ChatGPT

Improving Clawdbot: A Technical Strategy

Notification Triggers

Clawdbot can proactively alert the user via native Windows toasts (using **win11toast** [1]) whenever it needs attention. For example, wrap key events in FastAPI background tasks:

```
from fastapi import FastAPI, BackgroundTasks
from win11toast import toast

app = FastAPI()

def notify_user(title, message):
    toast(title, message)

@app.post("/ask_question")
async def ask_question(background_tasks: BackgroundTasks):
    question = "I have a question: should I start organizing your tasks for
today?"
    # Trigger a Windows toast notification
    background_tasks.add_task(notify_user, "Clawdbot", question)
    return {"status": "notification sent"}
```

Clawdbot might send a toast (and speak it) in scenarios such as: *it has a clarification question*, *its confidence is low*, or *it has a helpful suggestion*. For example, if the Gemini API response is uncertain or empty, trigger:

```
if response.confidence < 0.5:
    message = "Hmm, I'm not sure about that. Can you clarify?"
    notify_user("Clawdbot asks", message)
    edge_tts.speak(message)
```

ADHD-friendly handling is important: as one user notes, too many notifications train the brain to ignore them. Notifications should be timely and adjustable in urgency [2]. For instance, allow the user to set a "nag level" so reminders escalate only when truly needed. Use `toast(title, message, icon=..., on_click=...)` to customize content or actions [3] (e.g. click-to-open links or scripts).

- **Scenarios**: Low-confidence answer, stalled workflow, upcoming deadline, long idle time, or completion of a task.
- **Implementation**: Use `win11toast.toast("Clawdbot", message)` in Python code when any trigger condition is met [1].

- **ADHD Considerations**: Make frequency/urgency configurable [2]; allow user to silence or snooze reminders if needed.

Voice Variation & Engagement

Clawdbot's TTS uses edge-tts to produce natural voices. To add personality, randomize voice choices and phrasing each time. For example, maintain lists of canned phrases by style:

- **Affirmations**:
- "You're doing great! 👍"
- "Way to go, keep it up!"
- **Cheeky Reminders**:
- "Hey, no daydreaming now 😉"
- "Don't hit snooze on me!"
- **Friendly/Playful**:
- "Need a boost of energy, buddy?"
- "Tell me when you're ready to rock!"

In Python, shuffle these at runtime:

```
import random
from edge_tts import Communicate

affirmations = ["You're doing great!", "Nice work, keep going!", "You've got this!"]
cheeky = ["Hey sleepyhead 😉", "Don't snooze on me now!", "Ready to crush it?"]
flirty = ["Looking sharp today! 😉", "Need some motivation, cutie?"]  # light-hearted tone

style = random.choice([affirmations, cheeky, flirty])
phrase = random.choice(style)

tts = Communicate(text=phrase, voice=random.choice(["en-US-JennyNeural","en-GB-RyanNeural"]))
await tts.save("output.mp3")
```

Here we pick a random voice (e.g. "JennyNeural" or "RyanNeural") and phrase. Use `edge-tts --list-voices` to see available voices with labels like *Friendly, Cheerful, or Empathetic* [4]. We can also randomly

vary `--rate` or `--pitch` for extra flair. Embedding emojis (like "") in TTS text can produce playful inflections.

- **Implementation**: After generating a phrase, call edge-tts either via its Python API or CLI to speak it. For example:

```
from edge_tts import Communicate
voice = random.choice(["en-US-GuyNeural","en-GB-SoniaNeural"])  # male/
female friendly voices
communicate = Communicate(text=phrase, voice=voice)
await communicate.save(f"response.mp3")
# Then play the MP3 or stream to speakers
```

- **Randomization**: Each time Clawdbot speaks, randomly select from each style's phrase list and optionally a voice. Use Python's `random.choice`. Ensure variation so the user doesn't get bored.
- **Persona Tone**: Embed cheeky or flirtatious cues in the text ("", "") and choose voices labeled "Cheerful" or "Empathetic" in edge-tts to match the mood.

Robustness & Reliability

Clawdbot must **gracefully handle errors** and always explain its actions. Key strategies:

- **Retry & Fallback**: Wrap Gemini API calls in retry logic with exponential backoff. For example, use Google's `api_core.retry` or the `tenacity` library. A sample using Gemini Python SDK's retry pattern:

```
from google.api_core import retry, exceptions
@retry.Retry(predicate=retry.if_transient_error, initial=1.0,
multiplier=2.0, maximum=60.0, timeout=120.0)
def query_gemini(prompt):
    response = client.models.generate_content(model="gemini-2.0-flash",
contents=prompt)
    return response
```

This automatically retries on timeouts or rate-limit errors [5]. Alternatively, Tenacity can be used:

```
from tenacity import retry, stop_after_attempt, wait_exponential
@retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1,
min=1, max=10))
def fetch_answer():
    res = requests.post(... )
    res.raise_for_status()
    return res.json()
```

[6].

