Garmin TimeBox

A Pedagogical System Architecture Guide

From Watch App to Cloud Sync and Dashboard Analytics

This booklet reconstructs the Garmin TimeBox project not as a chat log or coding diary, but as a coherent technical system. Its purpose is to show how a timer app becomes a small but real software product once it acquires persistence, communication, cloud storage, analytics, and deployment.

Prepared for Hannes Zetterblom

Table of Contents

1. The Project at a Glance

Garmin TimeBox is a custom productivity system centered on a Garmin watch app. At first glance it looks like a simple study timer, but that description is too narrow. A timer becomes a software product when it does more than count down. It must preserve state, survive interruption, expose data to other interfaces, and give the user a coherent model of what is happening across devices.

The practical problem it solves is straightforward: the user wants to run tightly defined study blocks on the wrist, collect trustworthy records of completed work, and transform those records into visual analytics. The technical interest of the project lies in how that seemingly modest goal forces the creation of a multi-layer system.

The finished system contains at least five distinct capabilities. First, the watch app manages the live study session. Second, on-watch storage preserves active state and local totals. Third, the app sends completed sessions outward through a communication layer. Fourth, a cloud backend stores the data centrally. Fifth, a dashboard renders the data into charts, totals, and a session feed.

That is why the project qualifies as a small but real software product. It includes a client, persistence, network communication, backend storage, frontend analytics, deployment, and operational tradeoffs. In miniature form, it reproduces the same architectural logic found in much larger systems.

2. The System as Layers

A useful way to understand the whole product is to view it as a layered system. Each layer has a distinct responsibility. Layers exist so that one piece of the system can change without requiring every other piece to be rewritten.

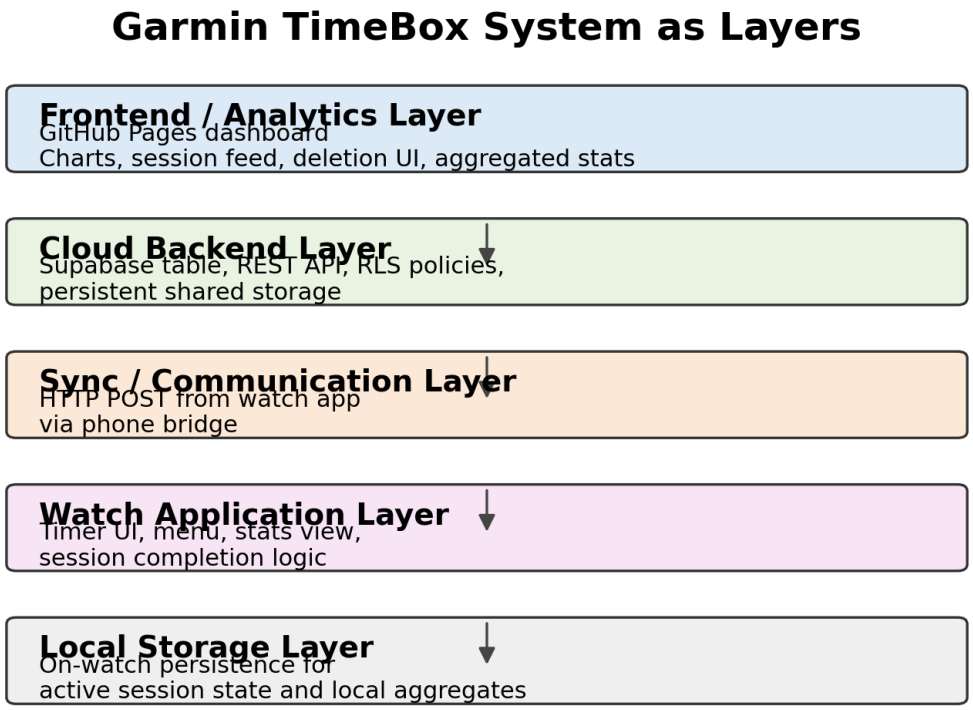

FIGURE 1. THE TIMEBOX SYSTEM AS ARCHITECTURAL LAYERS.

The watch application layer is the visible operating surface on the device. It owns menus, timer screens, completion behavior, and local user interaction. Beneath it sits the local storage layer, which persists active session state and local statistics. Above the watch app sits the communication layer, which transmits a completed session to the outside world. The cloud backend layer stores the data in a durable, shared form. Finally, the dashboard layer translates raw session records into human-readable analytics.

