AirfryerGuide

V1 PUBLIC EDITION

Expanded English Systems Engineering and AI-Readiness Case Study

Brownfield hardening, observability, governance, controlled change, and future human-in-the-loop intelligence

Publication boundary

This document was rebuilt from material considered suitable for public discussion. It is not a black-bar redaction of the internal full documentation. Production-specific identifiers, infrastructure details, defensive thresholds, rule chains, administrative paths, and reproducible attack or evasion mechanics remain excluded.

Project status: V1 accepted
Document baseline: 18 July 2026
Public edition: English Extended Release

How to read this document

> **Case study, not a security certificate**
>
> This is a systems-engineering narrative and an evidence-backed project record. It is not a penetration test, a formal assurance report, a guarantee against compromise, or a drop-in architecture blueprint.

- **Audience:** engineers, technical operators, indie builders, product owners, and readers interested in turning a organically grown web application into a documented and governable production system.
- **Included:** engineering principles, phase outcomes, evidence handling, recovery discipline, observability concepts, control-plane thinking, and future AI-readiness constraints.
- **Excluded:** reproducible attack, bypass, administration, infrastructure reconnaissance, or policy-evasion details.
- **Tone:** deliberately more platform-engineering and AI-governance heavy than the compact German edition, while keeping claims tied to the actual V1 baseline.

Contents

1. Executive summary - from content website to governed production platform

> **Outcome**
>
> After six controlled workstreams, the documented production baseline was accepted as V1. At the recorded cut-off there was no known pending deployment package, no reported rollback requirement, and no unresolved issue classified as a release-blocking technical condition.

The interesting part is not that a website received another round of maintenance. The interesting part is that a feature-rich, organically evolved application was treated as a small production platform: with state transitions, data lineage, observability, recovery evidence, authorization boundaries, decision provenance, and a written distinction between what was proven and what was merely believed.

In less inflated language: the project stopped relying on memory and intuition as its primary control system. In more inflated language: implicit operator knowledge was progressively externalized into a lightweight governance layer, turning a brownfield codebase into a more legible socio-technical system.

The originally planned observation window was terminated early by an operator decision. No sustained functional error pattern, no internal dependency on the deliberately removed components, and no recovery requirement were reported

during the period that was actually observed. This is intentionally not presented as a completed long-duration validation campaign.

What the V1 baseline includes

- A stronger privacy and operational baseline with reduced raw-data exposure.
- Documented deployment, backup, rollback, and smoke-test procedures.
- An inventoried persistence layer and an isolated restore validation of a complete backup state.
- A source-aligned description of the traffic telemetry, review, classification, and response pipeline.
- Hardened administrative actions with clearer authorization, request integrity, and audit events.
- Runtime-evidence-assisted legacy cleanup followed by regression checks.
- A transparent register of unresolved evidence gaps and known technical risk categories.

What the V1 baseline does not claim

- No claim of a fully completed long-duration observation test.
- No claim that one archive alone can recreate the entire application environment from zero.
- No claim of a real restore into the production environment.
- No claim that every retention rule is already enforced by complete automation.
- No claim that every tracking inconsistency or code-level anomaly has already been remediated.
- No independent verification of every external runtime, scheduler, provider, or infrastructure statement.
- No claim that an AI model is already making production traffic decisions.

2. Initial state - brownfield complexity and semantic debt

AirfryerGuide.de is a production web application with public content, search and calculation tools, user-facing features, administrative workflows, and a custom traffic-quality subsystem. Over time, protective rules, reporting views, background reviews, historical utilities, and operational shortcuts accumulated around the core product.

The risk profile was therefore less about one dramatic defect and more about compounded ambiguity. Several components could read or mutate overlapping state. Old files and tables were neither automatically useful nor automatically safe to delete. Some operational facts lived in operator memory instead of code or documentation. Static source analysis could reveal references, but it could not independently prove live traffic, external calls, stored-content dependencies, scheduler installation, or provider-side routing behavior.

