**📘 Metatron Tutor Backend System Design – v1.2**

This document defines the backend architecture, data flow, and orchestration logic for the **Metatron Tutor Engine**, with specific focus on modularity, progress tracking, and prompt generation.

**🧠 Consolidated View: Core Backend Components**

| **🔧 Component** | **🧠 Function** | **🔌 API Role** | **🗃️ Database Tables** | **💬 Used During** |
| --- | --- | --- | --- | --- |
| **Prompt Compiler** | Builds prompt from all sources (master, goal, insights, trackers) | Generates final session prompt | None (reads from all trackers) | Session Start |
| **Session Starter Engine** | Accepts goal ID, calls compiler, starts chat session | /session/start | sessions, users, goals | Session Start |
| **Goal Plan Tracker** | Static structure from uploaded content or manual config | /goal-plan/{goal\_id} | goal\_plans, milestones, topic\_tree | Prompt build, tutoring |
| **Progress Summary Engine** | Tracks actual learner performance from chat | /progress/summarize | chat\_logs, progress\_snapshots, insight\_tags | Mid-session & End |
| **Tutoring Cycle Tracker** | Tracks pedagogical stages (Engage → Repeat) | /tracker/{goal\_id} | cycle\_stages, stage\_repeats, topic\_flags | During session |
| **Goal Content & Plan Progress Tracker** | Tracks learner progress against content & milestones | (Updated via Progress Engine) | plan\_progress\_snapshots | Mid-session |
| **Dynamic Placeholder Engine** | Injects real-time context into prompts ({{user}}, {{stage}}) | (Internal only) | None | Prompt Build |
| **Report Generator** | Builds per-session reports (PDF, JSON) | /report/generate | reports, summaries, flags | Session End |
| **Tutor Brain Backend Engine** | Orchestrates all engines above, routes calls, logs activity | (Internal service layer) | None (routes and orchestrates) | Always |

**🔄 High-Level Backend Flow (with Updated Snapshots)**

[❶ Upload File / Define Goal]

→ Goal Plan created from file (topics, milestones, headings, summary)

→ Goal Plan DB snapshot stored in `goal\_plans`

→ Initializes empty trackers:

- Progress Snapshot Tracker → `plan\_progress\_snapshots`

- Tutoring Cycle Tracker → `cycle\_stages`

- Goal Plan Tracker → `goal\_plans`

→ Vector store and knowledge base created if embeddings used

[❷ Tutor Session Starts]

→ `/session/start` called with goal ID

→ SessionMeta entry created

→ Prompt Compiler pulls:

- Master prompt

- Goal-specific prompt

- Dynamic placeholders (`{{user\_name}}`, `{{topic}}`)

- Goal plan

- Most recent progress + cycle snapshot

[❸ Chat Begins]

→ Messages stored in `chat\_logs`

→ Every N messages or idle timeout:

→ Progress Summary Engine:

- Analyzes content understanding

- Evaluates lifecycle position

- Updates progress snapshots

- Triggers insight generation

[❹ Tutor Adapts Based on Tracker Data]

→ Struggles → Repeat prior stage

→ Mastered → Skip ahead, quiz

→ Uncertain → Injects reflection questions

[❺ Session Ends]

→ Final progress + cycle + plan snapshots saved

→ Session summary report generated (`reports`)

→ Tracked for future recall

[❻ Future Sessions]

→ Past snapshot + plan data injected into prompt

→ Tutor resumes seamlessly

**📡 API Interface Plan**

| **API Route** | **Purpose** |
| --- | --- |
| POST /session/start | Initiate session, compile prompt, create session record |
| GET /prompt/compiled/{session\_id} | Return generated prompt |
| GET /goal-plan/{goal\_id} | Return structured milestones, plan |
| PATCH /goal-plan/update | Update or override plan structure |
| POST /progress/summarize | Analyze last N messages, return tags, update tracker |
| GET /progress/{session\_id} | Fetch latest progress snapshot |
| GET /tracker/{goal\_id} | Fetch current cycle or lifecycle stage |
| POST /report/generate | Create downloadable report from session state |

**🗂️ CENTRAL FILES TO BE DESIGNED**

| **File** | **Purpose** |
| --- | --- |
| models.py | All schemas and data types |
| api/session.py | Start sessions, store session state |
| api/goal\_plan.py | Serve and update plan structure |
| api/progress.py | Summarize chat logs and return insight |
| prompt\_engine/compiler.py | Compiles master prompt with metadata |
| chat\_engine/handler.py | Orchestrates chat using prompt + memory |
| utils/placeholder.py | Injects dynamic data into prompt slots |

**🧠 PHASE 1: SYSTEM ANALYSIS & MAPPING**

Before writing code, we define all "Process → Data → Table → API" relationships:

| **Spec Piece** | **Description** |
| --- | --- |
| **Purpose** | What does this module do? Who needs it? |
| **Process Flow** | What steps does it perform and when? |
| **Input Requirements** | What data does it need and from where? |
| **Output/Actions** | What does it return, store, or call next? |
| **DB Fields Required** | Table name and key fields |
| **APIs Needed** | GET, POST, PATCH routes with expected params |

**✅ Next Design Steps (Before Coding)**

**🔲 Step 1: Finalize Field Schema Maps**

Each tracker must define:

* Field names
* Data types
* Unique keys
* Foreign keys (e.g., session\_id, goal\_id)
* Validation rules (nullable, enum, etc.)

✅ **Ready to begin now**

**🔲 Step 2: Build System Flow Diagrams**

Recommended diagrams:

* One master “Tutor Session Lifecycle”
* One per engine:
  + Prompt Compiler
  + Tracker Coordination
  + Progress Analyzer

**💡 Design Guidelines**

* **Orchestration via Tutor Brain Backend Engine**: All modules interact via this layer
* **Snapshot-Based Tracking**: All changes are saved as time-based snapshots for auditing and re-entry
* **Tracker Extensibility**: Trackers support dynamic lifecycle stages to allow future modifications
* **Placeholder Personalization**: Prompts adapt to context and learner dynamically
* **Decoupled APIs**: APIs expose only what's needed to the frontend — never raw engine access
* **Embeddings Optional**: Knowledge Base is modular and can swap between OpenAI and OSS models

Let me know if you’d like this content converted back into a Word or PDF file now that the full body has been reconstructed.

Would you like to proceed with **Step 1: Tracker Schema Field Maps** now?