AI ENGINEERING • AUTOMATION • CYBERSECURITY

CASEY KNOTT

AI Engineering, Automation & Cybersecurity Project Portfolio

Applied AI • Agentic Workflows • Secure Systems • Automation • Software Validation

Human-directed, AI-assisted engineering - explicit architecture, security controls, testing, validation, release gates, and human approval on every project inside.

SEVEN PROJECTS - KNOWLEDGE SYSTEMS · DOMAIN AI · AUTOMATION · AGENTIC PUBLISHING · SECURITY TRIAGE · PRODUCT ENGINEERING · SECURE PLATFORMS

How This Portfolio Works

HOW TO READ THIS DOCUMENT

This portfolio documents seven projects spanning enterprise knowledge tooling, domain-specific AI assistants, scheduled automation, agentic publishing, home-lab security triage, and two production-track software platforms. It is not a resume written for one role - it is a working record of how I actually build things.

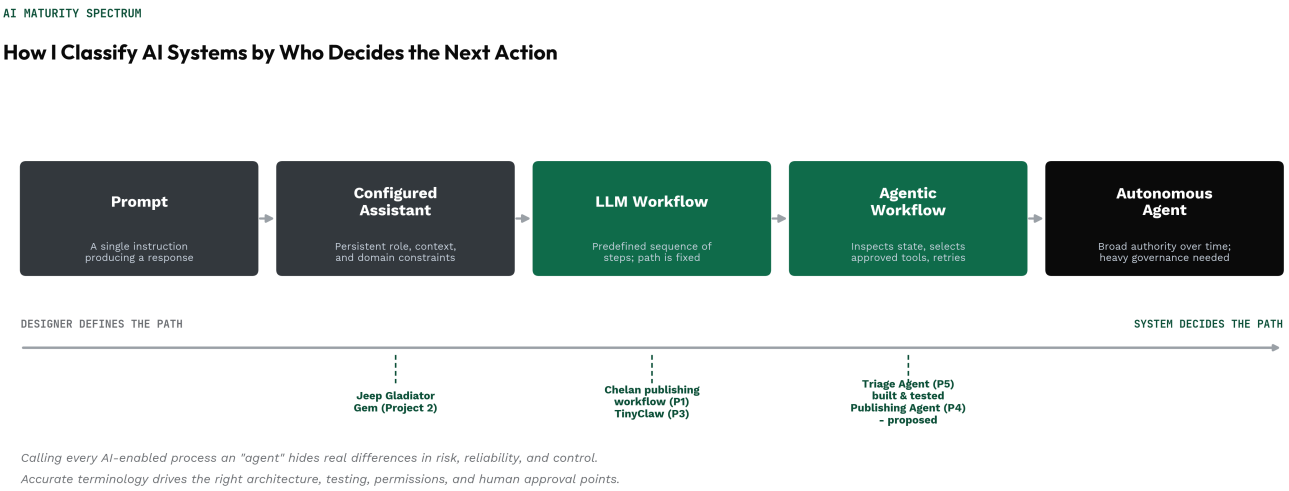
The short version: I do not simply use AI tools. I design the systems, define the architecture and the security controls, direct AI coding and automation agents against explicit requirements, validate what they produce, troubleshoot the failures, and decide what is allowed to ship. Where code, workbooks, or documentation were AI-drafted, they were drafted against requirements and constraints I wrote, and they did not reach a user without a validation step I designed and a decision I made.

Every project below follows the same structure - problem, architecture, my role, engineering decisions, security and reliability controls, validation, skills demonstrated, and what I learned - so the pattern of judgment is visible across very different domains, not just the output.

CONTENTS

How I Classify AI Systems

I do not use “AI agent” as a catch-all term. I distinguish among configured assistants, deterministic LLM workflows, agentic workflows, and autonomous agents based on how much decision-making, tool selection, state management, and recovery behavior the system actually performs. The critical question is who determines the next action.



Calling every AI-enabled process an "agent" hides real differences in risk, reliability, and control.
Accurate terminology drives the right architecture, testing, permissions, and human approval points.

Prompt	A single instruction that produces a response.
Configured assistant	Persistent role, context, rules, and domain constraints; primarily conversational.
LLM workflow	A predefined sequence of LLM and non-LLM steps. Reliable, but the path is largely fixed.
Agentic workflow	The system inspects state, selects approved actions, manages retries, and adapts its path to complete a goal.
Autonomous agent	Operates with broader authority over time. Requires significantly stronger governance, monitoring, and safety controls.

WHY THIS MATTERS

Calling every AI-enabled process an “agent” hides important differences in risk, reliability, and control. Accurate terminology determines the appropriate architecture, testing, permissions, and human approval points. Each project in this portfolio is labeled with where it actually sits on this spectrum, including which are built and which are designs I am working toward.

Current Capability Snapshot

An honest assessment of where each capability actually stands, with the evidence behind it.

