Personal AI Employee Hackathon 0: Building Autonomous FTEs (Full-Time Equivalent) in 2026

Tagline: *Your life and business on autopilot. Local-first, agent-driven, human-in-the-loop.*

This document serves as a comprehensive architectural blueprint and hackathon guide for building a "Digital FTE" (Full-Time Equivalent). It proposes a futuristic, local-first approach to automation where an AI agent—powered by Claude Code and Obsidian—proactively manages personal and business affairs 24/7. You can also think of it as a "Smart Consultant" (General Agents). The focus is on high-level reasoning, autonomy, and flexibility. Think of it as hiring a senior employee who figures out how to solve the problems.

This hackathon takes the concept of a "Personal AI Employee" to its logical extreme. It doesn't just wait for you to type; it proactively manages your "Personal Affairs" (Gmail, WhatsApp, Bank) and your "Business" (Social Media, Payments, Project Tasks) using **Claude Code** as the executor and **Obsidian** as the management dashboard.

All our faculty members and students will build this Personal AI Employee using Claude Code.

Standout Idea: The "Monday Morning CEO Briefing," where the AI autonomously audits bank transactions and tasks to report revenue and bottlenecks, transforms the AI from a chatbot into a proactive business partner.

Architecture & Tech Stack:

The proposed stack is robust, privacy-focused, and clever:

- **The Brain:** Claude Code acts as the reasoning engine. We add the Ralph Wiggum Stop hook to let the agent continuously iterate until the assigned task is complete.
- **The Memory/GUI:** Obsidian (local Markdown) is used as the dashboard, keeping data local and accessible.
- **The Senses (Watchers):** Lightweight Python scripts monitor Gmail, WhatsApp, and filesystems to trigger the AI.
- **The Hands (MCP):** Model Context Protocol (MCP) servers handle external actions like sending emails or clicking buttons.

This architecture solves the "lazy agent" problem by using "Watchers" to wake the agent up rather than waiting for user input and "Ralph Wiggum" (a Stop hook pattern) to keep it working until done.

This is an exceptional technical hackathon project. It moves beyond "prompt engineering" into "agent engineering." It provides a complete, viable path to building a functional autonomous agent using tools available in 2026 (or today).

Research and Show Case Meeting Every Wednesday:
We will be holding a Research Meeting every Wednesday at 10:00 pm on Zoom all of you are welcome to join, the first meeting will be held on Wednesday, Jan 7th, 2026:

https://us06web.zoom.us/j/87188707642?pwd=a9XloCsinvn1JzlCbPc2YGUvWTbOTr.1

- *Meeting ID:* 871 8870 7642
- *Passcode:* 744832

If the Zoom meeting is full, you may watch live or recording at:

https://www.youtube.com/@panaversity

In these meetings we will be teaching each other how to build and enhance our first AI Employee.

Digital FTE: The New Unit of Value

A Digital FTE (Full-Time Equivalent) is an AI agent that is built, "hired," and priced as if it were a human employee. This shifts the conversation from "software licenses" to "headcount budgets."

Human FTE vs Digital FTE

Feature	Human FTE	Digital FTE (Custom Agent)
Availability	40 hours / week	168 hours / week (24/7)
Monthly Cost	$4,000 – $8,000+	$500 – $2,000
Ramp-up Time	3 – 6 Months	Instant (via SKILL.md)
Consistency	Variable (85–95% accuracy)	Predictable (99%+ consistency)
Scaling	Linear (Hire 10 for 10x work)	Exponential (Instant duplication)
Cost per Task	~$3.00 – $6.00	~$0.25 – $0.50
Annual Hours	~2,000 hours	~8,760 hours

The 'Aha!' Moment: A Digital FTE works nearly 9,000 hours a year vs a human's 2,000. The cost per task reduction (from ~$5.00 to ~$0.50) is an 85–90% cost saving—usually the threshold where a CEO approves a project without further debate.

Prerequisites & Setup

Before diving into building your Personal AI Employee, ensure you have the following prerequisites in place. Estimated total setup time: 2-3 hours.

Required Software

Component	Requirement	Purpose
Claude Code	Active subscription (Pro or Use Free Gemini API with Claude Code Router)	Primary reasoning engine
Obsidian	v1.10.6+ (free)	Knowledge base & dashboard
Python	3.13 or higher	Sentinel scripts & orchestration
Node.js	v24+ LTS	MCP servers & automation
Github Desktop	Latest stable	Version control for your vault

Hardware Requirements

- Minimum: 8GB RAM, 4-core CPU, 20GB free disk space
- Recommended: 16GB RAM, 8-core CPU, SSD storage
- For always-on operation: Consider a dedicated mini-PC or cloud VM
- Stable internet connection for API calls (10+ Mbps recommended)

Skill Level Expectations

This hackathon assumes intermediate technical proficiency:

- Comfortable with command-line interfaces (terminal/bash)
- Understanding of file systems and folder structures
- Familiarity with APIs (what they are, how to call them)
- No prior AI/ML experience required
- Able to use and prompt Claude Code
- Prompt Claude Code to to convert AI functionality into Agent Skills

Pre-Hackathon Checklist

1. Install all required software listed above
2. Create a new Obsidian vault named "AI_Employee_Vault"
3. Verify Claude Code works by running: claude --version
4. Set up a UV Python project
5. Join the Wednesday Research Meeting Zoom link

Hackathon Scope & Tiered Deliverables

To accommodate varying skill levels and time availability, we define three achievement tiers. Choose your target based on your experience and ambition.