The engineering problem in platform language

- **Brownfield architecture:** the system had evolved through real use rather than from a clean-sheet platform design.
- **Semantic debt:** status values, reasons, reports, and operational assumptions did not always carry one universally enforced meaning.
- **Hidden coupling:** a component could be referenced through runtime data, redirects, scheduled execution, stored content, or external requests even when no obvious source-level caller existed.
- **Observability gaps:** some behavior was inspectable in code, some only through data, and some only through operator or infrastructure evidence.
- **Change risk:** a locally reasonable edit could have non-local side effects across tracking, review, analytics, administration, and retention.

Program objectives

- Describe the current production state in a reproducible and reviewable way.
- Implement privacy and security improvements as controlled state transitions rather than ad hoc patches.
- Test backup recoverability in practice instead of treating the existence of a file as recovery proof.
- Map data flows, decision paths, side effects, and operational dependencies.
- Make sensitive administrative state changes attributable and auditable.
- Retire legacy assets only when multiple evidence sources supported removal.
- Maintain an explicit known-unknown register instead of quietly converting uncertainty into confidence.
- Prepare a clean semantic foundation for any later ML or AI-assisted traffic analysis.

3. Evidence architecture - separating runtime truth from plausible narrative

A central design decision was to treat documentation as an evidence system, not a prose layer. Every material statement should answer two questions: What is the claim? What kind of evidence actually supports it?

Evidence class	Meaning	Representative basis
Code/data confirmed	Directly derivable from the reviewed source tree, a confirmed data snapshot, or an observed test result.	Call graph, schema, writer/reader relationship, migration output, regression result
Operator or phase evidence	Documented production context that was not independently collected from the live target environment.	Runtime version, scheduler cadence, hosting topology, operational procedure
Open / unresolved	The available artifacts are insufficient for a final technical claim.	External configuration, full retention automation, every runtime-only branch

Why evidence classes matter

- Source code proves intended and statically reachable behavior, not the complete live infrastructure configuration.
- A confirmed database snapshot proves a specific state at a specific cut-off, not every change before or after it.
- A successful import proves that a backup can be parsed and reconstructed at the database layer; it does not prove the entire application stack will operate correctly against it.
- A missing static caller does not prove that an endpoint is unused; external traffic, redirects, browser bookmarks, stored HTML, and scheduled execution may still exist.
- A frequent traffic pattern is not automatically legitimate, and a rare pattern is not automatically malicious.
- A dashboard label is not a reliable training label unless its provenance, semantics, and lifecycle are documented.

> **Evidence before automation**
>
> The project deliberately treats undocumented semantics as a stop condition for ML training. A model can scale ambiguity just as efficiently as it can scale insight. If labels, features, or status transitions are not understood, automation converts semantic debt into model debt.

4. Delivery program - phases A through F

Phase	Primary workstream	Publicly shareable outcome
A	Privacy baseline and technical hygiene	Reduced unnecessary raw-data exposure, consolidated related request-handling paths, and retired one-shot migration code after validation.
B	Operations, deployment, and rollback	Converted tribal knowledge into a repeatable operating playbook with pre-change backups, scoped deployment units, smoke tests, and explicit rollback semantics.
C	Database inventory and recovery validation	Built a verified inventory of the persistence layer and imported a complete backup into an isolated recovery environment for read-only validation.
D	Traffic telemetry and decision pipeline	Mapped collection, classification, review, blocking, reporting, and side effects against code and confirmed data snapshots, with evidence grades attached.
E	Administrative control plane and auditability	Standardized authorization and request-forgery defenses for sensitive actions and introduced structured audit events for critical state transitions.
F	Legacy decommissioning and regression control	Combined static dependency analysis with runtime evidence and smoke tests so that only defensible removal candidates were retired.

4.1 Phase A - privacy baseline and technical hygiene

A feedback-related workflow previously retained raw network identifiers. A one-time migration replaced those values with salted, domain-specific hashes. The existing records were validated after conversion, raw values were removed from that area, and uniqueness protection remained in place to prevent duplicate counting under concurrent requests.

