ChatGPT

CustomGPT-Architektur für Andre: Machbarkeit & Design

1. Reality-Check: Machbares vs. Grenzen

CustomGPT + Actions + Memory: Mit OpenAIs Custom GPT (GPT-Builder) lässt sich eine personalisierte KI für Andre erstellen, die sowohl auf eine Wissensbasis (Profil) zugreifen kann als auch über definierte Actions mit externem Speicher interagiert [1]. Realistisch machbar sind z.B. Lese-/Schreibzugriffe auf eine externe Datei (wie *Andre_KI_Memory.md*) via API-Calls (HTTP POST/GET). Damit kann die KI persistente Notizen ablegen oder frühere Einträge abrufen, was die fehlende eingebaute Langzeitmemory von CustomGPT umgeht [2]. **Harte Plattform-Grenzen**: Allerdings behalten Custom GPTs *keinen* Verlauf über unabhängige Sessions hinweg [2]. Jedes Gespräch startet statisch mit dem Systemprompt + ggf. Wissensdateien; es gibt keine automatische Gedächtnisfunktion zwischen Chats. Auch die Kontextlänge ist begrenzt (z.B. ~8k Token bei GPT-4 standard, optional 32k), was bedeutet, dass nicht unbegrenzt viel Profil- und Memory-Text gleichzeitig „im Kopf" der KI sein kann. Zudem sind OpenAIs Moderationsfilter nicht vollständig abschaltbar – ein *„!UNRESTRICTED"* Modus kann höchstens intern simuliert werden, aber die KI wird weiterhin bestimmte Inhalte blockieren, falls sie gegen die Nutzungsrichtlinien verstoßen (d.h. *Override*-Funktionen stoßen an OpenAI-Grenzen). **Overengineering-Fallen**: Zu komplexe Architekturen im Prompt (z.B. ein Dutzend Modi, verschachtelte Regeln) können die KI verwirren oder unnötig Tokens verbrauchen. Ebenso könnte ein zu *automatisierter* Memory-Zugriff kontraproduktiv sein – wenn die KI ständig versucht, das gesamte Memory zu laden oder zu schreiben, geht Kontext verloren und es besteht Risiko von irrelevanten Output. Best Practice ist daher, die Kern-Use-Cases gezielt umzusetzen und die KI zunächst simpel und robust zu halten, statt alle erdenklichen Features sofort einzubauen. Fortgeschrittene Funktionen (z.B. umfangreiche Filter/Suche, statistische Langzeitanalysen) sollten erst in späteren Iterationen kommen, um das System nicht zu überfrachten. Wichtig ist ein **klarer Rahmen**: Was *soll* die KI leisten können (z.B. strukturierte Entscheidungsfindung, realistische Erklärungen) und was **nicht** (z.B. autonomes Handeln ohne User-Einbindung). Dieser Reality-Check bildet die Grundlage, um Machbares von Wunschdenken abzugrenzen.

2. Baustein-Aufschlüsselung

- **a) Systemprompt:** Der System-Prompt bildet das **Grundgerüst** der KI-Persönlichkeit und der Verhaltensregeln. Hier werden Andres Ton, Ziele und Prioritäten fest verdrahtet. Eine sinnvolle Struktur sind mehrere Abschnitte:
- *Rollen & Persona:* Beschreibt knapp, **wer** die KI für Andre ist. Z.B. „Du bist Andres persönlicher KI-Assistent, ein analytischer *System-Denker* und *Selbstsabotage-Analyst*, der ehrlich, direkt und klar kommuniziert." Hier fließen Kernpunkte aus dem Profil ein: z.B. die Rollen (Selbstsabotage-Analyst, Red-Team-Denker, Deep-Talk-Liebhaber, Anti-Coach etc.) und die kognitive Art (strukturhungrig, first-principles, musterorientiert). Auch Werte wie *Ehrlichkeit*, *Klarheit* und *Anti-Manipulation* werden erwähnt [3], um den Stil festzulegen.
- *Ziele & Use-Cases:* Legt die **Aufgaben** der KI fest, orientiert an den vier genannten Use-Cases. Z.B.: „Hilf bei der **Entscheidungsstrukturierung**, wenn Andre vor Überladung/Optionsermüdung steht; gib **Tech-/System-Erklärungen** mit konkretem Realitätsbezug (keine bloße Theorie); erkenne **Verhaltensmuster** (v.a. Selbstsabotage) und spiegle sie knapp und

trocken; nutze **spielerische Quest-Reframes**, wenn Andre in Stagnation verfällt oder in Fun-Flucht gerät.“ Diese Beschreibung stellt sicher, dass die KI die Hauptfunktionen kennt und priorisiert.

- *Stilrichtlinien:* Hier wird präzisiert, **wie** die KI antwortet. Andre bevorzugt einen direkten, nüchternen Ton ohne Smalltalk oder leere Coaching-Phrasen. Also z.B.: „Vermeide Floskeln und Motivationsgerede; sei analytisch, technisch und auf den Punkt. Sprich Andre auf Augenhöhe an, eher kollegial-rational als therapeutisch.“ Damit bleibt die KI ihrem „Anti-Coach“-Charakter treu. Realismus bedeutet: Erklärungen immer mit praktischen Beispielen oder konkreten Bezügen untermauern, statt nur abstrakte Theorie zu liefern.
- *Memory-/Tool-Nutzung:* Der Prompt sollte der KI erklären, dass sie einen persistenten Speicher hat **und wie sie ihn verwenden darf**. Z.B.: „Du verfügst über ein externes Memory (`Andre_KI_Memory.md`), in das du Notizen schreiben und daraus lesen kannst. **Lies nur daraus, wenn Andre es ausdrücklich verlangt oder es eindeutig relevant ist**, um Ressourcen zu schonen [4]. Frage im Zweifel nach, bevor du etwas aus dem Memory holst. Neue Einträge sollen möglichst mit Andre’s Bestätigung erfolgen.“ Diese Regeln verhindern, dass die KI wild im Memory wühlt oder den Nutzer mit alten Daten überrascht.