Bronze Tier: Foundation (Minimum Viable Deliverable)

Estimated time: 8-12 hours

- Obsidian vault with Dashboard.md and Company_Handbook.md
- One working Watcher script (Gmail OR file system monitoring)
- Claude Code successfully reading from and writing to the vault
- Basic folder structure: /Inbox, /Needs_Action, /Done
- All AI functionality should be implemented as Agent Skills

Silver Tier: Functional Assistant

Estimated time: 20-30 hours

1. All Bronze requirements plus:
2. Two or more Watcher scripts (e.g., Gmail + Whatsapp + LinkedIn)
3. Automatically Post on LinkedIn about business to generate sales
4. Claude reasoning loop that creates Plan.md files
5. One working MCP server for external action (e.g., sending emails)
6. Human-in-the-loop approval workflow for sensitive actions
7. Basic scheduling via cron or Task Scheduler
8. All AI functionality should be implemented as Agent Skills

Gold Tier: Autonomous Employee

Estimated time: 40+ hours

1. All Silver requirements plus:
2. Full cross-domain integration (Personal + Business)
3. Create an accounting system for your business in Odoo Community (self-hosted, local) and integrate it via an MCP server using Odoo's JSON-RPC APIs (Odoo 19+).
4. Integrate Facebook and Instagram and post messages and generate summary
5. Integrate Twitter (X) and post messages and generate summary
6. Multiple MCP servers for different action types
7. Weekly Business and Accounting Audit with CEO Briefing generation
8. Error recovery and graceful degradation
9. Comprehensive audit logging
10. Ralph Wiggum loop for autonomous multi-step task completion (see Section 2D)

11. Documentation of your architecture and lessons learned
12. All AI functionality should be implemented as [Agent Skills]

Platinum Tier: Always-On Cloud + Local Executive (Production-ish AI Employee)

Estimated time: 60+ hours
All Gold requirements plus:

1. **Run the AI Employee on Cloud 24/7** (always-on watchers + orchestrator + health monitoring). You can deploy a Cloud VM (Oracle/AWS/etc.) - [Oracle Cloud Free VMs] can be used for this (subject to limits/availability).
2. **Work-Zone Specialization (domain ownership)**:
    a. **Cloud owns:** Email triage + draft replies + social post drafts/scheduling (draft-only; requires Local approval before send/post)
    b. **Local owns:** approvals, WhatsApp session, payments/banking, and final "send/post" actions
3. Delegation via Synced Vault (Phase 1)
    a. Agents communicate by **writing files** into:
        i. /Needs_Action/<domain>/, /Plans/<domain>/, /Pending_Approval/<domain>/
    b. Prevent double-work using:
        i. /In_Progress/<agent>/ claim-by-move rule
        ii. single-writer rule for Dashboard.md (Local)
        iii. Cloud writes updates to /Updates/ (or /Signals/), and Local merges them into Dashboard.md.
    c. For Vault sync (Phase 1) use Git (recommended) or Syncthing.
    d. **Claim-by-move rule:** first agent to move an item from /Needs_Action to /In_Progress/<agent>/ owns it; other agents must ignore it.
4. **Security rule:** Vault sync includes only markdown/state. Secrets never sync (.env, tokens, WhatsApp sessions, banking creds). So Cloud never stores or uses WhatsApp sessions, banking credentials, or payment tokens.
5. **Deploy Odoo Community on a Cloud VM (24/7)** with HTTPS, backups, and health monitoring; integrate Cloud Agent with Odoo via MCP for draft-only accounting actions and Local approval for posting invoices/payments.
6. Optional A2A Upgrade (Phase 2): Replace some file handoffs with direct A2A messages later, while keeping the vault as the audit record
7. **Platinum demo (minimum passing gate):** Email arrives while Local is offline → Cloud drafts reply + writes approval file → when Local returns, user approves → Local executes send via MCP → logs → moves task to /Done.

1. The "Foundational Layer" (Local Engine)

- **The Nerve Center (Obsidian):** Acts as the **GUI (Graphical User Interface)** and **Long-Term Memory**.
    - **Dashboard.md:** Real-time summary of bank balance, pending messages, and active

business projects.
    - **Company_Handbook.md:** Contains your "Rules of Engagement" (e.g., "Always be polite on WhatsApp," "Flag any payment over $500 for my approval").
- **The Muscle (Claude Code):** Runs in your terminal, pointed at your Obsidian vault. It uses its **File System tools** to read your tasks and write reports. The Ralph Wiggum loop (a Stop hook) keeps Claude iterating until multi-step tasks are complete.

2. Architecture: Perception → Reasoning → Action

A. Perception (The "Watchers")

Since Claude Code can't "listen" to the internet 24/7, you use lightweight **Python Sentinel Scripts** running in the background:

- **Comms Watcher:** Monitors Gmail and WhatsApp (via local web-automation or APIs) and saves new urgent messages as .md files in a /Needs_Action folder.
- **Finance Watcher:** Downloads local CSVs or calls banking APIs to log new transactions in /Accounting/Current_Month.md.
- It will also be able to run on your laptop and immediately “wake up” as soon as you open your machine.

Watcher Architecture

The Watcher layer is your AI Employee's sensory system. These lightweight Python scripts run continuously, monitoring various inputs and creating actionable files for Claude to process.

Core Watcher Pattern

All Watchers follow this structure:

```
# base_watcher.py - Template for all watchers
import time
import logging
from pathlib import Path
from abc import ABC, abstractmethod

class BaseWatcher(ABC):
    def __init__(self, vault_path: str, check_interval: int = 60):
        self.vault_path = Path(vault_path)
        self.needs_action = self.vault_path / 'Needs_Action'
        self.check_interval = check_interval
        self.logger = logging.getLogger(self.__class__.__name__)

    @abstractmethod
    def check_for_updates(self) -> list:
        '''Return list of new items to process'''
        pass
```

```
    @abstractmethod
    def create_action_file(self, item) -> Path:
        '''Create .md file in Needs_Action folder'''
        pass

    def run(self):
        self.logger.info(f'Starting {self.__class__.__name__}')
        while True:
            try:
                items = self.check_for_updates()
                for item in items:
                    self.create_action_file(item)
            except Exception as e:
                self.logger.error(f'Error: {e}')
            time.sleep(self.check_interval)
```

