Design, Validation and Verification document for Focus Enhancement Tool

Team 31
Title: Focus Enhancement Tool(Growin)
Chengyuan Wen (wenc15)
Chiyu Huang (huanc10)
Jingyao Sun (sun250)
Zhecheng Xu (xu562)
Zikai Lu (luz98)

1. Version..... 1
2. Purpose Statement..... 2
3. Component Diagram: Components and Their Relationships..... 2
4. Relationship Between Components and Requirements..... 4
5. Component Specifications..... 7
6. Details On User Interface..... 13
7. Validation & Verification Plan (V&V)..... 14
8. Appendix..... 19

1. Version

- **Version 0 (Submission for Jan 23, 2026)**

  This version describes the design that is realistic and supported by the current codebase, including the implemented backend APIs for focus sessions, user profile, whitelist presets, session history, and website usage logging.

2. Purpose Statement

- **Project purpose**
  Growin is a Windows productivity app that helps users build focus habits through a "plan → focus → feedback" loop. It supports timed focus sessions, monitors the active foreground app, enforces whitelist rules to reduce distractions, and records local history/usage for review, with an extensible design for future features (e.g., rewards/pet system, mini-games).
- **Document purpose**
  This document presents the system's component-level design (boundaries, responsibilities, data flows, and interfaces) and serves as the basis for Validation & Verification by specifying what must be tested to demonstrate the implementation meets requirements. It outlines unit, integration, and performance test plans and measurable criteria to support final evidence at semester end.

3. Component Diagram: Components and Their Relationships

This section presents the high-level architecture of the system by identifying its major components and illustrating the relationships between them. The diagram focuses on **component boundaries and interaction paths**, rather than internal behavior or implementation details. Detailed design, APIs, and component internals are described later in Section 5.

