ChatGPT

Phase 1: Tech Stack & Architecture

- **Slack Bot Framework:** Use Slack's official "Bolt" framework, which is available for both JavaScript/TypeScript and Python [1]. For example, you could write the bot in Node.js using Bolt for JavaScript/TypeScript (or in Python with Bolt for Python) to handle Slack events and messages. Slack's developer guide notes that both Bolt for Python and Bolt for JavaScript are supported and even allow custom functions in workflows [1]. If you prefer Next.js, you can embed Bolt into a Next.js API route. A common approach is to use a tool like `next-connect` (NextConnectReceiver) so that Slack's HTTP events work smoothly within Next's API routes [2]. Alternatively, a standalone Node/Express or Python (FastAPI/Flask) server also works.
- **LLM Backend (Groq):** Use Groq's API (which supports LLaMA 3 models) for generating plans. Groq provides client libraries (e.g. `pip install groq` for Python or an NPM package for Node) that let you send chat completion requests. For example, in Python you would initialize a client with your API key and call:

```
chat_completion = client.chat.completions.create(
    messages=[{"role": "user", "content": "…"}],
    model="llama3-70b-8192",
)
```

  This returns the LLM's response [3] [4]. (In Node/TypeScript you'd use the analogous Groq library or direct REST calls.) By abstracting the LLM calls into a service layer, you can easily switch models or providers later.
- **Database & Persistence:** Use MongoDB to store all persistent data: the user's **profile** (background, preferences, routines, etc.), weekly goals, and daily task logs. For example, you might have collections like `UserProfile`, `WeeklyPlans`, `DailyLogs`. On first use, populate `UserProfile` with the user's info; thereafter, update it via explicit functions (see Phase 4 below). Because this is a single-user, personal assistant, you don't need a multi-tenant design—just one MongoDB instance (local or hosted) is fine.
- **Scheduling Messages:** Use Slack's scheduling or a server-side cron to trigger the bot. The Slack API provides `chat.scheduleMessage` to post messages at future times [5]. (You can schedule messages up to 120 days ahead.) For example, every Sunday at 9 PM you'd schedule a DM to the user saying "Let's plan your week" [5]. Similarly, schedule a daily DM each evening at 10 PM to review the day and plan tomorrow. (Alternatively, you can run a cloud cron or use something like AWS Lambda / Google Cloud Function on a schedule to call `chat.postMessage` or your own bot logic at those times.)
- **Overall Architecture:** Conceptually, your Slack app will receive events via HTTPS (for messages or slash commands) and also initiate DMs on schedule. A possible cloud architecture is illustrated below – the bot runs in a backend (e.g. server or Lambda) that handles Slack events and calls Groq for LLM completions. (The image shows a similar setup with Slack Bolt and an LLM service.)

Figure: Example high-level architecture for a Slack AI assistant. Slack events (DMs, etc.) go to your backend (Bolt app on a Node/Python server). The app uses Groq's LLM API to generate responses/plans. Persist user data in MongoDB.

```
Slack messages (events) → [Bolt App / API] → calls Groq LLM → returns plan →
posts back to Slack【1†L98-L100】.
A hosting choice could be a cloud function or a container (e.g. on Vercel, AWS
Lambda, Heroku, etc.).
```

Phase 2: Weekly Planning Conversation

- **Sunday Planning Prompt:** Every Sunday evening at 9 PM, the bot initiates a DM (either via a scheduled message or a cron job) saying "Hey! Let's plan your week. What are your main goals?" The bot should engage in a conversational thread, asking follow-up questions as needed. For example, it might clarify due dates or priorities ("You mentioned a MongoDB event on Tuesday – what time is that?"). Each reply from the user is captured and stored temporarily. Continue the back-and-forth until the user's weekly goals are clear.
- **LLM-Driven Plan Generation:** Once the bot has the goals (e.g. "MongoDB event on Tuesday", "CS556 homework by Friday"), it invokes the LLM (via Groq) to create a realistic weekly schedule. The prompt to the LLM should include: the user's profile info (from MongoDB), their stated weekly goals, known routines/constraints (energy levels, classes, etc.), and any tasks already completed. The LLM is instructed to output a day-by-day plan. For example, you might feed in a prompt like: *"This user has high energy in mornings. Tasks: Event on Tue, Homework due Fri (finish early). Plan the week with timed activities."* and receive a structured plan.
- **Saving the Plan:** After generating the weekly plan, save it to MongoDB (`WeeklyPlans` collection) and also push it to Slack (e.g. as a message or thread summary). The plan can be formatted as a plain-text schedule or Slack blocks. You might use Slack's markdown (mrkdwn) to format times and tasks. No matter what, ensure it's clear and actionable.