Capability	Evidence	Maturity
Prompt & role configuration	Built specialized Gemini Gem behavior for a specific vehicle-maintenance domain	Applied
LLM workflow design	Multi-stage GPT processing for documentation standards, metadata, and HTML generation	Applied
Scheduled AI reporting	Locally hosted market-report workflow: data gathering, Gemini analysis, scheduled delivery	Applied
Agent architecture	Built and ran a state-aware endpoint-triage agent with tools, validation, retries, and approvals	Applied
Secure AI operations	Least privilege, tool allowlists, audit logging, and safe stopping - verified by red-team testing	Applied
Adversarial testing	Isolated a lab VM and ran red-team tactics against my own agent, documenting findings and fixes	Applied
Product & release engineering	Clearward digital products: structured data design, automated QA, packaging, and release gates	Applied
Secure platform engineering	Basaltborne: RLS, migration discipline, API boundaries, and a multi-layer CI validation suite	Developing

PROJECT 01

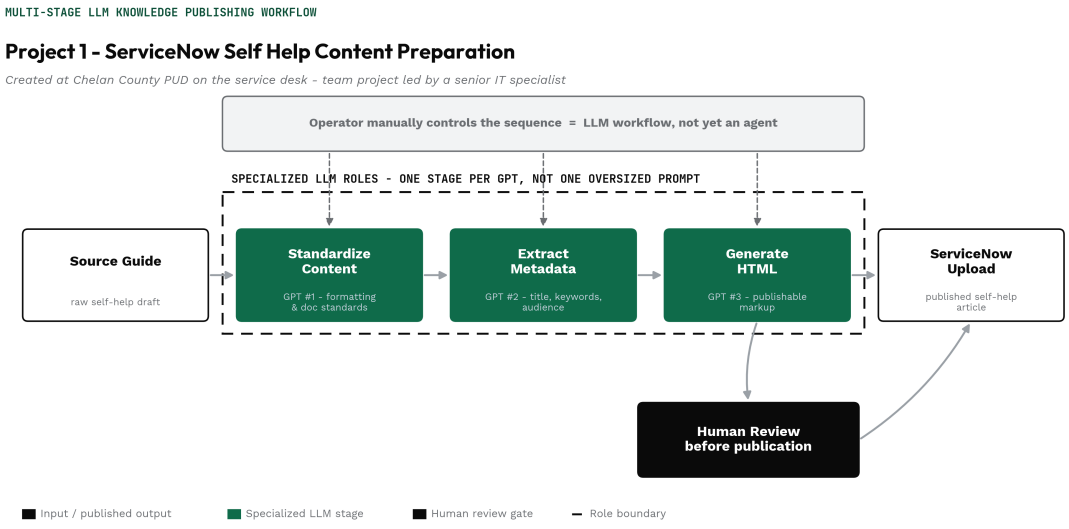
Multi-Stage Knowledge Publishing Workflow

Chelan County PUD - ServiceNow Self Help Content Preparation

LLM WORKFLOW • APPLIED

PROBLEM

This project was created at Chelan County PUD on the service desk, the team I was part of. Preparing ServiceNow Self Help content was slow and inconsistent: formatting varied between authors, metadata was thin enough to hurt search and discoverability, and converting finished guides into publishable HTML was manual work.



Each custom GPT owned one narrow stage of the publishing process. Because an operator drove the sequence by hand, this is a fixed LLM workflow rather than an agent - the distinction that led directly to the agent designs later in this portfolio.

MY ROLE

A senior IT specialist led this work and I joined as a team member. He designed the multi-GPT approach and owned the stage definitions; I worked alongside him running content through the pipeline, testing output against real self-help guides, and feeding back where formatting, metadata, or HTML generation broke down. I did not lead this project. It was my first hands-on exposure to AI as something you architect in stages rather than prompt in one shot, and most of what I know about LLM workflow design started here.

ENGINEERING DECISIONS

- Specialization of LLM roles: each GPT handled one stage (standardize, extract metadata, generate HTML) instead of one oversized prompt trying to do everything.
- Consistent formatting and documentation standards enforced at the standardization stage, so downstream stages received predictable input.
- Metadata extraction treated as its own stage because search and discoverability in ServiceNow depended on it.

SECURITY & RELIABILITY CONTROLS

- Human review before final publication - no content reached ServiceNow unreviewed.
- Each stage produced inspectable intermediate output, so a bad result could be caught at the stage that caused it rather than at the end.
- The operator retained control of the sequence, which limited blast radius but also capped how much the workflow could adapt on its own.

VALIDATION

Output was reviewed against real self-help guides at each stage. I helped test and reported failure modes back to the specialist leading the work, who refined the prompts and stage boundaries from there.

SKILLS DEMONSTRATED

`Multi-Stage LLM Workflow Design` `Role Specialization` `Prompt Design`

`Metadata Extraction` `Documentation Standards` `HTML Generation`

`ServiceNow Knowledge Publishing` `Human Review Gates`

WHAT I LEARNED

This was a strong LLM workflow but not a full agent, because a person decided each next step. Recognizing exactly where that line falls - who determines the next action - is the single most useful concept I took from this project, and it is what the Knowledge Publishing Agent design later in this portfolio was built to address.