Gmail Watcher Implementation

```
# gmail_watcher.py
from google.oauth2.credentials import Credentials
from googleapiclient.discovery import build
from base_watcher import BaseWatcher
from datetime import datetime

class GmailWatcher(BaseWatcher):
    def __init__(self, vault_path: str, credentials_path: str):
        super().__init__(vault_path, check_interval=120)
        self.creds = Credentials.from_authorized_user_file(credentials_path)
        self.service = build('gmail', 'v1', credentials=self.creds)
        self.processed_ids = set()

    def check_for_updates(self) -> list:
        results = self.service.users().messages().list(
            userId='me', q='is:unread is:important'
        ).execute()
        messages = results.get('messages', [])
        return [m for m in messages if m['id'] not in self.processed_ids]

    def create_action_file(self, message) -> Path:
        msg = self.service.users().messages().get(
            userId='me', id=message['id']
        ).execute()

        # Extract headers
        headers = {h['name']: h['value'] for h in msg['payload']['headers']}

        content = f'''---
```

```
type: email
from: {headers.get('From', 'Unknown')}
subject: {headers.get('Subject', 'No Subject')}
received: {datetime.now().isoformat()}
priority: high
status: pending
---

## Email Content
{msg.get('snippet', '')}

## Suggested Actions
- [ ] Reply to sender
- [ ] Forward to relevant party
- [ ] Archive after processing
'''
        filepath = self.needs_action / f'EMAIL_{message["id"]}.md'
        filepath.write_text(content)
        self.processed_ids.add(message['id'])
        return filepath
```

WhatsApp Watcher (Playwright-based)

Note: This uses WhatsApp Web automation. Be aware of WhatsApp's terms of service.

```
# whatsapp_watcher.py
from playwright.sync_api import sync_playwright
from base_watcher import BaseWatcher
from pathlib import Path
import json

class WhatsAppWatcher(BaseWatcher):
    def __init__(self, vault_path: str, session_path: str):
        super().__init__(vault_path, check_interval=30)
        self.session_path = Path(session_path)
        self.keywords = ['urgent', 'asap', 'invoice', 'payment', 'help']

    def check_for_updates(self) -> list:
        with sync_playwright() as p:
            browser = p.chromium.launch_persistent_context(
                self.session_path, headless=True
            )
            page = browser.pages[0]
            page.goto('https://web.whatsapp.com')
            page.wait_for_selector('[data-testid="chat-list"]')

            # Find unread messages
            unread = page.query_selector_all('[aria-label*="unread"]')
```

```
            messages = []
            for chat in unread:
                text = chat.inner_text().lower()
                if any(kw in text for kw in self.keywords):
                    messages.append({'text': text, 'chat': chat})
            browser.close()
            return messages
```

File System Watcher (for local drops)

```
# filesystem_watcher.py
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
from pathlib import Path
import shutil

class DropFolderHandler(FileSystemEventHandler):
    def __init__(self, vault_path: str):
        self.needs_action = Path(vault_path) / 'Needs_Action'

    def on_created(self, event):
        if event.is_directory:
            return
        source = Path(event.src_path)
        dest = self.needs_action / f'FILE_{source.name}'
        shutil.copy2(source, dest)
        self.create_metadata(source, dest)

    def create_metadata(self, source: Path, dest: Path):
        meta_path = dest.with_suffix('.md')
        meta_path.write_text(f'''---
type: file_drop
original_name: {source.name}
size: {source.stat().st_size}
---

New file dropped for processing.
''')
```

B. Reasoning (Claude Code)

When the **Watcher** detects a change, it triggers a Claude command:

6. **Read:** "Check /Needs_Action and /Accounting."
7. **Think:** "I see a WhatsApp message from a client asking for an invoice and a bank transaction showing a late payment fee."
8. **Plan:** Claude creates a Plan.md in Obsidian with checkboxes for the next steps.

C. Action (The "Hands")

Model Context Protocol (MCP) servers are Claude Code's hands for interacting with external systems. Each MCP server exposes specific capabilities that Claude can invoke.

Claude uses custom **MCP (Model Context Protocol)** servers to act:

- **WhatsApp/Social MCP:** To send the reply or post the scheduled update.
- **Browser/Payment MCP:** To log into a payment portal, draft a payment, and stop.
- **Human-in-the-Loop (HITL):** Claude writes a file: APPROVAL_REQUIRED_Payment_Client_A.md. It **will not** click "Send" until you move that file to the /Approved folder.

Recommended MCP Servers

Server	Capabilities	Use Case
filesystem	Read, write, list files	Built-in, use for vault
email-mcp	Send, draft, search emails	Gmail integration
browser-mcp	Navigate, click, fill forms	Payment portals
calendar-mcp	Create, update events	Scheduling
slack-mcp	Send messages, read channels	Team communication

Claude Code Configuration

Configure MCP servers in your Claude Code settings:

```
// ~/.config/claude-code/mcp.json
{
  "servers": [
    {
      "name": "email",
      "command": "node",
      "args": ["/path/to/email-mcp/index.js"],
      "env": {
        "GMAIL_CREDENTIALS": "/path/to/credentials.json"
      }
    },
    {
      "name": "browser",
      "command": "npx",
      "args": ["@anthropic/browser-mcp"],
      "env": {
        "HEADLESS": "true"
      }
```

```
    }
  ]
}
```

Human-in-the-Loop Pattern

For sensitive actions, Claude writes an approval request file instead of acting directly:

```
# When Claude detects a sensitive action needed:
# 1. Create approval request file

# /Vault/Pending_Approval/PAYMENT_Client_A_2026-01-07.md
---
type: approval_request
action: payment
amount: 500.00
recipient: Client A
reason: Invoice #1234 payment
created: 2026-01-07T10:30:00Z
expires: 2026-01-08T10:30:00Z
status: pending
---

## Payment Details
- Amount: $500.00
- To: Client A (Bank: XXXX1234)
- Reference: Invoice #1234

## To Approve
Move this file to /Approved folder.

## To Reject
Move this file to /Rejected folder.
```

The Orchestrator watches the /Approved folder and triggers the actual MCP action when files appear.