Phase 3: Daily Check-In & Adaptation

- **Evening Check-In:** Each weekday evening (around 10 PM), the bot checks in. It should DM the user something like: *"How did your day go? Which tasks did you complete? Any changes?"* Gather the user's input on progress: which tasks are done, what's left, or any new priorities. Store this in MongoDB (`DailyLogs`).
- **Adjusting Tomorrow's Plan:** Using the updated status, call the LLM again to schedule **tomorrow**. The prompt should include: the remaining tasks for the week (from the original plan minus completed items), any newly added tasks or urgent items, and the user's feedback (e.g. "I fell behind on homework" or "I finished early"). Ask the LLM to produce a realistic plan for the next day's schedule. For example: *"You finished problems 1–3 of homework and have 2–5 left. Tomorrow is Tuesday with an event at 3 PM. Plan tomorrow."* The LLM output is then sent as a DM (and/or thread message) for the user's morning, e.g.:

```
Tomorrow's Plan:
- 8:00 AM – Morning routine
- 9:00 AM – Continue CS556 hw: aim to finish problems 4-5
- 12:00 PM – Lunch
- 1:00 PM – MongoDB event prep
- 3:00 PM – Attend MongoDB event
- 5:00 PM – Review university lecture notes
- 7:00 PM – Dinner
- 9:00 PM – Light review for Wednesday
- 10:00 PM – Check-in about tomorrow
```

You can format this with Slack's markdown or block kit.
- **Threaded Conversation:** To keep each day's plan and check-in organized, use Slack threads. For example, the nightly check-in message can start a thread where follow-ups happen. If you need to send the *next morning's plan*, you could reply in that thread or start a new DM. By including the same `thread_ts` on follow-up posts, Slack groups the messages. As one guide explains, "To send a message as a thread … add the `ts` key to the request. The resulting threaded message now looks like this" [7]. This makes the interaction feel cohesive.

Phase 4: Memory & Profile Management

- **Profile Storage:** Store the user's personal details (background, schedules, preferences) in MongoDB. For example, `UserProfile` might contain fields like `name`, `energyLevels`, `classSchedule`, `personalGoals`, etc. Initially populate this through the onboarding conversation (e.g. bot asks about routines, deadlines, preferences) and save the answers.
- **Updating Profile via Functions:** Whenever the user shares new personal info ("I work late at night" or "I have a meeting every Thursday at 2 PM"), update the profile with a function call. For instance, you can design an "updateProfile(field, value)" function that the bot invokes. This is similar to structured memory systems where new facts are maintained via functions. (In research, for example, a memory architecture "calls a maintenance function before storing a preference" to decide how to update stored data [8].) Concretely, after each relevant user message, parse out any new facts (possibly by pattern-matching or even asking the LLM to extract facts) and write them to MongoDB. Over time this profile grows with your habits and commitments.
- **Contextual Prompts:** When calling the LLM, always include relevant profile data from MongoDB in the prompt. Because we're not using a vector DB/RAG, you will manually assemble context. For example, your prompt template might start with: "User profile: Name=…, MorningEnergy=High, Works Late=Yes, Loves Walking=Yes. Goals for week: …" etc. This ensures the assistant "knows" your background. (This approach sidesteps RAG: all needed memory is inserted as text into each prompt.)

Phase 5: Slack UX & Interaction Style

Figure: Sample Slack DM conversation (from the "Lucy" bot) where the assistant learns and recalls user details. The assistant should behave like a friendly, proactive human assistant. For example, it can recall the user's name or preferences. In one Slack example, the bot asked the user's name ("I don't have your name yet.

What’s your name?”) and then later greeted them by name (“Hey Kuba!”) [9]. Similarly, your bot should address you personally and use past details naturally.