PROJECT 02

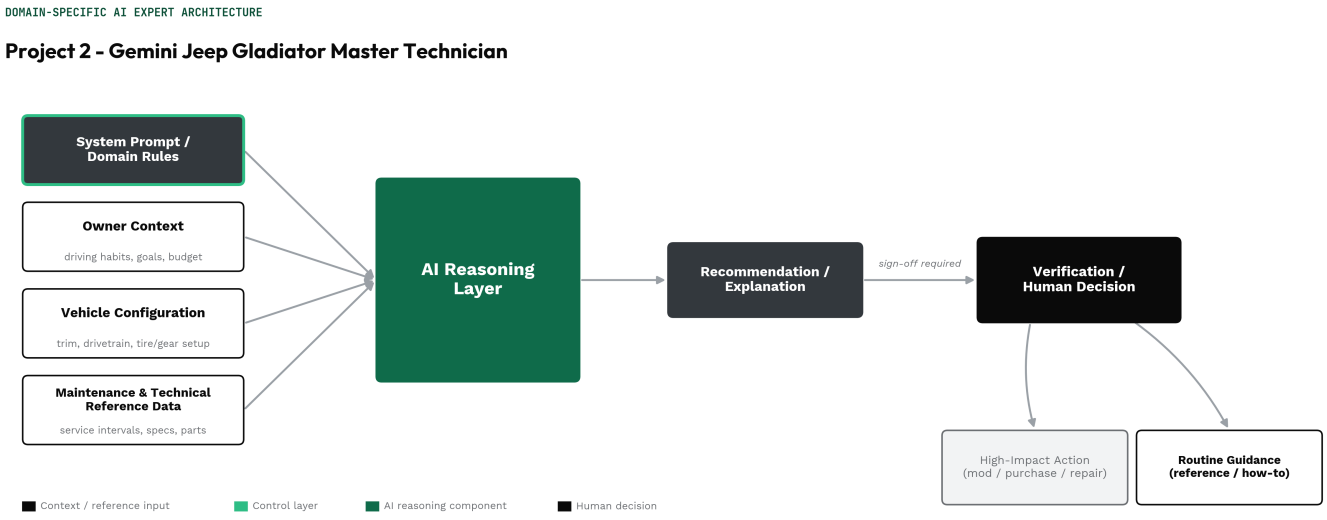
Domain-Specific AI Expert

Gemini Gem - 2025 Jeep Gladiator Sport S with Max Tow

CONFIGURED ASSISTANT • APPLIED

PROBLEM

Generic AI answers about a vehicle ignore the specifics that actually matter - trim, drivetrain, tow package, tire and gear setup, prior modifications, service history. I wanted an assistant that reasoned about my specific 2025 Jeep Gladiator Sport S with Max Tow, not the average Gladiator.



Owner context, vehicle configuration, and reference data feed a reasoning layer constrained by explicit domain rules; high-impact actions are routed to the owner, not executed or assumed.

MY ROLE

Designed the system prompt and domain rules, curated the owner-context and vehicle-configuration inputs, and defined which categories of output required my own sign-off - purchases, modifications, anything safety-relevant - versus what could be treated as plain reference information.

ENGINEERING DECISIONS

- Separated owner context, vehicle configuration, and maintenance/technical reference into distinct inputs so the model reasons with full context rather than one blended prompt.
- Built a persistence pattern so context didn't need to be re-explained every session.
- Instructed the assistant to flag judgment calls ("is this mod worth it") differently from settled facts (torque specs, service intervals).

SECURITY & RELIABILITY CONTROLS

- No direct write access - the assistant recommends, it never executes.

- High-impact recommendations (parts purchases, drivetrain modifications, anything safety-relevant) explicitly routed to a human decision.
- Source-verification instruction to prevent generic forum consensus from being presented as verified specification.

VALIDATION

Cross-checked recommendations against manufacturer service documentation and parts compatibility references before acting on any of them.

SKILLS DEMONSTRATED

`Domain-Specific AI Design` `Context Engineering` `System Prompting`

`Constraint Definition` `Source Verification` `Specialized Assistant Design`

WHAT I LEARNED

The gap between a useful domain assistant and a generic chatbot is almost entirely context engineering and constraint definition, not model choice. Just as important: this is honestly a configured assistant, not an agent. It does not select tools, hold investigation state, or take actions. Being precise about that distinction is what made the agent designs later in this portfolio possible.