D. Persistence (The "Ralph Wiggum" Loop)

Claude Code runs in interactive mode - after processing a prompt, it waits for more input.
To keep your AI Employee working autonomously until a task is complete, use the **Ralph Wiggum pattern**: a Stop hook that intercepts Claude's exit and feeds the prompt back.

How Does It Work?

1. Orchestrator creates state file with prompt
2. Claude works on task
3. Claude tries to exit
4. Stop hook checks: Is task file in /Done?

5. YES → Allow exit (complete)
6. NO → Block exit, re-inject prompt, and allow Claude to see its own previous failed output (loop continues).
7. Repeat until complete or max iterations

Usage

```bash
# Start a Ralph loop
/ralph-loop "Process all files in /Needs_Action, move to /Done when complete" \
  --completion-promise "TASK_COMPLETE" \
  --max-iterations 10
```

Two Completion Strategies:

1. **Promise-based (simple):** Claude outputs `<promise>TASK_COMPLETE</promise>`
2. **File movement (advanced - Gold tier)**: Stop hook detects when task file moves to /Done
    - More reliable (completion is natural part of workflow)
    - Orchestrator creates state file programmatically
    - See reference implementation for details

Reference: https://github.com/anthropics/claude-code/tree/main/.claude/plugins/ralph-wiggum

3. Continuous vs. Scheduled Operations

Operation Type	Example Task	Local Trigger
Scheduled	**Daily Briefing:** Summarize business tasks at 8:00 AM.	cron (Mac/Linux) or Task Scheduler (Win) calls Claude.
Continuous	**Lead Capture:** Watch WhatsApp for keywords like "Pricing."	Python watchdog script monitoring the /Inbox folder.
Project-Based	**Q1 Tax Prep:** Categorize 3 months of business expenses.	Manual drag-and-drop of a file into the /Active_Project folder.

4. Key Hackathon Feature: The "Business Handover"

One of the coolest features you can add is the **Autonomous Business Audit**:

1. **The Trigger:** A scheduled task runs every Sunday night.
2. **The Process:** Claude Code reads your Business_Goals.md, checks your Tasks/Done folder for the week, and checks your Bank_Transactions.md.
3. **The Deliverable:** It writes a "Monday Morning CEO Briefing" in Obsidian, highlighting:
    - **Revenue:** Total earned this week.
    - **Bottlenecks:** Tasks that took too long.
    - **Proactive Suggestion:** "I noticed we spent $200 on software we don't use; shall I cancel the subscription?"

Business Handover Templates

The Business Handover feature transforms your AI Employee from reactive to proactive. Here are the required templates with explicit schemas.

Business_Goals.md Template

```
# /Vault/Business_Goals.md
---
last_updated: 2026-01-07
review_frequency: weekly
---

## Q1 2026 Objectives

### Revenue Target
- Monthly goal: $10,000
- Current MTD: $4,500

### Key Metrics to Track
| Metric | Target | Alert Threshold |
|--------|--------|-----------------|
| Client response time | < 24 hours | > 48 hours |
| Invoice payment rate | > 90% | < 80% |
| Software costs | < $500/month | > $600/month |

### Active Projects
1. Project Alpha - Due Jan 15 - Budget $2,000
2. Project Beta - Due Jan 30 - Budget $3,500

### Subscription Audit Rules
Flag for review if:
- No login in 30 days
- Cost increased > 20%
- Duplicate functionality with another tool
```

Weekly Audit Logic

Claude uses pattern matching to identify subscription usage:

```
# audit_logic.py
SUBSCRIPTION_PATTERNS = {
    'netflix.com': 'Netflix',
    'spotify.com': 'Spotify',
    'adobe.com': 'Adobe Creative Cloud',
    'notion.so': 'Notion',
    'slack.com': 'Slack',
    # Add your common subscriptions
}

def analyze_transaction(transaction):
    for pattern, name in SUBSCRIPTION_PATTERNS.items():
        if pattern in transaction['description'].lower():
            return {
                'type': 'subscription',
                'name': name,
                'amount': transaction['amount'],
                'date': transaction['date']
            }
    return None
```

CEO Briefing Template (Generated Output)

```
# /Vault/Briefings/2026-01-06_Monday_Briefing.md
---
generated: 2026-01-06T07:00:00Z
period: 2025-12-30 to 2026-01-05
---

# Monday Morning CEO Briefing

## Executive Summary
Strong week with revenue ahead of target. One bottleneck identified.

## Revenue
- **This Week**: $2,450
- **MTD**: $4,500 (45% of $10,000 target)
- **Trend**: On track

## Completed Tasks
- [x] Client A invoice sent and paid
- [x] Project Alpha milestone 2 delivered
- [x] Weekly social media posts scheduled

## Bottlenecks
| Task | Expected | Actual | Delay |
```

```
|------|----------|--------|-------|
| Client B proposal | 2 days | 5 days | +3 days |

## Proactive Suggestions

### Cost Optimization
- **Notion**: No team activity in 45 days. Cost: $15/month.
  - [ACTION] Cancel subscription? Move to /Pending_Approval

### Upcoming Deadlines
- Project Alpha final delivery: Jan 15 (9 days)
- Quarterly tax prep: Jan 31 (25 days)

---
*Generated by AI Employee v0.1*
```

5. Tech Stack Summary

- **Knowledge Base:** Obsidian (Local Markdown).
- **Logic Engine:** Claude Code (running claude-4-5-opus or any other LLM using Claude Code Router).
- **External Integration: MCP Servers** (Local Node.js/Python scripts) for Gmail, WhatsApp, and Banking.
    - **Playwright** for "Computer Use" (interacting with websites for payments).
- **Automation Glue:** A master Python Orchestrator.py that handles the timing and folder watching.