3.1 System Component Diagram

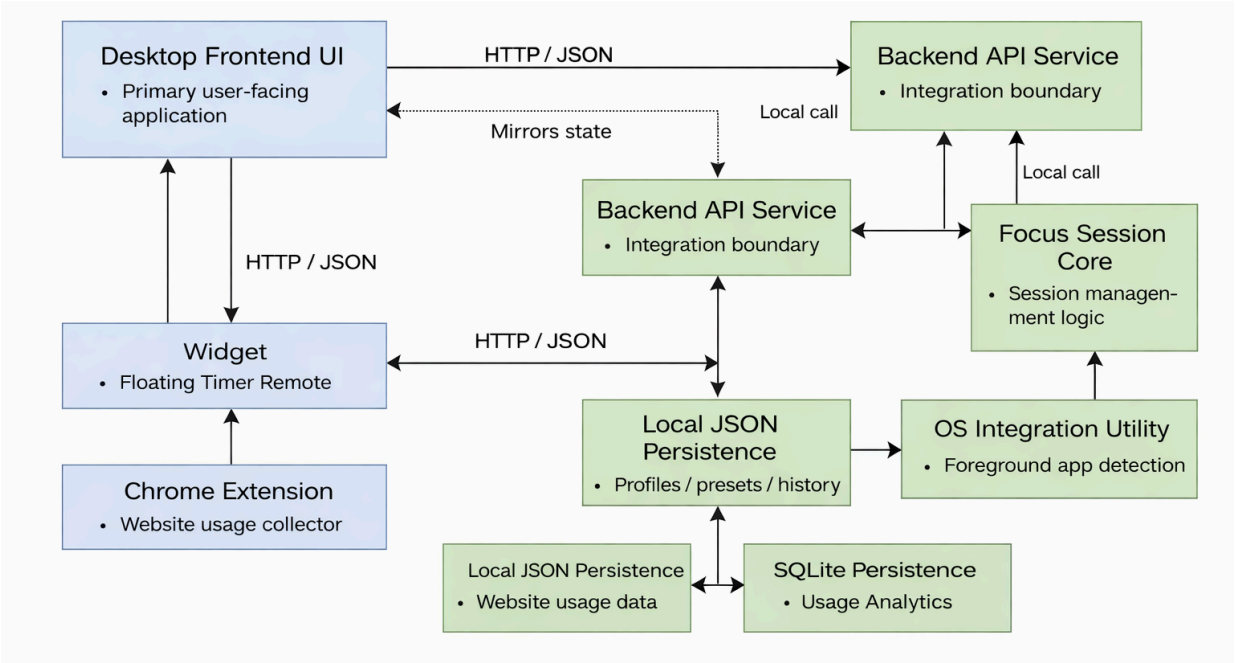


Figure 3.1: System Component Diagram

3.2 Secondary Component Diagram

- Focus Session Control Flow

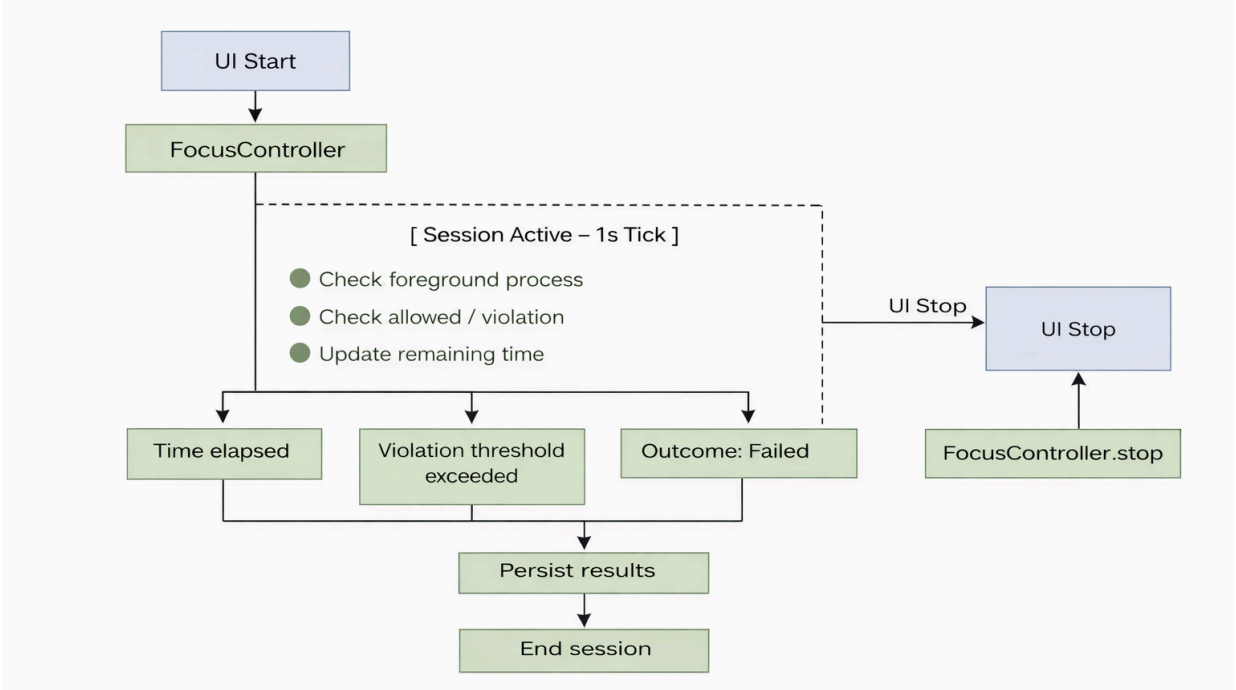


Figure 3.2.1: Focus Session Lifecycle

- Widget Panel Internal Structure

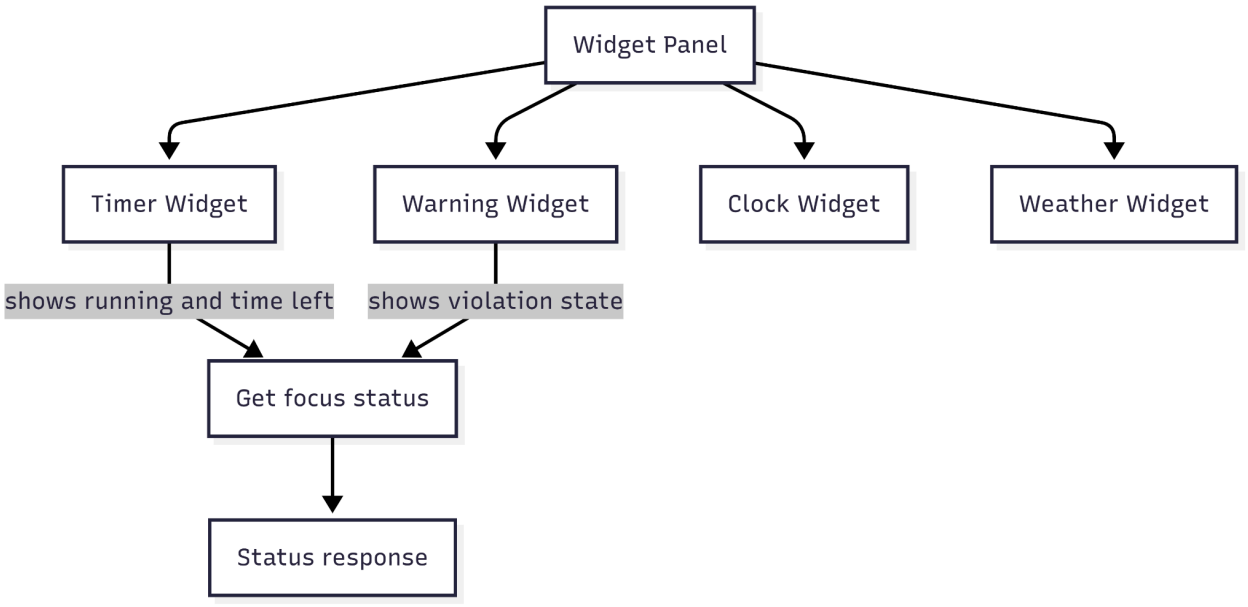


Figure 3.2.2: Widget Panel Internal Structure

4. Relationship Between Components and Requirements

4.1 Requirement Groups Used in This Document

Functional requirement groups (from SRS Section 6.2):

- **FR-P0:** Core Timer, UI, Widgets, Backend Monitoring, Basic User Profile, Local Persistence
- **FR-P1:** Gacha, Pet, Music Player
- **FR-P2:** Mini-Games, Collection, Achievements
- **FR-P3:** Global Dashboard
- **FR-P4:** Friend System & Social Ranking

Non-functional requirement groups (from SRS Section 7):

- **NFR-UI:** Look and Feel (7.1)
- **NFR-UX:** Usability & Humanity (7.2)
- **NFR-PERF:** Performance (7.3)
- **NFR-OPS:** Operational/Environmental (7.4)
- **NFR-MAINT:** Maintainability/Support (7.5)
- **NFR-SEC:** Security (7.6)
- **NFR-CULT:** Cultural (7.7)
- **NFR-COMP:** Compliance (7.8)

4.2 Component-to-Requirement Mapping (Narrative)

4.2.1 Desktop Frontend UI (Windows App)

- **Role:** Primary user interface for configuring focus sessions, viewing live status, and displaying analytics.
- **Meets functional requirements:**
    - **FR-P0:** Provides the timer controls, session setup, widget panel display, profile view, history view, and usage view.
    - **FR-P1–FR-P4 (design-level):** Hosts future UIs for gacha/pet/music, achievements/collections, dashboards, and social features.
- **Meets non-functional requirements:**
    - **NFR-UI, NFR-UX:** Implements clean, distraction-free UI, accessibility, and low-friction flows.
    - **NFR-OPS:** Supports offline-first behavior where applicable.
- **Depends on:** Backend API Service (all core actions), optional Chrome Extension for website usage analytics.

4.2.2 Widget Panel (Timer/Warning/Clock/Weather)

- **Role:** Always-visible feedback during focus sessions.
- **Meets functional requirements:**
    - **FR-P0:** Implements “widgets” requirement (timer + warning, plus clock/weather panel).
- **Meets non-functional requirements:**
    - **NFR-UX:** Immediate and understandable feedback when distraction occurs.
    - **NFR-PERF:** UI state updates within responsiveness targets by polling lightweight status.
- **Depends on:** Focus session status endpoint from the backend.