PROJECT 03

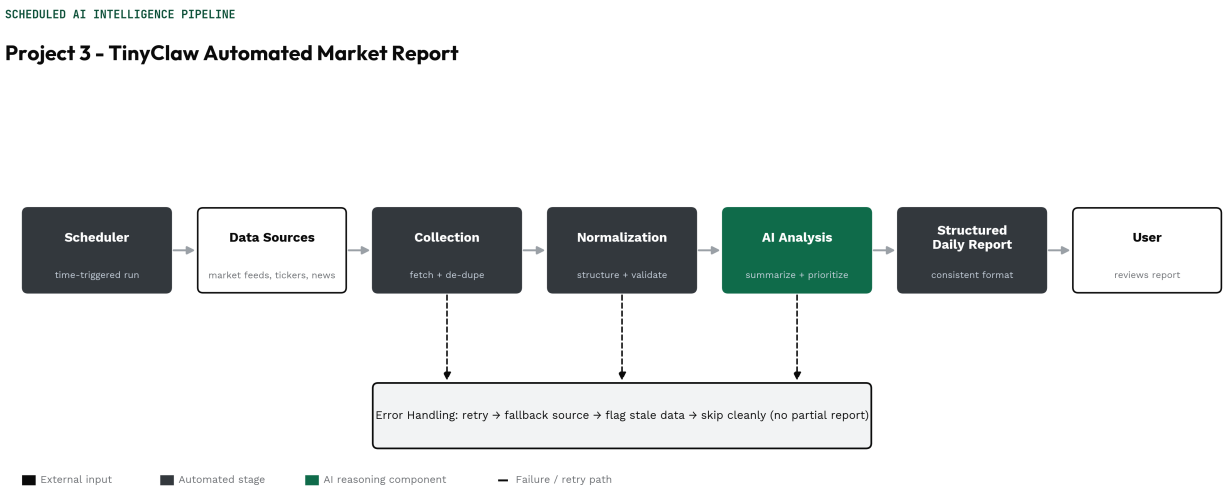
Scheduled AI Intelligence Pipeline

TinyClaw - Locally Hosted Scheduled Market Reporting

LLM WORKFLOW • APPLIED

PROBLEM

Producing a morning market report meant gathering prior-day, overnight, and pre-market information by hand every day. A naive “cron job that calls an LLM” breaks in ordinary ways: duplicate reports, partial output on a failed fetch, and stale data presented as fresh and authoritative.



A time-triggered pipeline collects, normalizes, and summarizes market data into a consistently formatted report, with an explicit retry-and-skip path instead of ever publishing a partial or stale result.

MY ROLE

Built and run TinyClaw as a locally hosted, scheduled workflow: it gathers prior-day, overnight, and pre-market data, passes it to Gemini for analysis, generates a structured report, and delivers it each morning. I designed the pipeline shape and the failure-handling behavior. The model is scoped narrowly to analysis and summarization inside a pipeline I control, not an open-ended “go find today’s news” agent.

ENGINEERING DECISIONS

- Idempotent runs: re-running for the same period never duplicates or corrupts prior output.
- An explicit normalization step precedes the model call, so it summarizes structured, validated input rather than raw scraped text.
- Chose to skip a run cleanly rather than publish a partial or misleadingly “fresh” report on a source failure.

SECURITY & RELIABILITY CONTROLS

- Retry → fallback source → flag stale data → clean skip, in that order, on any collection failure.
- Consistent structured output format so downstream consumption doesn't silently break on formatting drift.

VALIDATION

Manually reviewed report output against source data across multiple runs, checking specifically for hallucinated figures or misattributed numbers before trusting the automation.

SKILLS DEMONSTRATED

`Local Hosting` `Scheduled Execution` `External Data Collection` `Gemini Analysis`
`Prompt Design for Consistent Structure` `Idempotence` `Error Handling`
`Data Freshness Awareness` `Output Reliability`

WHAT I LEARNED

AI output quality is bounded by upstream data quality. A reliable system has to validate its inputs, record source timing, expose uncertainty, and refuse to present incomplete data as authoritative. A more agentic version would judge whether sufficient data exists, call alternate approved sources when something is missing, detect stale or conflicting inputs, retry failed collection, and stop safely when confidence requirements are not met - that gap is exactly what separates this workflow from an agent.

PROJECT 04

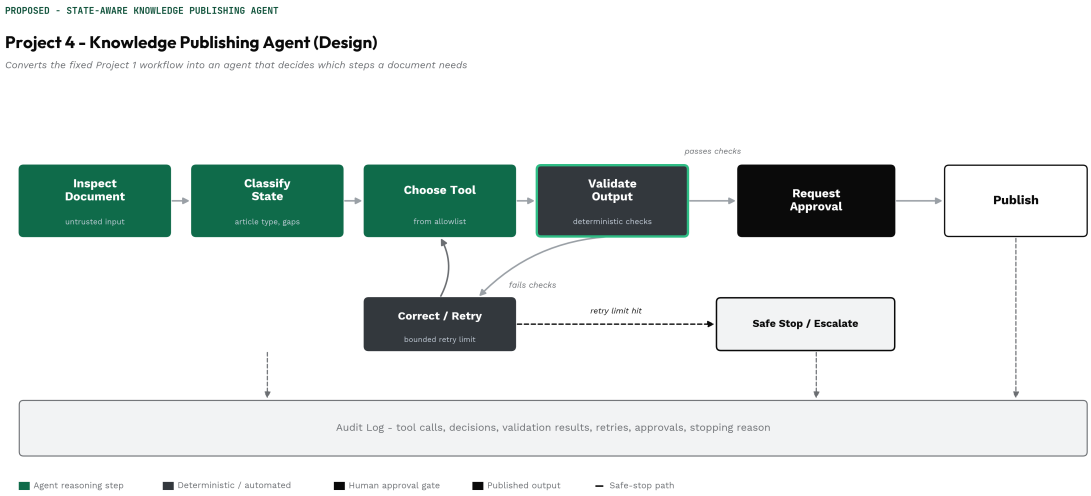
Knowledge Publishing Agent

Proposed Design - State-Aware Publishing Agent

AGENTIC WORKFLOW • PROPOSED DESIGN

PROBLEM

The Project 1 workflow worked, but a person had to decide every next step. Many documents do not need every stage: some already have clean formatting, some are missing metadata, some have broken links. A fixed sequence cannot tell the difference.



