

ExecAI

An Intelligent Executive Assistant for Calendar and Email Management

Senior Design Capstone Project

Knight Foundation School of Computing and Information Sciences

Florida International University

Team Members

Rachelle Falcon

Zack Kohlman

Johana Huertas

Faculty Advisor: Masoud Sadjadi

Spring 2026

1. Executive Summary

ExecAI is an intelligent executive assistant that interprets natural language requests and executes common coordination tasks scheduling meetings, drafting emails, checking availability, and replying to messages through real integrations with Google Workspace and Microsoft 365. Knowledge workers spend a significant portion of their day on repetitive coordination tasks that interrupt focused work and rarely benefit from deep human judgment. ExecAI replaces that friction with a single conversational interface that interprets a user's intent and acts on it across calendar and email services.

The system uses a hybrid intent parser combining an LLM (OpenAI as primary, Groq as fallback) with rule-based extractors, an orchestration layer that makes conflict-aware decisions, and a set of integration services wrapping Google Calendar, Gmail, and Microsoft Graph behind a uniform interface. When LLM services are unavailable, the system gracefully falls back to rule-based parsing and template-driven drafts, ensuring the assistant always produces a usable response.

The delivered system supports eleven distinct intents from listing events to combined workflows like "reply to this email and schedule the meeting" and includes inline draft editing, contextual reply generation, conflict detection with alternative slot suggestions, and in-session follow-up commands like "make it shorter" or "more formal." The final system runs end-to-end with real OAuth-connected accounts and demonstrates how natural language interfaces can unify multiple productivity workflows within a single assistant architecture.

2. Problem & Requirements (O1)

2.1 Context and Stakeholders

Modern knowledge workers manage their schedule and inbox through rigid menu-driven interfaces that require manually navigating between calendar apps, email clients, and contact lists. Every routine action scheduling a meeting, drafting a routine email, finding a free slot involves multiple clicks across multiple tools. ExecAI's stakeholders are individual professionals, students, and researchers who already use Google Workspace or Microsoft 365 and want to reduce the coordination overhead of their day.

2.2 Personas

- The Researcher: spends most of the day in focused work and loses flow every time they have to schedule a meeting or respond to a routine request. Wants to dispatch coordination tasks in a single natural language command.
- The Student: juggles classes, study groups, and part-time work. Wants to quickly check the week ahead, respond to group emails, and block study time without opening three different apps.
- The Small-Team Lead: coordinates with a handful of collaborators. Wants to draft follow-up emails and book meetings in one flow, without manually retyping dates and names.

2.3 Functional Requirements

- Accept free-form natural language input through a chat interface.
- Classify user input into one of the supported intents with high accuracy.
- Extract structured entities: timeframes, times, durations, attendees, topics, tone.
- List upcoming calendar events over a requested window.
- Create calendar events with title, time, duration, description, and attendees.
- Detect scheduling conflicts and suggest alternative time slots automatically.
- Find available meeting slots when no specific time is requested.
- List and read inbox emails.
- Draft, reply to, edit, and send emails through the UI.
- Generate contextual replies using the original email content.
- Support combined workflows: reply-and-schedule, draft-and-schedule.
- Revise drafts via follow-up instructions ("make it shorter," "more formal").
- Resolve contact names to email addresses using recent email history.

2.4 Non-Functional Requirements

- Reliability: must never crash on unexpected input or API failure.
- Graceful degradation: must produce a usable response even when LLMs are unavailable.
- Response latency: typical end-to-end responses under 6 seconds for single-step actions.

- Transparency: the system's reasoning (intent, entities, decision) must be inspectable.
- Extensibility: new intents and providers must be addable without rewriting the orchestrator.
- Security: OAuth tokens must be stored locally and never committed to source control.

2.5 Constraints

- Single-user, local deployment only no cloud hosting or multi-user authentication.
- OAuth provider support limited to Google Workspace and Microsoft 365.
- Microsoft Graph integration is blocked in the FIU tenant and only works with personal accounts.
- LLM calls depend on external API availability and rate limits.
- No persistent storage across sessions — state is lost when the app restarts.