4.2.3 Chrome Extension (Website Activity Collector)

- **Role:** Collect browser usage events and provide domain-level usage signals to the system.
- **Meets functional requirements:**
    - **FR-P0:** Enables website usage logging (part of monitoring/analytics capability described in P0).
    - **FR-P3 (design-level):** Future global dashboard/aggregations can reuse usage signals.
- **Meets non-functional requirements:**
    - **NFR-SEC, NFR-COMP:** Supports data minimization by sending only domain/time metadata.
    - **NFR-PERF:** Lightweight reporting without impacting browsing.
- **Depends on:** Backend usage ingestion endpoint.

4.2.4 Backend API Service (ASP.NET Core Web API)

- **Role:** Integration hub and stable interface boundary between frontend/extension and backend logic/persistence.
- **Meets functional requirements:**
    - **FR-P0:** Provides endpoints for timer lifecycle, whitelist presets, profile/history retrieval, and usage logging.

    - **FR-P1–FR-P4 (design-level):** Provides a scalable boundary for future reward/social endpoints.
- **Meets non-functional requirements:**
    - **NFR-MAINT:** Encapsulates logic behind clear APIs and supports testing via controller/service separation.
    - **NFR-OPS, NFR-SEC:** Localhost-only operation and controlled access patterns.
- **Depends on:** FocusSessionService, persistence layers (JSON + SQLite).

4.2.5 Focus Session Core (FocusSessionService)

- **Role:** Core state machine for focus sessions (timer enforcement + monitoring + outcomes).
- **Meets functional requirements:**
    - **FR-P0:** Implements session timing, enforcement loop, rule checks against whitelist, and outcome generation.
- **Meets non-functional requirements:**
    - **NFR-PERF:** Supports frequent checks and fast response to disallowed activity.
    - **NFR-MAINT:** Keeps session logic centralized and testable.
- **Depends on:** ActiveWindowHelper (foreground process), LocalDataService (write results).

4.2.6 OS Integration Utility (ActiveWindowHelper)

- **Role:** Reads foreground process/app signal from Windows OS.
- **Meets functional requirements:**
    - **FR-P0:** Enables “backend monitoring” requirement for app-level distraction detection.
- **Meets non-functional requirements:**
    - **NFR-PERF:** Must be efficient enough for frequent polling.
    - **NFR-OPS:** Works on Windows 10/11 user-level installs.
- **Depends on:** OS APIs only (no extra services).

4.2.7 Whitelist Presets (WhitelistPresetsController + JSON storage)

- **Role:** Store and manage reusable allowed-app lists.
- **Meets functional requirements:**
    - **FR-P0:** Provides whitelist rule configuration and preset persistence.
- **Meets non-functional requirements:**
    - **NFR-UX:** Reduces setup friction; supports quick start.
    - **NFR-MAINT:** Clear separation of preset data model from UI.
- **Depends on:** LocalDataService JSON.

4.2.8 Profile and History (ProfileController, FocusHistoryController + JSON storage)

- **Role:** Provide user summary stats and session logs for reflection.
- **Meets functional requirements:**
    - **FR-P0:** Basic user profile and session logs stored locally.

    - **FR-P2 (design-level):** Achievements/collections can derive showable history-based metrics.
- **Meets non-functional requirements:**
    - **NFR-SEC, NFR-COMP:** Local-first, minimal data storage.
    - **NFR-OPS:** Works offline.
- **Depends on:** LocalDataService JSON.

4.2.9 Website Usage Persistence (UsageController + SQLite via EF Core)

- **Role:** Ingest and aggregate website usage for daily analytics.
- **Meets functional requirements:**
    - **FR-P0:** Provides local logging and “today by domain” analytics.
    - **FR-P3 (design-level):** Enables future dashboard aggregation logic.
- **Meets non-functional requirements:**
    - **NFR-PERF:** Efficient storage + aggregation for many usage records.
    - **NFR-SEC:** Stores only necessary metadata.
    - **NFR-MAINT:** Uses EF Core for maintainable data access.
- **Depends on:** AppDbContext + SQLite database.

5. Component Specifications

5.1 Frontend Components

5.1.1 Desktop Frontend UI (Overall)

a) Normal behavior: The Desktop Frontend UI serves as the primary user-facing component. It allows users to configure focus sessions, start and stop sessions, and view live feedback and historical analytics. During a running focus session, the UI continuously reflects backend session state and violation feedback. It also provides internal views for managing whitelist presets, viewing session history and statistics, and accessing auxiliary features such as the pet system.

b) API (inputs/outputs + exchanged data)

- **Interface type:** HTTP client to localhost backend (REST).
- **Inputs:**
    - `StartFocusRequest` - Purpose: submit user-defined session configuration when starting a focus session.
    - `SaveWhitelistPresetRequest` - Purpose: create or update whitelist presets.
- **Outputs:**
    - `FocusStatusResponse` - Purpose: provide live session state for UI rendering and widget updates.
    - `WhitelistPreset`
    - `UserProfile`

c) Implementation (classes, methods, relationships)

- UI event handlers initiate backend requests (e.g., start/stop focus, save/delete presets).
- A centralized API client module issues HTTP requests and parses JSON responses into shared data structures.
- A shared focus-status state is maintained so that multiple UI components (e.g., Focus Session View and Widget Panel) can consume the same session data without redundant polling.
- All UI views depend on the Backend API Service for authoritative session state and persisted data.
- The Widget Panel and Focus Session View are synchronized through shared focus-status data derived from FocusStatusResponse.
- Presets and profile data retrieved from the backend are reused across multiple views to ensure consistency.

d) Potential undesired behaviors

- UI state desynchronization caused by stale polling results or delayed backend responses.
- Excessive or redundant API polling leading to unnecessary resource usage.
- Poor error handling when backend services are unavailable, resulting in blocked UI states or unclear feedback.
- Performance degradation when rendering large datasets (e.g., long session histories or analytics views).

