Deployment and Future Notes

- **CORS Configuration (Security)**
  In development, CORS is set to allow all origins. For production, you must restrict it to the frontend domain:

```
allow_origins = ["https://your-frontend-domain.com"]
```

  Allowing all origins (`"*"`) can introduce security vulnerabilities and should not be used in production.

- **Frontend-Supplied Location**
  The current system uses a hardcoded location for weather-related queries:

```
location = {"latitude": 29.9866, "longitude": 31.4406}
```

  This should be replaced with dynamic location data passed from the frontend to reflect the user's actual location.

- **Logging (Best Practice)**
  Replace all `print()` statements with Python's `logging` module. This allows better control over log levels and is more appropriate for production systems. Example:

```
import logging
logger = logging.getLogger(__name__)
logger.info("Task completed.")
```

- **Deployment Command (Production)**
  When moving to a production environment, use Gunicorn with Uvicorn workers:

```
gunicorn -k uvicorn.workers.UvicornWorker app.main:app --bind
    ↪ 0.0.0.0:8000
```

  This setup supports concurrency, better performance, and cleaner shutdowns.

- **Concurrency and Shared Model Queue**
  The system uses shared instances for STT (Whisper) and TTS models to avoid repeated loading overhead. To prevent race conditions:
  - Requests should be queued to ensure only one is handled at a time by the model.
  - This can be implemented using Python's `queue.Queue()` or an async queue like `asyncio.Queue()`.
  - Audio files should be named according to the userID. ex: temp(userID)

  This approach is essential to avoid memory overload and cross-user interference when multiple users access the service concurrently.

- **Mail Tool Extension (Optional)**
  The system already supports:
  - Sending emails (via SMTP)
  - Fetching unread emails (via IMAP)

  Optional features to consider:
  - HTML support in outgoing emails (currently plain text only)
  - Asking for user confirmation before sending emails
  - Future support for replying to or forwarding received messages

- **Registry Pattern for Tools (Important)**
  Tool function registration is centralized using a registry pattern. This improves scalability, allows clean decoupling, and helps in tool lookup during tool calling. Example implementation:

```
TOOL_REGISTRY = {}

def register_tool(name):
    def wrapper(fn):
        TOOL_REGISTRY[name] = fn
        return fn
    return wrapper
```

  Each tool function is decorated with `@register_tool("tool_name")` for auto-registration.

- **Error Return Type Consistency**
  Ensure all functions return a consistent data type. For example, returning `dict` on success and a `str` on error causes issues when consuming the function. Instead, always return a `dict`, even in error cases:

```
return {"error": "Invalid input"}
```

  This helps with safe parsing and avoids unexpected exceptions.