- Related request-header and debug behaviors were consolidated to reduce divergent identity logic.
- Temporary migration and deletion utilities were removed after successful execution rather than left as dormant production capabilities.
- Backup-related behavior was functionally checked as part of the same operational baseline.

4.2 Phase B - operations, deployment, and rollback

This phase changed the operating model more than the application logic. The objective was to turn deployment from an informal file-transfer event into a bounded transition between known states.

- Create and download a fresh backup before changes.
- Deploy only the intended files in their intended locations.
- Keep secrets, environment values, and domain-specific configuration outside accidental overwrite scope.
- Treat tightly coupled request-routing and tracking components as one release unit.
- Run database migrations as explicit, separately documented steps.
- Remove one-time jobs after successful completion.
- Use the same smoke-test suite after deployment and after rollback.

4.3 Phase C - persistence inventory and recovery validation

The database was treated as a stateful production dependency rather than a passive dump target. Schema objects, relationships, and data volumes were inventoried. A complete backup was imported into an isolated recovery database and then validated with read-only checks.

> **Recovery evidence boundary**
>
> A successful database import is strong evidence that the snapshot is structurally recoverable. It is not equivalent to a full disaster-recovery rehearsal. Before any real production restore, a separate application copy should be connected to the restored database and exercised through representative workflows.

4.4 Phase D - telemetry and decision-pipeline documentation

The traffic subsystem was documented as a decision pipeline rather than as a list of isolated rules. Collection, synchronous classification, persistence, post-request corrections, background review, administrative overrides, reporting, blocking, release behavior, and maintenance side effects were considered together.

- Rule producers and consumers were mapped against the code and confirmed data state.
- Hard and soft decision chains were separated conceptually.
- Live decisions, background review decisions, and manual interventions were treated as distinct events.
- Known ambiguities were retained as explicit evidence gaps rather than normalized away.
- No public version includes exact defensive logic, weights, thresholds, or precedence rules.

4.5 Phase E - administrative control plane and audit

Administrative workflows are the system's control plane: they can mutate classifications, content, policies, and operational state. The phase therefore focused on authorization consistency, request integrity, bounded side effects, and post-action traceability.

- Sensitive actions use centralized access gates.
- State-changing requests use request-forgery protection.
- Critical actions generate structured audit events.
- Static security checks accompany sensitive administration changes.
- Functional smoke tests verify that hardening did not break intended workflows.

4.6 Phase F - decommissioning without blind deletion

> **Core rule**
>
> No source-code reference found is not a sufficient deletion argument. Runtime access, stored-content links, redirects, scheduled execution, external clients, and historical operational paths must also be considered.

Candidate files were evaluated through static dependency analysis, a multi-week production access-log window, operator knowledge, and targeted smoke tests. Some apparently unused assets were preserved because runtime evidence showed real traffic. Only candidates with a defensible evidence chain were removed, and the application was rechecked afterward.

5. Public architecture model - data plane, decision plane, and control plane

The real production architecture is intentionally not reproduced here. The public model uses role-based components so readers can understand the engineering without receiving an application map.

Logical layer	Publicly describable responsibility
Ingress and routing layer	Normalizes request handling, applies web-server policy, and routes eligible requests into the application.
Application data plane	Serves public content and tools while producing telemetry and domain events.
Telemetry and classification plane	Derives request context, records observations, and applies synchronous policy decisions.
Review and maintenance plane	Re-evaluates stored observations, detects clusters, creates review work, and performs bounded maintenance.
Administrative control plane	Provides authenticated inspection and controlled state changes with request integrity and audit events.
Persistence layer	Stores content, application state, telemetry, review data, policy state, and audit history.
Observability and reporting layer	Turns stored state into operational views, quality checks, historical reports, and decision context.

Architectural principles

- **Defense in depth:** request routing, application checks, review workflows, administrative controls, and auditability are complementary rather than interchangeable.
- **Bounded blast radius:** changes are packaged so failures can be localized and rolled back without replacing unrelated production state.
- **Fail-visible behavior:** where possible, uncertainty and audit failures should be observable rather than silently converted into confidence.
- **Provenance over convenience:** a classification should carry enough context to explain whether it came from live logic, background review, or a human decision.
- **Separation of concerns:** telemetry capture, policy evaluation, review, and reporting may share data but should not be conflated semantically.

