ChatGPT

Step 1: Add Instrumentation and Logging for Diagnosis

Before making big changes, implement **structured logging** in both the parsing and scheduling components. Log key details for each task (ID, text, estimated_minutes, importance scores, flexibility flags, chosen slot, conflicts, preemptions, etc.). This will help verify the effects of each fix. Initially, just record data without changing behavior. These logs will provide a baseline to confirm that later fixes (like routine adjustments or deadline handling) are working as intended.

Step 2: Normalize Routine Tasks (Sleep/Meals) Across All Inputs

Ensure that **all** incoming tasks or events related to sleep or meals go through the same normalization rules (whether they come from user input or calendar sync). Key actions:

- **Classify and Tag Routines:** If a capture's text implies sleep or a meal, set `task_type_hint` to `"routine.sleep"` or `"routine.meal"` accordingly.
- **Set Proper Constraints:** For sleep tasks, treat them as non-overlapping and primarily **night-time tasks**. For example, convert a request like "Sleep today at 12am" into a flexible **night window** (roughly 22:00 to 08:00 local time) rather than a fixed start in the middle of the day. Mark `start_flexibility = "soft"` (it can slide a bit) and `duration_flexibility = "fixed"` (total sleep time fixed, no splitting). Meals should get a preferred time window (e.g. lunch around midday) and also `cannot_overlap = true`.
- **Bypass Working-Hours for Routines:** Update the scheduler logic so that routine tasks **do not use the 08:00–22:00 working-hour filter**. Sleep should be scheduled during night hours by default, and meals in appropriate meal windows, even if those fall outside standard "work" hours.
- **No Auto-Freeze for Routines:** Avoid automatically freezing routine tasks when scheduled. Only lock them in place if the user explicitly set a fixed time. This way, a sleep block can still slide within the night window if needed to accommodate higher-priority tasks.
- **Adjust Priority Scaling:** Downweight the priority of routine tasks so they don't unfairly dominate over important work. For example, cap sleep urgency at a moderate level (not always maximum) and use a multiplier (e.g. 0.5 or 0.6 of normal priority) for routine tasks in the priority score calculation. This ensures critical deadlines can override routines when necessary (while still scheduling the routine somewhere acceptable).
- **One-Time Cleanup:** As a maintenance step, find any existing scheduled sleep tasks that are incorrectly placed during the day (overlapping working hours). Unfreeze those and let the new rules reschedule them into proper night slots. Do the same for any meals at odd times if needed.

Step 3: Enforce Time-of-Day Preferences for Tasks

Update the scheduling algorithm to **respect user-specified times of day** (like "morning", "tonight", "before bed"). This involves narrowing the search window for a task to the preferred period first:

- **Define Standard Time Bands:** For example, define "morning" as roughly 8am–12pm, "afternoon" as 12pm–5pm, "evening" as 5pm–9pm, and "night" as 9pm–2am.

- **Use Preferences in Scheduling:** If a task has `time_preferences.day` (e.g. “today” or “tomorrow”) or `time_preferences.time_of_day` (e.g. “evening”), initially restrict scheduling to that day and time band. For instance, a task tagged for “tonight” should first try to fit into today’s **evening/night** hours. Do not immediately place it the next morning.
- **Two-Pass Scheduling:** Attempt to find a slot within the preferred window on the specified day. If nothing is available there, expand the search gradually (e.g. allow a slightly larger window or try the same time band on the next day), and apply a **penalty cost** for scheduling outside the preferred time. This way, the scheduler will only violate the user’s preferred time if it truly cannot fit the task otherwise – and even then, it will “know” that it’s a compromise (which could be communicated to the user).
- **Example:** “Code and plan day tonight” should end up scheduled during the evening or night **today** if at all possible. Only if today’s evening is completely full would it spill into tomorrow, and that should be a clearly penalized last resort. Similarly, “Apply to jobs before sleep” would be targeted for the late evening on the same day, before the scheduled sleep block.