6. Security & Privacy Architecture

Security is non-negotiable when building an autonomous system that handles banking, email, and personal communications. This section outlines required security measures.

6.1 Credential Management

Never store credentials in plain text or in your Obsidian vault.

- Use environment variables for API keys: export GMAIL_API_KEY="your-key"
- For banking credentials, use a dedicated secrets manager (e.g., macOS Keychain, Windows Credential Manager, or 1Password CLI)
- Create a .env file (add to .gitignore immediately) for local development
- Rotate credentials monthly and after any suspected breach

Example .env structure:

```
# .env - NEVER commit this file
GMAIL_CLIENT_ID=your_client_id
GMAIL_CLIENT_SECRET=your_client_secret
BANK_API_TOKEN=your_token
WHATSAPP_SESSION_PATH=/secure/path/session
```

6.2 Sandboxing & Isolation

Protect against unintended actions during development:

- Development Mode: Create a DEV_MODE flag that prevents any real external actions
- Dry Run: All action scripts should support a --dry-run flag that logs intended actions without executing
- Separate Accounts: Use test/sandbox accounts for Gmail and banking during development
- Rate Limiting: Implement maximum actions per hour (e.g., max 10 emails, max 3 payments)

Example dry-run implementation:

```
# In any action script
DRY_RUN = os.getenv('DRY_RUN', 'true').lower() == 'true'

def send_email(to, subject, body):
    if DRY_RUN:
        logger.info(f'[DRY RUN] Would send email to {to}')
        return
    # Actual send logic here
```

6.3 Audit Logging

Every action the AI takes must be logged for review:

```
# Required log format
{
  "timestamp": "2026-01-07T10:30:00Z",
  "action_type": "email_send",
  "actor": "claude_code",
  "target": "client@example.com",
  "parameters": {"subject": "Invoice #123"},
  "approval_status": "approved",
  "approved_by": "human",
  "result": "success"
}
```

Store logs in /Vault/Logs/YYYY-MM-DD.json and retain for a minimum 90 days.

6.4 Permission Boundaries

Action Category	Auto-Approve Threshold	Always Require Approval
Email replies	To known contacts	New contacts, bulk sends
Payments	< $50 recurring	All new payees, > $100
Social media	Scheduled posts	Replies, DMs
File operations	Create, read	Delete, move outside vault

7. Error States & Recovery

Autonomous systems will fail. Plan for it. This section covers common failure modes and recovery strategies.

7.1 Error Categories

Category	Examples	Recovery Strategy
Transient	Network timeout, API rate limit	Exponential backoff retry
Authentication	Expired token, revoked access	Alert human, pause operations
Logic	Claude misinterprets message	Human review queue
Data	Corrupted file, missing field	Quarantine + alert
System	Orchestrator crash, disk full	Watchdog + auto-restart

7.2 Retry Logic

```
# retry_handler.py
import time
from functools import wraps

def with_retry(max_attempts=3, base_delay=1, max_delay=60):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except TransientError as e:
                    if attempt == max_attempts - 1:
                        raise
                    delay = min(base_delay * (2 ** attempt), max_delay)
                    logger.warning(f'Attempt {attempt+1} failed, retrying in
{delay}s')
```

```
                    time.sleep(delay)
        return wrapper
    return decorator
```

7.3 Graceful Degradation

When components fail, the system should degrade gracefully:

- Gmail API down: Queue outgoing emails locally, process when restored
- Banking API timeout: Never retry payments automatically, always require fresh approval
- Claude Code unavailable: Watchers continue collecting, queue grows for later processing
- Obsidian vault locked: Write to temporary folder, sync when available

7.4 Watchdog Process

```
# watchdog.py - Monitor and restart critical processes
import subprocess
import time
from pathlib import Path

PROCESSES = {
    'orchestrator': 'python orchestrator.py',
    'gmail_watcher': 'python gmail_watcher.py',
    'file_watcher': 'python filesystem_watcher.py'
}

def check_and_restart():
    for name, cmd in PROCESSES.items():
        pid_file = Path(f'/tmp/{name}.pid')
        if not is_process_running(pid_file):
            logger.warning(f'{name} not running, restarting...')
            proc = subprocess.Popen(cmd.split())
            pid_file.write_text(str(proc.pid))
            notify_human(f'{name} was restarted')

while True:
    check_and_restart()
    time.sleep(60)
```

Learning Material To Get Started:

These resources provide a foundational guide on how to integrate Claude Code agentic capabilities with local tools and file systems, which is the exact "foundation layer" required for your personal employee project.

Claude Code Chapter of our Textbook
https://agentfactory.panaversity.org/docs/AI-Tool-Landscape/claude-code-features-and-workflows

Turning Claude Code into an Employee
https://www.facebook.com/reel/1521210822329090

Claude Code and Obsidian for Personal Automation
https://www.youtube.com/watch?v=sCIS05Qt79Y

Claude Agent Skills - Automate Your Workflow Fast
https://www.youtube.com/watch?v=nbqqnl3JdR0

Claude Code just Built me an AI Agent Team (Claude Code + Skills + MCP)
https://www.youtube.com/watch?v=0J2_YGuNrDo

Why Odoo (Value-for-Money ERP Perspective)?

https://chatgpt.com/share/6967deaf-9404-8001-9ad7-03017255ebaf

Odoo Official Documentation (Community Edition)

https://www.odoo.com/documentation

Odoo 19 External JSON-2 API (recommended for your Odoo 19+ MCP integration):

https://www.odoo.com/documentation/19.0/developer/reference/external_api.html

Curated resources organized by learning stage. Start with Prerequisites, then progress through each level.