5.1.2 Widget Panel (Timer / Warning / Clock / Weather)

a) Normal behavior: The Widget Panel provides persistent, always-visible feedback during focus sessions without requiring the user to remain on the main page. When a focus session is running, the panel displays session progress (remaining time) and real-time violation feedback.

b) API (inputs/outputs + exchanged data)

The Widget Panel does not initiate session state changes; it is a read-only consumer of backend session status.
Polling: `GET /api/focus/status` → `FocusStatusResponse`
Inputs used (typical fields): `isRunning`, `remainingSeconds`, `isViolating`, `currentProcess`
optional: `violationSeconds`, `isFailed`, `failReason`

c) Implementation (classes, methods, relationships)

- A render function that updates the displayed remaining time and visual state based on subscribed session status.
- Event handlers for the play/stop control that forward user intent to the main Desktop Frontend UI rather than invoking backend APIs directly.
- A single polling timer should ideally update a shared `FocusStatus` state to avoid duplicated requests.

- **Relationship**: The Widget depends on the Desktop Frontend UI as the authoritative source of session state.

d) Potential undesired behaviors

- Visual desynchronization
- Over-polling (multiple widgets each polling independently) causing CPU/network overhead.
- Warning spam or unclear messaging (violates NFR-UX).

5.1.3 Chrome Extension

a) Normal behavior: The Chrome Extension monitors active browser tab changes and periodically records domain-level usage during browsing. It aggregates usage durations over time and batches collected records before submitting them to the backend service. In normal operation, the extension ensures that usage data is delivered reliably and tolerates temporary backend unavailability by deferring transmission until connectivity is restored.

b) API (inputs/outputs + exchanged data)

- `POST /api/usage` with `UsageItemDto[]`
  - Example fields: domain, startTime/endTime or durationSeconds, timestamp, source marker
- Expected responses:
  - 200 OK on accept; 400 on invalid payload

c) Implementation (classes, methods, relationships)

- MV3 background/service worker.
- Domain extraction logic from active tab URL.
- Buffer queue for batching, with flush interval and size limit.

d) Potential undesired behaviors

- Missing events (sleep/tab close), clock drift.
- Sends too frequently (performance impact).
- Privacy leakage (sending full URL or page title instead of domain-only metadata).

5.2 Backend Components

5.2.1 Backend API Service (Overall)

a) Normal behavior

- **Triggers:** app startup (backend hosted locally).
- **Expected behavior:**
  - Starts server on localhost, routes requests to controllers, returns JSON responses.

    - Handles validation and consistent error responses.
- **Outputs/state updates:**
    - Logs meaningful errors; does not crash on malformed requests.

b) API (inputs/outputs + exchanged data)

- REST endpoints are exposed under `/api/*` (see controllers below).
- Content type: JSON; UTF-8.

c) Implementation (classes, methods, relationships)

- ASP.NET Core pipeline: routing, serialization, DI, controller endpoints.
- Services injected into controllers (FocusSessionService, LocalDataService, AppDbContext).

d) Potential undesired behaviors

- Port conflicts / binding failure.
- Misconfigured serialization → missing fields / wrong casing.
- Unhandled exceptions leading to 500s and no useful error body

5.2.2 FocusSessionService

a) Normal behavior

- Initializes session state (duration, allowed list, start time).
- Runs periodic checks:
    - reads foreground process via ActiveWindowHelper
    - compares against allowed list
    - updates remaining time and violation state
- Determines terminal outcome (success/failed/cancelled).
- Persists results to profile/history at session end.

b) API (inputs/outputs + exchanged data)

- Internal service methods (examples):
    - `StartSession(durationSeconds, allowedProcesses)`
    - `GetStatus() -> FocusStatusResponse`
    - `StopSession() -> FocusStatusResponse`
- Uses:
    - `SessionOutcome` enum (Success/Failed/Cancelled)
    - `SessionHistoryItem` record for history persistence

c) Implementation (classes, methods, relationships)

- **Class:** `CapstoneBackend.Services.FocusSessionService`
- **Key fields/state:** `_isRunning`, `_remainingSeconds`, `_whitelist(HashSet)`, `_violationSeconds`, `_currentProcess`, `_failReason`, `_timer`
- **Methods:**

    - `StartSession(StartFocusRequest req)`
        - builds `_whitelist`（Normalize 后的进程名）
        - initializes `_startAt/_endAt/_plannedDurationSeconds/_grace`
        - starts `_timer = new Timer(CheckLoop, ..., 1s)`
    - `CheckLoop(object? state)` *(private)*
        - updates remaining time
        - calls `ActiveWindowHelper.GetActiveProcessName()`
        - checks whitelist; tracks `_violationStart/_violationSeconds`
        - triggers `EndSession(SessionOutcome.Failed)` if grace exceeded
    - `StopSession()` → calls `EndSession(SessionOutcome.Aborted)`
    - `EndSession(SessionOutcome outcome)` *(private)*
        - stops timer; computes `elapsedSeconds`
        - calls `_dataService.RecordSession(outcome, elapsedSeconds)`
        - calls `_dataService.AddSessionHistory(new SessionHistoryItem{...})`
    - `GetStatus()` → returns `FocusStatusResponse` (running, remaining, violating, fail reason, current process)
    - `IsRunning()` → returns `_isRunning`
- **Relationships:**
    - `FocusSessionService` → `ActiveWindowHelper` (foreground process detection)
    - `FocusSessionService` → `LocalDataService` (profile + history persistence)

d) Potential undesired behaviors

- Timer drift or missed ticks (incorrect remaining time).
- Foreground detection false positives → wrongful violations.
- Persistence failure → lost history or profile totals not updated.
- Threading issues (simultaneous reads/writes causing inconsistent status).