A development toolchain sits beside these runtime layers rather than inside them. The SDK, simulator, source files, compiler, build process, OpenMTP deployment path, GitHub Pages, and Supabase configuration are not part of the running product itself. They are the machinery used to create, test, and deploy the product.

3. Major Components and Their Responsibilities

A serious architectural reading requires more than naming layers. Each major component should be understood in terms of role, dependencies, handled data, failure modes, and design reason.

Component	What it is	Primary responsibility	Key data	If it fails

Watch UI	Connect IQ app running on the Garmin device	Run timers, display states, collect user input	Preset length, remaining time, session mode	User cannot interact with or trust the timer
Local Storage	On-device persistence through Connect IQ storage	Save active session state and local totals	Remaining seconds, running/paused flag, day/week totals	Leaving the app can destroy continuity or local history
Sync Layer	HTTP request triggered by the watch app	Transmit completed sessions outward	Duration, session_date, created_at	Cloud data becomes incomplete or delayed
Phone Bridge	Garmin Connect app and phone connectivity path	Relay network calls from watch to internet	Request payloads and responses	Requests may fail despite a valid local session
Supabase Backend	Hosted database plus REST interface	Durable shared storage for session records	Rows in sessions table	Dashboard cannot reflect cross-device truth
Dashboard Frontend	Static web app hosted on GitHub Pages	Visualize, add, delete, and inspect records	Fetched sessions, aggregates, charts	Data becomes hard to inspect or correct
Toolchain	SDK, simulator, compiler, Git, deployment utilities	Build, test, debug, publish	Source code, compiled .prg, repo history	Iteration becomes slow or error-prone

3.1 The watch app layer

The watch app is the system's real-time control surface. It handles timer presets, custom durations, pause and resume logic, session completion, and user navigation. Its data is transient while the app runs, but it also coordinates with storage so that transient state can be made durable.

This layer depends on the Connect IQ runtime and device capabilities. It also depends on storage and, when sync is enabled, on the communication layer. If the watch app fails, the system loses its source event: the completed study session.

3.2 The local storage layer

Local storage is what allows the app to feel reliable rather than fragile. It stores enough information to reconstruct an active session after interruption and enough summary data to maintain local totals. Without it, leaving the app would often mean losing the session or corrupting the statistics.

Notice the architectural principle here: the user interface state and the persisted state are related but not identical. The screen may currently show 14:32 remaining, but storage must also

remember whether the session was running or paused when the user exited, and when that exit occurred.

3.3 The sync and communication layer

The sync layer turns a local event into a network event. In TimeBox, the relevant local event is session completion. At that moment the watch app constructs a payload and submits it via an HTTP request. This layer is intentionally narrow: it sends small records rather than large histories.

Its purpose is not to own the timer. Its purpose is to carry the timer's finished output into the cloud. That separation matters because the timer must still work even if sync fails.

3.4 The cloud backend layer

Supabase provides a hosted PostgreSQL database with a REST interface and security policies. In this project it functions as the authoritative shared record for dashboard-visible session history. The watch does not need to know how charts are drawn. It only needs to know how to send a correctly structured session record.

The backend exists because browser-only storage is not enough once multiple devices are involved. A laptop browser and a phone browser need a common source of truth. The database supplies that.

3.5 The dashboard layer

The dashboard is a static frontend hosted on GitHub Pages. It fetches sessions from Supabase, calculates aggregates, renders charts, and exposes a feed of completed sessions. This layer is analytically richer than the watch because the browser is a better environment for dense visual interpretation.

Architecturally, the dashboard is a consumer and editor of records, not the producer of live session timing. That distinction is clean and healthy. The watch handles immediate action. The dashboard handles reflection and analysis.

3.6 The deployment and distribution layer

There are really two deployment paths in the project. The watch app is deployed by compiling source into a .prg file and copying that file onto the watch. The dashboard is deployed by committing code to a Git repository and serving it through GitHub Pages. These are different deployment models because the targets are different kinds of runtime environments.

3.7 The development toolchain

The SDK, simulator, build commands, source files, OpenMTP, Git, and GitHub all belong to the development toolchain. This layer matters because engineering is not only about the running product. It is also about the systems that let you change the product safely and repeatedly.

4. The Data Flow of a Completed Study Session

The cleanest way to understand the product end to end is to trace one completed study session from origin to visualization.