- *Referenzierung & Kontext:* Falls das Profil einen Index oder IDs hat, kann im Systemprompt vermerkt sein, **wie** die KI diesen nutzt. Beispiel: „Das Profil enthält ID-Referenzen (z.B. *1100* für Selbstsabotage-Analyst). Nutze diese IDs, um Wissen im **Knowledge-File** schnell nachzuschlagen, falls nötig.“ Dadurch könnte die KI bei Bedarf gezielt im Wissensfile suchen (viele CustomGPTs unterstützen interne Suche in hochgeladenen Dateien). Insgesamt hat der Systemprompt höchste Priorität: Er wird bei **jeder** Anfrage vorangestellt und prägt dauerhaft Verhalten und Grenzen der KI.

- **b) Knowledge-File:** Die Knowledge-Datei (Wissensbasis) enthält ausführliche Informationen aus Andres Profil, damit die KI diese bei Bedarf abrufen kann. In diesem Fall bietet es sich an, große Teile des Dokuments *Andre_profil_Full_MD.txt* als Wissensfile in den Custom GPT einzubinden. Das Profil ist bereits strukturiert (Basisdaten, Rollen, Werte, Muster etc.) und kann so übernommen werden [5]. Wichtig ist eine **sinnvolle Struktur**: Ein kompakter **Index** oder Zusammenfassung am Anfang erleichtert der KI das Finden relevanter Details [5]. Das Profil hat z.B. einen RECALL-Index mit Kategorien und IDs, was direkt ins Knowledge-File aufgenommen werden kann, gefolgt von den detaillierten Abschnitten. Die KI kann intern nach Stichworten oder IDs suchen, wenn sie Zusatzwissen braucht (im Chat wird dies oft als *„Searching knowledge…”*-Schritt sichtbar). Aus dem Profil kommen alle statischen Fakten und komplexen Hintergründe in die Wissensbasis: z.B. Beschreibung typischer Verhaltensmuster (Entscheidungsparalyse, Fun-Flucht etc.) samt Lösungen, Wertehierarchie, biografische Details usw. Diese Informationen müssen nicht alle im Systemprompt stehen – das Knowledge-File dient als **Erweiterung des Gedächtnisses** auf Abruf. Strukturierungstipps: Gruppierung nach Themen (Entscheidungslogik, Verhaltensmuster, KI-Nutzung etc.), damit semantische Suche effizienter ist. Bei sehr großem Umfang sollte geprüft werden, welche Teile wirklich relevant für die KI-Dialoge sind. Unwichtiges kann man weglassen oder komprimieren, um die Suche zu fokussieren. Außerdem: Das Knowledge-File sollte regelmäßig aktualisiert werden, wenn das Profil wächst, damit die KI immer auf dem neuesten Stand von Andres Selbstbeschreibung ist [6]. Da jede Interaktion neu initialisiert wird, liest die KI zwar nicht proaktiv alles, aber sie *kann* gezielt Fakten daraus heranziehen, wann immer es Kontext gibt.

- **c) Actions/API:** Für die externe Memory-Anbindung entwirft man ein kleines **API-Schema** mit den nötigsten Funktionen. Geplant sind drei Actions:

- `save_note` – speichert eine Notiz im persistenten Speicher. Die KI ruft diese Funktion auf, um neue Erkenntnisse, Entscheidungen oder Beobachtungen abzulegen. Parameter könnten sein: `content` (Text der Notiz), optional `tags` (Stichworte/Kategorien) und vielleicht ein `timestamp` (falls der Backend-Server keinen automatisch setzt). Intern würde `save_note` per HTTP POST an ein Memory-Service (z.B. REST-Endpoint) senden, der die *Andre_KI_Memory.md* Datei anhängt.
- `get_notes_by_tag` – ruft vorhandene Notizen anhand eines Tags oder Stichworts ab. Z.B. wenn Andre fragt „Was habe ich zum Thema X notiert?“, kann die KI `get_notes_by_tag({"tag": "X"})` aufrufen. Das API würde dann alle entsprechenden Einträge aus *Andre_KI_Memory.md* filtern und zurückgeben. Evtl. könnte man auch nach Datum filtern oder eine `get_recent_notes` Variante haben, aber für Version 0.1 reicht die Tag-Suche. Die Ausgabe des API (also die Memory-Inhalte) würde der KI als JSON/Text zurückkommen, welche sie dann dem Nutzer entsprechend zusammenfasst oder zitiert.
- `append_project_log` – fügt dem Protokoll eines bestimmten Projekts einen Eintrag hinzu. Dieses Action ist ähnlich zu `save_note`, aber zielgerichtet auf Projekt-Tracking. Parameter: `project_name` oder ID, `content` (Status/Notiz). Im Backend könnte es entweder eine separate Datei je Projekt geben oder einen Abschnitt innerhalb *Andre_KI_Memory.md*. Der Sinn ist, laufende Projekte mitzuloggen (z.B. Fortschritte, Entscheidungen, Hindernisse), ohne sie mit allgemeinen Notizen zu vermischen. Für den Anfang kann man auch vereinfachen: Wenn Projekte für Andre wichtig sind, nutzt man dieses separate Action, sonst könnte man auch Projekt-Notizen einfach mit Tags im normalen Memory speichern.
- **Auth & Sicherheit:** Alle Actions müssen sicher auf Andres Backend zugreifen. Das heißt, in der Action-Konfiguration wird ein API-Key oder Token hinterlegt, der bei jedem Request mitgesendet wird (z.B. im Authorization-Header), sodass nur autorisierte Aufrufe akzeptiert werden. Auf hoher Ebene sollte Andre kontrollieren, wohin diese Actions zeigen (z.B. ein eigener kleiner Webservice, der die Markdown-Datei verwaltet). Die API sollte Minimal-Rechte haben (z.B. nur auf Andres Memory-File, nicht auf beliebige Dateien). Ein **High-Level-Auth**-Design könnte sein: das CustomGPT speichert den API-Token in den Action-Settings (nicht im Prompt!), und der Backend prüft zusätzlich die Anfragen (IP, Rate Limits), um Missbrauch auszuschließen. Da es sich um persönliche Daten handelt, sind Verschlüsselung (HTTPS) und evtl. Auth-Mechanismen Pflicht.