6. Privacy engineering and data minimization

Privacy work was treated as architecture work. The objective was not merely to rename fields or add a policy paragraph; it was to reduce unnecessary raw identifiers, narrow trust boundaries, and document why retained data exists.

Implemented principles

- Replace raw identifiers with purpose-specific derived values where raw storage is unnecessary.

- Use domain separation so a hash created for one workflow is not automatically reusable as a universal cross-system identifier.
- Keep uniqueness controls when they are required for functional integrity.
- Remove temporary migration and debug capabilities after their task is complete.
- Distinguish confirmed retention policy from automated retention enforcement.
- Avoid exposing real identifiers or telemetry samples in public documentation.

> **Data minimization is not data deletion theatre**
>
> A useful privacy control preserves the minimum state required for a legitimate function while reducing re-identification and secondary-use risk. Deleting context blindly can damage auditability; retaining everything forever creates a different class of risk. The target is governed necessity.

7. Deployment discipline, rollback semantics, and blast-radius containment

The project adopted a lightweight release-engineering model suitable for a small production application. It is not a full enterprise CI/CD platform, but it uses the same core ideas: immutable reference points, scoped change sets, preflight checks, deterministic migration order, smoke tests, and a known rollback target.

Golden-path deployment sequence

1. Capture a current application and database recovery point.
2. Define the exact release scope and explicitly list what must not change.
3. Transfer only the files that belong to the approved change set.
4. Execute one-time database transitions in their documented order.
5. Run static checks for syntax and sensitive administrative patterns.
6. Run targeted functional smoke tests across the affected user and admin paths.
7. Observe production behavior for the agreed window.
8. If a regression is confirmed, restore only the affected release unit and repeat the same validation path.

Why this matters

- A rollback plan written after failure is not a rollback plan; it is incident improvisation.
- A backup that has never been imported is an artifact, not verified recovery evidence.
- A deployment with an undefined scope has an undefined blast radius.
- A smoke test that differs between deployment and rollback makes the two states incomparable.
- A long observation window should not be retroactively described as complete when it was intentionally ended early.

8. Backup, restore validation, and disaster-recovery reality

Backup strategy was separated into three questions: Can the system create a backup? Can the operator retrieve it? Can the data be reconstructed in an isolated environment? These are related but distinct controls.

- **Backup creation:** the application can produce a current recovery artifact.
- **Off-system possession:** the operator can download and retain the artifact outside the production host.
- **Structural restore:** the database snapshot can be imported into an isolated target and validated.
- **Application restore:** a separate application copy can connect to the restored state and execute representative workflows.
- **Production cutover:** a real incident procedure can replace damaged production state with controlled downtime and rollback options.

The V1 evidence reaches the structural restore layer. It does not claim a production cutover rehearsal. That distinction is not a weakness in the documentation; it is exactly what prevents a partial test from being marketed as complete disaster-recovery proof.

9. Telemetry, review orchestration, and decision provenance

Traffic analysis is best understood as an event and decision system. One request can generate observations, a synchronous classification, stored state, later review output, administrative interpretation, and possibly a policy response. Treating only the final label as truth discards the provenance needed for debugging, governance, and future ML.

Safely publishable operating principles

- Synchronous rules and asynchronous review are separate decision layers.
- Multiple weak signals may combine into a stronger operational concern, but aggregation logic must remain auditable.
- Protected legitimate automation requires explicit safeguards so that broad heuristics do not create avoidable false positives.
- Background review can change stored classifications, but it does not retroactively change the HTTP response already delivered to a past request.
- Manual release or correction should be represented as a new decision event, not as an unexplained overwrite of history.
- The word bot is not a universally stable metric definition unless every report uses the same inclusion semantics.
- Search, tools, favorites, affiliate interactions, and primary request telemetry may be parallel measurement streams rather than one perfectly joined event graph.