5.2.3 ActiveWindowHelper

a) Normal behavior

- Returns the current foreground process name reliably and quickly.
- Returns a safe null/empty result if OS query fails.

b) API (inputs/outputs + exchanged data)

- `GetActiveProcessName() -> string?`

c) Implementation (classes, methods, relationships)

- **Class:** `CapstoneBackend.Utils.ActiveWindowHelper`
- **Method:** `GetActiveProcessName() -> string?`
- **Relationships:**
  - only called by `FocusSessionService.CheckLoop()`; must be lightweight + safe on failure

d) Potential undesired behaviors

- Returns null/empty frequently (permissions/edge cases).
- Incorrect mapping for UWP apps or special windows.
- Performance overhead if called too frequently without throttling.

5.2.4 LocalDataService (JSON Persistence)

a) Normal behavior

- Reads/writes profile, presets, and history JSON under the app's local storage directory.
- Provides safe defaults when files are missing.
- Writes should be atomic to avoid partial corruption.

b) API (inputs/outputs + exchanged data)

- `GetUserProfile() / SaveUserProfile(profile)`
- `LoadPresets() / SavePresets(list)`
- `LoadHistory() / AppendHistory(item)`

c) Implementation (classes, methods, relationships)

- **Class:** `CapstoneBackend.Services.LocalDataService`
- **Methods:**
  - `GetUserProfile()` / `SaveUserProfile(UserProfile)` *(private)*
  - `RecordSession(SessionOutcome outcome, int focusSeconds)` → updates totals and counts
  - `GetSessionHistory()` / `AddSessionHistory(SessionHistoryItem entry)`
  - `GetWhitelistPresets()` / `SaveWhitelistPreset(SaveWhitelistPresetRequest)` / `DeleteWhitelistPreset(string id)`
- **Relationships:**
  - used by `ProfileController`, `WhitelistPresetsController`, `FocusHistoryController`, `FocusSessionService`

    - uses `LocalStoragePaths.*FilePath` and `System.Text.Json` for persistence

d) Potential undesired behaviors

- Corrupted JSON prevents loading.
- Concurrent writes lead to partial content or lost updates.
- Crash during write causes an empty file unless atomic replace is used.

5.2.5 AppDbContext + SQLite

a) Normal behavior

- Creates/connects to `growin.db`.
- Stores website usage records with timestamps/domains/durations.
- Supports fast aggregation queries used by UsageController.

b) API (inputs/outputs + exchanged data)

- EF Core operations via DbSet:
    - insert usage rows
    - query by date range
    - group by domain
- May use `EnsureCreated()` or migrations depending on project setup.

c) Implementation (classes, methods, relationships)

- Entity: `WebsiteUsage` (domain, start/end or duration, timestamp).
- Context: `AppDbContext` with `DbSet<WebsiteUsage>`.
- Used by UsageController for ingestion and aggregation.

d) Potential undesired behaviors

- Migration mismatch / schema drift across versions.
- File permission issues preventing DB open.
- Database growth over time → slower queries without indexes/cleanup.

6. Details On User Interface

The interface is organized into three primary views: Timer, Statistics, and Pet, accessible through a persistent left-side navigation panel. Session-related summaries (focus time, distractions, and collected items) are displayed in a read-only format. Screenshots captured directly from the implemented application are provided in the Appendix to illustrate the internal UI layout and interaction flow.**Refer to Appendix 8 for the Timer, Statistics, and Pet view screenshots**

6.1 Layout and Item Placement

The interface uses a two-column structure. A fixed left-side navigation panel provides access to the three primary views: Timer, Statistics, and Pet. In the default Timer view, a large central countdown display provides clear visual feedback of the configured session length. The Statistics view presents aggregated historical focus data. The Pet view is currently in development. It represents a gamification component that is currently under development.

6.2 User Interactions

In the Timer view, users adjust session duration, select allowed applications, and start or stop a focus session. Once a session begins, interaction is restricted to prevent accidental state changes. The pet view contains several interaction buttons that users could interact with the virtual pet.

6.3 Visual Design

The UI employs soft green and neutral background tones to convey calmness and reduce visual fatigue. Actionable elements, such as the Start button, use darker accent colors to provide clear affordance. Typography follows a sans-serif font style with high contrast between primary information and secondary labels. Font size and weight are used to establish hierarchy, with the timer display receiving the strongest emphasis. All views share a consistent visual structure, spacing, and color scheme to reduce cognitive load.

7. Validation & Verification Plan (V&V)

7.1 Purpose and Scope

This Validation & Verification (V&V) plan defines how Team 31 will verify that **Growin (Focus Enhancement Tool)** satisfies the implemented **P0** requirements and the most relevant non-functional expectations. Because the current Version 0 contains **no machine-learning component**, verification focuses on software correctness and runtime performance. The plan combines (1) automated backend unit/integration tests, (2) manual API acceptance checks using Swagger and VS Code REST Client, and (3) performance measurements on localhost to validate responsiveness and stability.

For **Version 0**, verification is scoped to the components that are already supported by the codebase: the focus session lifecycle (start/status/stop) and its internal state machine behavior; local persistence of profile aggregates, whitelist presets, and focus history via JSON files; and website usage ingestion plus daily aggregation via SQLite/EF Core. Frontend testing is included as a verification plan (unit and E2E), but full automated execution depends on whether the frontend build is integrated end-to-end with the current backend at submission time. Features from later phases (P1–P4) such

as gacha/pet/music, achievements, dashboards, and social functionality are out of scope for executed verification in Version 0 and will be covered in later V&V iterations.

7.2 Test Environment, Tooling, and Evidence

Backend verification uses **xUnit** as the primary test framework and **FluentAssertions** for expressive correctness checks where appropriate. Controller/API integration is performed with **ASP.NET Core WebApplicationFactory**, allowing tests to run against an in-process test host and validate HTTP-level behavior and JSON responses. For data storage behaviors, tests rely on **EF Core + SQLite** in a test configuration (in-memory or temporary database file, depending on the test host setup). Manual acceptance testing is performed through **Swagger UI** and **VS Code REST Client** using the project's `focus-api.rest` file.

The current executed evidence includes successful test discovery and execution using:

- `dotnet test --list-tests` (to enumerate available tests), and
- `dotnet test --logger "console;verbosity=detailed"` (to capture detailed run output).