- **Threading & Replies:** Use Slack threads to group related messages, as mentioned. This prevents spamming your DM with multiple standalone messages. The bot’s initial question can start a thread, and all further prompts/responses go under it (using `thread_ts`) [7]. This makes it feel like an ongoing chat.
- **Emoji Reactions:** Add emoji reactions to messages to give feedback or encouragement. For example, if you report “I completed the task early,” the bot could react with a emoji. Slack’s API method `reactions.add` lets the bot attach an emoji by specifying the channel and timestamp of the message [10]. This small gesture makes the bot feel more engaged and human-like.
- **Tone and Personal Touch:** Ensure the bot’s language is supportive and conversational. You might explicitly instruct the LLM in the system prompt to be “helpful, friendly, and encouraging.” Use casual phrasing (e.g. “Great job on finishing that!”). Mention celebrating wins to keep motivation high. While this is about prompt engineering rather than external sources, it’s key for a natural experience.

Putting It All Together

1. **Setup:** Create a Slack App and enable a bot user. Grant it scopes like `chat:write`, `chat:write.public`, `im:write`, and event subscriptions for messages or shortcuts. Host your code (Node/Python) on a serverless platform or VM. Connect to MongoDB for data. Initialize Groq client with your API key.
2. **Onboarding:** First time the bot runs, prompt the user for profile info (name, timezone, classes, preferences). Store these in `UserProfile`.
3. **Scheduled Prompts:** Use Slack’s scheduler or an external timer to send the weekly prompt on Sunday night and daily prompts each weekday night [5].
4. **Conversation Handlers:** Write handlers for incoming messages (in your Bolt app) that capture user replies in the planning process and daily check-ins. When the user provides info, save it or use it in the prompt. After gathering goals or status, trigger the LLM to generate the next plan.
5. **Memory Updates:** After each user message, optionally parse it (manually or with the LLM) for new facts and update the MongoDB profile via a helper function. For example, if you say “I usually sleep at 1 AM”, call `updateProfile("sleepTime", "1 AM")`. This mimics a structured memory update [8].
6. **Iterate & Improve:** Monitor how well the assistant’s plans match reality. You can log actual vs planned tasks in MongoDB. Over time, feed this into prompts (e.g. “Last week you only completed 60% of planned tasks, so be more realistic.”). Gradually refine the prompts and planning logic based on feedback.

By following these phases – choosing the right tech stack, integrating Groq, leveraging MongoDB for state, and using Slack’s APIs for scheduling, threading, and reactions – you’ll build a Slack bot that proactively plans your week, checks in daily, and adapts to you. Over weeks it will feel more personal (learning your habits and pace) and help bridge the gap between your goals and actual productivity.

Sources: We used Slack’s developer docs and tutorials on building Slack bots [1] [5] [7] [10], a Slack example integration with LLMs [6], guides on integrating Slack Bolt into Next.js [2], and examples of LLM usage (Groq API) [4] [3]. We also drew on a memory architecture paper [8] and a Slack AI assistant case study [9] to inform our design.

[1] Building on Slack Just Got a Lot Easier — New Tools for Developers and Admins Available Today | Slack
https://slack.com/blog/developers/developer-program-launch

[2] Building the Backend for a Slack App with NextJS and Vercel | by Ali BaderEddin | Medium
https://medium.com/@alibadereddin/building-the-backend-for-a-slack-app-with-nextjs-and-vercel-e1503b938e6b

[3] [4] Using Groq API with LLAMA 3 using Langchain — The more conceptual Understanding | by Priyanka Neogi | Medium
https://medium.com/@priyanka_neogi/using-groq-api-with-llama-3-8b0265a88770

[5] chat.scheduleMessage method | Slack Developer Docs
https://docs.slack.dev/reference/methods/chat.scheduleMessage/

[6] Create a generative AI assistant with Slack and Amazon Bedrock | Artificial Intelligence
https://aws.amazon.com/blogs/machine-learning/create-a-generative-ai-assistant-with-slack-and-amazon-bedrock/

[7] Programatic message threading with your Slack Bot | by Sean Rennie | Medium
https://sean-rennie.medium.com/programatic-message-threading-with-your-slack-bot-688d9d227842

[8] CarMem: Enhancing Long-Term Memory in LLM Voice Assistants through Category-Bounding
https://arxiv.org/html/2501.09645v1

[9] Creating Lucy: Developing an AI-Powered Slack Assistant with Memory - DEV Community
https://dev.to/kuba_szw/creating-lucy-developing-an-ai-powered-slack-assistant-with-memory-134o

[10] reactions.add method | Slack Developer Docs
https://docs.slack.dev/reference/methods/reactions.add/