Step 4: Improve Handling of Hard Deadlines (Urgent Tasks and Missed Deadlines)

Tasks with hard deadlines need special treatment so they don’t get dropped or scheduled too late:

- **Emergency Scheduling for Imminent Deadlines:** If a new task arrives with a **hard deadline very soon** (e.g. due in the next 2–4 hours) and it doesn’t fit in the remaining free time, invoke an “emergency mode.” In this mode, scan the calendar between now and the deadline for lower-priority tasks that can be moved. Proactively **preempt** or bump those tasks to free up a contiguous block for the urgent task. Choose tasks to delay based on priority (move the ones with the lowest priority_score first, that aren’t critical or externally fixed). Continue bumping tasks until enough time is cleared for the urgent task, or until you run out of movable candidates. Then schedule the urgent task in the reclaimed slot. After placing it, try to reschedule the displaced tasks for later in the day (or flag them for the user if they can’t be fit). This ensures that an urgent assignment due soon gets scheduled **immediately**, at the cost of deferring less important things.
- **Deadline-Proximity Freezing:** Change the logic for freezing tasks with deadlines. Instead of freezing a deadline-task as soon as it’s scheduled, only mark it non-movable when it’s very close to its deadline (for example, within a few hours of due time). This way, a task due tomorrow night that you scheduled today can still be shifted tomorrow morning if something even more urgent comes up today. It becomes “locked in” only as its deadline nears, to avoid last-minute thrash.
- **Handle Missed Deadlines Gracefully:** If a hard-deadline task is captured **after its deadline has already passed** (or with not enough time left to schedule), avoid the current behavior of outright failing (HTTP 409 with no plan). Instead, mark the task’s status as `"missed_deadline"` internally and decide how to proceed:
- **Do not auto-schedule by default** (since the deadline was missed), but provide a clear outcome. For example, return a status indicating the deadline was missed and maybe suggest asking the user if they still want to allocate time for it.
- **Option to Schedule Late:** If the user indicates they still want to do it, treat it as a flexible task (or a soft deadline) and schedule it at the earliest available time slot after “now.” When doing so, annotate the scheduled chunk as `late: true` and perhaps record how many minutes past the deadline it will start. This late scheduling should be an explicit choice to ensure the assistant doesn’t silently schedule things after they were due.

- **Partial Credit (Future Enhancement):** (Optional) If a task is big and only part of it could be done before the deadline, you might later implement a partial scheduling suggestion. For now, focus on all-or-nothing: either fit the task fully by the deadline with emergency moves, or handle it as a missed deadline case.

Step 5: Rebalance Priorities Between Routine and Important Tasks

Adjust the priority computation and related constraints so that essential deliverables always outrank routine activities when there's a conflict, while still preserving routines in some form:

- **Tune Importance Values from LLM:** Ensure DeepSeek (or whichever parsing logic provides `importance`) assigns **moderate urgency/impact** to routine tasks. For example, going to bed is important, but usually not more *urgent* than a task due in a few hours. Cap routine tasks' urgency around medium (e.g. 3 out of 5) unless explicitly marked urgent by the user.
- **Priority Score Scaling:** In the scheduler's priority scoring formula, apply a scaling factor or cap for routines. For instance, even if a sleep task has somewhat high urgency, multiply its priority by ~0.5 (or set a max threshold) so that a high-priority work task can still preempt it if needed. This prevents "sleep" or "lunch" from always winning over pressing deadlines.
- **Limit Routine Time Consumption:** Put sane limits on routine durations within the schedule. For example, don't allocate more than ~8 hours for overnight sleep (or whatever is healthy), and maybe ~1 hour for meals unless specified otherwise. This avoids extreme cases where a routine accidentally eats up an entire day. If the user adds multiple "nap" or extra routine tasks, the system should recognize not to over-schedule routine time at the cost of important work.
- **Allow Gentle Routine Shifts:** When a critical task conflicts with a routine, permit small adjustments rather than cancellation. For example, if an urgent meeting overlaps slightly with a lunch break, the lunch could shift earlier or later (within a reasonable range) rather than strictly keeping its original time. Similarly, a sleep block could start a bit later or end a bit earlier one day to accommodate a late-night deadline, but **never remove it entirely or push it into daytime**. The goal is to maintain routines but flex them when absolutely necessary for high-priority conflicts.

Step 6: Fine-Tune Preemption and Scheduling Algorithms

With the major changes above, revisit the overall scheduling logic to ensure it behaves optimally under the new rules:

- **Recalculate Priority Formula if Needed:** After adjusting importance and adding penalties (for time-of-day mismatches, etc.), review the priority scoring and net gain calculations used for deciding moves. Make sure that urgency, impact, blocking, and time-sensitivity (deadlines) are weighted appropriately. For example, a task that is both urgent and high-impact should clearly outrank a routine or a low-impact task in the net gain computation that decides preemptions.
- **Expand Ripple Search Range:** Currently the engine might only consider moving tasks that overlap directly. You can increase the search horizon for freeing up time. For instance, allow the scheduler to consider swapping or moving tasks that are coming up later today if it helps fit a high-priority item now (within reasonable limits). This goes hand-in-hand with the emergency scheduling for urgent deadlines.
- **Guardrails for Too Much Churn:** Implement limits to avoid excessive task shuffling. For example, limit the number of tasks that can be moved (or the total minutes of tasks moved) in one scheduling

run, especially for non-urgent situations. And never move tasks that have a very high reschedule_penalty (e.g. user specifically doesn't want it rescheduled) unless it's absolutely necessary (like in an emergency deadline scenario). These safeguards will keep the system from becoming chaotic after all the new flexibility is introduced.
- **Test and Iterate:** Using the instrumentation from Step 1, closely monitor how often tasks get moved or delayed with these new rules. Adjust parameters like the emergency trigger threshold, freeze horizon, or routine priority cap based on observed behavior to strike a good balance between respecting the user's schedule and accommodating new tasks.

Step 7: (Optional) Implement Arbiter LLM for Complex Conflicts

For especially tricky scheduling conflicts that deterministic rules can't resolve well (e.g., multiple high-priority tasks all clashing at the same time), you can integrate a secondary LLM-based "arbiter" to help decide. This is an optional advanced step once the core system is stable:

- **Arbiter Trigger:** Identify scenarios with several tasks of similar top priority where it's ambiguous which to delay or how to split time (for instance, three deadlines all due today, not enough hours for all). In these cases, prepare a summary of the conflict (tasks involved, their importance, candidate time slots each could move to) and invoke the LLM with a specialized prompt to advise on an optimal compromise.
- **Constrained Output:** The LLM arbiter should *not* choose arbitrary times, but rather pick from the candidate slots you provide it for each task. For example, it might suggest "Task A stays now, Task B move to 3-4pm, Task C move to tomorrow (miss deadline but low impact)" based on the context.
- **Validation:** Treat the arbiter's suggestion as advice. Your scheduler should validate that the suggestion respects all constraints (no overlaps, within allowed windows) before applying it. If it's not valid, you'd fall back to your deterministic rules or ask the arbiter again with more guidance.
- **Usage:** Keep this for the rare tie-breaker cases. In normal operation, the improved priority and preemption logic from earlier steps should handle most conflicts. The arbiter is a safety net for the few truly difficult scheduling dilemmas.

Step 8: User Interface and Feedback Improvements

Finally, refine how the system communicates scheduling changes to the user, to make the experience smooth and transparent:

- **Late Task Messaging:** If a task had to be scheduled after its deadline (or marked as missed), clearly inform the user. For example, "⚠ *'Finish report' was scheduled 1 hour after its original deadline.*" This gives the user awareness that something is off timeline, and they can adjust if needed.
- **Preemption/Debounce Notifications:** When the system moves or delays tasks (especially due to the emergency scheduling in Step 4), let the user know what was moved. e.g. "To fit your urgent task, **2 tasks** were pushed to later in the day." This kind of summary ensures the user isn't surprised by silent changes.
- **Plan Summary Cleanup:** Improve the plan summary output to be more concise. You can hide or group trivial entries in the log. For instance, if a scheduling run only scheduled 1 task and didn't move anything else, you don't need to display a full "scheduled:1, moved:0, unscheduled:0" line – that can be implicit or shown in a collapsed details section. Focus the visible summary on meaningful changes or conflicts.

- **Confirmations for Critical Decisions:** Where applicable, give the user control. For example, if a deadline is missed, present a button or prompt “Schedule anyway” (which toggles `allowLatePlacement` and re-runs scheduling for that task as a late item). Similarly, if the system had to drop a low-priority task entirely to make room, mark it as deferred and perhaps ask the user if that’s okay or if they want to reschedule it manually.
- **Overall UX Polish:** Once the scheduling logic is solid, making the interface intuitive will build trust. Ensure that things like routine adjustments (sleep shifting by 30 minutes, etc.) are either subtle enough not to alarm the user or are explained with a note. By cleaning up the summary and providing context for changes, the user stays in the loop and feels in control despite the automation.

By following these steps, you’ll create a much more robust scheduling assistant. In summary: **first fix the foundational parsing and routine placement issues**, then **enforce user preferences for time**, handle **deadlines and emergencies smartly**, and finally **fine-tune priority logic and UX details**. Each step builds on the previous ones, so implement and test incrementally. Good luck – with these changes, your DiaGuru system should feel far more logical and user-friendly!
