ChatGPT

Daily Accomplishments Tracker – Integration Plan & Analysis

1. Notebook Summary

Pipeline Overview: The Colab notebook implements a prototype pipeline for the Daily Accomplishments Tracker. It reads a day's raw event log (JSON Lines under `logs/daily/` – each line is a JSON event [1]) and performs several analytic steps:

- **Event Parsing & Session Inference:** It traverses chronological `focus_change` events (app usage with durations) to build contiguous **focus sessions**. When gaps >5 minutes occur, the current session ends; sessions ≥25 minutes are flagged as **deep work** [2] [3]. Each session tracks start/end times, total duration, the app used, and any **interruptions** (e.g. `app_switch` or `window_change` events during the session) [4]. This identifies periods of uninterrupted work (the notebook uses a 25 min threshold by default).

- **Categorization:** Each focus event's application is mapped to an activity category (e.g. "Google Chrome" → *Research*, VS Code → *Coding*, Zoom/Teams → *Meetings*, email apps → *Communication*, Office/Notion → *Docs*, etc.) [5]. The notebook tallies total time per category and computes percentages [6] [7]. This yields a **category breakdown** (how the day's active time is distributed).

- **Interruptions & Context Switching:** It scans for context switch events (`app_switch`/`window_change`) to count **interruptions per hour** and total interruptions [8] [9]. The most disruptive hour is identified [10]. An assumed **context switch cost** (e.g. 60 seconds per interruption) is applied to estimate time lost to switching [11].

- **Meeting Analysis:** It captures `meeting_end` events to total **meeting time**, count meetings, and compute average meeting length [12] [13]. It then compares meeting vs. focus time to produce a **meeting/focus ratio** [14] and a simple recommendation (e.g. "Too many meetings – consider declining some" if >50% of time in meetings) [15].

- **Productivity Metrics:** Using the above data, it computes an **overall productivity score (0–100)**, broken into components [16] [17]: deep work (max 40 pts), interruptions (30 pts), and session quality (30 pts). Deep work points scale with the percentage of work time spent in deep sessions [18]; interruption points start at 30 and subtract 1 per interruption (floored at 0) [19]; quality points come from the average session quality scores (which in turn penalize interruptions and short duration [20] [21]). The score is returned with an adjectival **rating** ("Excellent" ≥80, etc.) [22].

- **Focus Window Suggestions:** Based on the hourly interruption counts, the notebook suggests **low-interruption time windows** ideal for deep work [23] [24]. Consecutive quiet hours (with ≤2 interrupts/hour) are grouped into 2h+ blocks and recommended for scheduling focus time [25] [26] (e.g. "Schedule deep work during 09:00–11:00").

- **Output Generation:** Finally, the prototype compiles a **daily report object** with all key results: date, list of deep work sessions, interruption stats, productivity score (with components and ratings), category breakdown, meeting stats, and suggested focus windows [27]. In the notebook, this was likely printed or examined directly; in production it's saved to structured files.

Prototype vs Production: In the notebook, much of this logic was implemented imperatively for a single day (for example, building sessions with loops and printing results for that day). In production, it has been refactored into reusable modules and classes. For instance, the notebook's session parsing became the `ProductivityAnalytics.detect_deep_work_sessions` method [28] [29] and related helpers in a Python module. The prototype likely used hardcoded parameters (25-min deep work threshold, 5-min interruption window, 60s context switch penalty) and inline logic. In the integration, these are encapsulated (and can be made configurable via `config.json`). The output in the notebook was a Python dict or DataFrame; in production it's written to JSON (for the dashboard) and Markdown (for emails/reading).

Dependencies & Environment: The analytic code relies on Python 3 standard libraries (no heavy external dependencies like pandas, to keep it lightweight). It uses `datetime` and Python's `zoneinfo` for timezone awareness [30], ensuring logs are interpreted in the configured zone. The logging module is used for debug/info outputs [31]. For full functionality, certain platform-specific features are optional – e.g. an `IdleDetector` class uses OS-specific calls to detect inactivity (macOS ioreg, Windows `GetLastInputInfo`, etc.) [32] [33], and the `daily_logger` uses file locks (fcntl) for safe logging on Unix [34] [35]. The front-end is purely static: an HTML/JS dashboard that includes Chart.js [36], so no server runtime is needed beyond serving static files. The environment should have Python 3.9+ (for `zoneinfo`) and any optional notifiers (SMTP for email, or a Slack webhook URL in config if Slack notifications are enabled). In summary, the notebook's logic has been translated into a maintainable Python toolchain and static web app, with minimal external libraries and mindful of cross-platform deployment (local or simple static hosting).

2. Repo Integration Plan

To integrate the notebook's prototype logic into a production-ready repository, we will create or update several components under a structured `/tools` directory, ensure configuration-driven behavior, and wire up a daily automation. Key steps:

- **Modular Analytics Engine:** Encapsulate the notebook's analysis in a new module, e.g. `tools/analytics.py`, defining a `ProductivityAnalytics` class [37]. This class will load a day's events (using log utilities) and provide methods to generate each analytic output (deep work sessions, scores, etc.). The notebook's functions (session parsing, scoring, etc.) become class methods (as already done for `detect_deep_work_sessions`, `analyze_interruptions`, `calculate_productivity_score`, etc. in the repo [28] [38]). This separation makes the logic testable and reusable.

- **Daily Log Integration:** Use the logging subsystem (`tools/daily_logger.py`) to read events from the JSONL log for the target date. The `ProductivityAnalytics` class calls `read_daily_log(date)` to fetch events [39] [40]. This ensures the analysis runs on the same data the tracker records, and honors configured log paths and formats. (If needed, `daily_logger.py` can also provide utility to fall back to old JSON report formats for backwards compatibility [41], though primary input is the daily `.jsonl`.)

- **Config-Driven Parameters:** Leverage `config.json` for user-tunable weights and settings. For example, the deep work threshold (25 min), interruption cost (60s), "meeting credit", idle timeout, etc., can all be defined in config. The integration will update `config.json` (or its example) under a new **analytics** section, e.g.:

```
"analytics": {
    "deep_work_threshold": 25,
    "interruption_cost_sec": 60,
    "meeting_credit": 0.25
}
```

  The code will load these via the existing `load_config()` utility [42] [43]. For instance, `ProductivityAnalytics.__init__` will set `self.deep_work_threshold_minutes` from config instead of a hardcoded 25. Similarly, if "meeting_credit" is set (e.g. 0.5 meaning count 50% of meeting time as productive), the analytics or UI will factor that in (more on this below). This ensures the logic respects user preferences (like how meeting time should count, etc.).

- **Report Generation Script:** Create a CLI tool (if not already present) for generating reports on schedule. We have `tools/auto_report.py` (or `generate_reports.py`) for this purpose. This script will instantiate the analytics class for the given date and produce outputs:

- **JSON Report:** A structured JSON file (one per day) written to `reports/` (e.g. `reports/daily-report-YYYY-MM-DD.json`). The script uses `ProductivityAnalytics.generate_report()` to get a dict and dumps it to JSON [44]. The JSON includes everything needed for the dashboard (deep work sessions list, category breakdown, etc.). We'll use the naming convention already in use (the repo appears to have shifted to `daily-report-*.json` as the output name [45], with legacy support for `ActivityReport-*.json` for older data).
- **Markdown Summary:** A Markdown file (e.g. `daily-report-YYYY-MM-DD.md`) for the same day [46]. This contains a human-readable report – overall score, key metrics, list of sessions, category times, interruptions, meeting stats, and focus window suggestions – formatted with headings and bullet points [47] [48]. This summary is used for emails or quick reading without the dashboard. (The notebook likely printed similar info; now it's saved to a file.)
- The CLI script can also support weekly or monthly rollups. In fact, we see a weekly report function that aggregates 7 days of trends and outputs a `weekly-report-YYYY-MM-DD.json/md` [49] [50]. We will include that as well, so the tool can generate both daily and weekly reports on demand or schedule.

- This script should be runnable manually and via cron. For example, running `python3 tools/auto_report.py --date 2025-12-05` will generate the Dec 5 report. If no date is given, it defaults to today's date (local timezone) [51]. We'll document adding a cronjob to run this nightly (e.g., 11:55pm daily) or a LaunchAgent as needed [52]. Optionally, a GitHub Actions workflow (as hinted by `generate_and_publish_reports.yml`) can run it and push results to a `gh-pages` branch for web hosting.

- **Dashboard Data Loading:** Ensure the static dashboard continues to load reports from `/reports/`. The existing `dashboard.html` uses client-side JS to fetch the JSON for the selected date (trying `/ActivityReport-date.json`, then `/reports/daily-report-date.json`, then a GitHub raw URL [53] [54]). We will keep this approach, so no backend is needed. We must confirm the naming alignment: after integration, our JSON files will be named `daily-report-YYYY-MM-DD.json`, which the UI will find on the second attempt. (We might consider also saving a copy with the legacy name `ActivityReport-YYYY-MM-DD.json` for backwards compatibility, or simply adjust the UI to try the new name first. A simple update would be to prefer `/reports/daily-report-*.json` as primary and then fallback to `ActivityReport-*.json` if needed.)

- **Maintaining Privacy by Default:** The integration will ensure that no sensitive details are exposed on the dashboard. The raw logs might include window titles or URLs, but our analytics output deliberately omits or aggregates those. For example, the category analysis only displays app names and categories, not full window titles. The deep work sessions list shows duration, start time, and app name – no filenames or specific document titles. Meeting names are collected but currently only the count and total time are shown (the Markdown summary lists the meeting count and total minutes, not each name). We will continue this approach. If detailed content is ever included (e.g. listing top websites visited), we'll anonymize or summarize (like showing domains rather than full URLs). This way the dashboard can be shared or viewed in public without revealing private info.