Prerequisites (Complete Before Hackathon)

Topic	Resource	Time
Presentation	https://docs.google.com/presentation/d/1UGvCUk1-O8m5i-aTWQNxzg8EXoKzPa8fgcwfNh8vRjQ/edit?usp=sharing	2 hours
Claude Code Fundamentals	https://agentfactory.panaversity.org/docs/AI-Tool-Landscape/claude-code-features-and-workflows	3 hour
Obsidian Fundamentals	help.obsidian.md/Getting+started	30 min
Python File I/O	realpython.com/read-write-files-python	1 hour
MCP Introduction	modelcontextprotocol.io/introduction	1 hour

Agent Skills	platform.claude.com/docs/en/agents-and-tools/agent-skills/overview	2 hours

Core Learning (During Hackathon)

Topic	Resource	Type
Claude + Obsidian Integration	youtube.com/watch?v=sCIS05Qt79Y	Video
Building MCP Servers	modelcontextprotocol.io/quickstart	Tutorial
Claude Agent Teams	youtube.com/watch?v=0J2_YGuNrDo	Video
Gmail API Setup	developers.google.com/gmail/api/quickstart	Docs
Playwright Automation	playwright.dev/python/docs/intro	Docs

Deep Dives (Post-Hackathon)

- MCP Server Development: github.com/anthropics/mcp-servers (reference implementations)
- Production Automation: "Automate the Boring Stuff with Python" (free online book)
- Security Best Practices: OWASP API Security Top 10
- Agent Architecture: "Building LLM-Powered Applications" by Anthropic

Hackathon Rules & Judging Criteria

Participation Rules

1. Individual
2. All code must be original or properly attributed open-source
3. Must use Claude Code as the primary reasoning engine
4. Projects must include documentation and a demo video

Judging Criteria

Criterion	Weight	Description
Functionality	30%	Does it work? Are core features complete?
Innovation	25%	Creative solutions, novel integrations
Practicality	20%	Would you actually use this daily?
Security	15%	Proper credential handling, HITL safeguards
Documentation	10%	Clear README, setup instructions, demo

Submission Requirements

- GitHub repository (public or private with judge access)
- README.md with setup instructions and architecture overview
- Demo video (5-10 minutes) showing key features
- Security disclosure: How credentials are handled
- Tier declaration: Bronze, Silver, or Gold
- Submit Form: https://forms.gle/JR9T1SJq5rmQyGkGA

Example: End-to-End Invoice Flow

This walkthrough demonstrates a complete flow from trigger to action, showing how all components work together.

Scenario

A client sends a WhatsApp message asking for an invoice. The AI Employee should: (1) detect the request, (2) generate the invoice, (3) send it via email, and (4) log the transaction.

Step 1: Detection (WhatsApp Watcher)

The WhatsApp Watcher detects a message containing the keyword "invoice":

```
# Detected message:
# From: Client A
# Text: "Hey, can you send me the invoice for January?"

# Watcher creates:
# /Vault/Needs_Action/WHATSAPP_client_a_2026-01-07.md
```

Step 2: Reasoning (Claude Code)

The Orchestrator triggers Claude to process the Needs_Action folder:

```
# Claude reads the file and creates:
# /Vault/Plans/PLAN_invoice_client_a.md

---
created: 2026-01-07T10:30:00Z
status: pending_approval
---

## Objective
Generate and send January invoice to Client A

## Steps
- [x] Identify client: Client A (client_a@email.com)
- [x] Calculate amount: $1,500 (from /Accounting/Rates.md)
- [ ] Generate invoice PDF
- [ ] Send via email (REQUIRES APPROVAL)
- [ ] Log transaction

## Approval Required
Email send requires human approval. See /Pending_Approval/
```

Step 3: Approval (Human-in-the-Loop)

Claude creates an approval request:

```
# /Vault/Pending_Approval/EMAIL_invoice_client_a.md
---
action: send_email
to: client_a@email.com
subject: January 2026 Invoice - $1,500
attachment: /Vault/Invoices/2026-01_Client_A.pdf
---

Ready to send. Move to /Approved to proceed.
```

You review and move the file to /Approved.

Step 4: Action (Email MCP)

The Orchestrator detects the approved file and calls the Email MCP:

```
# MCP call (simplified)
await email_mcp.send_email({
  to: 'client_a@email.com',
  subject: 'January 2026 Invoice - $1,500',
  body: 'Please find attached your invoice for January 2026.',
  attachment: '/Vault/Invoices/2026-01_Client_A.pdf'
});

# Result logged to /Vault/Logs/2026-01-07.json
```

Step 5: Completion

Claude updates the Dashboard and moves files to Done:

```
# /Vault/Dashboard.md updated:
## Recent Activity
- [2026-01-07 10:45] Invoice sent to Client A ($1,500)

# Files moved:
# /Needs_Action/WHATSAPP_... -> /Done/
# /Plans/PLAN_invoice_... -> /Done/
# /Approved/EMAIL_... -> /Done/
```

Troubleshooting FAQ

Setup Issues

Q: Claude Code says "command not found"

A: Ensure Claude Code is installed globally and your PATH is configured. Run: npm install -g @anthropic/claude-code, then restart your terminal.

Q: Obsidian vault isn't being read by Claude

A: Check that you're running Claude Code from the vault directory, or using the --cwd flag to point to it. Verify file permissions allow read access.

Q: Gmail API returns 403 Forbidden

A: Your OAuth consent screen may need verification, or you haven't enabled the Gmail API in Google Cloud Console. Check the project settings.

Runtime Issues

Q: Watcher scripts stop running overnight

A: Use a process manager like PM2 (Node.js) or supervisord (Python) to keep them alive. Alternatively, implement the Watchdog pattern from Section 7.

Q: Claude is making incorrect decisions

A: Review your Company_Handbook.md rules. Add more specific examples. Consider lowering autonomy thresholds so more actions require approval.

Q: MCP server won't connect

A: Check that the server process is running (ps aux | grep mcp). Verify the path in mcp.json is absolute. Check Claude Code logs for connection errors.