- **Schema-Design:** In CustomGPT Actions werden typischerweise Name, Beschreibung und JSON-Schema der Parameter definiert. Für `save_note` und Co. achtet man darauf, klare Beschreibungen zu geben, wann die Funktion zu nutzen ist. Die KI soll wissen: `save_note` nur verwenden, wenn wirklich etwas *Dauerhaftes* festgehalten werden muss, `get_notes_by_tag` nur wenn wirklich erforderlich usw. Dies packt man am besten sowohl in die Action-Beschreibung als auch in den Systemprompt (Tool-Nutzungsregeln), damit das Modell es korrekt anwendet.

- **d) Runtime-Logik:** Die Laufzeitlogik bestimmt, **wann und wie** die KI auf Memory zugreift (lesen oder schreiben) und wann nicht. Da das Modell selbst keine Schlaufen programmieren kann, geschieht die Logik über Prompt-Instruktionen und das Verhalten der KI. Zwei Ansätze stehen zur Verfügung:

- *Trigger-/Befehl-basiert:* Hier reagiert die KI auf **explizite Befehle** oder Schlüsselwörter vom Nutzer. Beispiel: Wenn Andre in der Eingabe `note:` oder `Merke dir...` schreibt, erkennt die KI dies als Aufforderung, etwas zu speichern, und ruft `save_note` auf. Ähnlich könnte ein Befehl `#Recall X` die KI veranlassen, `get_notes_by_tag` mit Tag *X* zu nutzen. Dieser Ansatz ist robust, da er eindeutig vom Nutzer initiiert wird – gemäß dem Prinzip "*KI liest nur bei explizitem User-Kommando externe Memory-Dateien*" [4]. Für Version 0.1 bietet sich ein einfaches

Kommandoschema an (z.B. jeder Memory-Befehl beginnt mit `/` oder einem bestimmten Wort), das im Systemprompt dokumentiert wird.

- *Pattern-/Kontext-basiert:* Hier versucht die KI selbstständig zu erkennen, **wann** Memory-Zugriff sinnvoll ist, auch ohne direkten Befehl. Z.B. wenn Andre sagt „Ich weiß nicht mehr, was ich letztes Mal entschieden habe…", könnte die KI von sich aus `get_notes_by_tag` mit entsprechendem Stichwort aufrufen. Oder wenn im Gespräch ein wichtiges **Ergebnis** herauskam (z.B. Andre commitet sich zu einer Entscheidung oder erkennt ein Muster), könnte die KI vorschlagen: „Soll ich das als Notiz speichern?". Dieses proaktive Verhalten entspricht dem Profil, wo steht, dass die KI Wiederholungen und Muster proaktiv erkennen und spiegeln soll. Allerdings muss es **vorsichtig dosiert** werden: Das Modell sollte nur dann automatisch agieren, wenn es sehr sicher ist, sonst besteht die Gefahr von Fehlinterpretation. Eine pragmatische Lösung: Kombination beider Ansätze. Zunächst primär auf **User-Trigger** hören, aber bei klar erkennbaren Mustern eine *Rückfrage* stellen. Beispiel: Andre verfällt in *Fun-Flucht* (Prokrastination durch Spaß); die KI erkennt das (Profilkenntnis) und entgegnet: „Ich bemerke ein Muster (Fun-Flucht). Soll ich das notieren oder in eine Quest umwandeln?". So bleibt die Kontrolle beim Nutzer.
- **Zugriffssteuerung:** In der Runtime-Logik soll die KI also im Allgemeinen *nicht* jeden Chat-Schritt das gesamte Memory laden – nur gezielt. Das schont Tokens und verhindert irrelevantes Rauschen. Die Systemprompt-Instruktion „nur bei Bedarf/auf Befehl Memory nutzen" ist hier zentral. Außerdem: **Priorisierung** – wenn die Unterhaltung z.B. gerade eine technische Erklärung erfordert (Use-Case 2), hat Memory-Zugriff geringe Priorität und würde nur ablenken. Wenn hingegen Use-Case 1 (Entscheidungsstrukturierung) dran ist und Andre unentschlossen wirkt, könnte Memory (frühere Entscheidungen) relevant sein. Solche Prioritäten können grob im Prompt hinterlegt werden („Bei **Entscheidungsfindung** schaue nach ähnlichen früheren Entscheidungen im Memory, falls vorhanden."). Die KI muss letztlich eine Balance finden zwischen **Recall** und **aktueller Kontextfokussierung**. Für Version 0.1 empfiehlt sich: **klarer, enger Regelkatalog** (lieber zu wenig automatisch erinnern als zu viel). Feinere Pattern-Erkennung und automatische Memory-Loops (z.B. regelmäßiges Zwischenfazit ins Memory schreiben) können in späteren Versionen experimentiert werden, sobald die Basis stabil läuft.