2.6 Success Criteria

The project is considered successful if a first-time user can connect a Google account, issue a natural language request such as "draft an email to Sarah about the proposal and create a meeting tomorrow at 2pm," and see both a real Gmail draft and a real calendar event created without ever opening Gmail or Google Calendar directly. Conflict detection must work; AI drafts must be contextual, and the system must fall back gracefully when any dependency fails.

2.7 Risks

- OAuth setup complexity: mitigated by building mocks first and migrating later.
- LLM rate limits or outages: mitigated by three-tier fallback strategy.
- Institutional tenant restrictions: mitigated by supporting personal Microsoft accounts.
- Ambiguous natural language input: mitigated by needs_clarification responses with helpful examples.

2.8 Prioritized Backlog

The following items were the top-priority requirements delivered during the project timeline:

#	Item	Status
1	Streamlit chat interface and FastAPI backend skeleton	Delivered
2	Rule-based intent parser and entity extractors	Delivered
3	LLM-based intent parser with fallback	Delivered
4	Google OAuth flow and token management	Delivered
5	Google Calendar: list, create, delete events	Delivered
6	Availability engine with conflict detection	Delivered
7	Alternative slot suggestion on conflict	Delivered
8	Gmail: list, read, draft, reply	Delivered

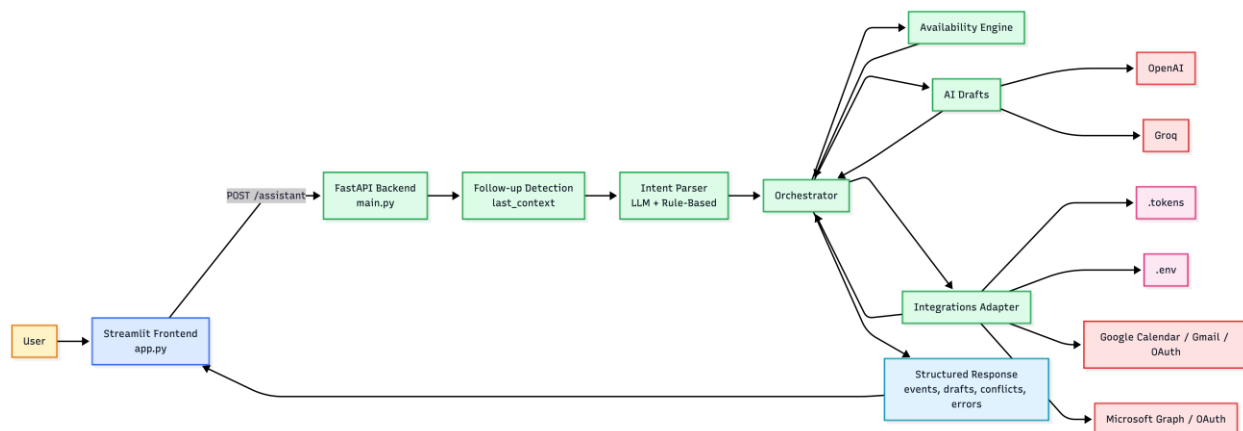
#	Item	Status
9	Send email from UI with inline editing	Delivered
10	Microsoft Graph integration	Delivered (personal accts)
11	Combined workflows (reply+schedule, draft+schedule)	Delivered
12	Contextual AI reply generation	Delivered
13	Follow-up revision commands	Delivered
14	Contact name resolution from recent contacts	Delivered
15	Debug panel for intent and decision transparency	Delivered

3. System Overview & Architecture (O2)

3.1 Architecture Overview

ExecAI follows a three-tier architecture with a clear separation between the conversational interface, the assistant's reasoning pipeline, and the external service integrations. The frontend is a Streamlit application that handles chat rendering, OAuth connection flows, and result display. The backend is a FastAPI service that exposes a single primary endpoint (/assistant) which routes user requests through four pipeline stages: intent parsing, orchestration, execution, and response formatting.

