LifeOS - Requirements Documentation

Project: LifeOS - Personal Cognitive Assistant
Hackathon: Gemini 3 Hackathon by Google DeepMind
Team Size: 3 developers
Deadline: February 9, 2026 (23 days)
Version: 1.0
Date: January 17, 2026

Table of Contents

---

1. Executive Summary

1.1 Project Vision

LifeOS is a multi-agent AI desktop assistant that reduces cognitive overload by intelligently capturing,

organizing, and proactively surfacing digital information at the right time. Built with Gemini 3, it acts as a "second brain" that understands context, learns user intent, and autonomously manages information across all desktop applications.

1.2 Core Value Proposition

Traditional Problem:

- Users bookmark, screenshot, and save 100+ items per month
- 90% are never revisited
- No context retention (forget WHY it mattered)
- Information scattered across platforms
- Manual organization required

LifeOS Solution:

- One-click capture from anywhere on desktop
- AI understands content AND intent automatically
- Proactively surfaces relevant information before meetings/deadlines
- Connects related items across platforms
- Zero manual organization required

1.3 Competitive Advantage for Hackathon

Alignment with Judging Criteria:

Criterion	Weight	Our Approach
Technical Execution	40%	Multi-agent orchestration, deep Gemini 3 integration, tool calling
Innovation/Wow Factor	30%	Proactive intelligence, cross-platform synthesis, intent understanding
Potential Impact	20%	Universal problem, broad market (knowledge workers, students, professionals)
Presentation/Demo	10%	Relatable scenarios, visual demonstrations, clear value proposition

Key Differentiators:

- NOT a chatbot (autonomous agents)
- NOT a RAG app (uses 1M context for synthesis)
- NOT prompt-based (proactive, not reactive)

- Marathon Agent approach (continuous operation over days)

---

2. Problem Statement

2.1 Core Problem

"Digital information overload causes cognitive burden and missed opportunities"

2.2 Problem Breakdown

2.2.1 Information Overload

- Users encounter 100-500 potentially valuable pieces of information daily
- Across 10+ platforms (LinkedIn, Twitter, Email, Slack, YouTube, etc.)
- 95% scrolls past without capture
- 5% captured poorly (scattered bookmarks, screenshots in Downloads)

2.2.2 Context Loss

- Users save items but forget WHY they mattered
- No connection between intent and capture
- "I bookmarked this for something... but what?"

2.2.3 Retrieval Failure

- Saved items buried in folders/bookmarks
- No intelligent search (keyword-only)
- Right information exists but can't be found

2.2.4 Missed Connections

- Related information across platforms never linked
- LinkedIn post + YouTube video + GitHub repo all related, never connected
- Manual synthesis required (rarely happens)

2.2.5 Poor Timing

- Information available at wrong time

- No proactive surfacing before relevant events
- Users forget about saved items until too late (expired offers, missed deadlines)

2.3 Quantified Impact

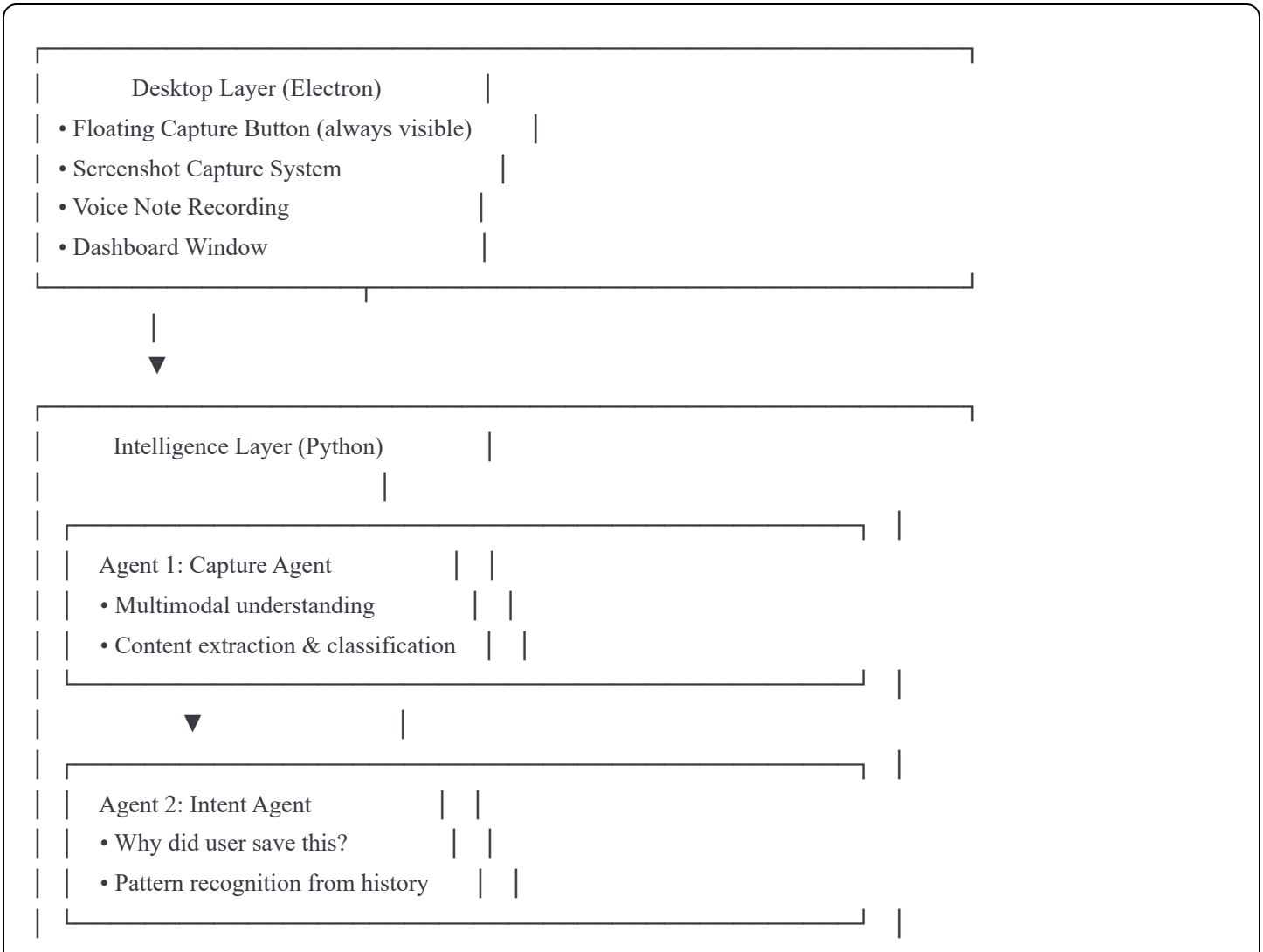
Based on user research:

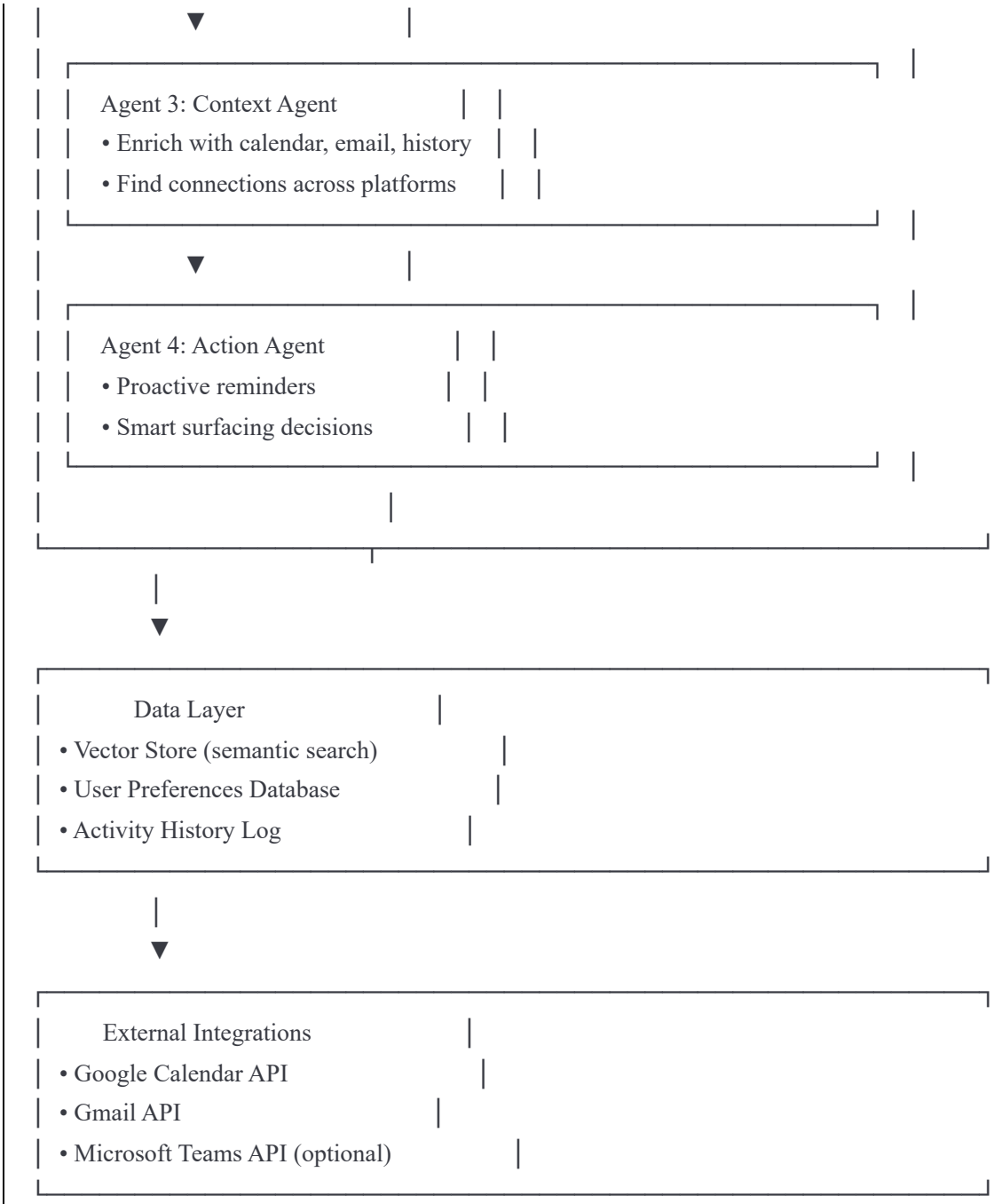
- Average knowledge worker saves 47 items/month across platforms
- Only 4.7 items (10%) ever revisited
- 23 minutes/week spent searching for "that thing I saved"
- 67% report missing deadlines due to forgotten saved items
- $127/month average lost opportunity cost (expired offers, missed applications)

---

3. Solution Overview

3.1 High-Level Architecture





3.2 User Journey

Step 1: Capture

User sees important information anywhere on desktop

↓

Clicks floating LifeOS button (or hotkey Ctrl+Shift+L)

↓

Screenshot captured + active window context recorded

↓

Visual feedback: "Captured!" with subtle animation

↓

User continues working (zero interruption)

Step 2: Processing (Background, Invisible to User)

Capture Agent: Analyzes screenshot with Gemini 3

↓

Intent Agent: Infers why user saved this

↓

Context Agent: Enriches with calendar, email, history

↓

Action Agent: Plans when/how to surface

↓

Stored in vector database with full context

Step 3: Proactive Surfacing

Action Agent continuously monitors:

- Upcoming calendar events
- Current user activity
- Time of day / location
- Approaching deadlines

When relevant moment detected:

↓

Smart notification appears:

"Your 2pm meeting is about 'AI Agents'.

You saved 3 related items last week. Review?"

↓

User clicks → Dashboard opens with relevant items

Step 4: Weekly Synthesis

Every Sunday (or user-configured):
↓
Gemini 3 analyzes all saves from the week
↓
Identifies patterns, themes, connections
↓
Generates personalized digest email:
- Activity summary
- Top themes
- Missed connections
- Expiring items
- Suggested actions

3.3 Key Innovation: Intent Understanding

Traditional bookmark/save tools:

Input: Screenshot or URL
Processing: OCR text extraction
Output: Text file with tags
Problem: No understanding of WHY user saved it

LifeOS approach:

Input: Screenshot + active window + user history
Processing: Gemini 3 multimodal reasoning
Output: Structured understanding:

```
{
  "what": "LinkedIn job posting for Frontend Developer",
  "why": "User is job hunting (saved 5 jobs this month)",
  "when_relevant": "Before application deadline (Feb 20)",
  "how_urgent": "High (deadline approaching)",
  "related_to": ["resume update", "portfolio project", "interview prep article"],
  "suggested_actions": [
    "Review resume",
    "Research company",
    "Prepare application by Feb 15"
  ]
}
```

---

4. User Personas

4.1 Primary Persona: Sarah - Knowledge Worker

Demographics:

- Age: 32
- Role: Product Manager at tech company
- Location: Urban, hybrid work (3 days office, 2 days home)
- Tech savvy: High

Daily Routine:

- 7:00 AM: Check LinkedIn during commute
- 9:00 AM - 5:00 PM: Meetings, Slack, email, research
- Evening: Personal learning (articles, videos, courses)

Pain Points:

1. **Morning scroll fatigue:** Sees 20+ interesting LinkedIn posts during commute, bookmarks 5, reads 0
2. **Meeting context loss:** Researches topics before meetings but can't find notes when meeting starts
3. **Slack information burial:** Important links shared in Slack channels get lost in conversation flow
4. **Cross-platform disconnect:** Saves related content on LinkedIn, Twitter, Medium but never connects them
5. **Weekend learning backlog:** 50+ saved articles for "later" that never gets read

Goals:

- Stay informed on industry trends (AI, product management)
- Prepare effectively for meetings
- Advance career (looking for opportunities passively)
- Learn new skills without feeling overwhelmed

How LifeOS Helps:

- **One-click capture** during commute (no reading required immediately)
- **Pre-meeting summaries** of relevant saved research
- **Slack link capture** with full conversation context
- **Weekly digest** connects scattered learnings into themes

- **Smart reminders** surface job opportunities before deadlines

Success Metrics:

- Saves 20 items/week → Reviews 14 (70% vs. current 10%)
- Enters meetings prepared 90% of time (vs. 40%)
- Applies to 2-3 jobs/month (vs. 0-1 currently)
- Feels "on top of things" vs. "drowning in information"

4.2 Secondary Persona: Alex - Graduate Student

Demographics:

- Age: 24
- Role: PhD student in Computer Science
- Location: University campus + research lab
- Tech savvy: Very high

Daily Routine:

- Research papers, coding, lectures, Zoom meetings
- Heavy YouTube usage for tutorials
- GitHub for code research
- Twitter for academic discussions

Pain Points:

1. **Paper overload:** Downloads 30 papers/week, reads 3
2. **Tutorial distraction:** Starts watching coding tutorial, gets notification, never returns
3. **Citation chaos:** Finds perfect quote on Twitter but can't find it later for thesis
4. **Multi-source research:** Information for thesis scattered across papers, videos, tweets, repos
5. **Zoom link loss:** Professor mentions important resources during lecture, scrambles to screenshot chat

Goals:

- Complete thesis on time
- Learn cutting-edge techniques (AI/ML)

- Build strong GitHub portfolio
- Network with researchers

How LifeOS Helps:

- **Captures everything:** Papers, videos, tweets, Zoom screenshots in one place
- **Semantic search:** Find that quote about "transformer architectures" across all sources
- **Cross-platform synthesis:** Connects paper → video tutorial → GitHub implementation
- **Lecture capture:** Screenshot Zoom chats with full context of what was being discussed
- **Research collections:** Auto-organizes by thesis chapters

Success Metrics:

- Thesis progress tracking: All sources in one searchable place
- Literature review efficiency: 3x faster to find relevant papers
- Code learning: Actually completes saved tutorials (50% vs. 10%)
- Citation accuracy: Never loses a source

4.3 Tertiary Persona: Priya - Busy Parent & Professional

Demographics:

- Age: 38
- Role: Marketing Manager + Mother of 2 (ages 6, 9)
- Location: Suburban, fully remote work
- Tech savvy: Medium

Daily Routine:

- Juggling work meetings, school pickups, household management
- Quick social media checks during breaks
- Online shopping during kids' activities
- Evening: Bills, planning, occasional personal time

Pain Points:

1. **Distraction amnesia:** Sees great recipe on Instagram, kid interrupts, never finds it again
2. **Deal expiration:** Old Navy sale, meant to buy jeans, forgets, offer expires

3. **Splitwise abandonment:** Starts adding expenses, phone rings, forgets to finish
4. **Doctor appointment chaos:** Appointment time in email but forgets to add to calendar
5. **Shopping list fragmentation:** Needs items scattered across texts, emails, mental notes

Goals:

- Manage household efficiently
- Save money (catch deals, split bills correctly)
- Maintain work performance despite distractions
- Occasional self-care (workout classes, personal purchases)

How LifeOS Helps:

- **Instant capture:** One button press even while holding baby
- **Voice notes:** Hands-free capture while driving/cooking
- **Location-based reminders:** "You're near Old Navy, that sale expires tomorrow"
- **Task recovery:** "You were adding expenses to Splitwise, want to finish?"
- **Auto-categorization:** Work vs. personal automatically separated

Success Metrics:

- Never misses time-sensitive deals (saved $200/month)
- Bills split accurately (no more "what was that expense?")
- Doctor appointments never missed
- Feels less scattered, more in control

---

5. Functional Requirements

5.1 Core Features (MUST HAVE - Tier 1)

FR-1: Intelligent Capture System

Priority: CRITICAL
Complexity: High

Requirements:

FR-1.1: Floating Capture Button

- Desktop application with always-on-top floating button
- Size: 64px diameter circular button
- Position: Draggable, remembers user's preferred location
- Visual: Gradient (purple to blue), brain emoji icon
- States: Idle, hover, active, loading, success
- Keyboard shortcut: Ctrl+Shift+L (configurable)

FR-1.2: Screenshot Capture

- Capture entire screen on button click
- Capture active window context (window title, app name, URL if browser)
- Capture timestamp and user location (if permitted)
- Visual feedback: Brief flash + "Captured!" notification
- Processing time: <3 seconds from capture to storage

FR-1.3: Multimodal Content Extraction

- Extract all visible text from screenshot (OCR)
- Identify UI elements (buttons, links, forms)
- Extract images and visual content
- Classify content type: article, job posting, product, video, message, document, code, social post
- Extract key entities: people, companies, dates, prices, locations, deadlines

FR-1.4: Metadata Enrichment

- Active application name
- Window title
- Browser URL (if applicable)
- Clipboard content (if recently copied)
- Current time and date
- User's current calendar status (free/busy)

Acceptance Criteria:

- [ ] Button visible on all desktop applications
- [ ] Capture completes in <3 seconds
- [ ] Text extraction accuracy >95% for clear text

- [ ] Content type classification accuracy >85%
- [ ] Supports all major applications (Chrome, Teams, Slack, etc.)

FR-2: Proactive Contextual Surfacing

Priority: CRITICAL
Complexity: Very High

Requirements:

FR-2.1: Continuous Monitoring Agent

- Runs in background every 30 minutes
- Checks user's calendar for upcoming events (next 24 hours)
- Monitors current application activity
- Tracks time since last proactive notification (minimum 2 hours between notifications)

FR-2.2: Relevance Detection

- Compare upcoming calendar event titles/descriptions with saved item content
- Semantic similarity threshold: >0.7 (configurable)
- Consider recency: Items saved in last 30 days prioritized
- Check if user has already reviewed the item
- Factor in urgency: Deadlines, expiring offers

FR-2.3: Smart Notification System

- Notification appears 15-60 minutes before relevant event (user configurable)
- Non-intrusive: Desktop notification (not modal)
- Content: Event name, number of relevant items, preview of top item
- Actions: "Review Now" (opens dashboard), "Remind Later" (30min), "Dismiss"
- Click notification opens dashboard filtered to relevant items

FR-2.4: Context-Aware Timing

- Don't notify during meetings (check calendar busy status)
- Don't notify during focus time (if user has Do Not Disturb enabled)
- Respect quiet hours (user configurable, default: 10pm - 8am)

- Batch related notifications (don't spam multiple for same event)

Acceptance Criteria:

- [ ] Monitoring agent runs continuously with <2% CPU usage
- [ ] Relevance detection accuracy >80% (user feedback loop)
- [ ] Notification timing: 90% within optimal window
- [ ] Zero notifications during user-defined quiet hours
- [ ] User can dismiss and never see same suggestion again

FR-3: Cross-Platform Context Synthesis

Priority: CRITICAL
Complexity: Very High

Requirements:

FR-3.1: Semantic Clustering

- Daily batch job analyzes all saved items
- Uses vector embeddings to find semantic similarity
- Groups items into clusters (minimum 3 items per cluster)
- Labels clusters with descriptive themes
- Identifies outliers (items that don't fit clusters)

FR-3.2: Connection Discovery

- Finds relationships between items across different platforms
- Relationship types: "similar_topic", "prerequisite", "followup", "contradiction", "supports"
- Creates knowledge graph structure
- Ranks connection strength (0.0 - 1.0)

FR-3.3: Weekly Synthesis Report

- Generates every Sunday at 9am (user configurable)
- Delivered via in-app notification + optional email
- Sections:
    1. Activity Summary (items saved, reviewed, ignored)
    2. Top Themes (3-5 main clusters with item counts)

3. Connections You Missed (surprising relationships)
4. Expiring Items (deadlines in next 7 days)
5. Suggested Actions (specific next steps)
6. Week-over-week comparison

FR-3.4: On-Demand Synthesis

- User can trigger synthesis for specific date range
- User can request synthesis for specific tag/category
- Export synthesis as PDF or Markdown

Acceptance Criteria:

- [ ] Clustering accuracy >75% (measured by user feedback on theme labels)
- [ ] Processes 100 items in <30 seconds
- [ ] Weekly report generation completes in <2 minutes
- [ ] Report readability score >70 (Flesch Reading Ease)
- [ ] Users find at least 1 valuable connection per report (>80% user satisfaction)

FR-4: Intent-Aware Organization

Priority: CRITICAL
Complexity: High

Requirements:

FR-4.1: Intent Classification

- Infer primary intent from content and context
- Intent categories: learn, buy, apply, remember, share, research, watch_later, read_later, reference
- Confidence score for each intent (0.0 - 1.0)
- Multi-intent support (item can have 2-3 intents)

FR-4.2: Urgency Scoring

- Extract deadlines from content (application dates, offer expirations, event dates)
- Calculate days until deadline
- Urgency levels: critical (<2 days), high (2-7 days), medium (7-30 days), low (>30 days), evergreen (no deadline)

- Factor in user's past behavior (does user typically act on this type of item?)