The latest run shows **7 total tests, 7 passed, 0 failed, 0 skipped**, completing in approximately **2–3 seconds**. EF Core logs during the test run confirm SQLite interaction (e.g., metadata queries against `sqlite_master`), demonstrating that the integration environment can initialize database access reliably. In addition, `focus-api.rest` was executed successfully against P0 endpoints, providing manual acceptance evidence that the backend behaves correctly under realistic request sequences.

For the final submission, the evidence package will include automated test logs (local run and/or CI output), REST acceptance logs or screenshots from Swagger/REST Client, and (if available) Playwright traces/screenshots for end-to-end UI smoke tests. Performance measurements will be reported in a short table comparing target vs measured values, along with a brief description of the measurement method.

7.3 Component Test Plan (Backend: Unit + Integration)

This section defines the component-level testing strategy. Each component is validated for normal behavior and at least one undesired behavior identified in the design section. For Version 0, items marked as "executed" already have passing automated tests; remaining items are planned and will be executed

once the corresponding scenarios are fully stabilized (e.g., deterministic violation simulation, complete history endpoint validation, extension-to-backend ingestion automation).

FocusSessionService (Core Logic).
FocusSessionService is the core focus-state machine and is validated primarily through unit/service-level integration tests. Two executed tests confirm that starting a session correctly sets `isRunning` and initializes remaining time, and that a successfully completed session updates persisted profile totals. Additional planned tests will extend coverage to the violation path (disallowed process triggers `isViolating` and increments violation time), the failure path (exceeding grace threshold sets `isFailed` with a non-empty reason and terminates the session), the cancel path (manual stop records a cancelled/aborted outcome), and resilience cases such as null/empty foreground process values not crashing the loop. Passing this component requires correct state transitions, no negative counters, and clean termination into a final state.

ProfileController and UserProfile.
ProfileController provides a read-only summary of user aggregates, so integration tests focus on JSON correctness and robustness to missing/corrupted persistence files. Executed tests confirm that GET `/api/Profile` returns valid JSON and that LocalDataService provides safe, non-negative default values. Planned tests ensure missing/corrupt JSON returns a default profile without server errors and that schema evolution (missing fields) does not crash deserialization. Passing requires stable 200 responses with valid objects and no 500s due to persistence edge cases.

WhitelistPresetsController (CRUD).
Whitelist presets are verified using controller-level integration tests and persistence round-trip tests. Executed tests confirm create/list/delete flows work end-to-end and that deleting a non-existing preset returns 404. A separate executed persistence test verifies JSON round-trip behavior for preset storage. Planned tests will clarify and enforce the duplicate-name policy, validate payload correctness (empty/invalid fields returning a defined error code), and test concurrency safety (multiple writes should not corrupt JSON; if last-write-wins is the chosen behavior, that behavior will be documented and asserted). Passing requires reliable CRUD semantics, correct HTTP codes, and durable persistence.

FocusController and FocusStatusResponse.
FocusController is currently validated through manual acceptance using Swagger and `focus-api.rest` for start/status/stop. Automated tests are planned to formalize the acceptance behavior: starting a valid session returns `isRunning=true` and `remainingSeconds>0`; subsequent status calls show time decreasing within tolerance; stopping transitions the system back to idle. Additional planned tests cover undesired behaviors: starting while already running returns a documented behavior (e.g., 409) without creating a second session; invalid payloads return 400; stopping while idle does not

crash. Passing requires correct HTTP behavior and consistent state transitions.

FocusHistoryController and SessionHistoryItem.
Focus history verification is planned as controller integration tests. The test suite will validate that missing history files return an empty list rather than errors, and that completing or cancelling a session appends a new record with valid fields (timestamp > 0, duration non-negative, and outcome from the allowed set). Undesired behavior coverage includes safe fallback on corrupted JSON without 500s and prevention of duplicate history append if session-end persistence is accidentally triggered twice. Passing requires stable list responses and consistent record structure across restarts.

UsageController + SQLite (EF Core).
UsageController will be verified through integration tests and performance checks. Current evidence from test logs confirms SQLite initialization works in the test environment. Planned tests will post a known set of usage items and verify that `GET /api/Usage/today` correctly groups by domain and sums durations, and that an empty database returns an empty list. Undesired behavior tests address duplicate event handling (policy must be documented and consistent), DB lock/failure responses (safe error handling without corruption), and correctness near day boundaries (timezone/date classification). Passing requires correct aggregation and stable behavior under repeated ingestion.

LocalDataService (JSON Persistence).
LocalDataService is validated by unit tests focusing on durability and safe defaults. Executed tests already cover profile safe defaults and whitelist preset persistence round-trip. Planned tests will cover missing files (history/presets) returning defaults, corrupted JSON triggering safe fallback (no unhandled exceptions), and atomic write/crash safety (if atomic replace is implemented). Passing requires that persistence never causes unhandled IO/JSON exceptions and that data remains recoverable.

7.4 Frontend Verification Plan (Planned: Unit + E2E)

Frontend verification will be executed once the UI is connected to the current backend endpoints and can run repeatably. Unit tests will use **Vitest + React Testing Library** to validate component rendering, input validation, and request payload generation. For the Focus Session view, tests will confirm Start sends the correct payload (duration, allowed list, grace seconds), UI state toggles correctly (Start disabled while running, Stop enabled), and invalid inputs are blocked with clear messages. Widget Panel tests will validate rendering given mocked FocusStatusResponse objects (timer display, warning state, failure state). History/Stats tests will validate correct aggregation and empty-state behavior. Website Usage view tests will validate rendering and sorting of `{domain, totalSeconds}` lists and graceful handling of backend failure.

End-to-end verification will be implemented with **Playwright** as a smoke suite: launch app, start a session, confirm running UI, stop session, confirm idle UI; create/select a preset and confirm it is used for starting; complete a session and confirm history count increases; and (where reproducible) verify violation warning/failure behavior. Evidence will include Playwright traces and screenshots.