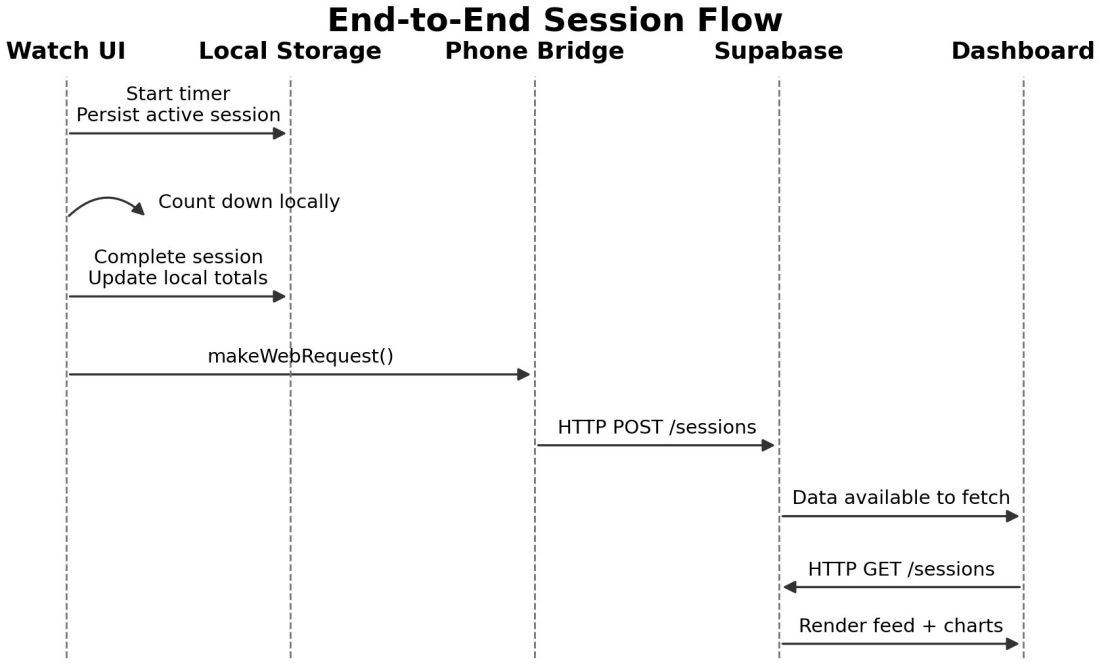

FIGURE 2. SEQUENCE VIEW OF A COMPLETED STUDY SESSION.

Step 1 - Timer start. The user selects a preset or custom duration on the Garmin watch. The watch UI enters an active session state and persists enough information locally so the session can survive interruption.

Step 2 - Local countdown. While running, the watch decrements the remaining time. This is purely local behavior. At this stage no cloud interaction is required because the session is not yet complete.

Step 3 - Completion event. When the remaining time reaches zero, the watch triggers completion behavior: vibration, completion screen, local aggregate update, and creation of a finished session record.

Step 4 - HTTP sync request. The completed session is serialized into a small JSON payload and sent as an HTTP POST request. This request does not travel directly to the browser dashboard. It first targets the backend API.

Step 5 - Phone bridge. On real hardware, the watch does not act like a fully independent laptop. The request is relayed through the phone bridge, typically via Bluetooth to the Garmin Connect environment, then out to the internet.

Step 6 - Supabase insertion. Supabase receives the request and inserts a new row into the sessions table. At this moment the session becomes cloud-visible and available to any authorized client that fetches the table.

Step 7 - Dashboard retrieval. The dashboard issues its own HTTP request, usually a GET, to fetch sessions. It receives rows from the backend and computes display-level structures such as totals, charts, and the session feed.

Step 8 - Rendering. The dashboard translates records into visualization: bars, lines, cards, feed items, and aggregates. The same session that began as a wrist interaction now exists as a rendered data point in a browser.

This chain illustrates a central systems insight. A single human action often becomes several distinct technical events. The user's experience is one continuous act - finishing a study block - but the machine interpretation is a pipeline of local state mutation, serialization, transport, database persistence, retrieval, and rendering.

5. State Management and Persistence

State is the system's current remembered condition. For a timer app, state is not merely the number on the screen. It includes whether the timer is running or paused, whether a session is active or completed, what local totals already exist, whether a completion has already been logged, and what should happen when the user leaves and re-enters the app.