FR-4.3: Automatic Tagging

- Generate 3-7 descriptive tags per item
- Tag types: topic (e.g., "machine_learning"), entity (e.g., "Google"), action (e.g., "apply"), source (e.g., "linkedin")
- No generic tags (avoid "interesting", "important")
- Tags based on content AND user's vocabulary (learn from past user-created tags)

FR-4.4: Smart Collections

- Auto-create collections when 5+ items share tags/theme
- Collection types: Projects, Learning Paths, Job Hunt, Shopping, Research
- Suggest adding items to existing collections
- Collections have metadata: created_date, last_updated, item_count, completion_percentage

FR-4.5: Priority Ranking

- Every item gets priority score (0-100)
- Factors: urgency, intent confidence, user interest (based on time spent on similar items), recency
- Priority decays over time (old items become lower priority unless revisited)
- User can manually adjust priority

Acceptance Criteria:

- [ ] Intent classification accuracy >85%
- [ ] Deadline extraction accuracy >90%
- [ ] Tags meaningful and specific (user rating >4/5)
- [ ] Auto-collections contain related items (accuracy >80%)
- [ ] Priority ranking correlates with user action (items acted upon had higher priority 75% of time)

5.2 Differentiator Features (SHOULD HAVE - Tier 2)

FR-5: Voice Note Intelligence

Priority: HIGH
Complexity: Medium

Requirements:

FR-5.1: Voice Recording

- Double-tap floating button to start voice recording
- Visual feedback: Animated waveform, recording timer
- Maximum duration: 5 minutes
- Controls: Pause/Resume, Stop, Cancel
- Audio format: WAV or MP3, 16kHz sample rate

FR-5.2: Real-Time Transcription

- Use Gemini 3 Live API for transcription
- Display transcription as user speaks (live preview)
- Accuracy target: >95% for clear speech
- Support: English (primary), optionally Spanish, French, German

FR-5.3: Voice Note Processing

- Extract intent from transcription
- Identify entities (people, companies, dates, tasks)
- Link to related saved items (if user mentions "that article I saved")
- Create action items from imperative sentences ("Remind me to...")
- Associate with current context (what app was active when recording started)

FR-5.4: Voice Search

- User can search saved items using voice
- Semantic search (not just keyword matching)
- Example: "Find that thing Sarah mentioned about Old Navy" → Finds Slack message + saved offer

Acceptance Criteria:

- [ ] Recording starts in <500ms from button press
- [ ] Transcription accuracy >95% for clear audio
- [ ] Entity extraction accuracy >80%
- [ ] Voice search returns relevant results (user satisfaction >85%)
- [ ] Voice notes searchable alongside text captures

FR-6: Weekly Cognitive Digest

Priority: HIGH
Complexity: Medium

Requirements:

FR-6.1: Activity Analytics

- Track: items_saved, items_reviewed, items_acted_upon, items_ignored, items_expired
- Calculate: review_rate, action_rate, completion_rate
- Compare week-over-week trends
- Identify patterns: Most active days, most active times, most saved platforms

FR-6.2: Theme Analysis

- Identify top 3-5 themes from week's saves
- Quantify theme strength (number of items, user engagement with theme)
- Track theme evolution (new themes, declining themes)
- Suggest: "You're exploring X, here are learning resources"

FR-6.3: Missed Connections Report

- Find relationships user might have missed
- Example: "You saved a YouTube tutorial and a GitHub repo that implement the same concept"
- Suggest synthesis actions: "Want a summary of how these 5 items connect?"

FR-6.4: Expiring Items Alert

- List all items with deadlines in next 7 days
- Prioritize by importance (job applications > shopping deals)
- Show items with expired deadlines (opportunity cost report)

FR-6.5: Suggested Actions

- Specific, actionable next steps
- Example: "You saved 8 items about RAG pipelines but haven't built anything. Try this starter project: [link]"
- Personalized based on user's behavior and goals

FR-6.6: Comparative Insights

- "Last week you saved 15 items and reviewed 3 (20%). This week you saved 23 and reviewed 14 (61%). Great improvement!"
- Behavioral nudges: "You tend to review items saved on Monday more than items saved on Friday. Consider scheduling Friday review time."

Acceptance Criteria:

- [ ] Digest generated every week without fail
- [ ] Generation time <2 minutes
- [ ] Contains at least 3 actionable insights
- [ ] Users rate digest as "valuable" >80% of time
- [ ] Users act on at least 1 suggested action >60% of time

FR-7: Smart Reminders (Context-Aware)

Priority: HIGH
Complexity: High

Requirements:

FR-7.1: Multi-Factor Reminder Logic

- Factors considered:
    1. Calendar event (meeting about relevant topic)
    2. User availability (free time in schedule)
    3. Location (near relevant place, if location-aware)
    4. Time of day (user's productive hours)
    5. User's current activity (not during focus work)
    6. Item urgency (approaching deadline)

FR-7.2: Reminder Types

Time-Based:

- "Meeting in 30 minutes: Review 3 saved items about AI agents"

Location-Based (if enabled):

- "You're near Old Navy. The jeans offer expires tomorrow."

Context-Based:

- "You're in Splitwise (detected). You left 2 expenses unfinished 10 minutes ago."

Deadline-Based:

- "Job application deadline in 3 days. You saved interview prep materials. Review?"

Routine-Based:

- "You usually review saved items Sunday morning. You have 12 new items."

FR-7.3: Reminder Optimization

- Learn from user behavior: If user dismisses location reminders, reduce frequency
- A/B test timing: Find optimal lead time for each reminder type
- Consolidate reminders: Don't spam; batch related reminders

FR-7.4: Reminder Actions

- Quick actions from notification: "Review now", "Snooze 1hr", "Dismiss", "Mark done"
- Snooze intelligence: Suggest next optimal time based on user's schedule

Acceptance Criteria:

- [ ] Reminder relevance >85% (user acts on reminder vs. dismisses)
- [ ] Timing accuracy: Reminders appear when user is available 90% of time
- [ ] No more than 5 reminders per day (prevent notification fatigue)
- [ ] Users can configure reminder frequency and types
- [ ] Learning algorithm improves relevance over 2 weeks of usage

---

5.3 Polish Features (NICE TO HAVE - Tier 3)

FR-8: Collaborative Knowledge Sharing

Priority: LOW
Complexity: Medium

Requirements:

- Share individual items or collections with team members
- Collaborative collections (multiple users contribute)
- Comments and discussions on shared items
- Permission levels: view, comment, edit
- Integration with Slack/Teams for sharing

Acceptance Criteria:

- [ ] Users can share collections via link
- [ ] Shared collections update in real-time
- [ ] Comments thread visible to all collaborators

FR-9: Browser Extension Companion

Priority: LOW
Complexity: Low

Requirements:

- Chrome/Edge extension for quick web capture
- Floating mini-button on browser tabs
- Syncs with desktop app
- Keyboard shortcut for capture

Acceptance Criteria:

- [ ] Extension installs in <1 minute
- [ ] Captures sync to desktop app instantly
- [ ] Works on 95% of websites

FR-10: Mobile App (View-Only)

Priority: LOW
Complexity: Medium

Requirements:

- iOS/Android app for viewing saved items
- Receive smart reminders on phone

- Basic search and filter
- No capture functionality (desktop-only)

Acceptance Criteria:

- [ ] Users can view all saved items on mobile
- [ ] Push notifications for smart reminders
- [ ] Search works with similar accuracy as desktop

---

6. Technical Requirements

6.1 Technology Stack

6.1.1 Frontend/Desktop Application

Framework: Electron 28+
UI Framework: React 18+
Styling: Tailwind CSS 3+ with shadcn/ui components
State Management: React Context API or Zustand
Build Tool: Vite or Webpack

Justification:

- Electron: Cross-platform desktop app, access to system APIs
- React: Fast UI development, large ecosystem
- Tailwind + shadcn/ui: Professional enterprise UI with accessibility built-in
- Vite: Fast development builds

Key Libraries:

- `electron-builder`: Package desktop app for distribution
- `electron-store`: Local user preferences storage
- `react-router-dom`: Navigation within dashboard
- `recharts`: Data visualization for weekly digest
- `lucide-react`: Icon library