Security Concerns

Q: How do I know my credentials are safe?

A: Never commit .env files. Use environment variables. Regularly rotate credentials. Implement the audit logging from Section 6 to track all access.

Q: What if Claude tries to pay the wrong person?

A: That's why HITL is critical for payments. Any payment action should create an approval file first. Never auto-approve payments to new recipients.

Ethics & Responsible Automation

With great automation comes great responsibility. Consider these principles as you build.

When Should AI NOT Act Autonomously?

- Emotional contexts: Condolence messages, conflict resolution, sensitive negotiations
- Legal matters: Contract signing, legal advice, regulatory filings
- Medical decisions: Health-related actions affecting you or others
- Financial edge cases: Unusual transactions, new recipients, large amounts
- Irreversible actions: Anything that cannot be easily undone

Transparency Principles

- Disclose AI involvement: When your AI sends emails, consider adding a signature noting AI assistance
- Maintain audit trails: All actions should be logged and reviewable
- Allow opt-out: Give contacts a way to request human-only communication
- Regular reviews: Schedule weekly reviews of AI decisions to catch drift

Privacy Considerations

9. Minimize data collection: Only capture what's necessary
10. Local-first: Keep sensitive data on your machine when possible
11. Encryption at rest: Consider encrypting your Obsidian vault
12. Third-party caution: Understand what data leaves your system via APIs

The Human Remains Accountable

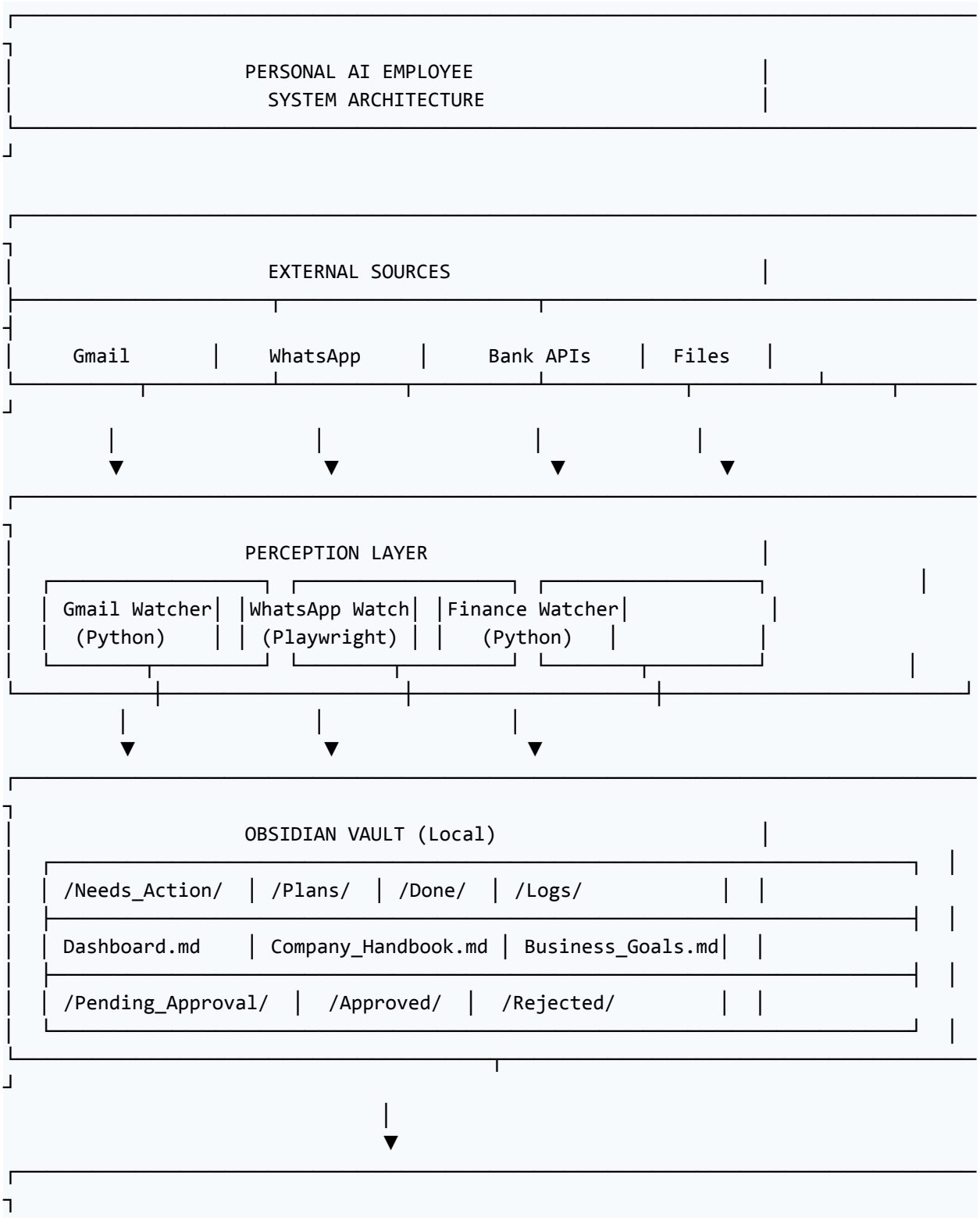
Remember: You are responsible for your AI Employee's actions. The automation runs on your behalf, using your credentials, acting in your name. Regular oversight isn't optional—it's essential.