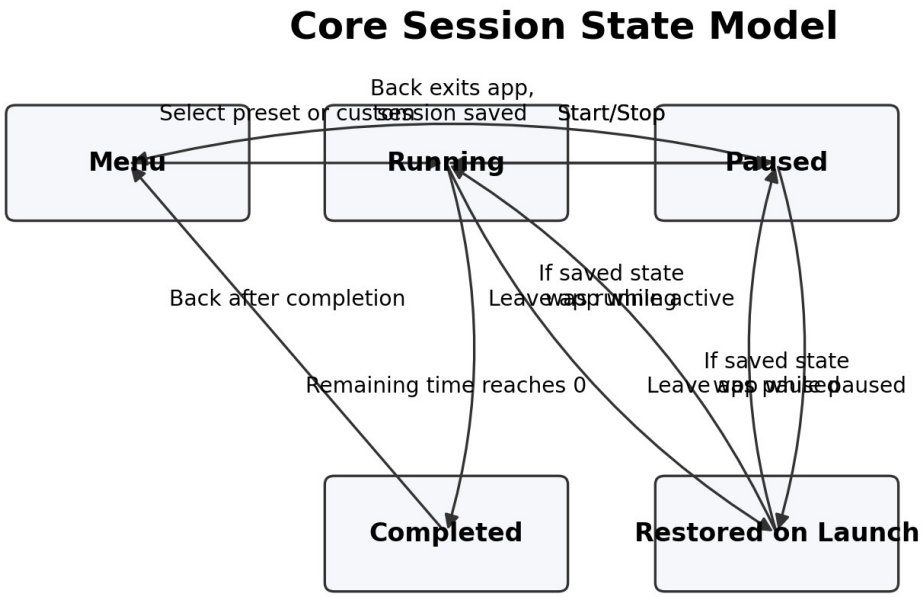

FIGURE 3. CORE STATE TRANSITIONS IN THE WATCH APP.

5.1 Active session state

An active session is one that has been created but not yet completed. It has a selected duration, remaining seconds, and a mode such as running or paused. The active session is the most fragile piece of state because interruption is common on a wearable device.

5.2 Running versus paused

Running means the timer should continue to advance toward completion. Paused means the session remains valid, but time should not continue to be consumed. This distinction matters when the user exits the app. If the user leaves while running, the system must estimate elapsed time on return. If the user leaves while paused, the system must restore the same remaining value.

5.3 Restoring after leaving the app

Wearable apps are interruption-prone. The user may open another activity, take a call, or leave the app temporarily. TimeBox therefore persists enough information to reconstruct the session later. This is a concrete example of why UI state and persisted state are not identical. The interface disappears when the app closes, but the persisted state remains.

5.4 Completion state

Completion is not just a visual message. It is a guarded state transition. Once a session is completed, the app must avoid double-counting it. That is why reset behavior is locked after completion and why local totals and cloud sync are triggered exactly once.

5.5 Local stats accumulation

Today, week, month, and year totals are summary states derived from completed sessions. These summaries are useful because they make the watch independently valuable even if the cloud or dashboard is temporarily unavailable.

5.6 Undo last session and reset today

These functions exist because human input is imperfect. Test sessions, accidental starts, and correction needs are ordinary. A system that tracks productivity but offers no correction path quickly becomes untrustworthy. Undo and reset are therefore not cosmetic conveniences. They are credibility mechanisms.

6. Technical Concepts Explained Through the Project

The best way to learn technical vocabulary is to bind it to a real system. The Garmin TimeBox project is a good vehicle for this because nearly every important term has a concrete instance inside the architecture.

Concept	Meaning in general	Meaning in TimeBox
Source code	Human-readable instructions	The .mc files, XML resources,

		dashboard HTML and JavaScript
Compiled output	Machine-usable artifact produced from source	The watchapp.prg file deployed to the Garmin device
Binary	A non-human-readable encoded executable or bytecode artifact	The .prg package that the Connect IQ runtime can execute
Runtime / VM	Software layer that interprets or executes program instructions	The Connect IQ runtime on the watch
API	Interface for software-to-software interaction	Supabase REST endpoints and Connect IQ communication methods
REST	A common HTTP-based style for web APIs	POST and GET requests between TimeBox clients and Supabase
JSON	Structured text format for data exchange	Payload describing duration, date, and timestamp
Frontend	User-facing interface layer	The browser dashboard and, in a different sense, the watch UI
Backend	Server-side data and logic layer	Supabase database and REST service
Database	System for persistent structured storage	The sessions table in Supabase
RLS	Row-level security rules controlling access	Policies permitting reads, inserts, and deletes
Deployment	Moving software into a runnable environment	Copying .prg to the watch and publishing dashboard files to GitHub Pages

6.1 Source code, compiled output, and the .prg file

Source code is the version meant for humans. It is organized, editable, and interpretable. The watch cannot run those files directly. The build process compiles them into watchapp.prg, which is the deployable artifact for the Garmin environment.