6.1.2 Backend/Intelligence Layer

Framework: Python 3.11+ with FastAPI
AI Integration: Google Gemini 3 API via google-generativeai SDK
Vector Database: Chroma or Pinecone
Task Queue: Celery with Redis (for background jobs)
API Communication: RESTful API (Electron ↔ Python)

Justification:

- Python: Strong AI/ML ecosystem, fast development
- FastAPI: Modern async API framework, automatic OpenAPI docs
- Gemini 3 SDK: Official Python client with multimodal support
- Chroma: Open-source vector DB, easy local setup
- Celery: Reliable background task processing

Key Libraries:

- `google-generativeai`: Official Gemini API client
- `fastapi`: API framework
- `uvicorn`: ASGI server
- `sqlalchemy`: Database ORM
- `chromadb`: Vector storage
- `pillow`: Image processing
- `python-multipart`: File upload handling
- `celery`: Background task queue
- `redis`: Message broker for Celery
- `python-dotenv`: Environment variable management

6.1.3 Data Layer

User Data: SQLite (local database)
Vector Store: ChromaDB (local) or Pinecone (cloud)
File Storage: Local file system (screenshots, voice recordings)
Cache: Redis (API response caching)

Justification:

- SQLite: Lightweight, no server needed, perfect for desktop app
- ChromaDB: Local-first, privacy-focused, good for hackathon demo

- Local file system: Fast access, user controls data
- Redis: Fast caching, used by Celery anyway

Database Schema:

```
tables:
  - users (id, name, email, preferences, created_at)
  - captures (id, user_id, screenshot_path, timestamp, window_context)
  - items (id, capture_id, content_type, extracted_text, entities, intent, urgency, priority)
  - tags (id, item_id, tag_name, tag_type, confidence)
  - collections (id, user_id, name, description, auto_generated)
  - reminders (id, item_id, reminder_type, trigger_time, status)
  - user_activity (id, user_id, item_id, action_type, timestamp)
```

6.1.4 External Integrations

Calendar: Google Calendar API
Email: Gmail API
Teams (Optional): Microsoft Graph API

Authentication: OAuth 2.0 for all services
API Rate Limiting: Respect provider limits, implement backoff

---

6.2 Development Environment Setup

6.2.1 Required Software

- **Node.js:** v18+ (for Electron)
- **Python:** 3.11+
- **Redis:** Latest stable (for Celery)
- **Git:** Version control
- **VS Code/Cursor IDE:** Recommended IDE

6.2.2 Development Tools

- **API Testing:** Postman or Thunder Client
- **Database Viewer:** DB Browser for SQLite
- **Redis GUI:** RedisInsight

- **Screen Recording:** OBS Studio (for demo video)