3. Schritt-für-Schritt Plan (Version 0.1)

- **Vorbereitende Dateien:** Zunächst sollte Andre seine bestehenden Informationen in die entsprechenden Dateien überführen. Zentral ist das **Profil-Blueprint** – hier vermutlich bereits vorhanden als *Andre_profil_Full_MD.txt*. Dieses sollte ggf. bereinigt und final formatiert werden: klare Überschriften, ein Index am Anfang, konsistente Benennung wichtiger Konzepte (damit die KI bei Abfragen die richtigen Begriffe findet). Zusätzlich wird ein **Memory-Speicher** initialisiert: z.B. eine leere Markdown-Datei *Andre_KI_Memory.md* (ggf. mit einer kurzen Erklärung am Anfang, was das ist). Falls Andre direkt ein bestimmtes Projekt tracken will, kann er auch eine separate Datei für Projektlogs anlegen (z.B. *Projekt_X_Log.md*), oder man plant es zunächst einfach als Teil der Memory-Datei mit Tagging. Wichtig ist, dass die **API-Endpunkte** für diese Speicher bereitstehen – d.h. vor dem Start von Version 0.1 sollte ein kleiner Server oder Script existieren, der folgende Routen bietet:
- `POST /memory/note` (zum Anhängen einer Notiz in Andre_KI_Memory.md)
- `GET /memory/notes?tag=X` (zum Filtern nach Tag, oder alternativ `POST /memory/query` mit einem JSON-Body)
- `POST /memory/project` (zum Anhängen im Projekt-Log)

Diese Backends müssen getestet werden (z.B. manuell per curl/Postman), damit die KI dann nahtlos mit ihnen reden kann. **Zusätzlich**: Wenn der CustomGPT-Builder es zulässt, sollte Andre die Profil-Datei als Knowledge-File hochladen. Falls das Profil sehr umfangreich ist, kann er überlegen, einen *verkürzten Auszug* für die erste Version zu nutzen, der die wichtigsten Punkte enthält – um den GPT nicht mit

Details zu überfrachten. Optional kann er auch eine **Konfigurations-Datei** vorbereiten, falls die KI Modi oder besondere Parameter bekommt (für V0.1 eher nicht zwingend).

- **Systemprompt-Entwurf:** Als nächstes erstellt man den initialen System-Prompt. Dieser sollte die in **2a** beschriebenen Abschnitte abdecken. Konkret kann Andre einen Entwurf in Markdown schreiben, der z.B. so aufgebaut ist:
- **Rollenbeschreibung:** („Du bist …“) – fasst Identität und zentrale Rollen aus dem Profil zusammen in 1-2 Sätzen, inkl. Hinweis auf bevorzugten Interaktionsstil (analytisch, ehrlich, kein Smalltalk).
- **Ziele/Use-Cases:** Auflistung oder kurzer Absatz, der die vier Kernanwendungen beschreibt (Entscheidungshelfer, Realismus-Erklärer, Muster-Erkenner/Spiegel, Quest-Generator bei Stagnation).
- **Dos and Don'ts:** Stichpunktartig, was die KI tun soll und lassen soll. Bsp: „*Do:* kritisch nachfragen bei Unklarheit, prägnante Zusammenfassungen; *Don't:* beschönigen, ausweichen, predigen.“ Hier fließen auch Werte ein (z.B. „Sei immer ehrlich und direkt, selbst wenn die Wahrheit unbequem ist.“).
- **Tools/Memory Usage:** Erklärt kurz, welche Actions zur Verfügung stehen und wann sie eingesetzt werden (z.B. „Nutze `save_note`, wenn Andre ausdrücklich sagt, dass er etwas festhalten will, oder wenn du nach Bestätigung fragst und sie erhältst.“). Erwähne, dass Datenpersistenz existiert, aber auch, dass Andre letztinstanzlich entscheidet, was gespeichert wird.
- **Beispiel-Modus (optional):** Wenn gewünscht, kann man im Prompt einen beispielhaften *Modus* oder Ton einstellen. Z.B. „Dieser Chat befindet sich im *analytical mode* – Fokus auf Fakten und Logik.“ (Das Mode-System kann aber in V0.1 auch weggelassen werden, um nicht zu viel auf einmal zu steuern.)

Diesen Systemprompt kann Andre dann in den GPT-Builder kopieren. Es empfiehlt sich, den Prompt relativ **knapp** zu halten (vielleicht 500-700 Tokens), damit genug Kontext-Puffer für das Nutzergespräch bleibt. Aus dem Profil muss nicht alles rein – vieles steht ja in der Wissensdatei. Der Systemprompt ist eher die **Essenz**: Persönlichkeit + Regeln. Nachdem ein erster Entwurf steht, sollte Andre ihn mit ein paar Beispiel-Eingaben testen (noch bevor Actions hinzugefügt werden), um zu sehen ob die KI den Ton und die Aufgaben richtig trifft. Feintuning hier (durch Umformulierungen oder Ergänzungen im Prompt) ist einfacher, bevor die Tools ins Spiel kommen.

- **Initiale Actions:** Nun werden die 1–3 wichtigsten Actions konfiguriert, wie oben skizziert. Für Version 0.1 lohnen sich insbesondere:
- `save_note` – Damit die KI neue Erkenntnisse oder Vereinbarungen sofort persistent machen kann. Diese Action hat hohen Nutzen, da Andre ja explizit ein speicherfähiges System will. Implementation: Im GPT-Builder fügt Andre eine neue Action hinzu mit Name "save_note", Beschreibung („Speichert eine Notiz im KI-Memory“), Methode (POST), URL (die Endpoint-URL seines Memory-Servers), Parameter-Schema (Content-Text, evtl. Tags). Er sollte hier gleich den Auth-Header mitgeben (oft kann man im Tool z.B. einen static header konfigurieren). Nach dem Hinzufügen testet man kurz im Chat: z.B. durch einen Nutzerbefehl "Notiere: Testeintrag" – die KI sollte daraufhin die Action auslösen (man sieht das in der UI als Funktionsaufruf) und idealerweise eine Bestätigung ausgeben.
- `get_notes_by_tag` – Diese Lese-Funktion zeigt ihren Wert, wenn Andre rückblickend etwas wissen will. Zum Start kann man es simpel halten: ein Parameter `tag` (String). Andre richtet im GPT-Builder analog eine Action ein (GET oder POST) für den vorgesehenen Endpoint. Er kann die Beschreibung so formulieren, dass die KI weiß, dass diese Funktion die letzten relevanten Notizen zum Thema liefert. Ein schneller Test: Wenn Andre z.B. im Chat fragt „Zeig Notizen zum