The core orchestration logic is deliberately independent of any specific LLM or integration. Rule-based parsing, AI parsing, and template-based draft generation all implement the same interface, and the orchestrator falls back through them in order of quality. This makes ExecAI resilient to service outages and allows it to function even when no paid AI provider is configured.



3.2 Pipeline Stages

- Intent Parser (intent.py): analyzes raw user text and classifies it into one of eleven supported intents, then extracts structured entities.
- Orchestrator (orchestrator.py): examines the parsed intent and decides which action to execute; checks calendar conflicts and generates alternatives for scheduling actions.
- Availability Engine (availability.py): computes free/busy windows, finds open meeting slots within working hours, and detects scheduling conflicts.
- AI Drafts (ai_drafts.py): generates, revises, and replies to email content using OpenAI as the primary provider, Groq as a fast fallback, and rule-based templates as a last resort.
- Integrations (integrations.py): wraps Google Calendar, Gmail, and Microsoft Graph APIs behind uniform service functions and handles OAuth token lifecycle.

3.3 Supported Intents

Intent	Description
list_events	Show upcoming calendar events over a requested window.
create_event	Create a calendar event with title, time, duration, and attendees.
meeting_scheduling	Find open slots across a timeframe when no specific time is given.
list_emails	Retrieve recent inbox messages.
read_email	Open a specific email by reference (latest, first, indexed).
email_drafting	Generate a new email draft with AI or templates.
reply_email	Generate a contextual reply to an existing email.
reply_and_create_event	Combined workflow: reply to an email and create a calendar event.
draft_email_and_create_event	Combined workflow: draft a new email and create a calendar event.
revise_draft	Revise an existing draft based on a natural language instruction.
unknown	Safe fallback when the user's intent cannot be classified.

3.4 Technologies and Services

Component	Technology
Backend Framework	FastAPI (Python 3.11+)
Frontend Framework	Streamlit
Data Validation	Pydantic
Primary LLM	OpenAI API (gpt-4o-mini or similar)
Fallback LLM	Groq API (llama-3.3-70b-versatile)
Calendar Providers	Google Calendar API v3, Microsoft Graph API v1.0
Email Providers	Gmail API v1, Microsoft Graph Mail API
Authentication	OAuth 2.0 (Google, Microsoft)
HTTP Client	requests
Environment Config	python-dotenv

3.5 API Surface

The backend exposes the following REST endpoints:

- GET /health — health check
- POST /parse-intent — intent parsing only, no execution
- POST /assistant — full pipeline: parse, orchestrate, execute
- GET /integrations/status — check connected providers

- GET /integrations/{provider}/auth-url — begin OAuth flow
- GET /integrations/{provider}/callback — OAuth callback
- POST /integrations/{provider}/list-events — list calendar events
- POST /integrations/{provider}/create-event — create a calendar event
- DELETE /integrations/{provider}/delete-event/{event_id} — delete an event
- POST /integrations/{provider}/freebusy — query free/busy windows
- GET /integrations/{provider}/list-emails — list inbox messages
- POST /integrations/{provider}/create-draft — create an email draft
- POST /integrations/{provider}/create-reply-draft — create a reply draft
- POST /integrations/{provider}/send-email — send an email

3.6 Data and Storage

ExecAI stores minimal local state access and refresh tokens persisted as JSON files in backend/.tokens/, which is excluded from source control via .gitignore. Conversation history lives in Streamlit's st.session_state and is discarded when the app restarts. Environment variables (API keys, OAuth client secrets) are loaded from a local .env file that is also excluded from version control.