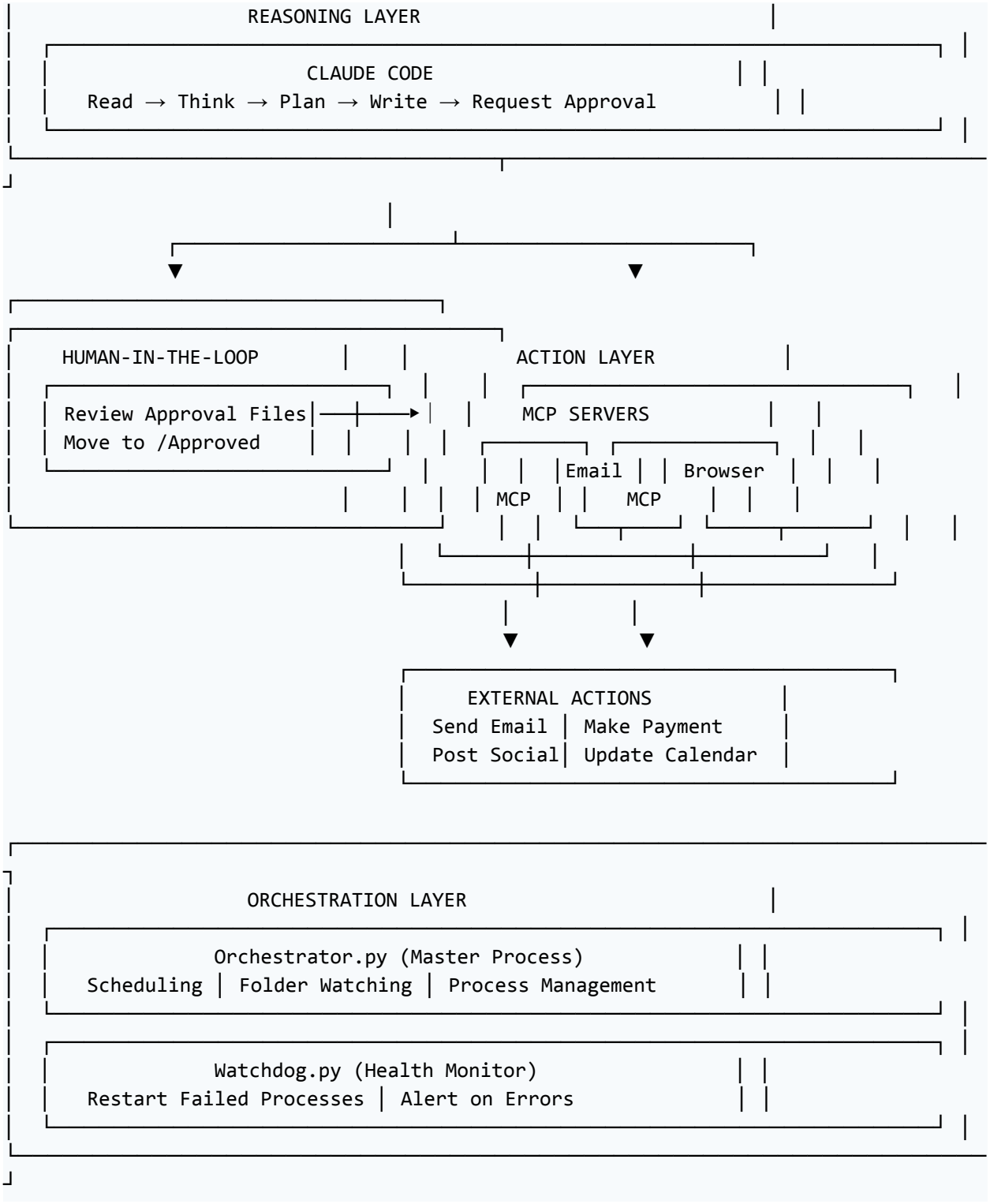
Suggested oversight schedule:

1. Daily: 2-minute dashboard check
2. Weekly: 15-minute action log review
3. Monthly: 1-hour comprehensive audit
4. Quarterly: Full security and access review

Architecture Diagram

The following ASCII diagram illustrates the complete system architecture:





Core Strengths:

Local-First: Privacy-centric architecture using Obsidian.

HITL Safety: Sophisticated file-based approval system prevents AI accidents.

Note for Developers: Why "Watchers" Need Process Management

In the proposed architecture, your "Watchers" (Gmail, WhatsApp listeners) are essentially **daemon processes**. They are designed to run indefinitely to poll for events[1].

However, standard Python scripts invoked via terminal (e.g., python watcher.py) are fragile:

- They terminate if the TTY/SSH session closes.
- They crash on unhandled exceptions (e.g., transient API timeouts).
- They do not auto-recover after a system reboot.

"Process Management" solves this by wrapping your scripts in a supervisor that ensures state persistence.

The Problem: Script Fragility

If you run python gmail_watcher.py and your internet blips for 5 seconds, the script throws an exception and exits. Your AI employee is now "dead" until you manually SSH in and restart it.

The Solution: A Process Manager (PM)

A PM (like **PM2**, **supervisord**, or **Systemd**) acts as a watchdog. It daemonizes your script and monitors its PID.

- **Auto-Restart:** If the process exits with a non-zero code (crash), the PM immediately restarts it[2].
- **Startup Persistence:** It hooks into the OS init system (e.g., systemd on Linux) to launch the script on boot[3].
- **Logging:** It captures stdout/stderr to log files, which is critical for debugging silent failures over long periods.

Quick Recommendation:
For this hackathon, PM2 is often the easiest developer-friendly tool (originally for Node, but handles Python perfectly):

```
# Install PM2
npm install -g pm2

# Start your watcher and keep it alive forever
pm2 start gmail_watcher.py --interpreter python3

# Freeze this list to start on reboot
pm2 save
```

`pm2 startup`

Alternatively, the hackathon document suggests writing a custom Python "Watchdog" script that loops and checks PIDs, effectively building a primitive process manager yourself.

Next Step: Advanced Custom Cloud FTE Architecture

Once you have built this local AI Employee, you can shift to building cloud based custom FTEs:

https://docs.google.com/document/d/15GuwZwIOQy_g1XsIJjQsFNHCTQTWoXQhWGVMhiH0swc/edit?usp=sharing