If Gemini repeatedly fails (e.g. 503 errors), degrade to a safe default or simpler model, and **notify the user**:

```
try:
    answer = query_gemini(user_prompt)
except Exception as e:
    toast("Clawdbot Error", "Service temporarily unavailable. Retrying...")
    edge_tts.speak("Sorry, I ran into a problem. Let me try again in a moment.")
    log_error(e)  # see logging below
    # Optionally: retry once more, or exit gracefully
```

For a higher level fallback, Clawdbot could switch to a cached answer or a simpler local rule.

- **Circuit Breakers**: To avoid spamming retries on persistent failures, implement a circuit breaker (e.g. with the `pybreaker` library). For instance:

  ```
  from pybreaker import CircuitBreaker
  cb = CircuitBreaker(fail_max=3, reset_timeout=60)

  @cb
  def safe_query():
      return query_gemini(user_prompt)
  try:
      answer = safe_query()
  except CircuitBreaker.Error:
      toast("Clawdbot", "Service is still down. Please try again later.")

  edge_tts.speak("It seems the AI service is still unavailable. I'll keep
  trying.")
  ```

  This opens the circuit after 3 failures (pausing new calls for 60s) [7].

- **Logging**: Maintain structured logs of **all user commands, bot actions, and failures**. Use Python's `logging` module with JSON formatting so that each entry includes timestamps, user IDs, input text, and outcome status. Follow best practices: use *meaningful, precise messages* that explain "what happened and why" [8], and *structured (JSON) logs* to capture context like user/session ID [9]. For example:

  ```
  import logging
  logger = logging.getLogger("clawdbot")
  handler = logging.FileHandler("clawdbot.log", encoding="utf-8")
  handler.setFormatter(logging.Formatter('%(asctime)s %(levelname)s %
  (message)s'))
  logger.addHandler(handler)
  logger.setLevel(logging.INFO)
  ```

```
def log_action(user, action, status):
    logger.info(f"user={user.id} action={action} status={status}")
```

Log every Gemini request and response summary, any exceptions, and final results. These logs enable post-mortem debugging and transparency.

- **Transparent Explanations**: After taking an action (e.g. scheduling a task, setting a reminder, asking a question), have Clawdbot briefly explain its reasoning. For instance: *"I suggested this task order because you seemed more focused in the morning. Does this work?"* Research shows that *"transparent explanations and accountability are a prerequisite for trust"* in AI systems [10]. You might implement an `/explain` endpoint in FastAPI that returns the rationale for the last action, pulling from the log or a simple rule-based summary. For example:

```
user_prompt = "Reschedule my meetings"
# Bot processes and decides to reschedule to afternoon
explanation = "Your calendar was busy this morning, so I moved meetings to
after lunch."
# Send explanation to user
toast("Clawdbot Explanation", explanation)
edge_tts.speak(explanation)
```

Memory & Contextual Learning

Clawdbot should **remember and adapt** based on user patterns. Build a lightweight on-device "memory" (structured JSON or SQLite) that logs key events (journaling) and adjusts behavior over time. Key elements:

- **Behavior Journaling**: Record time-stamped events like "started working on X", "took a break", "completed task Y". For example, append to a JSON file or local DB:

```
{"timestamp": "2026-02-05T10:00", "type": "start_task", "task": "Write
report"}
{"timestamp": "2026-02-05T11:15", "type": "break", "reason": "coffee"}
{"timestamp": "2026-02-05T13:00", "type": "complete_task", "task": "Write
report"}
```

This log lets Clawdbot analyze *time patterns* (e.g. "you usually take breaks ~10:00AM") and *session lengths*.

- **Pattern Analysis**: Periodically (or on-demand), parse the journal to detect habits. For instance, use simple statistics or heuristics:

```
import pandas as pd
log = pd.read_json("clawdbot_memory.json")
# Example: find hours when the user is most/least active
times = log[log.type=="start_task"].timestamp.dt.hour
peak_hour = times.mode()[0]
```

If the user often loses focus after e.g. 90 minutes, the bot could suggest a break. If procrastination is detected (long idle between tasks), Clawdbot might proactively ask, "It looks like your attention drifted around 3pm yesterday. Want to set a timer to refocus?"