4. Discipline-Specific Depth (O6)

4.1 Algorithms and Data Structures

Available Slot Detection

The availability engine slot-finding algorithm generates candidate time windows within the user's working hours and filters out any window that overlaps an existing busy block. The approach runs in $O(n \cdot m)$ time, where n is the number of candidate slots generated for the requested timeframe (bounded by the length of the search window divided by the slot increment, typically around 16 per day for a 30-minute increment across an 8-hour workday) and m is the number of busy blocks returned by the FreeBusy API.

The algorithm operates in three phases: (1) generate candidate slots by iterating day-by-day, skipping weekends, and aligning slot boundaries to a 30-minute increment; (2) parse busy blocks into a list of (start, end) tuples sorted by start time; (3) for each candidate, check overlap against every busy block and append to the result list if no overlap is found, terminating early once MAX_SUGGESTIONS slots are found.

Since MAX_SUGGESTIONS is 3 and the search window is usually bounded to a single day or week, the algorithm runs in effectively constant time for typical inputs. A sorted interval tree would reduce the per-candidate check from $O(m)$ to $O(\log m)$, but the constant factors of the simple approach are lower for the small input sizes we observe in practice.

Intent Classification Fallback Chain

The intent parser uses a hybrid classification strategy. When an LLM API key is available, user text is sent to the LLM with a strict system prompt instructing it to return JSON with an intent and entity object. The LLM output is then normalized and enriched using rule-based extractors as safety nets. If the LLM call fails — for quota, auth, or network reasons — the system falls back silently to pure rule-based parsing.

Each rule-based classifier is an $O(k)$ string-matching operation, where k is the length of the input text. The classifier runs through a series of heuristic checks in priority order (compound intents first, then single-action intents, then catch-all fallbacks), returning as soon as a match is found. This ordering matters because compound intents like "reply to this email and schedule a meeting" must be detected before the simpler "reply to email" or "schedule a meeting" heuristics.

Contextual Follow-Up Detection

To support in-session follow-ups, the frontend attaches the previous decision and results to every assistant request as a last context object. The backend uses this context to distinguish true standalone intents from follow-up modifications such as "make it shorter" or "more formal." The detection is performed before the main intent parser runs: if the current input matches a revision pattern, AND last context contains a draft or reply; the request is routed to a revision handler instead of a fresh intent parse.

4.2 Architecture and Patterns

Exec AI uses several recognizable software patterns. The orchestrator follows the Chain of Responsibility pattern — each intent has a dedicated decision-building function selected based on the parsed intent type. The integration layer implements the Adapter pattern, wrapping two dissimilar provider APIs (Google's REST endpoints and Microsoft's Graph API) behind a unified service interface. The AI drafts module uses the Strategy pattern, with OpenAI, Groq, and template generators as interchangeable strategies selected at runtime based on configuration and availability.

The separation of concerns between intent parsing, orchestration, and execution made the codebase straightforward to modify as requirements evolved. When follow-up context handling was added late in the project, only the intent parser and main.py needed changes — the orchestrator and integration layers were untouched. When the draft revision feature was added, the change was limited to ai_drafts.py and the orchestrator.

4.3 Engineering Evidence

Observed end-to-end latency by operation class, measured during demo testing:

Operation	Typical Latency
List events / read email (single API call)	1–2 seconds
Create event (intent + conflict check + API call)	2–3 seconds
Draft email with Groq LLM	2–4 seconds
Draft email with OpenAI LLM	3–6 seconds
Combined workflow (reply + schedule)	4–8 seconds
OAuth token refresh overhead	+200–400 ms

LLM response time dominates the total latency for all AI-dependent operations. Groq's specialized inference hardware delivered noticeably faster responses than OpenAI's gpt-4o-mini at comparable quality for the short-form prompts ExecAI issues.