7.5 Performance Tests and Metrics

Performance verification for Growin focuses on responsiveness and stability under normal use, because the system is a local-first desktop tool without ML evaluation metrics. Three API endpoints will be measured for localhost latency (p50/p95 over 200 requests): `GET /api/Focus/status`, `GET /api/focus/history`, and `GET /api/Usage/today`. Targets are p95 ≤ 50 ms for status, p95 ≤ 100 ms for history, and p95 ≤ 150 ms for today usage. UI update latency will be measured (once frontend is integrated) as the time from receiving a status response to rendering the updated UI, targeting ≤ 200 ms.

Timer accuracy will be evaluated by measuring drift over a multi-minute session, comparing wall-clock elapsed time (Stopwatch) against expected countdown progression. Resource usage (CPU and memory) will be measured during an active session with widget polling, targeting average CPU < 5%, peak CPU < 20%, and memory ≤ 300 MB (baseline targets; may be refined based on actual deployment). Finally, SQLite ingest and aggregation performance will be measured by sending synthetic batches to `POST /api/Usage` and timing `GET /api/Usage/today`, with a target such as < 200 ms aggregation at ~10k rows (adjusted if dataset expectations differ).

7.6 Pass/Fail Criteria and Reporting

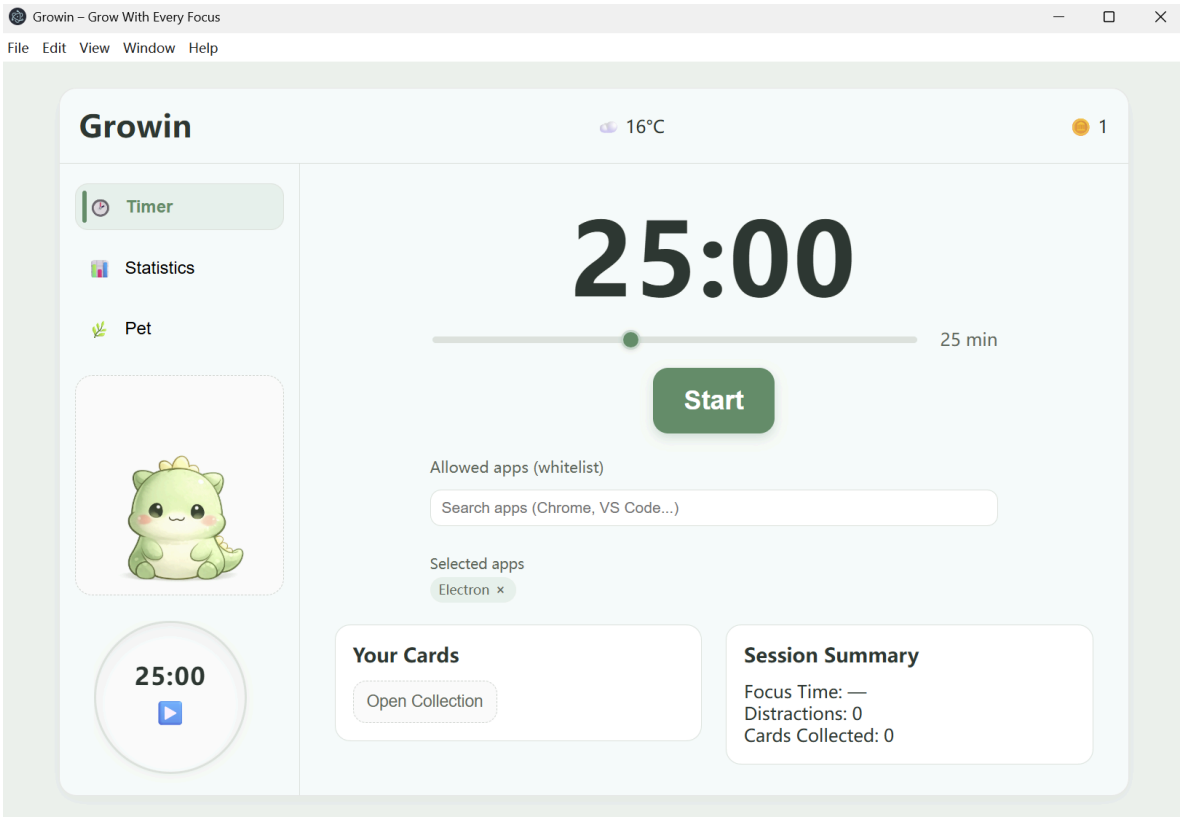
For **Version 0**, the build is considered verified when: (1) automated backend tests pass, (2) manual API acceptance tests via `focus-api.rest` and/or Swagger pass for all implemented P0 endpoints, and (3) performance targets are met or measured values are reported with a mitigation plan. The current executed status satisfies the automated-test requirement with **7/7 passing xUnit tests**, covering FocusSessionService start and completion persistence, LocalDataService default/profile and preset persistence, ProfileController JSON validity, and WhitelistPresetsController CRUD behaviors. The manual acceptance requirement is satisfied by successful runs of `focus-api.rest` against core P0 endpoints.