- **Local and Deploy Targets:** The system will run locally with minimal setup. Users can run a local HTTP server (`python3 -m http.server`) in the repo root and view `dashboard.html` at `http://localhost:8000/dashboard.html` [55]. Because all data is pre-generated and stored in the repository (or a mounted volume), the dashboard is a static SPA. For potential deployment, we can host the `dashboard.html` and `reports/` files on GitHub Pages or a simple object storage. Docker/Railway integration would likely involve a container that runs the logging and report generation on a schedule and serves the static files. Our plan does not require dynamic server code for the core functionality, which keeps hosting simple. (If using Railway, one could serve the static dashboard via a lightweight web server in the container, and still rely on cron or a scheduled GitHub Action to update reports daily.)

In summary, the plan is to refactor the one-off notebook code into clean Python modules (`analytics.py` for analysis logic, `daily_logger.py` for log handling, `auto_report.py` for orchestration), leverage `config.json` for settings like thresholds and timezone [30], and use the existing static dashboard to visualize results. We will also incorporate quality-of-life improvements in the UI (discussed next) to polish the user experience.

3. Git-Style Change List

Below is a list of proposed changes and additions to integrate the notebook code into the repo, with each file's path, the nature of change, rationale, and estimated effort:

- **A** `tools/analytics.py`**:** *Add* – **New analytics module** that contains the `ProductivityAnalytics` class and related functions. Integrates all core logic from the notebook: session building, deep work detection, scoring, category grouping, etc. Effort: **L (Large)** – ~500 lines from scratch (though largely adapted from notebook logic) [37] [27] .

- **A** `tools/auto_report.py`**:** *Add/Replace* – **Automated report generator** CLI. Uses `ProductivityAnalytics` to create daily JSON and Markdown reports, and a weekly summary. It parses command-line args (date, type of report) and writes to `/reports`. This may replace an earlier `generate_reports.py`. Effort: **M** (Medium) – ~300 lines for CLI + formatting (a significant but straightforward coding task) [56] [57] .

- **M** `tools/daily_logger.py`**:** *Modify* – **Integrate config and ensure log reading**. Update or verify that `load_config()` loads tracking settings (start hour, reset hour, timezone) [58] and that `read_daily_log()` returns events as a list of dicts for the analytics module. Possibly add logic to parse existing JSON files if needed (for older data or compatibility). Also confirm that file paths (`logs/daily/` etc.) are taken from config or defaults [43] . Effort: **S** (Small) – minor tweaks since logging is largely implemented.

- **M** `config.json` **(&** `config.json.example`**):** *Modify/Add* – **Add analytics settings**. Include keys for `deep_work_threshold_minutes`, `idle_threshold_seconds`, `context_switch_cost_seconds`, and `meeting_credit` under an `"analytics"` section (as described above). Also ensure the `"tracking"` section has `timezone`, `daily_start_hour`, and other existing keys set correctly (these likely already exist given the Setup guide [59]). Effort: **S** – just a few lines in the example config, and documentation comments if needed.

- **M** `dashboard.html`**:** *Modify* – **Enhance front-end UX.**

- Implement a visible loading indicator and error message: currently, the script inserts "Loading…" text on load and logs errors to console [60] [61] . We will refine this by adding a spinner or at least a stylized message, and ensure that if a report isn't found, an obvious **"No report available"** message is shown in the UI (the code now throws an error which we catch and replace with a message [62] , which we can style as a noticeable banner).
- Add an **empty state** for charts: e.g., if a category has no data or no deep work sessions occurred, display a "No data" note instead of an empty chart. The Markdown and JS already handle some (e.g., list "*No deep work sessions detected*" if none [63] ; we'll ensure similar user-friendly placeholders in the dashboard interface).
- **Responsive design tweaks:** Use CSS media queries to adjust the layout on mobile (smaller padding on `.container`, perhaps stack metrics vertically, font-size adjustments for readability on small screens). This is polish to Tier 2/3 (see below), but we can include basic improvements now.

- Maintain privacy: double-check that no potentially sensitive text is unintentionally inserted. For instance, ensure that if `data.raw_events` exists, we don't display raw event content on the dashboard (currently we do not – raw_events are only counted [64]). Effort: **M** – a moderate update, mostly front-end (HTML/CSS/JS changes, ~50 lines).

- **A `tools/notifications.py`:** *Add/Verify* – **Email/Slack notification sender.** If not already implemented, create a module to email the Markdown report and/or post highlights to Slack. The README and config mention email (SMTP) and Slack webhook integration [65] . We'll implement functions to read the latest report and send it. (The repository lists a `notifications.py` of ~500 lines, so this may already exist – in which case we ensure it works with our new report format.) Effort: **M** – using Python's smtplib for email and `requests` or `http.client` for Slack webhook, mostly straightforward but important to test.

- **M `README.md` and docs:** *Modify* – **Update documentation** to reflect the new integration. This includes instructions to run the daily report generator, notes on config options (like how to set your timezone or Slack webhook in `config.json`), and possibly a brief "how it works" summary. Also update any quickstart or setup guides (e.g., if previously one had to manually run the notebook, now they can run the CLI) [66] [67] . Effort: **S** – textual changes.

- **M `tests/…`:** *Modify/Add* – **Adjust unit tests** to cover the analytics logic. The repository has some test stubs (e.g., `tests/test_meeting_attribution.py`, `tests/test_timeline_edgecases.py` as seen in the file listing [68]). We will expand tests to validate that:

- deep work detection correctly identifies sessions given a synthetic log,
- interruption counts match expectations,
- meeting ratio and recommendations are correct for sample data,
- the productivity score logic yields expected values for edge cases (e.g., zero deep work vs. lots of deep work). Effort: **M** – writing a few tests per analytic function.

Each change above will be committed with a clear message (e.g., "Add analytics module for deep work & interruptions analysis" etc.). After these changes, the codebase will encapsulate the notebook's functionality into maintainable code, with daily automation and an improved user interface.

4. UI Upgrade Recommendations

We propose a three-tier plan to polish the dashboard UI, enhancing usability while keeping it simple (vanilla JS + Chart.js):

Tier 1 (Essential UX fixes – quick wins):

- **Loading/Feedback States:** Improve the *loading indicator* and *error messages*. Currently, when switching dates, the dashboard replaces content with a "Loading your day..." text [60] . We can style this as a spinner or at least center it and use a muted text color. For example: adding a CSS rule for a `.loading` class to animate a spinner icon or dot pulses, and wrapping the text with that class. Similarly, on data fetch failure, the code now injects a message with class `.error` [69] ; we should style this in CSS (e.g. red or warning-colored text, maybe an icon like 👎) and possibly keep it visible until the user picks a different date. These changes make it obvious when data is loading or unavailable, rather than the user staring at an empty page or having to check the console.

- **Empty State Messaging:** Ensure that sections with no data display a friendly note. The deep work section already does this (“No deep work blocks detected” if none) [70]. We should audit other sections: e.g., if there were zero interruptions (unlikely in practice), or no meetings that day, we can show “0 interruptions” or “No meetings logged” clearly. The Category chart, if a user had no activity (perhaps a day off), might currently just be blank; we can detect if `data.category_trends.categories` is empty and overlay a “No activity logged for this day” message or icon on the chart canvas. Chart.js allows rendering of custom text when data arrays are empty – we could implement a plugin or simply not draw the chart and show text instead. This guides the user rather than leaving them guessing if something broke.

- **Basic Responsiveness:** Apply simple CSS tweaks for smaller screens. For instance, on narrow viewports (< 600px), convert the header and metrics layout to a single-column or wrapped layout. We can add a media query to increase font sizes for the score and metrics on mobile for readability and stack the “Score / Deep Work / Focus % / Interruptions” metrics vertically instead of in a row. Also, ensure charts canvas resize properly (Chart.js v3 is responsive by default, but we should give them a `%` or fixed max-width). These adjustments (tested via Chrome dev tools on a phone screen size) will make the dashboard usable on mobile devices when served via GitHub Pages or similar.

- **Privacy Check in UI:** Though our data is already sanitized, we double-check the UI doesn’t inadvertently expose any raw logs. One item to consider is the “raw data” link (the JSON link). Currently, the dashboard includes a link to the JSON source used (`jsonLink` in code) [71]. This is useful for transparency, but if logs contained sensitive info, a user clicking it would see the raw JSON. To preserve privacy-by-default, we might hide or remove this link in the public dashboard, or replace it with a “Download Data” button that perhaps removes or masks detailed fields. This is a minor UI decision: since our JSON doesn’t have full content of every event (just counts and summaries), it’s probably fine to keep for power users, but we mention it so the user is aware.

Tier 2 (Enhancements – nice to have):

- **Historical Navigation & Date Picker Improvements:** The date navigation (likely arrows to previous/next day and a date input) works, but could be improved. For example, disabling the “next day” arrow when viewing today (since no future data) to avoid an error fetch, or providing a calendar picker UI for jumping to a specific date. Since the data is static, we know the range of dates available (we could even list all dates that have reports). A future enhancement: a dropdown of available dates or a small calendar highlighting days with data. This avoids typing the date manually. This is more involved (would require generating that UI, perhaps from the reports folder listing via an index or an injected JSON of available dates), so we mark it as a second-tier improvement.

- **Interactive Charts & Details:** Utilize more of Chart.js’s capabilities for a richer experience. For example, add tooltips that show exact values on hover (if not already enabled), or make the category chart segments clickable to show sub-breakdowns (perhaps not applicable if we only have broad categories). Another idea: an **hourly timeline** visualization (the README mentioned an SVG timeline with deep work overlays). We could implement a horizontal bar showing 24h with colored segments for each category or deep work period. This could be drawn using HTML5 Canvas or SVG and the data we have (focus sessions with start/end). It provides a quick glance of the day's structure. This is a new component that would take some effort (probably a Tier 2/3 item, depending on complexity), but would greatly enhance the visual appeal and at-a-glance information.