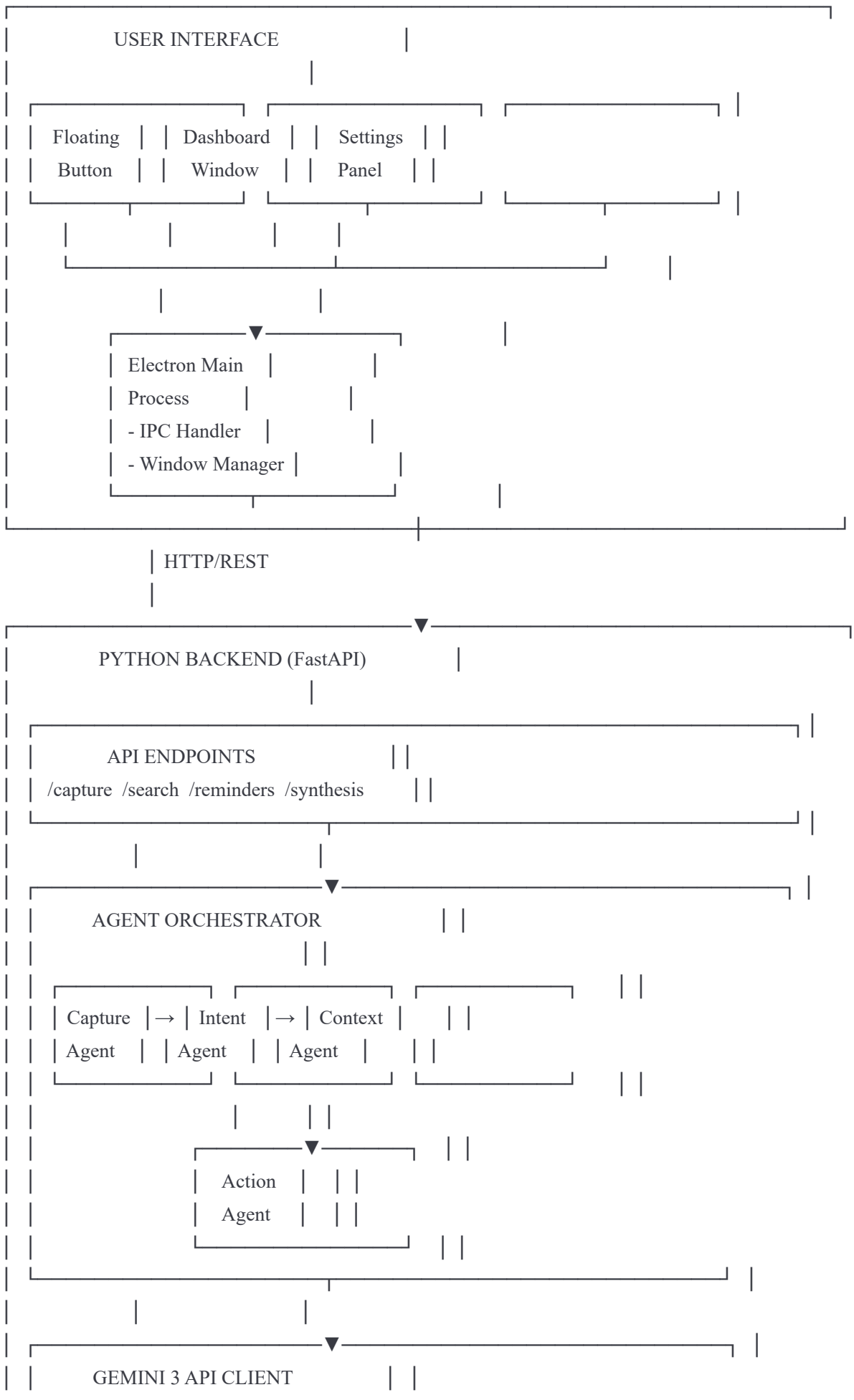
6.2.3 Environment Variables

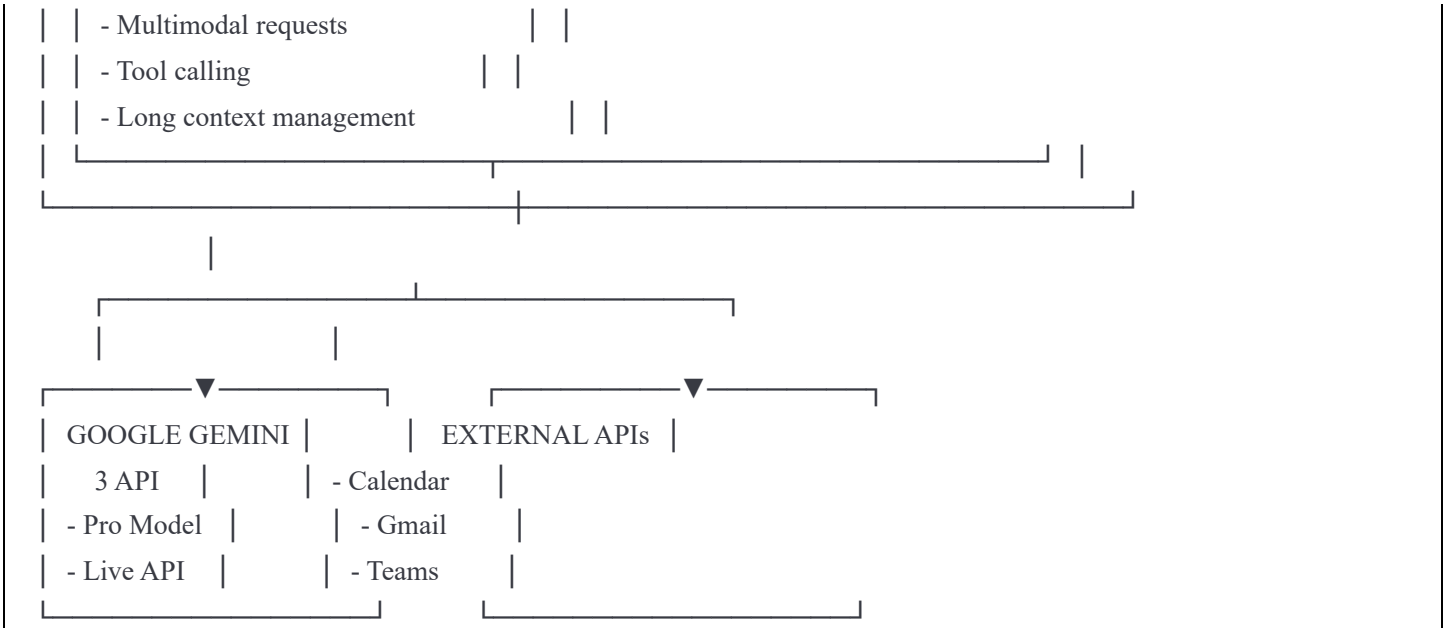
```
# Python Backend (.env)
GEMINI_API_KEY=your_api_key_here
DATABASE_URL=sqlite:///./lifeos.db
REDIS_URL=redis://localhost:6379
VECTOR_DB_PATH=./chroma_db
GOOGLE_CALENDAR_CLIENT_ID=...
GOOGLE_CALENDAR_CLIENT_SECRET=...
GMAIL_CLIENT_ID=...
GMAIL_CLIENT_SECRET=...

# Electron Frontend (.env)
VITE_API_URL=http://localhost:8000
VITE_ENV=development
```

6.3 System Architecture Diagrams

6.3.1 Component Architecture


USER INTERFACE
Floating Button
Dashboard Window
Settings Panel
Electron Main Process
- IPC Handler
- Window Manager
HTTP/REST
PYTHON BACKEND (FastAPI)
API ENDPOINTS
/capture /search /reminders /synthesis
AGENT ORCHESTRATOR
Capture Agent
Intent Agent
Context Agent
Action Agent
GEMINI 3 API CLIENT



6.3.2 Data Flow: Capture to Reminder

1. USER CAPTURES

User clicks button → Electron captures screen

↓

2. SEND TO BACKEND

POST /api/capture {screenshot, context}

↓

3. CAPTURE AGENT

Gemini 3: Extract text, classify, identify entities

↓

4. INTENT AGENT

Gemini 3: Analyze user history, infer intent

↓

5. CONTEXT AGENT

- Fetch calendar events (Google Calendar API)
- Search email for mentions (Gmail API)
- Find related saved items (Vector DB query)
- Gemini 3: Synthesize full context

↓

6. ACTION AGENT

- Calculate urgency score
- Determine optimal reminder time
- Create reminder records in DB
- Gemini 3: Generate notification text

↓

7. STORAGE

- Save to SQLite (structured data)
- Save to ChromaDB (vector embeddings)
- Save screenshot to file system

↓

8. BACKGROUND MONITORING

Celery task runs every 30 minutes:

- Check upcoming calendar events
- Query ChromaDB for relevant items
- If match found → Send notification

↓

9. PROACTIVE NOTIFICATION

Desktop notification appears

User clicks → Dashboard opens with filtered items

7. Gemini 3 Integration Requirements

7.1 API Access & Configuration

Gemini 3 Models to Use:

- **Primary:** `gemini-3-pro-20250514` (Claude Sonnet 4.5 equivalent)
- **Fallback:** `gemini-3-flash` (if rate limited)
- **Voice:** `gemini-3-live` (for voice transcription)

API Key Management:

- Store in environment variables (never commit to git)
- Use Google AI Studio for free tier access
- Rate limits: Monitor and respect (100 requests/minute for free tier)

Configuration:

```python
import google.generativeai as genai

genai.configure(api_key=os.environ["GEMINI_API_KEY"])

model_config = {
    "temperature": 0.7,  # Balanced creativity/consistency
    "top_p": 0.95,
    "top_k": 40,
    "max_output_tokens": 8192,
    "response_mime_type": "application/json"  # For structured output
}

model = genai.GenerativeModel(
    model_name="gemini-3-pro-20250514",
    generation_config=model_config,
    safety_settings={...}
)
```

7.2 Feature-Specific Gemini Usage

7.2.1 Capture Agent (Multimodal Understanding)

Gemini 3 Features Used:

- ✅ **Multimodal input:** Image (screenshot) + Text (window context)
- ✅ **Vision capabilities:** Extract text, identify UI elements, understand layouts
- ✅ **Entity extraction:** Recognize people, companies, dates, products

Implementation:

```python
async def analyze_capture(screenshot_path: str, window_context: dict):
    """
    Use Gemini 3 to understand screenshot content
    """
    # Load image
    image = PIL.Image.open(screenshot_path)

    # Construct prompt
    prompt = f"""
    Analyze this screenshot captured from {window_context['app_name']}.
    Window title: {window_context['title']}

    Extract:
    1. Content type (article, job_posting, product, video, social_post, etc.)
    2. All visible text (maintain structure)
    3. Key entities (people, companies, dates, prices, deadlines)
    4. Visual elements (buttons, links, images)
    5. Main topic/theme

    Return as structured JSON.
    """

    # Call Gemini 3
    response = model.generate_content([prompt, image])

    return json.loads(response.text)
```

Expected Input:

- Screenshot: 1920x1080 PNG
- Window context: `{"app_name": "Google Chrome", "title": "Job Posting - Google Careers"}`

Expected Output:

```json
{
  "content_type": "job_posting",
  "extracted_text": "Frontend Developer\nGoogle • Mountain View, CA\nFull-time • $150k-$200k\nDeadline: Feb 20, 2026...",
  "entities": {
    "company": "Google",
    "role": "Frontend Developer",
    "location": "Mountain View, CA",
    "salary_range": {"min": 150000, "max": 200000},
    "deadline": "2026-02-20"
  },
  "visual_elements": ["Apply Now button", "Save Job button", "Company logo"],
  "main_topic": "job_opportunity_frontend_development"
}
```

7.2.2 Intent Agent (Reasoning)

Gemini 3 Features Used:

- ✅ **Reasoning:** Infer user motivation from minimal data
- ✅ **Long context:** Access user's full history (100+ previous saves)
- ✅ **Pattern recognition:** Identify behavioral patterns

Implementation:

```python
```

```python
async def infer_intent(capture_data: dict, user_history: list):
    """
    Use Gemini 3 to understand WHY user saved this
    """
    # Prepare context (last 100 saves)
    history_summary = "\n".join([
        f"- {item['date']}: {item['content_type']} about {item['topic']}"
        for item in user_history[-100:]
    ])

    prompt = f"""
    A user just captured this content:
    Type: {capture_data['content_type']}
    Main topic: {capture_data['main_topic']}
    Entities: {capture_data['entities']}

    User's recent activity (last 100 saves):
    {history_summary}

    Analyze:
    1. Primary intent (learn, buy, apply, remember, share, research, reference)
    2. Confidence level (0.0-1.0)
    3. Why you think this (reasoning)
    4. How urgent is this? (critical/high/medium/low/evergreen)
    5. Suggested actions (3-5 specific next steps)
    6. Estimated effort required

    Return as structured JSON.
    """

    response = model.generate_content(prompt)
    return json.loads(response.text)
```