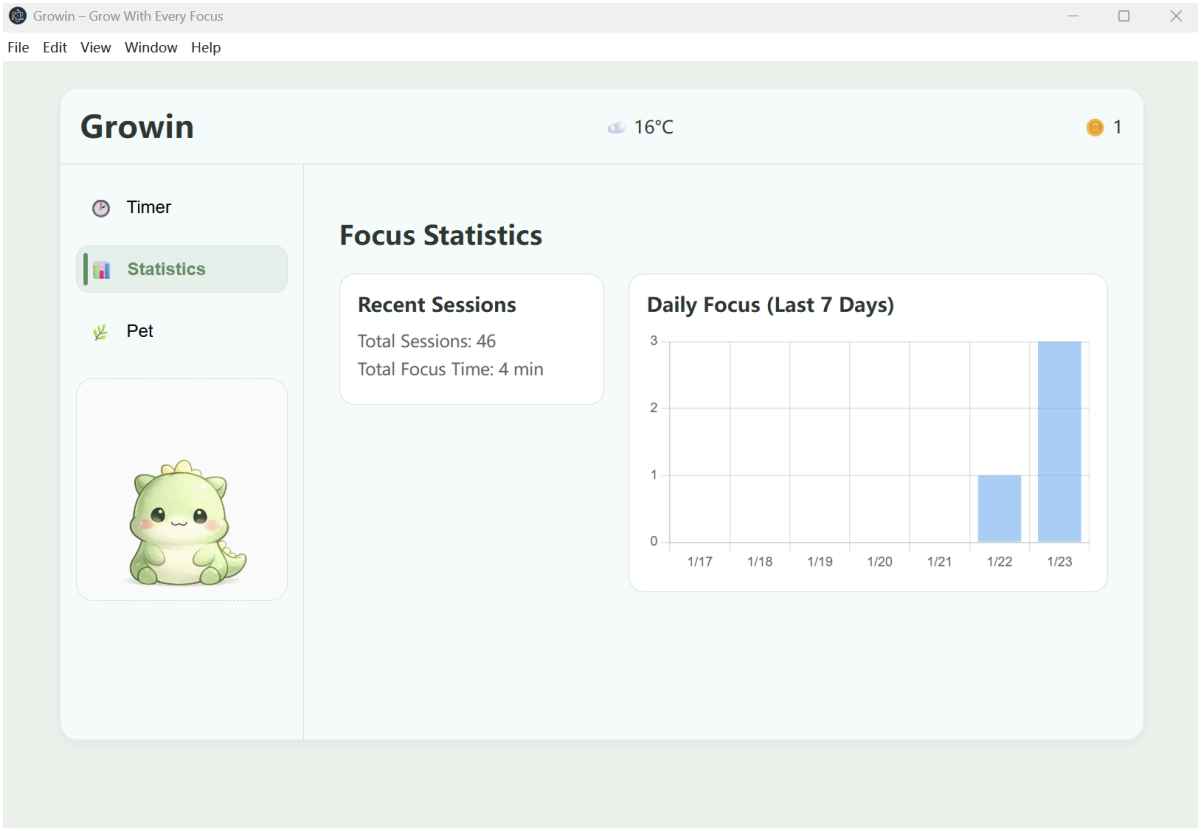
For reporting, the final submission will include: test run logs (and CI logs if available), screenshots or logs from REST acceptance checks, and a concise performance table comparing targets vs measured results. If frontend E2E

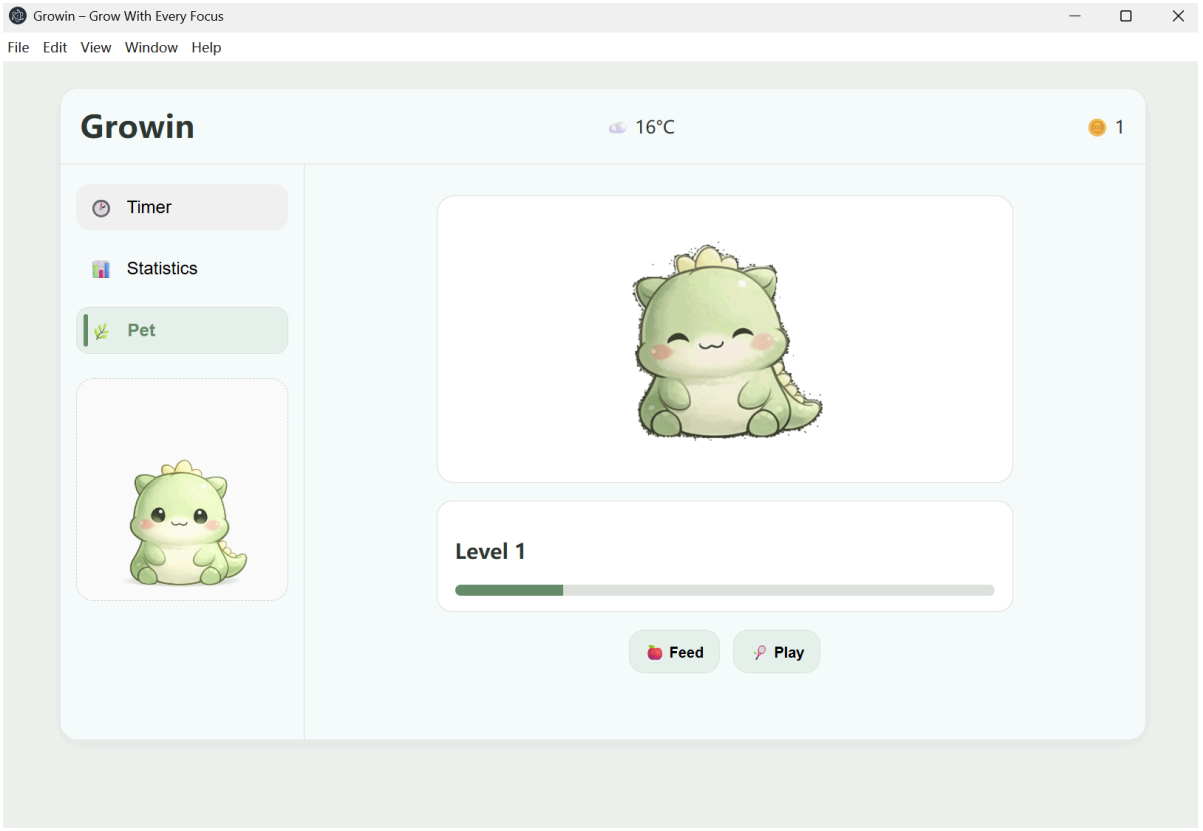
testing is implemented by that time, Playwright traces/screenshots will also be included as evidence.

8. Appendix

UI Screenshots from the Running Application




Growin – Grow With Every Focus
File Edit View Window Help
Growin
16°C
1
Timer
Statistics
Pet
Focus Statistics
Recent Sessions
Total Sessions: 46
Total Focus Time: 4 min
Daily Focus (Last 7 Days)
3
2
1
0
1/17
1/18
1/19
1/20
1/21
1/22
1/23


Growin – Grow With Every Focus
File Edit View Window Help
Growin
16°C
1
Timer
Statistics
Pet
Level 1
Feed
Play