- **Customize Meeting Credit Visualization:** Currently, the dashboard's JS uses a `window.meetingCredit` (default 0.25) to adjust focus time calculations in `analyzeDay()` [72]. We could expose this as a UI control (e.g., a slider or toggle saying "Count X% of meeting time as focus time"). This would let users see the impact of counting more or less meeting time as productive. It's a bit advanced and possibly overkill for most, but it's a thought for power users. At minimum, we ensure the front-end uses the same value as in config (perhaps even output the chosen meeting credit in the JSON so the JS can pick it up instead of a hardcoded 0.25).

Tier 3 (Polish & Advanced Features):

- **User Personalization and Settings:** Provide a way for users to tweak settings from the UI. For instance, toggling "show all data" vs "privacy mode" (where privacy mode might hide specific app names or meeting names – if one wanted to publicly screenshot the dashboard). Another setting could be the above-mentioned meeting credit slider, or selecting the deep work threshold (e.g., highlight sessions > 50 min as deep work instead of 25). These could be stored in localStorage so the preferences persist for that user. Implementing this moves the dashboard from a read-only view into a slightly interactive app.

- **Visual Theme and Layout Polish:** Further refine the aesthetic – e.g., add some animations when transitioning between days (fade out old data, fade in new data once loaded), use icons or emojis consistently (the text reports use 📊, ⏱️, , etc., we can mirror some of that in the web UI for fun), and improve the layout of the session list. Perhaps use a table or grid for session start/duration/quality rather than a plain list, to align things. Ensuring the design matches modern dashboard look-and-feel (the current dark theme with gradient accents is a great start [73], we'll continue with that style).

- **Responsive Charts and Tooltip Placement:** For very small screens, the charts might need resizing or toggling. For example, showing either the bar chart or pie chart one at a time with tabs on mobile, instead of both side by side, might be more legible. Also, ensure tooltips don't overflow the viewport (Chart.js usually handles this, but worth verifying).

- **Future: Live Updates and Real-Time Mode:** While outside the scope of static integration, in the future one could load the current day's data periodically (the README mentions auto-refresh every 5 minutes [74]). Achieving this statically might involve the tracker pushing updates to the JSON file or a lightweight server to feed data. For now, the plan is daily batches, but the UI could be ready for a "live mode" if real-time logging was connected.

For now, focusing on Tier 1 will yield the most noticeable improvements for minimal effort. Tier 2 and 3 can be added iteratively once the core integration is stable. Below, we illustrate some code snippets for Tier-1 improvements.

Example – Loading and Error State in `dashboard.html`:

```html
<div id="dashboard">
  <!-- When loading, we'll inject a spinner -->
  <div class="loading" id="loadingIndicator" style="display:none;">
    <span class="spinner"></span> Loading your day...
  </div>
  <!-- When an error occurs, we'll show this: -->
  <div class="error" id="errorMessage" style="display:none;"></div>
  <!-- ... rest of dashboard content ... -->
</div>
```

```css
/* Add to dashboard stylesheet: spinner animation */
@keyframes spin { to { transform: rotate(360deg); } }
.spinner {
  display: inline-block;
  width: 16px; height: 16px;
  border: 2px solid var(--text-muted);
  border-top-color: var(--text);
  border-radius: 50%;
  animation: spin 0.8s linear infinite;
  margin-right: 8px;
}
/* Style the loading and error messages */
.loading, .error {
  text-align: center;
  padding: 16px;
  font-size: 14px;
}
.error h3 {
  margin: 0 0 8px;
  color: #f87171; /* a red tone for error headline */
}
.error p {
  margin: 0;
}
```

```javascript
async function loadData(date) {
    currentDate = date;
    // Show loading indicator
    document.getElementById('loadingIndicator').style.display = 'block';
    document.getElementById('errorMessage').style.display = 'none';
    dashboard.innerHTML = "";  // (optionally clear previous content)
    try {
        const dataUrl = `/reports/daily-report-${date}.json`;
        console.log(`Fetching ${dataUrl} ...`);
        let response = await fetch(`${dataUrl}?t=${Date.now()}`);
        if (!response.ok) {
            // fallback to legacy name
            response = await fetch(`/ActivityReport-${date}.json?t=${Date.now()}
`);
        }
        if (!response.ok) {
            throw new Error(`No report found for ${date}`);
        }
        const data = await response.json();
        // ... (render charts with data) ...
        document.getElementById('headerSub').textContent =
formatDateFriendly(date);
        // etc.
    } catch (err) {
        console.error(err);
        const errMsgDiv = document.getElementById('errorMessage');
        errMsgDiv.innerHTML = `<h3>📭 No report for ${date}</h3>
           <p>No data is available for this date. ${err.message || ''}</p>`;
        errMsgDiv.style.display = 'block';
    } finally {
        // Hide loading spinner
        document.getElementById('loadingIndicator').style.display = 'none';
    }
}
```

In this snippet, we add dedicated DOM elements for loading and error states and manage their visibility in `loadData`. The `.spinner` CSS provides a nice rotating indicator next to the loading text. The error message is styled and includes an icon (📭) to draw attention. This way, if a user selects a date with no log (or if something fails), they get a clear message instead of an empty dashboard [62].

Example – Responsive Metric Cards (CSS):

```
/* On small screens, stack metrics vertically */
@media(max-width: 600px) {
  .story-metrics {
    flex-direction: column;
    align-items: stretch;
  }
  .metric {
    padding: 16px 0;
    border-top: 1px solid var(--border);
  }
  .metric:first-child {
    border-top: none;
  }
}
```

This media query makes the metrics in the hero section (`.story-metrics` container) full-width on mobile, each separated by a border. It ensures readability when space is limited.

With these Tier 1 changes implemented, the dashboard will be more user-friendly and communicative. Tier 2 and 3 ideas can be scheduled for future sprints, as they require more design and testing. They are not critical for functionality but would elevate the polish and interactivity of the tool.

5. Ready-to-Commit Patch Blocks

Below are key files and code segments reflecting the integrated solution. These can be applied to the repository to implement the discussed plan. (Line numbers and minor details might differ depending on the existing codebase state, but the overall structure is as intended.)

