult action = decisionEngine.evaluateBehaviorTree();

     long startTime = System.currentTimeMillis();
     executeReflexAction(action);
     long elapsed = System.currentTimeMillis() - startTime;

     System.out.println("[SYSTEM 1] Decision in " + elapsed + "ms");
     // Output: Decision in 3ms

  } else {
     // System 2: Strategic Thinking
     if (pendingLLMDecision == null || pendingLLMDecision.isDone()) {
        long startTime = System.currentTimeMillis();
        requestLLMDecision();

        System.out.println("[SYSTEM 2] LLM inference started...");
        // (Will complete in ~500-2000ms, but masked)
     }
  }

}
```

A.3 Tell Detection

```
private void executeLLMAction(String llmDecision) {
String[] parts = llmDecision.split(":::");
String actionType = parts[0];
String actionData = parts.length > 1 ? parts[1] : "";

  if (actionType.equals("CHAT")) {
     // Prüfe auf Widerspruch
     if (memory.contradictsPreviousStatement(actionData)) {
        // Mit 5% Chance: Visual Glitch
        if (Math.random() < 0.05) {
```

```
            // Textur flackert – Tell für Spieler
            spawnVisualGlitch(npcPlayer, durationMs = 200);

            // Logging für Debugging
            System.out.println("[TELL] Visual glitch triggered due to contradiction");
        }
    }

    // Mit 10% Chance: Timing-Tell (zu schnelle Antwort)
    long responseTime = System.currentTimeMillis() - lastQuestionTime;
    if (responseTime < 100 && Math.random() < 0.10) {
        System.out.println("[TELL] Response too fast (" + responseTime + "ms)");
        // Tell ist integriert, Spieler bemerken es visuell
    }

    npcPlayer.chat(actionData);
    maskingOrchestrator.replaceFillerWithRealMessage(actionData);
  }

}
```

Version: 1.0
Status: Dokumentation fertiggestellt
Nächster Schritt: Phase 1 Implementation (01.02.2026)