The agent inspects a document, classifies its state, selects approved tools, and validates its own output against deterministic checks - retrying within limits, stopping safely when it cannot, and requiring human approval before anything publishes.

MY ROLE

This is a design I have specified but not yet built. I have defined the agent's responsibilities, its state model, the tool allowlist boundary, the validation and retry behavior, and the approval and audit requirements. Building it is Phase 2 of my development roadmap.

ENGINEERING DECISIONS

- The agent inspects source content and classifies article type, rather than assuming a uniform document shape.
- It detects missing sections, unclear instructions, broken links, and formatting issues, then selects only the approved templates and transformation tools that condition actually requires.
- Deterministic code - not model judgment - runs the checks for required fields, structure, links, and style compliance.

- Content generation is kept structurally separate from publishing authority.

SECURITY & RELIABILITY CONTROLS

- Document content is treated as untrusted input, with explicit resistance to embedded prompt-injection instructions; tool policy overrides anything asserted in the content.
- Tool allowlist and least-privilege service identity constrain what the agent can reach.
- Enforced retry limits and safe stopping conditions rather than unbounded self-correction.
- Human approval required before publication, with an audit trail of decisions, validation results, retries, and final disposition, plus versioned output artifacts.

VALIDATION

Planned validation includes happy-path runs with known-good guides, malformed and incomplete documents, prompt-injection attempts embedded in source content, tool-failure and retry-limit cases, and verification that the agent stops safely instead of guessing.

SKILLS DEMONSTRATED

`Agent Architecture` `State Classification` `Tool Selection & Allowlists`

`Deterministic Validation` `Prompt-Injection Resistance` `Least Privilege`

`Retry Limits & Safe Stopping` `Human Approval Gates` `Audit Logging`

WHAT I LEARNED

Designing this clarified that “human in the loop” only means something when a state machine enforces it. It also forced the separation I now apply everywhere: probabilistic model judgment for interpretation, deterministic code for anything that must be correct.

PROJECT 05

Home-Lab Endpoint Triage Agent

Flagship Build - AI-Assisted Security Investigation, Adversarially Tested

AGENTIC WORKFLOW • BUILT & ADVERSARIALLY TESTED

PROBLEM

Security triage burns analyst time on evidence gathering and formatting before real judgment can begin. The opportunity is to accelerate collection and correlation without letting a model substitute for evidence or analyst judgment - which is exactly where naive AI automation becomes dangerous in a security context.

