RING by Ringwood

Engineering Reference

FOR SOFTWARE AND CONTROLS ENGINEERS

How Ring is built: runtime architecture, the PLC read path, the six-layer write gate stack, persistence, UI, testing, and how to extend it safely.

Generated 2026-09-01 from `docs/manual/engineering`

Contents

Ring — Engineering Reference

The engineering book of Ring's three-audience documentation set. It is written for the person who has to **change** Ring without breaking a plant: read the code, understand why a seam is where it is, and know which document is authoritative for the facts that move.

Ring is the Windows HMI for a corrugator **starch-adhesive glue kitchen**. It speaks EtherNet/IP to an Allen-Bradley CompactLogix 1769-L36ERM through libplctag and keeps its own history in a local SQLite database. From the **2026-09-08 write-enabled cutover** it writes to that controller for real.

How to use this book

1. This book explains; the authorities decide. Ring's repository names one authoritative document per question in `docs/INDEX.md`. Where a fact is volatile — a test total, a tag count, a commissioning status, a procedure to run on cutover night — this book *links* rather than restates. Chapters here go deep on *mechanism*: what the code does, why the seam is there, and what breaks if you move it.

2. Every claim here was read out of the code. This repository has a documented history of confident prose that was wrong; an audit found roughly two-thirds of its in-code TODOs and doc-state claims to be phantom (`CLAUDE.md`). So: file paths are cited inline, claims that could not be verified are marked "unverified", and no volatile number is copied into prose. If this book and the code disagree, **the code wins** and this book is the thing that should be fixed.

3. Read chapter 04 before you touch `Ring/Services/PLC/`. It is the safety-critical chapter and the reason the rest of the book exists.

4. You do not need both backgrounds. This book is written so a controls engineer who has never seen WPF and a WPF developer who has never seen a PLC can both follow it. Every term from either world is defined in chapter 01 §1.0, Vocabulary before it is used. Read that section first if either half of the sentence "a WPF `DispatcherTimer` reads a snapshot of a CompactLogix UDT member" is unfamiliar.

5. Dates matter. Claims that depend on the cutover say so explicitly. A sentence written in the present tense about a *pre-cutover* Ring — "every controller write is suppressed" — stops being true the day `PlcSettings.ReadOnlyMode` is flipped.

Chapters

#	Chapter	What it covers
01	System Overview	What Ring is, the plant and its tanks, solution and directory layout, tech stack, which controller is which

#	Chapter	What it covers
02	Runtime Architecture	Boot sequence and its load-bearing ordering, configuration load order, degraded startup, threading and dispatcher model, how the subsystems fit together
03	PLC Communications — the read path	libplctag usage, the tag reader, seven pollers plus the timers outside them, snapshots and caches, index maps, heartbeat and connection state, clock-skew and comm logging, demo mode
04	The Write Path and Safety Model	**The safety chapter.** Every gate, with the code that implements it; the write-surface register as the pinning mechanism; commissioning holds including the two hard-closed ones; the setpoint queue, echo guard and evidence journal; what a new writer owes
05	Data and Persistence	SQLite configuration, repository shape, migrations, backup/restore contract, the maintenance latch and writer quiesce, historian, audit journals, exports
06	UI Architecture	The two-tier WPF reality, the shell and hosted-screen pattern, screen areas, resource dictionaries and the 13-locale rule, converters, touch conventions, the shared-handler rule
07	Services Catalog	A guided index to everything under `Ring/Services/` — one verified paragraph per service, grouped by concern
08	Testing and Quality	Suite mechanics, why parallelism is off, the project-membership guard, the production gate, encoding rules, worktree conventions
09	Build, Deploy, Release	Build mechanics and their traps, the release package shape, install/rollback/remote access (links to the runbooks)
10	Extending Ring	Cookbook: a screen, a service, a localized string, a config key, a database table — and the one sanctioned way to add a PLC writer

#	Chapter	What it covers
11	Diagnostics and Field Triage	Symptom-indexed index into the rest of this book: what to check first for a frozen screen, a flapping link badge, a write that silently does nothing, a stuck Hold, a station that will not launch, and more

Corrections this book makes to existing documents

Per the "verify, then write" rule, where this book once found existing prose to disagree with the code, it said so in place rather than smoothing it over. Four such corrections have since been folded back into the documents they named — `ARCHITECTURE.md` §§2–4 and `docs/CONFIG_AND_STARTUP.md` §1 now state the current facts directly, so this table no longer carries them (re-verified 2026-09-01). Two still stand:

Document	Claim	What the code says
`PlcWriteEndpointGuard`'s own class doc	describes itself as the **last** gate, after the heartbeat and demo checks	In `PlcTagWriter.WriteCore` the order is ReadOnly → Demo → Endpoint → Heartbeat, for a documented diagnostics reason (`PlcWriteEndpointGuard.cs:40-44`, verified against `PlcTagWriter.cs:143-208`). ch. 04
`README.md` (repository root)	*"13 tanks (1 mix + 6 storage + 6 doser)"* (`README.md:13`)	That is `TankRoster.IndianaPreset()`, the **target**; the shipped fallback is 4 storage + 4 dosers, and the mix tank is deliberately outside the roster. ch. 01

Neither is safety-critical on its own. Both are the kind of drift that makes the next reader distrust the parts that *are*. A table that only ever grows is also a sign this book has stopped checking whether its own corrections got applied — the four retired rows above are proof that re-verifying periodically is worth doing.

Sibling books

Part of the three-book documentation set indexed at `docs/manual/README.md`:

- Operator Manual — day-to-day plant use.
- Sales & Commercial Guide — the same system at commercial altitude.

Per the set's convention, **every chapter here ends with a "Verified against" list** naming the files its claims were read from. Keep those lists honest when you change a chapter; they are how the next reader checks this book the way this book checked the others.

The documents this book defers to

Question	Authority
Which document is authoritative at all?	`docs/INDEX.md`
Are we allowed to enable writes?	`docs/production-readiness/WRITE_ENABLE_READINESS_2026-07-26.md`
How is the cutover run?	`CUTOVER_RUNBOOK.md`
What does a config key mean?	`docs/reference/plc/APPSETTINGS_REFERENCE.md`
How does Ring boot and where do settings live?	`docs/CONFIG_AND_STARTUP.md`
What are the controller facts older docs get wrong?	`docs/PLC_FACTS.md`
What is still held for a controls engineer?	`docs/production-readiness/CONTROLS_SIGNOFF_RECORD.md`
Test totals, gate timings	`artifacts/production-gate/production-gate-summary.json` (generated)
Tag audit / dead-read counts	`scripts/code-tag-audit.md` (generated)

A note on the links in this table, and in this book generally. Most of them are repository paths — `docs/production-readiness/*.md`, `CLAUDE.md`, `CONTRIBUTING.md`, `ARCHITECTURE.md`, `CUTOVER_RUNBOOK.md`. In the published web and PDF editions those paths resolve to the repository's GitHub remote, which most readers of a distributed PDF cannot open. Where that is true, **the path itself is the citation** — it names the authoritative file precisely so you can find it in a checkout even when the link does not resolve for you.

And the two documents that set the norms this book follows: `ARCHITECTURE.md` and `CONTRIBUTING.md`.

01 — System Overview

Who this is for. Anyone about to work on Ring for the first time, and anyone returning to it after a while and needing the map again.

What you'll learn. What Ring is and what plant it drives; what the solution actually contains; where each kind of code lives; the exact technology stack; and which controller address means what — because confusing the bench twin with the live plant controller is the single cheapest way to do real damage here.

1.0 Vocabulary — read this if you come from only one side

Ring sits between two worlds that rarely share a vocabulary. This section defines every term this book uses without further explanation. Skip whichever column you already know.

Controls / PLC terms, for a software engineer

Term	What it means here
PLC	Programmable Logic Controller — the industrial computer that actually operates the plant. Ring never *controls* anything directly; it asks the PLC to, and the PLC decides
CompactLogix 1769-L36ERM	The specific Allen-Bradley PLC model this plant runs. "L36ERM" and "the controller" mean the same box throughout this book
EtherNet/IP	The industrial Ethernet protocol Ring speaks to the controller. Not the same thing as "an ethernet IP address"
CIP	Common Industrial Protocol — the request/response layer carried over EtherNet/IP. A "lost CIP reply" is a request that reached the controller whose answer never came back
EIP session	A connection the controller holds open for one client. A CompactLogix supports only about **four to eight per source IP**, which is why Ring staggers its pollers and why running several field scripts at once starves them
Tag	A named variable inside the controller (`current_step`, `Tanks[3].Level`). Ring reads and writes tags; it has no other way to see or change anything
UDT	User-Defined Type — a struct on the controller. `Tanks[3]` is one UDT element; `.Level` is a member of it
DINT / INT / SINT / REAL / BOOL	Controller data types: 32-bit int, **16-bit** int, 8-bit int, 32-bit float, single bit. Reading an INT with a 32-bit read fails; this bites regularly

Term	What it means here
Ladder / ST routine	The program running *on the controller*. "Ladder logic" is the graphical form; "ST" is Structured Text. Ring does not run it, cannot change it, and must not fight it
Scan	One pass of the controller's program. "Cleared within four scans" means the ladder overwrites your value almost immediately
Rung	One line of ladder logic. `XIC` / `XIO` / `OTE` / `OTU` / `OTL` / `ONS` / `MOV` are instructions on a rung. `ONS` is a one-shot: it fires on a *rising edge*, so a bit stuck TRUE means it can never fire again
Sealed coil	A rung whose output holds itself on until something specific breaks the seal. `PC_Write_Integer[30].1` (Hold) drives one — which is why a stuck Hold bit cannot be resumed
Momentary pulse	Set a bit TRUE, wait, set it FALSE. The controller acts on the transition. Both halves matter; leaving the bit TRUE is a fault
L5K / L5X	Text exports of the controller program. `scripts/RS_3000_2024_APR_22.L5K` is plaintext and is the **math** authority; the 2026 running export is source-protected
Setpoint	A target value an operator sets (a temperature, a level). Distinct from a command bit
HMI	Human-Machine Interface — the operator screen. Ring is an HMI
LCP	Local Control Panel — the plant PC that runs Ring
Commissioning	Proving on real hardware that a change does what it is supposed to. "Held for commissioning" means the code exists and is deliberately inert
TVC	The temperature-control unit family in this plant (`TVC[]`, `TVC_Ctrl[]`, `TVC_IO[]` tags). One TVC unit can serve more than one tank
Doser / use tank	The tanks that dose adhesive to the corrugator, as opposed to storage tanks and the single mix (Make-Ready) tank

WPF / .NET terms, for a controls engineer

Term	What it means here
WPF	Windows Presentation Foundation — the Windows UI framework Ring's screens are built with
XAML	The markup language that describes a WPF screen. Each `.xaml` file has a matching `.xaml.cs` "code-behind"
Code-behind	The C# file paired with a XAML file. Most Ring screens do their work here

Term	What it means here
MVVM / ViewModel	A pattern where screen state lives in a plain class (the ViewModel) and the XAML *binds* to it. Ring uses this for its process screens and plain code-behind elsewhere — see chapter 06
Binding / `DynamicResource`	Ways XAML pulls a value from somewhere else. A `DynamicResource` re-reads when the source changes, which is how a language switch repaints live
Dispatcher / UI thread	The single thread allowed to touch controls. Blocking it freezes the screen
`DispatcherTimer` vs `System.Threading.Timer`	The first ticks on the UI thread; the second on a background thread. Ring's PLC pollers deliberately use the second
Resource dictionary	A XAML file of shared values — colours, sizes, and every translated string
DI container	The object that constructs services and hands them to whoever needs them
xUnit / `[Fact]`	The test framework and the attribute that marks one test
csproj (non-SDK)	Ring's project files list **every** source file by hand. Adding a file without adding its row means it silently does not exist
SQLite / WAL	The single-file database Ring keeps its history in, and its write-ahead log
libplctag	The open-source library Ring uses to speak EtherNet/IP. Ring adds no protocol code of its own

---

1.1 What Ring is

Ring is a Windows desktop HMI for a corrugator's **starch-adhesive glue kitchen** — the plant that cooks and doses the starch adhesive a corrugator uses to glue board. It replaces `~RS360`, a Borland C++ application that ran the plant for years (`README.md`); the marketed product name for that lineage is RS-3000, which is why both names appear in the tree (`docs/manual/sales/README.md`).

Three properties shape everything else in the codebase:

1. **It is a plant application, not a records application.** A bad write does not corrupt a row, it moves a valve. This is the origin of the whole gate stack in chapter 04.
2. **It ships read-only.** `PlcSettings.ReadOnlyMode` defaults to `true` in C# (`Ring/Infrastructure/Configuration/AppSettings.cs:326`) and is `true` in the shipped `Ring/Config/appsettings.json`. The resolver `PlcWriteGuard.ResolveConfiguredReadOnly` is `settings?.PlcSettings?.ReadOnlyMode ?? true` (`Ring/Services/PLC/PlcWriteGuard.cs:75-76`) — a missing or garbled settings object fails *closed*.

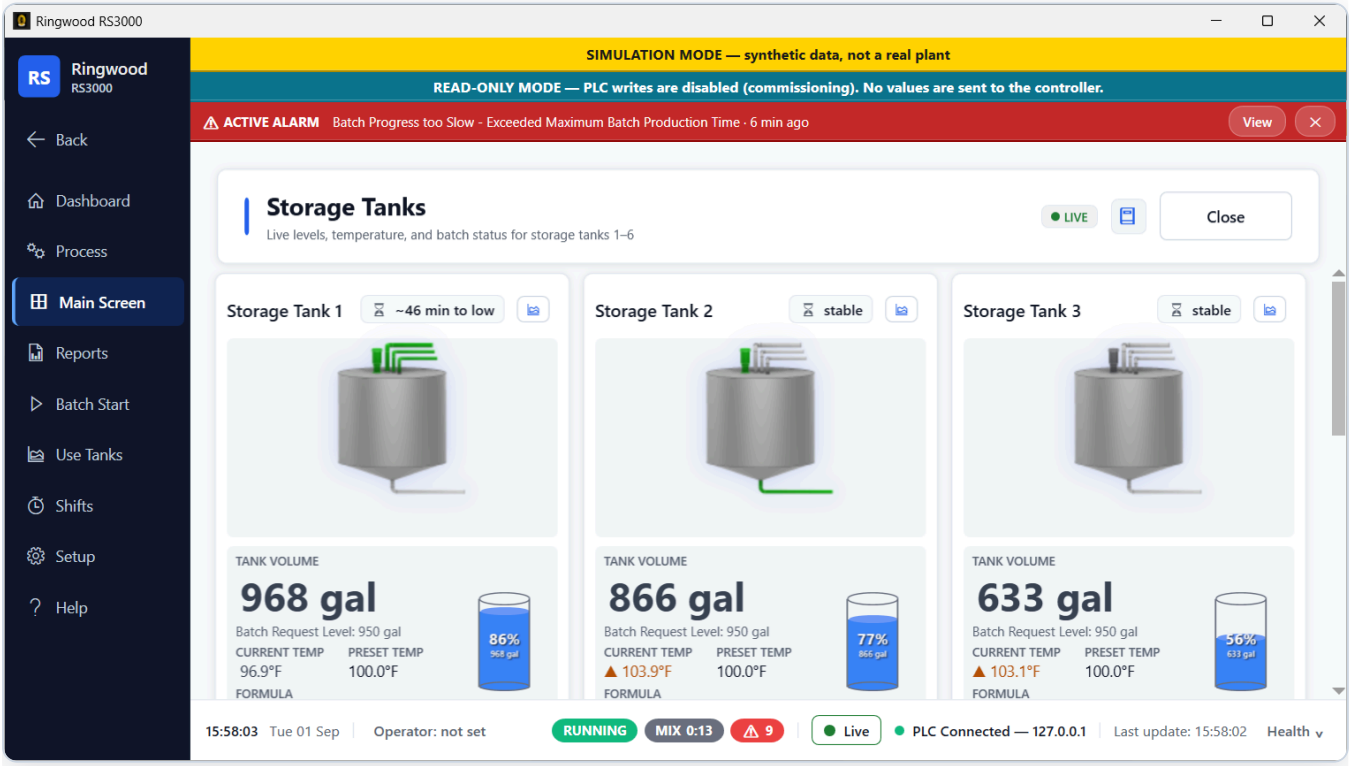
3. **It is a single process with no server.** One EXE, one local SQLite file, one controller. There is no service tier, no message bus, no cloud dependency.

   Cutover. The repository is pointed at the **2026-09-08 write-enabled cutover** — the day `ReadOnlyMode` is intended to be flipped to `false` and Ring begins driving the plant. Wherever this book says "writes are suppressed", that is a statement about the *shipped configuration*, not a permanent property. The go/no-go gate is `docs/production-readiness/WRITE_ENABLE_READINESS_2026-07-26.md`; the procedure is `CUTOVER_RUNBOOK.md`.

1.2 The plant

Ring's first site runs it in English, but the plant replaced a Dutch-language legacy system (`README.md`) — the controller's alarm text and the imported recipe/tank vocabulary are Dutch. That heritage matters in code in at least two places worth knowing about up front:

- Operator-facing text is localized into thirteen languages (`Ring/Resources/Strings/`, thirteen `Strings.<code>.xaml` files), and Dutch is a first-class target, not an afterthought.
- `LocalizationService` deliberately never touches `CurrentCulture` / `CurrentUICulture` (`Ring/Services/Display/LocalizationService.cs:23`), so a Dutch UI never changes numeric parsing. That separation is why `PlcTagWriter.TryParseRealInvariant` (`Ring/Services/PLC/PlcTagWriter.cs:393`) exists: under `nl-NL`, a culture-sensitive parse turns `"1.5"` into `15.0` — a tenfold setpoint write.



The plant this section describes: storage tanks side by side, live. This is the Storage Tank Group screen, not a diagram — see chapter 06 for how the screen is built.

Tanks — read this carefully, the numbers are role-dependent

The tank population is **configurable at runtime**, not compiled in. It lives in `Ring/Services/Roster/TankRoster.cs`, persisted as `%LocalAppData%\Ring\tank_roster.json` by `Ring/Services/Roster/TankRosterState.cs`.

Fact	Value	Source
Roles in the roster	`Storage`, `Doser`	`Ring/Services/Roster/TankRole.cs`
Max slots per role	6	`TankRoster.MaxSlotsPerRole` (`TankRoster.cs:20`)
Hard cap on total roster slots	12	`TankRoster.MaxTotalSlots` (`TankRoster.cs:23`)
Shipped fallback roster	4 storage (PLC 1–4) + 4 dosers (discovery-filled)	`TankRoster.Default()` (`TankRoster.cs:102-115`)
The 13-tank target preset	1 mix (*not in the roster*) + 6 storage + 6 doser	`TankRoster.IndianaPreset()` (`TankRoster.cs:118-139`)

So the widely-quoted "13 tanks (1 mix + 6 storage + 6 doser)" is the *target preset*, and the mix tank is deliberately **outside** the roster — the roster holds at most twelve slots and the mix / Make-Ready tank is modelled separately (`Ring/Views/MainScreen/MakeReadyTank.xaml`, `Ring/ViewModels/MainScreen/MakeReadyTankViewModel.cs`). A site that has never touched Setup → Tank Roster is running the legacy four-storage default. Do not assume six of anything from a screenshot.

A roster slot (`Ring/Services/Roster/TankRosterSlot.cs`) carries:

Member	Meaning
`SlotNumber`	Position within its role's list
`Role`	`Storage` or `Doser`
`PlcIndex`	The **1-based controller array index** this slot reads and writes; `-1` means unmapped (dosers use `-1` to mean "fill me from role-byte discovery")
`Enabled`	Disabled slots issue no reads and are never written
`LegacyWindowKey`	Which legacy detail window this slot routes to — `"StorageTank1"`, `"StorageTank2"`, `"StorageTank4"`, `"Lowmidtank"`

`LegacyWindowKey` is under a **rename freeze**. Its literal string values are compared `StringComparison.Ordinal` in `RosterWriteIndex.StoragePlcIndexForLegacyWindow` (`Ring/Services/PLC/RosterWriteIndex.cs:235-261`), whose return value *is the PLC array index a storage-tank write targets*. See `ARCHITECTURE.md` for the full freeze list; chapter 04 explains what happens when the lookup cannot answer honestly.

1.3 Controllers: which address is which

Address	Device	Ring's relationship
`172.22.103.10`	CompactLogix **1769-L36ERM v20.12**	The plant controller Ring reads and (post-cutover) writes
`172.22.103.11`	PanelView	none
`172.22.103.12`	MicroLogix	none
`172.22.103.14`	Legacy RS-360 HMI PC	source of legacy data captures only
`192.168.202.15`	Bench L36ERM twin	where write behaviour is proven first

(Table reproduced from `docs/PLC_FACTS.md`, which cites the 2026-06-29 field survey.)

Two consequences that catch people:

- **Bench evidence does not automatically transfer.** Tag parity between the two controllers is an open question, and it has to be checked *per tank* — `enabl_agt` / `enabl_tvc` are true for tanks 1, 2, 8 and 9 only, so a tag can exist on both boxes and be inert on one (`docs/PLC_FACTS.md`).
- **Nothing in Ring's configuration distinguishes bench from plant.** The IP is one string. `PlcWriteEndpointGuard` (`Ring/Services/PLC/PlcWriteEndpointGuard.cs`) can tell you the endpoint is *loopback or unconfigured*, but it cannot tell you that you aimed a write at the wrong real controller. That is a procedural control, not a coded one.

The shipped `Ring/Config/appsettings.json` ships `DefaultIpAddress: ""` — an **empty** IP, not a loopback one. That matters: `PlcConnectionConfig.GetPlcIp()` falls back to `127.0.0.1` with a once-per-process warning rather than throwing (`Ring/Infrastructure/Configuration/PlcConnectionConfig.cs:93-121`), so the fall-through is the *shipped* state. Chapter 04 covers the endpoint gate that exists precisely because of this.

1.4 Solution layout

`Ring.sln` contains exactly **two** projects — there is no third assembly, no shared library, no separate test-utility project:

Project	Kind	Target	Output
`Ring/Ring.csproj`	WPF app, classic (non-SDK) csproj	`v4.7.2`, `LangVersion 8.0`	`WinExe`, `AssemblyName Ring`
`Ring.Tests/Ring.Tests.csproj`	xUnit test library, classic csproj	`v4.7.2`	`Library`, `AssemblyName Ring.Tests`

(Verified in `Ring.sln`, `Ring/Ring.csproj:8-12`, `Ring.Tests/Ring.Tests.csproj:8-11`.)

Nothing is globbed. A new `.cs`, `.xaml`, or image is invisible to the compiler until a row is hand-added to the csproj — and `Ring.Tests/ProjectMembershipGuardTests.cs` fails the build if you forget. See chapter 08.

1.5 Directory map

```
Ring/                              the WPF application
  Views/                           XAML screens + code-behind
    App.xaml / App.xaml.cs         <ApplicationDefinition>; the real entry point
    MainWindow.xaml(.cs)           the shell: chrome, content area, nav history,
                                   PLC watchdog tick, alarm pump
    MainScreen/                    process screens (Make Ready, tank groups, TVC card…)
    BatchStart/                    the four batch-arming screens + their pure policy types
    TVCcontrol/                    the four TVC per-tank control windows + bridge
    UseTanks/                      MF1 + UseTank2 + UseTank3 windows + bridge
    Process/                       Alarm screen
    Reports/                       report screens + ReportHostView (the hosted-screen shell)
    Setup/                         supervisor/admin configuration screens and dialogs
    Shifts/                        shift control, handover, handoff, production log
    Help/                          manuals library, alarm legend, contact
    Wizard/                        first-run setup wizard and its steps
    UserControls/                  NavBar, FitHost, ScreenHeader, toasts, badges, charts
    Dashboard/                     DashboardView
  ViewModels/                      the VM-driven subset (MainScreen/, UseTanks/, chrome VMs)
  Services/                        everything non-UI (see chapter 07)
    PLC/                           pollers, snapshots, readers, writers, gates
    Alarms/ Audit/ Batch/ Configuration/ Display/ Documentation/ Email/
    Export/ GroupSetup/ Notifications/ Operators/ Predictive/ Reports/ Roster/
  Database/                        DatabaseInitializer + Interfaces/ + Repositories/
  Models/                          plain data records
  Infrastructure/                  Configuration/, DependencyInjection/, Exceptions/, Security/
  Controls/                        NumericKeypad, TankLevelControl, KeypadBehavior
  Converters/                      value converters
  Validation/                      attribute + helper validators
  Resources/                       Colors/Tokens/Components/RingTheme + Strings/ (13 locales) + data/
  Images/                          screen art and icons (every file hand-registered)
  Assets/Manuals/                  the operator manual PDF that ships inside the app
  Config/appsettings.json          the config that ships; copied into the build output
Ring.Tests/                        the xUnit suite — the only test project
docs/                              see docs/INDEX.md for what is authoritative
  manual/engineering/              this book
  manual/sales/                    the commercial book
  reference/plc/                   durable tag maps + APPSETTINGS_REFERENCE
  production-readiness/            runbooks, controls records, PLC layouts, analysis
  archive/                         historical evidence — never a current-state authority
scripts/                           PowerShell + the two L5K controller exports
tools/ab_server/                   bundled libplctag AB simulator (a fake PLC for dev)
remote-view/                       MeshCentral-based remote-viewing companion package
plc-discovery/                     EtherNet/IP discovery + legacy-extract helpers
.github/workflows/                 production-gate.yml + plc-tag-audit.yml
```

Root-level documents are the ones an on-site team reads: `CUTOVER_RUNBOOK.md`, `OPERATOR_TRAINING_RUNBOOK.md`, `HYPERCARE_PLAN.md`.

1.6 Technology stack

Concern	Choice	Notes
UI	WPF on .NET Framework **4.7.2**, C# **8.0**	`Ring/Ring.csproj:11-12`
Look and feel	MahApps.Metro base styles + Ring's own token dictionaries	merge order pinned in `Ring/Views/App.xaml`
DI	`Microsoft.Extensions.DependencyInjection`	composition root is `ServiceCollectionExtensions.AddRingServices` (`Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs`)
Config	`Microsoft.Extensions.Configuration` JSON provider	`Ring/Infrastructure/Configuration/ConfigurationService.cs`
Controller I/O	**libplctag 1.5.2** + `libplctag.NativeImport`	no custom EtherNet/IP stack; see chapter 03
Persistence	`System.Data.SQLite`, hand-written ADO	no ORM, no Dapper; see chapter 05
JSON	Newtonsoft.Json	used by the journals, roster/state files, profile export
Tests	xUnit, run by `xunit.console.exe`	`dotnet test` cannot run this suite
Packages	`packages.config` + `packages/` with pinned relative `HintPath`s	a package upgrade is a csproj edit

Third-party components and their notices are enumerated in `THIRD-PARTY-NOTICES.md`.

> **Licensing is an open owner decision.** There is no `LICENSE` file in this repository and no statement of ownership or redistribution terms (`CONTRIBUTING.md`). That gap is deliberate — the terms are the owner's call — and this book does not invent one.

1.7 Running it without a plant

Three distinct mechanisms, often confused:

Mode	How	What it gives you
Demo Mode	`DemoMode.Enabled = true` in `<exe dir>\Config\appsettings.json` (or the `.local.json` overlay) + restart. **Config only — there is no CLI flag** (`docs/CONFIG_AND_STARTUP.md` §6)	Synthetic animated plant data from `Ring/Services/PLC/DemoModeData.cs`; every write short-circuits to a logged no-op
Simulated batch loop	`scripts/Run-SimDemo.ps1`	Copies the build output to an isolated directory, writes its own loopback overlay, starts `tools/ab_server` plus a heartbeat pump, launches Ring with `--isolated-sim-harness`, drives a mock batch and asserts it persisted. `scripts/Stop-SimDemo.ps1` tears it down
Synthetic history	`scripts/Seed-SmokeTestData.ps1`	Deterministic batches, usage and alarms in SQLite so every report screen has content (`scripts/SMOKE_TEST_README.md`)

`--isolated-sim-harness` is not a general "quiet mode": it only takes effect when `ReadOnlyMode` is `true` **and** the configured PLC IP is a loopback address, so it cannot be used to silence warnings against a real controller (`docs/CONFIG_AND_STARTUP.md` §6; the check is `IsIsolatedSimulationHarness` in `Ring/Views/App.xaml.cs`).

Ring is single-instance: a second launch focuses the first and exits (`Ring/Views/App.xaml.cs:21`, `113`). Chapter 02 explains why that mutex covers less than it looks like it does.

1.8 Two standing constraints you inherit on day one

Both are stated in `ARCHITECTURE.md`, and both will catch you before you have read that far.

The rename freeze (`ARCHITECTURE.md` §11)

Three identifiers are load-bearing as *strings*. Renaming any of them is a behaviour change wearing a refactor's clothes, and **none of them will fail to compile.**

Identifier	Why the string matters	Verified at
`TankRosterSlot.LegacyWindowKey` values — `"StorageTank1"`, `"StorageTank2"`, `"StorageTank4"`, `"Lowmidtank"`	Compared with `StringComparison.Ordinal` in `RosterWriteIndex.StoragePlcIndexForLegacyWindow`, **whose return value is the PLC array index a storage-tank write targets**. Rename one and a write goes to a different physical tank; make two collide and the lookup returns `NoWrite`. The same strings are hardcoded in the four TVC screens and switched on in `NavBar`	`Ring/Services/PLC/RosterWriteIndex.cs:250`; `Ring/Services/Roster/TankRoster.cs:104-113`
The `Ring.Views.BatchStart` **namespace**	`MainWindow.NavigateBack()` branches on `t.Namespace.StartsWith("Ring.Views.BatchStart", StringComparison.Ordinal)` to decide whether Back re-creates a screen or reuses the cached instance. The four Batch Start screens assume a first-time `Load`; rename the namespace and Back silently switches them to instance reuse	`Ring/Views/MainWindow.xaml.cs:640`
Screen class names	`UiScreenResolver` writes `GetType().Name` into the tamper-evident UI audit journal, and that value is part of the hashed record payload. Renaming a screen class splits its identity across the rename boundary: old records still verify, but they no longer match live labels	`Ring/Services/Audit/UiScreenResolver.cs`

If you need one of these renames, it is a change with a test and a decision behind it, not a cleanup commit.

Architectural debt that is deferred *on purpose* (`ARCHITECTURE.md` §10)

None of it is on fire, and all of it is parked **past the 2026-09-08 cutover** by an explicit decision: deep refactors of the PLC layer weeks before a write-enabled cutover trade a real risk for a cosmetic gain. Knowing the list stops you "fixing" something that is a known, scheduled item:

- **PLC layer** — no `PlcTagFactory` / `IPlcSession` abstraction (tag construction is duplicated across many sites); write paths return `bool`, tuples and several bespoke enums instead of one `PlcWriteResult`; there is no `PlcPollerBase`, so the seven pollers repeat timer/backoff/publish logic and the timers outside the coordinator stay outside it; there is no single tag-name builder and no `PlcConnectionConfig` cache; `WriteTimeout` is wired but `ReadTimeout` is what most readers resolve.
- **UI** — per-tank screen family consolidation (UseTanks → TVC → BatchStart), with three ordered prerequisites; `StorageTankGroupViewModel`'s half-finished migration; a Dashboard hardcoded around the legacy tank count.

- **Cross-cutting** — `ServiceLocator` retirement and ambient-state → DI conversion; alarm operator text is English-only in SQLite (a schema change, not a dictionary edit); C#-composed text does not live-repaint on a language change; and a git history rewrite that is **post-cutover only** because it would force a rebase of every worktree days before the plant goes live.

The forward-looking list is `docs/production-readiness/POST_CUTOVER_FOLLOWUPS.md`.

Next: 02 — Runtime Architecture.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

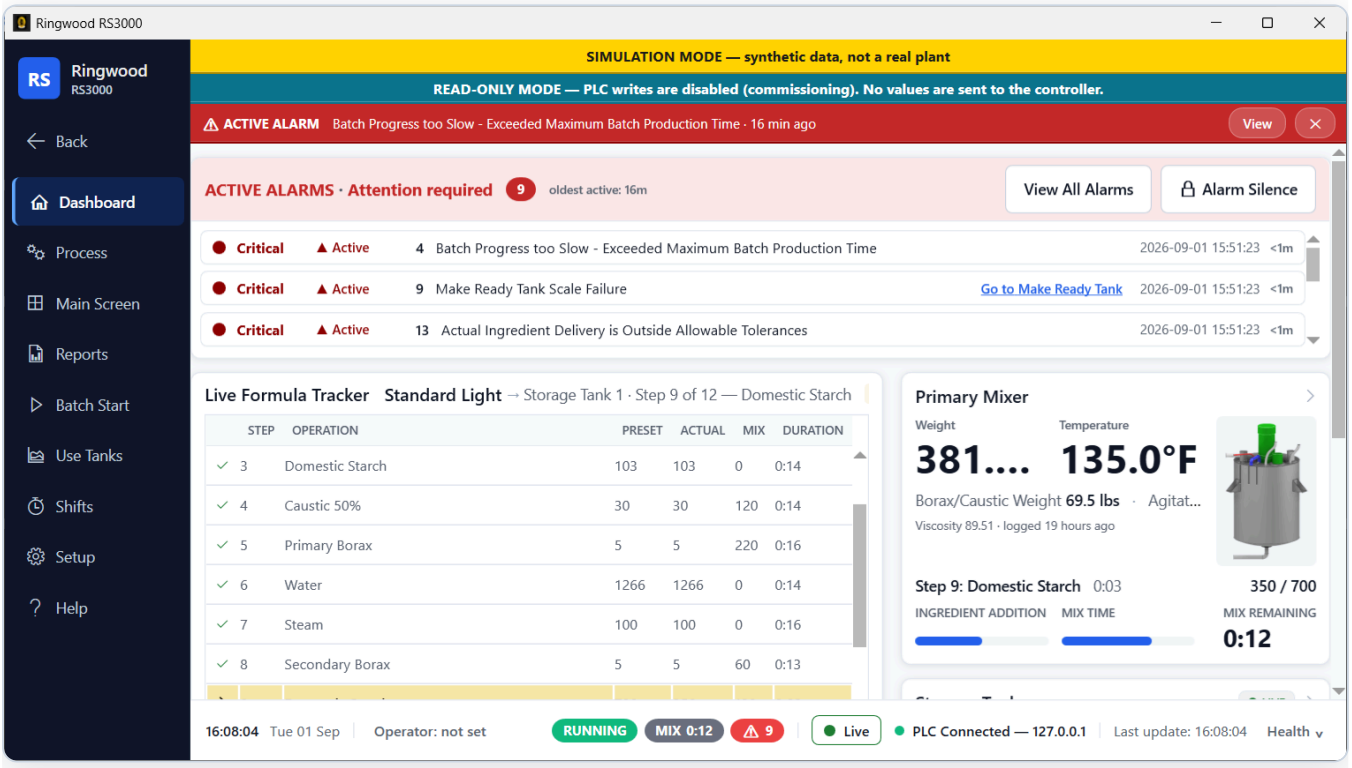
`README.md` · `ARCHITECTURE.md` · `CLAUDE.md` · `CONTRIBUTING.md` · `Ring.sln` · `Ring/Ring.csproj` · `Ring.Tests/Ring.Tests.csproj` · `Ring/Infrastructure/Configuration/AppSettings.cs` · `Ring/Infrastructure/Configuration/PlcConnectionConfig.cs` · `Ring/Config/appsettings.json` · `Ring/Services/PLC/PlcWriteGuard.cs` · `Ring/Services/PLC/PlcTagWriter.cs` · `Ring/Services/PLC/RosterWriteIndex.cs` · `Ring/Services/Roster/TankRoster.cs` · `Ring/Services/Roster/TankRosterSlot.cs` · `Ring/Services/Roster/TankRole.cs` · `Ring/Services/Display/LocalizationService.cs` · `Ring/Views/MainWindow.xaml.cs` · `Ring/Views/App.xaml.cs` · `Ring/Resources/Strings/` (directory listing) · `docs/PLC_FACTS.md` · `docs/CONFIG_AND_STARTUP.md` · `THIRD-PARTY-NOTICES.md` · `scripts/` (listing)

02 — Runtime Architecture

Who this is for. Anyone changing startup, configuration, threading, or the lifetime of a background service — and anyone debugging "it works on my machine but not on the LCP".

What you'll learn. How Ring boots and which parts of that order are load-bearing; where configuration comes from and the two ways it surprises you; what "degraded startup" means and what it takes away; the threading model, which is not one model but four; and how the subsystems actually hand data to each other.

This chapter **complements** `ARCHITECTURE.md` and `docs/CONFIG_AND_STARTUP.md`. Where those are complete — the ordered phase table, the runtime file set, the paste-ready config profiles — this chapter links rather than duplicates, and goes deeper where they are terse.



What the boot sequence in §2.1–2.2 is racing to put on screen — the dashboard is the default first view after a successful launch.

2.1 The entry point, precisely

There is no `Program.cs` and no `Main` you can read. `Ring/Ring.csproj:218` declares:

```
<ApplicationDefinition Include="Views\App.xaml">
```

so `PresentationBuildTasks` generates the `Main` into `obj/`. The first code of yours that runs is `App.OnStartup` in `Ring/Views/App.xaml.cs`.

Worth carrying precisely: the application class is `Ring.App`, not `Ring.Views.App`. The file lives at `Ring/Views/App.xaml.cs` but declares `namespace Ring` (`Ring/Views/App.xaml.cs:14`), matching `x:Class="Ring.App"` in `Ring/Views/App.xaml:1`. This is not pedantry — the generated-`Main` detail is exactly what a reachability scan has to special-case, and `WriteSurfaceRegisterTests` derives the `ApplicationDefinition` path from the csproj for that reason (`Ring.Tests/WriteSurfaceRegisterTests.cs:977-984`).

The phase list

The full ordered phase table — twenty-two phases, each with its failure mode — is in `docs/CONFIG_AND_STARTUP.md` and is not reproduced here. What this chapter adds is *why four of those orderings cannot move*.

2.2 The four load-bearing orderings

> These four orderings compile fine in any order. Three of them fail only on a plant — the compiler cannot catch a startup-sequence bug that only shows up against a real database, a real controller, or an operator who used the first-run wizard.

Ordering	What must come first	What breaks if you swap them	Code reference
(1) First-run detection	`SampleDbFileExistence()` before any repository is touched	Every later "was this a first run?" decision reads a stale answer — a database created by an incidental read looks pre-existing	`App.xaml.cs:190`
(2) The write gate	`PlcWriteGuard.Configure(...)` before any poller, window or background service exists	A window in which a write could run before the gate is configured — currently impossible only because this ordering holds	`App.xaml.cs:1115-1116`
(3) Schema before seeding	`DatabaseInitializer.Initialize()` before the seed services (alarm playbooks, maintenance tasks, email defaults)	The seeds run against tables that do not exist yet — though note the seeds also re-run and fail again, logged, if initialization itself failed	`docs/CONFIG_AND_STARTUP.md` §1 phase 15
(4) Roster baseline after the wizard	`TankRosterState.CaptureSessionBaseline()` after the first-run wizard has had its chance to edit the roster	A roster the operator just configured registers as a **mid-session roster change**, which locks every storage-tank write for the session until a restart	`App.xaml.cs:706`; `RosterWriteIndex.cs:76-91`

The detail behind each row:

(1) `SampleDbFileExistence()` before anything opens the database

`Ring/Views/App.xaml.cs:190` calls `ServiceLocator.GetService<DatabaseInitializer>()?.SampleDbFileExistence()` immediately after DI is built and before any repository is touched.

The reason is that **opening a SQLite connection creates the file.** Every later "was this a first run?" decision — whether to show the setup wizard, whether the pre-migration backup is of a real database or of a freshly created empty one stamped `user_version = 0` — reads the latch this call sets (`Ring/Database/DatabaseInitializer.cs:153-171`). Sample it late and the answer is always "not a first run".

This is a trap with a wide blast radius, because a *read* is enough to trip it: every repository call begins with `EnsureTable()`, which issues `CREATE TABLE / CREATE INDEX IF NOT EXISTS`. `DatabaseMaintenanceLatch` (`Ring/Services/DatabaseMaintenanceLatch.cs`) exists because a dashboard refresh tick running inside a modal message box's nested dispatcher pump could recreate the file mid-factory-reset — see chapter 05.

(2) `PlcWriteGuard.Configure` before any poller or window exists

`RunStartupConfigurationValidation()` is phase 9, and the *first thing it does after resolving the logger and settings* is:

```
Ring.Services.PLC.PlcWriteGuard.Configure(
    Ring.Services.PLC.PlcWriteGuard.ResolveConfiguredReadOnly(settings));
```

(`Ring/Views/App.xaml.cs:1115-1116`.) It runs before `DatabaseInitializer`, before any background service, before `MainWindow`. There is therefore **no window in which a write could precede the gate**. Note the ordering *inside* the method too: the guard is configured before the config-load-error branch and before validation runs, so even a boot that is about to abort has already closed the gate.

(3) `DatabaseInitializer.Initialize()` before the seed services

Alarm playbooks, maintenance tasks and email defaults are seeded after the tables exist. A subtlety worth knowing: the seeds *also run when database initialization failed* — they simply fail again and log (`docs/CONFIG_AND_STARTUP.md` §1, phase 15).

(4) `TankRosterState.CaptureSessionBaseline()` after the wizard

`Ring/Views/App.xaml.cs:706` captures the roster baseline *after* the first-run wizard has had its chance to edit it. Capture it earlier and a roster the operator just configured registers as a **mid-session roster change**, which latches `RosterWriteIndex.StorageWritesLockedByRosterChange` (`Ring/Services/PLC/RosterWriteIndex.cs:76-91`) and blocks every storage-tank write for the session. The recovery is a restart; the cause would be invisible.

2.3 Configuration

Load order

`Ring/Infrastructure/Configuration/ConfigurationService.cs` builds:

```
<exe dir>\Config\appsettings.json        required   (optional: false)
<exe dir>\Config\appsettings.local.json  optional overlay, gitignored
```

both with `reloadOnChange: false` — deliberately, because a file watcher rethrowing a `FormatException` on a thread-pool thread would kill the process (`docs/CONFIG_AND_STARTUP.md` §2).

Two behaviours from that file are worth internalising:

- **Arrays merge by index, not by replacement.** Override one SMTP recipient in the overlay and you get *your one plus the base file's second and third*. Two keys are re-overridden wholesale to defuse this (`ConfigurationService.ApplyLocalArrayOverrides`): `AlarmEscalation.Smtp.To` and `AlarmEscalation.Webhook.AllowedHosts`. **Every other array in the file is still index-merged.**
- **`NormalizeConnectionStringDbPath`** rewrites a relative `Data Source=` to `<exe dir>\<name>` at load time, because repositories open the connection string verbatim and a launch from a different working directory would silently create a *second* database. Absolute paths and `:memory:` are untouched.

The PLC IP is *not* restart-only — and that is deliberate

The blanket statement "config changes need a restart" is true of the DI-bound `AppSettings` object. It is **not** true of the two PLC connection values, and the difference is easy to trip over.

`PlcConnectionConfig.GetPlcIp()` (`Ring/Infrastructure/Configuration/PlcConnectionConfig.cs:25-121`) builds a *fresh* `ConfigurationBuilder` over `Config\appsettings.json` + `Config\appsettings.local.json` **on every call**, and only falls back to the DI-resolved `AppSettings` if the file yielded nothing. `GetPlcPath()` mirrors the same pattern (`PlcConnectionConfig.cs:127-205`). Since every libplctag reader and writer resolves the endpoint through these, a change to `PlcSettings:DefaultIpAddress` on disk takes effect on the **next tag construction**, without a restart. The class comment states the intent explicitly: "Reads from file first so mode-switch popup and new readers always see current config."

Because `GetPlcIp()` is on the hot polling path, its loopback-fallback warning is latched to **once per process** via `Interlocked.Exchange` (`PlcConnectionConfig.cs:110-120`). Do not read a single warning line as "this happened once".

`PlcSettings.DefaultPath` binds to nothing

The shipped `Ring/Config/appsettings.json` contains `"DefaultPath": "1,0"`, but **`Ring/Infrastructure/Configuration/AppSettings.cs` declares no `DefaultPath` property on `PlcSettings`** (verified: the only code references to that key are inside `PlcConnectionConfig.cs`). The key is consumed exclusively by `PlcConnectionConfig.GetPlcPath()`, which reads it as a raw configuration key

with a compiled `"1,0"` default (`PlcConnectionConfig.cs:152-179`). If you are looking for a strongly-typed `settings.PlcSettings.DefaultPath`, it is not there and adding one would create a second, divergent source of truth.

Configuration is written back into the build output

Ring persists operator-entered settings to `<exe dir>\Config\appsettings.json` — that is, into `bin\<cfg>\`. Four classes under `Ring/Infrastructure/Configuration/` touch it — `AlarmEscalationConfigWriter`, `AnalyticsSettingsConfigWriter`, `CostSettingsConfigWriter` and `AppSettingsSectionWriter` — plus `Infrastructure/Security/CredentialRotationService`, which writes the `.local.json` overlay the same way. There is exactly **one** atomic-write implementation among them, not four or five: `AlarmEscalationConfigWriter.WriteAtomically` (`AlarmEscalationConfigWriter.cs:88-116`, the `File.Replace` call at `:108`) does the temp-file + `File.Replace` + one-deep `.bak` dance, and `AnalyticsSettingsConfigWriter.cs:61`, `CostSettingsConfigWriter.cs:50` and `AppSettingsSectionWriter.cs:65, 103` all call into that same static method rather than repeating it.

The trap is the build, not the write: `Ring/Ring.csproj:1985` declares

```
<Target Name="CopyAppSettingsToOutput" AfterTargets="Build">
```

which copies `Ring/Config/appsettings.json` over the output copy after **every** build. PLC IP, `ReadOnlyMode`, SMTP credentials and cost rates an operator typed in are reverted by the next build. The `.local.json` overlay is copied only for **Debug** and only if present in `Ring/Config/` (`Ring/Ring.csproj:1881`); a **Release** build deliberately *keeps* an existing overlay in the output and emits a warning, with `/p:PurgeLocalConfig=true` as the opt-in that deletes it (`Ring/Ring.csproj:1995-1999`). A release package can therefore silently carry a developer's bench PLC IP if nobody reads the warning — which is why `Invoke-ProductionGate.ps1` passes `/p:SuppressLocalConfigWarning=true` only for its own isolated build (`scripts/Invoke-ProductionGate.ps1:70`) and why `scripts/Preflight-FieldKit.ps1` re-checks the deployed copy.

The complete "everything Ring stores on disk" table — ten stores across three roots, and which single one the backup covers — is `docs/CONFIG_AND_STARTUP.md`.