5. Implementation Notes (O2)

5.1 Repository Structure

```
execai/
├── backend/
│   ├── main.py          FastAPI entry point and routing
│   ├── intent.py        Hybrid intent parser
│   ├── orchestrator.py   Decision-making layer
│   ├── availability.py   Slot detection and conflict logic
│   ├── ai_drafts.py      Email draft generation
│   ├── integrations.py   Google and Microsoft API wrappers
│   └── .tokens/          OAuth token storage (not committed)
├── frontend/
│   └── app.py            Streamlit UI and render functions
├── docs/                Architecture diagrams and screenshots
├── .env                 Environment variables (not committed)
├── requirements.txt
└── README.md
```

5.2 Coding Conventions

- Each backend module has a single clear responsibility and is named after that responsibility.
- Service functions take a provider parameter and return normalized dictionaries; the orchestrator never writes provider-specific code.
- All external I/O is wrapped in try/except blocks that return structured error responses instead of propagating exceptions to the user.
- Pydantic models validate all request payloads at the API boundary.
- Render functions in app.py follow a uniform pattern: read the decision dictionary, dispatch to the correct renderer, display the result.

5.3 Notable Patterns and Tradeoffs

The most consequential design decision was the three-tier fallback strategy in ai_drafts.py. Each tier (OpenAI, Groq, templates) returns the same response shape, so the orchestrator is unaware of which tier was used. The tradeoff is that template-tier output is noticeably less fluent than LLM output, but the alternative — failing the entire request when the primary LLM is unavailable — would have been unacceptable for a demo-oriented project.

Another meaningful tradeoff was the choice to generate reply bodies in main.py rather than in the orchestrator. The orchestrator cannot fetch the original email content (that would cross the separation between orchestration and integration), so it initially passed only a generic template to the draft generator. We moved reply generation to the main handler, where the target email has already been resolved, so the AI has access to the original subject, sender, and body. This fix significantly improved reply quality at the cost of a small amount of code duplication between the reply and reply-and-create-event branches.

6. Testing & Evaluation (O3)

6.1 Testing Strategy

ExecAI was validated through manual end-to-end scenario testing and targeted unit checks on the intent parser and availability engine. Because the system depends on live OAuth connections to Google and Microsoft, automated integration tests were not practical within the project timeline. Instead, the team maintained a scenario checklist of representative prompts exercising each intent and ran through it after every significant change.

6.2 Scenario Checklist

- List calendar events for today, tomorrow, this week, and next week.
- Create an event with an explicit time, duration, and attendee list.
- Create an event that conflicts with an existing event and verify alternatives are returned.
- Find meeting slots when no specific time is given.
- List the last five emails and read the most recent one.
- Draft a new email with a recipient, topic, and tone.
- Reply to the latest email with AI-generated contextual content.
- Combined workflow: reply to an email and create a follow-up meeting.
- Combined workflow: draft a new email and create a meeting.
- Follow-up revision: after a draft, ask to make it shorter or more formal.
- Edit a draft inline and send it; verify delivery in the provider's sent folder.
- Delete an event and verify it is removed from the calendar.

6.3 Intent Parser Evaluation

The intent parser was evaluated against a set of approximately 40 representative prompts covering all eleven intents, including ambiguous and compound phrasings. The rule-based path alone correctly classified the majority of straightforward prompts, while the LLM path improved accuracy on ambiguous, conversational, and compound requests. Both paths agreed on the canonical test set; divergences occurred mainly on phrasings that mixed multiple intents in a single sentence, which the LLM handled more consistently.

6.4 Availability Engine Evaluation

The availability engine was tested against synthetic busy-block scenarios covering edge cases: back-to-back meetings, gaps smaller than the requested duration, meetings crossing working-hour boundaries, and weekends (which the engine skips). The slot generator correctly returned up to three non-overlapping options for each scenario and handled timezone normalization between the default America/New_York zone and the API responses.

6.5 Failure Mode Testing

- OAuth token expired — the system automatically refreshes using the stored refresh token.
- OAuth token revoked — the system returns a structured error prompting the user to reconnect.
- LLM API unavailable — the system falls back to rule-based parsing and template drafts silently.
- Provider API returning unexpected data — the system catches exceptions and returns a user-facing message rather than crashing.
- Missing required entities (event without title) — the system returns a needs_clarification response with a helpful example.