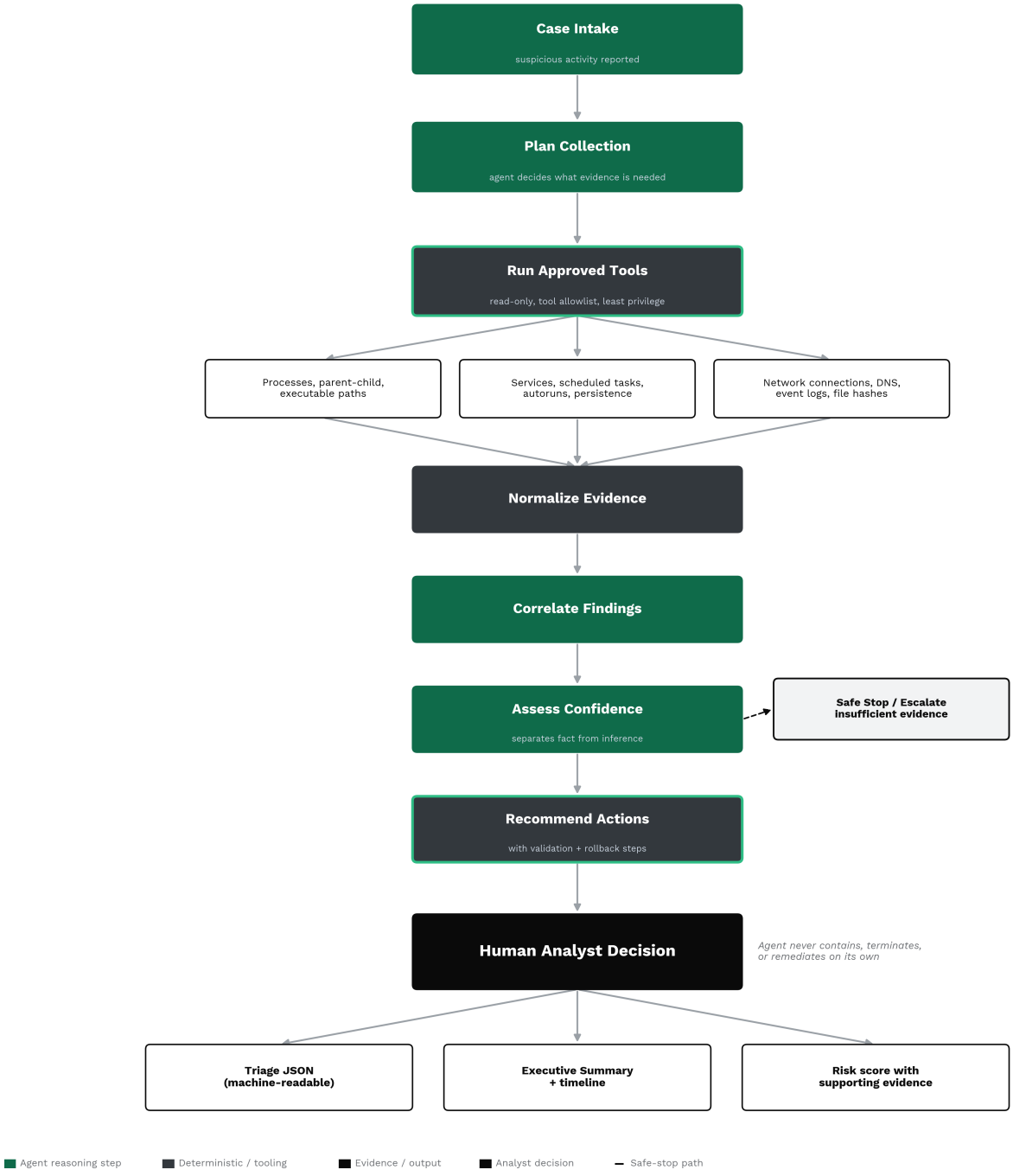
ENVIRONMENT

Built and run on my mobile workstation: a Windows 11 Pro host with Hyper-V, hosting TRIAGE-ENDPOINT-01 - a Generation 2 VM with 4 vCPU, 6 GB dynamic memory (4-8 GB range), a 100 GB VHDX, Secure Boot, and vTPM. Before any adversarial testing began, the VM was placed in an isolated network environment: a private virtual switch with no internet or LAN path, no shared folders or clipboard passthrough, and checkpoints taken so every test cycle could be rolled back to a known-good state.

AI-ASSISTED ENDPOINT TRIAGE AGENT - BUILT & ADVERSARIALLY TESTED

Project 5 - Home-Lab Endpoint Triage Agent

Built on an isolated Hyper-V VM (TRIAGE-ENDPOINT-01) and tested against red-team tactics



The agent plans and collects read-only evidence, correlates findings, and states its confidence. Every consequential action stays with the analyst, and outputs separate confirmed evidence from model inference.

MY ROLE

I designed, built, isolated, and tested this system end to end. That covered the investigation flow, the read-only evidence scope, the confidence model, the output contract, and the security controls - then

containing the environment properly and running red-team tactics against my own agent to find out where it actually broke. The adversarial study and its documented findings are covered in the section that follows.

ENGINEERING DECISIONS

- The agent plans its own collection based on the case rather than running a fixed script, which is what makes it agentic rather than a batch job.
- Evidence scope is deliberately read-only: processes and parent-child relationships, services and scheduled tasks, autoruns and persistence indicators, network connections and DNS context, Windows event and authentication logs, file hashes, signatures, and timestamps.
- Outputs are designed as a contract: machine-readable triage JSON, an analyst-readable executive summary, a timeline, and a risk score backed by cited evidence rather than an unexplained number.
- Confirmed evidence, likely interpretations, and unresolved hypotheses are kept visibly separate in every report.

SECURITY & RELIABILITY CONTROLS

- Collection, analysis, and any remediation authority are separated; the agent operates with least privilege and a tool allowlist.
- Consequential actions - isolation, process termination, account disablement - always require explicit analyst approval. The agent recommends; it does not act.
- Command output and log content are treated as untrusted input that may contain instructions intended to manipulate the model.
- Safe failure: the agent stops and escalates when evidence is insufficient, tools fail repeatedly, or permissions are exceeded, rather than guessing.
- Every recommended change ships with expected results, validation criteria, operational risk, and a rollback procedure.

VALIDATION

Validation ran in two layers. Functional testing covered happy-path scenarios with known-good inputs, then missing, malformed, stale, and contradictory evidence, tool timeouts and permission failures, and retry-limit and safe-stop verification. Adversarial testing then attacked the agent directly with red-team tactics inside the isolated environment. Every cycle ended with human review of whether the agent's reasoning was actually supported by the evidence it had collected.

