Gerne! Hier ist ein **cleanes, detailliertes Fachkonzept** für dein Terminal-Programm mit klarer Trennung von Steuerung, Eingabe und Funktion – **ohne nano**, mit Fokus auf `argparse` und `prompt_toolkit`. Es ist wie eine technische Blaupause für dein CLI-Projekt und orientiert sich an **echten professionellen Tools**.

🧩 Fachkonzept: CLI-Programm mit `argparse` + `prompt_toolkit`

🔷 Ziel des Programms

Ein reines Terminal-Programm für Linux, das dauerhaft E-Mails überprüft, bei neuen E-Mails automatisiert antwortet, dabei Bash-Skripte ausführt – **alles gesteuert über das Terminal**, ohne GUI.

📁 Projektstruktur

```
projekt/
│
├── main.py                 # Einstiegspunkt mit argparse
├── config.py               # Lesen und Schreiben von Konfiguration
├── modes/                  # Interaktive Modi
│   ├── __init__.py
│   ├── setup_mode.py       # Interaktive Konfiguration
│   └── run_mode.py         # Dauerhafte Ausführung
├── mail/                   # E-Mail-Logik
│   ├── check.py            # Abruf neuer E-Mails
│   └── reply.py            # Antwort erzeugen
└── settings.json           # Konfigurationsdatei (wird interaktiv beschrieben)
```

🧭 Steuerung mit `argparse`

🔹 Aufgabe

`main.py` entscheidet **per Argument**, **welcher Programm-Modus gestartet wird**.

🔹 Beispielaufrufe

```
python3 main.py --mode=setup      # Konfiguration interaktiv durchführen
python3 main.py --mode=run        # Programm dauerhaft ausführen
```

🔹 Argumentdefinition

- `--mode`: Pflichtargument
    - `setup`: Interaktive Eingabe von Zugangsdaten, Antworttext usw.

- `run`: Programm beginnt dauerhaft zu laufen (z. B. E-Mail abrufen)

📥 Interaktive Eingabe mit `prompt_toolkit`

🔹 Warum?

Für User ohne Skript-Erfahrung: einfach, sicher, fehlertolerant. Ermöglicht:

- Passwortmaskierung (`is_password=True`)
- Autovervollständigung (z. B. für Domains)
- Eingabevalidierung
- Benutzerfreundliche Menüs

🔹 Beispielablauf `setup_mode`

```
📧 E-Mail-Konfiguration

1. E-Mail-Adresse eingeben
2. Passwort eingeben
3. Antworttext festlegen
4. Zeitintervall (Sekunden) setzen
5. Speichern und Testen
```

> Auswahl z. B. mit `prompt_toolkit.shortcuts.radiolist_dialog`
> Eingaben z. B. mit `prompt()`, validiert und farblich formatiert

🔹 Konfiguration wird gespeichert in:

```
{
  "email": "user@example.com",
  "password": "geheim123",
  "reply_text": "Danke für Ihre Nachricht!",
  "interval": 30
}
```

⚙️ Lauf-Modus `run`

🔹 Aufgabe

- Endlosschleife prüft im Hintergrund E-Mails (`while True`)
- Falls neue E-Mail ankommt:
  - Inhalte analysieren
  - Antwort erzeugen
  - ggf. Bash-Skript ausführen

🔹 Technischer Ablauf

```
with ThreadPoolExecutor() as executor:
    executor.submit(check_loop)     # Prüft Mails im Intervall
    executor.submit(script_loop)    # Führt Skripte unabhängig aus
```

- Kein Prozess blockiert den anderen
- Das Programm bleibt reaktiv
- Optionaler Stopp über Tastenkombi oder `SignalHandler`

📌 Funktionen der CLI – Übersicht

Funktion	Umsetzung
Moduswahl	`argparse` (`--mode`)
Interaktive Konfiguration	`prompt_toolkit`
Passwort sicher eingeben	`is_password=True`
Validierung von Eingaben	`prompt_toolkit.validation`
Bash-Skripte ausführen	`subprocess.run(...)`
E-Mail prüfen & antworten	`mail/check.py`, `mail/reply.py`
Parallele Ausführung	`concurrent.futures.ThreadPoolExecutor`
Konfiguration speichern	JSON-Datei `settings.json`

🧠 Beenden eines Prozesses (z. B. nur Script-Executor stoppen)

- Prozesse laufen in Threads
- Ein `threading.Event()`-Objekt kann benutzt werden, um bestimmte Prozesse kontrolliert zu beenden
- Beispiel:

```
stop_event = threading.Event()

def script_loop():
    while not stop_event.is_set():
        # führe Skript aus
        ...

# Beenden von außen:
stop_event.set()
```

🧠 Optional: Erweiterungsideen

Idee	Nutzen
`--mode=test`	Testet Mail-Einstellungen

Idee	Nutzen
Logging mit `logging`-Modul	Diagnose, Debugging
Lokale Mail-Vorschau	Zeigt die Antwort vor dem Senden
`argparse` mit `--set` später	Für Power-User ohne Interaktion

✅ Fazit

Dein Programm folgt damit einem **klar strukturierten, modularen, bewährten Konzept**, wie es auch bei komplexen CLI-Tools eingesetzt wird:

- **Einfache Steuerung durch `argparse`**
- **Benutzerfreundliche Eingabe durch `prompt_toolkit`**
- **Modular, testbar, erweiterbar**

Wenn du willst, können wir jetzt gemeinsam:

- ein Beispiel `setup_mode.py` planen,
- oder die `ThreadPoolExecutor`-Struktur für den `run_mode` aufbauen,
- oder den `prompt_toolkit`-Dialog für Eingaben entwerfen.

Was wäre dein nächster Schritt?