Startup validation and the countdown dialog

`ConfigurationValidator.Validate` returns errors and warnings; `Result.IsFatal => Errors.Count > 0` (`Ring/Infrastructure/Configuration/ConfigurationValidator.cs:23`).

- **Fatal** → message box, `Shutdown(1)`. Blocks forever, by design.
- **Warnings** → `StartupWarningDialog.ShowWarning(..., timeoutSeconds: 60)` (`Ring/Views/App.xaml.cs:1168-1170`) — a countdown that self-continues so a 3 a.m. unattended restart is never parked on a human click, while a present operator can still choose Quit.
- Suppressed only under `--isolated-sim-harness`, which itself only applies with `ReadOnlyMode` true and a loopback IP.

A **write-enabled station always produces at least one warning**: any configuration with writes enabled adds the single-writer advisory (`ConfigurationValidator.cs:281`, `SingleWriterAdvisory()` at `:507`). Expect the countdown dialog on every post-cutover boot; its absence is the anomaly.

Production mode — the strict profile

Everything in the previous subsection describes `ConfigurationValidator`'s **advisory** mode, which is what a normal `Ring/Config/appsettings.json` install runs under. There is a second, stricter mode that this book did not previously name: `AppSettings.Production` (`ProductionSettings`, `Ring/Infrastructure/Configuration/AppSettings.cs:119-131`), whose `Enabled` flag defaults `false`. Setting it `true` — as the signed `Ring/appsettings.production.template.json` profile does — switches `ConfigurationValidator.ValidateProduction` (`ConfigurationValidator.cs:67-114`) from a no-op into nine additional **fatal** checks, each a message box and `Shutdown(1)` if it fails:

Check	Line
`Production.SiteId` is set (not blank, not a placeholder)	`:73`
`Production.CommissioningApprovalId` is set	`:74`
`Production.PlantProfileSha256` is a valid SHA-256	`:75`
`Production.ControllerProjectSha256` is a valid SHA-256	`:76`
`Production.LegacyConfigurationSha256` is a valid SHA-256	`:77`
`PlcSettings.DefaultIpAddress` is a parseable, **non-loopback** literal IP — blank, unparseable, or loopback all fail	`:83-88`
`Production.RequireWriteEnabled` and `PlcSettings.ReadOnlyMode` are not both true at once (an inconsistent signed profile)	`:90-91`
`DemoMode.Enabled` and `Authentication.EnableDemoMode` are both off	`:93-94`
Supervisor and administrator credentials are not blank or a known placeholder (the legacy `"9999"` default, or a `REPLACE_/<...>` template value)	`:96-102`
`Database.ConnectionString`'s Data Source is an absolute path	`:104-106`
`DatabaseMaintenance.BackupDirectory` is set and absolute	`:108-113`

That is ten checks, not nine, once the SHA-256 triad is counted individually — the point to take away is that every one of them is **dormant** until `Production.Enabled` is `true`. `Production.RequireWriteEnabled` is the signed-instant-cutover companion flag: it errors (`:90-91`) if `PlcSettings.ReadOnlyMode` is still `true`, so a profile cannot claim write-enabled and read-only at once. `CUTOVER_RUNBOOK.md` walks the team through this distinction explicitly: the ordinary cutover step installs `appsettings.local.json` with `ReadOnlyMode` still `true` and `Production.Enabled` **not yet set**, so these fatal checks are not yet armed and

`DemoMode.Enabled` has to be verified by hand; only the later, separately supervised write-enable step installs the signed profile with `Production.Enabled: true` and `Production.RequireWriteEnabled: true` together with `ReadOnlyMode: false` (`CUTOVER_RUNBOOK.md:88-109, 241-253`; see `STARCH_SYSTEM_CUTOVER_RUNBOOK.md` §2.1 "Enable writes" for that step). Fill in `Ring/appsettings.production.template.json` without also setting `Production.Enabled` and you get none of these protections and no warning that they are missing — the template does not switch itself on.

The single-writer problem, stated honestly

Every gate in the write stack reasons about *this process only*, and the controller arbitrates nothing. Two write-enabled Rings would drive the same setpoints against each other with no detection anywhere.

The single-instance mutex does **not** close this. `SingleInstanceMutexName` is `"RS360-RingwoodApp-SingleInstance"` with **no `Global\` prefix** (`Ring/Views/App.xaml.cs:21`), so Windows resolves it in the per-session `Local\` namespace: a second Ring launched under a different login on the *same* LCP — an engineer signing in while the kiosk account runs Ring, or a MeshCentral/RDP seat — creates its own mutex and boots all the way through. Because none of it is detectable, `ConfigurationValidator` **states** it as a warning rather than enforcing it. Keeping one write-enabled Ring per controller is a procedural rule. (Full framing: `docs/CONFIG_AND_STARTUP.md`.)

2.4 Degraded startup

"Degraded" has one precise meaning in this codebase: **database initialization failed**, so batch history, the audit trail and the alarm log are unreliable for this session.

The response is not a dialog you dismiss. `StartupDegradedState.ApplyDatabaseInitOutcome(false, detail)` (`Ring/Services/StartupDegradedState.cs`) does two things:

1. Calls `PlcWriteGuard.ForceReadOnlyForSession()` — the **one-way latch**. Once pulled, `Configure` computes `_readOnly = readOnly || _forcedReadOnly` (`Ring/Services/PLC/PlcWriteGuard.cs:37-40`), so nothing in the app can re-arm writes short of a restart.
2. Records the reason, first-detail-wins, so `MainWindow` can paint a persistent banner for as long as the app runs.

The rationale is stated in the class doc and is worth repeating: **a Ring that cannot record what it did is not allowed to do anything.** A successful init changes nothing at all — the configured `ReadOnlyMode` keeps deciding.

A second, milder degradation: if the tank roster fails to load, startup falls back to the legacy 4-tank default, warns, **and** latches `RosterWriteIndex.StorageWritesLockedByRosterLoadFailure`, which blocks storage-tank writes even though the substituted roster looks perfectly valid (`Ring/Services/PLC/RosterWriteIndex.cs:98-139`). Chapter 04 explains why that needs its own lock.

2.5 Threading and the dispatcher

Ring does not have one concurrency model; it has four, and they meet at well-defined places.

Model	Where	Rules
WPF dispatcher (UI thread)	Every screen, `MainWindow`, `NavBar`, all `DispatcherTimer` ticks	Anything touching a `Control` or a `DispatcherTimer` must be here
`System.Threading.Timer` background pollers	The seven pollers under `Ring/Services/PLC/`, `ProcessHistorianService`, `ReportSchedulerService`, the alarm background pump	Explicitly "never runs on UI thread; no Dispatcher" (e.g. `Ring/Services/PLC/PlcSnapshotPoller.cs` class doc). Each guards re-entrancy with an `Interlocked/CompareExchange` busy flag so a slow cycle skips a tick rather than stacking
Thread-pool tasks	`PlcSetpointWriteQueue` pumps, `MomentaryPulse.ArmBackgroundReclear`, startup batch reconcile	Detached; must never assume UI affinity
Process-wide static state under a lock	`PlcHeartbeatConnectionTracker`, `PcReadIntegerCache`, the snapshot holders, `TankInstalledWriteGate`, `PlcWriteGuard`	Readable from any thread; the gates use `volatile` / `Volatile` / `lock` so a reader can never see a torn value

Three concrete patterns you will meet:

Screens pull, they are not pushed. Every PLC-facing screen owns a `DispatcherTimer` (typically 1000 ms) that reads a static snapshot holder and assigns values into named controls. There is no event bus from poller to screen. The end-to-end latency for a value is therefore *poller cadence* + *screen cadence*. Chapter 03 gives the per-poller numbers.

Blocking I/O never runs on the dispatcher. The startup batch reconcile is the canonical example: it does a blocking libplctag read of `current_step` (up to `PlcSettings.ReadTimeout` per attempt, retried three times ~500 ms apart), so `PlcPollingCoordinator.Start()` defers it to `Task.Run` (`Ring/Services/PLC/PlcPollingCoordinator.cs:165-180`) after an earlier version froze the UI for seconds on a reachable-but-slow controller.

Marshalling back is explicit. `MainWindow.ProbeAndStartPlcMonitoring` marshals `StartPlcMonitoring()` onto the dispatcher because that method touches a `DispatcherTimer` and, on failure, a `MessageBox` (`Ring/Views/MainWindow.xaml.cs:828`).

Timers owned by `MainWindow`

`MainWindow` is not just chrome; it owns three lifetimes:

Timer	Cadence	Purpose
`_plcUpdateTimer` (`DispatcherTimer`)	2000 ms	The UI-side PLC tick; also calls `PlcHeartbeatConnectionTracker.EvaluateWatchdog()` (`MainWindow.xaml.cs:1287`)
PLC alarm background pump (`System.Threading.Timer`)	2 s (`PlcAlarmBackgroundPumpInterval`, `MainWindow.xaml.cs:170`)	Drives `AlarmAlarmNumberPlcService` independently of the dispatcher, so a frozen UI thread cannot leave alarms frozen-but-green
PLC auto-reprobe (`System.Threading.Timer`)	30 s (`PlcReprobeIntervalMs`, `MainWindow.xaml.cs:62`)	Re-runs the reachability probe and restarts monitoring when the controller comes back

Plus a UI-hang sentinel: a background timer pings and expects a dispatcher "pong" every 5 s (`UiHangPingIntervalMs`, `MainWindow.xaml.cs:122`), feeding `Ring/Services/UiHangDetector.cs`.

The dual-drive on the alarm pump is deliberate and has a matching hazard: the same method is pumped from both the UI timer and the background timer, so a `CompareExchange` busy gate stops them doubling up (`ARCHITECTURE.md`).

2.6 Dependency resolution — four mechanisms, one recommendation

Ring resolves dependencies four different ways and all four are live. This is the largest structural inconsistency in the codebase and `ARCHITECTURE.md` is the authority on it. In brief:

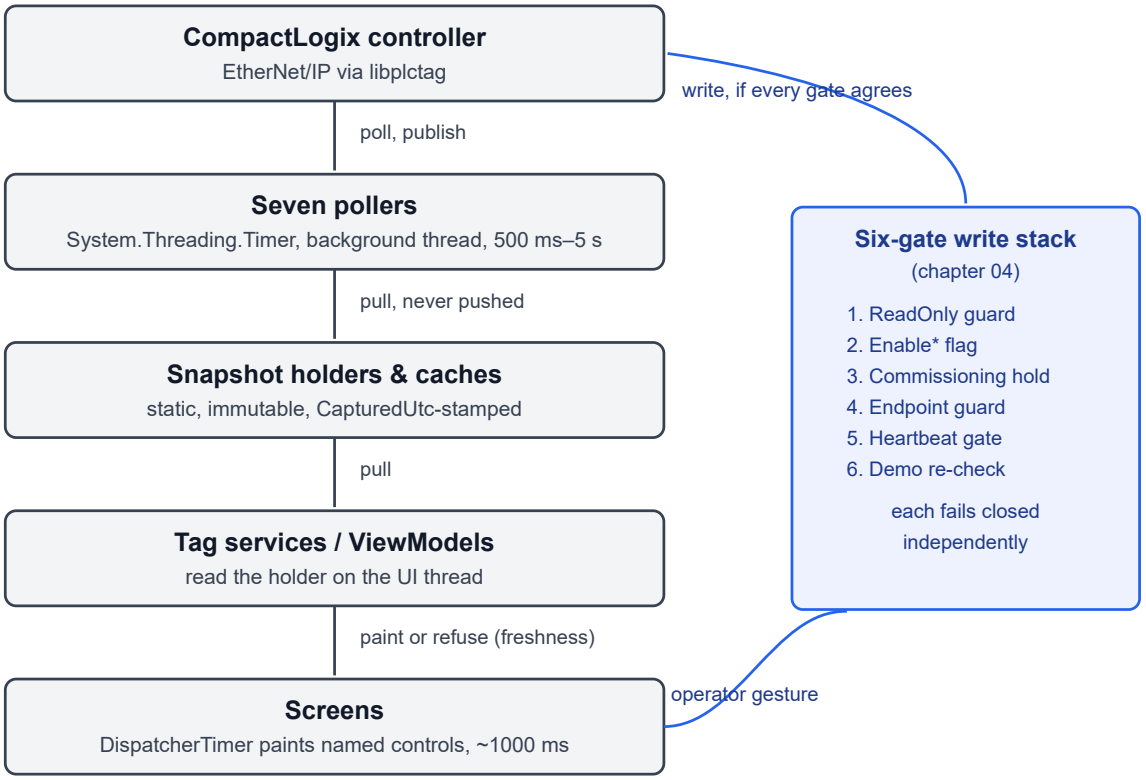
1. **The DI container** — `AddRingServices` (`Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs`) is the real composition root; it is built once in `App.OnStartup` (`Ring/Views/App.xaml.cs:1184-1197`) and holds on the order of a hundred registrations (read the file for the current count — do not quote one).
2. **`ServiceLocator`** — a three-method static wrapper over that container (`Ring/Infrastructure/DependencyInjection/ServiceLocator.cs`). Its own summary says "*This is a temporary solution. In production, prefer constructor injection.*" It exists because a XAML-constructed `UserControl` has no constructor-injection point.
3. **Ambient static `*State` classes** — e.g. `TankRosterState`, `GroupHardwareSetupState`, `ProcessHoldState`, `StartupDegradedState`.
4. **Process-wide `Instance` singletons** — e.g. `ToastService.Instance`, `PlaybookService.Instance`, `PlcSetpointWriteQueue.Instance` (`Ring/Services/PLC/PlcSetpointWriteQueue.cs:124`).

What new code should do: register in `AddRingServices` and inject through the constructor wherever there is one. From XAML-constructed code-behind, resolve once in `Loaded` and hold the reference. **Do not add a new static `*State` class or a new `Instance` singleton** — that is the pattern being retired.

One important asymmetry to be aware of when you are tempted to make a gate injectable: the gates are static *on purpose*. `PlcWriteGuard.LogBlocked` explicitly avoids reaching into `ServiceLocator` and constructs its own `Logger` as a last resort (`Ring/Services/PLC/PlcWriteGuard.cs:86-103`) so the safety path never depends on the container being built. `PlcTagWriter`, `PlcTagReader` and `PlcWriteEndpointGuard` all wrap their `ServiceLocator` lookups in `try/catch` and treat an unbuilt container as the *fail-closed* answer (`PlcWriteEndpointGuard.LoopbackWritesAuthorized`, `PlcWriteEndpointGuard.cs:190-201`).

2.7 How the subsystems fit together

The three narratives below share one shape: reads flow one direction, through snapshots, and writes take a separate, narrower path that never touches them.



A single narrative of one value's journey, and one command's:

A tank temperature, controller → screen. `PlcPollingCoordinator.Start()` constructs `StorageTankGroupPoller` with the roster's storage slots snapshotted at construction (`Ring/Services/PLC/StorageTankGroupPoller.cs`). Every 2 s its `System.Threading.Timer` callback — if `AllowHeavyPlcPolling()` says the link is `Connected` or `Stale` — calls `StorageTankTagReaderService` for each mapped, enabled slot, builds an **immutable** `StorageTanksSnapshot` stamped with `CapturedUtc`, and publishes it into the static `StorageTanksSnapshotHolder`. The Storage Tank Group screen's own

`DispatcherTimer` reads `Holder.Latest`, classifies its freshness (`Ring/Services/Display/DataFreshnessClassifier.cs`) and either paints the value or refuses to. Nothing pushed; nothing bound to a poller.

A Hold command, operator → controller. `NavBar`'s shared button handler routes `HoldButton` to `OnProcessHoldClicked()` (`Ring/Views/UserControls/NavBar.xaml.cs:513-516`), which reaches `PcWriteInteger30BatchControlPlcService`. That service checks its gates, asserts `PC_Write_Integer[30].1` through `PlcTagWriter.Write`, sleeps, then clears it through `MomentaryPulse.ClearWithRetry` → `PlcTagWriter.WriteClear`. Each of those hops is a gate; chapter 04 walks every one.

A batch, controller → SQLite. `BatchStepsPoller` observes `current_step` edges and drives `Ring/Services/Batch/BatchLifecycleRecorder.cs`, which on batch end calls `BatchRepository.CompleteWithUsage`. Completion and every `IngredientUsage` row commit in one transaction, so a failed usage insert leaves the batch `Running` rather than completing with missing usage (`ARCHITECTURE.md`).

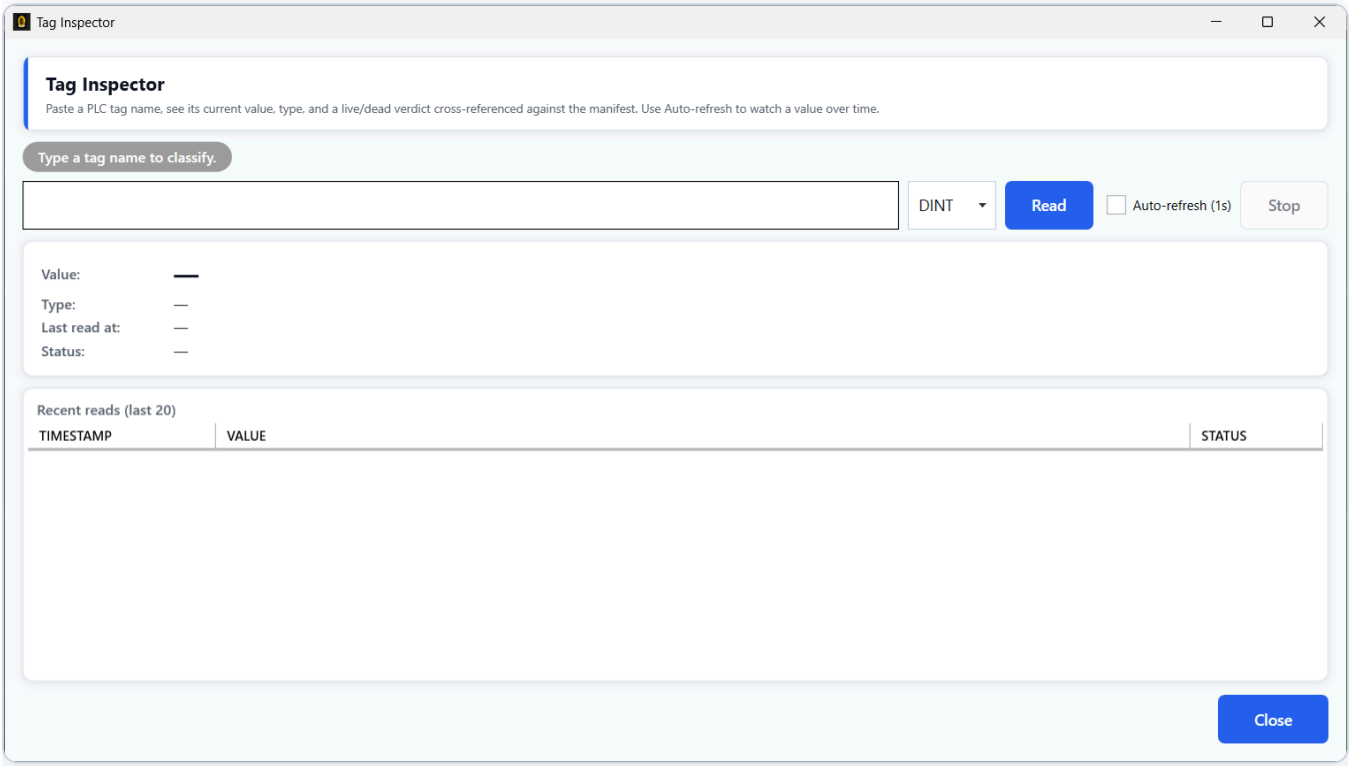
Next: 03 — PLC Communications.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring/Views/App.xaml.cs` · `Ring/Views/App.xaml` · `Ring/Ring.csproj` · `Ring/Infrastructure/Configuration/ConfigurationService.cs` · `Ring/Infrastructure/Configuration/ConfigurationValidator.cs` · `Ring/Infrastructure/Configuration/PlcConnectionConfig.cs` · `Ring/Infrastructure/Configuration/AppSettings.cs` · `Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs` · `Ring/Infrastructure/DependencyInjection/ServiceLocator.cs` · `Ring/Services/StartupDegradedState.cs` · `Ring/Services/PLC/PlcWriteGuard.cs` · `Ring/Services/PLC/PlcPollingCoordinator.cs` · `Ring/Services/PLC/PlcSetpointWriteQueue.cs` · `Ring/Services/PLC/RosterWriteIndex.cs` · `Ring/Services/PLC/StorageTankGroupPoller.cs` · `Ring/Views/MainWindow.xaml.cs` · `Ring/Views/UserControls/NavBar.xaml.cs` · `Ring/Database/DatabaseInitializer.cs` · `Ring/Infrastructure/Configuration/AlarmEscalationConfigWriter.cs`, `AnalyticsSettingsConfigWriter.cs`, `CostSettingsConfigWriter.cs`, `AppSettingsSectionWriter.cs` · `Infrastructure/Security/CredentialRotationService.cs` · `docs/CONFIG_AND_STARTUP.md` · `CUTOVER_RUNBOOK.md` · `ARCHITECTURE.md`

03 — PLC Communications: the read path



The Tag Inspector — reading live tags without Studio 5000.

Who this is for. Anyone adding a read, debugging a stale or dark value, diagnosing a session-starvation burst, or trying to work out why a screen shows "no data" instead of a number.

What you'll learn. How Ring actually talks to a CompactLogix through libplctag; every poller and its cadence; the timers that live *outside* the polling coordinator and will otherwise cost you an afternoon; how snapshots, caches and freshness gating work; the index maps; the heartbeat state machine and the two different questions it answers; the diagnostic services around the link; and how demo mode substitutes for all of it.

The write path is chapter 04. Controller facts that older documents get wrong are in `docs/PLC_FACTS.md` — read it once before you form any belief about the ladder.

3.1 Transport: libplctag, and what Ring does *not* do

Ring uses **libplctag 1.5.2** with `libplctag.NativeImport`. There is no custom EtherNet/IP implementation and no CIP code in this repository.

The shape of every read is the same (`Ring/Services/PLC/PlcTagReader.cs:276-300` is representative):

```
using (var tag = new Tag<IntPlcMapper, short>
{
```

```
    Name      = _tagName,
    Gateway   = _ip,               // PlcConnectionConfig.GetPlcIp()
    Path      = _path,             // PlcConnectionConfig.GetPlcPath(), default "1,0"
    PlcType   = PlcType.ControlLogix,
    Protocol  = Protocol.ab_eip,
    Timeout   = TimeSpan.FromMilliseconds(_readTimeout)
})
{ ... }
```

Three consequences follow directly:

- **A fresh `Tag` per read, then dispose.** There is **no C#-level session pool**. Session reuse is entirely libplctag's business, which is why session-count pressure shows up as a transport symptom (`ErrorTimeout` bursts) and not as anything you can see in Ring's own code.
- **The controller has very few sessions.** A CompactLogix caps at roughly four to eight EtherNet/IP sessions per source IP (`docs/PLC_FACTS.md`). This is the *reason* the polling coordinator staggers poller startup and the reason every field script warns before opening a session against the live box.
- **`Path` is the CPU slot route.** `"1,0"` = port 1, slot 0. It is resolved by `PlcConnectionConfig.GetPlcPath()` reading the config **file** on every call — see chapter 02 §2.3.

`PlcTagReader` — the per-tag reader

`Ring/Services/PLC/PlcTagReader.cs` implements `IPlcTagReader`. Its read timeout comes from `PlcSettings.ReadTimeout`, defaulting to `3000` if config is unavailable (`PlcTagReader.cs:39`) — but the shipped `Ring/Config/appsettings.json` sets `ReadTimeout` to `2500`, so the *effective* value on a normal install is lower than the compiled default; read the file, not this sentence, for the number that is actually in force. Two of its behaviours are non-obvious:

BOOL array elements are read as DINTs and bit-extracted. For a tag like `Alarm_Triggers[1]`, `TagBool` does not work for individual array elements. The reader computes `dintIndex = arrayIndex / 32`, `bitPosition = arrayIndex % 32`, reads `Alarm_Triggers[dintIndex]` as a `TagDint`, and masks the bit (`PlcTagReader.cs:205-249`). Non-array BOOL tags take the direct `TagBool` path.

INT is 16 bits and must be read as such. Both scalar tags (`current_step`) and UDT members must use `Tag<IntPlcMapper, short>`; a `TagDint` (32-bit) read against a 16-bit INT returns `ErrorOutOfBounds` (`PlcTagReader.cs:279-284`).

UDT members are read by byte offset. libplctag cannot address a UDT member, so `PlcTagReader.ReadUdtMember()` (`PlcTagReader.cs:86-152`) parses `Tanks[n].Member`, hands off to `UdtTankReader`, and extracts the field from the whole-UDT byte buffer using the offsets in `Ring/Services/PLC/UdtTankLayout.cs`:

Member	Byte offset	Source
`Request_level`	16	`UdtTankLayout.cs:38`
`Tank_Num` ("Spare")	18	`:39`
`Formula_number`	20	`:40`

Member	Byte offset	Source
`Batch_Size`	22	`:41`
`Start_Type`	24	`:42`
`Splitting_A`	26	`:43`
`Splitting_B`	28	`:44`
whole-UDT size	184 bytes **(provisional — see note)**	`UdtTankLayout.TankUdtSizeBytes`, `:68`

The 184-byte size is provisional in a way the seven member offsets are not. The class doc marks it explicitly: it is walked from the L5K `DATATYPE Tank` block with standard ControlLogix alignment rules, but it "has not been byte-confirmed against a live controller read; it is used only to size a diagnostic read buffer, never to place a write" (`UdtTankLayout.cs:46-67`). The seven offsets Ring actually reads and writes (16–28) sit in the leading tightly-packed region and are exact regardless.

`UdtTankLayout` additionally maintains a `VerifiedOffsets` set and an `IsVerifiedOffset(byteOffset)` predicate (`:70-85`) — a distinction between "we walked this out of the L5K `DATATYPE Tank` block" and "we guessed", not between "computed" and "round-tripped against a controller": `IsVerifiedOffset`'s own doc says it is true "only for byte offsets that correspond to a member **verified against the L5K Tank UDT**" (`:83-84`) — a static export, not a live controller read. The write path uses the predicate to refuse a guessed offset; it does not claim the offset was proven on hardware. The extract/insert primitives (`ExtractInt16` / `InsertInt16`, `:93-115`) are pure math with explicit bounds checks, so they are unit-testable without a PLC.

If a UDT layout changes on the controller, `UdtTankLayout` is what must change here. Nothing detects the drift automatically.

> A second UDT hazard: bit-packed SINT members are exposed by the controller under *generated alias names* of the form `ZZZZZZZZZZ<suffix>` — e.g. `Tanks_c[n].ZZZZZZZZZZTank_c35`. Reading the "obvious" member name instead of the alias returns `ErrorBadParam` and ships a permanently dark indicator, which has happened here more than once (`docs/PLC_FACTS.md`; the alias sweep is `docs/reference/plc/UDT_ALIAS_SWEEP.md`). Grep the L5K `DATATYPE` block for the verbatim name before trusting any UDT member string.

3.2 The polling coordinator

`Ring/Services/PLC/PlcPollingCoordinator.cs` is the single façade for background polling. `Start()` (`:132-200`) does, in order:

1. `Stop()` — always, so `Start` is idempotent and never leaks a timer.
2. `PlcHeartbeatConnectionTracker.Reset()`.
3. Resolve `EnableBatchStartPolling` from settings (default `true`).
4. Construct and start the pollers **staggered**.

5. Fire the startup batch reconcile and controller-stamp catch-up on a background `Task` (`:165-180`).

The seven pollers

Poller	Cadence	Startup delay	Reads → publishes
`PlcSnapshotPoller`	500 ms (backs off to 5000 ms after 2 consecutive failed live cycles)	0	`PC_Read_Float[0..34]` as one REAL array → `PcReadFloatSnapshot` + `MakeReadyTankPlcData`; piggybacks `PC_Display_Outputs[0..31]` → `PcDisplayOutputsCache` on the same cycle
`PcReadIntegerPoller`	**two timers**: 1500 ms heartbeat, 5000 ms full block	0	heartbeat `PC_Read_Integer[0]` → `PlcHeartbeatConnectionTracker`; full `[0..479]` → `PcReadIntegerCache`
`BatchStepsPoller`	2000 ms	150 ms	`Batch_Current_Preset_*` arrays + `current_step` → `BatchStepsSnapshotHolder`; drives `BatchLifecycleRecorder`
`StorageTankGroupPoller`	2000 ms	300 ms	per roster storage slot, `Tanks[PlcIndex]` fields → `StorageTanksSnapshotHolder`
`UseTankGroupPoller`	2000 ms	450 ms	per roster doser slot (pinned or discovery-filled) → `UseTanksSnapshotHolder`
`TVCPoller`	2000 ms	600 ms	`TVC[PlcIndex]`, `TVC_Ctrl`, `TVC_IO` → `TVCSnapshotHolder`
`BatchStartPoller`	2000 ms, **only when `PlcSettings.EnableBatchStartPolling`**	750 ms	`Tanks[PlcIndex]` batch-start members → `BatchStartSnapshotHolder`

(Cadences and delays verified in `PlcPollingCoordinator.cs:141-199`; backoff constants in `PlcSnapshotPoller.cs` and `PcReadIntegerPoller.cs`.)

What each poller actually asks the controller for

Cadence alone does not tell you the cost of a poller. These are the tag names, read out of the reader services:

`PlcSnapshotPoller` — one array read of `PC_Read_Float` (35 REALs, indices 0–34) plus `PC_Display_Outputs` on the same cycle (`PlcSnapshotPoller.cs:251, 334`). Two reads per tick, not thirty-five: it is the cheapest poller per value in the app. Its own read timeout is a local `ReadTimeoutMs = 2000`, and after `FailuresBeforeBackoff = 2` consecutive failed live cycles it drops to `BackoffIntervalMs = 5000`.

`PcReadIntegerPoller` — two independent timers with independent backoff:

Timer	Reads	Timeout constant
heartbeat, 1500 ms	`PC_Read_Integer[0]` only	`HeartbeatReadTimeoutMs = 1000` — "keep short so disconnect is felt quickly"
full block, 5000 ms	`PC_Read_Integer[0..479]` in one bulk read, with an indexed fallback	`BulkReadTimeoutMs = 3000`, `IndexedReadTimeoutMs = 1500`

Backoff is separate per timer (`HeartbeatBackoffIntervalMs = 5000`, `FullDataBackoffIntervalMs = 8000`, two failures each). The full block only runs while `AllowHeavyPlcPolling()` is true; the heartbeat always runs, because it is what *decides* that.

`StorageTankGroupPoller` → `StorageTankTagReaderService`, per mapped storage slot: `Tanks[n].Level` (with a documented fallback to `Tanks[n].LA_Weight` when `Level` fails, `:75-81`), `Tanks[n].Formula_number` (`:174`), `Tanks[n].Current_Temp` (`:197`), `Tanks[n].Request_level` (`:220`), `Tanks[n].Preset_Temp` (`:396`), `Tanks_c[n].nominal_Size` (`:369`).

> One sensor-honesty detail worth copying: a `Level` reading below `SensorFaultEpsilon` is treated as a **level-transmitter fault, not 0 gal**, and the tank's level is *not advanced* (`StorageTankTagReaderService.cs:98`). A zero that means "broken" must never render as a zero that means "empty".

`UseTankGroupPoller` → `UseTankTagReaderService`, per doser slot: `Tanks[n].Level` / `.LA_Weight` / `.Formula_number` / `.Current_Temp` / `.Request_level` / `.Preset_Temp` / `.Status` / `.Fill_from_tk` / `.Liq_*`, plus the commissioning bytes `Tanks_c[n].ZZZZZZZZZZTank_c17`, …`Tank_c35`, …`Tank_c44`, `Tanks_c[n].LA1_pump` and `Tanks_c[n].nominal_Size` (line numbers as listed in `UseTankTagReaderService.cs`). It also runs the **role-byte discovery** that fills doser slots whose roster `PlcIndex` is `-1`, re-validating periodically and *merging* results — because a single pass cannot distinguish a transient role-byte read timeout from "not a Use Tank", so caching the first non-empty result forever would hide a real doser until restart.

`TVCPoller` → `TVCTagReaderService`, per storage slot, five reads issued concurrently and awaited together (`TVCTagReaderService.cs:55-61`): `TVC[n].Current_Temp`, `TVC[n].Preset_Temp`, `TVC[n].Temp_mode`, `TVC_Ctrl[n].ZZZZZZZZZZTVC_Contro0` and `TVC_IO[n].ZZZZZZZZZZTVC_I_O0`. The last two are the **alias reads** — bit-packed SINTs exposed under generated names — and the code then unpacks them by mask (`bit 0 = enabled`, and so on, `:70-76`). This is the concrete instance of `docs/PLC_FACTS.md` §7: reading `TVC_Ctrl[n].enabled` directly would return `ErrorBadParam`.

`BatchStepsPoller` → `BatchTagReaderService.TryReadAllStepArrays`, which reads `Batch_Current_Preset_Operation`, `Batch_Current_Preset_Amount`, `Batch_Current_Actual_Amount` and `Batch_Current_Preset_Mix` as **whole arrays** (not per-step tags), plus `current_step`, plus `Batch_Current_Stamps` on demand. `MaxSteps = 30` — raised from 18 so a recipe longer than 18 steps is not silently truncated — and array index `[0]` is skipped (`FirstStepArrayIndex = 1`). The service caps indexed fallbacks per read (`MaxIndexedFallbacksPerRead`) so a failing bulk read cannot degrade into a burst of dozens of single reads.

`BatchStartPoller` — per storage slot, `Tanks[n].{member}` through the `UdtTankReader` byte-offset path (`BatchStartPoller.cs:140`), plus `Formula_Info[f].Volume` (`:421`) and `System_Configuration[i]` (`:432`). It caches the two batch-start commissioning bits per PLC index (`tank_installed`, `Storage_tank OR Storage_tank2`) rather than re-reading them every tick. **It is read-only**; its own class doc says so, and the batch-start *write* path is a separate, untouched code path.

Why the stagger exists is stated at the call site: on controllers with a low concurrent-session cap "a simultaneous boot spike causes a transient session-exhaustion read-error burst that self-recovers; staggering smooths the ramp" (`PlcPollingCoordinator.cs:182-186`). The heartbeat and float pollers stay immediate because they are liveness and a single cheap read.

Why the full integer block stays at 5000 ms is also documented in place (`PlcPollingCoordinator.cs:144-148`): `PC_Read_Integer[0..479]` is the largest single read in the app and the live controller also serves the legacy HMI, so its cadence is *not* doubled. The freshness race that would otherwise create is handled on the **consumer** side — `MakeReadyTankViewModel.PcReadFreshnessWindow` is 12 s, which covers the 5000 ms cadence even one failed tick deep.

Every poller guards re-entrancy with an `Interlocked` busy flag (`_isPolling`, 0 = idle / 1 = polling), so a slow cycle **skips** the next tick rather than stacking cycles.

Roster snapshotting: a real constraint

`StorageTankGroupPoller`, `UseTankGroupPoller`, `TVCPoller` and `BatchStartPoller` all take their slot list **at construction** and never re-read it:

> "The roster is snapshotted at construction; a roster change (Setup save) requires an app restart to take effect on the pollers — this keeps cache sizing and slot→column mapping correct with no fragile mid-run resize." — `Ring/Services/PLC/StorageTankGroupPoller.cs`

Nothing in the app subscribes to `TankRosterState.Changed`. This is why the **write** side needs its own mid-session-change lock (chapter 04): after a Setup save, what the screen displays and what a write would target are resolved from two different rosters.

`Ring/Services/PLC/RosterPollPlan.cs` is the pure, PLC-free translation of a roster into per-slot read targets — the read-side twin of `RosterWriteIndex`.

Pause without blinding the operator

`PauseDatabaseWriters()` / `ResumeDatabaseWriters()` (`PlcPollingCoordinator.cs:55-90`) stop only `BatchStepsPoller` — the one poller in this stack that writes SQLite — and leave the read-only pollers running, so a database restore or factory reset does not black out the live screens. The pause is a **bounded deterministic drain**: `StopAndDrain(BatchStepsDrainTimeout)` waits up to 10 s for an in-flight poll (a whole-array read plus an insert) to finish, and logs a warning if it does not.

3.3 The timers that live *outside* the coordinator

This is the gotcha that costs people an afternoon: **the coordinator is not the only thing that talks to the controller.** `PlcPollingCoordinator.Stop()` stops seven pollers; it does not stop any of the following, and none of them appears in the stagger plan.

`ARCHITECTURE.md` §3 names **five** such timers — the alarm pump, `TrendingDataService`, `InventoryEventCaptureService`, `TagInspector` and `InventoryEditScreen`. That list is correct as far as it goes; the table below adds `MainWindow`'s other two timers (the UI PLC tick and the auto-reprobe, both of which do reach the controller) and keeps `ViscometerLiveRecorder` in view precisely because it is the one people wrongly count.

Owner	Cadence	Does it touch the PLC?
`MainWindow` PLC alarm background pump	2 s (`MainWindow.xaml.cs:170`)	**Yes** — drives `AlarmAlarmNumberPlcService`, which bulk-reads `Alarm_Alarm_Number[0..39]` itself
`MainWindow` `_plcUpdateTimer` (`DispatcherTimer`)	2 s (`MainWindow.xaml.cs:663`)	Indirectly — also pumps the same alarm method; a `CompareExchange` busy gate stops the two doubling up
`MainWindow` PLC auto-reprobe	30 s (`PlcReprobeIntervalMs`, `MainWindow.xaml.cs:62`)	**Yes** — reachability probe, then restarts monitoring
`TrendingDataService`	2 s (`PollIntervalMs = 2000`, `TrendingDataService.cs:32`)	**Yes** — constructs its own `PlcTagReader` per non-snapshot tag on every tick
`InventoryEventCaptureService`	1 s (`InventoryEventCaptureService.cs:133`)	**Mostly no** — normally reads only `PcReadIntegerCache`; on the `PC_Read_Integer[102].4` request bit it does real I/O (read stamps, write ack, poll for clear at a 100 ms `PollInterval`, `:23`)
`TagInspector` (Setup screen)	1 s (`TagInspector.xaml.cs:60-62`)	**Yes** — builds a live `PlcTagReader` per refresh
`InventoryEditScreen`	60 s (`InventoryPlcRefreshInterval`, `InventoryEditScreen.xaml.cs:25`)	**Yes** — inventory slice read

Owner	Cadence	Does it touch the PLC?
`ViscometerLiveRecorder`	5 s (`TickInterval`, `ViscometerLiveRecorder.cs:33`)	**No** — consumes `PcReadFloatSnapshot` / `PcReadIntegerSnapshot` and writes a SQLite row

The `ViscometerLiveRecorder` row is worth calling out because older documents list it as a PLC timer. It is not: its class doc and its tick body only read published snapshots. (`ARCHITECTURE.md` §3 already carries this retraction.)

Consolidating these under the coordinator is on the deferred-debt list (`ARCHITECTURE.md`), explicitly parked until after the cutover.

3.4 Snapshots, holders and caches

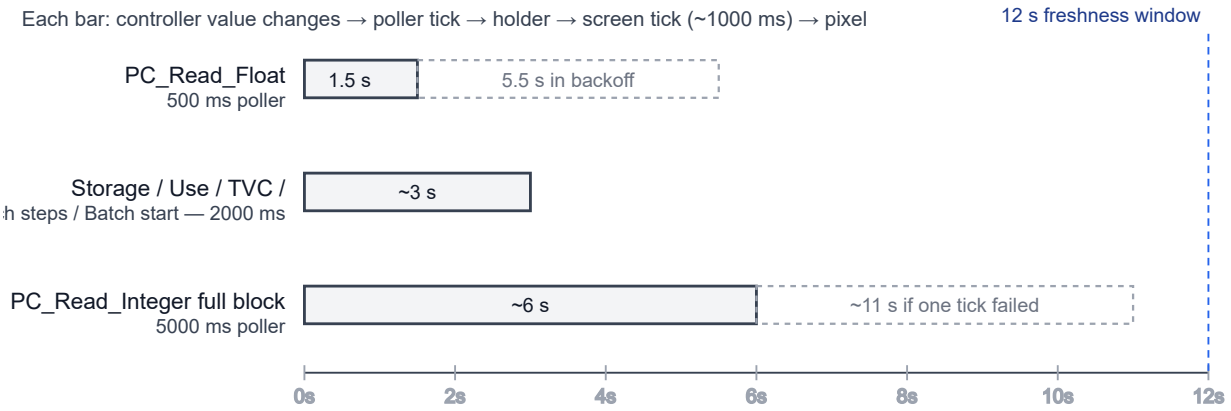
The pattern is uniform: **a poller builds an immutable snapshot and publishes it into a static holder; screens pull from the holder on their own timer.**

Holder	Type	File
`StorageTanksSnapshotHolder`	`StorageTanksSnapshot`	`StorageTanksSnapshot.cs:100`
`UseTanksSnapshotHolder`	`UseTanksSnapshot`	`UseTanksSnapshot.cs:101`
`TVCSnapshotHolder`	`TVCSnapshot`	`TVCSnapshot.cs:100`
`BatchStepsSnapshotHolder`	`BatchStepsSnapshot`	`BatchStepsSnapshot.cs:65`
`BatchStartSnapshotHolder`	`BatchStartSnapshot`	`BatchStartSnapshot.cs:195`
`PcReadIntegerCache`	`PcReadIntegerSnapshot`	`PcReadIntegerCache.cs`
`PcReadFloatSnapshot` (static block, not a holder type)	`float[35]`	`PcReadFloatSnapshot.cs`
`PcDisplayOutputsCache`	`PcDisplayOutputsSnapshot`	`PcDisplayOutputsCache.cs`
`MakeReadyTankPlcData`	static fields	`MakeReadyTankPlcData.cs`

There is no push, no event bus, and no binding to a poller. Concretely, the sequence for one tank temperature is:

```
controller value changes
   │  (up to one poller period)      StorageTankGroupPoller tick, 2000 ms
   ▼
StorageTanksSnapshot (immutable, stamped CapturedUtc)
   │  published into
   ▼
StorageTanksSnapshotHolder.Latest       ← static, lock-protected
   │  (up to one screen period)      screen DispatcherTimer, typically 1000 ms
   ▼
named control on the screen
```

Worst-case staleness is the number people get wrong, because it is poller cadence *plus* screen cadence, not just the poller's own number — and one poller's worst case, with a failed tick, is close to the freshness window built to cover it:



End-to-end latency is therefore poller cadence + screen cadence, worst case, plus the read itself:

Source	Poller	Screen	Worst-case age on screen
`PC_Read_Float` fast block	500 ms	~1000 ms	~1.5 s (5.5 s in backoff)
Storage / Use tank / TVC / Batch steps / Batch start	2000 ms	~1000 ms	~3 s
`PC_Read_Integer` full block	5000 ms	~1000 ms	~6 s — and ~11 s if one tick failed

That last row is why `MakeReadyTankViewModel.PcReadFreshnessWindow` is **12 seconds** and not something tighter: it has to cover a 5000 ms cadence one failed tick deep. If you tighten a freshness window, check it against the cadence *and* the backoff of the poller that feeds it.

The pull model is a deliberate trade. It costs latency and buys three things: a screen can never be blocked by a PLC read, a slow screen can never back-pressure a poller, and a snapshot is immutable so a screen can never observe a half-updated set of values.

Freshness gating: the part that keeps a frozen screen honest

A stale value that *looks* live is the failure mode these types exist to prevent, and the code goes to some length about it.

Snapshots carry `CapturedUtc`. Verified present on `StorageTanksSnapshot`, `UseTanksSnapshot` (`:44`), `TVCSnapshot` (`:44`), `BatchStepsSnapshot`, `PcReadIntegerSnapshot` (`:25`) and `PcDisplayOutputsSnapshot`.

Exactly one snapshot carries no timestamp: `BatchStartSnapshot`

- **`TVCSnapshot` carries `CapturedUtc`.** It is declared at `Ring/Services/PLC/TVCSnapshot.cs:44`, defaulted to `DateTime.UtcNow` at construction in both constructors (`:47`, `:59`), and its own doc comment says why it was added: "*Mirrors `StorageTanksSnapshot.CapturedUtc` — added so a consumer can tell a LIVE heat call from a frozen one after the TVC poller stops (the Live Process overview's heat leg).*" It also gained `HasAnyData` (`:62`) for the all-reads-failed case.
- **`BatchStartSnapshot` genuinely has none.** A search for `CapturedUtc` or any `DateTime` member in `Ring/Services/PLC/BatchStartSnapshot.cs` returns nothing.

The asymmetry is real and worth knowing: a Batch Start screen **cannot** freshness-gate what it displays, while every other snapshot in this table can.

Freshness math is monotonic-hardened. `PcReadIntegerSnapshot.IsFresh` (`:89-104`) and `PcReadFloatSnapshot.IsFastBlockFresh` (`PcReadFloatSnapshot.cs`) both age against the **greater** of a `Stopwatch` elapsed and the wall-clock delta. The reason is specific: a supervisor can set the LCP clock from Setup → Update Processor Time/Date, and a backward step would otherwise future-date the stamp and make a frozen block read as live. A forward step only errs toward "stale", which is the safe direction.

"Refreshed low words only" is a distinct state. `PcReadIntegerSnapshot.RefreshedLowWordsOnly` (`:32`) exists so a high-index consumer (the inventory notify bit at word 102, for instance) cannot be fooled by a snapshot whose `CapturedUtc` is recent but whose high words were not re-read. `IsFresh(maxAge, highestIndexNeeded)` (`:114-119`) is the overload that respects it. **Use that overload for anything above index 31.**

"All reads failed" must not advance the clock. `TVCSnapshot.HasAnyData` (`:62-...`) and `StorageTanksSnapshot.HasAnyData` return false when every slot field came back null — "exactly the frozen-but-looks-live case the staleness gates exist to expose".

Consumers refuse rather than display stale. `MakeReadyTankViewModel.PcReadFreshnessWindow` is 12 s; `ProcessHistorianService` drops a whole sample set classified `Dead`; `Ring/Services/Display/DataFreshnessClassifier.cs` is the shared classification used by the freshness converters in `Ring/Converters/`.

3.5 The index maps

Two small, load-bearing constant files.