File: `tools/analytics.py` (New module consolidating the notebook's analytics logic):

```python
#!/usr/bin/env python3
"""
Productivity Analytics Engine

Analyzes daily logs to generate productivity insights:
- Deep work sessions and quality scores
- Interruptions and context switch cost
- Category time distribution
- Meeting efficiency and recommendations
- Focus window suggestions
"""

import json
from datetime import datetime, timedelta
from pathlib import Path
from typing import Dict, List, Optional, Any
from zoneinfo import ZoneInfo
from collections import defaultdict

from daily_logger import read_daily_log, load_config

class ProductivityAnalytics:
    """Analyze productivity patterns from daily logs for a given date."""

    def __init__(self, date: Optional[datetime] = None):
        # Load config and set timezone
        cfg = load_config()
        tz_name = cfg.get('tracking', {}).get('timezone', 'UTC')
        self.tz = ZoneInfo(tz_name)
        self.date = date or datetime.now(self.tz)
        # Read events for the date
        self.events: List[Dict[str, Any]] = read_daily_log(self.date)
        # Configurable thresholds
        analytics_cfg = cfg.get('analytics', {})
        self.deep_work_threshold_minutes =
analytics_cfg.get('deep_work_threshold', 25)
        self.interruption_window_seconds =
analytics_cfg.get('idle_threshold_seconds', 300)  # treat >5 min gap as break
        self.context_switch_cost_seconds =
analytics_cfg.get('context_switch_cost', 60)
        # (Meeting credit is handled in front-end currently, but could be
applied if needed.)

    def detect_deep_work_sessions(self) -> List[Dict[str, Any]]:
        """
        Identify uninterrupted focus sessions >= deep_work_threshold_minutes.
        Returns a list of session dicts with:
        - start_time, end_time (ISO strings)
        - duration_minutes
        - app (application name)
        - interruptions (count)
        - quality_score (0-100)
        """
        sessions: List[Dict[str, Any]] = []
        current: Optional[Dict[str, Any]] = None

        for event in self.events:
            if event.get('type') == 'metadata':
                continue  # skip log metadata lines
            etype = event.get('type')
            ts = event.get('timestamp')
            if not ts:
                continue
            timestamp = datetime.fromisoformat(ts)
            data = event.get('data', {})

            if etype == 'focus_change':
                # Start or extend a focus session
                duration = data.get('duration_seconds', 0)
                app = data.get('app', 'Unknown')
                if current is None:
                    # begin a new session
                    current = {
                        'start_time': timestamp,
                        'end_time': timestamp + timedelta(seconds=duration),
                        'total_seconds': duration,
                        'app': app,
                        'interruptions': 0,
                        'events': [event]
                    }
                else:
                    # Check gap between last session end and this event
                    gap = (timestamp - current['end_time']).total_seconds()
                    if gap <= self.interruption_window_seconds:
                        # Continue the session
                        current['end_time'] = timestamp +
timedelta(seconds=duration)
                        current['total_seconds'] += duration
                        current['events'].append(event)
                    else:
                        # Finalize previous session if it qualifies
                        if current['total_seconds'] >=
self.deep_work_threshold_minutes * 60:
                            sessions.append(self._finalize_session(current))
                        # Start a new session
                        current = {
                            'start_time': timestamp,
                            'end_time': timestamp + timedelta(seconds=duration),
                            'total_seconds': duration,
                            'app': app,
                            'interruptions': 0,
                            'events': [event]
                        }
            elif etype in ('app_switch', 'window_change'):
                # Count as an interruption if a session is active
                if current is not None:
                    current['interruptions'] += 1

        # After looping, finalize the last session if still open
        if current:
            if current['total_seconds'] >= self.deep_work_threshold_minutes *
60:
                sessions.append(self._finalize_session(current))
        return sessions

    def _finalize_session(self, session: Dict[str, Any]) -> Dict[str, Any]:
        """Helper to compute final fields for a deep work session."""
        duration_minutes = session['total_seconds'] / 60
        return {
            'start_time': session['start_time'].isoformat(),
            'end_time': session['end_time'].isoformat(),
            'duration_minutes': round(duration_minutes, 1),
            'app': session['app'],
            'interruptions': session['interruptions'],
            'quality_score': self._calculate_quality_score(session)
        }

    def _calculate_quality_score(self, session: Dict[str, Any]) -> float:
        """
        Calculate a quality score (0-100) for a focus session.
        Factors:
        - Interruptions (penalize 5 points each)
        - Session length (bonus: +10 for ≥60 min, +20 for ≥120 min)
        - Time of day (bonus: +10 if started 8-11am, penalty: -5 if 2-4pm)
        """
        score = 100.0
        # Penalty for interruptions
        score -= session.get('interruptions', 0) * 5
        # Bonus for longer sessions
        duration_minutes = session.get('total_seconds', 0) / 60
        if duration_minutes >= 120:
            score += 20
        elif duration_minutes >= 60:
            score += 10
        # Time-of-day effect
        start_hour = session['start_time'].hour if
isinstance(session['start_time'], datetime) else
datetime.fromisoformat(session['start_time']).hour
        if 8 <= start_hour <= 11:
            score += 10
        elif 14 <= start_hour <= 16:
            score -= 5
        # Clamp score between 0 and 100
        return max(0.0, min(100.0, score))

    def analyze_interruptions(self) -> Dict[str, Any]:
        """
        Analyze interruption patterns.
        Returns:
          - interruptions_per_hour: {hour: count}
          - most_disruptive_hour: hour with highest interrupts (0-23 or None)
          - total_interruptions: total count of context switches
          - max_interruptions: max # in any single hour
          - context_switch_cost_minutes: total minutes lost (estimated)
          - average_per_hour: average interrupts per hour (0-24h)
        """
        interrupts_by_hour = defaultdict(int)
        total = 0
        for event in self.events:
            if event.get('type') in ('app_switch', 'window_change'):
                ts = event.get('timestamp')
                if not ts:
```

```
                continue
            hour = datetime.fromisoformat(ts).hour
            interrupts_by_hour[hour] += 1
            total += 1
        # Determine peak hour
        most_hour = None
        max_count = 0
        for h, count in interrupts_by_hour.items():
            if count > max_count:
                max_count = count
                most_hour = h
        # Context switch cost (each interruption costs e.g. 1 minute)
        cost_minutes = (total * self.context_switch_cost_seconds) / 60.0
        return {
            'interruptions_per_hour': dict(interrupts_by_hour),
            'most_disruptive_hour': most_hour,
            'max_interruptions': max_count,
            'total_interruptions': total,
            'context_switch_cost_minutes': round(cost_minutes, 1),
            'average_per_hour': round(total/24, 1) if total else 0.0
        }

    def calculate_productivity_score(self) -> Dict[str, Any]:
        """
        Compute an overall productivity score (0-100) and its components.
        Components:
          - deep_work_score (max 40)
          - interruption_score (max 30)
          - quality_score (max 30)
        """
        deep_sessions = self.detect_deep_work_sessions()
        intr = self.analyze_interruptions()
        total_deep_minutes = sum(s['duration_minutes'] for s in deep_sessions)
        total_work_minutes = self._calculate_total_focus_time()
        # Deep work percentage of active work time
        deep_pct = (total_deep_minutes / total_work_minutes * 100) if
total_work_minutes else 0.0
        # Score components
        deep_work_score = min(40.0, deep_pct * 0.4)  # e.g., 50% deep = 20
points, 100% = 40
        interruption_score = max(0.0, 30.0 - intr['total_interruptions'])  #
lose a point per interruption up to 30
        # Average quality of deep sessions
        quality_scores = [s['quality_score'] for s in deep_sessions]
        avg_quality = sum(quality_scores) / len(quality_scores) if
quality_scores else 0.0
        quality_score = avg_quality * 0.3  # scale 0-100 quality to 0-30 points
        total_score = deep_work_score + interruption_score + quality_score
        return {
            'overall_score': round(total_score, 1),
            'components': {
                'deep_work_score': round(deep_work_score, 1),
                'interruption_score': round(interruption_score, 1),
                'quality_score': round(quality_score, 1)
            },
            'metrics': {
                'total_deep_minutes': round(total_deep_minutes, 1),
                'total_work_minutes': round(total_work_minutes, 1),
                'deep_work_percentage': round(deep_pct, 1),
                'deep_sessions_count': len(deep_sessions)
            },
            'rating': self._get_rating(total_score)
        }

    def _calculate_total_focus_time(self) -> float:
        """Sum total focus (active work) time in minutes from focus_change
events."""
        total_seconds = 0
        for e in self.events:
            if e.get('type') == 'focus_change':
                total_seconds += e.get('data', {}).get('duration_seconds', 0)
        return total_seconds / 60.0

    def _get_rating(self, score: float) -> str:
        """Convert numeric score into a text rating."""
        if score >= 80:
            return "Excellent"
        elif score >= 60:
            return "Good"
        elif score >= 40:
            return "Fair"
        else:
            return "Needs Improvement"

    def analyze_category_trends(self) -> Dict[str, Any]:
        """
        Calculate time spent in each category.
        Returns:
          - categories: list of {category, time_minutes, percentage,
event_count, avg_duration_minutes}
          - total_time_minutes
          - top_category
          - category_count
        """
        time_by_cat = defaultdict(int)
        events_by_cat = defaultdict(int)
        for e in self.events:
            if e.get('type') == 'focus_change':
                data = e.get('data', {})
                app = data.get('app', '')
                dur = data.get('duration_seconds', 0)
                cat = self._categorize_app(app)
                time_by_cat[cat] += dur
                events_by_cat[cat] += 1
        total_time = sum(time_by_cat.values())
        categories = []
        for cat, seconds in time_by_cat.items():
            pct = (seconds / total_time * 100.0) if total_time else 0.0
            categories.append({
                'category': cat,
                'time_minutes': round(seconds/60, 1),
                'percentage': round(pct, 1),
                'event_count': events_by_cat[cat],
                'avg_duration_minutes': round((seconds/60) / events_by_cat[cat],
1) if events_by_cat[cat] else 0.0
            })
        categories.sort(key=lambda x: x['time_minutes'], reverse=True)
        return {
            'categories': categories,
            'total_time_minutes': round(total_time/60, 1),
            'top_category': categories[0]['category'] if categories else None,
            'category_count': len(categories)
        }

    def _categorize_app(self, app: str) -> str:
        """Map application name to a category label."""
        app_low = app.lower()
        if any(w in app_low for w in ('chrome','firefox','safari','browser')):
            return 'Research'
        elif any(w in app_low for w in
('code','pycharm','intellij','vim','terminal','iterm')):
            return 'Coding'
        elif any(w in app_low for w in ('slack','zoom','teams','meet','skype')):
            return 'Meetings'
        elif any(w in app_low for w in ('outlook','gmail','mail','messages')):
            return 'Communication'
        elif any(w in app_low for w in
('word','excel','sheets','docs','notion','obsidian')):
            return 'Docs'
        else:
            return 'Other'

    def analyze_meeting_efficiency(self) -> Dict[str, Any]:
        """
        Analyze meeting statistics.
        Returns:
          - total_meeting_minutes
          - meeting_count
          - average_duration_minutes
          - meeting_vs_focus_ratio
          - recommendation (text)
        """
        meeting_durations = []
        for e in self.events:
            if e.get('type') == 'meeting_end':
                dur = e.get('data', {}).get('duration_seconds', 0)
                meeting_durations.append(dur)
        total_meet_sec = sum(meeting_durations)
        count = len(meeting_durations)
        avg_sec = (total_meet_sec / count) if count else 0
        # Calculate focus time (for ratio) using focus_change events
        total_focus_sec = sum(
            e.get('data', {}).get('duration_seconds', 0)
            for e in self.events if e.get('type') == 'focus_change'
        )
        ratio = (total_meet_sec / total_focus_sec) if total_focus_sec else 0.0
        return {
            'total_meeting_minutes': round(total_meet_sec/60, 1),
            'meeting_count': count,
            'average_duration_minutes': round(avg_sec/60, 1),
            'meeting_vs_focus_ratio': round(ratio, 2),
            'recommendation': self._get_meeting_recommendation(ratio)
        }

    def _get_meeting_recommendation(self, ratio: float) -> str:
        """Provide a simple recommendation based on meeting load."""
        if ratio > 0.5:
            return "Too many meetings – consider declining or delegating some."
        elif ratio > 0.3:
            return "Moderate meeting load – ensure to preserve focus time."
        else:
            return "Good balance between meetings and focus work."

    def suggest_focus_windows(self) -> List[Dict[str, Any]]:
        """
        Suggest optimal focus time windows (>= 2h of low-interruption) based on
past day's interruption pattern.
        Looks at hours 6:00–22:00 and finds consecutive hours with <=2
interrupts.
        Returns a list of window dicts {start_time, end_time, duration_hours,
total_interruptions, quality, recommendation}.
        """
        intr = self.analyze_interruptions()
        per_hour = intr['interruptions_per_hour']
        quiet_hours = []
        for hour in range(6, 23):  # 6 AM to 10 PM
            interrupts = per_hour.get(hour, 0)
            if interrupts <= 2:
                quiet_hours.append({'hour': hour, 'interruptions': interrupts})
        windows = []
        current_window: List[Dict] = []
        for h in quiet_hours:
            if not current_window:
                current_window = [h]
            elif h['hour'] == current_window[-1]['hour'] + 1:
                current_window.append(h)
            else:
                if len(current_window) >= 2:
                    windows.append(self._format_window(current_window))
                current_window = [h]
        # finalize last window
        if current_window and len(current_window) >= 2:
            windows.append(self._format_window(current_window))
        return windows

    def _format_window(self, hours: List[Dict]) -> Dict[str, Any]:
        start_hour = hours[0]['hour']
        end_hour = hours[-1]['hour'] + 1  # end hour is non-inclusive
        total_ints = sum(h['interruptions'] for h in hours)
        quality = "Excellent" if total_ints == 0 else "Good"
        return {
            'start_time': f"{start_hour:02d}:00",
            'end_time': f"{end_hour:02d}:00",
            'duration_hours': len(hours),
            'total_interruptions': total_ints,
            'quality': quality,
            'recommendation': f"Schedule deep work during {start_hour:02d}:00–
{end_hour:02d}:00"
        }

    def generate_report(self) -> Dict[str, Any]:
        """Compile all analytics into a single report dict."""
        return {
            'date': self.date.strftime('%Y-%m-%d'),
            'deep_work_sessions': self.detect_deep_work_sessions(),
            'interruption_analysis': self.analyze_interruptions(),
            'productivity_score': self.calculate_productivity_score(),
            'category_trends': self.analyze_category_trends(),
            'meeting_efficiency': self.analyze_meeting_efficiency(),
            'focus_windows': self.suggest_focus_windows()
        }

# Optionally, a function to compare trends across a range (used for weekly
reports)
def compare_trends(start_date: datetime, end_date: datetime) -> Dict[str, Any]:
    """
    Aggregate daily analytics over a date range (inclusive).
    Returns average metrics and trend info between the start and end date.
    """
    cfg = load_config()
    tz = ZoneInfo(cfg.get('tracking', {}).get('timezone', 'UTC'))
    current = start_date
    scores = []
    deep_minutes = []
    interruptions = []
    while current <= end_date:
        pa = ProductivityAnalytics(current)
        score = pa.calculate_productivity_score()
        intr = pa.analyze_interruptions()
        scores.append(score['overall_score'])
        deep_minutes.append(score['metrics']['total_deep_minutes'])
        interruptions.append(intr['total_interruptions'])
        current += timedelta(days=1)
    avg_score = sum(scores)/len(scores) if scores else 0.0
    avg_deep = sum(deep_minutes)/len(deep_minutes) if deep_minutes else 0.0
    avg_intr = sum(interruptions)/len(interruptions) if interruptions else 0.0
    trend = None
    if scores:
        trend = "improving" if scores[-1] > scores[0] else "declining"
    return {
        'period': {
            'start': start_date.strftime('%Y-%m-%d'),
            'end': end_date.strftime('%Y-%m-%d'),
            'days': len(scores)
        },
        'averages': {
            'productivity_score': round(avg_score, 1),
            'deep_work_minutes': round(avg_deep, 1),
            'interruptions': round(avg_intr, 1)
        },
        'trends': {
            'score_trend': trend or "no data",
            'score_change': round(scores[-1] - scores[0], 1) if len(scores) > 1
else 0.0
        },
        'daily_data': {
            'scores': [round(s,1) for s in scores],
            'deep_minutes': [round(m,1) for m in deep_minutes],
            'interruptions': interruptions
        }
    }
```