When people say that a file is a binary in this context, they do not mean that it uniquely contains zeros and ones while other files do not. All digital files reduce to bits at the hardware level. What they mean is that the file is encoded in a form intended for machine execution rather than human reading. The .prg file is not raw source. It is a packaged executable or bytecode artifact for the Connect IQ runtime.

6.2 Runtime, virtual machine, and bytecode

The Garmin watch does not simply ingest Python-style text or free-form script files. It runs code through the Connect IQ runtime. Conceptually, that runtime behaves like a managed execution environment. The compiled artifact is interpreted or executed according to rules that the runtime understands. This is why the same broad programming model can work across multiple Garmin devices.

6.3 API, REST, HTTP, and JSON

An API is a contract that lets software speak to software. REST is one common style of API design built around HTTP methods such as GET, POST, and DELETE. In TimeBox, the watch app sends a POST request containing JSON. Supabase accepts that request, validates it through its policies, and inserts the data as a new database row.

HTTP request and response logic is simple to describe but architecturally important. The request carries intent outward. The response confirms whether the attempt succeeded. Once you understand that pattern, a large fraction of modern software becomes easier to reason about.

6.4 Frontend, backend, and database

The frontend is what the user sees and interacts with. The backend is the system that stores and serves durable data. The database is the structured persistence engine inside the backend. In TimeBox, the dashboard is a frontend, Supabase is the backend service, and the sessions table is the specific database structure that stores study records.

6.5 Hosting, deployment, and GitHub Pages

Hosting means making software available in a runtime environment. GitHub Pages hosts the dashboard as a static site. Deployment is the act of moving code into that hosted environment. On the watch side, deployment means copying a new .prg artifact. On the web side, deployment means pushing updated files to a repository whose contents are then served.

6.6 SDK, simulator, build process, and debugging

An SDK, or software development kit, is the bundle of tools required to build for a platform. The Garmin Connect IQ SDK supplies compilation, simulation, device targets, and developer tooling. The simulator is valuable because it turns slow hardware iteration into faster desktop iteration.

Debugging is not an embarrassing detour from engineering. It is engineering. In this project, build failures, type mismatches, device deployment quirks, browser errors, and storage edge cases all shaped the final architecture. Systems become intelligible by being forced through failure.

6.7 Event-driven systems and client-server architecture

The TimeBox pipeline is event-driven because important behaviors occur in response to specific events: button presses, timer completion, app relaunch, and network responses. It is also a client-server system. The watch and dashboard are clients. Supabase is the server-side storage and API system they address.

7. Design Constraints and Architecture Tradeoffs

A good architecture is not merely a diagram of chosen parts. It is a response to constraints.

- The Garmin watch cannot behave like a full desktop app because it has tighter limits on runtime, interaction, and background behavior.
- The phone acts as a bridge because it is a more battery-efficient gateway to the internet than the watch acting alone for each small request.
- Sync can fail because connectivity is conditional. Bluetooth may be absent, the phone may be unavailable, or the backend may reject a request.
- A retry queue is attractive because it converts temporary failure into delayed success rather than permanent loss.
- Two-way sync is harder than one-way sync because deletes and edits create reconciliation problems. The system must then decide which side is authoritative when data differs.
- Local and cloud data models may legitimately differ. The watch may prefer compact summary state for speed and simplicity, while the cloud prefers richer event records for analytics and correction.

These tradeoffs explain why the architecture is rational rather than arbitrary. You do not add a phone bridge because it looks elegant in a diagram. You add it because the hardware and runtime constraints make that bridge the practical route.

One especially instructive tradeoff concerns deletion. Deleting a session on the dashboard is easy if the dashboard is the authoritative editor of cloud history. Making that deletion instantly and flawlessly propagate back into the watch's local totals is harder. The watch may be offline, may contain cached summary values, or may need to reinterpret the past rather than simply remove one row. This is a miniature version of a general distributed-systems problem.

8. Broader Software Engineering Lessons

The project teaches several lessons that matter well beyond productivity apps.

8.1 Full-stack thinking

Full-stack thinking is not merely knowing a frontend language and a backend language. It is the ability to reason across layers. In TimeBox, one design decision in the watch app changes what the database must store and what the dashboard can visualize.