6.6 KPIs and Metrics

- Intent classification accuracy on the canonical 40-prompt test set: approximately 95% with the LLM path, approximately 85% with the rule-based path alone.
- End-to-end latency for single-step actions: 1–3 seconds typical.
- End-to-end latency for LLM-dependent actions: 2–8 seconds typical.
- Scenario checklist pass rate for the final demo build: 12 of 12 scenarios passing.

7. Security, Privacy, Accessibility & Compliance (O5)

7.1 Security

ExecAI handles sensitive OAuth credentials and personal email content, so security was considered throughout the project. OAuth access and refresh tokens are stored locally in `backend/.tokens/`, which is excluded from source control via `.gitignore`. API client secrets and LLM API keys are loaded from a local `.env` file, also excluded from version control. Because the system runs only on localhost and never exposes its backend to the public internet, there is no network-accessible attack surface during normal use.

OAuth tokens are automatically refreshed when they expire, and expired refresh tokens trigger a structured error response instructing the user to reconnect. The system never logs token contents, and debug panels only expose non-sensitive metadata (intent, entities, decision) rather than raw API responses or credentials.

7.2 Privacy

ExecAI operates entirely on the user's own account and does not share data with any party other than the configured LLM provider. When an LLM is used for intent parsing or draft generation, the user's input text and relevant context are sent to OpenAI or Groq as part of the prompt. Users running the system should be aware that LLM providers may retain prompts according to their own data retention policies. Users who do not want any data sent externally can run the system without configuring an LLM API key; the rule-based parser and template drafts produce fully local results.

The OAuth scopes requested are the minimum necessary for the supported operations: calendar read/write and Gmail compose/readonly for Google, and equivalent `Calendars.ReadWrite` and `Mail.ReadWrite` scopes for Microsoft. The system does not request broader permissions such as Contacts, Drive, or Admin.

7.3 Accessibility

The Streamlit frontend uses standard HTML form elements (text areas, buttons, select boxes) that are keyboard-navigable and screen-reader-compatible out of the box. Text areas have descriptive labels and color contrast follows Streamlit's default theme. A known accessibility gap is that the frontend does not provide an explicit high-contrast mode or adjustable font sizes beyond what the browser itself supports.

7.4 Ethical and Societal Considerations

Automating email drafting raises a legitimate concern about AI-generated content being sent on a user's behalf. ExecAI addresses this by requiring explicit user review and a Send click for every outgoing email the system never sends anything automatically, and drafts are always editable before dispatch. This design deliberately keeps the human in the loop and prevents the assistant from operating in a fully autonomous mode that could misrepresent the user's intent.

The team also considered the risk of LLM hallucination when generating drafts. Because drafts are short and always presented for user review, the risk of hallucinated content being sent without awareness is low. However, users should still verify factual claims in generated emails before sending.

8. Deployment & Operations

8.1 Local Deployment

ExecAI is currently a local-only application. Deployment consists of cloning the repository, creating a Python virtual environment, installing dependencies from requirements.txt, configuring a local .env file with OAuth credentials, and running the backend and frontend in two separate terminal windows. Full setup instructions are provided in Appendix A.

8.2 Environment Variables

The system requires the following environment variables configured in a local .env file:

```
GOOGLE_CLIENT_ID=your_google_client_id
GOOGLE_CLIENT_SECRET=your_google_client_secret
GOOGLE_REDIRECT_URI=http://127.0.0.1:8000/integrations/google/callback
MICROSOFT_CLIENT_ID=your_microsoft_client_id
MICROSOFT_CLIENT_SECRET=your_microsoft_client_secret
MICROSOFT_TENANT_ID=common
MICROSOFT_REDIRECT_URI=http://localhost:8000/integrations/outlook/callback
OPENAI_API_KEY=your_openai_key_optional
GROQ_API_KEY=your_groq_key_optional
```

8.3 Running the System

Start the backend from the project root:

```
python -m uvicorn backend.main:app --reload
```

In a second terminal, start the frontend:

```
streamlit run frontend/app.py
```

The backend serves on <http://127.0.0.1:8000> and the frontend on <http://localhost:8501>. Open the frontend in a browser, connect a Google or Microsoft account from the sidebar, and begin issuing natural language requests.