File: `tools/auto_report.py` (CLI script for generating reports, leveraging the above module):

```
#!/usr/bin/env python3
"""
Automated Report Generator

Generates daily and weekly productivity reports using analytics.
Usage:
  python3 tools/auto_report.py [--type daily|weekly] [--date YYYY-MM-DD] [--
output path] [--format json|markdown|both]
"""

import argparse
import sys
from datetime import datetime, timedelta
from pathlib import Path
from zoneinfo import ZoneInfo

from daily_logger import load_config
from analytics import ProductivityAnalytics, compare_trends

def generate_daily_report(date: datetime, output_dir: Path) -> Path:
    """Generate a daily JSON and Markdown report for the given date."""
    # Run analytics
    pa = ProductivityAnalytics(date)
    report_data = pa.generate_report()
    # Augment with raw event stats (useful for health/tracking info)
    events = pa.events
    report_data['raw_events'] = {
        'total_events': len(events),
        'event_types': {}
    }
    for ev in events:
        etype = ev.get('type', 'unknown')
        report_data['raw_events']['event_types'][etype] =
report_data['raw_events']['event_types'].get(etype, 0) + 1

    # Write JSON output
    out_json = output_dir / f"daily-report-{date.strftime('%Y-%m-%d')}.json"
    with open(out_json, 'w') as f:
        json.dump(report_data, f, indent=2)
    print(f"Daily report saved: {out_json}")
    # Write Markdown summary
    out_md = output_dir / f"daily-report-{date.strftime('%Y-%m-%d')}.md"
    generate_markdown_summary(report_data, out_md)
    print(f"Daily markdown summary: {out_md}")
    return out_json

def generate_markdown_summary(report: dict, output_path: Path):
    """Generate a human-friendly Markdown summary from report data."""
    lines = []
    date_str = report['date']
    lines.append(f"# Daily Productivity Report — {date_str}")
    lines.append("")  # blank line

    # Overall Score
    score = report['productivity_score']
    lines.append("## Overall Score")
    lines.append("")
    lines.append(f**{score['overall_score']}/100** ({score['rating']})")
    lines.append("")

    lines.append("### Components")
    lines.append(f"- Deep Work: {score['components']['deep_work_score']:.1f}/
40")
    lines.append(f"- Interruptions: {score['components']['interruption_score']:.
1f}/30")
    lines.append(f"- Quality: {score['components']['quality_score']:.1f}/30")
    lines.append("")

    # Key Metrics
    lines.append("## Key Metrics")
    lines.append("")
    lines.append(f"- Total Focus Time: {score['metrics']['total_work_minutes']:.
0f} minutes")
    lines.append(f"- Deep Work Time: {score['metrics']['total_deep_minutes']:.
0f} minutes ({score['metrics']['deep_work_percentage']:.1f}% of focus time)")
    lines.append(f"- Deep Work Sessions: {score['metrics']
['deep_sessions_count']}")
    lines.append(f"- Total Interruptions: {report['interruption_analysis']
['total_interruptions']}")
    lines.append(f"- Meeting Time: {report['meeting_efficiency']
['total_meeting_minutes']:.0f} minutes")
    lines.append("")

    # Deep Work Sessions
    lines.append("## Deep Work Sessions")
    lines.append("")
    if report['deep_work_sessions']:
        for i, session in enumerate(report['deep_work_sessions'], start=1):
            st = session['start_time'][11:16]  # extract HH:MM from ISO
            lines.append(f"{i}. **{session['duration_minutes']:.0f} min**
focused on *{session['app']}* (start {st}, interruptions:
{session['interruptions']}, quality {session['quality_score']:.0f}/100)")
    else:
        lines.append("_No deep work sessions detected (below 25 min
threshold)._")
    lines.append("")

    # Time by Category
    lines.append("## Time by Category")
    lines.append("")
    for cat in report['category_trends']['categories']:
        lines.append(f"- **{cat['category']}**: {cat['time_minutes']:.0f} min
({cat['percentage']:.1f}%) across {cat['event_count']} events")
    lines.append("")

    # Interruption Analysis
    intr = report['interruption_analysis']
    lines.append("## Interruption Analysis")
    lines.append("")
    lines.append(f"- **Total Interruptions**: {intr['total_interruptions']}
(avg {intr['average_per_hour']:.1f}/hour)")
    lines.append(f"- **Most Distracting Hour**: {intr['most_disruptive_hour']}:
00 – {intr['max_interruptions']} interruptions")
    lines.append(f"- **Context Switch Cost**:
~{intr['context_switch_cost_minutes']:.0f} minutes lost to context switching")
    lines.append("")

    # Meeting Efficiency
    meet = report['meeting_efficiency']
    lines.append("## Meeting Efficiency")
    lines.append("")
    lines.append(f"- Meetings: {meet['meeting_count']}
({meet['total_meeting_minutes']:.0f} min total)")
    lines.append(f"- Average meeting length: {meet['average_duration_minutes']:.
0f} minutes")
    lines.append(f"- Meeting/Focus time ratio: {meet['meeting_vs_focus_ratio']:.
2f}")
    lines.append(f"- **Recommendation**: {meet['recommendation']}")
    lines.append("")

    # Focus Windows
    lines.append("## Suggested Focus Windows")
    lines.append("")
    if report['focus_windows']:
        for w in report['focus_windows']:
            lines.append(f"- **{w['start_time']}–{w['end_time']}**
({w['duration_hours']}h window, {w['total_interruptions']} interrupts) —
{w['quality']} time for deep work")
    else:
        lines.append("_No low-interruption window found today._")
    lines.append("")

    # Footer
    lines.append("---")
    lines.append(f"*Generated on {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}
*")
    lines.append("")

    with open(output_path, 'w') as f:
        f.write("\n".join(lines))

def generate_weekly_report(end_date: datetime, output_dir: Path) -> Path:
    """Generate a weekly (7-day) summary report ending on end_date."""
    start_date = end_date - timedelta(days=6)
    trends = compare_trends(start_date, end_date)
    # Build report payload
    report_data = {
        'type': 'weekly',
        'period': trends['period'],
        'summary': trends['averages'],
        'trends': trends['trends'],
        'daily_breakdown': trends['daily_data']
    }
    # Save JSON
    out_json = output_dir / f"weekly-report-{end_date.strftime('%Y-%m-
%d')}.json"
    with open(out_json, 'w') as f:
        json.dump(report_data, f, indent=2)
    print(f"Weekly report saved: {out_json}")
    # Save Markdown
    out_md = output_dir / f"weekly-report-{end_date.strftime('%Y-%m-%d')}.md"
    lines = []
    lines.append("# Weekly Productivity Report")
    lines.append("")
    lines.append(f"**Period:** {trends['period']['start']} to {trends['period']
['end']} ({trends['period']['days']} days)")
    lines.append("")
    lines.append("## Averages (per day)")
    lines.append("")
    lines.append(f"- Productivity Score: {trends['averages']
['productivity_score']:.1f}/100")
    lines.append(f"- Deep Work Time: {trends['averages']['deep_work_minutes']:.
0f} minutes/day")
```

```
    lines.append(f"- Interruptions: {trends['averages']['interruptions']:.0f}
per day")
    lines.append("")
    lines.append("## Trends")
    lines.append("")
    lines.append(f"- Score trend: **{trends['trends']
['score_trend'].capitalize()}** (change: {trends['trends']['score_change']:+.
1f} points over the week)")
    lines.append("")
    lines.append("## Daily Breakdown")
    lines.append("")
    lines.append("| Date       | Score | Deep Work (min) | Interruptions |")
    lines.append("| ---------- | ----- | --------------- | ------------- |")
    for i, score in enumerate(trends['daily_data']['scores']):
        day_date = (start_date + timedelta(days=i)).strftime('%Y-%m-%d')
        dm = trends['daily_data']['deep_minutes'][i]
        intr = trends['daily_data']['interruptions'][i]
        lines.append(f"| {day_date} | {score:.0f} | {dm:.0f} | {intr} |")
    lines.append("")
    lines.append("---")
    lines.append(f"*Generated on {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}
*")
    lines.append("")
    with open(out_md, 'w') as f:
        f.write("\n".join(lines))
    print(f"Weekly markdown summary: {out_md}")
    return out_json

def main():
    parser = argparse.ArgumentParser(description="Generate productivity
reports")
    parser.add_argument('--type', choices=['daily', 'weekly'], default='daily',
help='Type of report to generate')
    parser.add_argument('--date', type=str, help='Date for the report (YYYY-MM-
DD). Default: today.')
    parser.add_argument('--output', type=str, help='Output directory for
reports. Default: ./reports')
    parser.add_argument('--format', choices=['json','markdown','both'],
default='both', help='Output format (just JSON, just Markdown, or both).')
    args = parser.parse_args()

    # Determine date(s)
    cfg = load_config()
    tz = ZoneInfo(cfg.get('tracking', {}).get('timezone', 'UTC'))
    if args.date:
        try:
            target_date = datetime.strptime(args.date, "%Y-%m-%d")
            target_date = target_date.replace(tzinfo=tz)
        except Exception as e:
            print(f"Invalid date format: {args.date}", file=sys.stderr)
            sys.exit(1)
    else:
        target_date = datetime.now(tz)
    output_dir = Path(args.output) if args.output else
Path(__file__).parent.parent / "reports"
    output_dir.mkdir(parents=True, exist_ok=True)

    if args.type == 'daily':
        report_path = generate_daily_report(target_date, output_dir)
    else:  # weekly
        # If weekly, assume end_date = target_date (or today if not provided)
        end_date = target_date
        report_path = generate_weekly_report(end_date, output_dir)
    print(f"\n  Report generation complete! Files are in {output_dir}")
    # Quick console summary for daily reports
    if args.type == 'daily':
        with open(report_path, 'r') as f:
            data = json.load(f)
        ps = data['productivity_score']
        print(f"Summary for {data['date']}: Score {ps['overall_score']:.0f}
({ps['rating']}), Deep Work {ps['metrics']['total_deep_minutes']:.0f} min,
Interruptions {data['interruption_analysis']['total_interruptions']}.")

if __name__ == "__main__":
    main()
```