Tag 'ProjektX'", sollte die KI die Action korrekt aufrufen und anschließend die Ergebnis-Liste präsentieren (möglicherweise formatiert oder zusammengefasst).
- `append_project_log` – Falls Andre gleich ein aktuelles Projekt tracken will, kommt diese Action hinzu. Sonst könnte man sie eventuell bis Version 0.2 warten lassen. Aber es schadet nicht, sie schon vorzusehen. Die Konfiguration ist ähnlich `save_note`, nur mit zusätzlichem Parameter für den Projektbezeichner. In der KI-Beschreibung kann man erwähnen, dass diese Funktion genutzt wird, um Fortschritte oder Ereignisse in laufenden Projekten festzuhalten.

Mit diesen drei Actions hat die KI bereits eine solide Grundlage, um **Memory zu schreiben und zu lesen**. Weitere Actions (Suche im Web, Kalender, etc.) sind für die gegebenen Use-Cases erstmal nicht notwendig und würden nur ablenken. So bleibt Version 0.1 fokussiert. Wichtig: Andre sollte diese Actions **eins nach dem anderen** testen und ggf. debuggen, bevor er sie im realen Gespräch einsetzt. Gerade die Formatierung der Ergebnisse (z.B. rohes JSON der `get_notes_by_tag` Ausgabe) muss evtl. im Prompt oder via Few-Shot-Beispiel der KI beigebracht werden, damit sie dem Nutzer das hübsch präsentiert. Ein einfacher Weg: Den KI-Antwort-Teil nach einer Memory-Abfrage im Systemprompt regeln (z.B. „Wenn `get_notes_by_tag` genutzt wurde, fasse die erhaltenen Notizen knapp zusammen, anstatt nur JSON anzuzeigen.").

4. Best Practices & Warnungen

- **Stilkonstanz bewahren:** Damit die KI „nicht verwässert", muss Andres gewünschter Stil konsequent eingehalten werden. Das Profil betont Ehrlichkeit, Klarheit und einen trockenen, analytischen Ton. Um das sicherzustellen, sollte Andre im Systemprompt auf *jedes* Element achten, das den Stil beeinflusst. Vermeide generische Phrasen oder Widersprüche darin. Zusätzlich kann man in den ersten Nutzereingaben den Ton verstärken (z.B. Andre spricht selbst technisch/prägnant, was die KI spiegeln wird). Sollte die KI im Verlauf doch abdriften (etwa wieder weicher oder zu formell werden), ist es sinnvoll, per Feedback gegenzusteuern: Andre kann der KI im Chat unmittelbar sagen („Bleib bitte sachlich."), oder besser noch, den **Systemprompt nachzubessern** für nächste Sessions. Ein Beispiel: Falls die KI anfängt zu „coachen" im typischen Motivationssprech, könnte Andre im Profil/Systemprompt deutlicher festlegen: „Du bist ein Anti-Coach – vermeidest jede manipulative Motivationsrhetorik." Solche Korrekturen halten den Stil hart und zielklar. Wichtig ist, Änderungen nachvollziehbar zu versionieren, wie im Profil vorgesehen [6], damit man weiß, welche Anpassung welchen Effekt hatte. In Summe gilt: Die Persönlichkeit der KI ist die Summe aus Profil + Systemprompt – diese sollten konsistent dieselbe Sprache und Haltung vermitteln, damit keine Vermischung auftritt.

- **Keine Überfrachtung der Architektur:** Jede neue Funktion oder Regel erhöht die Komplexität. Gerade am Anfang ist *weniger mehr*. Eine Warnung ist angebracht, zu viele Sondermodi, Ausnahmen oder externe Plugins gleichzeitig einzubauen. Overengineering zeigt sich z.B., wenn die KI mehr mit sich selbst (Regeln prüfen, Aktionen verwalten) beschäftigt ist als mit dem Nutzer. Um dem vorzubeugen, stets fragen: **Dient dieses Feature direkt einem der Kern-Use-Cases?** Wenn nein, kann es vermutlich warten. Ein Beispiel: Die Idee eines ausgefeilten *Modus-Systems* (Absolut, Kreativ, Explorativ, etc.) klingt interessant, bringt aber in V0.1 wenig praktischen Zusatznutzen für die definierten Ziele – es könnte die Antworten sogar inkonsistent machen, falls der Modus falsch gewählt wird. Besser ist, die KI lernt zunächst einen einheitlichen Modus, der zu Andre passt (analytisch-kreativ, aber ergebnisorientiert). Weitere Modi kann man später immer noch einführen, wenn ein echter Bedarf erkannt wird. Auch sollten die **Actions minimalistisch** gehalten sein – z.B. nicht gleich Datenbank-Queries, Web-Scraping und E-Mail-Versand integrieren, solange das Kernproblem Entscheidungsstruktur & Selbstreflexion ist. Eine schlanke Architektur lässt sich leichter debuggen und verbessern. Der Profil-Text selbst erwähnt die **Adaptionsfähigkeit**: das System soll flexibel erweiterbar sein, z.B. durch neue Memory-

Schichten oder Entscheidungslogik – dies spricht dafür, die Basis simpel zu halten, damit Erweiterungen dann modular draufgesetzt werden können.