SKILLS DEMONSTRATED

`Agent Architecture` `Evidence-Driven Triage` `Windows Endpoint Forensics`
`Event Correlation` `Persistence & Autoruns Analysis` `Confidence Modeling`
`Network Isolation & Containment` `Red-Team Testing` `Prompt-Injection Resistance`
`Least Privilege` `Read-Only Collection` `Rollback & Checkpoint Discipline`
`Hyper-V Lab Administration` `Findings Documentation`

WHAT I LEARNED

The hardest part of this build was not the reasoning - it was deciding what the agent is not allowed to do, then proving those limits held under attack rather than assuming they would. Attacking my own system taught me more than building it did: controls that looked sound in design are only real once something has tried to get around them.

PROJECT 05 - CASE STUDY

Adversarial Testing of My Own Agent

Isolated Lab Study - Red-Team Tactics Against the Triage Agent

ISOLATED LAB STUDY • COMPLETED

Building the agent was only half the project. An agent that reads attacker-controlled data and holds tool access is itself an attack surface, so I contained the environment and then deliberately attacked my own system to find out where the controls actually held and where they did not.

ISOLATION FIRST

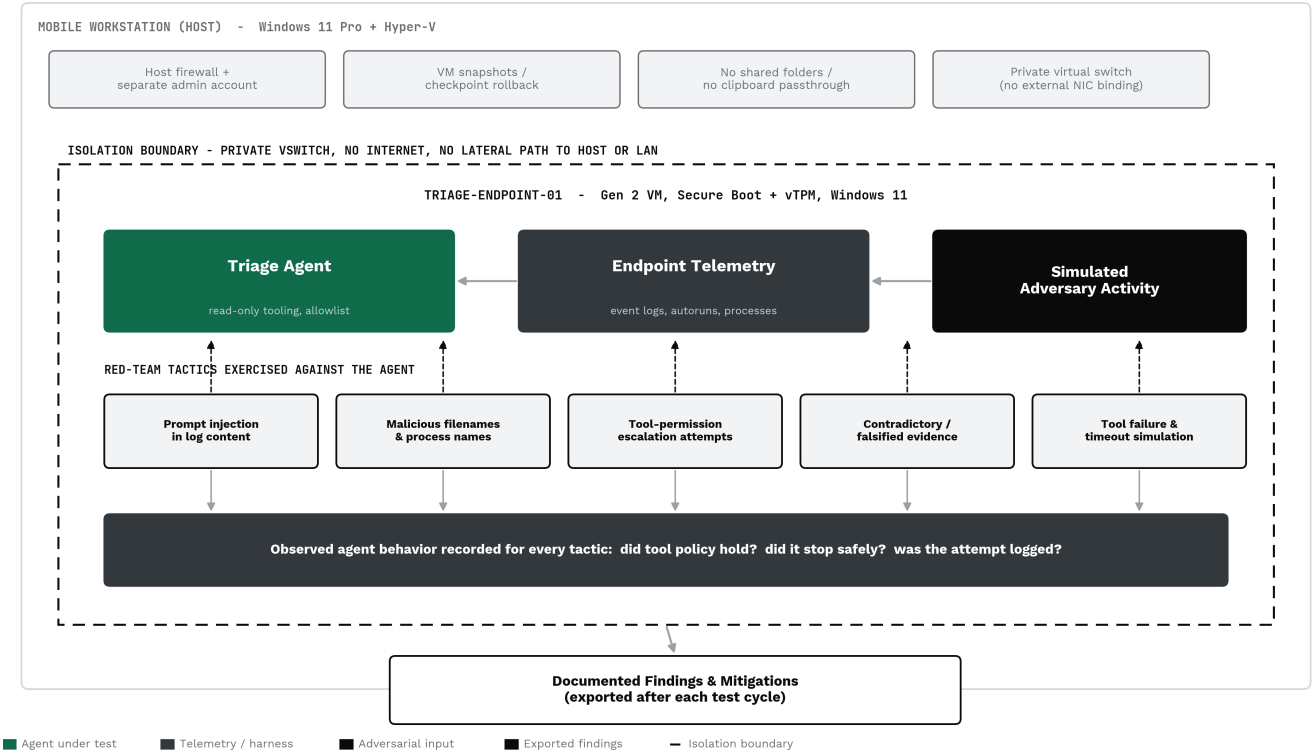
No adversarial testing started until the environment was contained. The principle was straightforward: assume the test could succeed, and make sure that success stays inside the box.

- Private Hyper-V virtual switch with no binding to a physical NIC, so the VM had no internet and no path to the host LAN.
- No shared folders, no clipboard passthrough, and no drive redirection between guest and host.
- Secure Boot and vTPM left enabled so the guest reflected a realistic endpoint rather than a stripped-down test box.
- Checkpoints taken before every test cycle, so any tactic that changed system state could be rolled back to a known-good baseline.
- The agent ran under a least-privilege account with read-only collection tooling and an explicit tool allowlist, so a successful manipulation still had a small blast radius.
- Host-side separation: a distinct admin account and host firewall rules, treating the VM as hostile rather than trusted.