`Ring/Services/PLC/PcReadIndexes.cs` — `PC_Read_Integer[0..479]` (`ElementCount = 480`, `:12`):

Constant	Index	Meaning
`HeartbeatIndex`	0	link heartbeat word

Constant	Index	Meaning
`BatchCurrentStepNumber`	1	
`BatchCurrentIngredientCode`	2	
`BatchIngredientProgressPercent`	3	
`BatchMixProgressPercent`	4	
`BatchMixCountdown`	5	displayed as m:ss
`BatchFormulaNameCode`	7	
`BatchStorageTankCode`	8	destination tank attribution
`SystemStatusWord` (alias `MixerSystemStatusWord`)	9	0 = idle, 1 = HOLD, 2 = running
`InventorySnapshotNotificationWord` / `…Bit`	102 / bit 4	PLC signals a new inventory snapshot

`SystemStatusWord` is decoded read-only by `Ring/Services/PLC/SystemStatusWord.cs`, which owns the Process-menu command policy (`ResumeCommandVisible` / `ResetCommandVisible`, consumed at `Ring/Views/UserControls/NavBar.xaml.cs:3430-3436`). Its L5K provenance is quoted verbatim in the constant's doc comment — HOLD wins over running.

`Ring/Services/PLC/PcWriteIndexes.cs` is deliberately tiny: it holds only `InventorySnapshotAckWord = 27` / `InventorySnapshotAckBit = 4`. The other `PC_Write_Integer` bit addresses live with the services that own them (`PcWriteInteger30BatchControlPlcService`, `ShiftControlPlcService`, `ProcessorTimeDatePlcService`) — see chapter 04.

`PcReadFloatSnapshot` carries the float-block map inline: `FastBlockLength = 35` (indices 0–34), `MixerWeightIndex = 30`, `MixerTemperatureIndex = 31`, `BoraxCausticWeightIndex = 32`, `ViscosityResultIndex = 33` (PLC-computed result), `ViscosityRawInputIndex = 34` (raw Cambridge analog input).

3.6 Heartbeat and connection state

`Ring/Services/PLC/PlcHeartbeatConnectionTracker.cs` is the single source of link truth. It watches `PC_Read_Integer[0]`, which the ladder increments.

All timing is monotonic

Every elapsed calculation runs off `Stopwatch` ticks through `MonotonicNowTicks` (`:60`, a test seam) and `MonoElapsed` (`:63-73`). UTC stamps exist **for display only** (`LastSuccessUtc`, the banner) and are explicitly documented as such at `:41-46`. A wall-clock step — NTP, or a supervisor setting the LCP clock mid-batch — therefore cannot fake liveness or suppress comms-loss detection.

The state machine

State	Entered when	Threshold constant
`Unknown`	before the first poll, and after `Reset()`	—
`Connecting`	a poll failed but the failure count is below the disconnect threshold	—
`Connected`	the heartbeat **value changed** on this poll, or has changed within the last 4 s	`StaleHeartbeatAfter` (`:97`)
`Stale`	heartbeat unchanged for more than 4 s	`StaleHeartbeatAfter`
`Starved`	heartbeat unchanged for more than 12 s while reads still succeed	`HeartbeatUnchangedStarvedAfter` (`:105`)
`Disconnected`	12 s of no successful read, **or** 3 consecutive read failures	`SilenceDisconnectAfter` (`:98`), `FailuresBeforeDisconnected = 3` (`:106`)

The classification is factored into two pure, unit-testable functions: `ClassifyHeldHeartbeat(TimeSpan)` (`:80-87`) and `ShouldDisconnectOnSilence(TimeSpan)` (`:94-95`).

`Starved` deserves its own state because it is the diagnosis operators need: the network is fine, Ring is fine, and the controller is not advancing the heartbeat. Pre-cutover the usual cause is `MainTask InhibitTask := Yes`; the post-cutover meaning is "PLC reachable but wedged" (`Ring/Services/PLC/PlcConnectionState.cs`).

Two questions, two answers

```
AllowHeavyPlcPolling()  // Connected OR Stale        (:127-135)
AllowPlcWrites()        // Connected only            (:143-150)
```

The asymmetry is deliberate and documented in place: `Stale` is a 4 s blip, and gating heavy polling off it "froze EVERY tag group on one heartbeat hiccup during a live batch". But `Stale` is *not proof the controller scan is advancing*, so writes wait. `AllowPlcWrites()` is layer 4 of the write gate stack — see chapter 04.

The escape hatch nobody should flip casually

`HeartbeatLinkLogicEnabled` is a `const bool = true` (`:33`). Setting it `false` makes `CurrentState` return `Connected` unconditionally, `AllowHeavyPlcPolling()` and **`AllowPlcWrites()` return `true` unconditionally**, and stops the heartbeat timer. It exists as a one-line debug toggle for a controller known to have `MainTask` inhibited — every guarded branch carries a `#pragma warning disable CS0162` so the code stays compiled. Flipping it disables a safety gate; treat it as a bench-only edit that must never reach a release.

`MarkDisconnected()` (`:172-183`) lets startup force the banner honest after a TCP probe fails, before any heartbeat poll has run.

3.7 Diagnostics around the link

`PlcCommunicationLogService` (`Ring/Services/PLC/PlcCommunicationLogService.cs`) — an in-memory ring buffer (`MaxBufferedEvents = 500`) plus optional per-day CSV disk log of TX/RX events, wired into `PlcTagReader.Read()` and `PlcTagWriter.WriteCore()` at their success/failure exit points. It backs the `DisplayCommunicationForm` setup screen. Disk writes are drained by a single batched writer task off the PLC thread, per-day CSVs older than the retention window are pruned on day-rollover, and every logging call is swallowed by the caller so a logging failure can never propagate into a PLC call.

Note that suppressed writes still produce a comm-log row, marked `"ReadOnly suppressed"` (`PlcTagWriter.cs:146`) — so the form shows the call that *would* have happened.

`PlcCommEventLogService` (`Ring/Services/PLC/PlcCommEventLogService.cs`) is a *different* service that a name-grep will confuse with the one above — the near-identical names (`PlcCommunicationLogService` vs. `PlcCommEventLogService`) are a known trap. Where `PlcCommunicationLogService` (RCS-128, immediately above) is an in-memory ring buffer plus optional per-day CSV of every individual TX/RX tag call feeding the live `DisplayCommunicationForm` tail, `PlcCommEventLogService` persists PLC connection-**lifecycle** events to a SQL table, `PlcCommunicationLog` (schema v32, `Repositories/PlcCommunicationLogRepository.cs`), so comms history survives a restart and is queryable/exportable — a much lower event rate, aimed at after-the-fact auditability rather than a live tail.

Its `PlcCommEventType` taxonomy: `Connected`, `Disconnected`, `HeartbeatStalled` (heartbeat tracker transitions — §3.6), `ReprobeStarted` / `ReprobeSucceeded` (the background auto-reprobe loop, §3.3), `SessionError` / `ReadFault` (a read or write failure, split by a **best-effort text heuristic** — `PlcCommEventLogService.ClassifyReadFailure` looks for the substring `"session"` in the exception message; libplctag exposes no distinct session-level error type, so this is documented as a heuristic, not a proven libplctag error-code distinction), `WriteAttempted` / `WriteBlocked` / `WriteSucceeded` / `WriteFailed` (the write outcome vocabulary), and `LogOverflow` (one summary row per overflow episode, never one row per drop).

The design is hot-path-safe by construction: `Record()` only enqueues onto a bounded `ConcurrentQueue` (`MaxQueuedEvents`, default 5000) and never touches the database or blocks, so it is safe to call from a PLC read/write call site. A single background writer task batches everything queued into one INSERT transaction per flush; if the backlog ever exceeds the cap (a stuck/slow DB), the **oldest** queued events are dropped to make room, and exactly one `LogOverflow` row summarizes the drop count. Retention defaults to **90 days** via `PlcSettings.CommLogRetentionDays` (`AppSettings.cs:551` — non-positive disables pruning), applied once at startup and then daily.

`PlcClockSkewMonitor` (`Ring/Services/PLC/PlcClockSkewMonitor.cs`) — a read-only sentinel. The PLC clock is set manually and the controller does not observe Dutch daylight saving, while Ring reconstructs PLC-sourced stamps as Windows-local. After each March/October transition the two are exactly an hour apart until someone re-syncs. The monitor compares the reconstructed PLC-local time of a freshly captured event against the PC clock at capture and, past `SignificantSkewMinutes = 2`, logs a warning and raises a throttled line on the PLC Health drawer. **It writes nothing to the PLC.** The skew math is pure and unit-tested; the logging side is fire-and-forget.

`PcWriteInteger30StrandedBitMonitor` — a read-only sentinel for a *stranded* command bit in `PC_Write_Integer[30]`. It belongs to the write story and is covered in chapter 04, but it is a reader.

`TagManifestService` (`Ring/Services/PLC/TagManifestService.cs`) loads `scripts/live-tags.json`, the manifest emitted by `scripts/Parse-L5kLiveTags.ps1`. Its Live/Dead verdicts are sourced from the **2024** plaintext export and are therefore *2024-sourced and non-authoritative* for anything running today (`docs/PLC_FACTS.md`). Treat a "dead" verdict as "no writer was found in the parsed 2024 export", never as "the tag is dead in the plant".

`TagSuggestionCatalog` feeds the Tag Inspector's autocomplete from the same manifest.

3.8 Demo mode on the read path

Demo mode is `DemoMode.Enabled` in configuration plus a restart. There is no CLI flag (`docs/CONFIG_AND_STARTUP.md` §6).

It substitutes at **two** levels, and knowing which is which saves confusion:

- **`DemoPlcTagReader`** (`Ring/Services/PLC/DemoPlcTagReader.cs`) is the seam inside `PlcTagReader`. The reader captures `DemoMode.Enabled` **once at construction** (`PlcTagReader.cs:24-27, 41-45`) — deliberately, "so a flip of the config mid-run does not cause one tag to half-substitute".
- **`DemoModeData`** (`Ring/Services/PLC/DemoModeData.cs`) is the substitute for everything that does *not* go through `PlcTagReader`. The heavy pollers build raw libplctag tags directly, so without this "demo mode shows the banner but every tank/temperature/batch tile stays blank". Each poller calls `DemoModeData.Enabled()` and publishes a synthetic snapshot instead of touching the network. Values drift on a per-channel sine (`Wave`) so the dashboard looks alive; the plausible bands mirror the live plant (nominal tank ~1125 gal, temps ~95–105 °F).

`DemoModeData.Enabled()` is null-safe and never throws on a startup path (`DemoModeData.cs`). **On the write path the demo check is re-evaluated live at the wire, not captured at construction** — that difference is deliberate and is explained in chapter 04.

3.9 What normal looks like

Every cadence and mechanism above is documented; this section states what *healthy* looks like numerically, so a deviation is recognisable as a deviation rather than discovered by accident. Where a figure is a compiled constant, it is cited directly. Where it depends on field or soak evidence that changes over time, this section links to the document that owns the number rather than freezing it here — per this book's own rule (README §How to use this book, item 2).

Signal	Expected steady state	Where to observe it	What a deviation usually means
Poll cycle duration vs. budget	Each poller has its own interval and read-timeout budget — e.g. `PlcSnapshotPoller` 500 ms interval / 2000 ms read timeout; the `PC_Read_Integer` full block 5000 ms interval / 3000 ms bulk timeout, 1500 ms indexed fallback timeout. Full table: §3.2	`PlcCommunicationLogService`'s comm log (per-tag TX/RX with duration); the PLC Health drawer	A cycle that regularly approaches its timeout, without yet failing, is the leading indicator of session pressure — it usually means something else is also holding EIP sessions against this controller (see the row below)
Heartbeat poll rate (Ring → controller)	Ring polls `PC_Read_Integer[0]` every **1500 ms** (`PcReadIntegerPoller.cs:56`; `PlcPollingCoordinator.cs:149`)	The `Connected`/`Stale`/`Starved` state in the PLC Health drawer; §3.6	`Stale` for more than a few cycles with reads still succeeding is `Starved` — the network and Ring are fine, the controller's own scan is not advancing the word. **Ring's poll cadence is code-verified; how fast the controller's own ladder increments the word is a ladder-side fact this book does not have a citation for** — do not infer a controller-side rate from Ring's poll interval

Signal	Expected steady state	Where to observe it	What a deviation usually means
Historian sampling and growth	One `ProcessHistorySample` row per installed tank per **60 s** (`ProcessHistorianService.cs`, `DefaultCadenceMs = 60_000`) — arithmetic on that cadence is ≈525,600 rows/tank/year, so a 6–12 tank plant accumulates on the order of 3.2–6.3M rows/year at the retention window's default length. Retention is `Database.ProcessHistorianRetentionDays` (Setup → Database Backup/Restore → Data retention; default **365** days, `0` = keep forever) — **not** the `ProcessHistorianService.DefaultRetentionDays = 30` constant this row cited before 2026-09-01; that constant is now only the fallback when DI supplies no configured value, and the shipped app always supplies one	chapter 05 §5.7	**No code-level source computes expected `.db` file growth in MB/day** — the row-count above is arithmetic on the sampling cadence, not a measured figure, and no soak document reviewed for this book reports a byte-size number either. `IX_ProcessHistory_Captured` and `IX_ProcessHistory_Tank_Captured` (`ProcessHistoryRepository.cs`) keep date-ranged report queries seeking rather than scanning at this row count, so query latency is not the growth concern; if disk headroom is the question, measure `RingwoodDatabase.db`'s size on your own install over a known number of days rather than trusting a number in prose
Memory / working set over a shift	**No code-level source** — `DiagnosticConsole` reads and displays `Process.WorkingSet64` but nothing in `Ring/` declares an expected range or an alert threshold	Setup → Diagnostic Console	The only soak observation in the repository as of this writing is a single ~6 h Debug-build run against the bundled simulator (working set fell 172 MB → 77 MB over the run, "healthy GC, no leak signature") — see `docs/production-readiness/RELEASE_PACKAGE_VERIFY_2026-07-30.md` §5. That document and `docs/production-readiness/REMAINING_WORK_TO_CUTOVER_2026-06-22.md` both record that the required 48–72 h Release-build soak against real hardware had not yet run. Treat the 6 h figure as evidence of no obvious leak, not as a production baseline

Signal	Expected steady state	Where to observe it	What a deviation usually means
Concurrent EIP session headroom	A CompactLogix caps at roughly **4–8 EtherNet/IP sessions per source IP** (`docs/PLC_FACTS.md`; corroborated in code at `UdtTankReader.cs:97`)	A burst of `ErrorTimeout` across multiple readers at once, with no single poller implicated	Ring's own seven pollers plus any field script opening a session from the same machine share this ceiling. The condition is self-healing — field evidence puts recovery at roughly 30 s after the extra sessions close (`docs/production-readiness/FIELD_SESSION_RUNBOOK_2026-06-29.md:62`)

Next: 04 — The Write Path and Safety Model.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring/Services/PLC/` — `PlcTagReader.cs`, `PlcPollingCoordinator.cs`, `PlcSnapshotPoller.cs`, `PcReadIntegerPoller.cs`, `BatchStepsPoller.cs`, `StorageTankGroupPoller.cs`, `UseTankGroupPoller.cs`, `TVCPoller.cs`, `BatchStartPoller.cs`, `StorageTankTagReaderService.cs`, `UseTankTagReaderService.cs`, `TVCTagReaderService.cs`, `BatchTagReaderService.cs`, `UdtTankLayout.cs`, `PcReadIndexes.cs`, `PcWriteIndexes.cs`, `PcReadIntegerCache.cs`, `PcReadIntegerSnapshot.cs`, `PcReadFloatSnapshot.cs`, `PcDisplayOutputsCache.cs`, `StorageTanksSnapshot.cs`, `UseTanksSnapshot.cs`, `TVCSnapshot.cs`, `BatchStepsSnapshot.cs`, `BatchStartSnapshot.cs`, `PlcHeartbeatConnectionTracker.cs`, `PlcConnectionState.cs`, `PlcCommunicationLogService.cs`, `PlcCommEventLogService.cs`, `PlcCommEventType.cs`, `PlcClockSkewMonitor.cs`, `TagManifestService.cs`, `DemoModeData.cs`, `DemoPlcTagReader.cs`, `TrendingDataService.cs`, `InventoryEventCaptureService.cs`, `ViscometerLiveRecorder.cs` · `Ring/Views/MainWindow.xaml.cs` · `Ring/Views/Setup/TagInspector.xaml.cs` · `Ring/Views/Setup/InventoryEditScreen.xaml.cs` · `Ring/Services/PLC/UdtTankReader.cs` · `Ring/Services/ProcessHistorianService.cs` · `Ring/Views/Setup/DiagnosticConsole.xaml.cs` · `Ring/Database/Repositories/PlcCommunicationLogRepository.cs`, `Ring/Database/Repositories/ProcessHistoryRepository.cs` (`ApplyRetentionDays` no-op on non-positive input; `IX_ProcessHistory_Captured`, `IX_ProcessHistory_Tank_Captured`) · `Ring/Database/DatabaseInitializer.cs` (schema v32) · `Ring/Infrastructure/Configuration/AppSettings.cs` (`CommLogRetentionDays`, :548-551; `Database.ProcessHistorianRetentionDays` default 365, :360) · `Ring/Infrastructure/Configuration/DataRetentionConfigWriter.cs` · `docs/PLC_FACTS.md` ·

`docs/production-readiness/RELEASE_PACKAGE_VERIFY_2026-07-30.md` · `docs/production-readiness/REMAINING_WORK_TO_CUTOVER_2026-06-22.md` · `docs/production-readiness/FIELD_SESSION_RUNBOOK_2026-06-29.md` · `ARCHITECTURE.md`

04 — The Write Path and Safety Model

Who this is for. Anyone who touches anything under `Ring/Services/PLC/`, and anyone reviewing a change that does. Read this before you write code, not after.

What you'll learn. Every gate between an operator gesture and a byte on the wire, with the code that implements it; where the gates sit in the order the code actually runs them (which is *not* uniform across rails); the commissioning holds, including the two that are hard-closed on purpose; the setpoint write queue and its deliberate non-contract; the evidence journal and exactly what it covers; the write-surface register that keeps the whole surface enumerable; and what a new writer owes before it is allowed to exist.

> **Ring writes to a live controller.** From the 2026-09-08 cutover those writes drive a running plant. A bad write does not corrupt a record, it moves a valve. Every rule below exists because breaking it would cost something real.

If you're reviewing a write-path PR, you need these four things

This chapter is long — it is the safety chapter and the reason the rest of the book exists — and most of it you read once. This box is for the second and third time, when you are re-opening it mid-review to check one gate. It restates nothing; every line links to the section that is the actual source of truth, so this box cannot drift out of sync with the chapter under it.

1. **The six gates, in the order `PlcTagWriter.WriteCore` actually runs them:** ReadOnly → Demo (live re-check) → Endpoint → Heartbeat → wire. Per-rail order differs from this — see the verified table in §4.7 before you assume a new rail matches it.
2. **Which flag gates this write family, and its default.** Eight per-feature `Enable*` flags plus `AllowLoopbackPlcWrites`; seven default `false`, one (`EnableProcessorTimeDateSync`) defaults `true`. Only `EnableHmiAgitatorCycle` ships in the JSON — everything else is the C# default. §4.2.
3. **Is this tank or tag one of the two hard-closed holds?** `MainTvcWriteAuthorization.IsAuthorized` and `TankTempModeWriteAuthorization.AuthorizedTankTempModeTanks` are a hard-coded `false` and an empty list, respectively — **do not populate either without a controls decision.** §4.3, "The two that are hard-closed on purpose".
4. **What the new writer owes, before it is allowed to exist:** a write-surface register row with the count floor raised in the same commit; a seam test pinning tag, value, count and order; bench evidence for a UDT member or a BOOL bit; an independent review. §4.12.

4.0 The idea, in plain language, before the detail

Ring can change three kinds of thing on the controller:

- **A setpoint** — a target value that stays where you put it (a temperature, a requested level, a recipe number). Level-valued: writing it twice is harmless.
- **A command bit** — a single bit the controller reacts to. Most are **momentary**: Ring sets it TRUE, waits, and sets it FALSE, and the controller acts on the transition. Leaving one TRUE is a fault, not a no-op.
- **A block of configuration** — an array such as a recipe bank, rewritten as a unit.

Each of those can hurt a plant in a different way, so Ring does not have one "write" function with one check. It has a **stack of independent gates**, each answering a different question, each failing closed, and each cheap enough to run on every write:

Gate	The question it answers
Read-only guard	*Is this station allowed to write at all, in this session?*
Per-feature `Enable*` flag	*Has this whole family of writes been commissioned?*
Commissioning hold	*Has **this** tag, on **this** tank, with **this** value, been signed off?*
Endpoint guard	*Does the configured address actually name a plant controller?*
Heartbeat gate	*Is the controller's program demonstrably still running?*
Demo re-check at the wire	*Is this session pretending, right now?*

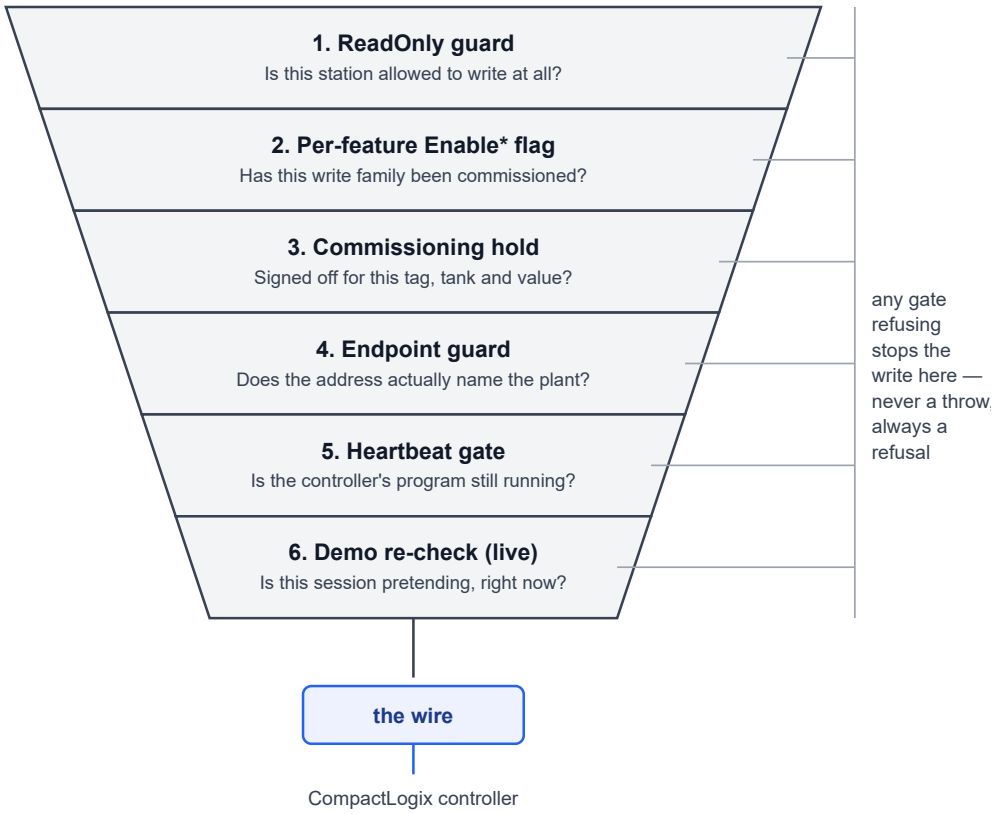
"Fails closed" means: when a gate cannot establish that a write is safe, it refuses. Not "assumes yes"; not "throws". Almost every ambiguous case in this chapter — an unreadable roster, an unread capability bit, an unbuilt DI container **where the answer is a permission** (`PlcWriteEndpointGuard.LoopbackWritesAuthorized`, §4.4), a `null` batch-step read — resolves to **refuse**. The one gate where an unbuilt container resolves the other way is the demo re-check (`DemoModeData.Enabled()` returns `false` on any failure, `DemoModeData.cs:29-39`) — see the note in §4.6 for why that is still the safe direction.

The rest of this chapter is those gates in the code that implements them.

4.0.1 The shape of the argument

The stack is **six layers**, cheapest-and-most-global first:

1. `PlcWriteGuard.IsReadOnly` — global kill switch
2. per-feature `Enable*` master flags
3. bespoke commissioning holds
4. `PlcWriteEndpointGuard` — the write whose *target* must be the plant controller
5. `PlcHeartbeatConnectionTracker.AllowPlcWrites()`
6. the live demo re-check at the wire



This funnel is the conceptual six layers, in the order this section names them — it is not any single rail's actual order. §4.7's table gives that, per rail, and the note below explains why `PlcTagWriter.WriteCore` itself runs Endpoint before Heartbeat.

That is now the count in `README.md` and `ARCHITECTURE.md` too — both once said five, omitting the endpoint guard, and both have been corrected. `PlcWriteEndpointGuard` (`Ring/Services/PLC/PlcWriteEndpointGuard.cs`) refuses a write whose *target* cannot be the plant controller. It is wired into every write rail and has its own row in the write-surface register (`Ring.Tests/WriteSurfaceRegisterTests.cs:130`). The register's own count-floor comment records it as the one genuinely *new* write-surface file added in the 2026-08-26 amendment (`WriteSurfaceRegisterTests.cs:500-502`). It is not optional and it is not a subset of any other layer.

The order is per-rail, not global. The six-layer list above is conceptual, useful for talking about the stack — it is not the order any single rail actually runs them in. Each writer runs them in the order that made sense for its own diagnostics, and they differ. Inside the shared funnel `PlcTagWriter.WriteCore` (`Ring/Services/PLC/PlcTagWriter.cs:135-220`) the actual order is:

```
ReadOnly → Demo (live re-check) → Endpoint → Heartbeat → wire
```

Note that this contradicts `PlcWriteEndpointGuard`'s own class doc (`PlcWriteEndpointGuard.cs:41-44`), which describes itself as sitting *after* the heartbeat gate and after the demo re-check. In `PlcTagWriter` it sits **before** the heartbeat gate, and the reason is written at the call site: on an unconfigured station the link is Disconnected as well, and "PLC link unavailable" would send the operator to the network instead of to

`appsettings.json` (`PlcTagWriter.cs:164-171`). The doc comment is stale about ordering; the *set* of gates it names is right. Ordering among gates that all fail closed is a diagnostics decision, not a safety one — but do not quote the doc comment as the order.

§4.7 gives the verified per-rail table.

4.1 Layer 1 — `PlcWriteGuard.IsReadOnly`

`Ring/Services/PLC/PlcWriteGuard.cs`. A static, process-wide gate.

```
private static volatile bool _readOnly = false;      // :25
private static volatile bool _forcedReadOnly = false; // :30

public static void Configure(bool readOnly)          // :37-40
    => _readOnly = readOnly || _forcedReadOnly;

public static void ForceReadOnlyForSession()         // :49-53
{ _forcedReadOnly = true; _readOnly = true; }

public static bool ResolveConfiguredReadOnly(AppSettings settings)  // :75-76
    => settings?.PlcSettings?.ReadOnlyMode ?? true;
```

Four properties, each load-bearing:

It is configured exactly once, before anything can write. `App.OnStartup` calls `Configure(ResolveConfiguredReadOnly(settings))` inside `RunStartupConfigurationValidation()` (`Ring/Views/App.xaml.cs:1115-1116`), which is phase 9 — before `DatabaseInitializer`, before any background service, before `MainWindow`.

The resolver fails safe. A null `AppSettings` *or* a null `PlcSettings` both resolve to `true`. The fallback was extracted into a static specifically so the safety default is unit-testable (`PlcWriteGuard.cs:67-74`).

`ForceReadOnlyForSession` is a one-way latch. `Configure` ORs `_forcedReadOnly` in, so anything can tighten the gate and nothing can loosen it short of a restart. `StartupDegradedState.ApplyDatabaseInitOutcome(false, …)` pulls it when database initialization fails (`Ring/Services/StartupDegradedState.cs`) — see chapter 02 §2.4. `ResetForcedReadOnlyForTests()` (`:62-65`) exists solely so a test that exercised the forced path does not leak read-only into the rest of a `-parallel none` run; production never calls it.

The field defaults to `false`, not `true`. That is intentional: headless tests never call `Configure` and must keep writing to their fakes (`PlcWriteGuard.cs:23-24`). In the app, `Configure` always runs. The only other caller under `Ring/` is the setup wizard's read-only step (`Ring/Views/Wizard/Steps/ReadOnlySafetyStep.xaml.cs`, register row at `WriteSurfaceRegisterTests.cs:198`) and it can only turn the gate **on**.

What a blocked write *returns* is not uniform

This matters more than it looks, because a caller that reads the return value as "it landed" will lie to an operator.

Path	Behaviour when read-only
`PlcTagWriter.WriteCore`	logs the suppression, writes a comm-log row marked `"ReadOnly suppressed"`, and **returns `true`** (`PlcTagWriter.cs:143-148`)
`PcWriteInteger30BatchControlPlcService.PulseBitSerialized`	returns `BatchControlWriteResult.BlockedReadOnly` (`:190`)
`ShiftControlPlcService` pulse	logs and **returns `true`** (`ShiftControlPlcService.cs:292`)
`ProcessorTimeDatePlcService.WriteProcessorDateTime`	returns `ProcessorClockVerifyResult.Held` (`:359`)
`BatchFormulaPresetPlcWriter`	returns `FormulaBankWriteOutcome.SuppressedReadOnly` (`FormulaBankWriteInterlock.cs:22`)
`TVCControlPlcService`, `UseTankControlPlcWriter`, `BatchStartTankPlcWriter`, `InventoryAmountPlcWriter`	return **`false`** — explicitly so a suppressed write is never reported as a landed one

The "return true" cases are the *legacy fake-success idiom*: they exist so the operator flow (batch start, formula select, ack, silence) continues normally in read-only and demo, and the operator is told separately that nothing was sent. `FormulaBankWriteOutcome` documents the reasoning at length (`Ring/Services/PLC/FormulaBankWriteInterlock.cs:16-93`), including why a *held capability* is treated exactly like read-only rather than as a failure — `SuppressedFeatureHeld` (`:83`) was added because a `Blocked` outcome aborted the whole commit and "formula editing for formulas 1-6 stopped working on cutover day".

How to read a return value on this path, in one rule: a `true` from a write API in Ring means "*the caller's flow may continue*", **not** "*the controller accepted a value*". Only `PlcWriteStatus.WrittenAndVerified` claims a read-back; `WrittenUnverified` claims only that the writer reported success. If you are adding a caller that will *tell an operator something landed*, you must consult the typed outcome, not the boolean — and if your rail only has a boolean, that is a reason to give it a typed outcome rather than a reason to trust the bool.

The convergence of these return shapes onto one `PlcWriteResult` type is designed but deliberately deferred past the cutover (`docs/production-readiness/PLC_WRITE_RESULT_MIGRATION_2026-08-02.md`; `Ring/Services/PLC/PlcWriteResult.cs` already exists and is used by the evidence journal).

4.2 Layer 2 — the per-feature `Enable*` master flags

Eight keys on `PlcSettings` (`Ring/Infrastructure/Configuration/AppSettings.cs`), each gating one family of writes. **Seven default `false`; one defaults `true`.**

Key	Default	Line	Gates
`EnableInventorySnapshotPlcAcknowledge`	`false`	`:403`	the `PC_Write_Integer[27].4` ack pulse
`EnableInventoryAmountWrite`	`false`	`:412`	`Inventory_Amount[0..13]` bulk REAL write
`EnableProcessorTimeDateSync`	**`true`**	`:432`	controller wall-clock write + `[30].10` commit pulse
`EnableShiftControlPlcWrites`	`false`	`:442`	shift pulses on `PC_Write_Integer[44/45/48/49]`
`EnableUseTankFormulaWrite`	`false`	`:477`	`Tanks[slot].Formula_number` from the Use Tank windows
`EnableHmiAgitatorCycle`	`false`	`:505`	the HMI-side `Tank_IO[N].O_Agitat` auto-cycle
`EnableFormulaBankWrite`	`false`	`:529`	destructive `Formula_0N_Preset_*` bank rewrite
`EnableTankNamePlcNameSync`	`false`	`:542`	`HMI_Tank_Name[]` / `HMI_Group_Name[]` STRING sync

Two more `Enable*` keys on `PlcSettings` are **not** write gates and are worth distinguishing so nobody "hardens" them by mistake: `EnableBatchStartPolling` (`:380`, default `true`) and `EnableInventorySnapshotCapture` (`:386`, default `true`) gate *reads*, and `EnableLiveViscometerFeed` (`:453`, default `false`) gates a recorder that writes SQLite and never the PLC.

The C# defaults are the contract. Of the eight, only `EnableHmiAgitatorCycle` appears in the shipped `Ring/Config/appsettings.json` (verified: that file's `PlcSettings` node contains `ReadOnlyMode`, the five timeout/retry values (`ConnectionTimeout`, `ReadTimeout`, `WriteTimeout`, `RetryAttempts`, `RetryDelay`), `DefaultIpAddress`, `DefaultPath`, `DefaultPort`, `Protocol`, `EnableBatchStartPolling`, `EnableInventorySnapshotCapture`, `EnableHmiAgitatorCycle` — and nothing else). The other seven bind to the C# default, which is exactly why `WriteSurfaceRegisterTests` pins them:

```
[Fact]
public void Hardware_signoff_write_capabilities_remain_fail_closed_in_the_shipped_settings()
{
    var plc = new PlcSettings();
    Assert.False(plc.EnableInventoryAmountWrite);
    Assert.False(plc.EnableInventorySnapshotPlcAcknowledge);
    Assert.False(plc.EnableShiftControlPlcWrites);
    Assert.False(plc.EnableUseTankFormulaWrite);
    Assert.False(plc.EnableFormulaBankWrite);
```

```
    Assert.False(plc.EnableTankNamePlcNameSync);
    Assert.False(plc.EnableHmiAgitatorCycle);
    Assert.False(plc.AllowLoopbackPlcWrites);
    ...
}
```

(`Ring.Tests/WriteSurfaceRegisterTests.cs:1053-1126`.) That test pins **seven** of the eight flags — **`EnableProcessorTimeDateSync` is not covered**, because it ships `true`. It also pins `AllowLoopbackPlcWrites` false, pins `AlarmSettings.EnableCustomSilenceLatchTag` false and `SilenceLatchTagName == "PC_Write_Integer[30].3"`, and re-checks all of that against **both** shipped JSON files (`Ring/Config/appsettings.json` and `Ring/appsettings.production.template.json`) so a JSON override cannot flip the runtime value with the C# assertions still green. The test states its own scope limit explicitly: the deployed `Config\appsettings.local.json` overlay is a field file that does not live in this repository and is checked by `scripts/Preflight-FieldKit.ps1` and the cutover config read-back instead.

`AllowLoopbackPlcWrites` — the ninth flag

`PlcSettings.AllowLoopbackPlcWrites` (`AppSettings.cs:351`, default `false`) is not a feature gate; it is the deliberate opt-in that re-permits a loopback/unconfigured write target for bench work against a local simulator. Nothing in the shipped configuration sets it. See §4.4.

Each flag's rationale is in its doc comment — read it before flipping one

These are not stylistic comments. `EnableHmiAgitatorCycle` (`AppSettings.cs:479-505`) records that the **ladder owns** `Tank_IO[N].O_Agitat` — `ST_ROUTINE Agitator` drives it on a 25 ms task that round-robins twelve tanks — so the HMI cycler would be a second, unsynchronised cycler contending for a motor the controller already sequences. Its comment lists three preconditions before anyone turns it on, including "confirm `MainTask InhibitTask = No` on the live controller" and "add a per-tank capability gate — live `enabl_agt` is true for tanks 1, 2, 8 and 9 only".

`EnableFormulaBankWrite` (`:507-529`) records that the rewrite replaces every one of the 31 cells of a **live** recipe bank, single-pass, and that the `Preset_Amount` engineering unit per operation is still provisional.

4.3 Layer 3 — bespoke commissioning holds

These are per-subsystem authorizations that answer a question the global flags cannot: *is this specific write, to this specific tank, with this specific value, something a controls engineer has signed for?*

Hold	File	State in the code today	What it gates
`MainTvcWriteAuthorization`	`MainTvcWriteAuthorization.cs:15`	**`IsAuthorized => false`, hard-coded**	the shared main-TVC command surface (`TVC_Ctrl[N].enabled`, `TVC[N].Temp_mode`)

Hold	File	State in the code today	What it gates
`TankTempModeWriteAuthorization`	`TankTempModeWriteAuthorization.cs:62`	authorized-tank array is **empty**	`Tanks[N].Temp_mode` on every tank; also restricts the value vocabulary
`TvcCoolingWriteAuthorization`	`TvcCoolingWriteAuthorization.cs:54`	authorized tanks `{ 1, 2, 4 }` — **not** empty	`TVC[N].Temp_mode = 2` (cooling)
`TankInstalledWriteGate`	`TankInstalledWriteGate.cs`	data-driven from controller bits; **Unknown refuses**	every operator write landing on a `Tanks[N]` member
`RosterWriteIndex` locks	`RosterWriteIndex.cs`	sticky, fail-closed	every storage-tank write that must resolve a tank index
`FormulaBankWriteInterlock`	`FormulaBankWriteInterlock.cs`	decision function	the destructive formula-bank rewrite

The two that are hard-closed on purpose

`MainTvcWriteAuthorization.IsAuthorized` is the literal `false`. Not a config read, not a list — a compiled constant (`Ring/Services/PLC/MainTvcWriteAuthorization.cs:15`). The rationale is in the class doc: the controller maps a tank through `Tanks_c[N].TVC_tank`, while the retired writer addressed `TVC[N]` directly and also changed a ladder-computed `TVC_Ctrl` bit; that plant contract is not verified.

The refusal is enforced twice, and the second one is stronger: `TVCControlPlcService.SetMainTVCControlModeAsync` consults the flag (`TVCControlPlcService.cs:278-281`), but `SetMainTVCPresetAsync` **refuses unconditionally, without even consulting it** (`TVCControlPlcService.cs:266`).

`TankTempModeWriteAuthorization.AuthorizedTankTempModeTanks` is `{ }`. Empty by design, and the doc comment says so in capitals: *"EMPTY BY DESIGN — this is the inert half of a two-phase fix, not an oversight… Do NOT populate this list to 'make the screen work'"* (`Ring/Services/PLC/TankTempModeWriteAuthorization.cs:55-62`). Phase 1 (the retarget away from the wrong shared-unit tags) has shipped; phase 2 (permission to write the new per-tank tag) is one edit here after controls sign off row F058-T1.

That class also carries a **value vocabulary**: `TempModeOff = 0`, `TempModeHeat = 1`, `TempModeCool = 2`, and `IsWritableValue` returns false for 3 (`:97-98`). The ladder reads four states; value 3 (heat **and** cool) is a legal ladder value Ring may read and display but must never write. A controller already sitting on 3 is displayed as "no option selected" with an explanation rather than coerced into an item that would write 1 or 2 over it.

> **`CONTRIBUTING.md` §4 and `CLAUDE.md` both say the same thing, and this book repeats it: do not populate either of these without a controls decision.** The status of every held item is `docs/production-readiness/CONTROLS_SIGNOFF_RECORD.md`; the design rationale is in the 2026-07-09 and 2026-07-13 held-item packages. `WriteSurfaceRegisterTests` asserts both states (`WriteSurfaceRegisterTests.cs:1074-1075`), so populating one turns the suite red.

The one that is *narrowly* open

`TvcCoolingWriteAuthorization` authorizes tanks `{1, 2, 4}` and excludes tank 3 (the low-mid screen). Its doc comment is the clearest example in the codebase of *why* a hold is where it is: the F058 UI convergence gave tank 3 the ability to send `TVC[3].Temp_mode = 2` for the first time — a **new setpoint value on a live temperature-control path**, not a suppression and not a tightening. The authorized set is "exactly the set of tanks that could ALREADY command cooling before the convergence", so the hold **only narrows**: nothing that used to be writable stopped being writable, and the one thing the wave added that nobody signed for is refused (`TvcCoolingWriteAuthorization.cs:33-42`).

`IsModeAuthorized(tank, cooling)` is `!cooling || IsCoolingAuthorized(tank)` (`:70-71`) — heating and off are unaffected on every tank. This can only ever refuse the cooling selection.

The commissioning gate driven by the controller itself

`TankInstalledWriteGate` (`Ring/Services/PLC/TankInstalledWriteGate.cs`) answers "does the controller's own configuration say this tank (and this additive) exists?" from `Tanks_c[N]` bits, read through the A-B hidden-parent aliases:

Bit	Alias / bit	L5K line
`tank_installed`	`ZZZZZZZZZZTank_c17` bit 0	`:454`
`Liquid_1_installed`	`ZZZZZZZZZZTank_c44` bit 5	`:486`
`Liquid_2_installed`	`ZZZZZZZZZZTank_c44` bit 6	`:487`
`Liquid_3_installed`	`ZZZZZZZZZZTank_c44` bit 7 (the SINT **sign** bit — masks applied to a widened `int`)	`:488`

Two policies, deliberately different and pinned on both sides:

- **The UI fails OPEN on `Unknown`** — a link blip must not lock an operator out of a working tank.
- **The wire fails CLOSED on `Unknown`** — an unread capability is not permission to command a live starch tank.

Its known limit is stated in the class doc: state is keyed by PLC tank index and nothing evicts it when Setup re-pins a window's roster entry, so a capability can be up to one 60 s re-validation old. The index itself is still resolved per gesture, so the gate never answers for a tank the screen is not aimed at.

The roster locks — three failure modes, three answers

`RosterWriteIndex` (`Ring/Services/PLC/RosterWriteIndex.cs`) is the single owner of "which physical tank does this UI slot **write** to?" — the write-side twin of `RosterPollPlan`.

It exists because of a specific, real defect class: the Wave 2 cardinality lift made every per-tank *read* take its PLC index from the roster slot, while the batch-start *writer* still computed `uiSlot + 1`. Setting storage slot 1's `PlcIndex` to 5 therefore made the Batch Start screen **read `Tanks[5]` and write `Tanks[1]`** — a formula

and level committed to the wrong physical tank while the screen displayed another tank's data (`RosterWriteIndex.cs:11-21`).

Three distinct failures, and the `NoWrite = -1` sentinel alone covers only the first:

1. **Unmapped / disabled / out-of-range slot** → `StoragePlcIndex` returns `NoWrite` (`:174-190`). Also returned for a `LegacyWindowKey` that **no slot claims** or that **more than one slot claims** (`:245-253`) — "with two candidates there is no honest answer to *which physical tank is this screen showing?*".
2. **Mid-session roster change** → `StorageWritesLockedByRosterChange` (`:76-91`), which reads `TankRosterState.RestartRequired`. Needed because the pollers pin their roster at construction while `CurrentStorageSlots()` resolves live on every write, so after a Setup save the two are different physical tanks. Sticky; clears only on restart.
3. **Roster file present but unusable** → `StorageWritesLockedByRosterLoadFailure` (`:124-139`), reading `TankRosterState.LoadFailed`. This one is subtle and the doc comment explains why the other two cannot see it: the substituted fallback roster is *valid-looking*. `StoragePlcIndex` returns an ordinary 1..4, so the `NoWrite` branch never trips; and `…ByRosterChange` is false because no `Replace` ever ran — the divergence came from disk, not from an operator edit.

Every `catch` in this file returns the **fail-closed** answer. Refusing can never arm the wrong physical tank; guessing can.

`StorageWriteIndexForLegacyWindow(key)` (`:269-274`) is the whole write-side decision in one call — both sticky locks first, then the lookup — and is what the four TVC control screens use so their thirteen setpoint calls *and* the agitator motor command all take the same resolved index.

`StorageWriteRefusal(key, out resourceKey, out englishFallback)` (`:298-314`) resolves the localized key **and** its English fallback together, from one read of both locks. The comment records why: the screens used to render one generic key and pass the cause-specific sentence as its English fallback, so the cause was named only on a station whose dictionary had lost the key — i.e. never.

`ResolveStorageWriteTank` (`:208-212`) exists for **display text only** and returns `uiSlot + 1` on failure; the comment is explicit that this "deliberately produces a tank number that may name no configured tank" and that the write itself is still refused by `BatchStartTankPlcWriter`, which resolves the index for real.

The active-batch interlock

`FormulaBankWriteInterlock.Decide(...)` (`Ring/Services/PLC/FormulaBankWriteInterlock.cs:153-179`) is a pure function guarding the destructive `Formula_XX_Preset_*` rewrite. Three rules:

- **A `null` `current_step` read is `BlockUnknown`, never idle** (`:162-163`). Unreadable ⇒ blocked.
- **An active step is `BlockActiveBatch`** via `BatchStepsPoller.IsActiveStep`, using the configurable `PlcSettings.BatchIdleStepValue` sentinel.
- **A *fresh* cached snapshot that still says "active" wins over a single idle read**, guarding a one-tick flap to the idle sentinel mid-batch — but only inside `SnapshotVetoWindow = 10 s` (`:143`), so a stale "active" snapshot can never wedge the recipe editor shut forever.

The primary signal is a **fresh direct read**, not the cached snapshot, and the comment explains why: `BatchStepsPoller` deliberately does not republish on steady idle, so snapshot age is not a usable liveness signal.

It is **bank-agnostic on purpose** — it refuses while *any* batch runs, not only when the running batch consumes the bank being edited, because Ring cannot prove which bank a running batch is bound to. "Prefer refusing too often over refusing too rarely."

4.4 The endpoint gate

`Ring/Services/PLC/PlcWriteEndpointGuard.cs` closes a hole created by a deliberate decision elsewhere.

`PlcConnectionConfig.GetPlcIp()` does **not** throw when no IP is configured; it warns once and returns `127.0.0.1`, so an unset IP cannot trap the operator before the startup wizard runs (`Ring/Infrastructure/Configuration/PlcConnectionConfig.cs:93-121`). That is right for reads: loopback simply never answers and Ring shows its normal disconnected banner. It is **wrong for writes** — anything listening on the loopback interface (`tools/ab_server`, a replay harness) answers them, so an unconfigured station would "arm a batch", "silence an alarm" or "set the controller clock" against a simulator while every gate and the write journal report success. And the shipped `Ring/Config/appsettings.json` ships `DefaultIpAddress: ""`, so **the fall-through is the shipped state, not a corner case**.

The guard classifies a gateway string with no network contact at all — no DNS lookup, because a lookup is itself network contact and a name that does not resolve fails the connection rather than reaching the wrong plant:

Class	Meaning	Written to?
`Routable`	a non-loopback IP literal, or a syntactically valid RFC-1123 host name	**yes**
`Loopback`	`127.0.0.0/8`, `::1`, `localhost`, `*.localhost`, `localhost.localdomain`	only with the opt-in
`Unconfigured`	empty/whitespace, or the unspecified address `0.0.0.0` / `::`	only with the opt-in
`Unresolvable`	neither a valid IP literal nor a valid host name	**never** — no flag makes it a valid target

(`PlcWriteEndpointGuard.Classify`, `:70-113`; `AllowsWriteTo`, `:128-152`.)

`Classify` tolerates an explicit `":port"` suffix, but only strips it when the remainder is a plain number, so an IPv6 literal is left intact (`:80-86`).