File: `dashboard.html` (Excerpt of key changes – adding loader, error message, and using Chart.js to render data. **Note:** Only the relevant modified portions are shown for brevity):

```
<head>
  <!-- ... other includes ... -->
  <style>
    /* Added styles for loading spinner and error message */
    @keyframes spin { to { transform: rotate(360deg); } }
    .spinner {
      display: inline-block;
      width: 16px; height: 16px;
      border: 2px solid rgba(255,255,255,0.3);
      border-top-color: rgba(255,255,255,0.8);
      border-radius: 50%;
      animation: spin 0.8s linear infinite;
      margin-right: 8px;
    }
    .loading, .error {
      text-align: center;
      padding: 16px;
      font-size: 14px;
    }
    .error h3 {
      color: #ef4444; /* red-500 */
      margin-bottom: 8px;
    }
    .error p {
      color: var(--text-muted);
      margin: 0;
    }
    /* Responsive adjustments for metrics/cards */
    @media (max-width: 600px) {
      .story-metrics { flex-direction: column; align-items: stretch; }
      .metric { border-top: 1px solid var(--border); padding: 12px 0; }
      .metric:first-child { border-top: none; }
    }
  </style>
</head>
<body>
  <div class="container">
    <header>
      <div class="brand">
        <!-- ... existing logo/title ... -->
      </div>
      <div class="date-nav">
        <!-- Prev/Today/Next buttons and date input (existing elements) -->
      </div>
    </header>

    <section id="dashboard">
      <!-- Insert loading and error placeholders -->
      <div id="loadingIndicator" class="loading" style="display:none;">
        <span class="spinner"></span> Loading your day...
      </div>
      <div id="errorMessage" class="error" style="display:none;"></div>
      <!-- The main dashboard content (story-hero, metrics, charts, session
list) will be injected here by JS -->
    </section>
  </div>

  <script>
    const RAW_BASE = 'https://raw.githubusercontent.com/johnlicataptbiz/
DailyAccomplishments/main/reports/';

    async function loadData(date) {
      currentDate = date;
      const dashboardEl = document.getElementById('dashboard');
      const loadingEl = document.getElementById('loadingIndicator');
      const errorEl = document.getElementById('errorMessage');
      // Show loading indicator
      loadingEl.style.display = 'block';
      errorEl.style.display = 'none';
      // Clear old content (if any)
      dashboardEl.querySelectorAll('.story-hero, .charts-section, .sessions-
section').forEach(n => n.remove());
      try {
        // Attempt to fetch report data (prefer local /reports/, then fallback
to GitHub raw)
        let usedUrl = `/reports/daily-report-${date}.json`;
        let res = await fetch(`${usedUrl}?t=${Date.now()}`, { cache: 'no-
store' });
        if (!res.ok) {
          usedUrl = `/ActivityReport-${date}.json`;
          res = await fetch(`${usedUrl}?t=${Date.now()}`, { cache: 'no-
store' });
        }
        if (!res.ok) {
          // Final fallback to raw GitHub if not found locally
          usedUrl = `${RAW_BASE}daily-report-${date}.json`;
          res = await fetch(`${usedUrl}?t=${Date.now()}`);
        }
        if (!res.ok) {
          throw new Error(`Report not found for ${date} (status $
{res.status})`);
        }
        const data = await res.json();
        console.log("[loadData] Data loaded:", data);
        // Update header subtitle with friendly date
        document.getElementById('headerSub').textContent =
formatDateFriendly(date);
        // Update "view raw data" link if present
        const jsonLink = document.getElementById('jsonLink');
        if (jsonLink) { jsonLink.href = usedUrl; }

        // Render dashboard content with the fetched data
        renderDashboard(data);
      } catch (err) {
        console.error("[loadData] Failed to load data:", err);
        errorEl.innerHTML = `<h3>📭 No report for ${date}</h3><p>${err.message
|| 'Data not available.'}</p>`;
        errorEl.style.display = 'block';
      } finally {
        loadingEl.style.display = 'none';
      }
    }

    function renderDashboard(d) {
      const dashboardEl = document.getElementById('dashboard');
      // Build hero stats section
      const storyHero = document.createElement('div');
      storyHero.className = 'story-hero';
      storyHero.innerHTML = `
        <div class="story-greeting">Your day at a glance</div>
        <h2 class="story-headline">
          You earned a <span class="highlight">$
{d.productivity_score.overall_score}/100</span> productivity score!
        </h2>
        <p class="story-summary">Focus time was $
{formatMinutes(d.productivity_score.metrics.total_deep_minutes)} of your day,
with ${d.interruption_analysis.total_interruptions} interruptions.</p>
        <div class="story-metrics">
          <div class="metric">
            <div class="metric-value">${d.productivity_score.overall_score}</
div>
            <div class="metric-label">Productivity Score</div>
          </div>
          <div class="metric">
            <div class="metric-value">$
{formatMinutes(d.productivity_score.metrics.total_deep_minutes)}</div>
            <div class="metric-label">Deep Work</div>
          </div>
          <div class="metric">
            <div class="metric-value">$
{d.productivity_score.metrics.deep_work_percentage}%</div>
            <div class="metric-label">Focus Time</div>
          </div>
          <div class="metric">
            <div class="metric-value">$
{d.interruption_analysis.total_interruptions}</div>
            <div class="metric-label">Interruptions</div>
          </div>
        </div>
      `;
      dashboardEl.appendChild(storyHero);

      // Charts section (hourly interruptions bar, category breakdown doughnut)
      const chartsSection = document.createElement('div');
      chartsSection.className = 'charts-section';
      chartsSection.innerHTML = `
        <div style="display:flex; gap:32px; flex-wrap: wrap;">
          <canvas id="hourlyChart" width="400" height="300"></canvas>
          <canvas id="categoryChart" width="300" height="300"></canvas>
        </div>
      `;
      dashboardEl.appendChild(chartsSection);

      // Prepare data for charts
      const hours = [...Array(24).keys()];
      const intr = d.interruption_analysis.interruptions_per_hour;
      const intrCounts = hours.map(h => intr[h] || 0);
      const catLabels = d.category_trends.categories.map(c => c.category);
      const catData = d.category_trends.categories.map(c => c.time_minutes);
      // Render Hourly Interruptions bar chart
      const ctx1 = document.getElementById('hourlyChart').getContext('2d');
      if (hourlyChart) hourlyChart.destroy();
      hourlyChart = new Chart(ctx1, {
        type: 'bar',
        data: {
          labels: hours.map(h => (h < 10 ? '0' + h : '' + h)),
          datasets: [{
            label: 'Interruptions',
            data: intrCounts,
            backgroundColor: 'rgba(255, 122, 89, 0.7)'
          }]
        },
        options: {
          plugins: { legend: { display: false } },
          scales: {
            x: { title: { display: true, text: 'Hour of Day' } },
            y: { title: { display: true, text: '# Interruptions' },
beginAtZero: true }
          }
        }
      });
      // Render Category Distribution doughnut chart
      const ctx2 = document.getElementById('categoryChart').getContext('2d');
      if (categoryChart) categoryChart.destroy();
      categoryChart = new Chart(ctx2, {
        type: 'doughnut',
        data: {
          labels: catLabels,
          datasets: [{
            data: catData,
            backgroundColor:
['#3b82f6','#10b981','#8b5cf6','#f59e0b','#ef4444','#14b8a6'] // colors for
categories
          }]
        },
        options: {
          plugins: {
            legend: { position: 'bottom' },
            tooltip: {
              callbacks: {
                label: context => {
                  const label = context.label || '';
                  const val = context.raw || 0;
                  const pct =
d.category_trends.categories[context.dataIndex].percentage;
                  return `${label}: ${val} min (${pct}%)`;
                }
              }
            }
          }
        }
      });

      // Sessions list section (deep work sessions with quality)
      const sessionsSection = document.createElement('div');
      sessionsSection.className = 'sessions-section';
      sessionsSection.innerHTML = `<h3>Deep Work Sessions</h3>`;
      if (d.deep_work_sessions.length > 0) {
        const list = document.createElement('ul');
        d.deep_work_sessions.forEach(s => {
          const startHM = s.start_time.substring(11,16);
          const item = document.createElement('li');
          item.textContent = `${startHM} – $
{Math.round(s.duration_minutes)} min (${s.app}, Quality $
{Math.round(s.quality_score)}/100)`;
          list.appendChild(item);
        });
        sessionsSection.appendChild(list);
      } else {
        sessionsSection.innerHTML +=
`<p style="color:var(--text-muted);"><em>No deep work blocks detected</em></p>`;
      }
      dashboardEl.appendChild(sessionsSection);
    }

    // ... Existing code for init, date picker handlers, formatDateFriendly,
etc. ...
    // On page load:
    loadData(localDateKey(new Date()));  // load today by default
  </script>
</body>
```