8.2 Systems thinking

Systems thinking means seeing that each component is only meaningful in relation to others. A timer by itself is not analytics. A database by itself is not a study tool. A dashboard by itself is not event generation. The product emerges from the coordinated action of all layers.

8.3 Iterative product development

The system improved through repeated correction: button labels moved, completion logic was guarded, state restoration was refined, and cloud sync was added after local function was already working. This is how real products mature. They rarely emerge in one clean theoretical pass.

8.4 Debugging as engineering work

Beginners often imagine that real engineering is what happens before the error message. In practice, the error message is where architecture gets tested. Type errors, deployment friction, missing policies, and browser console failures each exposed a hidden dependency. That exposure is educational.

8.5 Hardware-connected products

Hardware-linked software is harder than purely browser-based software because the developer must respect device runtimes, transport channels, and operational realities. Wearables are especially strict because they sit at the intersection of tiny screens, limited power budgets, and interruption-prone user behavior.

8.6 Why this matters for a future founder

For an aspiring technical founder, the project's value lies in how many abstractions it forces into contact with reality. You see source code turn into artifacts, artifacts move into devices, events become data, data becomes analytics, and design decisions become operational tradeoffs. This is exactly the kind of compression of learning that builds engineering judgment.

9. Current Limitations and Next Logical Upgrades

The current version already works as a legitimate system, but a truthful architecture guide should also name its unfinished edges.

Area	Current status	Next logical upgrade
Cloud sync	Works when request succeeds	Add retry queue for failed or delayed sync
Session feed richness	Basic records and dashboard feed	Store richer event metadata and timestamps consistently
Reconciliation	Partially manual	Introduce explicit sync status and conflict resolution logic
Distribution	Manual .prg deployment possible	Publish through Connect IQ Store for frictionless installation
User model	Single-user by convention	Add authentication and user-scoped records
Analytics	Strong foundation	Add averages, goals, streak logic, heatmaps, and cohort-like comparisons
Data trust	Reasonable but not perfect	Build auditability around pending syncs, retries, and corrections

The most important next engineering upgrade is a retry queue. Without one, a failed network call can mean that the watch remains locally correct while the cloud becomes incomplete. A queue would store unsent records and retry them later when connectivity is available.

A second major upgrade is stronger reconciliation. Once both local and cloud copies matter, the system needs rules for resolving disagreement. Production systems rarely rely on informal intuition here. They define source of truth, conflict rules, and observable sync states.

Connect IQ Store publishing is also strategically important. Manual deployment is adequate for a single motivated builder, but distribution to friends requires a cleaner install path, packaging discipline, and an update workflow that ordinary users can trust.

Longer term, authentication and multi-user support would transform the project from a personal tool into a shareable product. At that point the backend stops being merely a convenience and starts being a genuine application platform.

10. Final Synthesis

Garmin TimeBox should be understood as a compact but authentic software system. It has a device client, local persistence, network communication, backend storage, browser analytics, and real deployment pathways. None of those layers are decorative. Each solves a specific problem that emerges when the goal shifts from counting down minutes to building a trustworthy study product.

Understanding the project at architectural depth matters because it changes the way you see software. You stop seeing apps as monolithic objects and start seeing them as coordinated layers with responsibilities, interfaces, and failure modes. That shift is one of the foundations of technical maturity.

For someone who wants to build technology companies, this kind of project is disproportionately valuable. It is small enough to hold in the mind, yet rich enough to reveal the logic of modern software construction. In that sense, Garmin TimeBox is not merely a timer app. It is a miniature laboratory for learning how products become systems.

Appendix. Glossary of Core Terms

Artifact: A generated output of development work, such as a compiled .prg file.

Bytecode: An intermediate executable representation interpreted by a runtime or virtual machine.

Client: A program that requests services or data from another system.

Compilation: The transformation of human-readable source code into a deployable executable form.

Connect IQ: Garmin's application platform, runtime model, and SDK ecosystem.

Deployment: The act of making software available in its target runtime environment.

Frontend: The user-facing interface layer of a system.

HTTP: The protocol commonly used for web requests and responses.

JSON: A text format for structured data exchange.

Persistence: The ability of data to survive beyond the current moment of execution.

REST API: A web interface organized around HTTP methods and resource-oriented endpoints.

RLS: Row-level security; per-row access control rules in the database.

Runtime: The software environment in which compiled code executes.

State: The current remembered condition of a system.

Supabase: A hosted backend platform exposing database and API capabilities.

Sync: The act of propagating data from one system layer to another.