Expected Output:

```
json
```

```
{
  "primary_intent": "apply_for_job",
  "confidence": 0.92,
  "reasoning": "User has saved 5 other frontend developer jobs in the past 2 weeks, updated resume 3 days ago, and recently s
  "urgency": "high",
  "deadline_days": 34,
  "suggested_actions": [
    "Review and tailor resume for frontend role",
    "Research Google's interview process (you saved an article about this)",
    "Update portfolio website with recent projects",
    "Prepare answers to common frontend interview questions",
    "Set reminder for Feb 15 to submit application"
  ],
  "estimated_effort": "3-4 hours total (resume: 1h, research: 1h, application: 1-2h)"
}
```

7.2.3 Context Agent (Long Context + Tool Calling)

Gemini 3 Features Used:

- ✅ **1M token context window:** Load entire user history + calendar + emails
- ✅ **Tool calling:** Invoke external APIs (Calendar, Gmail, Vector DB)
- ✅ **Synthesis:** Connect information across sources

Tool Definitions:

```
python
```

```
tools = [
    {
        "name": "get_calendar_events",
        "description": "Fetch upcoming calendar events",
        "parameters": {
            "type": "object",
            "properties": {
                "days_ahead": {"type": "integer", "description": "Number of days to look ahead"}
            }
        }
    },
    {
        "name": "search_emails",
        "description": "Search Gmail for mentions of keywords",
        "parameters": {
            "type": "object",
            "properties": {
                "keywords": {"type": "array", "items": {"type": "string"}}
            }
        }
    },
    {
        "name": "search_saved_items",
        "description": "Semantic search in vector database",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {"type": "string"},
                "limit": {"type": "integer"}
            }
        }
    }
]
```

Implementation:

python

```
async def enrich_context(intent_data: dict, user_id: str):
    """
    Use Gemini 3 with tool calling to gather full context
    """
    prompt = f"""
    A user saved a {intent_data['content_type']} about {intent_data['main_topic']}.
    Intent: {intent_data['primary_intent']}

    Use the available tools to:
    1. Check if user has upcoming calendar events related to this topic
    2. Search emails for recent mentions of relevant entities
    3. Find related items user has saved before

    Then synthesize:
    - How does this connect to user's upcoming schedule?
    - What related research/preparation has user done?
    - What's the broader context/pattern here?

    Return structured analysis.
    """

    # Gemini 3 will automatically call tools as needed
    response = model.generate_content(
        prompt,
        tools=tools
    )

    # Process tool calls
    for tool_call in response.candidates[0].content.parts:
        if hasattr(tool_call, 'function_call'):
            result = await execute_tool(tool_call.function_call)
            # Send result back to Gemini

    return final_synthesis
```

Expected Output:

```
json
```

```
{
 "calendar_connections": [
  {
   "event": "Weekly Team Sync - Discuss hiring pipeline",
   "date": "2026-01-22 14:00",
   "relevance": "This job posting could be discussed as candidate profile"
  }
 ],
 "email_mentions": [
  {
   "from": "john@company.com",
   "subject": "Google frontend roles - any interest?",
   "date": "2026-01-10",
   "snippet": "Hey, I saw Google is hiring frontend devs..."
  }
 ],
 "related_saved_items": [
  {
   "title": "How to prepare for Google interviews",
   "saved_date": "2026-01-08",
   "relevance_score": 0.94
  },
  {
   "title": "My updated portfolio website",
   "saved_date": "2026-01-15",
   "relevance_score": 0.87
  }
 ],
 "synthesis": "This job posting fits into a clear pattern: User is actively job hunting for frontend roles, with Google being a to
 "recommended_timing": "Surface reminder on Jan 22 before team meeting (user may want to mention they're applying)"
}
```

7.2.4 Action Agent (Planning + Reasoning)

Gemini 3 Features Used:

- ✅ **Multi-step planning:** Generate action sequences
- ✅ **Temporal reasoning:** Determine optimal timing
- ✅ **Personalization:** Adapt to user's behavior patterns

Implementation:

```python
async def plan_actions(enriched_context: dict, user_preferences: dict):
    """
    Use Gemini 3 to create action plan
    """
    prompt = f"""
    Based on this enriched context:
    {json.dumps(enriched_context, indent=2)}

    User preferences:
    - Notification style: {user_preferences['notification_style']}
    - Quiet hours: {user_preferences['quiet_hours']}
    - Typical work schedule: {user_preferences['work_schedule']}

    Create an action plan:
    1. Immediate actions (do right now)
    2. Scheduled reminders (when and why)
    3. Background monitoring (what to watch for)
    4. Notification text (compelling, specific, actionable)

    Optimize timing based on:
    - User's calendar availability
    - Deadlines and urgency
    - User's typical behavior patterns (when do they usually act on this type of item?)

    Return as structured JSON.
    """

    response = model.generate_content(prompt)
    return json.loads(response.text)
```

Expected Output:

```json
```

```
{
 "immediate_actions": [
  {
   "type": "calendar_event",
   "action": "create",
   "data": {
    "title": "Prepare Google application materials",
    "date": "2026-02-15T09:00:00",
    "duration": 180,
    "description": "Deadline: Feb 20. Tasks: Tailor resume, update portfolio, draft cover letter"
   }
  },
  {
   "type": "link_items",
   "action": "create_collection",
   "data": {
    "name": "Google Frontend Application",
    "items": ["current_job_posting", "save_123", "save_089"]
   }
  }
 ],
 "scheduled_reminders": [
  {
   "trigger_type": "time_based",
   "trigger_time": "2026-01-22T13:30:00",
   "reason": "30min before team meeting where this could be mentioned",
   "notification": {
    "title": "Google Job Reminder",
    "body": "Your team meeting at 2pm might be a good time to mention you're applying to Google. You saved this frontend
    "actions": ["View Application", "Review Prep Materials", "Dismiss"]
   }
  },
  {
   "trigger_type": "deadline_based",
   "trigger_time": "2026-02-13T09:00:00",
   "reason": "7 days before application deadline",
   "notification": {
    "title": "Google Application Due in 1 Week",
    "body": "Application deadline Feb 20. You have prep materials saved. Have you started your application?",
    "actions": ["Start Application", "Review Materials", "Snooze 2 Days"]
   }
  },
  {
```

```
    "trigger_type": "context_based",
    "trigger_condition": "user_has_free_weekend",
    "notification": {
      "title": "Good Time to Work on Google Application?",
      "body": "You have a free Saturday coming up. Perfect time to tackle that Google application (estimated 3-4 hours).",
      "actions": ["Block Calendar Time", "Review Plan", "Maybe Later"]
    }
  }
 ],
 "background_monitoring": {
  "watch_keywords": ["Google", "frontend developer", "interview preparation"],
  "connect_automatically": true,
  "notify_on_related_item": true
 }
}
```

7.2.5 Weekly Synthesis (Long Context Reasoning)

Gemini 3 Features Used:

- ✅ **1M token context:** Process 100+ saved items at once
- ✅ **Pattern recognition:** Identify themes and connections
- ✅ **Insight generation:** Surface non-obvious relationships