8.4 Operational Notes

- OAuth tokens persist across restarts; users do not need to reconnect every session.
- Expired tokens are refreshed automatically; revoked tokens require reconnection.
- The system produces no log files by default; errors surface in the Streamlit UI and backend console.
- No database is required — all runtime state lives in memory or on the filesystem.

9. Limitations & Future Work

9.1 Known Limitations

- Single-user, local-only deployment — no cloud hosting, multi-user authentication, or persistent sessions.
- No conversation memory across sessions — preferences, past requests, and scheduling habits are forgotten when the app restarts.
- Limited multi-turn depth — the assistant handles simple follow-ups like draft revisions but cannot manage complex multi-step planning or reference earlier turns.
- Limited to two providers — only Google (Calendar + Gmail) and Microsoft (Outlook via Graph). No Apple Calendar, Yahoo, Zoom, or other platforms.
- No shared or team calendar support — can only access the authenticated user's own calendars.
- LLM dependency and cost — relies on OpenAI primary with Groq fallback. If both are unavailable, output quality drops significantly.

9.2 Prioritized Future Work

Cloud Deployment

Hosting the FastAPI backend and Streamlit frontend on AWS or GCP would enable multi-user access, persistent sessions, and production-grade reliability. This would require replacing local token storage with a proper secrets manager, implementing user-scoped OAuth flows, and switching the Google OAuth consent screen from testing to production mode.

Persistent Preferences & Cross-Session Memory

Adding persistent storage for user preferences — default meeting durations, preferred tones, frequent contacts, working hours — would let the assistant become more personalized over time rather than forgetting everything between sessions.

Multi-User Authentication and Team Calendars

Multiple authenticated users with role-based access and cross-calendar availability queries would enable team scheduling use cases. The current architecture would need a user-scoping layer added to the orchestrator and integration services.

Advanced NLU Improvements

Better entity extraction for ambiguous dates, relative timeframes, and complex compound requests would improve robustness on edge-case phrasings. Upgrading to a more capable LLM and fine-tuning on domain-specific examples would help.

Additional Provider Integrations

Expanding beyond Google and Microsoft to support Apple Calendar (via CalDAV), Zoom scheduling, Slack notifications, and other platforms would broaden the system's reach. The integration layer is already structured to accommodate new providers without changing the orchestrator.

Proactive Notifications and Reminders

Push notifications and follow-up reminders without requiring the user to initiate every interaction would make ExecAI feel more like a true assistant and less like a command-response tool.

10. References

The following external APIs, libraries, and documentation were referenced during the development of ExecAI:

- Google Calendar API Documentation. <https://developers.google.com/calendar/api/v3/reference>
- Gmail API Documentation. <https://developers.google.com/gmail/api/reference/rest>
- Microsoft Graph API Documentation. <https://learn.microsoft.com/en-us/graph/api/overview>
- OpenAI API Reference. <https://platform.openai.com/docs/api-reference>
- Groq API Documentation. <https://console.groq.com/docs>
- FastAPI Documentation. <https://fastapi.tiangolo.com>
- Streamlit Documentation. <https://docs.streamlit.io>
- Pydantic Documentation. <https://docs.pydantic.dev>
- RFC 6749 — The OAuth 2.0 Authorization Framework.
<https://datatracker.ietf.org/doc/html/rfc6749>

Appendices

Appendix A — Installation Guide

A.1 Prerequisites

- Python 3.11 or newer
- pip and venv
- A Google Cloud project with Calendar API and Gmail API enabled
- OAuth 2.0 Client ID (Desktop application type)
- Optional: a Microsoft 365 personal account for Outlook testing
- Optional: OpenAI or Groq API key for AI-assisted drafts