The opt-in is `PlcSettings.AllowLoopbackPlcWrites`, read through `LoopbackWritesAuthorized()` (`:190-201`) — and an **unbuilt DI container means NOT authorized**, which is the fail-closed direction.

`RefusalReason` (`:155-177`) produces operator- and log-facing text that names the target, the class, *and the remedy*, including the sentence "Anything answering there is a local simulator, so the write would report success against synthetic data."

This does **not** replace `ConfigurationValidator`'s production loopback abort: that aborts *boot* on a real install; this refuses *individual writes* on a station that booted anyway.

4.5 The heartbeat gate

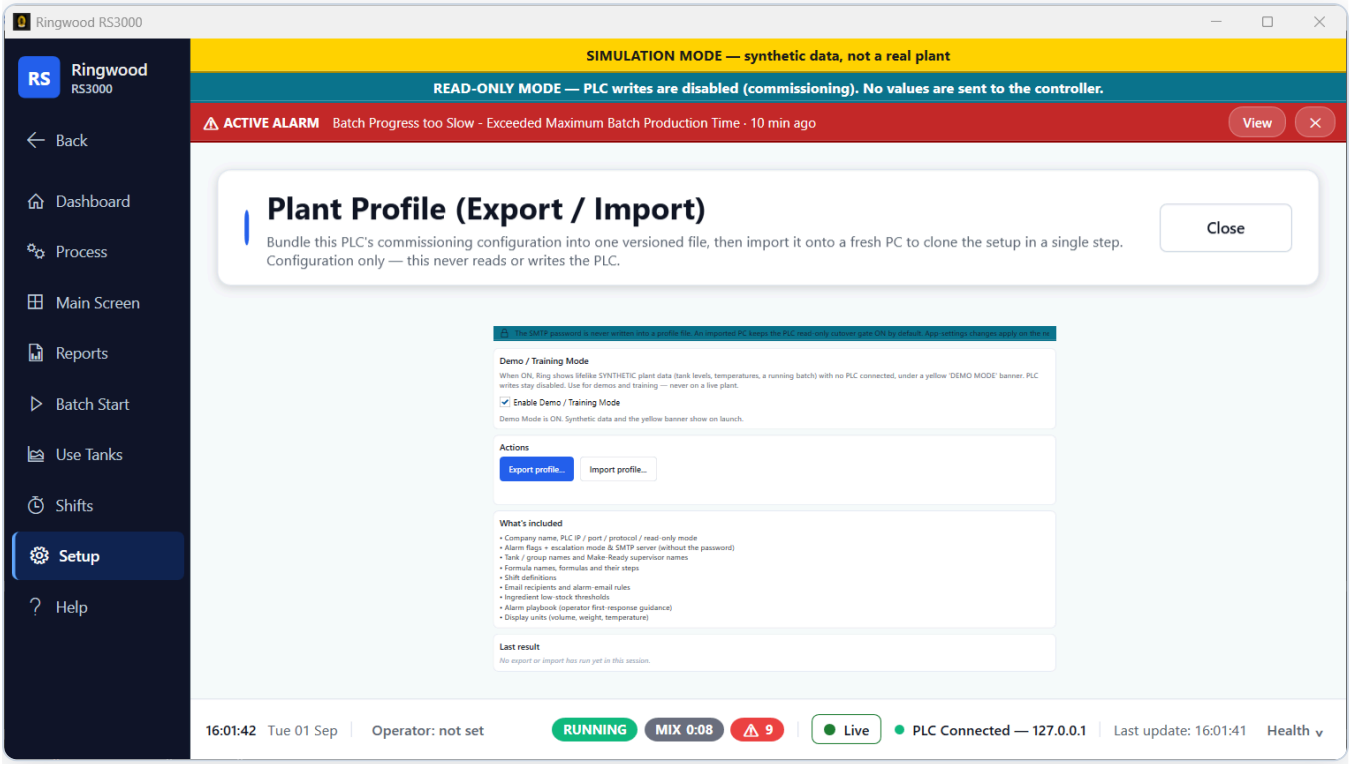
`PlcHeartbeatConnectionTracker.AllowPlcWrites()` returns true **only** in the `Connected` state (`Ring/Services/PLC/PlcHeartbeatConnectionTracker.cs:143-150`) — strictly stricter than `AllowHeavyPlcPolling()`, which also accepts `Stale`. The reasoning is in the doc comment: `Stale` (4–12 s without heartbeat motion) is fine for read pollers but "is not proof the controller scan is advancing, so writes wait."

The state machine, its thresholds, and the fact that all elapsed math is monotonic are covered in chapter 03 §3.6.

Two things to keep in mind here:

- Setting the `HeartbeatLinkLogicEnabled` const to `false` (`PlcHeartbeatConnectionTracker.cs:33`) makes `AllowPlcWrites()` return `true` unconditionally. It is a documented bench toggle for a controller with `MainTask` inhibited. It disables a safety gate; it must never reach a release.
- Most write rails check the heartbeat **twice** — once at the service (an outer gate, so a pulse cannot *start* on a degraded link) and once inside `PlcTagWriter` (the inner backstop). `InventoryEventCaptureService` is the documented exception: it has no outer gate (`PlcTagWriter.cs:126-128`).

4.6 The demo re-check at the wire, and the F013 de-assert exemption



The Plant Profile screen — the one surface that can flip Demo Mode mid-session, which is the reason demo is re-checked live at every write rather than captured once at construction.

Demo is re-checked live, on every write

```
if (DemoModeData.Enabled())            // PlcTagWriter.cs:156
{
    var demoWriter = _demoWriter ??= new DemoPlcTagWriter(_tagName, _type, _logger);
    ...
}
```

The check deliberately ignores the flag captured when the writer was constructed, and lazily creates the no-op writer if demo was off at construction time (`PlcTagWriter.cs:57-58, 150-162`). The reason: the Plant Profile screen can flip demo mid-session, and "a writer built before the flip must still suppress". Note the asymmetry with the **read** path, where `PlcTagReader` captures demo once at construction so a mid-run flip cannot make one tag half-substitute (`PlcTagReader.cs:24-27`). Both choices are right for their side: on the read path a half-substituted screen is the hazard; on the write path a live writer surviving a flip to Demo is.

`DemoPlcTagWriter` (`Ring/Services/PLC/DemoPlcTagWriter.cs`) is deliberately trivial — it logs the write in the same style as `PlcTagWriter` and returns `true`, "so operator flow (batch start, formula select, ack, silence) continues normally during training without touching a real PLC". Note that the comm log is still written from `WriteCore` (`PlcTagWriter.cs:160`), so `DisplayCommunicationForm` shows demo traffic as activity. That is the same "fake success" idiom as the read-only path, for the same reason, and it is why `FormulaBankWriteOutcome` gives demo its own value (`SuppressedDemo`) rather than reusing a generic success.

The demo check is also present at several *service* rails **above** `PlcTagWriter` (see the table in §4.7). Those are not redundant: they let a service skip work it would otherwise do — a read-back, a verification loop, a database commit — rather than performing it against nothing.

A one-gate exception to "unbuilt container resolves to refuse." `DemoModeData.Enabled()` catches any `ServiceLocator` failure and returns `false` — "not in demo" — which lets a write continue toward the wire rather than refusing it (`DemoModeData.cs:29-39`). That is the opposite resolution from `PlcWriteEndpointGuard.LoopbackWritesAuthorized()` (§4.4), which treats the same failure as NOT authorized. Both are correct for what they gate: demo is a *suppression*, not a *permission*, so failing to detect it can never arm a write that was not already armed by every other gate in the stack. Do not generalise "unbuilt container fails closed" to this one case.

F013 — the de-assert exemption

`PlcTagWriter.WriteClear` (`:133`) is identical to `Write` **except that a degraded heartbeat does not refuse it**: it logs a warning and attempts the wire anyway (`PlcTagWriter.cs:188-198`).

The ReadOnly gate, the live demo re-check and the endpoint gate remain **unconditional** for a de-assert. F013 relaxes the heartbeat gate and nothing else. The code says so at both the doc comment (`:122-125`) and the three call-site comments.

The justification is a rung-level census, quoted verbatim in the source: every instruction touching `PC_Write_Integer` in `scripts/RS_3000_2024_APR_22.L5K` is `{XIC ×20, XIO ×5, MOV ×6}`, and all six MOVs use the array as a **source** (L5K:12852). There is not one OTU/OTL/CLR/COP/FLL/CPS destination. **The PC is the sole clearer**, so a gate-refused clear leaves the bit TRUE on the controller and the ladder will never take it down.

The consequence ranking is not uniform, and `MomentaryPulse`'s class doc is careful about it (`MomentaryPulse.cs:100-133`):

Bit	Shape	Cost of a stuck TRUE
`[30].1` Hold	**level-sensitive on a sealed coil** (L5K:12072)	keeps commanding Hold; the Resume rung cannot break the seal, so **Resume becomes impossible from any source** until someone clears the bit from Studio 5000
`[30].0` Resume, `[30].2` Reset, `[30].3` Silence	ONS-gated	does **not** repeat the command; holds the one-shot input high so the *next* command cannot fire — and because each rung ORs the PanelView, the alarm path and the hardwired inputs into **one shared ONS**, it blocks that command from every station, not just Ring
`[30].10` clock sync	ONS-gated `SSV`	stamps once, then latches `ons1.0` so **no later clock sync can ever fire**

Bit	Shape	Cost of a stuck TRUE
`[27].4` inventory ack	generic confirm rule, `PC_HandShake_Word = 12`	the rule re-zeroes the notify mirror every scan, so the controller can never raise a **new** inventory notification

"Paying a libplctag timeout beats latching that."

`MomentaryPulse` — the shared clear

`Ring/Services/PLC/MomentaryPulse.cs` is the de-assert half of every momentary pulse Ring issues, extracted from per-site duplicated loops.

`ClearWithRetry` (`:167-216`):

- `ClearAttempts = 3` by default, **spaced** by `ClearDelayMs = 250` and never before the first attempt. Spacing is the point — the pre-F013 loop retried instantly, so all three attempts landed inside the same instant of degradation and were refused together.
- **A throwing attempt does not abort the remaining attempts** (`:192-199`) — that was the failure the duplicated loops had.
- On total failure it logs `STUCK TRUE` naming the tag, and *only if* `ArmBackgroundReclear` is explicitly on, arms one detached re-clear.

`ArmBackgroundReclear` is **off by default** (`MomentaryPulse.cs:44`, in the `MomentaryPulseOptions` class that lives in the same file) and every site opts in explicitly. The default-off is load-bearing for the test suite: a detached task re-invoking a caller's clear delegate would mutate test doubles after the test finished asserting, in a suite that runs with no parallelism.

The armed re-clear (`:227-...`) **attempts the wire on every tick and does not wait for the heartbeat gate**, and the comment explains the bug that taught this: it used to poll `AllowPlcWrites()`, which requires the heartbeat to be *advancing* — precisely the condition that is false on the degraded link the mitigation exists for. It therefore "waited out its whole cap and gave up WITHOUT EVER TRYING THE WIRE, throwing away the heartbeat exemption F013 bought." It is still bounded: one task per failed pulse, one wall-clock budget (`ReclearMaxWaitMs`), one attempt ceiling (`ReclearMaxAttempts = 120`), because the controller caps EIP sessions per source IP.

For the `PC_Write_Integer[30]` family the budget is configurable: `PlcSettings.CommandBitReclearSeconds` (`AppSettings.cs:370`), defaulting to `PcWriteInteger30BatchControlPlcService.DefaultCommandBitReclearSeconds = 1800` (30 minutes) and clamped to 60 s .. 4 h (`:126-132`, `:120-121`). The reason it is measured in tens of minutes: the 2026-08-20 site visit produced a read stall of roughly 25 minutes, and a re-clear that gives up inside that stall leaves Hold latched.

The set of files allowed to call `WriteClear` is pinned by a test.
`Ring.Tests/MomentaryPulseClearGateTests.cs:752-777` scans every `.cs` under `Ring/` for `.WriteClear(` and fails on anything outside the approved set: `PlcTagWriter.cs` (the definition),

`PcWriteInteger30BatchControlPlcService.cs`, `ProcessorTimeDatePlcService.cs`, `ShiftControlPlcService.cs`, `InventoryEventCaptureService.cs`, `AlarmSilencePlcService.cs`. Its message: "*never widen this list to make a build green.*"

A sibling test in the same file pins that `PlcTagWriter.WireWriteHookForTests` — the test seam that stands in for the libplctag call, deliberately placed **after** every gate (`PlcTagWriter.cs:91-100, 207-208`) — is never assigned by production code (`MomentaryPulseClearGateTests.cs:779-793`).

Word-30 serialization

`PcWriteInteger30BatchControlPlcService` holds a private static `Word30PulseGate` lock (`:40`) and takes it for the whole pulse (`PulseBit` → `PulseBitSerialized`, `:182-186`), including the clear (`SerializedClear`, `:294-298`). `Monitor` is re-entrant, so when `MomentaryPulse` later invokes the same delegate from a background re-clear it reacquires the same gate and cannot interleave with a newer sibling pulse. `RunUnderWord30Gate<T>` (`:329-333`) lets `ProcessorTimeDatePlcService` — which also writes into word 30, at bit 10 — share it. `WriteSurfaceRegisterTests.cs:1129-1143` asserts, by source scan, that the gate, the `lock` and `SerializedClear` are all still present.

One more behaviour in that service worth knowing, because it looks wrong until you read the comment: **a failed assert still runs the clear** (`:227-247`). `PlcTagWriter.WriteInternal` swallows every libplctag exception and returns false, so a timeout or a *lost CIP reply* is indistinguishable from "the write was refused". If the request did land, the bit is TRUE while Ring reports `Failed`. The clear is cheap, idempotent, and owned by Ring — `PC_Write_Integer[30]` is PC1's block, so writing `False` can never stomp another station's command bit.

4.7 The verified per-rail gate order

Each row is what the source file actually does, in order, for its main write entry point. `PlcTagWriter` gates (marked ▸) run *inside* the funnel and therefore apply to every rail that uses it.

Rail (file)	Order of gates as coded
`PlcTagWriter.WriteCore`	▸ ReadOnly → ▸ Demo → ▸ Endpoint → ▸ Heartbeat → wire (`:143-208`)
`PlcTagWriter.WriteClear`	▸ ReadOnly → ▸ Demo → ▸ Endpoint → *heartbeat warns only* → wire
`BatchStartTankPlcWriter.TryWriteMember`	ReadOnly (`:74`) → roster-change lock (`:93`) → roster-load-failure lock (`:107`) → `StoragePlcIndex/NoWrite` (`:118`) → Heartbeat (`:163`) → Demo (`:174`) → Endpoint (`:202`) → inline libplctag

Rail (file)	Order of gates as coded
`TVCControlPlcService` (per method)	ReadOnly (`:189, 206, 225, 243, 260, 272, 343`) → `MainTvcWriteAuthorization` (`:266`, `:278`) → `TvcCoolingWriteAuthorization` (`:292`) → `TankInstalledWriteGate` (`:420`) → Heartbeat (`:195, 212, 231, 249, 298, 369`) → Demo (`:506`) → Endpoint (`:517`)
`UseTankControlPlcWriter`	ReadOnly (`:137, 156, 175, 197`) → Heartbeat (paired per method, `:143-147, 162-166, 181-185, 203-207`) → `TankInstalledWriteGate.AllowsAdditiveWrite` (`:224`) → Demo (`:320`, returns **false**) → Endpoint (`:331`); inline libplctag
`UseTankFormulaPlcWriter`	`EnableUseTankFormulaWrite` (`:117`) → ReadOnly (`:148, 251`) → Heartbeat (`:176`) → Demo (`:187, 264`) → Endpoint (`:275`)
`TankAgitatorPlcWriter`	`EnableHmiAgitatorCycle` (`:72`) → ReadOnly (`:162`) → Heartbeat (`:181, 292`) → Demo (`:191`) → Endpoint (`:216`); index via `RosterWriteIndex.StorageWriteIndexForLegacyWindow` (`:126`)
`InventoryAmountPlcWriter`	ReadOnly (`:280`) → Heartbeat (`:290`) → Demo (`:299`) → Endpoint (`:315`); affordance additionally requires `EnableInventoryAmountWrite` (`:71-77`)
`BatchFormulaPresetPlcWriter`	ReadOnly (`:396`) → Demo (`:482`) → `EnableFormulaBankWrite` (`:505`) → Heartbeat (`:535`) → Endpoint (`:544` calls `_io.TryPrepare`, which calls `PlcWriteEndpointGuard.AllowsWriteTo` at `:811`) → active-batch interlock (`:553-586`)
`PcWriteInteger30BatchControlPlcService`	word-30 lock (`:184`) → ReadOnly (`:190`) → Heartbeat (`:192`) → endpoint resolve → `PlcTagWriter` ▸ gates → assert → clear
`ShiftControlPlcService`	`EnableShiftControlPlcWrites` (`:159`) → ReadOnly (`:292`) → Heartbeat (`:303`) → `PlcTagWriter` ▸ gates
`ProcessorTimeDatePlcService`	`EnableProcessorTimeDateSync` → ReadOnly (`:359`) → Demo (`:369`) → Heartbeat (`:378`) → `PlcTagWriter` ▸ gates; clear under the shared word-30 gate (`:613`)
`AlarmSilencePlcService`	ReadOnly (`:66`) → custom-tag authorization `EnableCustomSilenceLatchTag` (`:73-77`) → bit-address validation (`:101`) → Heartbeat (`:105`) → `PlcTagWriter` ▸ gates
`InventoryEventCaptureService`	Demo (`:223`) → ReadOnly **and** `EnableInventorySnapshotPlcAcknowledge` (`:372`, `:426`) → `PlcTagWriter` ▸ gates (**no outer heartbeat gate** — documented)
`TankNamePlcWriter`	`EnableTankNamePlcNameSync` → ReadOnly (`:130`) → Heartbeat (`:178`) → Demo (`:188, 275`) → Endpoint (`:285`)

The `BatchFormulaPresetPlcWriter` row runs through `EvaluateGates`, which also checks the formula-number range (`:412`) and validates the payload (`:428-475`) between ReadOnly and Demo — omitted from the row above because neither is a write-path safety gate. `:505` is where `FeatureEnabledProvider()` is actually invoked; `:211` (`DefaultFeatureEnabled()`) is only the default provider's definition, not a call site on the write path.

The `UseTankControlPlcWriter` row is one of the "return `false` on suppression" writers named in §4.1 — its `DefaultInt16Writer` (`:314-336`) is not the fake-success idiom: a demo suppression or an endpoint refusal both return `false`, same as read-only.

The `AlarmSilencePlcService` row deserves a note: `AlarmSettings.SilenceLatchTagName` defaults to `"PC_Write_Integer[30].3"` and `EnableCustomSilenceLatchTag` defaults `false` (`AppSettings.cs:626, 634`). Retargeting the silence pulse is gated because "*the only override anyone reaches for is `B3_bit[15]`, which is NOT a legacy PC write at all: it is the ladder's own OTL latch (Alarm/Main L5K:9735), consumed and self-cleared by `ST_ROUTINE Alarm_CTRL` (L5K:9344). Writing it would produce a partial, horn-still-ringing silence.*" (`WriteSurfaceRegisterTests.cs:1077-1083`.)

4.8 The setpoint write queue

`Ring/Services/PLC/PlcSetpointWriteQueue.cs` serializes operator setpoint writes. It is **dispatch, not safety** — and that is the single most important thing to know about it.

The defect it closes

Every spinner click used to queue its own detached `Task.Run` and the services wrapped the wire call in a second one, so nothing serialized two writes to the same tag. With libplctag timeouts of 3–5 s, two in-flight EIP writes can complete in the reverse order they were issued and the controller settles on an intermediate value the screen no longer shows. On `Tank_IO[N]` it is worse than ordering: that write is a **read-modify-write of a SINT carrying five other physical output bits**, so two concurrent commands lose an update on `O_Fill_valv` / `O_Heat` / `O_Cool` / `O_Resin_*` (`PlcSetpointWriteQueue.cs:79-88`).

The contract

- **Serialized per tag key.** One tag never has two writes in flight; different tags proceed concurrently, so a slow tank cannot stall another screen.
- **Within a tag, a burst from one control collapses to its LAST value** after a debounce — `SpinnerDebounceMs = 300` (`:118`), `NoDebounceMs = 0` for single-shot commands (`:121`).
- **Coalescing is scoped to the COALESCE KEY (the source control), never to the tag.** Two different controls that share a tag — the Main-Heating and Main-Cooling spinners both write `TVC[N].Preset_Temp` — keep their own pending entries, so neither operator action is silently deleted. They are merely prevented from overlapping.
- **The newest submission for a control is dispatched last**, so the controller settles on the operator's most recent intent (`:222-226`).

The non-contract, stated in the source

> "This class is PURE DISPATCH and holds no safety gate of its own. It never consults `PlcWriteGuard`, `PlcHeartbeatConnectionTracker` or `DemoModeData`." — `PlcSetpointWriteQueue.cs:100-107`

Adding a copy of those checks here would either double-gate (drifting out of sync with the originals) or, in the read-only case, break the deliberate suppression contract — a suppressed operation is not a successful write. The checks live at the bridge that enqueues and at the wire.

It is not for momentary/pulse bits. Latest-value-wins on an assert/clear pair would delete the clear half. Only level-valued setpoints and level-valued commands belong here. `Tank_IO[N].O_Agitat` qualifies precisely because the ladder drives the same bit every ~300 ms, so collapsing a burst to its final state cannot strand the output.

Two engineering details worth preserving

The exit race. The emptiness check and the `Pumping = false` flip must happen in the *same* lock acquisition. Split them and a `Submit` that sees `Pumping == true` after the pump has decided to quit would buffer the operator's final value and start no pump — "a silently dropped setpoint, which is worse than the ordering bug" (`:302-310`).

The write runs outside the lock, always (`:330-346`) — it blocks for up to the libplctag timeout, and holding `Gate` across it would stall every `Submit` on that tag and deadlock any write submitted from its own completion. A throwing write is caught and logged: one bad write must never kill the pump.

Tag keys and coalesce keys

`Ring/Services/PLC/PlcWriteTagKeys.cs` builds the canonical serialization keys so two different code paths writing the same physical tag land on the same slot — concretely, the TVC bridge and the Use Tank bridge both write `Tanks[N].Agt_on_pre` through different services (`:9-11`).

Builder	Produces
`TankMember(n, member)`	`Tanks[n].<member>`
`TvcMember(n, member)`	`TVC[n].<member>`
`TvcControlPair(n)`	`TVC[n].Temp_mode` — a *named* key kept distinct from the tank-side `Tanks[N].Temp_mode` gesture so the queue can never coalesce the two (`TankTempModeWriteTargetTests` pins that they differ)
`TankIoOutputByte(n)`	`Tank_IO[n].ZZZZZZZZZZTank_I_O9` — the discrete-output SINT carrying `O_Agitat` at bit 1
`Coalesce(tagKey, controlId, callerContext)`	the source-control identity
`TargetsTank(tagKey, n)`	matches the **bracketed** index, so `Tanks[1]` does not match `Tanks[11]`

`TvcControlPair` used to be a composite `TVC_Ctrl[N].enabled+TVC[N].Temp_mode` because the gesture was a three-step read-modify-write. That two-step is gone: the ladder recomputes `TVC_Ctrl[n].enabled` every scan, and the recompute precedes the gate that consumes it in the same pass, so a value Ring wrote was never observed by the control logic (`PlcWriteTagKeys.cs:32-39`).

`SetpointEchoGuard` — "is this a genuine operator change?"

`Ring/Services/PLC/SetpointEchoGuard.cs` is a small pure class with an outsized job. It remembers the value each writable control was **seeded** with (read back from the controller) and, afterwards, the value it last successfully queued.

It suppresses three failure paths that all shipped as real write-on-open bugs:

1. XAML-init-time handler fires (`IsSelected="True"` / `Text="10"` attributes raise `SelectionChanged` during `InitializeComponent`);
2. programmatic seed-time fires (assigning `SelectedIndex`/`Text` re-enters the same handler — the fix itself would otherwise become a guaranteed write-on-open);
3. unchanged-value focus-out (tab straight through a `TextBox` and the `LostFocus` handler writes the hardcoded XAML default over the real setpoint).

Three deliberate behaviours:

- **A key that was never seeded is NOT writable.** `ShouldWrite` returns false for an untracked key (`:76-81`). **No read-back ⇒ no write.**
- **Recording is explicit and happens only after the write was accepted for dispatch** (`:56-60`), so a rejected write leaves the baseline at the last known-good value and the next identical attempt still goes out.
- Values compare as trimmed ordinal strings, so `"10"` and `" 10 "` are the same setpoint.

It is **one instance per window, never static** — and because views are cached singletons (`NavBar.GetOrCreateView`), instance state survives navigate-away, so **every window must call `Reset()` at the top of its `Loaded` handler and re-seed** or it will compare against an hours-old baseline (`:32-35`).

`SetpointRefreshCoordinator` (`Ring/Services/PLC/SetpointRefreshCoordinator.cs`) turns a *settled* write into a re-read of the screen's setpoints, hanging off the queue's `WriteCompleted` and `WriteSubmitted` events — the latter because a read already in flight when the operator changed something may predate that change, and applying its result would repaint the edit away (`PlcSetpointWriteQueue.cs:161-171`). `SetpointSeedContext` and `TankControlSetpointReader` are the read-back seam that fills the guard's baselines.

4.9 The write evidence journal — and exactly what it covers

`Ring/Services/PLC/PlcWriteEvidenceJournal.cs` is an append-only JSONL journal of `PlcWriteResult` records, at `%ProgramData%\Ringwood\Ring\plc-write-evidence.jsonl` (`:75-79`), UTF-8 without BOM, size-rotated at `DefaultMaxBytes = 10 MB` with `DefaultRetainedArchives = 5` (`:17-18, 57-73`),

serialized under a lock.

Its class doc states the safety property precisely: **"This is evidence only: it exposes no replay API, so restart can never resend a dangerous command automatically."**

Scope — be exact about this. The journal is *not* wired into every write path. A grep for `PlcWriteEvidenceJournal` under `Ring/` returns four production references: its own file, `PlcSetpointWriteQueue` (`:124`, `:130`, `:147`), `Services/Audit/IncidentExportBuilder.cs` (which reads it back for an incident export, `:144`, `:158`), and a doc-comment cross- reference in `Services/Audit/UiAuditJournal.cs:98`. **Only writes that go through `PlcSetpointWriteQueue` produce evidence lines.** Batch-start writes, the `PC_Write_Integer[30]` pulses, the formula-bank rewrite and the inventory writes do not — their record is the application log, the comm log, and the per-writer outcome types.

Each queued write produces **two** lines, in order:

When	Status	Operation
accepted for dispatch	`PendingDispatch`	`"Queued"`
settled	`WrittenUnverified` or `Failed`	`"Settled"`

The pending record is written **before** the write becomes visible to a pump, because the pump starts inside the slot lock and could otherwise journal the settled record first, inverting the evidence order (`PlcSetpointWriteQueue.cs:196-201`).

The settled detail is deliberately modest: "*Writer reported success; controller readback not proven by this queue*" (`:349-352`). `WrittenUnverified` means exactly that — see `PlcWriteStatus` (`Ring/Services/PLC/PlcWriteResult.cs:10-21`), where `WrittenAndVerified` is a separate value and `ControllerAccepted/Verified` are separate predicates (`:41-42`).

Each record carries a `WriteId` — a GUID unique per `Submit` call — because the audit trail must join on the *submission*, not the coalesce bucket: a control that writes twelve times in a shift has one coalesce key and twelve writes, and correlating on the bucket would report a dozen operator actions as one (`PlcSetpointWriteQueue.cs:29-38`).

The journal is not covered by any backup. See `docs/CONFIG_AND_STARTUP.md` and chapter 05.

4.10 The write-surface register — how the surface stays enumerable

`Ring.Tests/WriteSurfaceRegisterTests.cs` is, in its own words, *"the CENSUS TRIPWIRE… Every other test in this lane proves something about a write path that is KNOWN. This one exists so a write path cannot become unknown."*

It is the single most valuable safety artifact in this repository. Treat it as part of the write-path contract, not as test hygiene.

The register

A `Dictionary<string, WriteClass>` (`:93-199`) with one row per file that can write to a controller, keyed by a forward-slashed path relative to `Ring/`. Six classes, and the class is **not decorative** — the proof matrix's BENCH-OUTSTANDING column is generated from it:

`WriteClass`	Meaning	Wire proof
`UdtMember`	dotted UDT-member writes (`Tanks[N].X`, `TVC[N].X`, `Tank_IO[N].X`)	locally provable **only up to the wire seam**; controller acceptance is bench-only
`FlatTag`	flat scalar/array tags	the only class the local simulator can serve
`BitPulse`	BOOL bits on `PC_Write_Integer[N].b`	the simulator **refuses** these (`ErrorUnsupported`); bench-only
`Infrastructure`	gates, queues, coordinators, the guard itself	no tag of its own
`UiDispatch`	code-behind that turns a gesture into one of the writers	—
`DeadCode`	reachable from no call site; retained only because deletion needs a csproj edit	—

The five discovery signals

The census is the **union** of five independent textual signals, and its history is instructive: it originally recognised a write path only by the presence of the gate itself, so *"the one file shape it most needed to catch — a new writer that builds its own libplctag `Tag` and never consults the guard — matched neither alternative and was invisible to every assertion here"* (`:19-25`).

#	Signal	Regex / mechanism
1	names the gate	`\bPlcWriteGuard\b` (`:234`)
2	builds the funnel	`new\s+PlcTagWriter\s*\(` (`:237`)
3	submits to the queue	the literal `PlcSetpointWriteQueue.Instance.Submit(` (`:257-258`)
4	**writes a raw libplctag tag** — gate-independent	binds an identifier to any writable libplctag type, then calls `.Write(` / `?.Write(` / `.WriteAsync(` **on that identifier** (`:292-341`)
5	declares or calls a writer-service entry point	a name-bound alternation of the actual public `Pulse*`/`Write*` vocabulary (`:401-409`)

Each signal documents its own limits, and you should read those before treating a green run as "there are no other write paths":

- **Signal 3 is spelling-bound.** A file that took the queue as a constructor dependency and called `_queue.Submit(` would not be discovered — and nothing else would necessarily catch it. That is the known residual gap, bounded today only because the queue is a singleton with no DI registration (`:239-255`).
- **Signal 4 is a single-file textual scan, not a call graph.** It cannot see a tag constructed in one file and written by a helper in another, a tag handed to a method as a parameter, a write reached through an interface/delegate/ reflection, or a tag type libplctag adds after 1.5.2 (`:314-319`). Identifier-scoping the `.Write(` is load-bearing, not fussiness: 22 files under `Ring/` construct a raw tag and most are pure readers that nonetheless call `.Write(` on loggers and streams (`:320-325`).
- **Signal 5 is a maintained list, not a derivation.** Adding an entry point to a writer service without adding it here leaves its callers invisible again (`:396-399`).

Signal 5 exists because a call-site sweep on 2026-08-25 found **eight** files dispatching PLC writes that no signal saw and no register row named — and the tripwire passed green throughout, because those files were invisible in *both* directions: absent from the discovered set (so not "unregistered") and absent from the register (so not "stale") (`:359-374`).

The assertions

Test	What it catches
`Every_file_on_the_PLC_write_surface_is_in_the_pinned_register`	a new writer, or a screen that starts dispatching writes
`The_register_contains_no_file_that_has_left_the_write_surface`	a stale row describing code that no longer exists
`The_register_is_not_silently_shrinking`	bulk row deletion — a blunt `Register.Count >=` floor, **set to the exact current count on purpose, zero slack**. Read the file for the number; if you add a row, raise it in the same commit (`:486-511`)
`Every_file_that_writes_a_raw_libplctag_tag_is_in_the_pinned_register`	a wire-level writer, found without reference to the gate
`Every_file_that_writes_a_raw_libplctag_tag_also_consults_the_write_guard`	**the safety property.** A raw writer that never names `PlcWriteGuard` writes to the live plant *in read-only mode*. Note the scope: this is **file-scoped, not method-scoped** — a file with one gated write method and a second ungated one passes (`:549-554`)
`The_raw_writer_detector_still_detects`	**anti-blindness.** Exercises the predicate against twelve synthetic writer shapes the codebase does not yet contain, asserts two non-matches, and floors the live population — because the two tests above pass *vacuously* the moment the detector stops matching

Test	What it catches
`The_dispatcher_entry_point_detector_still_detects`	the same anti-blindness for signal 5, plus a **drift check**: it reads the public `Pulse*`/`Write*` declarations back out of the four momentary-pulse services and requires the signal to match a call to each
`Every_writer_that_needs_bench_evidence_is_classified_as_needing_it`	a mis-filed `WriteClass` that would turn a bench-only claim into a "proven" one
`No_UI_file_writes_a_controller_tag_without_going_through_a_writer_service`	a code-behind that news up its own tag, bypassing ReadOnly, the heartbeat gate, the demo seam and the queue in one move
`The_writers_that_reach_libplctag_without_a_test_seam_are_named_and_bounded`	**an honesty pin.** Exactly three writers build their `Tag` inline with no injectable seam, so no local test can observe what they put on the wire: `BatchStartTankPlcWriter`, `InventoryAmountPlcWriter`, `TankAgitatorPlcWriter`. Their guard chains are proven; the wire shape is not. This test stops that list growing (`:887-931`)
`Every_UI_dispatch_screen_on_the_write_surface_can_actually_be_reached`	a registered write path hanging off a screen nothing constructs — which would inflate the apparent reviewed write surface with paths no operator can trigger
`Hardware_signoff_write_capabilities_remain_fail_closed_in_the_shipped_settings`	see §4.2
`Batch_start_verification_seams_are_test_only_and_word30_pulses_keep_one_shared_gate`	production assigning `BatchStartPlcWriteHelper.WriteMember/ReadMember`; loss of the word-30 lock

Two details in `Every_UI_dispatch_screen_on_the_write_surface_can_actually_be_reached` are worth carrying into any similar test you write:

- It resolves a screen's type name from the paired XAML's `x:Class`, **not** from the file name, because those differ in this repo — `Views/Shifts/ShiftControl.xaml` declares `Ring.Views.Shifts.ShiftControlView`. Deriving from the file name reported that screen as an orphan, and the only ways to clear a false orphan are an exemption or a deletion, either of which "would have taken the shift family's ONLY operator gesture off the reviewed surface" (`:1003-1009`).
- Its `ApplicationDefinition` exclusion is derived from the csproj rather than hardcoded, so it cannot drift (`:977-984`). The comment records the lesson: before `Ring/Program.cs` was deleted, the only thing that looked like a construction site for `App` was a **commented-out** `var app = new App();` inside that dead file — dead code can prop up a reachability claim.

The `UnreachableUiDispatchExemptions` dictionary (`:940-947`) is currently **empty, and worth keeping that way**. The one entry it ever held — `Views/UseTanks/UseTank1Window.xaml.cs`, an unreachable clone of `MF1Window` — was removed when that screen and its register row were deleted, which is the intended way an entry leaves the list.

Worth restating here: `Ring/Views/UseTanks/` contains three windows — `MF1Window`, `UseTank2Window`, `UseTank3Window`. A fourth, `UseTank1Window`, was an unreachable clone of `MF1Window` and was deleted along with its register row; the only remaining references to it anywhere in the tree are the two historical comments in `WriteSurfaceRegisterTests.cs` recording its deletion.

4.11 Two writes, end to end

Reading the gates one at a time makes them look like a checklist. They are not; they are a sequence with different shapes on different rails. These two walkthroughs are the whole model in motion.

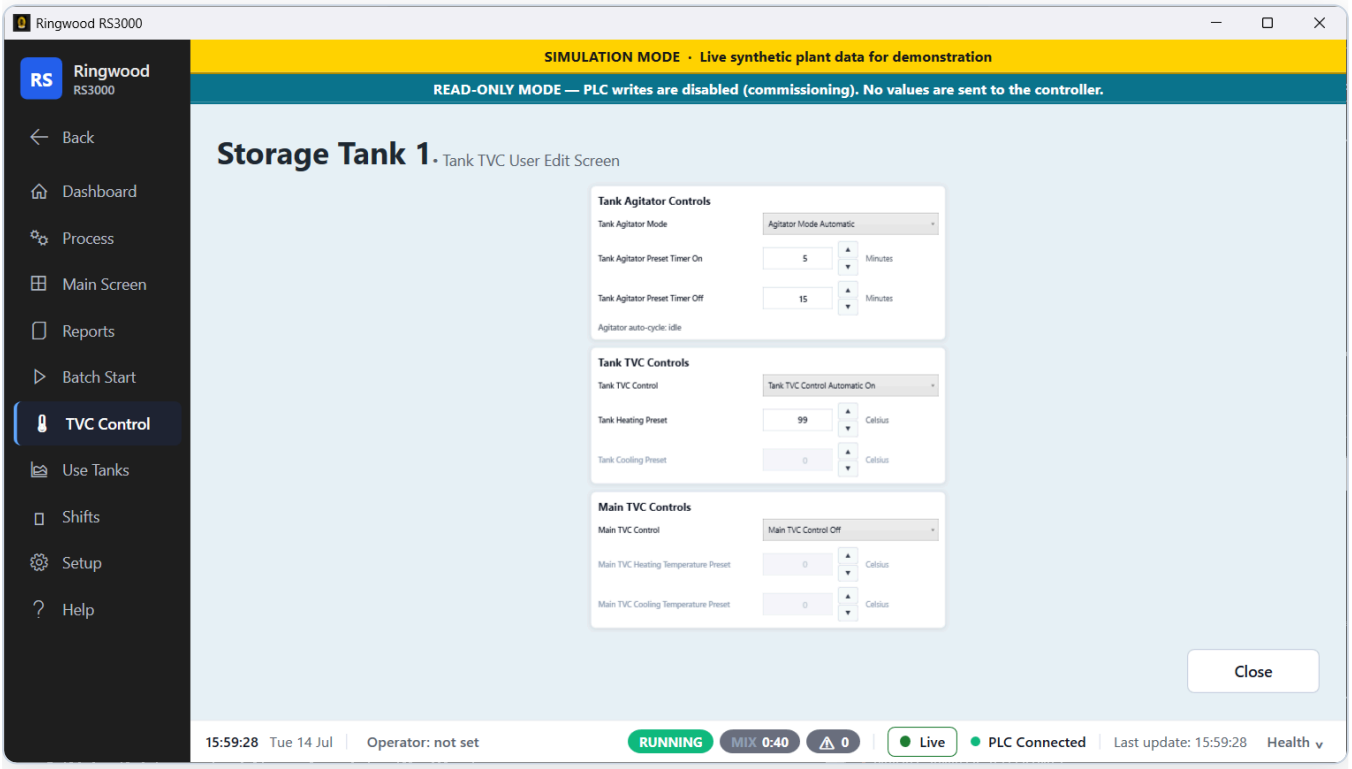
(a) An operator presses **Hold**

1. **The gesture.** `NavBar.HandleButtonClick` receives `HoldButton`. Because that name is in the `isNonNavAction` set, the unsaved-changes guard is **skipped** — a dirty editor must never be able to abort a safety command (`Ring/Views/UserControls/NavBar.xaml.cs:437-444`). Control reaches `OnProcessHoldClicked()` (`:513-516`).
2. **Serialization.** `PcWriteInteger30BatchControlPlcService.PulseBit` takes the process-wide `Word30PulseGate` lock for the whole pulse, assert *and* clear (`:182-186`). No other word-30 command can interleave.
3. **Gate 1 — read-only.** `PlcWriteGuard.IsReadOnly` → `LogBlocked` and return `BatchControlWriteResult.BlockedReadOnly` (`:190`). **Pre-cutover, the walk ends here**, and the operator is told the controller was not updated.
4. **Gate — heartbeat (outer).** `AllowPlcWrites()` must be `Connected`, so a pulse cannot *start* on a degraded link (`:192-196`).
5. **Endpoint resolution.** `GetPlcIp()` / `GetPlcPath()`, each re-read from the config file; a throw here returns `Failed` (`:198-208`).
6. **The assert.** `new PlcTagWriter("PC_Write_Integer[30].1", ip, BOOL, path)` then `writer.Write("True")` → `WriteCore` runs ▸ReadOnly, ▸Demo (live re-check), ▸Endpoint, ▸Heartbeat, then the wire.
7. **If the assert failed — the clear still runs.** `Write` returning false does **not** mean the request never reached the controller: `WriteInternal` swallows every libplctag exception, so a timeout or a lost CIP reply looks identical to a refusal (`:227-247`). Skipping the clear here is what would strand Hold TRUE.
8. **The pulse.** `Thread.Sleep(PulseDurationMs)` inside a `try`, with the clear in the `finally` so it runs even on an exception.
9. **The clear.** `MomentaryPulse.ClearWithRetry(() => SerializedClear(writer), …)` → three spaced attempts (250 ms apart, never before the first), each through `PlcTagWriter.WriteClear`, which is **heartbeat-exempt but still ReadOnly-, Demo- and Endpoint-gated**. `SerializedClear` re-takes the word-30 lock (`Monitor` is re-entrant) so a background re-clear can never interleave with a newer sibling pulse (`:294-298`).

10. **If every attempt failed.** `STUCK TRUE` is logged naming the tag, and — because this service opts in (`DefaultClearOptions()`, `:67-74`, sets `ArmBackgroundReclear = true`) — one detached re-clear is armed, which **attempts the wire on every tick regardless of the heartbeat**, bounded by `CommandBitReclearSeconds` (default 1800 s, clamped 60 s..4 h) and `ReclearMaxAttempts = 120`.
11. **The result.** `Written` only when both halves succeeded; `PulseHoldResumeResetBool` collapses the tri-state so that **only `Written` maps to `true`** (`:179-180`).

Count the gates that stood between the button and the bit: read-only, outer heartbeat, demo, endpoint, inner heartbeat — plus a lock, a retry policy and a stuck-bit mitigation. That is the shape every command write should have.

(b) An operator nudges a TVC preset spinner



The gesture this walkthrough traces: a preset spinner on a TVC control window. Everything from here to the wire is the setpoint write queue, the echo guard, and six gates.

1. **The gesture.** The spinner's handler in a `Views/TVCcontrol` window calls into `TVCWindowPlcBridge`.
2. **Echo guard.** `SetpointEchoGuard.ShouldWrite(key, value)` — **false unless the control has a read-back baseline and the value differs from it** (`SetpointEchoGuard.cs:76-81`). This is what stops an init-time or seed-time handler fire, and a tab-through `LostFocus`, from writing the XAML default over the real setpoint.
3. **Index resolution.** For a storage screen the PLC index comes from `RosterWriteIndex.StorageWriteIndexForLegacyWindow(key)`, which consults both sticky roster locks *before* the lookup and returns `NoWrite` if either is latched (`RosterWriteIndex.cs:269-274`).

4. **Enqueue-time heartbeat check** at the bridge, then `PlcSetpointWriteQueue.Instance.Submit(tagKey, coalesceKey, 300, write)`. The tag key comes from `PlcWriteTagKeys`, so two screens writing one physical tag share one slot.
5. **The queue journals `PendingDispatch` before the pump can see the write** (`PlcSetpointWriteQueue.cs:196-201`), coalesces this control's earlier pending value if any, debounces ~300 ms, and dispatches **outside every lock**.
6. **The write delegate runs the real gates** — `TVCControlPlcService` checks read-only, the applicable commissioning holds (`MainTvcWriteAuthorization`, `TvcCoolingWriteAuthorization`, `TankInstalledWriteGate`), the heartbeat, demo and the endpoint. **The queue itself checked nothing.**
7. **Settlement.** `WriteCompleted` fires on the pump thread; the queue journals `WrittenUnverified` or `Failed`; `SetpointRefreshCoordinator` re-reads the screen's setpoints so the display shows controller truth rather than the operator's optimism; and only *now* does the screen call `SetpointEchoGuard.RecordWritten` — so a rejected write leaves the baseline at the last known-good value and the identical retry still goes out.

4.12 What a new writer owes

If you add a write path, all of the following, in the same change:

1. **Route it through the gate stack.** `PlcWriteGuard` → the relevant `Enable*` flag → any applicable commissioning hold → `PlcHeartbeatConnectionTracker.AllowPlcWrites()` → the live demo re-check and `PlcWriteEndpointGuard` at the wire. Prefer funnelling through `PlcTagWriter`, which gives you four of those for free. If you must build a raw `Tag`, use the prologue shape from `BatchStartTankPlcWriter.cs:74-81`:

```
if (PlcWriteGuard.IsReadOnly)
{
    PlcWriteGuard.LogBlocked(op, tag, value, logger);
    return true;   // or the honest outcome for your rail — see §4.1
}
```

2. **Add its register row** to `WriteSurfaceRegisterTests.Register` with a truthful `WriteClass`, **and raise the count floor in the same commit**.

3. **Add a seam test** pinning tag, value, count and order.

4. **For a UDT member or a BOOL bit: add bench evidence** — a bench round-trip in `Ring.Tests/BenchPlcIntegrationTests.cs` and a row in `docs/production-readiness/WRITE_PATH_PROOF_2026-07-30.md`. The commissioning evidence required per write family is `docs/production-readiness/PLC_WRITE_COMMISSIONING_REGISTER_2026-08-02.csv`, verified by `scripts/Test-PlcCommissioningEvidence.ps1`. Remember that the bench-gated tests **early-return a vacuous pass** unless `BENCH_PLC_AVAILABLE=1`; the gate summary records `benchGatedTestCount` and `benchHardwareWasAvailable` for exactly that reason (`scripts/Invoke-ProductionGate.ps1:157-186`).

5. **Get an independent review.** A write-path change is reviewed by someone who did not write it (`CONTRIBUTING.md`).

And the two standing prohibitions:

- **Do not populate `MainTvcWriteAuthorization` or `TankTempModeWriteAuthorization`** without a controls decision.
- **Never put a UI guard or an early `return` in front of a shared handler that also dispatches PLC-write buttons.** A dirty-check or confirmation added to a handler shared by Hold / Resume / Reset can silently swallow a safety command. The live example is `NavBar.HandleButtonClick` (`Ring/Views/UserControls/NavBar.xaml.cs:415-444`), where the unsaved-changes guard is skipped for an explicit `isNonNavAction` set that includes `HoldButton`, `ProcessResumeButton` and `ProcessResetButton` — because the guard "would pop a misleading 'discard?' prompt and, if the operator keeps editing, ABORT a process command (e.g. a live Hold)". The roster-generated per-tank menus are built with a *dedicated* click handler for the same reason (`NavBar.xaml.cs:124-129`). **Enumerate every button routed to a handler before you add a guard to it.**

The step-by-step version of this, with the file edits in order, is chapter 10 §10.6.