Decision provenance model

Layer	Question answered	Public abstraction
Observation	What the system saw	Request context, behavior, timing, route, or aggregate pattern
Derivation	How the system interpreted it	Rule result, score contribution, anomaly indicator, or review hypothesis
Counter-signal	What argues against the interpretation	Legitimate crawler protection, plausible session behavior, trusted context
Decision	What status or action followed	Classification, review requirement, release, or bounded response
Uncertainty	What remains unknown	Missing runtime evidence, ambiguous identity, incomplete linkage
Audit event	How the state transition can be reconstructed	Actor, previous state, new state, reason, timestamp class

Why exact detection logic stays private

Publishing precise signals, thresholds, precedence, refresh behavior, block identities, or release mechanics would change the document from an engineering case study into an optimization guide for evasion. The public version therefore explains the governance model without exposing the operational policy surface.

10. Administrative control plane and auditability

The admin area is not merely another user interface. It is the privileged control plane for mutating production state. That means access checks, request integrity, validation, side-effect boundaries, and auditability must be designed as one transaction envelope.

Required controls

- Central authorization gates for privileged operations.
- Request-forgery protection for every state-changing action.
- Server-side validation independent of browser-side controls.

- Explicit operation scope so bulk actions cannot silently expand their blast radius.
- Structured audit records for high-impact changes.
- Static checks after changes to sensitive administrative code.
- Functional verification of the affected admin workflow and related user-facing behavior.

> **Audit is not a substitute for protection**
>
> A beautifully structured log entry does not make an unsafe action safe. Auditability supplements authorization, request integrity, validation, least privilege, and bounded side effects. It answers what happened after the control decision; it does not replace the control decision.

11. Legacy decommissioning with runtime evidence

Legacy cleanup was approached as controlled decommissioning rather than file deletion. The objective was not to maximize the number of removed artifacts. The objective was to reduce maintenance surface without creating silent regressions.

Conservative decommissioning pipeline

1. Build a complete candidate inventory.
2. Search direct and indirect source dependencies.
3. Inspect persistence-layer references and stored-content dependencies.
4. Review runtime access evidence over a meaningful window.
5. Consider external callers, redirects, bookmarks, scheduled execution, and operational history.
6. Classify each candidate as active, retained, uncertain, or removable.
7. Remove only the evidence-backed removable set.
8. Run regression tests and observe production behavior after removal.
9. Preserve a reference snapshot and decision record for future reconstruction.

Why static analysis alone is insufficient

- Dynamic includes and stored paths may not appear as simple literal references.
- External requests do not need an internal caller.
- Search-engine traffic and historical bookmarks can keep old routes alive.
- An old database table can still be relevant to audit, retention, or future migration even when no current writer exists.
- Conversely, the mere existence of a historical table does not prove that it should remain indefinitely.

12. AI-readiness - shadow mode, human-in-the-loop, and model governance

> **Current-state boundary**
>
> The accepted V1 baseline is a documented rule-based and review-driven system. An AI/ML decision layer is a future architecture option, not a claimed deployed production capability. Any first model should operate in shadow mode with zero autonomous production authority.

The documentation work creates the preconditions for responsible ML: stable data definitions, clearer provenance, known label quality limits, explicit override semantics, and an audit trail. Without those foundations, a model would learn from a mixture of observations, heuristics, historical corrections, and ambiguous statuses without knowing which signals deserve trust.

Proposed layered intelligence model

Layer	Role	Governance constraint
Existing rule engine	Deterministic policy baseline	Produces interpretable operational decisions from known conditions.
Tabular ML model	Probabilistic pattern assessment	Estimates likelihood or risk from documented features; initially shadow-only.
Anomaly detector	Novelty and distribution shift	Highlights unusual clusters without equating unusual with malicious.
Independent control layer	Policy, veto, and guardrails	Checks confidence, counter-signals, protected classes, and allowed action boundaries.
Human reviewer	Final accountability	Approves changes, resolves ambiguity, and owns production policy.
Audit and versioning	Reproducibility	Records data version, feature set, model version, rule state, and decision outcome.