- **Structured State Object**: Maintain a "memory" state (user profile, preferences, ongoing goals) locally [11]. For example, track known user facts ("You prefer Pomodoro technique", "You hate harsh reminders") and update them when revealed. Use a persistent state dictionary or JSON file, updated after each session. For instance:

```
state = {
    "work_pref": {"pomodoro": True},
    "break_alerts": "gentle",
    "goals": ["Exercise daily", "Finish report by Friday"]
}
```

Inject this state into prompts or use it to tweak dialogue. OpenAI's guidance emphasizes a *"local-first memory store"* for personalization [11] – we can adopt a simple homegrown version.

- **Adaptive Persona & Support**: Based on observed data, Clawdbot can adapt its tone and support level. E.g. if a user often misses deadlines, Clawdbot might become more proactive: *"I noticed tasks slipped last week. Would you like me to audit your schedule or help set reminders?"* Example code trigger:

```
# If user repeatedly logs incomplete tasks with same tag
if repeats_found:
    toast("Clawdbot Suggestion", "I see you're repeating the same task
daily. Want a habit tracker for it?")
    edge_tts.speak("It looks like you do that every day. Should I help you
build a habit tracker for it?")
```

This follows the idea of personalized nudges: e.g. "Want me to run a schedule audit?" or "I'm noticing distraction cycles – need a quick motivation?"

- **Support for Goals/Habits**: Let Clawdbot take initiative on goals. For example, after noticing the user often writes goals in the journal, Clawdbot can suggest an AI-powered habit tracker or daily planner. You might code a check:

```
if "finish report" in state["goals"] and today_is("Friday"):
    notify_user("Clawdbot",
"Final report was due today. Need help wrapping it up?")
    edge_tts.speak("Friendly reminder: your report is due. Ready to finish
strong?")
```

Across all memory tasks, keep processing on-device (local Python storage and computation). Avoid cloud for privacy and speed. As OpenAI notes, personalization means "maintaining structured state" locally and injecting only relevant parts into context [12]. By consolidating logs into meaningful insights, Clawdbot becomes a **persistent collaborator** rather than a stateless chatbot [12]. Over time, analyze accumulated memory to refine suggestions: e.g. if user becomes more productive at certain hours, proactively re-shuffle future tasks to those times. Always frame suggestions in a supportive tone, respecting the user's ADHD profile (e.g. allow easy dismissal of reminders, celebrate small wins).

Sources: We use the win11toast API for Windows notifications [1], retry/backoff libraries for Gemini resilience [5] [6] [7], structured logging best practices [13], trust research on explainability [10], and agent personalization patterns (state-based memory) [11] [12] to inform this design. Each code pattern and persona strategy reflects these principles and ensures Clawdbot is proactive, reliable, and ADHD-friendly.

---

[1] [3] GitHub - GitHub30/win11toast: Toast notifications for Windows 10 and 11 based on WinRT
https://github.com/GitHub30/win11toast

[2] ADHD AI assistant : r/ADHD_Programmers
https://www.reddit.com/r/ADHD_Programmers/comments/12rkx1j/adhd_ai_assistant/

[4] GitHub - rany2/edge-tts: Use Microsoft Edge's online text-to-speech service from Python WITHOUT needing Microsoft Edge or Windows or an API key
https://github.com/rany2/edge-tts

[5] How to Implement Retry Logic in the New Python SDK? - Gemini API - Google AI Developers Forum
https://discuss.ai.google.dev/t/how-to-implement-retry-logic-in-the-new-python-sdk/83052

[6] [7] Resilient APIs: Retry Logic, Circuit Breakers, and Fallback Mechanisms | by Fahim Ahmed | Medium
https://medium.com/@fahimad/resilient-apis-retry-logic-circuit-breakers-and-fallback-mechanisms-cfd37f523f43

[8] [9] [13] 10 Best Practices When Logging in Python | Rollbar
https://rollbar.com/blog/10-best-practices-when-logging-in-python/

[10] Trust in AI: progress, challenges, and future directions | Humanities and Social Sciences Communications
https://www.nature.com/articles/s41599-024-04044-8?error=cookies_not_supported&code=3eeefceb-572e-4c8a-a170-584dcbe54cbe

[11] [12] Context Engineering for Personalization - State Management with Long-Term Memory Notes using OpenAI Agents SDK
https://cookbook.openai.com/examples/agents_sdk/context_personalization