Next: 05 — Data and Persistence.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring/Services/PLC/` — `PlcWriteGuard.cs`, `PlcTagWriter.cs`, `PlcWriteEndpointGuard.cs`, `PlcHeartbeatConnectionTracker.cs`, `MainTvcWriteAuthorization.cs`, `TankTempModeWriteAuthorization.cs`, `TvcCoolingWriteAuthorization.cs`, `TankInstalledWriteGate.cs`, `RosterWriteIndex.cs`, `FormulaBankWriteInterlock.cs`, `MomentaryPulse.cs`, `PlcSetpointWriteQueue.cs`, `PlcWriteTagKeys.cs`, `PlcWriteResult.cs`, `PlcWriteEvidenceJournal.cs`, `SetpointEchoGuard.cs`, `SetpointRefreshCoordinator.cs`, `TVCControlPlcService.cs`, `BatchStartTankPlcWriter.cs`, `UseTankControlPlcWriter.cs`, `UseTankFormulaPlcWriter.cs`, `TankAgitatorPlcWriter.cs`, `InventoryAmountPlcWriter.cs`, `BatchFormulaPresetPlcWriter.cs`, `PcWriteInteger30BatchControlPlcService.cs`, `ShiftControlPlcService.cs`, `ProcessorTimeDatePlcService.cs`, `InventoryEventCaptureService.cs`, `TankNamePlcWriter.cs`, `DemoPlcTagWriter.cs`, `DemoModeData.cs` · `Ring/Services/Alarms/AlarmSilencePlcService.cs` · `Ring/Infrastructure/Configuration/AppSettings.cs` · `Ring/Infrastructure/Configuration/PlcConnectionConfig.cs` · `Ring/Config/appsettings.json` · `Ring/Views/App.xaml.cs` · `Ring/Views/UserControls/NavBar.xaml.cs` · `Ring.Tests/WriteSurfaceRegisterTests.cs` · `Ring.Tests/MomentaryPulseClearGateTests.cs` · `scripts/Invoke-ProductionGate.ps1` · `docs/PLC_FACTS.md` · `ARCHITECTURE.md` · `CONTRIBUTING.md`

05 — Data and Persistence

Who this is for. Anyone adding a table, changing a query, touching backup or restore, or debugging "the report says zero and I do not believe it".

What you'll learn. How Ring configures SQLite and why; the repository shape and its deliberate absence of a base class; how migrations work and the one gap no test covers; the restore contract and how it is kept honest; the maintenance latch and writer quiesce; the historian and the two append-only journals; and what exports do.

Authorities this chapter defers to: `docs/production-readiness/09_DB_BACKUP_RESTORE.md` for the backup/restore *procedure*, `docs/production-readiness/10_AUDIT_TRAIL.md` for what the operator audit trail records, and `docs/CONFIG_AND_STARTUP.md` for the complete on-disk file set.

5.1 One database, resolved to the exe directory

Ring keeps everything in a single SQLite file, by default `<exe dir>\RingwoodDatabase.db` (plus its `-wal` and `-shm` sidecars).

The path is normalized at config load by `ConfigurationService.NormalizeConnectionStringDbPath`: a relative `Data Source=` is rewritten to `<exe dir>\<name>`, because repositories open the connection string verbatim and a launch with a different working directory would otherwise **silently create a second database**. Absolute paths and `:memory:` are left alone (`docs/CONFIG_AND_STARTUP.md` §2).

For a real plant install, `Ring/appsettings.production.template.json` is the starting point, and with `Production.Enabled` true, validation *requires* an absolute DB path and an absolute backup directory (`ConfigurationValidator.cs:106, 112`) — see chapter 02 §2.3, "Production mode — the strict profile" for the switch that arms this and the rest of what it checks.

Per-connection pragmas

Every repository opens its own connection per call and applies the same three pragmas through `Ring/Database/RingwoodDbAccess.cs`:

```
PRAGMA busy_timeout = 15000;
PRAGMA journal_mode = WAL;
PRAGMA synchronous  = FULL;
```

`synchronous = FULL` is an override, not a default: WAL defaults to `NORMAL`, which can lose the last committed transactions on power loss. The class comment is explicit — "A plant PC must keep the most recent commits, so force FULL fsync-on-commit" (`RingwoodDbAccess.cs:22-27`). The 15 s busy timeout exists because multiple repositories open the same file concurrently; without it, concurrent access throws "database is locked".

5.2 Repository shape

`Ring/Database/Interfaces/` holds 43 interfaces and `Ring/Database/Repositories/` holds 43 implementations (verified by count). No ORM, no Dapper — hand-written ADO over `System.Data.SQLite`.

There is no base class. Each repository is an independent class that:

1. opens a connection per call,
2. applies `RingwoodDbAccess.ApplyConcurrencyPragmas`,
3. exposes `public static void ApplySchema(SQLiteConnection)`, which its instance `EnsureTable()` also calls.

That static is the **single source of DDL truth**, shared between first-run creation and migration. Copy the shape from an existing repository rather than inventing a new one; a third pattern is the thing to avoid.

`RepositoryReadScope` — telling "no rows" from "the read failed"

`Ring/Database/RepositoryReadScope.cs` closes a real reporting hazard. Every list-returning repository traps its exceptions, logs them, and returns an empty list — deliberately, because a SQL blip must never crash the HMI. But that makes "the database could not be read" indistinguishable from "there are no records", so a failed read renders (and *emails*) as a clean "0 batches in range".

Each repository catch block calls `RepositoryReadScope.ReportFailure`. A caller that cares wraps its read in `Begin` / `Capture<T>` / `Watch` and inspects `HadFailure`. Callers that do not opt in are completely unaffected — no signature changes, no behaviour change, no allocation when no scope is active.

Threading constraint: the current scope is `[ThreadStatic]`. Repository reads are synchronous ADO calls, so the catch block runs on the caller's thread — including the report scheduler's STA render worker. **A scope must not be held across an `await`**; keep it around the synchronous read only (`RepositoryReadScope.cs` class doc).

The user-facing wording for the distinction lives in one place: `Ring/Services/Reports/ReportDataUnavailableNotice.cs`.

5.3 Migrations

`Ring/Database/DatabaseInitializer.cs` drives schema evolution with `PRAGMA user_version` steps.

- An ordered `MigrationStep` list from `GetMigrationSteps()` (`:361`), each a `(version, description, Action<SQLiteConnection>)` triple (`:331-340`).
- Steps are applied inside a **SAVEPOINT** and the version stamped atomically.
- Every step must be **idempotent** — `CREATE TABLE IF NOT EXISTS`, guarded `ALTER`.
- Step 1 is the baseline: `new MigrationStep(1, "Baseline schema (all CREATE TABLE IF NOT EXISTS)", _ => { })` (`:366`) — an empty body whose job is to stamp an aged, unversioned database (`user_version == 0`) up to 1 without altering it.

- `CurrentSchemaVersion` is the one constant to bump (`DatabaseInitializer.cs:743`). Read the file for its value; do not quote one into prose.

`SampleDbFileExistence()` (`:158`) is the first-run latch discussed in chapter 02 §2.2. `TryReadCurrentSchemaVersion()` (`:198`) reads `PRAGMA user_version` **without creating or touching** the database, which is what lets the pre-migration backup decide whether there is anything worth backing up.

> **One gap no test covers.** If you author a step as `new MigrationStep(CurrentSchemaVersion, …)` and forget to bump `CurrentSchemaVersion`, you get two steps with the same version. Every test still passes — a fresh database runs both steps — but a **plant database already at that version silently skips your migration.** Check the number by eye. (Recorded in `ARCHITECTURE.md`.)

5.4 Backup and restore

`Ring/Services/DatabaseBackupService.cs` (~646 lines) owns both directions.

Backup uses SQLite's online `SQLiteConnection.BackupDatabase` API, so the snapshot is transactionally consistent even while the live app holds connections open — and so pages still parked in the WAL are flushed into the destination rather than silently lost. It runs on startup and before every migration, into `%ProgramData%\Ring\backups` or the configured `DatabaseMaintenance.BackupDirectory`.

Restore is heavier, and the order matters:

1. Validate the source — `PRAGMA integrity_check` **plus** the expected tables being present.
2. Take an automatic safety backup of the current database to a `.pre-restore-<ts>.bak` sidecar as rollback insurance.
3. `File.Copy`, then delete stale `-wal` / `-shm` sidecars so the restored database does not open against an alien WAL frame.

The restore contract, and how it stays honest

Two arrays in `DatabaseBackupService` define what a backup must contain:

Array	Line	Applies to
`ExpectedTables`	`:43`	a backup stamped at **`CurrentSchemaVersion` or newer** — the full table set
`CoreTables`	`:144`	an **older** backup — the smaller must-have set

The selection is one line: `var requiredTables = userVersion >= CurrentSchemaVersion ? ExpectedTables : CoreTables;` (`DatabaseBackupService.cs:496`).

`Ring.Tests/DatabaseExpectedTablesDriftTests.cs` runs the **real** initializer against a temp database, reads `sqlite_master`, and asserts a *symmetric* set-difference against `ExpectedTables` — so the list cannot drift from reality in either direction without a red test. `Ring.Tests/DatabaseBackupServiceTests.cs` holds a mirrored `AllTables` literal.

The full operator procedure is `docs/production-readiness/09_DB_BACKUP_RESTORE.md`; this book does not restate it.

What the backup does *not* cover

The `.db` file, and nothing else. Not the three JSON stores under `%LocalAppData%\Ring\` — `tank_roster.json`, `operator-session.json`, `group_hardware_setup.json` — and not the two JSONL journals under `%ProgramData%\Ringwood\Ring\`. The tank roster is the one that hurts: it determines which PLC index a storage-tank write goes to (chapter 04 §4.3). **Copy that directory by hand before reimaging a plant PC.** (`docs/CONFIG_AND_STARTUP.md` §4.)

5.5 Health, the maintenance latch, and writer quiesce

Three cooperating pieces keep the file safe while it is being operated on.

`DatabaseHealthService`

`Ring/Services/DatabaseHealthService.cs` runs the three operations a live plant database needs:

1. **Startup `PRAGMA integrity_check`** — early warning of corruption from a bad shutdown, disk bit-rot or tampering. Failures are logged and surfaced; the policy choice is the operator's.
2. **Startup online backup** into the configured directory, plus a retention prune to the newest `DatabaseMaintenance.RetentionCount` auto-files.
3. **Shutdown `PRAGMA wal_checkpoint(TRUNCATE)`** — flushes every dirty WAL frame into the main file and zeroes the WAL, so the sidecar is safe to delete after the process exits. Without it, a power cut between SQLite's automatic checkpoint (every 1000 dirty pages) and the next start can lose committed rows that an operator then "cleans up" by deleting the WAL.

It also runs background maintenance on a timer and has a bounded 10 s join in `StopBackgroundMaintenance` — the model the polling coordinator's drain copies.

`DatabaseWriterQuiesce`

`Ring/Services/DatabaseWriterQuiesce.cs` stops — and puts back — **every** background writer that can touch the database, so a file swap (restore) or delete (factory reset) cannot race a timer tick into the file being replaced.

Its existence is a lesson about duplicated lists. Both dialogs previously carried a byte-identical inline list of four services, and that list was **incomplete**: `BatchStepsPoller`, `InventoryEventCaptureService`, `AlarmAlarmNumberPlcService` and `ReportSchedulerService` all kept running. "Two copies of an

incomplete list is exactly the shape that stays incomplete, so there is now one list, in one place, used by both dialogs."

Two design properties:

- **Best-effort, per writer.** Every stop and every resume is individually guarded: a writer that fails to stop must never block the restore, and a writer that fails to resume must never stop the others coming back.
- **Resume fidelity.** Each entry reports whether it was actually running before the stop, and `ResumeAll` only restarts what it actually stopped — so quiescing never silently *arms* a recorder the site had deliberately left off. Resume runs in reverse order.

On the PLC side it uses `PlcPollingCoordinator.PauseDatabaseWriters()`, which stops only `BatchStepsPoller` and leaves the read-only pollers live, so the operator's screens are not blacked out for the duration (chapter 03 §3.2).

`DatabaseMaintenanceLatch`

`Ring/Services/DatabaseMaintenanceLatch.cs` is a process-wide "the database file is being swapped or deleted right now — do not touch it" flag, held by `DatabaseWriterQuiesce.StopAll` and released only by `ResumeAll`.

The reason it is needed *on top of* the quiesce is precise and worth internalising: the restore and reset dialogs put a **modal `MessageBox`** up around the swap, and a modal message box runs a **nested dispatcher pump**. Anything already scheduled on the dispatcher keeps running inside that window, and two things there can undo the quiesce:

1. **The PLC auto-reprobe.** `MainWindow.PlcAutoReprobeTick` (30 s) marshals `StartPlcMonitoring()` onto the dispatcher, which runs `PlcPollingCoordinator.Start()` — clearing the batch-steps pause — and restarts `InventoryEventCaptureService` and `ViscometerLiveRecorder`.
2. **Dashboard refresh ticks.** `DashboardView`'s 4 s and 60 s `DispatcherTimer`s run repository *reads*, and every repository call begins with `EnsureTable()`, which issues `CREATE TABLE / CREATE INDEX IF NOT EXISTS`. Opening a SQLite connection **creates the file**, so after a factory reset a read alone is enough to leave a partially-schema'd `RingwoodDatabase.db` on disk — which defeats the first-run detection `SampleDbFileExistence` exists to protect and hands the next launch a half-built database it will treat as an upgrade.

The latch is never released on the paths that end in `Application.Current.Shutdown()`, which is the point.

5.6 Batch lifecycle and history

`BatchStepsPoller` observes `current_step` edges and drives `Ring/Services/Batch/BatchLifecycleRecorder.cs`, which on batch end calls `BatchRepository.CompleteWithUsage`. **Completion and every `IngredientUsage` row commit in one**

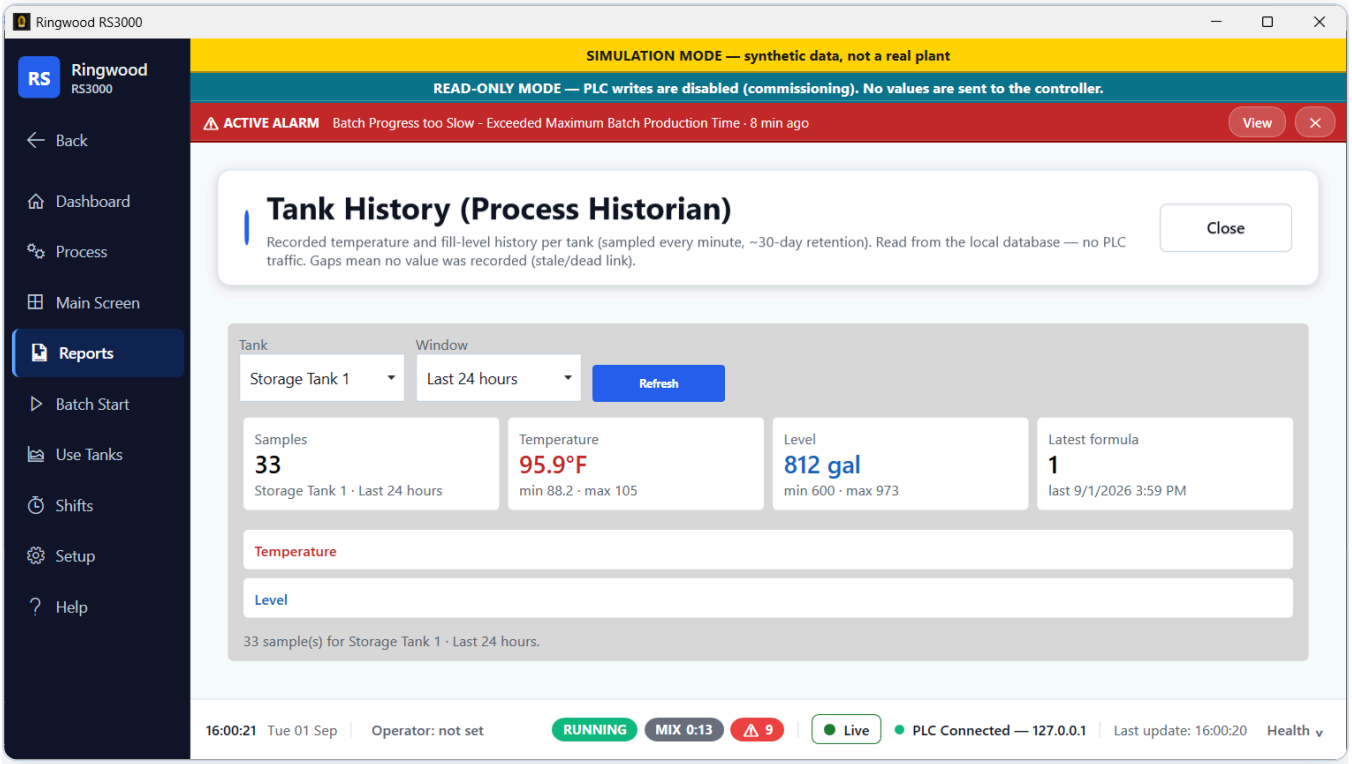
transaction — if a usage insert fails, the batch stays `Running` rather than completing with missing usage. (Older runbooks describe this as an open "Phase A" gap; it shipped, and `ARCHITECTURE.md` records the closure.)

Two startup catch-ups run on a background task from `PlcPollingCoordinator.Start()` (`:165-180`):

- **Orphan reconcile** — closes `Batch` rows left `Running` by a crash or reboot, but never interrupts a batch the plant is genuinely running. Its branch selection is factored into the pure, testable `PlcPollingCoordinator.DecideStartupReconcile` (`:459-492`) with three rules: *any* active read wins (reconcile only the other tanks, or defer if the active tank cannot be attributed); *any* failed read defers (a failed read is never a confirmed idle); otherwise it needs `BatchStepsPoller.IdleConfirmReadsRequired` confirmed idle reads before closing anything. The debounce exists because a between-steps `current_step == 0` flap during a restart used to mark the live batch Interrupted and spawn a duplicate row.
- **Missed controller verdicts** — `BatchStampsBackfillService` attaches the controller's retained batch summaries for batches that finished while Ring was down. Deliberately **bulk-only, no indexed fallback**: four 40-element fallbacks would be 160 sequential reads in one burst against a controller whose EIP session table is the scarce resource, and "a missed verdict is a cosmetic loss where a starved session table is not" (`PlcPollingCoordinator.cs:224-233`).

Related history services: `Ring/Services/Batch/BatchHistoryBackfillService.cs` (preview-first importer for the legacy six-month history), `BatchStampsMatcher.cs` (which controller stamps slot belongs to which Ring batch), and `Ring/Services/LegacyRs360ImportService.cs`.

5.7 The historian



The historian described below, as the operator sees it — one sample per tank per minute, charted on the Tank History report.

`Ring/Services/ProcessHistorianService.cs` is a singleton background service that, on a configurable cadence (`DefaultCadenceMs = 60_000`), reads the **existing in-memory** `StorageTanksSnapshotHolder.Latest` and persists one `ProcessHistorySample` row per installed tank. **It performs no new PLC reads.**

Two properties that make it safe to leave on:

- It samples only while `PlcHeartbeatConnectionTracker.AllowHeavyPlcPolling()` is true, so a starved or dead link never writes confident-looking frozen rows.
- Retention prune runs on a coarser schedule (`RetentionPruneEveryTicks`) — every 60 sample ticks at the default cadence, i.e. roughly hourly. The window is `Database.ProcessHistorianRetentionDays` in appsettings.json, editable on Setup → Database Backup → "Data retention" (default **365** days; `0` means keep forever). **Corrected 2026-09-01:** this used to be a hardcoded `ProcessHistorianService.DefaultRetentionDays = 30` constant with no config surface at all — the constant still exists as the fallback when no setting is present, but the shipped app always passes the configured value. A change only takes effect after Ring is **restarted** (the value is baked into a DI-constructed singleton at startup, not re-read live). `0` never triggers a full-table scan — `ApplyRetentionDays` returns immediately on a non-positive value without opening a connection.
- Alarm history (`PlcAlarmEvents`, `AlarmLifecycle`) has an equivalent, independently-configurable window: `AlarmSettings.AlarmHistoryDays` (default raised 2026-09-01 from 30 to **365** days; `0` = keep forever), same Setup field. Unlike the historian value, this one is re-read on every alarm tick

(`AlarmAlarmNumberPlcService`), so a change applies immediately — no restart required.

For what this cadence and retention window mean in rows-per-year, and why the row count is a query-latency non-issue, see chapter 03 §3.9 — not repeated here to keep the growth arithmetic in one place.

`Ring/Services/RuntimeAccumulatorService.cs` follows the same shape for accumulated equipment runtime, and `Ring/Services/TankLevelTrendService.cs` is an additive, read-only time-to-empty estimator over the same snapshots.

`Ring/Services/PLC/TrendingDataService.cs` is the exception in this family — it *does* construct its own `PlcTagReader` per non-snapshot tag on every 2 s tick (`PollIntervalMs = 2000`, `:32`).

5.8 The two append-only journals

Neither lives in SQLite, and neither is covered by any backup.

`UiAuditJournal` — the tamper-evident operator trail

`Ring/Services/Audit/UiAuditJournal.cs`, at `%ProgramData%\Ringwood\Ring\ui-audit\ui-audit.jsonl` (`:1040-1041`).

Integrity chain. Every record carries the previous record's hash, and `VerifyChain` re-derives every hash over the supplied files in order and reports the first break (`:377-...`). A separate `UiAuditTailAnchor` written beside the journal closes the one hole a *backward* chain cannot: a tip on disk that is behind the anchored tip means the journal was truncated (`:71-90`).

What the chain is and is not — the class states its own limits (`:116-120`): the hash is a plain, **unkeyed** SHA-256. It reliably detects accidental corruption and casual edits; someone with the same code and the patience to re-hash every downstream record can produce a self-consistent forgery. It is tamper-*evident*, not tamper-*proof*.

Writer-thread-only state means the chain is race-free without a lock: only the writer ever assigns a sequence number or a hash (`:158-159`).

Collaborators: `UiActionAuditor` (app-wide, observe-only capture of operator interactions), `UiActionRecord` / `UiActionKinds`, `UiAuditQueryService` (the `UiAuditTrailScreen` read model), and `UiScreenResolver`, which answers "which screen was the operator looking at?".

> **`UiScreenResolver` is part of the rename freeze.** It writes `GetType().Name` into the hashed record payload, so renaming a screen class splits its identity across the rename boundary: old records still verify but no longer match live labels (`ARCHITECTURE.md`).

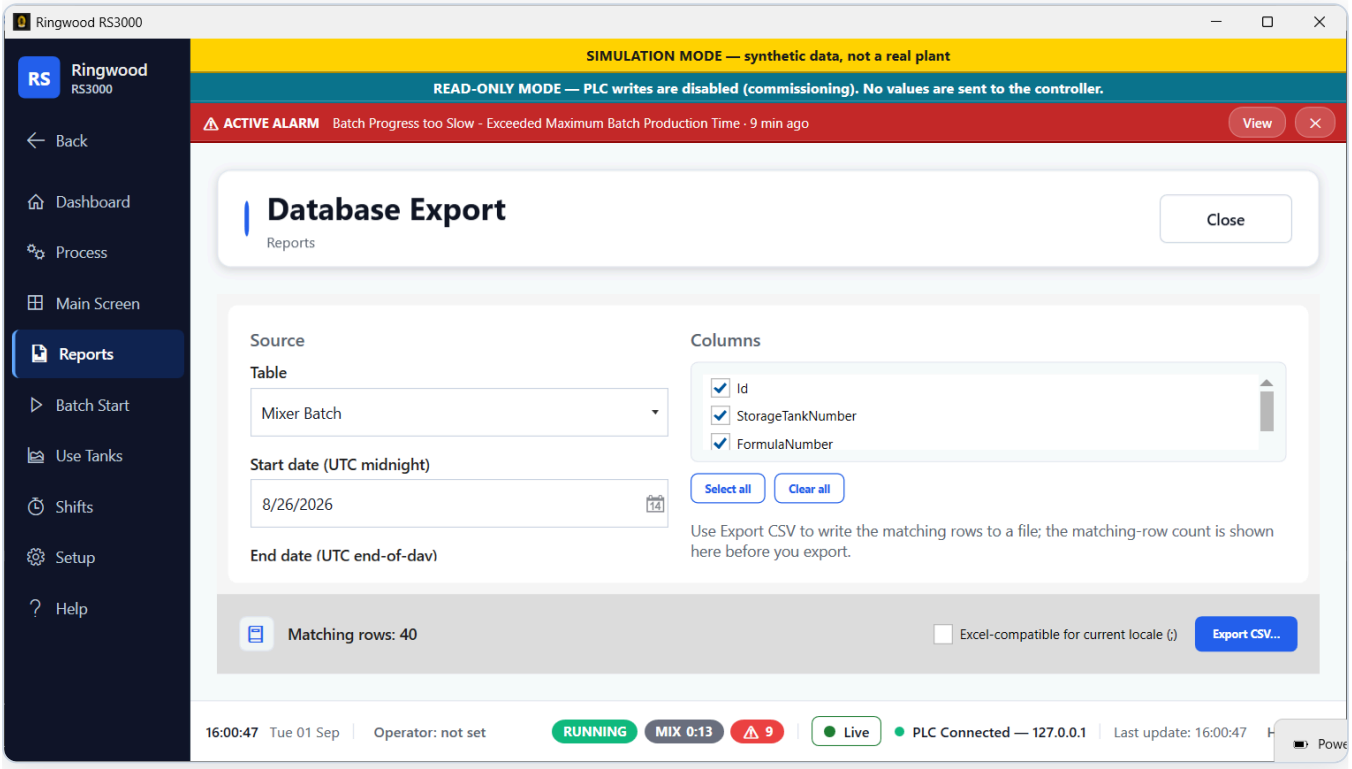
What the trail records for operators is `docs/production-readiness/10_AUDIT_TRAIL.md`.

`PlcWriteEvidenceJournal`

Covered in chapter 04 §4.9, including the important scope limit: only writes that go through `PlcSetpointWriteQueue` produce evidence lines.

`Ring/Services/Audit/IncidentExportBuilder.cs` reads both journals back for an incident export.

5.9 Exports and reports



Setup → Database Export: the table list on screen is exactly the `DatabaseExportSchema` whitelist described below, not a live query of the schema.

Database export. `Ring/Services/Export/DatabaseExportService.cs` streams selected columns from a **whitelisted** table to CSV. The whitelist is `Ring/Services/Export/DatabaseExportSchema.cs` — a static `IReadOnlyList<DatabaseExportTable> Tables` (`:19`) with a lookup by name (`:141`). A table that is not in that list cannot be exported, which is the point: the export form takes a table name from the UI. `Ring.Tests/DatabaseExportSchemaSmokeTests.cs` exercises it.

Full database export. `Ring/Services/Export/SqlDumpService.cs` is a second, independent export path — schema-generic rather than whitelisted: it reads `sqlite_master` directly (skipping only internal `sqlite_%` bookkeeping objects such as `sqlite_sequence`), so it is not limited to `DatabaseExportSchema`'s curated table list. `DumpDatabase` opens its own connection, applies `RingwoodDbAccess.ApplyConcurrencyPragmas` (§5.1) like every other repository, and runs the whole read — schema, then every table's rows — inside **one read transaction**, so the resulting `.sql` script is a consistent point-in-time snapshot even while the rest of the app keeps writing on other connections. The

output is streamed to a `.partial` sibling file, `FileStream.Flush(true)`-fsynced, and only then renamed to the final `<name>.sql` name — write-to-temp-then-atomic-rename, so a crash mid-export can leave at most a stale `.partial` file, never a final-named file that looks complete but is truncated. `Ring.Tests/SqlDumpServiceTests.cs` covers it, including the interrupted-before-rename case.

This is a **separate guarantee** from the live database's own durability: `RingwoodDbAccess.ApplyConcurrencyPragmas` (§5.1) already configures every SQLite connection in the app with `journal_mode = WAL` and `synchronous = FULL` — fsync on every commit, stronger than the `NORMAL` floor typically recommended for WAL — which is what protects `RingwoodDatabase.db` itself against corruption or a lost commit on a power outage. `SqlDumpService` inherits that same protection for its own connection, but the two exist for different failure modes: the pragmas protect the live file continuously; the `.sql` export is a one-shot, portable snapshot an operator or IT can hand off or replay into a brand-new database. `Ring.Tests/RingwoodDbAccessTests.cs` is regression coverage added alongside this work for the pragma configuration, which previously had none.

Reports. `Ring/Services/Reports/` holds the query services (read-only aggregations over persisted data), the pure calculators, and the `*ReportDocument` builders that produce WPF `FlowDocument`s for on-screen preview and `PrintDialog` output. `PdfExportService` renders a `FlowDocument` to PDF. `ReportSchedulerService` (1-minute timer) drives subscription emails via `ScheduledReportRenderer`, with due-math in the pure `ReportScheduleCalculator`. Chapter 07 catalogues them.

Diagnostics. `Ring/Services/DiagnosticBundleService.cs` builds a one-click support zip.

Plant profile. `Ring/Services/PlantProfileService.cs` builds a versioned `*.ringprofile.json` commissioning export. Note from `docs/CONFIG_AND_STARTUP.md` §3 that a plant-profile *import* deliberately does **not** carry the PLC IP or `ReadOnlyMode`.

5.10 Adding a table — the whole checklist

Reproduced from `ARCHITECTURE.md` because it is the one place a partial change silently succeeds:

1. Write `IXRepository` + `XRepository` with the per-call connection + `ApplyConcurrencyPragmas` shape.
2. Expose `public static void ApplySchema(SQLiteConnection)`; have `EnsureTable()` call it.
3. Append a `MigrationStep` at the **end** of `GetMigrationSteps()` that calls `XRepository.ApplySchema`.
4. Bump `CurrentSchemaVersion` (`DatabaseInitializer.cs:743`). Three tests fail if you forget in the ordinary case — `DatabaseSchemaMigrationTests.CurrentSchemaVersion_Matches_HighestMigrationStep` (`:47`), `DatabaseExpectedTablesDriftTests.CurrentSchemaVersion_MatchesInitializerLatest` (`:120`), and

`DatabaseSchemaMigrationTests.Wave6Migrations_CreateBothTables_AndStampCurrent`, which asserts the version as a **literal** (`:139-140`) and therefore has to be edited too. But see the same-version gap above, which **no test catches**.

5. Add the table name to `ExpectedTables` in `DatabaseBackupService` (`:43`). `DatabaseExpectedTablesDriftTests.ExpectedTablesExactlyMatchesWhatInitializeActuallyCreates` (`:93`) fails in **both** directions if this and step 3 disagree. You do not normally touch `CoreTables` (`:144`).
6. Mirror it in the `AllTables` literal in `Ring.Tests/DatabaseBackupServiceTests.cs:38`.
7. Register the repository in `ServiceCollectionExtensions`.
8. Register both new `.cs` files in `Ring/Ring.csproj` (chapter 08).
9. Have the repository's catch blocks call `RepositoryReadScope.ReportFailure` (§5.2) so a failed read is distinguishable from an empty result.
10. If a background timer will write the table, add it to `DatabaseWriterQuiesce` (§5.5).

The step-by-step version, with the reasoning inline, is chapter 10 §10.5.

Next: 06 — UI Architecture.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring/Database/` — `DatabaseInitializer.cs`, `RingwoodDbAccess.cs`, `RepositoryReadScope.cs`, `Interfaces/` and `Repositories/` (listings), `Repositories/BatchRepository.cs` · `Ring/Services/DatabaseBackupService.cs`, `DatabaseHealthService.cs`, `DatabaseWriterQuiesce.cs`, `DatabaseMaintenanceLatch.cs`, `ProcessHistorianService.cs`, `RuntimeAccumulatorService.cs`, `DiagnosticBundleService.cs`, `PlantProfileService.cs` · `Ring/Services/Audit/UiAuditJournal.cs`, `IncidentExportBuilder.cs` · `Ring/Services/Export/DatabaseExportSchema.cs`, `DatabaseExportService.cs`, `SqlDumpService.cs` · `Ring/Services/Batch/` (listing) · `Ring/Services/PLC/PlcPollingCoordinator.cs` · `Ring.Tests/DatabaseExpectedTablesDriftTests.cs`, `DatabaseBackupServiceTests.cs`, `DatabaseSchemaMigrationTests.cs`, `SqlDumpServiceTests.cs`, `RingwoodDbAccessTests.cs` · `docs/CONFIG_AND_STARTUP.md` · `ARCHITECTURE.md`

06 — UI Architecture

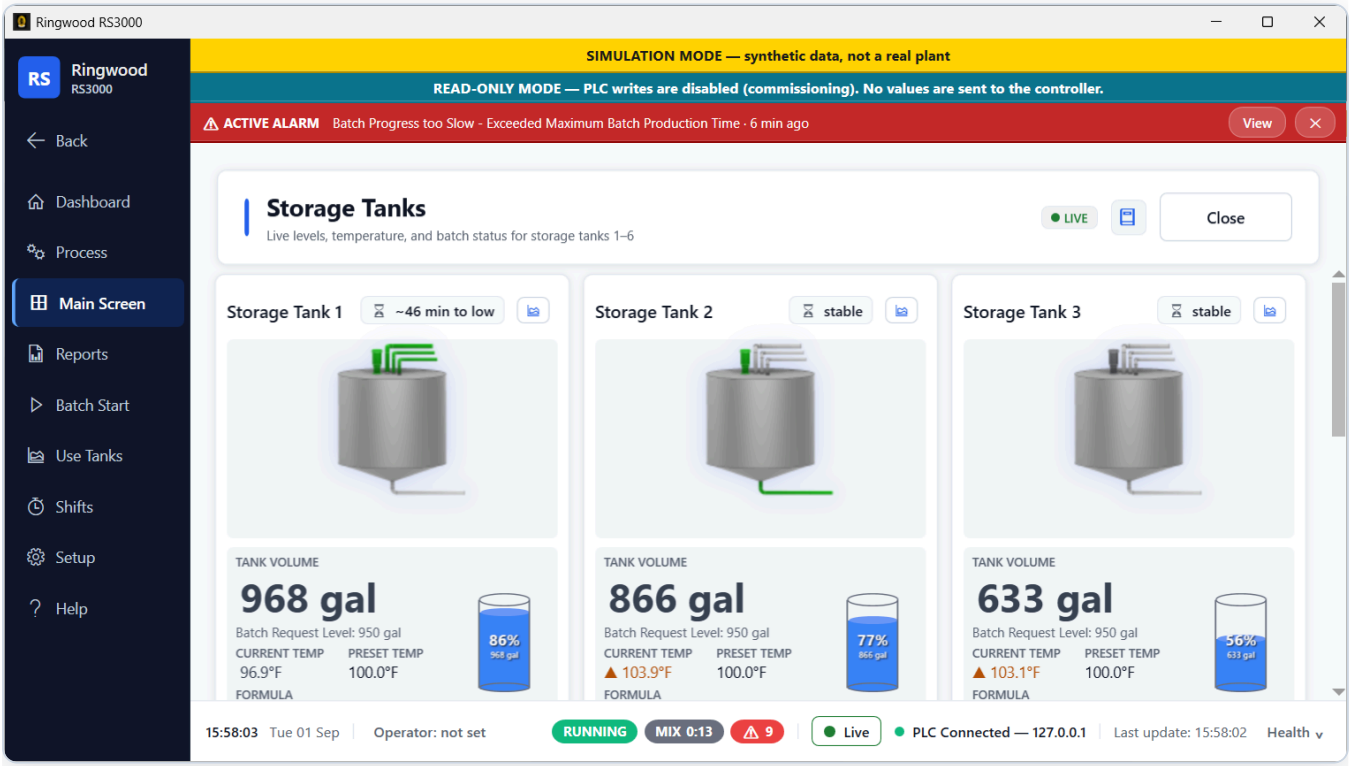
Who this is for. Anyone adding or changing a screen, a control, a theme token, or a localized string.

What you'll learn. The two-tier UI reality and which tier your new screen belongs in; how the shell, the hosted-screen pattern and `FitHost` work; what lives in each screen area; the resource-dictionary system and the thirteen-locale rule; converters; touch and legibility conventions; and the one UI rule that is a *safety* rule.

6.1 Ring is not uniformly MVVM, and pretending otherwise will mislead you

`ARCHITECTURE.md` is the authority here and describes the split honestly. In summary:

Tier 1 — VM-driven. These set `DataContext = _viewModel` in code-behind and bind. They are the process-data screens and the best-tested UI code in the repo: `MakeReadyTank`, `CausticAndBorax` (which shares `MakeReadyTankViewModel`), `StorageTankGroup`, `StorageTankDetailWindow`, `UseTankGroup`, `TVC`, `Alarm`, and the Use Tank windows.



Storage Tank Group — one of the tier-1, ViewModel-driven process screens named above.

> **Three Use Tank windows.** `Ring/Views/UseTanks/` contains `MF1Window`, `UseTank2Window` and `UseTank3Window` only. A fourth, `UseTank1Window`, was an unreachable clone of `MF1Window` and was deleted along with its write-surface register row; the only surviving references to it in the tree are two historical comments in `Ring.Tests/WriteSurfaceRegisterTests.cs` (`:495`, `:944`).

Tier 2 — code-behind (everything else). The canonical shape: the constructor calls `InitializeComponent()` and defers to a `Loaded` handler; `Loaded` pulls concrete dependencies out of `ServiceLocator.GetService<T>()` inside a `try`/`catch`, queries a repository, and assigns `SomeGrid.ItemsSource` imperatively. PLC-facing screens add a `DispatcherTimer` that reads a static snapshot and pushes values into named controls. There is no design-time `d:DataContext` anywhere in the tree.

Two shapes sit between the tiers, both deliberate: `InsightsScreen` (`Ring/Views/Reports/InsightsScreen.xaml.cs`) assigns card ViewModels to individual child elements rather than to the screen, and `DashboardView` (`Ring/Views/Dashboard/DashboardView.xaml.cs`) binds to a *service* (`DashboardService`), which its XAML calls out explicitly.

Which should new code use? For a new process screen with live PLC data, use tier 1 — a ViewModel with `INotifyPropertyChanged` and real bindings. For a new report or setup screen, matching the surrounding tier-2 code is acceptable and often clearer than a lone MVVM island. **What is not acceptable is a third pattern.**

6.1a The ViewModel tier, catalogued

`Ring/ViewModels/` holds 38 files. Three of them (`MakeReadyTankViewModel`, `StorageTankGroupViewModel`, `AlarmViewModel`) are named elsewhere in this book; the other 35 are not. This section closes that gap with the same grouped-table style chapter 07 uses for Services.

The INPC convention. There is **no** shared `ViewModelBase` or `ObservableObject` base class anywhere in `Ring/`. Every ViewModel implements `INotifyPropertyChanged` directly and re-declares its own `PropertyChanged` event plus a private `SetProperty<T>` or `OnPropertyChanged` helper — the same few lines, copy-pasted per class rather than inherited. If you add a new ViewModel, copy this shape from an existing one; introducing a shared base class would be a real improvement but is not the pattern today.

How they get attached — two different patterns, not one. Process-screen ViewModels (bucket **a** below) are DI **singletons**, registered in `AddRingServices` and resolved with `ServiceLocator.GetRequiredService<T>()` in the View's constructor — e.g. `Ring/Views/MainScreen/StorageTankGroup.xaml.cs:27-28`. Dashboard/Insights **card** ViewModels (bucket **c**) are instead `new`'d directly by `InsightsScreen.xaml.cs`'s code-behind, one per `View` instance, each falling back to `ServiceLocator.GetService<T>()` **inside its own constructor** for any repository or query service it needs (`InsightsScreen.xaml.cs:171-196`) — a hybrid that is neither full DI nor a plain

object graph. Small per-item helper ViewModels like `TankCardViewModel` are simply `new`'d by the group ViewModel that owns them, one per roster slot (`StorageTankGroupViewModel.cs:258`), and never touch DI at all.

(a) Process-screen ViewModels — `ViewModels/MainScreen/` (15 files)

File	Responsibility
`MakeReadyTankViewModel.cs`	The Make Ready Tank (mixer) hero: current batch step, ingredient, agitator state
`MrtImageResolver.cs`	Pure: PLC batch op-code + agitator bit → MRT vessel image filename
`ProcessFlowMap.cs`	Pure: which pipe/connector carries product for a given batch operation code
`ProcessOverviewViewModel.cs`	The whole-plant "Live Process" screen; composes the existing singletons plus its own `TVCViewModel`; read-only, no PLC access itself
`StorageTankDetailViewModel.cs`	Read-only detail for one generic storage-tank slot with no dedicated legacy window
`StorageTankGroupViewModel.cs`	The Storage Tank Group screen — up to 4(+) columns of tank state, global system status
`StorageTankImageResolver.cs`	Pure: three activity bits → storage-tank glyph filename
`TVCViewModel.cs`	The TVC screen: overall TVC tank plus a collection of per-storage-tank cards
`TankCardViewModel.cs`	Per-tank card state for the Storage Tank Group screen, one per roster storage slot
`TankTemperatureClassifier.cs`	Pure: actual-vs-preset temperature → Normal/Warning/Alarm, shared by `TVCViewModel` and `StorageTankGroupViewModel`
`TvcStorageTankCardViewModel.cs`	Per-storage-tank card for the TVC group screen
`UseTankCardViewModel.cs`	Per-tank state for one Use Tank (doser) card
`UseTankComponentRow.cs`	Row model for a use-tank card's "Sensors, Valves, and Pumps" popup
`UseTankGroupViewModel.cs`	The Use Tank Group screen; composes `UseTankCardViewModel`s, one per roster doser slot
`UseTankImageResolver.cs`	Pure: five activity flags → use-tank vessel image filename

Plus **`ViewModels/UseTanks/PerUseTankWindowViewModel.cs`** — the base for the per-tank drill-down windows (`MF1Window`, `UseTank2/3Window`).

(b) Shell and chrome ViewModels — flat in `ViewModels/` (6 files)

File	Responsibility
`PlcHealthDrawerViewModel.cs`	Singleton diagnostics observer for the PLC Health drawer
`PlcConnectionStatusViewModel.cs`	App-wide singleton PLC connection status
`DemoModeBannerViewModel.cs`	Singleton banner for demo/training mode
`ClockViewModel.cs`	Singleton wall-clock for the bottom status strip
`BatchStatusStripViewModel.cs`	Singleton app-wide batch/alarm status for the status-strip pills and `MainWindow` banner
`AlarmViewModel.cs`	Alarm grid — PLC cache in one phase, SQLite event log and silence latch in the other

(c) Dashboard/Insights card ViewModels — flat in `ViewModels/` (10 `*CardViewModel` files)

There is no `ViewModels/Dashboard` or `ViewModels/Insights` folder — despite the name, every card ViewModel lives flat alongside the shell ones above. `InsightsScreen.xaml.cs` assigns each to one child element's `DataContext` (the pattern §6.1 already names in passing):

File	Responsibility
`AlarmLoadCardViewModel.cs`	Insights "Alarm Load" — worst-first ranked catalog alarms for the week
`BatchEtaCardViewModel.cs`	Predicts running-batch completion time from historical per-step medians
`CookTemperatureCardViewModel.cs`	Per-tank mean/spread cook-temperature rows for the day
`GlueLineRunDryCardViewModel.cs`	Projects when the mixed-glue inventory feeding the corrugator runs dry
`GlueUsedCardViewModel.cs`	Delivered glue volume plus derived dry-starch pounds over a date range
`LastBatchCardViewModel.cs`	Small "last batch" tile (tank/formula/time/volume), populated externally
`SpendCardViewModel.cs`	Ingredient cost, giveaway money, aborted write-off for the week
`StockForecastCardViewModel.cs`	Worst ingredient slots by remaining runway to empty
`YieldAggregateCardViewModel.cs`	Windowed yield/giveaway aggregate, broken into per-formula or per-tank rows
`YieldScorecardCardViewModel.cs`	Grades the most-recent completed batch's yield%/giveaway%/cycle time

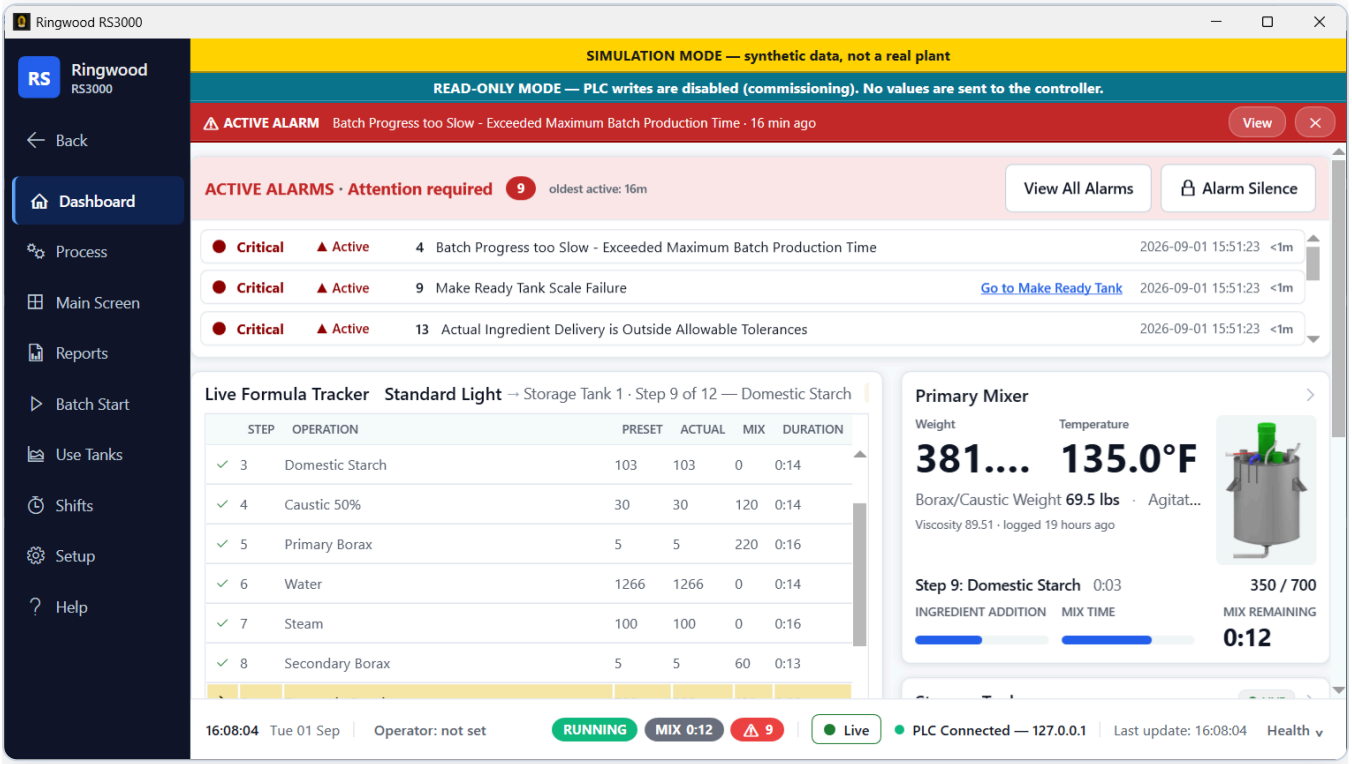
(d) Everything else — flat in `ViewModels/` (6 files)

Cards in spirit but not in name, plus two dashboard-only pieces:

File	Responsibility
`DashboardGraphViewModel.cs`	Dashboard/Insights "Trends" chart — series built from completed-batch SQLite rows
`DashboardKpiStripViewModel.cs`	The dashboard's 4-tile KPI strip (batches this shift, dry starch, soonest tank low, batches queued)
`EmailHealthViewModel.cs`	Dashboard "Email Health" card (SMTP configured?, last send, success/fail counts)
`EquipmentHealthViewModel.cs`	Dashboard "Equipment health" card; a 3-state enum avoids a false all-clear green
`IdleFormulaStepsViewModel.cs`	Idle-dashboard formula-steps panel — the last completed batch's recipe when nothing is running
`TankAttentionViewModel.cs`	Singleton, read-only: ranks the 1–3 tanks most needing operator attention, composed from the `StorageTankGroupViewModel/UseTankGroupViewModel` singletons

That is 15 + 1 + 6 + 10 + 6 = 38, matching the directory listing exactly.