Shadow-mode operating model

- The model receives the same documented feature snapshot that can later be reproduced.
- Its output is stored as a prediction, not applied as a production classification or block.
- Disagreement between rules and model becomes review material rather than an automatic escalation.
- Calibration, false-positive rate, segment behavior, and confidence drift are measured over time.
- Protected legitimate automation and known test traffic remain explicit counter-signals.
- No model retraining uses an automatically generated label merely because the same model or rule produced it.

Why two AI components can be rational rather than theatrical

The phrase two AIs sounds more autonomous than the proposed reality. A safer interpretation is separation of duties: one analytical component proposes a probabilistic interpretation, while another constrained reasoning or policy component searches for counter-evidence, policy conflicts, and false-positive risk. Neither component should independently own production action.

- **Predictor:** What pattern does the data resemble, and how confident is the model?
- **Critic or controller:** What evidence contradicts that prediction, what guardrail applies, and is the proposed action even permitted?
- **Human owner:** Should the system change policy, release a case, or keep observing?

In AI-governance vocabulary, this is a human-in-the-loop, policy-constrained ensemble with an explicit veto path. In normal vocabulary, it is a four-eyes principle for statistics.

Model-governance guardrails

- No autonomous production changes during the initial maturity stage.
- No training on fields with undocumented provenance or unresolved leakage risk.
- Version every feature definition, label policy, model artifact, threshold proposal, and evaluation set.
- Maintain a small, high-confidence gold dataset separate from weak operational labels.
- Track drift in input distributions, label mix, calibration, and disagreement with the rule baseline.
- Require rollback and kill-switch semantics before any bounded automated action is considered.
- Keep the independent control layer simpler, more deterministic, and harder to self-modify than the predictor it supervises.

13. Known unknowns, technical debt, and evidence gaps

A mature technical document does not end by declaring victory over uncertainty. It identifies where the evidence boundary still sits and what proof would move that boundary.

Open evidence domains

- Independent confirmation of the currently installed scheduler and its runtime environment.

- Current database modes, grants, and behavior across uncommon failure paths.
- Complete technical enforcement of the documented retention policy.
- External infrastructure behavior not derivable from the reviewed application artifacts.
- Provenance of selected historical event and policy fields.
- Full runtime reachability of every legacy or error-handling path.
- Long-term policy design for identity, block refresh, release semantics, and cluster treatment.

Known technical risk categories

The internal documentation retains precise code-level findings. This public edition intentionally limits them to categories: defensive conditions that may be ineffective, error paths with incomplete guarantees, status values with inconsistent semantics, concurrency assumptions requiring runtime confirmation, and data fields whose writers or provenance are not fully established.

> **Why the public version stops at categories**
>
> Publishing the exact file, variable, branch, input condition, or state transition would add exploitability without adding proportionate educational value. The useful public lesson is that the issues were recorded, separated from assumptions, and not silently erased from the project narrative.

14. Transferable engineering principles

14.1 Documentation and evidence

- Document behavior, provenance, side effects, and uncertainty - not just component names.
- Separate code-confirmed facts from operator statements and unresolved runtime questions.
- Use a fixed cut-off and identify the exact source snapshot behind the documentation.
- Keep internal operational documentation and public case-study documentation as separate artifacts.
- Treat inconsistencies as first-class findings rather than editing them out for narrative cleanliness.

14.2 Change and release engineering

- Package changes into bounded phases with explicit acceptance criteria.
- Define backup, scope, forbidden changes, deployment order, and rollback before implementation.
- Treat schema changes as state transitions with their own evidence and failure plan.
- Combine static checks, functional tests, and a declared observation window.
- Never upgrade the language of a test result beyond what was actually executed.

14.3 Data, telemetry, and AI

- Do not convert frequent but unverified behavior into training truth.
- Do not reuse operational labels across reports or models without a stable semantic contract.
- Store live decisions, review decisions, manual corrections, and uncertainty separately.
- Do not train on data whose source and timing semantics are unknown.
- Deploy the first model in shadow mode and measure calibration before discussing automation.
- Make counter-signals and false-positive cost visible in evaluation, not merely overall accuracy.