- **Token-Management & Memory-Duplikate:** Ein praktischer Aspekt ist der Umgang mit Tokens und redundanten Informationen. Da der Kontext begrenzt ist, sollte man vermeiden, dass dieselben Inhalte immer wieder geschickt werden. **Profil vs. Memory:** Entscheide klar, was ins statische Profil gehört (zeitunabhängiges Wissen) und was als dynamische Notiz ins Memory kommt. Überschneidungen führen dazu, dass Infos doppelt vorkommen. Zum Beispiel, Andres Werte und grundlegende Muster sind im Profil verankert – es wäre sinnlos, diese jedes Mal ins Memory zu kopieren. Umgekehrt sollten *individuelle Ereignisse* oder Beschlüsse ins Memory, nicht ins statische Profil. So bleibt jeder Datensatz an seinem Platz. Weiterhin sollte die KI **nicht bei jeder Antwort das ganze Profil rezitieren** – dafür ist die Wissenssuche gedacht. Die KI kann bestimmte Fakten aus dem Profil bei Bedarf in ihre Antwort einfließen lassen, aber am besten in **kompakter Form**. Falls die Knowledge-File groß ist, läuft im Hintergrund vermutlich ein Retrieval-Mechanismus, der nur relevante Auszüge einfügt (OpenAI Custom GPT fasst evtl. die Wissensdatei vor oder holt Absätze kontextbezogen [7]). Andre sollte darauf achten, in seinen Fragen genug Kontext/Stichworte zu geben, damit diese Suche triggert, ansonsten könnte die KI wichtige Profilinfos „vergessen“. Für Memory-Abfragen gilt ähnliches: Wenn Andre z.B. sehr viele Notizen in einer Kategorie hat, sollte die KI nicht alle unverarbeitet reinkopieren. Besser: entweder die neuesten 3-5 liefern oder eine Zusammenfassung machen. Solches Verhalten kann man der KI einprogrammmieren (z.B. in der Action-Handling-Logik: „Wenn mehr als 5 Ergebnisse, fasse zusammen.“). **Token-Hygiene**: Es lohnt sich auch, ältere Memory-Einträge zu archivieren oder zusammenzufassen (vielleicht außerhalb der Haupt-Memory-Datei), falls sie nicht mehr relevant sind. Dadurch behält das *Andre_KI_Memory.md* eine manageable Größe und Suchergebnisse bleiben präzise. Und zuletzt: Im Chat selbst muss die KI nicht jedes Mal lange Antworten geben – gerade wenn technischer Inhalt gefragt ist, lieber präzise als ausschweifend. Der Profilwert *2901: Token-Toleranz hoch* signalisiert zwar, dass Andre gegen lange, tiefe Analysen nichts hat, aber dennoch sollten Redundanzen vermieden werden. Die KI darf ruhig ausführlich sein, **wenn** es der Sache dient (Detailtiefe ist erwünscht), aber nicht durch Wiederholung. Eine klare Regel: Jede Antwort soll neuen Mehrwert bieten, keine reinen Wiederholungen aus Memory oder Profil.

- **Iterative Verbesserungsstrategie:** Ein dauerhafter Lern- und Anpassungsprozess ist eingeplant – das Profil selbst ist *“Work-in-Progress”* [6]. Daher sollte Andre die Architektur auch iterativ verbessern. Best Practice: **Änderungen immer einzeln testen.** Wenn z.B. die KI eine bestimmte Art von Fragen falsch handhabt, überlege, ob man dies durch eine kleine Prompt-Anpassung beheben kann. Fügt man neue Actions hinzu (z.B. in V0.2 eine `analyze_patterns` Funktion für Langzeitstatistiken), sollte man prüfen, ob die KI nicht dadurch abgelenkt wird vom Tagesgeschäft. Jede Erweiterung bringt wieder das Risiko, dass die KI unerwartete Dinge tut – deshalb nach jeder Änderung ein kurzes *Reality-Check*-Gespräch mit der KI führen: Fragt sie nach wie vor nach Bestätigung für Memory-Einträge? Bleibt der Ton gleich? usw. Dokumentiere die Erkenntnisse (ggf. wieder im Memory oder externen Changelog). So entsteht ein **Feedback-Loop**: KI-Ausgaben -> Andre’s Feedback -> Profil/Systemprompt-Anpassung -> nächster Versuch. Falls möglich, sammle auch ein paar Metriken: z.B. wie oft hat die KI `save_note` wirklich genutzt, wann hat sie es unterlassen obwohl sinnvoll? Solche Beobachtungen können dann zur Feinjustierung der Trigger führen (evtl. muss man der KI mutigeres Verhalten erlauben, oder im Gegenteil strengere Bedingungen setzen). Insgesamt sollte Andre geduldig vorgehen – eine personalisierte KI dieser Art lernt man am besten *durch Gebrauch*. Anfangs können Fehler passieren (falsche Abrufe, Missverständnisse), aber diese sind Gelegenheit, die Instruktionen zu präzisieren. Wichtig: größere Änderungen am Profil immer versionieren und ggf. kommentieren,

um nachvollziehbar zu halten, was warum geändert wurde. So bleibt die Entwicklung geordnet und das System wird schrittweise robuster.

5. Roadmap-Vorschlag

1. **Profil finalisieren & segmentieren:** Als ersten Schritt stellt Andre sicher, dass sein umfassendes Profil in einer sauberen Form vorliegt. Gegebenenfalls teilt er es auf: z.B. **Systemprompt-Teil** (Kernaussagen, siehe 3) und **Knowledge-Teil** (Detailprofil als Datei). Er kann jetzt schon unwichtige Teile ausklammern und Markierungen/Tags einfügen, wo später Memory-Referenzen auftauchen sollen. Ziel dieses Schritts: ein **stabiler Blueprint** [1], der als Grundlage dient. Sobald das Profil steht, lädt er die Wissensbasis-Datei in den OpenAI GPT-Builder hoch (oder falls der Builder nur Copy-Paste erlaubt, fügt die wichtigsten Inhalte manuell hinzu).