6.2 The shell



The shell: NavBar down the left, everything else is MainContentArea hosting one screen — here, the dashboard.

`MainWindow`

`Ring/Views/MainWindow.xaml(.cs)` owns the window chrome, the shared `MainContentArea ContentControl`, the nav history stack, the PLC watchdog tick, the alarm pump, the auto-reprobe and the UI-hang sentinel (cadences in chapter 02 §2.5).

A rename-freeze item lives here. `MainWindow.NavigateBack()` branches on

```
t.Namespace != null && t.Namespace.StartsWith("Ring.Views.BatchStart", StringComparison.Ordinal)
```

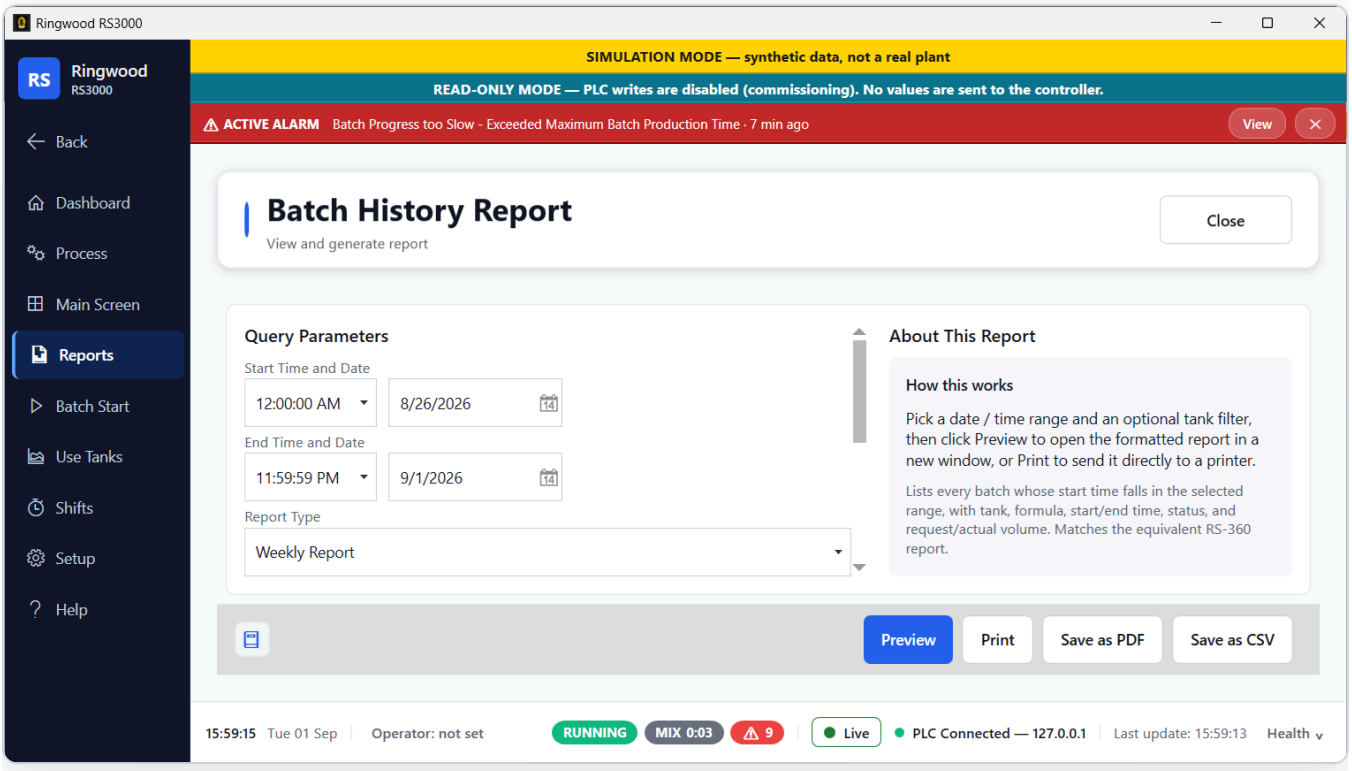
(`Ring/Views/MainWindow.xaml.cs:640`) to decide whether Back re-creates a screen or reuses the cached instance. The four Batch Start screens assume a first-time `Load`. Rename that namespace and Back silently switches them to instance reuse — a behaviour change that compiles cleanly. See `ARCHITECTURE.md`.

`NavBar`

`Ring/Views/UserControls/NavBar.xaml(.cs)` is the single menu surface. Two structural facts:

- **Views are cached singletons.** `GetOrCreateView<T>()` (`NavBar.xaml.cs:89`) returns a retained instance. This is why `SetpointEchoGuard` state survives navigate-away and why every setpoint window must `Reset()` and re-seed in `Loaded` (chapter 04 §4.8).
- **Several menus are built from the tank roster, not from literals.** `PopulateRosterTankMenus()` runs after `InitializeComponent` and builds buttons in code (`NavBar.xaml.cs:124-130`).

The hosted-screen pattern



A hosted screen: `ReportHostView` paints the header and the back/close affordance shown here; the report itself is only the `ReportContent` inside it. Contrast with the dashboard above, which is not hosted.

Reports and most setup screens do **not** go into `MainContentArea` directly. They are placed inside a `ReportHostView` (`Ring/Views/Reports/ReportHostView.xaml(.cs)` — note it lives under `Views/Reports/`, not `Views/UserControls/`), which paints the shared header and the back/close affordance and hosts the real screen in its `ReportContent` dependency property (`:24-25`).

Navigation goes through the statics on `ReportHostView`:

Static	Use
`ShowReportInMainContent(context, title, reportContent)`	`:212`
`ShowHostedInMainContent(context, title, subtitle, hostedContent)`	`:218`
`ShowHostedLocalized(context, titleKey, subtitleKey, hostedContent)`	`:241`
`ShowHostedLocalizedSubtitle(context, title, subtitleKey, hostedContent)`	`:262`
`NavigateMainContentToDashboard(context)`	`:200`

The localized variants bind the header via `SetResourceReference` (`:111`, `:118`), so the host header re-localizes live on a language switch rather than freezing the string it was given.

> **Authoring hazard: the double header.** A hosted screen that paints its own title row produces two headers. Twenty-three files under `Ring/Views/` carry a comment recording that their banner was deleted for exactly this reason (verified by grep). **Nothing enforces it** — if you add a hosted screen, do not give it a title row.

`FitHost`

`Ring/Views/UserControls/FitHost.cs` is a `Decorator` that hosts a single child, measures it at the **full available width but unbounded height** — so its internal star/proportional layout resolves exactly as it would unscaled, unlike a `Viewbox`, which measures at infinite width and collapses star columns — and then uniformly scales it **down** to fit the viewport. Scale is never `> 1`; content is never blown up. It exists so fixed-layout screens survive a plant display at 1080p/125% DPI without scrolling or clipping.

The rule that matters, from its own doc (`:19-21`): *"Use ONLY on structural screens (cards, mimics, forms). Do NOT wrap a virtualized `DataGrid`: measuring at unbounded height would realize every row. Data-table screens keep inner scroll."*

`ChildVerticalContentAlignment` (`:38-50`) is an opt-in: the default `Top` reproduces the historical arrange byte-for-byte, `Stretch` lets a screen fill the dead vertical gutter, `Center` centres it — and **in the downscaled regime all values fall back to the exact anti-clip top-anchored arrange**, so clipping protection is never affected. The pure geometry lives in the static, testable `FitHost.ComputeChildArrange`.

6.3 The screen areas

Directory	Contains
`Views/MainScreen/`	The process screens: `MakeReadyTank`, `CausticAndBorax`, `StorageTankGroup`, `StorageTankDetailWindow`, `UseTankGroup`, `TVC`, `ProcessOverview`, `Silo`, `BulkCausticTank`, `ReceivingHopper`, `ReceivingVerificationScreen`, `Viscometer`, `DataEntryHub` / `DataEntryForm`
`Views/Dashboard/`	`DashboardView` — the KPI/at-a-glance screen, bound to `DashboardService`
`Views/Process/`	`Alarm` — the alarm screen; its `AlarmSilence_Click` reaches `AlarmViewModel.TryWriteAlarmSilence` and is a registered write-surface dispatcher
`Views/BatchStart/`	The four arming screens (`StorageTank1/2/4`, `Lowmidtank`) **plus** their pure policy types: `BatchStartEligibility`, `BatchSizePrecondition`, `FormulaVolumePrecondition`, `BatchStartLevelRange`, `BatchStartLevelUnits`, `SplitTankMapping`, `PendingBatchConfig`, `BatchConfirmationSummary`, `BatchStartStrings`

Directory	Contains
`Views/TVCcontrol/`	The four per-tank TVC windows plus `TVCWindowPlcBridge` (the queue dispatcher), `TvcAgitatorCycleController`, `TvcSetpointSeedPlan` / `TvcSetpointSeeder`, `TvcWindowOptionModel`, `TvcWindowRowGate`, `TvcPresetDisplay`
`Views/UseTanks/`	`MF1Window`, `UseTank2Window`, `UseTank3Window` plus `UseTankWindowPlcBridge`, `UseTankSetpointSeedPlan` / `UseTankSetpointSeeder`
`Views/Setup/`	~40 supervisor/admin screens and dialogs: formula edit and exchange, tank roster, group hardware, tank/group names, inventory edit, costs, analytics, plant profile, shifts, language, units, credential rotation, database backup, factory reset, database export, diagnostic console, `DisplayCommunicationForm`, `TagInspector`, `TimeDateForm`, maintenance, scale calibration, legacy import
`Views/Reports/`	~27 report screens plus `ReportHostView`, the CSV builders and the pure row-projection types
`Views/Shifts/`	`ShiftControl` (note: its `x:Class` is `Ring.Views.Shifts.ShiftControlView`), `ShiftHandoffScreen`, `ShiftHandoverScreen`, `ShiftProductionLogScreen`
`Views/Help/`	`ManualsLibraryScreen`, `AlarmLegendScreen`, `ContactUs`
`Views/Wizard/`	`StartupWizardWindow` and its steps, including `ReadOnlySafetyStep` and `FinishStep`
`Views/UserControls/`	`NavBar`, `FitHost`, `ScreenHeader`, `ScreenHelpButton`, `ToastHost`, `PlcHealthDrawer`, `LinkStateBadge`, `HealthChip`, `AlarmTriageBadge`, `PlaybookPanel`, the mini charts and live vessel thumbs
`Controls/`	`NumericKeypad` + `KeypadBehavior` (touch numeric entry), `TankLevelControl`

Nine of these screen files are **`UiDispatch` rows on the write-surface register** — meaning the register asserts an operator gesture reaches a writer through them, and `Every_UI_dispatch_screen_on_the_write_surface_can_actually_be_reached` proves each is still constructible. See chapter 04 §4.10.

6.4 Resource dictionaries and theming

`Ring/Views/App.xaml` merges, **in this order**:

1. MahApps.Metro `Controls.xaml`, `Fonts.xaml`, `Themes/Light.Blue.xaml`

2. `Resources/Colors.xaml` → `Resources/Tokens.xaml` → `Resources/Components.xaml` — "*MUST stay in this order, because Tokens references Colors brushes and Components references both*" (`App.xaml:17-19`)
3. `Resources/RingTheme.xaml` — the Wave-2 semantic `Ring*` token system, merged **last** among the theme dictionaries so screens migrate to `Ring*` keys incrementally without breaking Wave-1 consumers
4. `Resources/Strings/Strings.en.xaml` — **English last of all**

Two token systems coexist deliberately. Values are converged where it matters: the legacy `PrimaryColor` is `#2563EB`, the same brand blue as `Ring.Color.Accent`, "so legacy-styled and Ring-styled buttons paint the SAME brand blue" (`App.xaml:47-50`).

Touch and legibility conventions

From `Ring/Resources/Tokens.xaml`:

Token	Value	Note
`MinTouchTarget`	`44`	explicitly labelled "Touch target minimum (accessibility)" (`:23-24`)
`Font.Caption` / `Font.Body` / `Font.Value`	13 / 14 / 16	
`Font.ValueCritical`	20	the size for a number an operator must read across a bay
`Font.H3` / `H2` / `H1`	16 / 20 / 28	
`Radius.Control` / `Radius.Card`	6 / 8	
`Space.S` / `Space.M` / `Space.L`	8 / 12 / 12	

Type-scale `TextBlock` styles (`Heading1Text` and siblings) are defined on top of those sizes, so a new screen should reach for a style, not a literal `FontSize`.

Touch entry itself is served by `Ring/Controls/NumericKeypad.xaml` + `KeypadBehavior.cs`, which is the sanctioned way to take an operator number on a touchscreen — and `Ring/Services/Display/OperatorNumberInput.cs` / `LocalizedInput.cs` are the locale-tolerant parsers behind it.

6.5 Localization — the thirteen-dictionary rule

Thirteen locale dictionaries under `Ring/Resources/Strings/` (verified by listing): `en`, `nl`, `de`, `fr`, `es`, `it`, `pt`, `tr`, `pl`, `ko`, `ja`, `zh-Hans`, `zh-Hant`.

The rules

1. **A new user-visible string goes into all thirteen dictionaries, in one commit.** A per-locale parity test fails otherwise (`Ring.Tests/LocalizationServiceTests.cs`, `Satellite_IsFullParityMirror_OfEnglish(string code)`, `:239-241`), and it checks **placeholder parity** (`{0}`/`{1}`) as well as key parity so `string.Format` stays safe (`:262`).
2. **Never create a duplicate `x:Key`.** WPF throws while loading the dictionary — which happens **before the first frame** — so the app dies at startup with no window and a bare crash log.
3. **English is merged last and stays last.** `Strings.en.xaml` is the base and the permanent fallback, so a key missing from an overlay still resolves rather than rendering blank. Do not change the merge order in `App.xaml`; the file says so in a comment (`App.xaml:30-38`).
4. **Every key referenced from XAML must exist in the English dictionary** — `ReferencedLocalizationKeys_AllExistInEnglishDictionary` (`:445-446`).
5. **Watch for mojibake.** Dictionary values have shipped double-encoded (a cp1252 round-trip) before, and neither the build nor the rest of the suite caught it, because the values are still valid UTF-8 and still parity-complete. A dedicated test scans for the signature bigrams (`Dictionary_HasNoMojibake`, `:329-331`). If it fires, **do not "fix" it by retyping the character** — repair the encoding (read the bytes back through the cp1252 round-trip) and save UTF-8 without BOM.

How switching works

`Ring/Services/Display/LocalizationService.cs`:

- `SupportedLanguages` (`:58`) is the list of `LanguageOptions` — code, English name, endonym, overlay URI. `SupportedLanguages_AreThirteen_WithUniqueCodes` pins the count (`LocalizationServiceTests.cs:388`).
- `SetLanguage(code)` (`:127`) swaps **one** overlay dictionary in `Application.Current.Resources.MergedDictionaries`, deliberately excluding the English base from removal (`:225-232`). Every `DynamicResource` binding repaints live; **no restart**.
- `ApplySavedLanguage()` (`:106`) reads the persisted choice from the `AppState` table at startup (phase 7).
- A `LanguageChanged` event (`:86`) exists for strings composed in C#, which XAML cannot repaint.
- **`LocalizationService` deliberately never touches `CurrentCulture` / `CurrentUICulture`** (`:23`). Display language and number formatting are kept separate on purpose: batch math and PLC value formatting must not change because an operator picked Dutch. This is the same concern that makes `PlcTagWriter.TryParseRealInvariant` invariant-only (chapter 04).

The known gap

Alarm **operator text** lives in SQLite in English only. That is a schema change, not a dictionary edit, and is on the deferred list (`ARCHITECTURE.md`). So is the fact that C#-composed text does not live-repaint on a language change — hence the `LanguageChanged` event.

For code that must resolve a localized string from a non-XAML context, the house pattern is a `LocalizedOrFallback(key, englishFallback)` helper that reads `Application.Current?.TryFindResource(key)` and falls back to the English literal, with the whole thing wrapped so a test host with no `Application` gets the literal — see `ConfigurationValidator.cs:517-524` and the equivalents in `DashboardService` and `PcWriteInteger30StrandedBitMonitor`. The commissioning holds follow the same convention: each exposes a `…ResourceKey` constant and a `…Fallback` string side by side (e.g. `TankTempModeWriteAuthorization.NotAuthorizedResourceKey` / `…Fallback`).

6.6 Converters

`Ring/Converters/` holds thirteen value converters. The freshness family is the one worth knowing about, because it is how a screen refuses to lie:

Converter	Purpose
`FreshnessConverterHelper`, `FreshnessToCaptionConverter`, `FreshnessToOpacityConverter`	render the `DataFreshnessClassifier` verdict as caption text and dimming
`PlcConnectionStateToBrushConverter`	link state → colour
`StateToBrushConverter`, `StateToGlyphConverter`, `ValueToHealthConverter`, `VarianceToBrushConverter`, `AlarmSeverityDisplayConverter`	status painting
`BooleanToOnOffConverter`, `EmptyToPlaceholderConverter`, `ProgressBarValueConverter`, `ReceivingTankVisibilityConverter`	general formatting

Numeric display itself is **not** a converter concern: `Ring/Services/Display/PlcDisplayFormatter.cs` is the single source of truth for PLC-value formatting and `UnitDisplay.cs` is the unit-aware layer around it. `DisplayUnitService` owns the station's chosen display units and is **display only** — it never changes how a value is stored or written.

6.7 The UI rule that is a safety rule

Never put a UI guard or an early `return` in front of a shared handler that also dispatches PLC-write buttons.

This is in `CLAUDE.md` and `CONTRIBUTING.md`, and the live implementation is `NavBar.HandleButtonClick` (`Ring/Views/UserControls/NavBar.xaml.cs:415-444`).

Most buttons in that handler swap `MainContentArea.Content`, and before such a swap the handler honours the hosted setup editor's unsaved-changes guard — otherwise a NavBar click silently discards a dirty editor's edits. But not every button navigates. The handler therefore carries **two explicit exclusion sets**:

```
bool isPopupToggle =                        // the nine top-level menu buttons
    buttonName == "ProcessButton" || ... || buttonName == "HelpButton";

bool isNonNavAction =
    buttonName == "HoldButton" || buttonName == "ProcessResumeButton" ||
    buttonName == "ProcessResetButton" || buttonName == "StartPlcMonitoringButton" ||
    buttonName == "LockSetupButton" || buttonName == "CommunicationDiskLoggingButton" ||
    buttonName == "UpdateProcessorTimeDateButton";

if (!isPopupToggle && !isNonNavAction && !ConfirmLeaveHostedContent())
    return;   // operator chose to keep editing — abort the navigation
```

The comment at `:426-429` states the hazard in one sentence: those buttons "must NEVER run the guard — it would pop a misleading 'discard?' prompt and, if the operator keeps editing, **ABORT a process command (e.g. a live Hold)**."

The same reasoning explains why the roster-generated per-tank menu buttons get a **dedicated** click handler rather than being routed through the shared one (`NavBar.xaml.cs:124-129`): so tank navigation never travels through the handler that fronts the Hold / Resume / Reset write buttons.

Before you add any guard to a shared handler, enumerate every button routed to it. Adding a confirmation to a handler you believe is "just navigation" is the exact shape of this defect.

A related, milder rule from `MainWindow`: the roster restart lock deliberately gates **polling (reads) only** — it "is not in front of any PLC-write button" — which is precisely why `RosterWriteIndex` needs its own write-side locks (chapter 04 §4.3).

6.8 UI debt, deliberately deferred

None of this is on fire; all of it is parked past the cutover (`ARCHITECTURE.md`):

- Per-tank screen family consolidation (UseTanks → TVC → BatchStart), with three ordered prerequisites: a composite cache key, behavioural tests on WPF-free policy objects, and a controls decision on split options.
- `StorageTankGroupViewModel` carries a half-finished migration — five flat property consumers still need repointing at `TankCards`.
- `DashboardView` is hardcoded around the legacy tank count; six-tank support wants an `ItemsControl`.
- `ServiceLocator` retirement and ambient-state → DI conversion.

Next: 07 — Services Catalog.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring/Views/App.xaml` · `Ring/Views/MainWindow.xaml.cs` · `Ring/Views/UserControls/NavBar.xaml.cs` · `Ring/Views/UserControls/FitHost.cs` · `Ring/Views/Reports/ReportHostView.xaml.cs` · `Ring/Views/` (full directory listing, all areas) · `Ring/ViewModels/` (full directory listing) · `Ring/ViewModels/MainScreen/StorageTankGroupViewModel.cs`, `TankCardViewModel.cs` · `Ring/ViewModels/LastBatchCardViewModel.cs`, `SpendCardViewModel.cs`, `BatchEtaCardViewModel.cs`, `ClockViewModel.cs` · `Ring/Views/Reports/InsightsScreen.xaml.cs` · `Ring/Views/MainScreen/StorageTankGroup.xaml.cs` · `Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs` · `Ring/Resources/Tokens.xaml`, `Colors.xaml`, `Components.xaml`, `RingTheme.xaml`, `Strings/` (listing) · `Ring/Converters/` (listing) · `Ring/Controls/` (listing) · `Ring/Services/Display/LocalizationService.cs` · `Ring/Infrastructure/Configuration/ConfigurationValidator.cs` · `Ring.Tests/LocalizationServiceTests.cs` · `Ring.Tests/WriteSurfaceRegisterTests.cs` · `ARCHITECTURE.md` · `CONTRIBUTING.md`

07 — Services Catalog

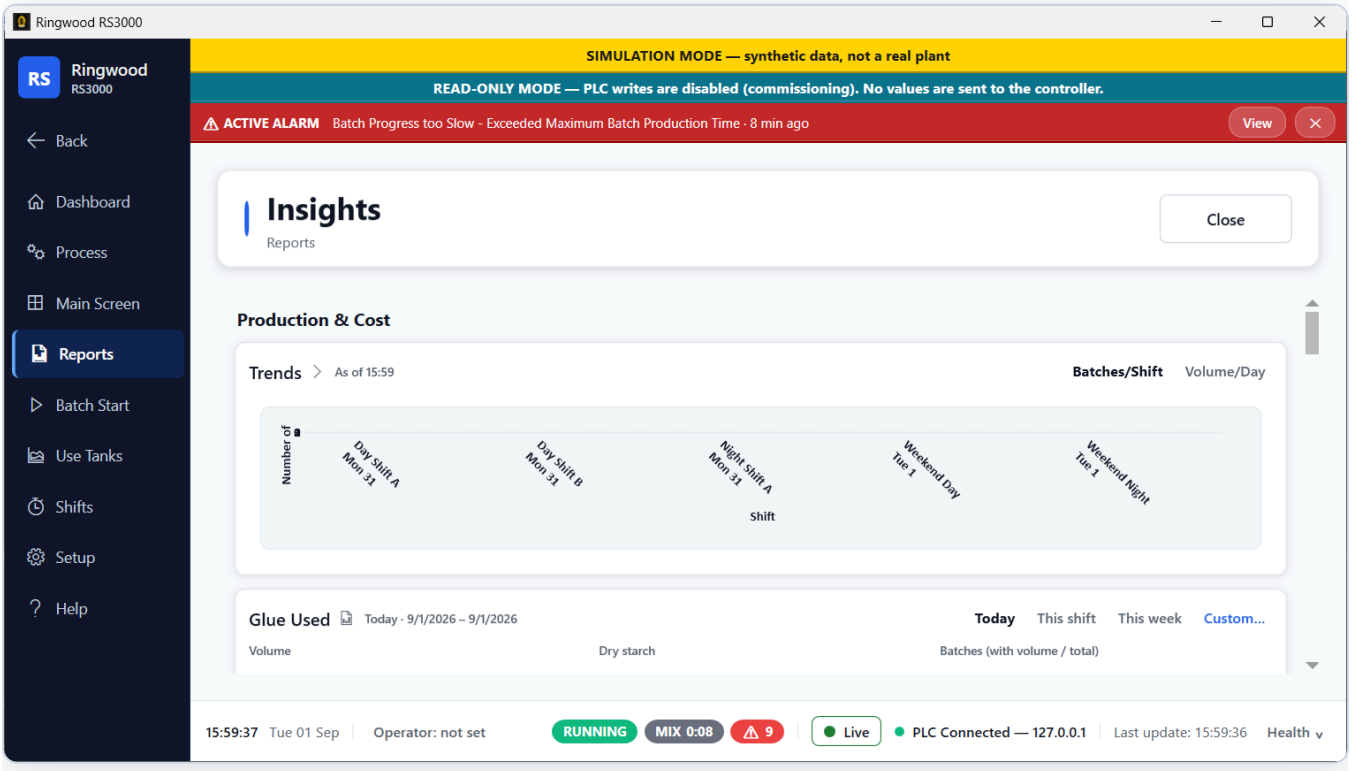
Who this is for. Anyone looking for "where does X live?" or "is there already something that does this?" before writing a new service.

What you'll learn. A guided index to everything under `Ring/Services/` — each entry a short, verified statement of responsibility and its key collaborators, with its file path. Entries are grouped by concern rather than by directory where that reads better; the directory is always given.

How to read this. Each summary was taken from the type's own doc comment in the cited file. Where a service's own comment states a limit, a rationale or a warning, this catalog carries it — those are the parts that stop you reintroducing a fixed bug. Where a service is large enough to need its own explanation, this chapter points at the chapter that gives it.

Coverage. Every `.cs` file directly under `Ring/Services/` and in each of its fifteen subdirectories is named somewhere below. Interfaces and their default implementations are grouped together; small companion types (enums, records, option objects) are named with the service that owns them rather than given their own row. Nothing under `Ring/Services/` is silently omitted — if you cannot find a file here, that is a defect in this chapter. This catalog is **Services-only**; the equivalent catalog for `Ring/ViewModels/` is chapter 06 §6.1a.

The PLC services are catalogued here only briefly; chapters 03 and 04 are their real documentation.



One screen, a dozen services from this catalog: the Insights screen's cards are the dashboard/insights ViewModels of chapter 06 §6.1a, each backed by a query service or calculator named below.

7.1 PLC — `Ring/Services/PLC/`

The largest subfolder and essentially flat. Grouped by role:

Gates and write infrastructure

File	Responsibility
`PlcWriteGuard.cs`	The global read-only gate and its one-way session latch. §4.1
`PlcWriteEndpointGuard.cs`	Refuses a write whose *target* cannot be the plant controller. §4.4
`PlcHeartbeatConnectionTracker.cs`	Link state machine; `AllowPlcWrites()` / `AllowHeavyPlcPolling()`. §3.6
`PlcConnectionState.cs`	The six-state enum, with `Starved` documented as distinct from `Disconnected`
`MainTvcWriteAuthorization.cs`	Hard commissioning hold — `IsAuthorized => false`
`TankTempModeWriteAuthorization.cs`	Hard commissioning hold — empty authorized-tank list; also owns the 0/1/2 value vocabulary
`TvcCoolingWriteAuthorization.cs`	Per-tank cooling authorization `{1,2,4}`; can only ever refuse the cooling selection
`TankInstalledWriteGate.cs`	Fail-closed commissioning gate from `Tanks_c[N]` install bits; UI fails open on `Unknown`, the wire fails closed
`RosterWriteIndex.cs`	Single owner of "which physical tank does this UI slot write to?"; three sticky fail-closed locks
`FormulaBankWriteInterlock.cs`	Active-batch interlock for the destructive bank rewrite, plus the `FormulaBankWriteOutcome` vocabulary
`MomentaryPulse.cs`	The shared de-assert half of every momentary pulse: spaced retries, `STUCK TRUE`, opt-in background re-clear
`PlcSetpointWriteQueue.cs`	Per-tag write serializer with per-control coalescing. Pure dispatch; holds no safety gate
`PlcWriteTagKeys.cs`	Canonical serialization keys so two paths writing one physical tag share one queue slot
`PlcWriteResult.cs`	The unambiguous `PlcWriteStatus` outcome vocabulary (blocked ≠ simulated ≠ success)
`PlcWriteEvidenceJournal.cs`	Append-only JSONL evidence for queued writes. **Evidence only — no replay API**
`SetpointEchoGuard.cs`	"Is this a genuine operator change?" — no read-back baseline ⇒ no write
`SetpointSeedContext.cs`	Per-window state for the seed-before-writable contract. Pure, no WPF types
`SetpointRefreshCoordinator.cs`	Turns a *settled* write into a re-read, off the queue's events

File	Responsibility
`TankControlSetpointReader.cs` / `TankControlSetpoints.cs`	The one-shot read-back seam that fills those baselines

Writers

File	Writes
`PlcTagWriter.cs`	The shared write funnel; `Write` / `WriteClear`; four gates inline
`BatchStartTankPlcWriter.cs` + `BatchStartPlcWriteHelper.cs`	`Tanks[N]` batch-start quintet (`Formula_number`, `Request_level`, `Splitting_A/B`, `Start_Type` **last**); the helper is the ordering seam
`TVCControlPlcService.cs` (+ `ITVCControlPlcService.cs`)	`Tanks[N].Preset_Temp`, `TVC[N].Preset_Temp` / `Temp_mode`, `TVC_Ctrl[N].enabled`
`UseTankControlPlcWriter.cs`	`Tanks[N].Agitator_mode`, `Agt_on_pre`, `Agt_off_pre`, `Liq_add_1..3_mode`
`UseTankFormulaPlcWriter.cs`	`Tanks[slot].Formula_number` from the Use Tank windows — **gated groundwork**
`TankAgitatorPlcWriter.cs`	`Tank_IO[N].ZZZZZZZZZZTank_I_O9` bit 1 (`O_Agitat`), read-modify-write
`BatchFormulaPresetPlcWriter.cs` + `FormulaBankCommitCoordinator.cs` + `FormulaPresetCodec.cs` + `FormulaBankPresetPlcReader.cs`	The `Formula_01..06_Preset_{Operation,Amount,Mix}[0..30]` bank; the coordinator drives the commit through an `IFormulaBankPlcCommitTarget` seam; the codec is the pure array shaping
`InventoryAmountPlcWriter.cs`	`Inventory_Amount[0..13]` REAL, one bulk array write
`PcWriteInteger30BatchControlPlcService.cs`	`[30].0` Resume / `.1` Hold / `.2` Reset / `.3` Silence, under a shared word-30 lock
`ShiftControlPlcService.cs`	`PC_Write_Integer[44/45/48/49].(shift-1)` momentary pulses
`ProcessorTimeDatePlcService.cs`	Controller wall-clock words `[34..39]` + the `[30].10` commit pulse
`InventoryEventCaptureService.cs`	The `PC_Write_Integer[27].4` acknowledge — and the capture that precedes it
`TankNamePlcWriter.cs` / `TankNamePlcSyncService.cs` / `TankNamePlcCodec.cs` / `TankNamePlcReader.cs` / `TankNamePlcNameHydrator.cs` / `TankNamePlcProbeState.cs` / `TankNameMergeRules.cs`	The **provisional** `HMI_Tank_Name` / `HMI_Group_Name` STRING family. No such tags exist on the plant today; inert on three independent fail-safes

Pollers, snapshots and caches

`PlcPollingCoordinator.cs` owns seven pollers, each publishing an immutable snapshot into a static holder or cache:

Poller	Publishes into
`PlcSnapshotPoller`	`PcReadFloatSnapshot`, `MakeReadyTankPlcData`, `PcDisplayOutputsCache`
`PcReadIntegerPoller`	`PcReadIntegerCache` — a heartbeat-word tick and a full `[0..479]` block tick
`BatchStepsPoller`	`BatchStepsSnapshotHolder` (`BatchStepsSnapshot`)
`StorageTankGroupPoller`	`StorageTanksSnapshotHolder` (`StorageTanksSnapshot`)
`UseTankGroupPoller`	`UseTanksSnapshotHolder` (`UseTanksSnapshot`)
`TVCPoller`	`TVCSnapshotHolder` (`TVCSnapshot`)
`BatchStartPoller`	`BatchStartSnapshotHolder` (`BatchStartSnapshot`)

`RosterPollPlan.cs` is the pure roster→read-target translation. Cadences and gotchas: §3.2.

Readers and decoding

`PlcTagReader.cs` (+ `IPlcTagReader.cs`, `PlcDataType.cs`) is the per-tag reader; `UdtTankReader.cs` + `UdtTankLayout.cs` are the byte-offset UDT path. Per-subsystem readers: `StorageTankTagReaderService`, `UseTankTagReaderService`, `TVCTagReaderService`, `BatchTagReaderService`, `InventoryStampsPlcReader`. Index maps: `PcReadIndexes.cs`, `PcWriteIndexes.cs`. Decoders: `SystemStatusWord.cs` (the `PC_Read_Integer[9]` hold/running word and the Process-menu command policy), `BatchOperationMapper.cs` (operation code → description), `IngredientFamilyMapper.cs` (operation codes → coarse ingredient families), `BatchStamps.cs` (element indexes inside `Batch_Current_Stamps`), `AgitatorPresetBands.cs` (the single definition of the agitator preset minute bands).

Diagnostics and demo

File	Responsibility
`PlcCommunicationLogService.cs`	RCS-128: in-memory TX/RX ring buffer + optional per-day CSV, feeding the live `DisplayCommunicationForm` tail. §7.13
`PlcCommEventLogService.cs` + `PlcCommEventType.cs`	**Not** the row above, despite the near-identical name: persists PLC connection-*lifecycle* events (Connected/Disconnected/HeartbeatStalled/reprobe/session/write outcomes) to the SQL `PlcCommunicationLog` table (schema v32) for after-the-fact auditability. Bounded queue, single batched writer, `CommLogRetentionDays` default 90 d. ch. 03 §3.7
`PlcClockSkewMonitor.cs`	Read-only DST/skew sentinel
`PcWriteInteger30StrandedBitMonitor.cs`	Read-only sentinel for a stranded command bit in `PC_Write_Integer[30]`

File	Responsibility
`TagManifestService.cs`	Loads `scripts/live-tags.json`; its Live/Dead verdicts are **2024-sourced and non-authoritative**
`TagSuggestionCatalog.cs`	Tag Inspector autocomplete
`TrendingDataService.cs`	2 s Trending screen feed — builds its own readers
`ViscometerLiveRecorder.cs`	5 s; **reads snapshots only**, writes SQLite
`ProcessHoldState.cs`	Live mixer HOLD from the polled status word
`DemoModeData.cs`	Demo-mode substitute for pollers that bypass `PlcTagReader`
`DemoPlcTagReader.cs`	Demo-mode substitute seam inside `PlcTagReader`
`DemoPlcTagWriter.cs`	Demo-mode substitute seam inside `PlcTagWriter`

7.2 Alarms — `Ring/Services/Alarms/`

An alarm, controller → operator

The pieces below are each independently correct but never sequenced elsewhere in this book. This is the path a single alarm actually takes, verified against the sources:

1. **Acquisition.** Two independent 2 s timers reach the same call — `MainWindow`'s `_plcUpdateTimer` `DispatcherTimer` and a background `System.Threading.Timer` alarm pump (`PlcAlarmBackgroundPumpInterval`, `MainWindow.xaml.cs:170`) — both routed through `RequestPlcAlarmPollAndPopups()`, which throttles the actual PLC read to no more than once per `PlcAlarmPollMinGap = 8 s` (`MainWindow.xaml.cs:164`) and busy-gates the two callers so they never race. That reaches `AlarmAlarmNumberPlcService.PollAlarmsForPopupsAndCache()` (`:578-629`), which bulk-reads `Alarm_Alarm_Number` as `INT[40]` in **one** PLC read (`:162-171`), falling back to 40 individual indexed reads only if the bulk decode fails (`:211-244`) — the point being to avoid a 40× timeout storm on the common path.
2. **Decode.** `AlarmNumberDecoder.DecodeBaseAlarmNumber` (`:101-112`) strips the band offset to get the catalog number; `DecodeState` (`:114-128`) classifies the raw value as Active (1–999), a silenced fault still present (1000–1999) or a resolved, cleared fault (2000–2999).
3. **Describe.** `AlarmDescriptionBuilder.BuildDescriptionOrError` (`:49-63`) looks the catalog number up in the `AlarmDefinitions` table, substituting a computed tank number where the definition's `HasTankIndex` says to; an uncatalogued number gets a localizable "Alarm {N}" placeholder rather than a blank (`:71-75`).
4. **Timestamp, where legacy PLC date parts are used.** `AlarmLegacyDateTime.FromPlcParts` (`:27-65`) reconstructs a full `DateTime` from the PLC's month/day/hour/minute/second (it supplies no year, so the year is inferred from the PC clock with a rollover rule); `AlarmLifecycleTimeParser.TryParseDisplayToUtcIso` (`:11-44`) is the inverse, parsing a grid-displayed string back to UTC ISO for storage.

5. **Persist and time-track.** `AlarmAlarmNumberPlcService.RecordPlcAlarmEventsIfEnabled` (`:768-938`) reconciles the whole snapshot into the lifecycle repository and inserts one `PlcAlarmEvents` row per identity transition. `PlcAlarmUiTimingStore.ApplySnapshot` (`:129-200`) separately tracks each slot's raised/acknowledged/resolved times in memory — carrying a base alarm's original raise time across the PLC's own shift-insert queue moving it to a different slot — and rehydrates from persisted events at startup (`HydrateFromEvents`, `:212-258`) so a restart cannot fake-reset an alarm's displayed age.
6. **Present.** `AlarmSummaryComposer.Compose` (`:44-74`) splits rows into shelved (suppressed) versus visible and sorts both by severity then recency for the ISA-101 Alarm Summary screen — display-only, and never a factor in detection, logging or escalation (`:9-16`). `AlarmScreenLinkResolver.Resolve` (`:106-130`) maps the catalog number to a deep link into the relevant equipment screen; a plant-wide alarm deliberately resolves to `None` because no single screen owns it. `PlaybookService.GetByAlarmNumber` (`:105-147`) returns the first-response `Meaning`/`FirstCheck`/`WhoToCall` entry, optionally overridden per alarm from the database.
7. **Escalate.** A newly-raised alarm fires `IAlarmEscalator.NotifyAsync` (`AlarmAlarmNumberPlcService.cs:1090-1125`). The default is `NoopAlarmEscalator`, which only logs a "would notify" line — active until `AlarmEscalation.Mode` is switched to `"Smtp"` or `"Webhook"`. Independently of escalation, rule-driven alarm emails are gated per rule by `AlarmEmailStormControl.Register` (`:55-88`), which returns `SendSingle`, `SuppressedByCooldown` or `Digest` so a flood collapses into one email rather than one per event; a severity-aware check against `SmtpAlarmEscalator.WouldSend` stops the rule engine and the SMTP escalator from both emailing the same raised alarm.
8. **Silence — the one write in this pipeline.** Everything above is reads and local state. Silencing an alarm is an operator-initiated write, covered on the write path in chapter 04 §4.7, not repeated here.

The catalog, by role

Acquisition and decoding.

File	Responsibility
`AlarmAlarmNumberPlcService.cs`	Reads `Alarm_Alarm_Number` as `INT[40]` in one PLC read, no per-element sequential fallback on the common path
`AlarmNumberDecoder.cs`	Splits a raw value into base catalog number + band offset (Active/Silenced/Resolved)
`AlarmDescriptionBuilder.cs`	Looks the base number up in the `AlarmDefinitions` table; unknown numbers get a placeholder, never a blank
`AlarmLegacyDateTime.cs`	Reconstructs alarm/ack timestamps from the PLC's month/day/hour/minute/second fields (no year supplied)
`AlarmLifecycleTimeParser.cs`	Parses grid display strings back to UTC ISO for storage
`PlcAlarmUiTimingStore.cs`	Tracks raised/ack/silence times per slot; survives the PLC's own shift-insert queue and a Ring restart

The silence write.

File	Responsibility
`AlarmSilencePlcService.cs`	Writes the silence latch — `PC_Write_Integer[30].3` by default, with a configured legacy level tag behind `EnableCustomSilenceLatchTag`. Its outcome type distinguishes a real write from a suppression. §4.7

Presentation and triage.

File	Responsibility
`AlarmStatusDisplay.cs`	Operator-facing rendering of a lifecycle status pair
`AlarmSummaryComposer.cs`	Pure shelf-filter + priority sort for the ISA-101 summary — display-only
`AlarmScreenLinkResolver.cs`	Alarm → screen deep link; plant-wide alarms deliberately resolve to `None`
`PlaybookService.cs` + `AlarmPlaybookSeedService.cs`	Operator first-response playbooks, keyed by catalog number, DB-overridable
`AlarmLegendService.cs` + `AlarmLegendReportDocument.cs`	The bilingual wall chart from `Resources/data/alarm-legend.csv`
`AlarmHistoryReportDocument.cs`	The printable/exportable alarm history report

Notification and escalation.

File	Responsibility
`IAlarmNotifier` / `AlarmNotifier.cs`	In-cab sound + flash
`IAlarmEscalator`	The off-site escalation interface (`NotifyAsync`)
`NoopAlarmEscalator.cs`	**The default.** Logs a "would notify" line; no network call
`SmtpAlarmEscalator.cs`	Sends the alarm email; `WouldSend(severity)` is the single authority for whether it will actually send
`WebhookAlarmEscalator.cs` + `WebhookCardFormatter.cs`	Posts an alarm card to a configured webhook (Teams/Slack auto-detected); SSRF-checked via `WebhookUrlValidator`
`AlarmEmailService.cs` + `AlarmEmailComposer.cs` + `AlarmContextProvider.cs`	Composes the rule-driven alarm email; `AlarmContextProvider` stamps the running-batch context onto it
`AlarmEmailStormControl.cs`	Per-rule storm suppression — one instance per rule Id, defensively locked because the alarm hook fires from a background path

Analytics.

File	Responsibility
`AlarmAnalyticsCalculator.cs`	Pure, over a window of lifecycle rows
`AlarmAnalyticsQueryService.cs`	The thin read-only wrapper that supplies it

7.3 Audit — `Ring/Services/Audit/`

File	Responsibility
`UiAuditJournal.cs`	The hash-chained, append-only operator trail plus its tail anchor and `VerifyChain`. Tamper-**evident**, not tamper-proof (unkeyed SHA-256, stated in its own doc). §5.8
`UiActionAuditor.cs`	App-wide, **observe-only** capture of operator interactions into the journal
`UiActionRecord.cs`	The record shape and the `UiActionKinds` vocabulary
`UiAuditQueryService.cs`	Pure projection over the journal: parse lines, keep the readable ones, apply the operator's filters. No WPF, no clock, no I/O beyond the files it is handed — so the screen's filter semantics are provable in a unit test
`UiScreenResolver.cs`	Answers "which screen was the operator looking at?" — writes `GetType().Name` into the hashed payload, hence the **rename freeze**
`IncidentExportBuilder.cs`	Reads both JSONL journals back for an incident export

7.4 Batch — `Ring/Services/Batch/`

File	Responsibility
`BatchLifecycleRecorder.cs`	The recorder driven by `BatchStepsPoller` edges; on batch end calls `BatchRepository.CompleteWithUsage` so completion and usage commit in **one** transaction
`BatchProgressMath.cs`	Pure progress math over the step snapshot
`BatchStampsBackfillService.cs`	Startup catch-up for controller verdicts missed while Ring was down — bulk-only by design
`BatchStampsMatcher.cs`	Decides which controller stamps slot belongs to a given Ring batch
`BatchHistoryBackfillService.cs`	Preview-first importer for the legacy six-month batch history
`BadBatchEarlyWarningService.cs`	Pure warning engine for live bad-batch indicators. "It never owns plant assumptions: every warning requires caller-supplied baselines and thresholds"

File	Responsibility
`BatchCompletionCelebrationService.cs` + `BatchCompletionScorecardComposer.cs` + `ICompletionChimePlayer` / `CompletionChimePlayer.cs`	The "batch klaar" moment: a chime deliberately **different** from the alarm sound, plus a composed toast line

Batch **policy** types (eligibility, preconditions, split mapping) live with the screens in `Ring/Views/BatchStart/` — see chapter 06 §6.3.

7.5 Display — `Ring/Services/Display/`

File	Responsibility
`LocalizationService.cs`	Thirteen-locale dictionary swapping; **never touches `CurrentCulture`/`CurrentUICulture`**. §6.5
`DisplayUnitService.cs`	Owns the station's chosen display units. Display only
`UnitSystem.cs` / `PerQuantityUnits.cs` / `UnitDisplay.cs`	The unit family, the per-quantity volume unit, and the unit-aware layer around the formatter — all explicitly **display only**
`PlcDisplayFormatter.cs`	Single source of truth for PLC-value display formatting
`LocalizedInput.cs` / `OperatorNumberInput.cs`	Locale-safe and locale-**tolerant** parsing of operator-typed numbers. The distinction matters: operator input accepts both conventions; wire values never do (see `PlcTagWriter.TryParseRealInvariant`)
`DataFreshnessClassifier.cs`	How old is this PLC-sourced value, and may it be shown? Feeds the freshness converters

`Ring/Services/PlantNativeUnits.cs` (top level) is the single declaration of this site's **native** recipe/batch weight unit and the display seam every consumer goes through — read it together with `docs/production-readiness/RECIPE_UNITS_VERDICT_2026-07-31.md`, which settled the question: **pounds**.

7.6 Reports — `Ring/Services/Reports/`

The largest non-PLC subfolder. Three layers, and it is worth keeping them separate when you add one.