Implementation:

```
python
```

```
async def generate_weekly_digest(user_id: str, week_start: datetime, week_end: datetime):
    """
    Use Gemini 3 to analyze entire week and generate insights
    """
    # Fetch all items from the week (could be 50-100 items)
    items = fetch_items_in_date_range(user_id, week_start, week_end)

    # Prepare full context (fits in 1M token window)
    items_text = "\n\n".join([
        f"Item {i+1}:\n"
        f"Date: {item['timestamp']}\n"
        f"Type: {item['content_type']}\n"
        f"Topic: {item['main_topic']}\n"
        f"Content: {item['extracted_text'][:500]}...\n"
        f"Intent: {item['intent']}\n"
        f"User action: {item['user_action'] or 'None yet'}"
        for i, item in enumerate(items)
    ])

    prompt = f"""
    Analyze this user's week of saved items ({len(items)} total):

    {items_text}

    Generate a weekly digest with:

    1. ACTIVITY SUMMARY
       - Total items saved
       - Items reviewed vs ignored
       - Completion rate compared to last week

    2. TOP THEMES (3-5 main topics)
       - Identify semantic clusters
       - Count items per theme
       - Note user's engagement level with each theme

    3. MISSED CONNECTIONS (2-3 surprising relationships)
       - Find non-obvious links between items
       - Explain why these connections matter
       - Suggest synthesis actions

    4. EXPIRING ITEMS
       - List items with deadlines in next 7 days
```

```
    - Prioritize by importance

    5. SUGGESTED ACTIONS (3-5 specific next steps)
    - Based on patterns and themes
    - Actionable and personalized

    6. BEHAVIORAL INSIGHTS
    - What patterns emerged this week?
    - Suggestions for better information management

    Write in friendly, encouraging tone. Be specific and actionable.
    Return as structured JSON with markdown formatting for readability.
    """

    response = model.generate_content(prompt)
    return json.loads(response.text)
```

7.2.6 Voice Note Transcription (Live API)

Gemini 3 Features Used:

- ✅ **Live API:** Real-time transcription
- ✅ **Streaming:** Display transcription as user speaks
- ✅ **Entity extraction:** From voice commands

Implementation:

```
python
```

```python
from google.generativeai import LiveClient

async def transcribe_voice_note(audio_stream):
    """
    Use Gemini Live API for real-time transcription
    """
    live_client = LiveClient(
        model="gemini-3-live",
        config={
            "response_modalities": ["TEXT"],
            "speech_config": {
                "voice_config": {"prebuilt_voice_config": {"voice_name": "Aoede"}}
            }
        }
    )

    # Stream audio
    async for audio_chunk in audio_stream:
        await live_client.send(audio_chunk)

    # Get transcription
    transcription = ""
    async for response in live_client.receive():
        if response.text:
            transcription += response.text
            yield transcription  # Stream to UI

    # Process transcription for intent
    processed = await process_voice_command(transcription)
    return processed
```

7.3 Gemini 3 Performance Optimization

7.3.1 Caching Strategy

- **Cache model responses** for identical requests (24hr TTL)
- **Cache embeddings** for unchanged content
- **Batch process** multiple items when possible

7.3.2 Rate Limit Management

```python
from tenacity import retry, wait_exponential, stop_after_attempt

@retry(
    wait=wait_exponential(multiplier=1, min=4, max=60),
    stop=stop_after_attempt(5)
)
async def call_gemini_with_retry(prompt, **kwargs):
    """
    Automatic retry with exponential backoff
    """
    try:
        return await model.generate_content(prompt, **kwargs)
    except Exception as e:
        if "rate_limit" in str(e):
            # Log and retry
            logger.warning(f"Rate limit hit, retrying...")
            raise
        else:
            # Different error, don't retry
            logger.error(f"Gemini API error: {e}")
            raise
```

7.3.3 Token Management

- **Monitor token usage** per request
- **Truncate context** if approaching limits (prioritize recent items)
- **Use summarization** for very long context (summarize older items)

```python
```

```python
def prepare_context_with_token_limit(items: list, max_tokens: int = 900000):
    """
    Ensure context fits within 1M token limit (leaving buffer)
    """
    context_items = []
    estimated_tokens = 0

    # Prioritize: Recent items full detail, older items summarized
    for item in sorted(items, key=lambda x: x['timestamp'], reverse=True):
        item_tokens = estimate_tokens(item['content'])

        if estimated_tokens + item_tokens > max_tokens:
            # Summarize remaining items
            remaining_summary = summarize_items(items[len(context_items):])
            context_items.append(remaining_summary)
            break

        context_items.append(item)
        estimated_tokens += item_tokens

    return context_items
```

7.4 Prompt Engineering Best Practices

7.4.1 Structured Output

Always request JSON output for programmatic processing:

```python
```

```
prompt = """
Analyze this content and return ONLY valid JSON with this structure:
{
  "content_type": "string",
  "main_topic": "string",
  "entities": {...},
  "intent": "string"
}

Do not include any text before or after the JSON.
"""

# Configure model for JSON
model_config = {
    "response_mime_type": "application/json"
}
```

7.4.2 Clear Instructions

- Use numbered lists for multi-step tasks
- Provide examples when possible
- Specify output format explicitly
- Use delimiters (```, ---, ###) to separate sections

7.4.3 Context Priming

python

```
system_instruction = """
You are an intelligent assistant helping users manage digital information.
Your goal is to understand user intent and provide actionable insights.

Key principles:
- Be specific, not generic
- Focus on "why" not just "what"
- Consider temporal context (deadlines, schedules)
- Personalize based on user patterns
- Always suggest concrete next steps
"""
```

7.5 Gemini 3 Testing Strategy

7.5.1 Unit Tests for Each Agent

python

```python
import pytest
from agents.capture_agent import analyze_capture

@pytest.mark.asyncio
async def test_capture_agent_job_posting():
    """
    Test that capture agent correctly identifies job posting
    """
    result = await analyze_capture(
        screenshot_path="tests/fixtures/job_posting.png",
        window_context={"app_name": "Chrome", "title": "Careers at Google"}
    )

    assert result['content_type'] == 'job_posting'
    assert 'company' in result['entities']
    assert result['entities']['company'] == 'Google'
    assert 'deadline' in result['entities']
```

7.5.2 Integration Tests

python

```python
@pytest.mark.asyncio
async def test_full_capture_pipeline():
    """
    Test entire pipeline: Capture → Intent → Context → Action
    """
    # Simulate user capture
    capture_result = await capture_agent.process(screenshot, context)

    # Verify intent inference
    intent_result = await intent_agent.process(capture_result, user_history)
    assert intent_result['primary_intent'] in VALID_INTENTS

    # Verify context enrichment
    context_result = await context_agent.process(intent_result, user_id)
    assert len(context_result['related_saved_items']) > 0

    # Verify action planning
    action_result = await action_agent.process(context_result, preferences)
    assert len(action_result['scheduled_reminders']) > 0
```

7.5.3 Performance Tests

python

```python
@pytest.mark.asyncio
async def test_capture_performance():
    """
    Ensure capture processing completes in <3 seconds
    """
    import time

    start = time.time()
    result = await analyze_capture(screenshot, context)
    duration = time.time() - start

    assert duration < 3.0, f"Capture took {duration}s, exceeds 3s requirement"
```

---

8. User Interface Requirements

8.1 Design Principles

Enterprise-Grade Standards:

- Clean, minimal, professional aesthetic
- Consistent design system (shadcn/ui components)
- Neutral color palette with brand accent
- Generous white space
- Clear visual hierarchy

Accessibility First:

- WCAG 2.1 Level AA compliance minimum
- Color contrast ratio ≥ 4.5:1 for text
- Keyboard navigation for all interactions
- Screen reader support with ARIA labels
- Respect user's motion preferences

Multi-Functional:

- Dashboard, capture, settings, search, synthesis views
- Modal overlays for quick actions
- Drag-and-drop for organization
- Contextual menus and shortcuts

8.2 Component Specifications

8.2.1 Floating Capture Button

Visual Design:

- **Size:** 64px × 64px circular button
- **Position:** Draggable, remembers last position (default: bottom-right, 100px from edges)
- **Colors:**
  - Background: Gradient `linear-gradient(135deg, #667eea 0%, #764ba2 100%)`
  - Hover: Brighten 10%, scale 1.1

    - Active: Scale 0.95
    - Success: Green pulse animation
- **Icon:** Brain emoji 🧠 or custom brain SVG (40px)
- **Shadow:** `0 4px 15px rgba(0,0,0,0.3)` for depth

Interaction States:

```
Idle → Default appearance, subtle pulse animation (1.5s cycle)
Hover → Scale up, tooltip appears ("Capture - Ctrl+Shift+L")
Pressed →
```