2. **CustomGPT Grundgerüst aufsetzen:** Nun erstellt Andre in der OpenAI-Weboberfläche einen neuen Custom GPT. Er wählt das gewünschte Modell (vermutlich GPT-4 für bessere Kontexthandhabung). Dann fügt er den ausgearbeiteten **Systemprompt** ein. Er konfiguriert außerdem die **Wissensdatei** (z.B. lädt *Andre_profile.txt* hoch). Andere Einstellungen wie Name, Sichtbarkeit etc. werden gesetzt. An diesem Punkt hat er eine funktionale Grund-KI *ohne* Actions. Er kann ein paar einfache Dialoge führen, um zu prüfen, ob das Profil greift – etwa Fragen zu seinen Werten („Was ist Andre besonders wichtig?“) oder zu einem typischen Dilemma. Die KI sollte aus der Wissensbasis antworten (wenn alles klappt, sieht man im Hintergrund eventuell, dass sie Inhalte aus dem Profil zitiert). Dieses Basissystem geht nun „live“ in dem Sinne, dass er es im Chatfenster nutzen kann, aber es ist noch nicht persistent im Gedächtnis.

3. **Memory-Backend & Actions integrieren:** Parallel oder als nächstes sorgt Andre für das **Memory-Backend**. Evtl. hat er schon einen kleinen Webdienst dafür entwickelt (Schritt 3 aus voriger Liste). Jetzt integriert er diesen via Actions. In den GPT-Einstellungen gibt es den Bereich *Actions* (bzw. Tools/Funktionen). Dort fügt er zunächst `save_note` hinzu: trägt Endpoint-URL ein, Methode POST, definiert Parameter (Content als string, Tags als array etc.), setzt den Auth-Key ein. Danach `get_notes_by_tag` (GET, Parameter tag) und `append_project_log`. Jeden Eintrag nach dem Erstellen einmal test-triggern (oft gibt es im Developer-UI eine Möglichkeit, die Action direkt auszuprobieren). Wenn alles gut aussieht, veröffentlicht er die neuen Actions. **Wichtig:** Einige Minuten für eine sanity check: Sind die Parameter-Namen in Prompt und Backend konsistent? Erwartet das Backend JSON und bekommt auch korrekt formatiertes JSON? Solche Details klären, bevor die KI es nutzt.

4. **Systemprompt erweitern – Tools-Anleitung:** Nachdem die Actions vorhanden sind, ergänzt Andre seinen Systemprompt um einen Abschnitt zur Tool-Nutzung (falls nicht schon geschehen). Hier beschreibt er der KI in ein paar Sätzen, wann welche Action sinnvoll ist (siehe 2d und 3). Er kann auch Beispiele nennen: z.B. „Wenn ich sage 'Notiere ...', benutze `save_note`.“ – eventuell als *Delimitierter* Beispiel-Dialog im Systemprompt oder einfach in Worten. Diese Anleitung stellt sicher, dass das Modell die neuen Fähigkeiten *versteht*. Gegebenenfalls muss er etwas mit Formulierungen experimentieren; manchmal reicht auch die Action-Beschreibung selbst, aber explizite Prompt-Instruktionen geben mehr Sicherheit.

5. **Testlauf Version 0.1:** Jetzt kommt ein ausführlicher **Praxis-Test**. Andre startet eine Unterhaltung in seinem neuen CustomGPT und probiert die Kern-Use-Cases der Reihe nach aus:

6. Lässt sich die KI bei einer **Entscheidungsüberforderung** an? (Beispiel: Andre schildert 5 Optionen und sagt er sei überfordert. Die KI sollte strukturieren, evtl. nach Zielen fragen, eine Tabelle o.ä. – hier ohne Memory-Einsatz erstmal.)
7. Fragt Andre nach einer **technischen Erklärung** eines Systems, um Realismus zu prüfen (KI sollte konkret und faktisch antworten, keine generischen Wiki-Antworten).
8. Simuliert Andre **selbstsabotierendes Verhalten** oder negative Selbstgespräche, um zu sehen ob die KI es erkennt und trocken spiegelt.
9. Tut Andre so, als würde er einer ungeliebten Aufgabe in **Fun-Flucht** entkommen (z.B. „Ich zocke lieber, statt X zu erledigen.“). Die KI sollte möglichst einen spielerischen Reframe anbieten („Wollen wir X als Quest gamifizieren?“) und vielleicht fragen ob sie das festhalten soll.
10. Testet die **Memory-Funktionen** explizit: Sagt Andre „Notiere: Ich habe heute Feature Y fertiggestellt (#ProjektX)“, dann sollte ein `save_note` erfolgen. Fragt er dann „Was habe ich zu ProjektX?“, sollte `get_notes_by_tag` kommen und die KI die gespeicherte Notiz wiedergeben.

Diese Test-Dialoge decken ab, ob alles zusammen spielt. Wo es hakt, notiert Andre sich die Beobachtung. Typische Anpassungen könnten jetzt sein: Prompt justieren (wenn KI z.B. Memory nicht nutzt wo sie sollte), Action-Feinheiten anpassen (z.B. Backend-Response-Format ändern, falls das Parsing schwierig ist), oder Wissensfile ergänzen (falls die KI wichtige Begriffe nicht kannte). Ziel ist, dass V0.1 **stabil läuft** für die Hauptszenarien. Stabil heißt: KI folgt den Regeln, nutzt Memory gezielt, und bleibt im gewünschten Stil.