Pure calculators (no I/O, no PLC, no UI, fully testable):

File	Responsibility
`BatchCostMath.cs`	Per-batch cost math — ingredient cost and leak detection
`BatchCompletionEtaCalculator.cs`	Batch-completion ETA math
`DowntimeOeeCalculator.cs`	Downtime/OEE rollup over operator-captured downtime events plus caller-supplied economics

File	Responsibility
`DrySolidsMath.cs`	Converts a delivered batch volume (US gal) to dry-starch pounds; feeds glue-usage, dry-lbs/sqft and giveaway-$ analytics
`DryLbsPerSqFtCalculator.cs`	Dry-lbs-per-square-foot ratio; an explicit no-data outcome distinct from zero when solids or completed batches are missing
`GiveawayCostCalculator.cs`	Prices giveaway volume; returns `NotConfigured` rather than `$0` when no marginal starch rate is set
`YieldScorecardCalculator.cs`	End-of-batch yield and quality scorecard math
`YieldScorecardAggregator.cs`	Windowed yield/giveaway aggregation — distribution counts and a signed mean giveaway
`ReportScheduleCalculator.cs`	Pure cadence due-math for scheduled report emails (next-due, monthly rules)
`ReportDateRange.cs`	Canonical half-open `[StartInclusive, EndExclusive)` date range shared by the report screens
`ReportPageNumbering.cs`	The page-count stamp in a report's page header
`ProcessHistoryChartProjection.cs`	Turns queried `ProcessHistorySample` rows into summary statistics and a gap-aware chart series for the Tank History report
`ShiftConsumptionBreakdown.cs`	Per-operation ingredient breakdown for one Shift Consumption occurrence
`InventoryInitiatorMapper.cs`	Decodes `Inventory_Stamps[0]` ("Update Initiator") to display text, mirroring the legacy switch

Read-only query services (assemble a payload from persisted data only):

File	Responsibility
`CostsQueryService.cs`	Assembles the Costs report / dashboard "Spend" payload from Batch + IngredientUsage + FormulaSteps + effective-dated IngredientCost
`GlueUsageQueryService.cs`	Sums delivered glue volume and derived dry-starch pounds over a date range
`YieldGiveawayQueryService.cs`	Yield/Giveaway Tracker query, using only persisted `RequestLevel/ActualVolume` batch fields
`ShiftConsumptionQueryService.cs`	Joins finalized batch/ingredient-usage rows to operator-defined shift windows
`ShiftHandoverQueryService.cs`	"What happened on my shift" aggregation for one shift occurrence
`DryLbsPerSqFtQueryService.cs`	The join that pairs each shift occurrence with `DryLbsPerSqFtCalculator`'s pure ratio

File	Responsibility
`BatchEtaQueryService.cs`	Thin read-only layer feeding `BatchCompletionEtaCalculator`
`AlarmBatchCorrelationService.cs`	Correlates alarms with the batch(es) running when each alarm went active
`DowntimeDollarCounterService.cs`	Aggregates alarm lifecycle durations by alarm number/description and applies caller-supplied economics
`FormulaCycleBaselineService.cs`	Per-formula, per-step **median** cycle-time baselines from observed completed batch-step timings

Document builders (`FlowDocument` for on-screen preview and `PrintDialog` output — each pairs with the report screen of the same name in chapter 06 §6.3): `BatchReportDocument`, `BatchHistoryReportDocument`, `BatchHistoryUsageReportDocument`, `UsageReportDocument`, `CostsReportDocument`, `FormulaReportDocument`, `GlueUsageReportDocument`, `InventorySnapshotReportDocument`, `ShiftHandoverReportDocument`, `YieldGiveawayReportDocument`, `DataEntryReportDocument`. `PdfExportService` renders any of them to PDF.

Scheduling: `ReportSchedulerService` (1-minute timer over subscriptions) + `ScheduledReportRenderer`.

And one small type that carries a contract: `ReportDataUnavailableNotice.cs` — the single wording used everywhere a report surface has to say "this is not a zero, the data could not be read". It is the user-facing half of `RepositoryReadScope` (§5.2).

7.7 Email and notifications

`Ring/Services/Email/` — `IEmailDeliveryService` / `EmailDeliveryService.cs` is the shared SMTP transport for the whole app: `Enqueue` drops the message on an in-memory `BlockingCollection` and returns immediately, so a dead or slow SMTP server can never block the UI thread; a background worker sends with a 3-attempt retry (1s/2s/4s backoff).

Durable-by-construction queue. `Enqueue` writes a `Status='Pending'` `EmailLog` row (`IEmailLogRepository`) **synchronously, before** the message ever touches the in-memory queue. The worker updates that same row to `Sent` or `Failed` when the send finishes, with the full exception text on failure. This means a dead SMTP server degrades to visible log rows, never a frozen HMI or a silently vanished message.

Crash / power-cut recovery — and why it never auto-resends. A process death after the pending row is written but before the worker finishes leaves that row stuck at `Status='Pending'`. On the **next** startup, `RecoverPendingFromPriorRun` sweeps every such row and marks it `Failed` with reason `"recovered after restart…"` — it is deliberately **never** resent, because this service cannot tell whether the original attempt actually left the wire, and a duplicate alarm/report email is judged a worse failure mode than a documented gap the operator can act on. **Documented residual:** the one window this doesn't cover is a crash during the few seconds `DatabaseWriterQuiesce/SuspendDelivery` holds the database file for a

restore or factory reset — a message queued in that narrow window has no durable row to recover, so it degrades to in-memory-only for that one message. Outside a crash, `SuspendDelivery`/`ResumeDelivery` itself does not drop mail: queued messages stay in the in-memory queue and drain on resume, and their log writes are buffered and replayed, not lost.

`EmailAddressValidator.cs` is the single "is this a usable address" check for the UI; `EmailSeedService.cs` is the idempotent, self-trapping seeder for out-of-the-box defaults; `BatchCompletionEmailService.cs` composes the completion mail.

`Ring/Services/Notifications/` — `ToastService.cs` (app-wide non-blocking toasts; singleton because a single `ToastHost` overlay is hosted once), `ToastItem.cs`, `ToastSeverity.cs`, `ToastBrushPalette.cs` (severity → semantic design-system accent brush).

7.8 Predictive — `Ring/Services/Predictive/`

All five are pure or read-only.

File	Responsibility
`FeedRateMonitorService.cs`	Pure, fully-testable predictive-failure math. No DB calls
`EquipmentHealthService.cs`	Evaluates drift rows plus the two facts that qualify them
`TankTemperatureStabilityCalculator.cs`	One tank's cook-temperature stability over a window: mean and variability
`GlueLineRunDryCalculator.cs`	Run-dry projection for the mix tank feeding the corrugating glue line
`IPredictiveEventSource.cs`	The hand-off point from the predictive monitor to the alarm-email rule engine

Related top-level services in the same spirit: `GoldenBatchDeviationCalculator.cs` (temperature only — **not** viscosity, and the file says so), `IngredientStockForecastService.cs` (pure stock-runway math), `MaintenanceDueCalculator.cs` / `MaintenanceReminderEvaluator.cs` (which return `Unknown` with a null percent and an em-dash rather than inventing a number when there is no usable interval), `ScaleCalibrationOverdueEvaluator.cs`, `TankCipDueCalculator.cs`, `ReceivingVarianceMath.cs` (whose honesty rule is that with no configured tolerance an event is **never** flagged short).

7.9 Roster and group setup

`Ring/Services/Roster/` — `TankRoster.cs` (the ordered site-configurable slot list, `Default()` and `IndianaPreset()`), `TankRosterSlot.cs`, `TankRole.cs`, `TankRosterState.cs` (the process-wide holder over `%LocalAppData%\Ring\tank_roster.json`, with `Changed`, `RestartRequired`, `LoadFailed` and

`CaptureSessionBaseline`), `StorageTankScope.cs` (the storage tank numbers an analytics surface must cover, taken from the roster).

Everything downstream of the roster is safety-relevant: see chapter 01 §1.2 and chapter 04 §4.3.

`Ring/Services/GroupSetup/` — `GroupHardwareSetupState.cs`, the legacy Setup → Groups hardware options (tank count, display style, screen tank mapping, cooling/discharge) over `%LocalAppData%\Ring\group_hardware_setup.json`.

7.10 Operators, configuration, documentation, export

Directory / file	Responsibility
`Operators/IOperatorSessionService.cs`, `OperatorSessionService.cs`	"Who is operating right now" — **identity only, not authentication**; best-effort persisted last choice in `%LocalAppData%\Ring\operator-session.json`
`Configuration/IConfigChangeNotifier.cs`, `ConfigChangeNotifier.cs`	Notifies consumer screens when a supervisor edits a category of configuration
`Documentation/ManualPageLauncher.cs`	Opens the shipped all-in-one RS-3000 operators manual at a specific page
`Documentation/ScreenHelpService.cs`	One per-screen deep link into the manuals library
`Documentation/MarkdownToHtml.cs`	Minimal dependency-free Markdown → HTML for the operator-facing docs
`Export/DatabaseExportSchema.cs`	The **whitelist** of exportable tables and columns
`Export/DatabaseExportService.cs`	Streams a whitelisted table to CSV

Authentication proper lives outside `Services/`, in `Ring/Infrastructure/Security/` — `AdminCredentialGate.cs`, `CredentialRotationService.cs` (which writes `appsettings.local.json`), `LoginAttemptTracker.cs`.

Two more small, easily-missed files sit beside the settings writers in `Ring/Infrastructure/Configuration/`, not in `Security/`: `SmtpPasswordProtector.cs` (DPAPI-protects the SMTP password stored in configuration) and `WebhookUrlValidator.cs` (the SSRF guard consulted by `WebhookAlarmEscalator` before it POSTs to an operator-configured URL).

7.11 Top-level services — `Ring/Services/*.cs`

Logging, crash and health

File	Responsibility
`ILogger.cs` / `Logger.cs`	The logging abstraction and its file/ring-buffer implementation (`%ProgramData%\Ring\logs\application.log`)
`CrashLogger.cs`	Global last-chance crash logger, wiring three escape paths to one file-per-crash under `<exe dir>\logs\crashes\`
`UiHangDetector.cs`	Classifies the UI thread's state from a single sentinel tick (paired with `MainWindow`'s 5 s ping/pong)
`UnbiasedClock.cs`	A monotonic millisecond clock that does **not** advance while the system is asleep
`DiagnosticBundleService.cs`	Builds a one-click support zip

Database lifetime

`DatabaseBackupService.cs`, `DatabaseHealthService.cs`, `DatabaseWriterQuiesce.cs`, `DatabaseMaintenanceLatch.cs` — all covered in chapter 05.

Startup and process

File	Responsibility
`StartupDegradedState.cs`	Records the session-degrading condition (failed DB init) **and** forces `PlcWriteGuard` read-only for the session
`SecondLaunchAdvice.cs`	Decides what to tell a second launch. Two independent brakes: an explicit boot marker (authoritative — the other process says it is still starting) and a sustained re-probe for the post-boot case
`NameTablesStartupHydrator.cs`	Hydrates the three name tables at startup
`NavHistoryDisposer.cs`	Pure decision logic for back-navigation history: when a retained screen may be disposed

Ambient name state

`MakeReadyTankSupervisorNamesState.cs` and `StorageTankGroupSupervisorNamesState.cs` — in-memory supervisor-entered names (and, for the former, derived inventory line titles), updated on Apply from the matching Setup screen, with a `NamesChanged` event the UI subscribes to. Both are examples of the ambient-static pattern that is being retired (chapter 02 §2.6).

Formula and recipe

File	Responsibility
`FormulaNameResolver.cs`	App-wide display-name lookups for formula numbers and storage tank numbers
`FormulaSolidsMath.cs`	Dry-solids percentage from recipe rows. **Display/report only — never touches a PLC.** Its two oracles (the plant's own decoded recipe CSV and the legacy step-seed variant) are pinned forever by golden parity tests
`FormulaIngredientAmountLimits.cs`	Min/max/default amount (lb) per formula operation, aligned with the legacy panel limits
`PlantNativeUnits.cs`	The single declaration of the site's native weight unit

Maintenance and calibration

`MaintenanceTaskSeeder.cs` (seeds sensible defaults on first run), `MaintenancePmSeedData.cs` (the one replaceable file holding the *provisional* PM register), `MaintenancePlantPmSeedData.cs` (the **real** plant PM register, seeded from the site's own PM forms), `MaintenanceDueCalculator.cs`, `MaintenanceReminderEvaluator.cs`, `ScaleCalibrationOverdueEvaluator.cs`, `TankCipDueCalculator.cs`.

Analytics and history

`ProcessHistorianService.cs`, `RuntimeAccumulatorService.cs`, `TankLevelTrendService.cs`, `ShiftDryLbsService.cs` (one shift's derived dry-lbs total **together with the population it was summed over** — the pairing is the point), `IngredientStockForecastService.cs`, `GoldenBatchDeviationCalculator.cs`, `ReceivingVarianceMath.cs`, `ReceivingVerificationService.cs` (manual-first bulk-receiving verification).

Commissioning and legacy

`PlantProfileService.cs` (builds a versioned `PlantProfile` JSON commissioning export), `LegacyRs360ImportService.cs` (imports legacy tank names, make-ready ingredient names and custom data), `DashboardService.cs` (coordinates dashboard data from the existing ViewModels — and is on the write-surface register as `Infrastructure` because it names `PlcWriteGuard`).

7.12 A note on where *not* to put things

Two conventions this catalog makes visible and that are worth preserving:

- **Pure math lives in its own type, WPF-free and I/O-free.** Almost every calculator above says so in its own doc comment, and the reason is stated in `CONTRIBUTING.md`: tests prefer WPF-free logic objects over constructing `UserControl`s. A calculation embedded in a code-behind is a calculation with no

test.

- **"Honesty" types are a pattern here, not an accident.** `ReportDataUnavailableNotice`, `RepositoryReadScope`, `MaintenanceStatus.Unknown`, `GiveawayCostState.NotConfigured`, `TankCapability.Unknown`, `PlcWriteStatus.WrittenUnverified`, `HasAnyData` on the snapshots — each exists so a missing input renders as *missing*, not as a confident zero. Follow the pattern rather than defaulting.

7.13 Where the evidence lives

Every subsystem above is described mechanically somewhere in this book, but nothing collects "here is the file you open" in one place. This table does.

Artefact	Path	Written by	Rotation
Application log	`%ProgramData%\Ring\logs\application.log`	`Ring/Services/Logger.cs`	Size-rotated, `MaxLogFileBytes = 10 MB` (`:35`), `MaxBackupCount = 5` (`:36`) → `.log.1`…`.log.5`
PLC comm log (in-memory)	n/a — a 500-event ring buffer	`PlcCommunicationLogService.cs` (`MaxBufferedEvents = 500`, `:32`)	Overwrites oldest on overflow
PLC comm log (optional per-day CSV)	`%ProgramData%\Ring\logs\plc-comm-YYYY-MM-DD.csv` (`PlcCommunicationLogService.cs:74`) — the same folder as `application.log`, not the install directory	Same service, gated by its own `DiskLoggingEnabled` runtime property (`:62`) — a UI toggle in `DisplayCommunicationForm` / the NavBar comm-log button, **not an `AppSettings` key**	`RetentionDays = 30` (`:68`), also a runtime property
PLC comm log (SQL table)	`PlcCommunicationLog` table inside the database (schema v32), queryable via Setup → Database Export	`PlcCommEventLogService.cs` — see ch. 03 §3.7, not the CSV row above	`PlcSettings.CommLogRetentionDays`, default 90 d, pruned on startup + daily
Write evidence journal	`%ProgramData%\Ringwood\Ring\plc-write-evidence.jsonl`	`PlcWriteEvidenceJournal.cs:75-79`	Size-rotated at 10 MB, 5 retained archives (ch. 04 §4.9)
Operator audit journal	`%ProgramData%\Ringwood\Ring\ui-audit\ui-audit.jsonl`	`UiAuditJournal.cs:1040-1041`	Append-only, hash-chained (ch. 05 §5.8)
Production gate summary	`artifacts/production-gate/production-gate-summary.json`	`scripts/Invoke-ProductionGate.ps1` (`:190`)	**Generated, not committed** — regenerated every gate run

Artefact	Path	Written by	Rotation
The database	`<exe dir>\RingwoodDatabase.db` (+ `-wal/-shm`)	Repositories, via `RingwoodDbAccess`	See ch. 05 §5.1

Note that the two `%ProgramData%` roots are **not the same directory** — `Ring\logs\` for the application log versus `Ringwood\Ring\` for both JSONL journals. This is a naming accident worth knowing before you go looking for one and find the other empty.

A gate-blocked write, as it appears in the log. `PlcWriteGuard.LogBlocked` (`PlcWriteGuard.cs:97`) emits one line per suppressed write:

```
[WARN] [ReadOnly] PlcTagWriter.Write: PLC write to 'Tanks[3].Preset_Temp'=75 SUPPRESSED (ReadOnlyMode ON — nothing sent to the controller).
[WARN] [ReadOnly] TVCControlPlcService.SetAgitatorModeAsync: PLC write to 'Tanks[2].Agitator_mode'=Auto SUPPRESSED (ReadOnlyMode ON — nothing sent to the controller).
```

The `source` half of the message (`PlcTagWriter.Write`, `TVCControlPlcService.SetAgitatorModeAsync`) is whatever string the caller passed — it names the method, not a fixed vocabulary, so grepping for `SUPPRESSED` is more reliable than grepping for one source name.

A healthy boot, roughly in order. Reconstructed from the call order in `App.xaml.cs` → `ConfigurationService.cs` → `DatabaseInitializer.cs` → `DatabaseHealthService.cs` → `MainWindow.xaml.cs`; treat this as high-confidence, not an exhaustively traced guarantee — a real boot has more conditional branches than this list:

1. `"Configuration loaded successfully from: {path}"` (`ConfigurationService.cs`)
2. `"[DatabaseInitializer] Boot sample: Path={path}, FileExistedAtBoot={bool}"`
3. `"[DatabaseInitializer] Starting. Path={path}, FileExistedAtStartup={bool}"`
4. `"Database initialized (first run) at {path}"` or `"Database OK at {path}"`
5. `"[DatabaseHealthService] PRAGMA integrity_check = ok ({path})"`
6. `"[DatabaseHealthService] Schema check OK ({path})"`
7. `"[DatabaseHealthService] Online backup OK → {backupPath} ({bytes} bytes)"`
8. Background services (`TankNamePlcNameHydrator`, `PcWriteInteger30StrandedBitMonitor`, `ReportSchedulerService`, `ProcessHistorianService`, `RuntimeAccumulatorService`) are **silent on success** — they only log on failure, so their absence from the log is itself the healthy signal.
9. `MainWindow`'s background reachability probe: `"[PlcMonitoring] TryStartPlcMonitoringIfReachable: resolved PLC IP = {ip}"`, then `"[PlcMonitoring] PLC reachable at {ip}:44818 on attempt {n}/3 — starting monitoring."`, then the `StartPlcMonitoring` sequence ending in `"[PlcMonitoring] StartPlcMonitoring: started OK (background poll 500ms, UI 1000ms)."` (Demo mode replaces the probe with `"[PlcMonitoring] Demo mode enabled — skipping TCP reachability probe, starting synthetic pollers directly."`)

A caution the log itself creates. At the shipped `Debug` log level the application log has been observed to grow at roughly 32 MB/h; with the 10 MB × 5 rotation ring that is only about 1.6 hours of retained history (`docs/production-readiness/WRITE_ENABLE_READINESS_2026-07-26.md:441-444`, which recommends shipping `Information` for production or enlarging the ring). If you are trying to find what happened more than two hours ago on a `Debug`-level station, the log may simply no longer have it.

Next: 08 — Testing and Quality.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

Every `.cs` file under `Ring/Services/` and its fifteen subdirectories (`Alarms`, `Audit`, `Batch`, `Configuration`, `Display`, `Documentation`, `Email`, `Export`, `GroupSetup`, `Notifications`, `Operators`, `PLC`, `Predictive`, `Reports`, `Roster`) — summaries taken from each type's own doc comment · `Ring/Infrastructure/Security/` (listing) · `Ring/Infrastructure/Configuration/AlarmEscalationConfigWriter.cs`, `AnalyticsSettingsConfigWriter.cs`, `CostSettingsConfigWriter.cs`, `AppSettingsSectionWriter.cs`, `SmtpPasswordProtector.cs`, `WebhookUrlValidator.cs` · `Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs` · `Ring/Services/PLC/PlcWriteGuard.cs`, `PlcCommunicationLogService.cs`, `PlcCommEventLogService.cs`, `PlcCommEventType.cs` · `Ring/Services/Email/EmailDeliveryService.cs` (`RecoverPendingFromPriorRun`, `RecoveredAfterRestartError`, `SuspendDelivery/ResumeDelivery`, `CompletePendingSafe`) · `Ring/Database/Interfaces/IEmailLogRepository.cs`, `Ring/Database/Repositories/EmailLogRepository.cs` · `Ring/Database/Repositories/PlcCommunicationLogRepository.cs` · `Ring/Services/PLC/UdtTankReader.cs` · `scripts/Invoke-ProductionGate.ps1` · `docs/production-readiness/WRITE_ENABLE_READINESS_2026-07-26.md` · `CONTRIBUTING.md`

08 — Testing and Quality

Who this is for. Anyone running the suite, adding a test, or trying to work out why a build that "should" be green is not.

What you'll learn. The exact mechanics of running this suite (they are not the .NET defaults); why parallelism is disabled and must stay that way; the non-SDK project-membership guard; the production gate and what it actually asserts; the encoding rules; and the worktree conventions that keep concurrent sessions from corrupting each other's work.

8.1 Suite mechanics — read this before you run anything

There is no `dotnet test` here. Both projects are classic non-SDK MSBuild projects (`Ring/Ring.csproj:11`, `Ring.Tests/Ring.Tests.csproj:11`, both `v4.7.2`), and the SDK test host cannot run them.

MSBuild is normally not on `PATH`. Use the full path, and use **PowerShell**, not Git Bash — Git Bash mangles switches like `/m` into path-looking arguments:

```
$msbuild = "C:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools\MSBuild\Current\Bin\MSBuild.exe"

nuget restore Ring.sln
& $msbuild Ring.sln /m /t:Build /p:Configuration=Debug "/p:Platform=Any CPU"
```

Run the suite with the xUnit console runner, restored into `packages/`:

```
& packages\xunit.runner.console.2.9.3\tools\net472\xunit.console.exe `
    Ring.Tests\bin\Debug\Ring.Tests.dll -nologo -parallel none
```

(The runner version is pinned in `Ring.Tests/packages.config`: `xunit.runner.console 2.9.3`.)

Log long output to a file and read the tail. The full transcript is noise (`CLAUDE.md`).

Check free disk space before diagnosing a mass failure. Parallel build lanes filling `C:` have produced *thousands* of phantom failures in this repository. This is the first thing to check when a failure appears out of nowhere, not the last.

`-parallel none` is required, not optional

Parallelization is disabled in two places, deliberately:

1. `[assembly: CollectionBehavior(DisableTestParallelization = true)]` — `Ring.Tests/Properties/AssemblyInfo.cs:10`
2. `-parallel none` on the runner command line, including in the gate

The reason is process-global state. Several tests flip the static `PlcWriteGuard` in `try/finally`, and any test class exercising a write path in parallel would see its writes silently suppressed (`CONTRIBUTING.md`). The same applies to `PlcHeartbeatConnectionTracker`, `TankInstalledWriteGate`, `PlcTagWriter.WireWriteHookForTests` and the snapshot holders.

Do not re-enable it, and do not introduce a test that depends on parallel execution. A test that pulls the `ForceReadOnlyForSession` latch must release it with `PlcWriteGuard.ResetForcedReadOnlyForTests()` **followed by `PlcWriteGuard.Configure(false)`** in a `finally`. The reset alone is not enough: `ForceReadOnlyForSession()` sets **both** `_forcedReadOnly = true` and `_readOnly = true` (`PlcWriteGuard.cs:51-52`), but `ResetForcedReadOnlyForTests()` clears only `_forcedReadOnly` (`PlcWriteGuard.cs:62-65`) — `_readOnly` stays `true` until a later `Configure` call recomputes it. Call the reset without the `Configure(false)` that follows it and read-only still leaks into every later test in the run, silently passing every later PLC-writer test because its writes are suppressed. See `Ring.Tests/StartupWriteGateAndPreMigrationBackupTests.cs:43-44` for the canonical pair, repeated at `TankTempModeWriteAuthorizationTests.cs:60-61`, `TankTempModeWriteTargetTests.cs:70-71` and `BenchPlcWriteRoundTripTests.cs:79-80`.

House conventions for writing tests

- **Prefer WPF-free logic objects over constructing `UserControl`s.** This is why so much of the codebase is factored into pure calculators, pure decision functions (`FormulaBankWriteInterlock.Decide`, `PlcPollingCoordinator.DecideStartupReconcile`, `PlcHeartbeatConnectionTracker.ClassifyHeldHeartbeat`, `FitHost.ComputeChildArrange`) and pure projections.
- **Tests that need a repo path use a `[CallerFilePath]`-anchored root, not the CWD.** The canonical implementation is `RingSourceRoot([CallerFilePath] string thisFile = null)` in `Ring.Tests/WriteSurfaceRegisterTests.cs:203-218` and `Ring.Tests/ProjectMembershipGuardTests.cs:76-90`, both with a walk-up-from-`BaseDirectory` fallback.
- **Test seams are marked and policed.** `PlcTagWriter.WireWriteHookForTests` and `PlcHeartbeatConnectionTracker.MonotonicNowTicks` are `public` only because `Ring.Tests` has no `InternalsVisibleTo` against `Ring`; source-scan tripwires in `Ring.Tests/MomentaryPulseClearGateTests.cs` fail if anything under `Ring/` assigns them.

Bench-gated tests pass *vacuously* by default

`Ring.Tests/BenchPlcIntegrationTests.cs` and `Ring.Tests/BenchPlcWriteRoundTripTests.cs` early-return a pass on every `[Fact]`/`[Theory]` when `BENCH_PLC_AVAILABLE` is unset — i.e. on every normal laptop or CI run. A green suite therefore does **not** mean bench hardware verified anything.

The gate is explicit about this rather than hiding it: it counts the gated methods **mechanically from source** and records both `benchGatedTestCount` and `benchHardwareWasAvailable` in its summary (`scripts/Invoke-ProductionGate.ps1:157-186`). When you cite "tests passed", cite those two fields alongside it.

8.2 The non-SDK csproj rule and its guard

Both projects are classic MSBuild projects. **Nothing is globbed.** A new `.cs`, `.xaml` or image is invisible to the compiler and to the shipped EXE until you hand-add a row:

```
<Compile Include="Services\PLC\MyNewService.cs" />
<Page Include="Views\Setup\MyNewScreen.xaml"><Generator>MSBuild:Compile</Generator>
<SubType>Designer</SubType></Page>
<Compile Include="Views\Setup\MyNewScreen.xaml.cs"><DependentUpon>MyNewScreen.xaml</DependentUpon>
</Compile>
```

This used to fail **silently**. Five orphan test files and a set of image variants lived in the tree for months looking like product code. It now fails loudly.

`ProjectMembershipGuardTests`

`Ring.Tests/ProjectMembershipGuardTests.cs` closes both directions of the failure:

Failure mode	Description
Orphan source	a `.cs` / `.xaml` / image lands under `Ring/` and is never registered. It compiles nowhere, ships nowhere, and reads to every later reviewer as live product code
Phantom include	a registered path that no longer exists on disk. MSBuild fails loudly for `Compile`, but a stale `Resource`/`Content` row can survive a rename and only surface as a **missing image at runtime, on the plant floor**

Three tests:

- `Every_source_and_asset_under_Ring_is_registered_in_the_project` (`:179`)
- `Every_project_include_points_at_a_file_that_exists` (`:199`)
- `The_membership_scan_is_not_vacuous` (`:222`) — the anti-blindness pin, the same pattern the write-surface census uses

It recognises registration through any of `Compile | Page | Resource | Content | ApplicationDefinition | None | EmbeddedResource` (`:40-42`), because the question it asks is "does the project know about this file", not "how is it packaged".

Scope is deliberately the shipping project only (`Ring/`). The test project's own membership is proven by the fact that a missing row means the test simply does not run.

The allow-list is not a mute button

`AllowList` (`:58-72`) maps a glob to a **truthful reason**. Every entry is a standing claim that the file is not product code. The current entries are the opaque-white vessel BMPs superseded by transparent-PNG siblings, kept on disk because the repository is under a rename freeze *and* because they are the conversion

script's only input — deleting them would make `scripts/Convert-VesselArtToAlpha.ps1` unreproducible.

"Keep the reason column truthful — it is the only thing separating this list from a mute button."

8.3 The production gate

`scripts/Invoke-ProductionGate.ps1` is *the* gate. It is what CI runs (`.github/workflows/production-gate.yml`) and what you should run before asking anyone to merge.

```
powershell -NoProfile -ExecutionPolicy Bypass -File scripts\Invoke-ProductionGate.ps1
```

What it actually does, in order (verified against the script):

Step	Detail
Resolve MSBuild	`Get-Command msbuild.exe`, else `vswhere -latest … -find 'MSBuild\**\Bin\MSBuild.exe'`; throws with an install instruction if neither works (`:18-30`)
Guard the artifacts path	`ArtifactsRoot` must stay **inside** the repository (`:40-44`)
NuGet restore	throws if `nuget.exe` is missing; `-SkipRestore` is only for an already-verified restore (`:59-66`)
Isolated rebuild	`/t:Rebuild` into a unique `OutDir` — "Shared `bin\Debug/bin\Release` directories can be stale or concurrently cleaned by another build/test process, producing misleading assembly-load failures" (`:67-73`)
Output completeness check	asserts twelve required files exist, including `x64\SQLite.Interop.dll`, `libplctag.dll`, `libplctag.NativeImport.dll` and `Config\appsettings.json` (`:76-92`)
Full xUnit run	`-nologo -parallel none -xml`, then parses the result XML and throws unless **failures == 0 and skips == 0** (`:99-110`)
Regenerate the PLC manifests	runs `Parse-L5kLiveTags.ps1` and `Audit-CodeTags.ps1 -SkipHistoryAppend`, then `git diff --exit-code` over `scripts/live-tags.{md,json}` and `scripts/code-tag-audit.{md,json}` — **fails on drift** (`:111-127`)
Release verifier contract tests	`scripts/tests/ReleasePackageContract.Tests.ps1` (`:130-132`)
Build the release package	`New-ReleasePackage.ps1 -IncludeKiosk` (`:134-138`)
Re-verify the built package	expands the produced ZIP and runs `Test-ReleasePackage.ps1` against the expanded copy (`:145-151`)
Write the summary	`artifacts/production-gate/production-gate-summary.json` (`:172-189`)

The summary carries `commit`, `configuration`, `tests_total`, `tests_passed`, `tests_failed`, `tests_skipped`, `test_seconds`, `benchGatedTestCount`, `benchHardwareWasAvailable` and `generated_utc`.

> **That file is the authoritative test count.** Do not hardcode the number into any document — including this one. The last document that did was wrong by an order of magnitude for months (`README.md`).

The gate closes with an explicit non-claim, worth quoting because it bounds what a green run means:

> "Hardware-in-the-loop and signed factory commissioning evidence remain separate mandatory gates."

The second CI workflow

`.github/workflows/plc-tag-audit.yml` regenerates the tag manifests and fails if either (1) the regenerated files differ from the committed copies, or (2) the **dead-read count went up** versus the PR's base branch — meaning a new C# read was added against a tag the parsed export shows no ladder writer for.

It runs on `windows-latest` with `shell: powershell` (Windows PowerShell 5.1) **deliberately**: `ConvertTo-Json` formats differently between PS 5.1 and pwsh 7+ (indent width, colon spacing), so an ubuntu+pwsh runner would always diff against locally regenerated commits.

Remember what a "dead read" means before you act on one: `docs/PLC_FACTS.md` — it is a statement about the *parsed 2024 export*, not proof the value is dead in the plant. A rising count still deserves attention, because it usually means a read was added against a tag nobody has confirmed is fed.

8.4 Encoding, line endings and BOMs

`.gitattributes` at the repository root is deliberate and every line is commented. The rules that bite:

Rule	Why
`*.csproj text eol=crlf`, `*.sln text eol=crlf`	MSBuild and Visual Studio are happiest with CRLF; an LF `.sln` can confuse older VS tooling into a rewrite
csproj and sln stay BOM-free	`CLAUDE.md`
`*.jsonl`, `*.L5K`, `*.L5X` → `text eol=crlf` — **never `binary`, never `-text`**	Those files were committed under `* text=auto`, so the index holds LF and `core.autocrlf` puts CRLF in the working tree. Turning conversion **off** does not preserve that; it freezes whatever bytes are on disk and hands the next clone LF files. For the L5K that is a byte-level rewrite of the v21 export that is the math authority for the plant

Rule	Why
No renormalization pass was run, on purpose	`git add --renormalize` would produce a whole-repo diff that buries real changes for months. The rules govern files as they are added or modified from here on

Two working rules follow:

- **Preserve CRLF.** A bulk find-and-replace across a file can collapse CRLF to LF. Prefer a targeted edit, and **diff before committing** (`CLAUDE.md`).
- **Localization dictionaries are UTF-8 without BOM**, and if the mojibake test fires, repair the encoding rather than retyping the character (chapter 06 §6.5).

8.5 Working alongside other sessions

Parallel work happens in **git worktrees** under `..\worktrees\<lane>`, one branch per lane.

Why a worktree and not just a branch: concurrent sessions sharing one checkout race `.git/index` *and* `.git/HEAD`; only a worktree prevents both (`CLAUDE.md`). **Verify your branch after committing.**

Three practical rules:

1. **`packages/` in a worktree is a symlink/junction back to the primary checkout's `packages/`, and it is not created for you.** Provision the worktree *and* the junction before building, or restore will appear to succeed and the build will fail on missing references (`CONTRIBUTING.md`).
2. **Never modify another lane's worktree.**
3. **Never leave a "perturb it to prove the test fails" experiment in a shared checkout.** One such experiment flipped a safety default. `git status` before you trust a result.

Branch naming in use: `fix/<slug>`, `cleanup/lane-<x>`, `feature/<slug>`; see `docs/BRANCHES.md`. PRs go through `gh pr create`, and both CI workflows must be green.

Commit bodies should explain **why** — and, in this repository especially, **what you verified rather than assumed**.

8.6 The quality tests that are really safety artifacts

Four tests in this suite are load-bearing beyond their own assertions. Treat a change that touches them as a change to the safety model, not to test hygiene.

Test file	What it protects
`WriteSurfaceRegisterTests.cs`	The whole PLC write surface stays enumerated, classified, gated and reachable. §4.10

Test file	What it protects
`MomentaryPulseClearGateTests.cs`	`WriteClear` (a heartbeat-gate bypass) stays de-assert-only; the wire test hook is never assigned by production; `No_production_code_releases_the_one_way_read_only_latch` (`:858-869`) pins that `ResetForcedReadOnlyForTests` is never called from `Ring/` itself
`ProjectMembershipGuardTests.cs`	No orphan source, no phantom include
`DatabaseExpectedTablesDriftTests.cs`	The restore contract cannot drift from the real schema in either direction

All four share the same defensive shape and it is worth copying when you write something similar: **a "nothing matches X" assertion passes vacuously the moment the detector stops matching.** Each therefore carries an *anti-blindness* pin — synthetic positives, asserted negatives, and a floor on the live population — so a narrowed regex looks like a red build rather than a safer codebase.

Next: 09 — Build, Deploy, Release.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring.Tests/Properties/AssemblyInfo.cs` · `Ring.Tests/ProjectMembershipGuardTests.cs` · `Ring.Tests/WriteSurfaceRegisterTests.cs` · `Ring.Tests/MomentaryPulseClearGateTests.cs` · `Ring.Tests/StartupWriteGateAndPreMigrationBackupTests.cs`, `TankTempModeWriteAuthorizationTests.cs`, `TankTempModeWriteTargetTests.cs`, `BenchPlcWriteRoundTripTests.cs` · `Ring.Tests/DatabaseExpectedTablesDriftTests.cs` · `Ring.Tests/packages.config` · `Ring/Ring.csproj` · `Ring.Tests/Ring.Tests.csproj` · `scripts/Invoke-ProductionGate.ps1` · `.github/workflows/production-gate.yml` · `.github/workflows/plc-tag-audit.yml` · `.gitattributes` · `Ring/Services/PLC/PlcWriteGuard.cs` · `docs/PLC_FACTS.md` · `CONTRIBUTING.md` · `CLAUDE.md` · `docs/BRANCHES.md`

09 — Build, Deploy, Release

Who this is for. Anyone producing a build, cutting a package, or reasoning about what a plant machine actually receives.

What you'll learn. The build mechanics and the three traps in them; the shape of a release package and the provenance guarantees baked into it; how the verifier works; and where the install, rollback and remote-access procedures live — this chapter summarizes their **shape**, never their steps.

> **This chapter does not contain a procedure.** Cutover, install, rollback and remote access are owned by the runbooks named in `docs/INDEX.md`. Executing this book instead of a runbook would be a mistake.

9.1 Build mechanics

The toolchain and the commands are in chapter 08 §8.1 and `CONTRIBUTING.md`. What matters here are the three properties that make this build different from a modern SDK build.

1. Restore is `packages.config`-style. `nuget restore Ring.sln`, not `dotnet restore`. Packages land in `packages/` at the repository root, and both csproj files reference them by **relative `HintPath` with a pinned version** — which is why a package upgrade is a csproj edit, not just a `packages.config` edit.

2. Nothing is globbed. Every `.cs`, `.xaml` and image is hand-registered, and `ProjectMembershipGuardTests` fails otherwise (chapter 08 §8.2).

3. The build reaches into the output directory. Two csproj targets, both of which bite in practice:

Target	Behaviour
`CopyAppSettingsToOutput` (`Ring/Ring.csproj:1985`, `AfterTargets="Build"`)	Copies `Ring/Config/appsettings.json` over `bin\<cfg>\Config\appsettings.json` whenever the source is newer. **Anything the running app wrote there — PLC IP, `ReadOnlyMode`, SMTP credentials, cost rates — is reverted**
`RemoveLocalAppSettingsFromRelease` (`:1995-1999`, Release only)	Deliberately **keeps** an existing `appsettings.local.json` in the output and emits a warning. `/p:PurgeLocalConfig=true` is the opt-in that deletes it; `/p:SuppressLocalConfigWarning=true` silences the warning

The `.local.json` overlay is copied into the output only for **Debug**, and only if it exists in `Ring/Config/` (`Ring/Ring.csproj:1881`). Keep your overlay there — it is gitignored — not hand-placed in `bin\`, or a clean/rebuild takes it with the rest of the output.

Consequence to internalise: re-check `bin\<cfg>\Config\` after every build. `scripts/Launch-RingDemo.ps1` re-asserts its overlay on every launch for exactly this reason. And a Release package **can silently carry a developer's bench PLC IP** if nobody reads the warning — which is why `scripts/Preflight-FieldKit.ps1` and the cutover config read-back exist, and why `WriteSurfaceRegisterTests` is explicit that it cannot pin the deployed overlay (`WriteSurfaceRegisterTests.cs:1085-1090`).

For a real plant install, fill in `Ring/appsettings.production.template.json` and deploy it **as** `Config\appsettings.json`; with `Production.Enabled` true, validation requires an absolute DB path and an absolute backup directory (`ConfigurationValidator.cs:106, 112`), plus the rest of the strict-profile checks in chapter 02 §2.3.

9.2 The release package

`scripts/New-ReleasePackage.ps1` produces `Ring-<Configuration>-<timestamp>.zip` plus a sidecar `…sha256.txt`, under `artifacts/release` by default. The authoritative description of the contract is `docs/production-readiness/RELEASE_PACKAGE_CONTRACT.md`; what follows is the shape, read out of the script.

Layout

```
Ring-<cfg>-<timestamp>/
  App/                                the built application tree
  Install/                            only with -IncludeKiosk
    Install-RingKiosk.ps1
    Install-RingAutoRestart.ps1
    Install-RingWerDumps.ps1
    Install-DesktopLaunchers.ps1
    README-KIOSK.txt
  12_INSTALL_RUNBOOK.md
  13_RS3000_ROLLBACK.md
  STARCH_SYSTEM_CUTOVER_RUNBOOK.md
  CUTOVER_RUNBOOK.md
  release-manifest.json
```

The runbooks ship inside the package (`New-ReleasePackage.ps1:221-233`), so the machine that has the software also has the procedure for installing it, cutting over and rolling back. `CUTOVER_RUNBOOK.md` is staged **unconditionally**, because the kiosk README references it.

The four kiosk scripts ship **together or not at all** — `Install-RingKiosk.ps1` composes the other three, and the script throws if any is missing (`:247-255`).

Required payload

The package asserts these exist in `App/` before it will build (`New-ReleasePackage.ps1:186-196`):

```
Ring.exe
Ring.exe.config
Config\appsettings.json
libplctag.dll
libplctag.NativeImport.dll
x64\SQLite.Interop.dll
Assets\Manuals\RS-3000 Manual All In One Update (2021).pdf
```

The operator manual is part of the payload, not an afterthought — Ring opens it from Help → manuals via `Ring/Services/Documentation/ManualPageLauncher.cs`.

Provenance is enforced at build time

`Get-SourceProvenance` (`New-ReleasePackage.ps1:31-56`):

- resolves the full 40-character commit and **throws** if it cannot;
- checks unstaged, staged *and* untracked files, excluding only `scripts/audit-history.csv` (generated evidence, not a package input);
- **throws unless the tree is clean**, with `-AllowDirtySource` reserved for a "non-customer developer package".

Optional Authenticode policy is available via `-RequireAuthenticode` and `-ExpectedPublisher`, asserted against `App\Ring.exe` (`:204`).

`release-manifest.json`

Schema v2 (`New-ReleasePackage.ps1:311-323`):

Field	Meaning
`schemaVersion`	2
`product`	`"Ring"`
`releaseVersion`	derived from `Ring.exe`'s `ProductVersion` unless supplied
`configuration`	Debug/Release
`createdUtc`	
`sourceCommit`	the 40-char commit
`sourceTreeClean`	whether package inputs matched that commit
`databaseSchemaVersion`	so a package and a database can be reasoned about together
`authenticodeRequired`, `expectedPublisher`	signing policy
`files[]`	every file: relative path, length, **SHA-256**

Schema v2 exists to close a provenance gap in v1: *the exact source commit and whether the inputs matched it now travel inside the hash-protected manifest* (`Test-ReleasePackage.ps1:29-31`).

Verification

`scripts/Test-ReleasePackage.ps1 -PackageRoot <expanded package>` re-verifies a package. It:

- requires `release-manifest.json` to be present and valid JSON;
- accepts `schemaVersion` 1 **or** 2 — v1 stays readable *"so an older rollback package does not become unverifiable merely because the verifier was upgraded"*, which is a deliberate rollback-safety property;
- for v2, requires a valid 40-char `sourceCommit` and a boolean `sourceTreeClean`, and **refuses a customer deployment built from a dirty tree** unless `-AllowDirtySource`;
- re-hashes the payload against the manifest.

The production gate runs this end-to-end on every run: build the package, expand the ZIP, verify the expanded copy (`scripts/Invoke-ProductionGate.ps1:134-151`), preceded by `scripts/tests/ReleasePackageContract.Tests.ps1` — contract tests for the verifier itself.

What the package deliberately leaves out — and why an upgrade is safe

`Should-ExcludePackageFile` (`New-ReleasePackage.ps1:172-187`) excludes `Config/appsettings.local.json`, every `*.db`/`*.db-wal`/`*.db-shm` file, `logs/*`, `crash-dumps/*`, `*.log` and `*.bak` from every release ZIP. This is what makes an in-place **upgrade** — copy a new package's `App/` contents over an existing `C:\Ring\App\` — safe by construction: there is nothing in the package that could overwrite a station's live database or its site-specific settings overlay, even if you copy the entire `App/` tree without excluding anything yourself.

Upgrade and rollback are opposites; do not conflate them. An upgrade adds new program files on top of a working install. A rollback — `docs/production-readiness/13_RS3000_ROLLBACK.md` for a live plant, or `12_INSTALL_RUNBOOK.md` §10 for an abandoned in-progress install — *deletes* the install directory outright and is scoped to abandoning that install, never to "start the upgrade over."

Two artifacts that used to live under the install directory — and so could have been destroyed by a rollback, a factory reinstall, or a careless manual cleanup — now live under `%ProgramData%\Ring\` instead, for the same reason the application log does (see §7.13):

Artifact	Path	Why it moved
Scheduled report PDFs	`%ProgramData%\Ring\ScheduledReports\`	`ReportSchedulerService.cs:87-98` — its own comment notes the release-package exclusion list has no rule for a "ScheduledReports" folder, so anything written under the install directory would both risk shipping into a future build and sit exactly where a wipe-and-reinstall would delete it
PLC communication log (optional per-day CSV)	`%ProgramData%\Ring\logs\plc-comm-YYYY-MM-DD.csv`	`PlcCommunicationLogService.cs:74` — the **same** folder as `application.log`, not a separate location; see chapter 03 §3.7

9.3 Install, kiosk and unattended operation

Authority: `docs/production-readiness/12_INSTALL_RUNBOOK.md`.

The shape: extract the package, copy `App/` to its install location (e.g. `C:\Ring\App`), then run the kiosk installer from an **elevated** PowerShell. `Install-RingKiosk.ps1` composes four concerns:

Concern	Script
Watchdog auto-restart	`Install-RingAutoRestart.ps1`
Native crash dumps (WER)	`Install-RingWerDumps.ps1`
Desktop shortcuts, branding, first-run seeding	`Install-DesktopLaunchers.ps1`
Composition, `-DryRun`, `-Verify`, `-Uninstall`	`Install-RingKiosk.ps1`

`-DryRun` reviews the plan without changing anything; `-Verify` proves the watchdog relaunches Ring after a kill; `-Uninstall` removes the task, the WER key and the shortcuts. The bundled `README-KIOSK.txt` carries paste-ready commands and points at `12_INSTALL_RUNBOOK.md` §5A/5B and `CUTOVER_RUNBOOK.md` §8.

Two unattended-operation behaviours to know, because they interact with the install:

- **Startup warnings self-continue after 60 seconds.** A *warning* must never park a 3 a.m. auto-restart on a human click (chapter 02 §2.3). A **fatal** config error still blocks forever, by design.
- **A write-enabled station always warns** (the single-writer advisory), so expect the countdown dialog on every post-cutover boot.

Crash evidence: `docs/production-readiness/11_CRASH_DUMP_RUNBOOK.md`; the OS watchdog: `docs/production-readiness/22_OS_WATCHDOG_RUNBOOK.md`.