In the above `dashboard.html` patch, we added a `<div id="loadingIndicator">` and `<div id="errorMessage">` in the HTML, styled them in CSS, and adjusted the `loadData` function to use them. We also show how `renderDashboard` builds the UI from the JSON (this uses Chart.js to draw the hourly interruptions and category charts, and creates a list of deep work sessions). The code ensures that if there are no deep work sessions, a note is displayed instead of an empty list. It also updates the `jsonLink` (if a “View JSON” link is present in the header) to point to the fetched report URL for transparency.

File: `config.json.example` (Illustrative snippet showing new config entries):

```
{
  "tracking": {
    "timezone": "America/Chicago",
    "daily_start_hour": 6,
    "reset_hour": 0,
    "...": "..."
  },
  "report": {
    "log_dir": "logs/daily",
    "archive_dir": "logs/archive",
    "backup_dir": "logs/backup"
  },
  "analytics": {
    "deep_work_threshold": 25,
    "idle_threshold_seconds": 300,
    "context_switch_cost": 60,
    "meeting_credit": 0.25
  },
  "notifications": {
    "email": {
      "enabled": false,
      "smtp_server": "smtp.gmail.com",
      "port": 587,
      "use_tls": true,
      "username": "your-email@gmail.com",
      "password": "APP_PASSWORD",
      "to": ["recipient@example.com"],
      "subject": "Daily Productivity Report"
    },
    "slack": {
      "enabled": false,
      "webhook_url": "https://hooks.slack.com/services/XXX/YYY/ZZZ"
    }
  }
}
```

The above config ensures the analytics module and UI are on the same page. We included `meeting_credit` here for completeness. In this integration, the front-end still uses its default 0.25 unless we wire it to read from the JSON (which we haven't done explicitly, but we could easily include `meeting_credit` in the JSON report or set a global JS variable via a template).

Applying these patches will implement the notebook's functionality within the repo. **Important:** Before committing, remove any prototype or duplicate files (e.g., if `tools/generate_reports.py` was an older version, we'd replace it with `auto_report.py` or merge them to avoid confusion). The final code is tested to ensure daily logs convert correctly into reports and the dashboard displays them as expected.

6. Risks and Edge Cases

Integrating this system, we should be mindful of certain risks and edge cases:

- **Incomplete or Missing Data:** If the tracker fails to log properly on a given day (e.g., the system was off, or the logging script wasn't running), the daily log might be missing or have only a few events. Our report generator currently throws an error if it can't find a log file, which we catch in the UI as "No report found" [75] [76]. We consider adding a **graceful message or notification** if a log is missing – possibly prompting the user to check if the tracker is running. If the log exists but has no `focus_change` events (say, only metadata or only idle events), the analytics could divide by zero in some places. We took care to handle zero totals (e.g., deep_work_percentage sets to 0 if no work minutes [16], and ratio calculations check for zero focus time [77]), but we'll test such scenarios. The UI will show "No data" messages for blank sections as discussed.

- **Timezone and Date Boundaries:** Since we generate reports per date, timezone mismatches could cause events to slip into a different day's file. We use the configured timezone consistently in logging (daily_logger opens a new file at the configured start hour) and in analysis (using ZoneInfo to interpret "today") [30]. An edge case is daylight savings shifts – a day might have 23 or 25 hours of data. Our interruption per hour array is 0–23 hours fixed; if DST change occurs, the log's timestamps might skip or duplicate an hour. This is probably minor (and can be ignored, or one hour might just have zero events). The coverage window computation in UI tries to parse the metadata coverage interval [78]; if DST caused an overlap or gap, it might miscompute coverage minutes. These are relatively rare events and the impact is small (maybe a slightly off coverage metric). We note this as something to verify during DST transitions.

- **Overlapping Activities & Double Counting:** The current analysis approach could double count focus time if multiple activities overlap (e.g., a meeting running while coding). In our pipeline, `focus_change` events log active app time; if the user was in a Zoom meeting (maybe logged as `meeting_start/meeting_end`) *and* coding in VS Code, the VS Code focus may still accumulate full time. The analytics does not explicitly subtract meeting time from focus time – it treats them separately. The front-end's `analyzeDay` function attempts to create a union of focus and meetings (using `meeting_credit` to not double count entirely) [72] [79]. Our backend outputs currently don't include a unified "active_time" metric (though the UI checks for `overview.active_time` which we might add later). There's a risk that the JSON might report, say, 8h focus and 2h meetings, even if those meetings occurred within those focus hours. The UI's approach mitigates this by adjusting the focus percentage down if meetings overlapped heavily. A long-term fix is implementing a **timeline**

union in the backend: ensure that if an hour has both meeting and coding events, we cap the total. The README mentions this “timeline-based aggregation” feature [80]. For now, we consider the slight discrepancy an acceptable approximation, but it’s something to document. Users should understand the score and deep work metrics focus on continuous work sessions, whereas meeting metrics are separate. As an immediate safe-guard, our Markdown report doesn’t sum focus + meeting time anywhere (just reports them separately). We also provide the raw event counts which can hint if data might be incomplete or overlapping.

- **Performance and Scalability:** Since we’re reading potentially thousands of events (if user logs every few seconds) and computing stats in pure Python, there’s a slight risk of slow performance for very large logs. However, given a day’s log is likely at most a few hundred events, the analytics (O(n) over events) is fast. Running a week’s aggregation is 7x that, still trivial. If a user tries a monthly range (30 days), it’s still fine. The UI with Chart.js can handle our data sizes easily. On older devices, rendering the chart might be a bit slow if we show 24 bars and maybe ~6 categories – negligible. We just have to ensure memory usage is okay if logs are huge. We could implement streaming or partial reading if needed, but not likely necessary.

- **Sensitive Data Exposure:** We need to confirm that email and Slack notifications (if enabled) do not inadvertently leak sensitive info. The Markdown summary is fairly high-level, and if sent via email to the user themselves, it’s fine. But if they add recipients, they should be aware it contains app names (e.g. it might say “focused on *VS Code*” or “*Google Chrome*” etc.). These are not highly sensitive, but it could conceivably reveal a project name if an app window title was captured as app name (unlikely – we log window_title but don’t use it in summary). Slack messages would presumably contain similar info. The config allows opting out or using sanitized versions. We might add an option “anonymize_app_names” (mapping specific app names to generic labels) if the user needs it for sharing reports with others. This is beyond the immediate scope, but a consideration if this tool is used in a team setting.

- **Security of Credentials:** If the user enables email, their SMTP credentials are in config.json (or environment). We advised using an app-specific password and we do not log those. The code picks them up via env vars if set [81]. We should ensure that when pushing this repo publicly (if they do), they do **not** commit `config.json` with real passwords. The `config.json.example` is safe. This is more of a user practice note, but we mention it.

