Technical Architecture Document: Discord Accountability Bot

1. Tech Stack

- **Language:** Python 3.11+
- **Discord Library:** `discord.py` (specifically utilizing `app_commands` for Slash Commands and `discord.ui` for Buttons).
- **Database:** SQLite (via `aiosqlite` for asynchronous, non-blocking database calls).
- **Task Scheduling:** `discord.ext.tasks` (built into discord.py) for the daily cron jobs (Wall of Shame, score deductions, board refreshes).
- **Containerization:** Docker & Docker Compose.

2. Database Schema (SQLite)

The database should consist of two primary tables to track users and their associated tasks.

Table: `users`

Tracks the global state of the user in the server.

- `discord_id` (TEXT, Primary Key) - The user's unique Discord ID.
- `score` (INTEGER, Default: 0) - The user's current point total.
- `private_channel_id` (TEXT) - The ID of the `#username-tasks` channel created for them.

Table: `tasks`

Tracks individual to-do items.

- `id` (INTEGER, Primary Key, Auto-Increment) - Unique task ID.
- `discord_id` (TEXT, Foreign Key -> users.discord_id) - Owner of the task.
- `description` (TEXT) - The actual task text.
- `status` (TEXT) - 'pending', 'completed', 'overdue'.
- `message_id` (TEXT) - The Discord ID of the specific message sent to the private channel (used to edit the message when a button is clicked).
- `due_date` (DATETIME) - When the task is expected to be finished.
- `recurrence` (TEXT) - 'none', 'daily', 'weekly'.
- `created_at` (DATETIME) - Timestamp of creation.

3. Core Asynchronous Loops (`discord.ext.tasks`)

The bot needs background loops running alongside the main event listener:

1. **The Overdue Checker (Runs Hourly):** Scans the `tasks` table for `status='pending'` where `due_date` < Current Time. Updates status to 'overdue', updates the Private Channel embed to Red, and deducts points.
2. **The Wall of Shame (Runs Daily at 9 PM):** Queries users with 'pending' or 'overdue' tasks and sends the aggregate ping to `#the-meat-grinder` .
3. **The Daily Reset (Runs Daily at Midnight):** Resets `status='completed'` recurring tasks back to 'pending', updates their `due_date` based on the recurrence interval, and generates new Inbox messages. Removes non-recurring completed tasks from the Accountability Board.

4. Docker Environment (Synology NAS Setup)

Because this is running on a Synology NAS, state persistence is critical. The SQLite database file must be stored on a mounted volume so it isn't wiped when the container updates.

Required `Dockerfile` instructions for AI:

- Base image: `python:3.11-slim`
- Set timezone (e.g., `ENV TZ="America/New_York"`) so the cron jobs run at the right time.
- Install requirements (`discord.py` , `aiosqlite` , `python-dotenv`).
- Set `CMD ["python", "bot.py"]` .

Required `docker-compose.yml` structure for AI:

- Define the service `accountability-bot` .
- Map a volume for the database: `- ./data:/app/data` (Ensure the Python script points the SQLite connection to `/app/data/database.db`).
- Pass the Discord Token via an environment variable file (`env_file: .env`).
- Set `restart: unless-stopped` so the bot survives NAS reboots.

5. Security & Permissions

- The bot requires the `bot` and `applications.commands` scopes in the Discord Developer Portal.
- Required Intents: `Message Content Intent` (to read basic commands if needed) and `Server Members Intent` (to create private channels and assign permissions).

- When the bot creates `#username-tasks`, it must set Discord channel overrides so that `@everyone` is denied `read_messages`, and only the specific `User` and the `Bot` are