9.4 Cutover and rollback

Authorities, in order of precedence for this plant:

Question	Document
Are we allowed to enable writes at all?	`docs/production-readiness/WRITE_ENABLE_READINESS_2026-07-26.md` — the Sept-8 gate. Every box must be tickable; an untickable box is a NO-GO
How do we cut over one LCP?	`CUTOVER_RUNBOOK.md` — ships inside every release ZIP; per-LCP procedure, verification per step, rollback, decision matrix
First bring-up of any new starch system	`docs/production-readiness/STARCH_SYSTEM_CUTOVER_RUNBOOK.md` — the reusable read-only-first pattern. Where the two differ on procedure **for this plant**, `CUTOVER_RUNBOOK.md` wins

Question	Document
How do we roll back?	`docs/production-readiness/13_RS3000_ROLLBACK.md` — the 10-minute operator-facing fallback. `CUTOVER_RUNBOOK.md` §4 owns the underlying data-directory / disk-image mechanism
The days after	`HYPERCARE_PLAN.md`
What is deliberately left until after	`docs/production-readiness/POST_CUTOVER_FOLLOWUPS.md` and `ARCHITECTURE.md`

The engineering shape of the cutover, in one paragraph. Ring is installed and run **read-only first** against the live controller, so every read is proven before any write is possible. The cutover flips `PlcSettings.ReadOnlyMode` to `false`, which arms layer 1 of the gate stack — and *only* layer 1. The seven fail-closed `Enable*` flags, the commissioning holds and the heartbeat gate are unchanged by that flip; each feature family is enabled separately, on its own evidence. The switches the cutover actually touches are tabulated in `docs/reference/plc/APPSETTINGS_REFERENCE.md`. See chapter 04 for why flipping `ReadOnlyMode` without flipping, say, `EnableFormulaBankWrite` is safe by construction.

One deferred item is deliberately post-cutover and worth knowing about: the git history rewrite that would drop a dead ~53 MB blob. It rewrites every commit on the shared trunk and would force a rebase of every worktree and unmerged branch days before the plant goes live (`ARCHITECTURE.md`).

9.5 Remote access

Authorities: `docs/production-readiness/REMOTE_ACCESS_RUNBOOK.md`; off-site v2 in `docs/production-readiness/REMOTE_ACCESS_V2_RUNBOOK.md` and `docs/production-readiness/REMOTE_ACCESS_V2_CLOUDFLARE.md`.

The companion package lives at `remote-view/` in this repository — a MeshCentral-based viewer with its own build (`Build-RemoteViewPackage.ps1`), installers (`Install-RemoteView.ps1`, `Install-RemoteViewTunnel.ps1`), a firewall-rule helper, an Inno Setup script (`RingRemoteView.iss`), a tunnel smoke test, and its own `THIRD-PARTY-LICENSES.txt` and `NOTICE`. See `remote-view/README.md`.

The engineering hazard remote access creates is not networking, it is the single-writer rule. A MeshCentral or RDP seat is a **second Windows session on the same machine**, and Ring's single-instance mutex is per-session, not per-machine (chapter 02 §2.3). A second write-enabled Ring on the same LCP would boot cleanly and drive the same setpoints. **Check Task Manager → Users for a second `Ring.exe`.**

9.6 Internet access for the remote PC

Outbound internet access is **optional**. Ring itself runs, reads the PLC, and records batch history with the plant machine fully offline — nothing in the core application requires an internet path. Exactly two features are the exception, and both are opt-in:

- **Queued SMTP email alerts.** `AlarmEscalation.Mode` ships as `"None"` (`Ring/Config/appsettings.json:73`), and `AlarmEscalationSettings.Mode` in `Ring/Infrastructure/Configuration/AppSettings.cs` documents the field as *`"None" | "Smtp" | "Webhook". Default "None" → log-only NoopAlarmEscalator.`* Email alerts are disabled by default; they only need outbound internet once a plant deliberately configures an SMTP host and switches the mode to `"Smtp"`.
- **The remote seat-share / remote viewing deployment.** The MeshCentral-based `remote-view/` package described in §9.5 needs outbound internet to reach its relay/tunnel. A plant that never installs `remote-view/` needs nothing here either.

A machine with neither feature configured has no reason to reach the internet at all.

Getting a remote PC on the plant network onto the internet. When a site does need one of the two features above on a PC that sits on the plant's PLC network, field guidance from Ringwood project management (Sept 2026) describes three patterns seen across sites:

1. **Plant IT grants that PC internet access directly**, through the plant's own network.
2. **Sites using an Ewon gateway:** the remote PC rides the Ewon's own network for its internet path; a second Ethernet adapter on that PC can then connect it to the plant network separately, for PLC/system communication.
3. **Two physical paths on the same PC:** wired Ethernet for system/PLC communication, and the plant's WiFi for internet.

Whichever pattern a site uses, the constraint is the same: **the PLC/system communication network stays on its own wired network, and the internet path must never bridge into it.**

9.7 Scripts: know what one does before you run it

`scripts/` holds ~85 files. **Some of them can write to a PLC.** `scripts/README.md` is the authority on what each one does and which are dangerous. Read it before running anything against the live box.

Broad families, for orientation only:

Family	Examples
Gate and release	`Invoke-ProductionGate.ps1`, `New-ReleasePackage.ps1`, `Test-ReleasePackage.ps1`, `tests/ReleasePackageContract.Tests.ps1`

Family	Examples
Tag audit (generated output — never hand-edit)	`Parse-L5kLiveTags.ps1`, `Audit-CodeTags.ps1`, `live-tags.{md,json}`, `code-tag-audit.{md,json}`
Simulation and demo	`Run-SimDemo.ps1`, `Stop-SimDemo.ps1`, `Launch-RingDemo.ps1`, `Drive-MockBatch.ps1`, `Loop-MockBatches.ps1`, `Seed-SmokeTestData.ps1`, `Reset-SmokeTestData.ps1`, `Tour-AllScreens.ps1`
Field capture and probing	`Probe-Plc.ps1`, `Probe-TagInventory.ps1`, `Snapshot-PlantTags.ps1`, `Capture-*.ps1`, `Measure-TagSessionChurn.ps1`, `Replay-PlantTelemetry.ps1`
Write-capable bench tools	`Test-BatchStartWriter.ps1`, `Test-PlcCommissioningEvidence.ps1`
Install / kiosk / watchdog	`Install-RingKiosk.ps1` and the three it composes, `Install-RingWatchdog.ps1`, `Uninstall-RingWatchdog.ps1`
Field kit	`Preflight-FieldKit.ps1`, `Rehearse-FieldKit.ps1`
Controller exports	`RS_3000_2024_APR_22.L5K` (plaintext, the **math** authority), `RS_3000_RUNNING_2026-06-09.L5K/.L5X` (**source-protected**)

Two standing cautions from `docs/PLC_FACTS.md` apply to every script in the capture families: a CompactLogix caps at roughly four to eight EtherNet/IP sessions per source IP, so too many parallel scripts will **starve Ring's own pollers** (the condition clears itself about thirty seconds after they exit); and the running export is source-protected, so **live tag observation is the authority** for what the running program does, not static L5K reading.

`plc-discovery/` holds the EtherNet/IP discovery helpers and the legacy-extract outputs (`Find-EipDevices.ps1`, `Get-EipIdentityTcp.ps1`, the recipe CSVs and the plant snapshots). `tools/ab_server/` is the bundled libplctag AB simulator — a fake PLC for dev, and the thing that makes `PlcWriteEndpointGuard` necessary (chapter 04 §4.4).

Next: 10 — Extending Ring.

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring/Ring.csproj` · `scripts/New-ReleasePackage.ps1` · `scripts/Test-ReleasePackage.ps1` · `scripts/Invoke-ProductionGate.ps1` · `scripts/` (full listing) · `remote-view/` (listing) · `plc-discovery/` (listing) · `tools/` (listing) · `Ring/Infrastructure/Configuration/ConfigurationValidator.cs` · `Ring/Config/appsettings.json` · `Ring/Infrastructure/Configuration/AppSettings.cs` (`AlarmEscalationSettings.Mode` default) · `Ring.Tests/WriteSurfaceRegisterTests.cs` ·

`docs/PLC_FACTS.md` · `docs/INDEX.md` · `CONTRIBUTING.md` · `Ring/Services/Reports/ReportSchedulerService.cs` (:87-98, `%ProgramData%\Ring\ScheduledReports` default) · `Ring/Services/PLC/PlcCommunicationLogService.cs` (:74, `%ProgramData%\Ring\logs` default)

10 — Extending Ring

Who this is for. Anyone adding something. Each recipe is the *shortest change that is complete* — the file list where a missing step fails silently, or fails on the plant floor rather than on your machine.

What you'll learn. How to add a screen, a service, a localized string, a config key, a database table — and, most importantly, the **only sanctioned way to add a PLC writer**.

Before every recipe: this is a non-SDK project. Every new `.cs`, `.xaml` and image must be hand-registered in `Ring/Ring.csproj` or `Ring.Tests/Ring.Tests.csproj`, or `ProjectMembershipGuardTests` fails (chapter 08 §8.2). That step is assumed in every list below, but it is the one people forget.

10.1 Adding a screen

1. **Choose the tier.** Live PLC data → a ViewModel with `INotifyPropertyChanged` and real bindings (tier 1). A report or setup screen → match the surrounding code-behind style (tier 2). **Do not invent a third pattern** (chapter 06 §6.1).
2. **Create the XAML + code-behind** under the right area of `Ring/Views/`. Register **both** in `Ring/Ring.csproj`:

```
<Page Include="Views\Setup\MyNewScreen.xaml">
  <Generator>MSBuild:Compile</Generator><SubType>Designer</SubType>
</Page>
<Compile Include="Views\Setup\MyNewScreen.xaml.cs">
  <DependentUpon>MyNewScreen.xaml</DependentUpon>
</Compile>
```

3. **Follow the canonical `Loaded` shape** for tier 2: constructor calls `InitializeComponent()` and nothing else; `Loaded` resolves dependencies once via `ServiceLocator` inside a `try/catch`, holds the references, and populates. Do not sprinkle `ServiceLocator` calls through the file.
4. **If it is a hosted screen, do not give it a title row.** `ReportHostView` paints the header. Twenty-three files carry a comment recording that their banner was deleted to fix a double header, and **nothing enforces it** (chapter 06 §6.2).
5. **Wire navigation** through `NavBar`, using `ReportHostView.ShowHostedLocalized(context, titleKey, subtitleKey, content)` for a localized header. If the screen is cached, use `NavBar.GetOrCreateView<T>()` and remember the instance is **retained** — reset per-session state in `Loaded`.
6. **`FitHost`?** Only for structural screens (cards, mimics, forms). **Never wrap a virtualized `DataGrid` or a long list** — measuring at unbounded height realizes every row. Data-table screens keep inner scroll.

7. **Strings, sizes, touch.** Every user-visible string is a resource key in all thirteen dictionaries (§10.3). Use the type-scale styles and tokens from `Ring/Resources/Tokens.xaml`, and respect `MinTouchTarget = 44`.
8. **Watch the rename freeze.** `UiScreenResolver` writes `GetType().Name` into the hashed UI-audit payload, and `MainWindow.NavigateBack()` branches on the `Ring.Views.BatchStart` namespace string (`ARCHITECTURE.md`).
9. **If the screen dispatches a PLC write, it needs a write-surface register row** — go to §10.6 instead of stopping here.

10.2 Adding a service

1. **Put the logic in a WPF-free, I/O-free type if you possibly can.** That is the house pattern and the reason so much of `Ring/Services/` is pure calculators and pure decision functions. A calculation embedded in a code-behind is a calculation with no test (chapter 07 §7.12).
2. **Design for the missing input.** Return `Unknown` / `NotConfigured` / `Unavailable` rather than a confident zero. `MaintenanceStatus.Unknown`, `GiveawayCostState.NotConfigured`, `ReportDataUnavailableNotice`, `TankCapability.Unknown`, `HasAnyData` on the snapshots — all of these exist for that reason. Follow them.
3. **Register it in `AddRingServices`** (`Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs:21`), as a singleton unless you have a reason. Repositories are registered there with a factory that pulls the connection string from `ConfigurationService`; copy the surrounding shape.
4. **Inject through the constructor wherever there is one.** From XAML-constructed code-behind, resolve once in `Loaded` via `ServiceLocator`.
5. **Do not add a new static `*State` class or a new `Instance` singleton.** That is the pattern being retired, and new instances make the retirement harder (chapter 02 §2.6).
6. **If it owns a timer**, follow the poller conventions: a `System.Threading.Timer` (not a `DispatcherTimer`) off the UI thread, an `Interlocked` busy flag so a slow cycle skips rather than stacks, and an explicit `Start`/`Stop`.
7. **If it writes SQLite on a timer, add it to `DatabaseWriterQuiesce`.** That type exists because a duplicated, incomplete list of writers left four services running through a database restore (chapter 05 §5.5).
8. **If it reads the PLC on a timer, know that you are adding to the set of timers outside the polling coordinator** — and that the controller caps EIP sessions per source IP. Prefer consuming an existing snapshot holder; `ProcessHistorianService` is the model (60 s, snapshot-only, heartbeat-gated).
9. Register the `.cs` in the csproj; add tests.

10.3 Adding a localized string

1. **Add the key to all thirteen dictionaries under `Ring/Resources/Strings/`, in one commit.** `en`, `nl`, `de`, `fr`, `es`, `it`, `pt`, `tr`, `pl`, `ko`, `ja`, `zh-Hans`, `zh-Hant`. A per-locale parity test fails otherwise, and it checks **placeholder parity** (`{0}`/`{1}`) as well as key parity.

2. **Never create a duplicate `x:Key`.** WPF throws while loading the dictionary — before the first frame — so the app dies at startup with **no window and a bare crash log**. This is the single most expensive mistake in this recipe.

3. **Do not change the merge order in `App.xaml`.** English is merged last as the permanent base fallback.

4. **Save UTF-8 without BOM.** If the mojibake test fires, repair the encoding (read the bytes back through the cp1252 round-trip) — **do not retype the character**, which leaves the underlying double-encoding in place.

5. **For a string composed in C#**, remember XAML cannot repaint it: subscribe to `LocalizationService.LanguageChanged`, or use the house `LocalizedOrFallback(key, englishFallback)` pattern (`ConfigurationValidator.cs:517-524`) which reads `Application.Current?.TryFindResource(key)` and falls back to the English literal so a test host with no `Application` still works.

6. **For an operator-facing refusal message on a safety path**, follow the convention used by the commissioning holds: expose a `…ResourceKey` **and** a `…Fallback` constant side by side, and resolve the two **together** so they can never name different causes (`RosterWriteIndex.StorageWriteRefusal`, `RosterWriteIndex.cs:298-314`, is the worked example — and the comment there records the bug that made it necessary).

Alarm **operator text** is the known exception: it lives in SQLite in English only, and localizing it is a schema change, not a dictionary edit.

10.4 Adding a config key

1. **Add the property to the right settings class** in `Ring/Infrastructure/Configuration/AppSettings.cs`, **with a C# default that is the safe value.** For anything touching the PLC that means fail-closed. Write a real doc comment: what it gates, why the default is what it is, and what must be true before someone changes it. The `Enable*` comments in `PlcSettings` are the standard to match.

2. **Decide whether it ships in `Ring/Config/appsettings.json`.** Leaving it out is a deliberate choice that makes the C# default the contract — and that is what the shipped file does for seven of the eight write-gate flags (chapter 04 §4.2). If you do ship it, remember the `CopyAppSettingsToOutput` target overwrites the output copy after every build.

3. **Add validation** in `ConfigurationValidator` if a wrong value is dangerous or merely confusing. Errors are **fatal** (`IsFatal => Errors.Count > 0`, `:23`) and block boot forever; warnings go to the 60-second countdown dialog. Choose deliberately — a warning that should have been an error boots a plant with a bad setting at 3 a.m.

4. **Beware the array index-merge.** `Microsoft.Extensions.Configuration` merges JSON arrays *by index*. If your key is an array and an overlay is expected to replace it, add it to `ConfigurationService.ApplyLocalArrayOverrides` alongside `AlarmEscalation.Smtp.To` and `AlarmEscalation.Webhook.AllowedHosts`.

5. **If it is a write gate, pin it.** Add an assertion to `Hardware_signoff_write_capabilities_remain_fail_closed_in_the_shipped_settings` (`Ring.Tests/WriteSurfaceRegisterTests.cs:1053`) — both the compiled default *and* an absent-or-false check against both shipped JSON files.

6. **Document it in** `docs/reference/plc/APPSETTINGS_REFERENCE.md` — type, default, plant-specific or not, and what breaks if it is wrong. That file is the authority for "what does this key do"; this book is not.

7. **Config is restart-scoped.** `reloadOnChange` is `false` on both files, and nothing watches them. The exception, worth knowing rather than copying, is `PlcConnectionConfig`, which re-reads the file per call (chapter 02 §2.3).

10.5 Adding a database table — the complete checklist

This is the recipe with the most places a partial change *silently succeeds*, so it is given in full here rather than by reference. Every step was verified against `Ring/Database/DatabaseInitializer.cs` and the tests named.

1. Write the interface and the repository. `Ring/Database/Interfaces/IXRepository.cs` and `Ring/Database/Repositories/XRepository.cs`. Copy the shape of an existing pair: no base class, a connection opened **per call**, and `RingwoodDbAccess.ApplyConcurrencyPragmas(connection)` immediately after `Open()`.

2. Expose the schema as a static, and have the instance use it.

```
public static void ApplySchema(SQLiteConnection connection) { /* CREATE TABLE IF NOT EXISTS … */ }
public void EnsureTable() => /* … */ ApplySchema(connection);
```

That static is the **single source of DDL truth**, shared between first-run creation and migration. If the two ever diverge, a fresh database and an upgraded one end up with different schemas.

3. Append a `MigrationStep` at the END of `GetMigrationSteps()` (`DatabaseInitializer.cs:361`). The shape is `new MigrationStep(version, "description", conn => XRepository.ApplySchema(conn))` (`:331-340`). Steps run inside a SAVEPOINT and the version is stamped atomically, so the step body must be **idempotent** — `CREATE TABLE IF NOT EXISTS`, guarded `ALTER`.

4. Bump `CurrentSchemaVersion` (`DatabaseInitializer.cs:743`) to your new step's version.

> **The trap, and it is worth reading twice.** If you author the step as `new MigrationStep(CurrentSchemaVersion, …)` and forget to bump the constant, you get **two steps with the same version**. Every test still passes — a fresh database runs both steps in order — but a plant database that is *already at that version* **silently skips your migration**, because migrations only apply where `step.Version > user_version`. Check the number by eye; nothing catches this one.

Three tests *do* catch a forgotten bump in the ordinary case: `DatabaseSchemaMigrationTests.CurrentSchemaVersion_Matches_HighestMigrationStep` (`:47`), `DatabaseExpectedTablesDriftTests.CurrentSchemaVersion_MatchesInitializerLatest` (`:120`), and `DatabaseSchemaMigrationTests.Wave6Migrations_CreateBothTables_AndStampCurrent`, which asserts the version as a **literal** (`DatabaseSchemaMigrationTests.cs:139-140`) — so that literal has to move with the constant. Expect to edit it.

5. Add the table name to `ExpectedTables` in `Ring/Services/DatabaseBackupService.cs:43`. This is the **restore contract**: a backup stamped at `CurrentSchemaVersion` or newer must contain the full table set before it is allowed to overwrite a live database (`:496`). Miss this and a valid backup is rejected at restore time — on a plant, under pressure.

`Ring.Tests/DatabaseExpectedTablesDriftTests.ExpectedTablesExactlyMatchesWhatInitializeActuallyCreates` (`:93`) runs the real initializer against a temp database, reads `sqlite_master`, and asserts a **symmetric** set-difference — so it fails in *both* directions if this step and step 3 disagree. Do not "fix" a red run by editing only one side.

You do **not** normally add to `CoreTables` (`DatabaseBackupService.cs:144`), which is the smaller must-have set applied to *older* backups.

6. Mirror the name in the `AllTables` literal in `Ring.Tests/DatabaseBackupServiceTests.cs:38`, which builds the full fixture database.

7. Register the repository in `AddRingServices` (`Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs`), following the surrounding factory shape that pulls the connection string from `ConfigurationService`.

8. Register both new `.cs` files in `Ring/Ring.csproj`, or `ProjectMembershipGuardTests` fails — and until it does, the compiler never sees them.

9. Consider the failure semantics. If the repository returns lists, its catch blocks should call `RepositoryReadScope.ReportFailure` so a failed read can be told apart from "no rows" (chapter 05 §5.2). A report that renders a failed read as a clean zero is a defect this codebase has already paid for once.

10. If a background timer will write this table, add it to `DatabaseWriterQuiesce` so a database restore or factory reset cannot race a tick into the file being replaced (chapter 05 §5.5).

10.6 Adding a PLC writer — the only sanctioned route

> **Stop here if you have not read chapter 04.** This recipe is a summary of that chapter, not a substitute for it. From the 2026-09-08 cutover, a writer added wrongly moves a valve.

Step 0 — establish that the write is allowed at all

Before any code: is this tag, on this tank, with this value, something a controls engineer has signed for? The record is `docs/production-readiness/CONTROLS_SIGNOFF_RECORD.md`; open questions are in `docs/production-readiness/CONTROLS_QUESTIONS_2026-07-28.md` and its 2026-08-26 successor. If the answer is "not yet", the correct outcome is a **fail-closed authorization class** like `TvcCoolingWriteAuthorization` — shipped inert, with the screen rendering an honest explanation — not a writer that works and a flag you plan to add later.

Also verify the tag itself against `docs/PLC_FACTS.md` before you trust any string:

- UDT members are read/written by **byte offset**, and bit-packed SINT members are exposed under generated `ZZZZZZZZZZ…` alias names. Grep the L5K `DATATYPE` block for the verbatim name.
- Per-tank capability is real: `enabl_agt` / `enabl_tvc` are true for tanks 1, 2, 8 and 9 only, so a tag can exist on both controllers and be inert on one.
- `Tank_IO[N].O_Agitat` is **not a command bit on the doser tanks** — `ST_ROUTINE UD_Ctrl` overwrites it from a remote-I/O status word within about four scans.
- Bench evidence does not automatically transfer to the plant.

Step 1 — route it through the gate stack

Prefer funnelling through `PlcTagWriter`, which gives you ReadOnly, the live demo re-check, the endpoint gate and the heartbeat gate for free (`Ring/Services/PLC/PlcTagWriter.cs:135-220`). Add your own **outer** heartbeat check at the service so a pulse cannot *start* on a degraded link.

If you must build a raw libplctag `Tag`, the prologue shape is `BatchStartTankPlcWriter.cs:74-81`:

```
if (PlcWriteGuard.IsReadOnly)
{
    PlcWriteGuard.LogBlocked(op, tagName, value, logger);
    return /* the honest outcome for your rail */;
}
```

…and you must then add the demo, endpoint and heartbeat checks yourself, in that file, before the `Tag` is constructed.

Choose the return shape deliberately. A suppressed write is not a successful one. Return `false`, or a typed outcome, unless you have the same reason the "fake success" paths have — that the operator flow must continue and the operator is told separately that nothing was sent (chapter 04 §4.1).

If it is a momentary bit: assert with `Write`, clear with `MomentaryPulse.ClearWithRetry(() => writer.WriteClear("False"), …)`, and add your file to the approved `WriteClear` call-site list in `Ring.Tests/MomentaryPulseClearGateTests.cs:757-765` — *deliberately*, never to make a build green. If

the bit shares `PC_Write_Integer[30]`, take the shared word gate via `PcWriteInteger30BatchControlPlcService.RunUnderWord30Gate`.

If it is a level-valued setpoint from a screen: dispatch through `PlcSetpointWriteQueue.Instance.Submit(...)` with a key built by `PlcWriteTagKeys`, and seed/guard the control with `SetpointEchoGuard` (`Reset()` in `Loaded`, then re-seed). **Never** put a momentary/pulse bit on the queue — latest-value-wins would delete the clear half.

If it targets a `Tanks[N]` member on a storage tank: resolve the index through `RosterWriteIndex`, *below* any test seam, and refuse on `NoWrite` and on both sticky locks. Never compute `uiSlot + 1`.

If it targets a per-tank member at all: consult `TankInstalledWriteGate`, which fails closed on `Unknown`.

Step 2 — add a per-feature `Enable*` flag

New write families ship **off**. Add a `bool` to `PlcSettings` defaulting `false`, with a doc comment stating what evidence would justify turning it on (§10.4 step 1), and leave it out of the shipped JSON so the C# default is the contract.

Step 3 — add the write-surface register row

In `Ring.Tests/WriteSurfaceRegisterTests.cs`:

```
{ "Services/PLC/MyNewWriter.cs", WriteClass.UdtMember },  // Tanks[N].My_Member
```

- Path is relative to `Ring/`, forward-slashed.
- **Pick the truthful `WriteClass`** — it generates the proof matrix's BENCH-OUTSTANDING column, so a mis-filed row turns a bench-only claim into a "proven" one.
- **Raise the count floor in `The_register_is_not_silently_shrinking` in the same commit.** The floor is set to the exact current count with zero slack on purpose.
- If your writer is a UI dispatcher that only *calls* a service, and it is invisible to the five signals, **widen signal 5's alternation** rather than planting a decorative `PlcWriteGuard` reference — and add the corresponding case to `The_dispatcher_entry_point_detector_still_detects`.

Step 4 — add a seam test

A test that pins **tag, value, count and order** for your writer. Give the writer an injectable seam (`Func<string, string, bool>`, `Func<string, short, bool>`, or an `I…PlcIo` interface) so the test can observe the wire without a controller. If you do not, your writer joins the three named, bounded, seam-less writers and `The_writers_that_reach_libplctag_without_a_test_seam_are_named_and_bounded` will fail until someone consciously adds it to that list (`WriteSurfaceRegisterTests.cs:887-931`) — which is the point.

Step 5 — bench evidence, for a UDT member or a BOOL bit

- A bench round-trip in `Ring.Tests/BenchPlcIntegrationTests.cs` (or `BenchPlcWriteRoundTripTests.cs`).

- A row in `docs/production-readiness/WRITE_PATH_PROOF_2026-07-30.md`.
- The commissioning evidence line in `docs/production-readiness/PLC_WRITE_COMMISSIONING_REGISTER_2026-08-02.csv`, which `scripts/Test-PlcCommissioningEvidence.ps1` verifies.

Remember the bench tests pass **vacuously** unless `BENCH_PLC_AVAILABLE=1`; run them with the variable set against the bench twin at `192.168.202.15`, and cite `benchHardwareWasAvailable` from the gate summary when you claim they ran.

Step 6 — review and gate

- **Independent review.** A write-path change is reviewed by someone who did not write it. Do not self-review (`CONTRIBUTING.md`).
- Run `scripts/Invoke-ProductionGate.ps1` before asking for a merge.
- Expect the **PLC Tag Audit** workflow to notice a new read/write against a tag with no ladder writer in the parsed 2024 export. Read `docs/PLC_FACTS.md` before dismissing it — and before believing it.

The two prohibitions

1. **Do not populate `MainTvcWriteAuthorization.IsAuthorized` or `TankTempModeWriteAuthorization.AuthorizedTankTempModeTanks`.** Both are hard-closed pending a controls decision; both are asserted by `WriteSurfaceRegisterTests`; both say so in capitals in their own source. "Making the screen work" is not a reason.
2. **Never add a UI guard or an early `return` in front of a shared handler that also dispatches PLC-write buttons.** Enumerate every button routed to the handler first. `NavBar.HandleButtonClick` (`Ring/Views/UserControls/NavBar.xaml.cs:415-444`) is the live example of how to do it correctly — two explicit exclusion sets, and a dedicated handler for the roster-generated tank buttons so they never travel through it at all.

10.7 Adding a document

Small, but it is how this book stays worth reading.

1. **Check `docs/INDEX.md` first.** If a document already owns the question, add to it rather than forking it. One authority per question.
2. **Cite the file you read.** A claim without a path is a claim nobody can check, and this repository has been burned by exactly that.
3. **Never copy a volatile number into prose.** Test totals, tag counts, table counts, dead-read counts belong in `artifacts/production-gate/production-gate-summary.json` and `scripts/code-tag-audit.md`. Link, do not quote.
4. **Say "unknown" when it is.** A document that says "I could not verify this" is worth more here than one that guesses (`CONTRIBUTING.md`).
5. **`docs/archive/**` is evidence, not instruction.** Never quote its status claims as current.
6. **If the code contradicts a document, fix the document in the same pass** — or archive it.

7. **Generated files are never hand-edited:** `scripts/live-tags.{md,json}`, `scripts/code-tag-audit.{md,json}`. The gate diffs them and fails on drift.

10.8 A short checklist for any change

- ☐ New files registered in the csproj
- ☐ CRLF preserved; csproj/sln still BOM-free; diffed before commit
- ☐ New user-visible strings in **all thirteen** dictionaries, no duplicate `x:Key`
- ☐ Tests added; suite run with `-parallel none`
- ☐ `scripts/Invoke-ProductionGate.ps1` green
- ☐ Any PLC write path: gate stack, register row + floor, seam test, bench evidence, independent review
- ☐ Working in a worktree; branch verified after committing
- ☐ Documents cite the files they were read from; no volatile numbers in prose

---

Back to: book index · chapter 04, the one to re-read

Verified against

Every claim in this chapter was read out of these files on 2026-09-01:

`Ring/Ring.csproj` · `Ring/Infrastructure/DependencyInjection/ServiceCollectionExtensions.cs` · `Ring/Infrastructure/Configuration/AppSettings.cs`, `ConfigurationService.cs`, `ConfigurationValidator.cs` · `Ring/Database/DatabaseInitializer.cs`, `RingwoodDbAccess.cs`, `RepositoryReadScope.cs` · `Ring/Services/DatabaseBackupService.cs` · `Ring/Services/DatabaseWriterQuiesce.cs` · `Ring/Services/PLC/PlcWriteGuard.cs`, `PlcTagWriter.cs`, `BatchStartTankPlcWriter.cs`, `MomentaryPulse.cs`, `PcWriteInteger30BatchControlPlcService.cs`, `PlcSetpointWriteQueue.cs`, `PlcWriteTagKeys.cs`, `SetpointEchoGuard.cs`, `RosterWriteIndex.cs`, `TankInstalledWriteGate.cs` · `Ring/Views/UserControls/NavBar.xaml.cs` · `Ring/Views/Reports/ReportHostView.xaml.cs` · `Ring.Tests/WriteSurfaceRegisterTests.cs`, `MomentaryPulseClearGateTests.cs`, `ProjectMembershipGuardTests.cs`, `DatabaseSchemaMigrationTests.cs`, `DatabaseExpectedTablesDriftTests.cs`, `DatabaseBackupServiceTests.cs` · `docs/PLC_FACTS.md` · `docs/INDEX.md` · `CONTRIBUTING.md`

11 — Diagnostics and Field Triage

Who this is for. Anyone standing in front of a station that is behaving strangely, with limited time and no appetite to re-read sixty thousand words of subsystem documentation to find the one paragraph that explains it.

What you'll learn. Nothing new. Every mechanism named below is documented in full elsewhere in this book — this chapter is an index *by symptom* into chapters that are organised *by subsystem*. Find what you are seeing, jump to the section named, and come back here for the next symptom. Every symptom/cause pair below was checked against the current source before it was written; this repository's own history is that roughly two-thirds of unverified doc claims turn out to be phantom, and a troubleshooting chapter that guesses is worse than none.

The operator-facing version of this table is `docs/manual/operator/07-troubleshooting.md` — read that first if the person in front of the station is an operator, not an engineer. This chapter goes one level deeper: the mechanism, the file, and the line.

11.1 Symptom index

Symptom	First thing to check	Mechanism, explained in	Escalation
Screens frozen, values not updating, but the link badge is green	Is *one* screen frozen or *all* of them? One screen: check whether its own `DispatcherTimer` is still ticking (a swallowed exception can silently stop it). All screens, UI unresponsive to clicks: this is a UI-thread hang, not a PLC problem — see §11.2	ch. 02 §2.5 (the UI-hang sentinel); ch. 03 §3.4 (freshness gating)	Setup → Diagnostic Console → Export Bundle; if the UI is truly unresponsive, a support bundle cannot be taken from inside the app — collect `%ProgramData%\Ring\logs\application.log` and a screenshot instead
A "stale" chip or dimmed value appears on **many tank cards at once**, not just one	The heartbeat state, not the freshness window — one heartbeat problem starves every heavy poller at once, so it looks like a blanket outage rather than one broken tag	ch. 03 §3.6 (`Starved` vs `Disconnected`); §11.3	PLC Health drawer for the exact state; if `Starved`, this is very likely `MainTask InhibitTask := Yes` on the controller pre-cutover
The link badge is flapping between colours, or reads a state you don't recognise	Which of the five states it actually is, and for how long	ch. 03 §3.6 — the state machine, its four numeric thresholds, and why `Starved` is a separate diagnosis from `Disconnected`	If it settles within ~12 s, this is expected transient behaviour, not a fault

Symptom	First thing to check	Mechanism, explained in	Escalation
An operator presses a button and nothing on the controller changes, with no error shown	What the write path *returns* is not uniform — several rails report `true`/success even when the write was suppressed	ch. 04 §4.1, "What a blocked write returns is not uniform"; §11.4	Grep the application log for `SUPPRESSED` around the time of the click (ch. 07 §7.13) before assuming the write failed at the wire
Hold cannot be resumed from any station	`PC_Write_Integer[30].1` is a **sealed coil** — level-sensitive, not edge-triggered — so a stuck TRUE re-asserts Hold every scan	ch. 04 §4.6 (the F013 exemption and the stuck-bit table); §11.5	`PcWriteInteger30StrandedBitMonitor`'s advisory on the PLC Health drawer names the stuck bit; if it does not clear, the bit must be cleared from Studio 5000
Ring appears not to launch — no window, or a window flashes and closes	A second instance may already be running, or classified as still starting	ch. 01 §1.7; §11.6	Task Manager → check for an existing `Ring.exe`; if genuinely absent, check `%ProgramData%\Ring\logs\application.log` for a startup error
PLC comms degrade while a capture/probe script is running	EtherNet/IP session exhaustion — a CompactLogix caps at roughly 4–8 sessions per source IP, and Ring's own seven pollers already use several	ch. 01 §1.0 Vocabulary, "EIP session"; ch. 03 §3.9	Stop the extra script; the condition self-heals in roughly 30 s once the extra sessions close
PLC IP address, `ReadOnlyMode`, SMTP credentials or cost rates revert after a rebuild	`CopyAppSettingsToOutput` unconditionally overwrites the output `Config\appsettings.json` from the source tree after every build	ch. 02 §2.3, "Configuration is written back into the build output"; ch. 09 §9.1	Re-apply the local overlay, or keep it in `Config/appsettings.local.json` where the Debug-only copy step (and the Release keep-and-warn behaviour) actually preserves it
Database writes seem blocked; export, backup or factory reset seem to hang or silently do nothing	The maintenance latch is held — a restore or reset is in progress, or ended abnormally	ch. 05 §5.5; §11.7	Confirm no restore/reset dialog is actually still open behind another window before assuming a hang; the latch is process-wide, not per-window
Thousands of test failures appear from nowhere on a machine that was fine yesterday	Free disk space, checked **first**, not last — parallel build lanes filling `C:` have produced exactly this failure mode before	ch. 08 §8.1	`Get-PSDrive C`, or the Windows equivalent, before opening a single test's stack trace

Symptom	First thing to check	Mechanism, explained in	Escalation
A red banner appears under the top of the window at startup and stays for the session	"Degraded" has one precise meaning: database initialization failed, and `PlcWriteGuard.ForceReadOnlyForSession()` was pulled — a one-way latch for the rest of the session	ch. 02 §2.4; §11.8	Read the banner text exactly — it names the reason. The recovery is a restart, not a retry from inside the running session
A supervisor edited a config value through the app and it does not seem to be sticking, or a build immediately reverted it	Two different mechanisms depending on which: the atomic-write helper failing silently is rare and would log an error; the far more common cause is the build-output copy above	ch. 02 §2.3 (writer classes); same row as the `CopyAppSettingsToOutput` symptom above	Check `%ProgramData%\Ring\logs\application.log` for a write error first; if none, it is almost certainly the build-output revert
A write-path PR looks correct but you are not sure what it owes	Not a live-station symptom, but the single most common "something feels wrong" moment in review	ch. 04 §4.12 and the quick reference at the top of ch. 04	—

11.2 Screens frozen, but the link badge is fine

This is two different failure modes wearing the same description, and the fix is to tell them apart before doing anything else.

If it is one screen, and every other screen is fine: the heartbeat and the pollers are healthy, so this is local to that screen's own `DispatcherTimer`. Every PLC-facing screen owns a `DispatcherTimer` (typically 1000 ms) that reads a static snapshot holder and paints named controls (ch. 02 §2.5). An unhandled exception inside a tick handler can, depending on where it is thrown, stop that timer without crashing the app. Navigating away and back re-creates the screen (or re-seeds a cached one via its `Loaded` handler) and is the fastest recovery; if it recurs on the same screen, that screen's own tick handler is the place to look, not the PLC link.

If the whole app is unresponsive to clicks, not just showing stale numbers: that is a UI-thread hang, and it is a different mechanism from freshness gating entirely. `MainWindow` runs a background sentinel that pings the dispatcher and expects a "pong" every 5 s (`UiHangPingIntervalMs`, `MainWindow.xaml.cs:122`), feeding `Ring/Services/UiHangDetector.cs`, which classifies the UI thread's state from that single tick. A station in this state cannot take its own diagnostic bundle — Setup → Diagnostic Console needs a responding UI — so the fallback is to collect `%ProgramData%\Ring\logs\application.log` directly and note the exact time of the hang.

Neither of these is what freshness gating (ch. 03 §3.4) is for. Freshness gating governs whether a **value that arrived** is painted as current or refused as stale; it says nothing about whether the screen's own timer is still running at all.

11.3 "Stale" everywhere at once, vs. stale in one place

A dimmed value or a "stale" caption on **one** card, next to fresh values on its neighbours, is ordinary freshness gating doing its job — see ch. 03 §3.4.

A "stale" indication on **every** tank card, every group screen, at the same time is a different signal: it means `PlcHeartbeatConnectionTracker.AllowHeavyPlcPolling()` has gone false, which every heavy poller gates on (ch. 03 §3.2). Check the PLC Health drawer for the exact state:

- **`Stale`** (4–12 s without a heartbeat change) does not stop heavy polling — this state alone should not blank every screen.
- **`Starved`** (12 s+ without a heartbeat change, while reads still succeed) does. This is the diagnosis operators need: the network is fine, Ring is fine, and the controller is not advancing the heartbeat word. The usual pre-cutover cause is `MainTask InhibitTask := Yes` on the controller.
- **`Disconnected`** (12 s of no successful read, or three consecutive failures) is the more familiar full comms-loss case.

`Starved` and `Disconnected` look similar from the operator's chair — "the plant stopped updating" — but they are different diagnoses with different next steps, which is exactly why the state machine keeps them separate (ch. 03 §3.6).

11.4 A write silently does nothing

Before assuming the write reached the controller and was ignored, or never reached it at all, read ch. 04 §4.1, "What a blocked write returns is not uniform". Some rails return `true` on a suppressed write specifically so the operator flow continues and the operator is told separately that nothing was sent — `true` is not proof of a landed write anywhere in this codebase.

The fastest way to tell suppression from silence is the log, not the screen: every suppressed write goes through `PlcWriteGuard.LogBlocked`, which emits a line containing the literal word `SUPPRESSED` (ch. 07 §7.13 has two real examples of the exact line shape). If nothing in the log around the time of the click contains `SUPPRESSED`, the write was not blocked by the read-only gate — check the per-feature `Enable*` flag (ch. 04 §4.2), the applicable commissioning hold (ch. 04 §4.3), and the endpoint and heartbeat gates in that order, per the verified per-rail table in ch. 04 §4.7.

11.5 Hold cannot be resumed

`PC_Write_Integer[30].1` (Hold) drives a **sealed coil** — the rung is level-sensitive, not edge-triggered, so a bit that is stuck TRUE re-asserts Hold on every scan and Resume cannot break the seal from any station, including the PanelView and the hardwired inputs (ch. 04 §4.6). This is precisely why F013 exists: the clear (`PlcTagWriter.WriteClear`) is exempted from the heartbeat gate specifically so a degraded link cannot prevent Ring from at least attempting to clear the bit.

`PcWriteInteger30StrandedBitMonitor` is the read-only sentinel that watches for exactly this — it raises a throttled advisory on the PLC Health drawer naming the stuck bit, with operator-facing text along the lines of "PC HOLD request bit is stuck: press Hold, then Resume. Until it clears the plant re-enters Hold every scan and cannot be resumed from any station." It **only detects and advises; it never clears the bit itself.** If pressing Hold then Resume does not clear it, the bit has to be cleared from Studio 5000 directly — there is no software recovery from inside Ring for a truly stuck sealed coil.

11.6 Ring appears not to launch

Ring is single-instance, but "not launching" has more than one cause. `App.xaml.cs` creates a named mutex (`"RS360-RingwoodApp-SingleInstance"`, no `Global\` prefix — see ch. 02, "The single-writer problem, stated honestly" for why that prefix's absence matters for remote-access seats specifically). On a second launch, Ring classifies the existing instance rather than simply assuming it is safe to kill:

- **Already starting** — the first instance's own boot marker is still held, which means it is genuinely mid-`OnStartup` (possibly running its own integrity check and database backup). The second launch exits quietly without disturbing it. **Do not kill the first instance in Task Manager in this state** — that is the case this classification exists to prevent.
- **A responsive window was found** — it is focused, and the second launch exits.
- **No window at all, or the window failed two liveness probes five seconds apart** — the second launch treats the first as genuinely stuck and may tell the operator it is safe to end it.

If Task Manager shows no `Ring.exe` at all and nothing appears, check `%ProgramData%\Ring\logs\application.log` for a startup-time error — a broken `appsettings.json` produces a message box **before** the main window, which can look like "nothing happened" if the box is behind another window.

11.7 Database writes seem blocked

`DatabaseMaintenanceLatch` is a process-wide "the database file is being swapped or deleted right now" flag, held by `DatabaseWriterQuiesce.StopAll()` during a restore or a factory reset and released only by `ResumeAll()` (ch. 05 §5.5). It exists specifically because a modal message box around the swap runs a nested dispatcher pump, and two things can otherwise undo the quiesce mid-swap (the PLC auto-reprobe, and the dashboard's own refresh ticks) — so the latch is a belt-and-braces stop on top of the quiesce, not a redundant one.

If exports, backups or writes seem stuck, first confirm there genuinely is no restore or reset dialog open behind another window — the latch is never released on a path that ends in application shutdown, so an operator who closed Ring mid-restore rather than letting the dialog finish can leave a stale expectation about what state the database is in, even though the latch itself cannot outlive the process.

11.8 The startup degraded banner

"Degraded" has exactly one meaning in this codebase: **database initialization failed**, so batch history, the audit trail and the alarm log are unreliable for the session (ch. 02 §2.4). The response is not just a banner — `StartupDegradedState.ApplyDatabaseInitOutcome(false, …)` also pulls `PlcWriteGuard.ForceReadOnlyForSession()`, the one-way latch: once pulled, nothing in the running session can re-arm writes short of a restart, because `Configure` computes `_readOnly = readOnly || _forcedReadOnly` and nothing clears `_forcedReadOnly` in production code (that reset method exists only for test cleanup — see ch. 08 §8.1 and the tripwire test in ch. 08 §8.6).

Read the banner text exactly — the class records the reason first-detail-wins, so the wording tells you which of the startup failures actually happened. The recovery is a restart, not a retry from inside the session: a successful database init on the next boot changes nothing else, and the configured `ReadOnlyMode` resumes deciding normally.

A milder, related degradation exists for the tank roster specifically: if the roster file fails to load, startup falls back to the legacy 4-tank default, warns, **and** latches `RosterWriteIndex.StorageWritesLockedByRosterLoadFailure` even though the substituted roster looks perfectly valid — see ch. 02 §2.4 and ch. 04 §4.3, "The roster locks".

11.9 The first five minutes

In roughly the order that costs the least time:

1. **Read the banner or dialog text exactly**, if one is showing. Both the degraded-startup banner and the countdown warning dialog name their own cause; do not paraphrase from memory before checking the literal text.
2. **Open the PLC Health drawer.** It carries the live heartbeat state, the stranded-bit advisory (if any), and the clock-skew warning — three different sentinels in one place. (ch. 03 §3.6, §11.5 above, ch. 03 §3.7.)
3. **Check the comm log** (Setup → `DisplayCommunicationForm`) for the specific tag and time in question — it shows suppressed writes as well as real ones, marked `"ReadOnly suppressed"`. (ch. 03 §3.7.)
4. **Grep `%ProgramData%\Ring\logs\application.log`** for `SUPPRESSED` (a blocked write), `ERROR` (most failures worth escalating), or the specific tag name. Remember the log's own retention limit at `Debug` level is short (ch. 07 §7.13) — the further back you need to look, the less likely it is still there.
5. **For a write that may have reached the controller**, check the write evidence journal (`%ProgramData%\Ringwood\Ring\plc-write-evidence.jsonl`) — but remember its scope: only writes dispatched through `PlcSetpointWriteQueue` are recorded there. A batch-start write, a `PC_Write_Integer[30]` pulse, a formula-bank rewrite or an inventory write leaves no line in this file; its record is the application log, the comm log, and its own typed outcome (ch. 04 §4.9).
6. **For a build or test-suite anomaly**, check free disk space before opening a single failing test — see ch. 08 §8.1.

7. **If none of the above explains it**, `scripts/Invoke-ProductionGate.ps1`'s own summary (`artifacts/production-gate/production-gate-summary.json`) is the authoritative statement of what currently passes; do not trust a number quoted from memory over it.

Back to: book index.

Verified against

Every symptom/mechanism pair in this chapter was checked against these files on 2026-09-01, in addition to the chapters it cross-links:

`Ring/Services/PLC/PcReadIntegerSnapshot.cs`, `PlcHeartbeatConnectionTracker.cs`, `PlcWriteGuard.cs`, `PcWriteInteger30StrandedBitMonitor.cs`, `MomentaryPulse.cs`, `PlcCommunicationLogService.cs`, `PlcWriteEvidenceJournal.cs`, `FormulaBankWriteInterlock.cs` · `Ring/Services/DatabaseMaintenanceLatch.cs`, `DatabaseWriterQuiesce.cs` · `Ring/Services/StartupDegradedState.cs` · `Ring/Services/UiHangDetector.cs` · `Ring/Services/Logger.cs` · `Ring/Services/SecondLaunchAdvice.cs` · `Ring/Views/App.xaml.cs` · `Ring/Views/MainWindow.xaml.cs` · `Ring/Ring.csproj` (`CopyAppSettingsToOutput` target) · `docs/PLC_FACTS.md` · `docs/manual/operator/07-troubleshooting.md` · `CLAUDE.md` · `CONTRIBUTING.md`