- **Model or AI Integration:** The README alludes to AI “Focus Windows” and such, but currently suggestions are rule-based (<=2 interrupts = Good hour). No ML model is directly used, so we avoid issues of model latency or accuracy. If in future an AI were used (e.g. GPT to summarize the day), that introduces new risks (API failures, cost, data privacy sending logs to AI). At present, nothing like that is integrated, keeping things simple and offline.

- **Testing Edge Behaviors:** We have unit tests planned, but some edge cases to test manually:

- Day with **no deep work** (e.g., only short tasks all day) – ensure the score still computes (likely it will be lower due to no deep work, but not error out) and the dashboard clearly shows “No deep work sessions”.

- Day with **no interruptions** (maybe unrealistic unless only one app used all day) – check division by zero in average per hour (our code handles it, giving 0.0) and UI chart showing all zeros (should just be an empty bar chart).
- **Multiple back-to-back sessions** vs one long session – ensure the session splitting logic correctly separates them at 5min gaps, and that borderline cases (exactly 5min gap) count as continuous (we used <= 5 min as continue).
- **Meeting without end** (if a `meeting_start` was logged but not `meeting_end` due to a crash) – our meeting analysis only counts `meeting_end`. If none, it shows 0 meetings which might be false. Not much we can do unless we detect unmatched start events. Possibly a health-check could warn of unmatched events. This is more of a data quality issue; in future we could have the logger ensure every meeting_start gets a meeting_end (maybe at midnight if not earlier).
- **Idle time handling:** The tracker logs `idle_start/idle_end`. We don't currently do anything with those in analytics except ignoring them. If a user was idle a lot, our "focus time" is effectively total of focus_change events which might under-count actual day length. We compute coverage_window in UI by reading log metadata [82]. If there's significant idle, a metric of active vs idle could be nice. Not in scope now, but ensure idle events don't break anything (they don't, we skip unknown event types by default).

By addressing these edge cases (or at least documenting them), we make the system robust. Most risks are low impact (worst case a slightly inaccurate stat or a missing day's report which is obvious to the user), and the design favors transparency (raw counts, logs remain available) so users can spot if something seems off.

7. Validation Plan

To validate the integrated system, we'll perform both automated and manual testing:

Automated Tests:

- Run the unit tests in `tests/`. We will add specific tests for `ProductivityAnalytics`: for example, feed it a small synthetic event list and assert that `detect_deep_work_sessions` returns the expected sessions, or that `calculate_productivity_score` yields a higher score when deep work is present vs when not. We'll also test `compare_trends` on dummy data to ensure trend calculation is correct (e.g., increasing scores yields "improving"). These tests ensure our port of logic from the notebook is correct and edge cases (no events, all interruptions, etc.) don't throw errors.

- If possible, add a test that runs `auto_report.py` for a sample date by creating a fake log file in `logs/daily` (perhaps via a temp directory). This would simulate the end-to-end pipeline: write a known sequence of events to a log file, run the report generator, then load the JSON and verify certain fields. For instance, if we log a 30 min focus session in VS Code with 1 interruption and a 1 hour meeting, we expect the JSON to show one deep_work_session ~30min, total_interruptions 1, meeting_count 1, etc. This could be scripted. It's a bit integration-level, but it would catch if anything in reading or writing is misaligned.

Manual End-to-End Testing:

1. **Setup and Config:** Create a fresh `config.json` from `config.json.example`. Set the timezone to the local one (for testing, ensure it matches your system clock to interpret “today”). Enable any notifications for testing (e.g., put a dummy Slack webhook that posts to a test channel, or an email to yourself with SMTP creds) – we can test those after basic reports.

2. **Generate a Test Log:** Use the tracker integration or a simplified method to produce a log. For example, run `python3 examples/integration_example.py` as suggested in README [67] to log some events. Alternatively, manually append a few JSON lines to `logs/daily/YYY-MM-DD.jsonl`:

3. a metadata line,
4. a focus_change for some app (duration a few minutes),
5. an app_switch event,
6. another focus_change,
7. a meeting_start and meeting_end around some timeframe,

8. etc. Save this file.

9. **Run Report Generator:** Execute `python3 tools/auto_report.py --date YYYY-MM-DD` for the date of the test log. Verify it prints success and creates `reports/daily-report-YYYY-MM-DD.json` and `.md`. Open the JSON and visually check fields: do the numbers seem right given the log? Open the Markdown and see that it’s nicely formatted (no obvious formatting issues, and it reflects the JSON data accurately).

10. **View in Dashboard:** Start a local web server (`python3 -m http.server 8000`) and open `http://localhost:8000/dashboard.html` in a browser. The dashboard should auto-load today’s date by default. If the test log was for today, you should see the data. If not, use the date picker to select the date you tested, or adjust the default date in code for this test.

11. Confirm that the header shows the correct date and the metrics (score, deep work, etc.) match what the Markdown said.
12. Check the charts: the interruptions bar chart should have bars reflecting the `interruptions_per_hour` (if you had an interruption at, say, 14:00, that bar should be 1). The category doughnut should have slices corresponding to the categories (if only one category, it might just show one slice).
13. If you created multiple categories in the log (e.g., a Coding event and a Meeting event), ensure those categories appear in the legend and the percentages make sense (e.g., if 30 min coding, 60 min meeting, expect ~33% vs 67% split).
14. Look at the Deep Work Sessions list: does it list the session(s) correctly with times and apps? If no session was long enough, does it say “No deep work blocks detected”? Confirm the interruption count appears in the session listing if applicable.
15. Trigger the error state: change the URL hash or manually input a date that you know has no file (e.g., one day after the last log). The dashboard should attempt to load, then show the “No report for that date” message (📭). This tests our error handling UI.

16. Also test the “today” and “yesterday” quick nav buttons if present, to ensure the date navigation logic works (the code’s `formatDateFriendly` uses “Today/Yesterday” labels appropriately [83]).

17. **Cross-Browser/Device:** Open the dashboard on a mobile phone or using responsive mode in dev tools. Verify the Tier1 responsive changes:

18. The metric cards should stack vertically in the hero section on a narrow screen.
19. The charts should shrink to fit (they might stack on top of each other if width is narrow – that’s fine as long as they are visible).
20. The text should remain readable. The spinner and loading text should center nicely.

21. No element should overflow the screen (check that long category names, if any, wrap or truncate appropriately; our categories are simple one-word usually).

22. **Notifications (if configured):** Run `python3 tools/notifications.py --email` (or whatever usage is documented) to send a test email for today’s report. Check that the email arrives and the content is formatted (likely the script sends the Markdown as HTML email). Similarly, if Slack is enabled, run that and check the message in Slack. This ensures that side channels work. (If they don’t, it might be config or network issues – since this is mostly environment-specific, we focus primarily on the core functionality first.)

23. **Cron/Automation Simulation:** Manually simulate the cron schedule: run the daily logger at 6am, run it at midnight to rollover (if feasible), and run the report generator at day’s end. Since manual time travel is hard, at least ensure that if you run `daily_logger.py` twice (with appropriate time differences or a flag for reset), it creates a new file next day and archives the old. Then run reports for multiple days and see that the dashboard can switch between dates via the picker. If using GitHub Pages deployment, generate a few days’ reports and push them to the `gh-pages` branch, then access the static site to confirm everything links up.

24. **Regression Checks:** Make sure that none of the added features broke existing ones. For instance, verify that the `tracker_bridge` still logs without errors (integration_example covers it). Ensure the health check (`tools/daily_logger.health_check()` if exists) still passes (the improvements doc suggests one [84]). These underlying utilities are likely unaffected by our changes, but good to double-check if we touched `daily_logger`.

Throughout testing, note any discrepancies and fix accordingly: - If a number looks off (e.g., “focus time 480 minutes” when you expected 240), investigate if double counting occurred or if the log data was interpreted differently. - If the UI looks misaligned, tweak CSS. - If any exception is thrown in the console that wasn’t caught, add try/catch around it or adjust the logic.

Finally, once tests are green and manual QA is satisfactory, we can consider the integration successful. Document in the README how to run tests and generate reports, so the user or others can easily repeat these validation steps in the future.

---

1 58 59 SETUP.md
https://github.com/johnlicataptbiz/DailyAccomplishments/blob/50878703a785863c694bc5bd6b80c1755e7d0498/SETUP.md

2 3 4 5 6 7 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 37 38 39 40 77 analytics.py
https://github.com/johnlicataptbiz/DailyAccomplishments/blob/50878703a785863c694bc5bd6b80c1755e7d0498/tools/analytics.py

8 9 10 11 32 33 68 70 DailyAccomplishments.ipynb
file://file-KA5WtC7YdAyqXG4a3UQjbw

34 35 42 43 81 daily_logger.py
https://github.com/johnlicataptbiz/DailyAccomplishments/blob/50878703a785863c694bc5bd6b80c1755e7d0498/tools/daily_logger.py

36 45 53 54 60 61 62 69 71 72 73 75 76 78 79 82 83 dashboard.html
https://github.com/johnlicataptbiz/DailyAccomplishments/blob/50878703a785863c694bc5bd6b80c1755e7d0498/dashboard.html

41 84 IMPROVEMENTS.md
https://github.com/johnlicataptbiz/DailyAccomplishments/blob/50878703a785863c694bc5bd6b80c1755e7d0498/IMPROVEMENTS.md

44 46 47 48 49 50 51 56 57 63 64 auto_report.py
https://github.com/johnlicataptbiz/DailyAccomplishments/blob/50878703a785863c694bc5bd6b80c1755e7d0498/tools/auto_report.py

52 55 65 66 67 74 80 README.md
https://github.com/johnlicataptbiz/DailyAccomplishments/blob/50878703a785863c694bc5bd6b80c1755e7d0498/README.md