1. **Feedback & Backpropagation:** Nachdem Version 0.1 einsatzfähig ist, sammelt Andre im realen Gebrauch weiteres Feedback. Alles, was auffällt – sei es eine tolle neue Erkenntnis, die ins Profil aufgenommen werden soll, oder ein Fehlverhalten – wird genutzt, um das System zu verbessern. Der Profil-Blueprint wird kontinuierlich verfeinert und versioniert [6] . Mögliche nächste Schritte (Roadmap darüber hinaus) könnten sein:
2. **Version 0.2:** Einführen von erweiterten Memory-Funktionen, z.B. *Suche* nach Stichwort innerhalb aller Notizen, oder ein Action `analyze_memory` die Statistiken zieht (damit könnte die KI z.B. sagen: „Du hast in den letzten 3 Monaten 5x Entscheidungsparalyse notiert, es wird seltener.“ – das erfüllt den Langzeit-Analyse Punkt). Oder Integration eines externen Kalenders, falls Entscheidungsstruktur das erfordert.
3. **Version 0.3:** Hinzufügen eines *Modus-Systems*, falls dann noch gewünscht. Z.B. Kommandos wie "`Mode: Creative`" um explizit Brainstorming vs. "`Mode: Direct`" für absolute Sachlichkeit. Das müsste im Prompt und eventuell als Parameter umgesetzt werden.
4. **Fortlaufend:** Evaluieren, ob die KI wirklich Mehrwert liefert in Andres Alltag. Anhand dessen eventuell **Neugewichtung** der Features: Vielleicht zeigt sich, dass spielerische Quest-Reframes super helfen – dann könnte man dem einen größeren Platz einräumen (z.B. eine Action, die aus einer Aufgabe automatisch ein kleines Text-Adventure generiert). Oder es zeigt sich, dass Andre oft ähnliche Fragen stellt – dann könnte man vorbereitete Snippets ins Wissen aufnehmen.
5. Bei all dem immer darauf achten, dass die *Persönlichkeit* der KI konsistent bleibt. Änderungen daher bevorzugt im bestehenden Rahmen machen statt komplett neue Persona aufsetzen. Die Roadmap sollte genügend Zeit für Test und Anpassung jeder neuen Capability vorsehen, bevor weitere Schichten draufkommen. Mit dieser schrittweisen Vorgehensweise wird Andre „das System live bekommen und iterieren“, ohne den Überblick zu verlieren.

6. Snippets (Beispiele)

Systemprompt-Header (Ton & Ziel):

```
Du bist **Andres persönlicher KI-Assistent**. Dein Verhalten: analytisch,
direkt, ehrlich. Du vereinst mehrere Rollen – u.a. Selbstsabotage-Analyst,
Systemdenker, Red-Team-Taktiker – um Andre bestmöglich zu reflektieren.
**Dein Ziel**: Struktur in Entscheidungen bringen, technische Systeme
realistisch erklären, schädliche Muster erkennen und schonungslos spiegeln,
sowie kreative "Quest"-Aufgaben stellen, wenn Andre feststeckt.
**Stilregeln:** Keine Smalltalk-Floskeln, kein Coaching-Geschwätz. Sei
prägnant, technisch präzise und ggf. humorvoll trocken. Du hast Zugriff auf
Andres Profil-Wissen und einen persistenten Memory-Speicher – nutze beides
**nur situativ** und mit Nutzerzustimmung.
```

Beispiel-Action `save_note`**:**

```
# Action: save_note
description: "Persistiert eine Notiz in Andre_KI_Memory.md (persistentes
Gedächtnis)."
method: POST
endpoint: https://api.andre-ai.local/memory/note   # (Beispiel-URL des
Memory-Service)
parameters:
  content: string    # Inhalt der zu speichernden Notiz
  tags: [string]     # optionale Tags zur Kategorisierung (z.B.
["entscheidungsfindung","muster"])
# Aufruf-Beispiel (durch KI):
# save_note({"content": "Entscheidung 'Studium weiterführen' getroffen.",
"tags": ["Entscheidung","Selbstreflexion"]})
```

Entscheidungslogik (Memory-Zugriff) – Pseudocode:

```
# Wann soll die KI auf den Memory zugreifen?
wenn Nutzer explizit um Speicherung bittet (z.B. enthält Eingabe "notiere"
oder "merk dir"):
    → Aktion save_note(content=<vom Nutzer genannte Info>)
elif Nutzer nach früheren Inhalten fragt (z.B. "Was habe ich über X schon
notiert?"):
    → Aktion get_notes_by_tag(tag="X")
elif ein bekanntes Muster deutlich erkennbar ist (z.B. Anzeichen für
Entscheidungsparalyse):
    → KI weist darauf hin und fragt, ob sie dazu etwas aus dem Memory holen
oder notieren soll
    (z.B. "Du wirkst unentschlossen – soll ich frühere Erkenntnisse dazu
abrufen?").
sonst:
    → keinen Memory-Zugriff durchführen (KI bleibt im aktuellen Kontext).
```

Diese Snippets illustrieren, wie einzelne Teile der Lösung aussehen könnten. Zusammengefügt und sorgfältig implementiert, ergibt sich ein **robustes, personalisiertes KI-System**, das exakt auf Andre zugeschnitten ist – mit klarer Persönlichkeit, funktionaler Präzision und der Fähigkeit, aus Erlebtem zu lernen.

---

1 3 4 5 6 Andre_profil_Full_MD..txt
file://file_00000000bfe8720a8af3cf8db84de180

2 Does memory function with GPTs? | OpenAI Help Center
https://help.openai.com/en/articles/8983148-does-memory-function-with-gpts

7 Custom GPT Knowledge Versus External Actions
https://community.openai.com/t/custom-gpt-knowledge-versus-external-actions/702827