A.2 Installation Steps

1. Clone the repository:

```
git clone <repository_url>
cd execai
```

2. Create and activate a virtual environment:

```
python -m venv .venv
# On Windows:
.venv\Scripts\activate
# On macOS/Linux:
source .venv/bin/activate
```

3. Install dependencies:

```
pip install -r requirements.txt
```

4. Create a .env file in the project root with the environment variables listed in 8.2.

5. Start the backend:

```
python -m uvicorn backend.main:app --reload
```

6. In a second terminal, start the frontend:

```
streamlit run frontend/app.py
```

7. Open <http://localhost:8501> in a browser and click Connect Google in the sidebar.

Appendix B — User Manual

B.1 Connecting a Provider

Open the Streamlit interface and locate the Connect Google or Connect Outlook button in the sidebar. Clicking either button will open a new browser tab with the provider's OAuth consent screen. Grant the requested permissions, and the callback page will confirm the connection. Return to the ExecAI tab and the sidebar status indicator will show the connected account.

B.2 Common Commands

- "show my calendar for next week" — list upcoming events
- "create an event called Team Sync tomorrow at 2pm for 45 minutes" — create an event
- "create a meeting with sarah@example.com tomorrow at 11am" — create with attendee
- "find a time to meet tomorrow afternoon" — ask for open slots
- "show my latest emails" — list inbox
- "read my latest email" — open the most recent message
- "draft an email to sarah@example.com about the proposal" — generate a draft
- "reply to my latest email saying I agree" — generate a contextual reply
- "make it shorter" or "more formal" — revise the most recent draft
- "reply to my latest email saying I'm available tomorrow at 2pm and create the meeting" — combined workflow

B.3 Reviewing and Sending Drafts

When the system generates a draft, it appears in the chat with an editable text area and a Send button. Review the draft, make any edits directly in the text area, then click Send to dispatch the email through the connected provider. After sending, the draft becomes read-only and a success message replaces the Send button.

Appendix C — Admin and Operations Guide

- OAuth tokens are stored in backend/.tokens/ as JSON files. Deleting a token file forces a reconnection on the next request.
- The .env file controls which LLM provider is used. Setting only GROQ_API_KEY (and leaving OPENAI_API_KEY empty) will route all AI calls through Groq.
- Backend logs appear in the Uvicorn console; frontend errors appear in the Streamlit console and in the UI.
- To reset all state, stop both servers, delete backend/.tokens/, and restart. Session history in Streamlit is also cleared on restart.

Appendix D — API Reference

The full list of backend endpoints is documented in 3.5. FastAPI also exposes an interactive OpenAPI browser at <http://127.0.0.1:8000/docs> while the backend is running, which provides live request/response schemas and a built-in test interface for every endpoint.

Appendix E — Presentation Artifacts

The final public-facing project artifacts for ExecAI are listed below:

- Poster (36" × 48" horizontal): https://github.com/johanahuertas/execai-capstone/blob/main/docs/ExecAI_Capstone_Poster.pdf

- Intro Video: <https://youtu.be/RZGYTSZdksw?si=IzWcBVIPcgHV6ure>
- User Guide Video: https://youtu.be/fD1_hFzKGWo
- Installation & Maintenance Video: https://youtu.be/zoRNUa2P_Vk
- Shortcomings & Wishlist Video: <https://youtu.be/n0p7dXCgRqs>
- Final Team Presentation (PDF): https://github.com/johanahuertas/execai-capstone/blob/main/docs/ExecAI_Presentation.pdf

Appendix F — Scrum Evidence Index

Development was organized into weekly sprints with Sprint Planning, Daily Standups via group chat, Sprint Reviews, and Retrospectives. Evidence of each ceremony (meeting notes, backlog updates, burn-down charts) is maintained in the project's shared Google Drive folder and linked from the team repository.