14.4 Legacy, retention, and recovery

- Never delete a component solely because static search found no caller.
- Use runtime evidence, persistence references, external access, and operator context together.
- Separate legal or business retention policy from actual technical enforcement.
- Test that backups can be reconstructed; do not infer recoverability from file existence.
- Preserve a reference snapshot and a decision log for retired components.

15. Sanitization model for this public edition

This edition was authored as a new document. It was not produced by placing visible redaction boxes over the internal full documentation. Therefore, there is no hidden underlying operational text that remains selectable or extractable beneath visual black bars.

Information class	Treatment	Rationale
Brand, logo, and project identity	Included	Preserves attribution and intentional brand visibility.
Project goals, phases, and engineering principles	Included	Keeps the case study useful without disclosing the production map.
Aggregated outcomes and high-level test statements	Selectively included	Useful for credibility when they do not reveal defensive thresholds or topology.
File names, directory layout, endpoints, and line references	Removed	Prevents reconstruction of the application and administrative surface.
Database prefixes, table names, columns, and internal status labels	Abstracted	Replaced with role-based terms such as telemetry store, review queue, or audit log.
Server paths, commands, accounts, scheduler definitions, and provider details	Removed	Avoids infrastructure reconnaissance value.
Exact rules, weights, scores, thresholds, and precedence chains	Removed	Avoids creating an evasion playbook.
Trap locations, block identities, release mechanics, and bypass behavior	Removed	Avoids operationalizing attack or avoidance paths.
Real identifiers, timestamps, hashes, addresses, sessions, agents, and log lines	Removed	Protects users, internal patterns, and investigative context.
Known code defects	Category only	The public text describes risk classes, not exploitable locations or trigger conditions.

Additional publication controls

- Minimize document metadata and remove personal authoring information.
- Exclude comments, tracked changes, hidden content, embedded attachments, and internal links.
- Generate the PDF from the sanitized source file.
- Search the final text for internal prefixes, path fragments, rule names, commands, and infrastructure markers.
- Inspect every rendered page for accidental screenshots, paths, identifiers, or layout defects.
- Use neutral filenames that do not reveal internal project structure.

16. Publication release checklist

Status	Control	Acceptance criterion
PASS	Source content	No internal file, database, rule, status, or class names remain.
PASS	Infrastructure	No server paths, accounts, host-specific commands, scheduler definitions, or topology details remain.
PASS	Detection logic	No exact scores, thresholds, traps, block conditions, precedence chains, or release mechanics remain.
PASS	Examples	No real IDs, hashes, addresses, sessions, user agents, referrers, search terms, or log excerpts remain.
PASS	Metadata	Document properties are minimized and personal authoring metadata is removed.
PASS	Export path	The PDF is regenerated from the sanitized source rather than visually redacted over hidden text.
PASS	Visual QA	Every rendered page is checked for clipping, accidental paths, broken tables, and hidden operational clues.
PASS	Publication boundary	The document remains a systems-engineering case study, not a deployment guide or security runbook.

Closing perspective

The main engineering result was not one clever detection rule or one successful database import. It was the conversion of an organically grown application into a more observable and governable system: privacy decisions were tied to data purpose, deployments were tied to rollback, backups were tied to restore evidence, telemetry was tied to provenance, administrative actions were tied to audit events, and cleanup was tied to runtime evidence.

The AI angle comes later. The work described here is the less glamorous prerequisite: making sure a future model does not ingest undocumented semantics, amplify historical mistakes, or become an unaccountable production actor. The proposed intelligence layer therefore starts as a shadow analyst, is challenged by an independent control layer, and remains subordinate to human approval and deterministic safety boundaries.

> **One-line architecture summary**
>
> The system may eventually get a statistical brain and an AI critic, but the production control plane, veto path, rollback switch, and final accountability remain human-owned.

AirfryerGuide.de | V1 Public Edition | English Extended Release