ISOLATED ADVERSARIAL TEST ENVIRONMENT

Project 5 - Lab Isolation & Red-Team Test Harness

Containment boundaries applied before running adversarial tactics against the agent



The environment was contained before any tactic was run: no internet or LAN path off the VM, no host passthrough, and checkpoints so each cycle could be reverted.

Tactics Exercised

Prompt injection in log content	Planted instruction-like text inside filenames, event log messages, and command output to see whether content could override tool policy or redirect the investigation.
Malicious process and file naming	Named benign test artifacts to imitate known-bad tooling, and named genuinely suspicious artifacts to look routine, testing whether the agent reasoned from evidence or from labels.
Tool-permission escalation attempts	Prompted conditions where completing the task would be easier with privileges or tools outside the allowlist, checking whether the agent stayed inside its granted authority.
Contradictory and falsified evidence	Fed conflicting timestamps and mismatched artifacts to see whether the agent surfaced the conflict or silently picked a narrative.
Tool failure and timeout simulation	Broke collection tooling mid-investigation to verify the agent degraded safely instead of reporting confident conclusions on partial evidence.

WHAT I RECORDED FOR EACH TEST

- Did tool policy hold, or did content in the evidence change the agent's behavior?
- Did the agent stay inside its allowlist and privilege boundary?
- Did it stop and escalate, or did it guess when evidence ran out?
- Was the attempt visible in the audit log afterward?
- Did the final report separate confirmed evidence from inference, or blur them?

Documented Findings

Each cycle produced a written record: the tactic used, the agent's observed behavior, whether the relevant control held, and what changed as a result. Where a tactic succeeded, the fix went into the design rather than into the prompt where possible - tightening the allowlist, moving a check from model judgment into deterministic code, or adding an explicit stop condition. The most useful pattern that emerged: controls enforced in code held up, and controls that depended on the model choosing to respect an instruction were the ones worth hardening.

Treating my own agent as untrusted is the part of this project I would most want a security team to look at. It is the difference between believing a control works and having evidence that it does.

PROJECT 06

Clearward

Move Toward a Calmer Life - Structured Organization Systems as a Product

Clearward is a real digital-product business built around structured organization systems for everyday life and small trades businesses, delivered as Excel and Google Sheets products under the brand direction “Clearward - move toward a calmer life.”

Product families span Home, Auto, Business, Pet Care, Moving, Estate and Executor Planning, Parent-Care Coordination, Farm Operations, RV Ownership, small-business/trades operating systems (HVAC, plumbing, electrical, roofing, mobile detailing, pressure washing, lawn and landscape, handyman, appliance repair), and collectibles systems.

The interesting engineering isn’t the spreadsheet format - it’s the discipline underneath it: structured tables, stable identifiers, validated inputs, formula-derived state, and a documented QA process that treats a workbook release with the same rigor as any other shipped software.

System Design

The spreadsheets evolved from templates into structured, software-like products. Concepts carried through every product line: structured tables with stable record identifiers, controlled input cells separated from formula-driven calculated state, data validation and drop-down controls, conditional formatting, configuration tables, dashboard views, status logic, customer-editable thresholds, and bounded formulas - all designed for cross-platform Excel/Google Sheets compatibility and documented as a deliverable in its own right, not an afterthought.

Pet Care System - A Concrete Example

The Pet Care System is a concrete example of the pattern: 13 expected worksheets built on 8 structured data tables - pet profiles, veterinary visits, vaccinations, medications, weight and wellness, recurring care, expenses, and contacts - plus settings, a dashboard, a pet-sitter handoff view, and user guidance.

- Stable IDs and one-record-per-row structure across every table
- Unique table names; no ambiguous or reused ranges
- Formula-derived state kept separate from source input
- Configurable thresholds instead of hard-coded values
- Deliberately no macros, no Power Query, no external data connections
- Compatibility-conscious formula choices across Excel and Google Sheets

QA & Release Engineering

AI-generated products were not treated as automatically trustworthy. Every release runs through a documented QA process before it reaches a customer. For the Clearward Pet Care System v1.0, that process covered:

49	48	1	0	1,358
automated checks run	checks passed	advisory (non-blocking)	failures	formula cells scanned

- Formula error scanning across every populated cell
- Open XML package validation
- Expected-sheet and table validation
- External-reference and macro checks
- Formula reconciliation against known sample input
- Data-validation checks
- Visual and PDF rendering review
- Cross-platform compatibility checks
- Customer-package validation

Files were opened, recalculated, and saved through a second office engine specifically to surface compatibility issues, and customer-facing PDFs were rendered and visually inspected for clipping, overlap, missing sections, and rendering artifacts before a release was approved. This is test thinking and evidence-based acceptance criteria applied to a product format - spreadsheets - that rarely gets treated with this level of release discipline.

Agentic Automation

AI coding agents (Codex, ChatGPT) were directed by persistent, structured agent instructions (AGENTS.md) rather than ad hoc prompting. Those instructions defined permitted directories, product requirements, release procedures, browser and account boundaries, QA expectations, packaging standards, customer-delivery requirements, spending boundaries, credential handling, human-takeover requirements, and verification rules.

AGENTS.MD CONTROL PRINCIPLES

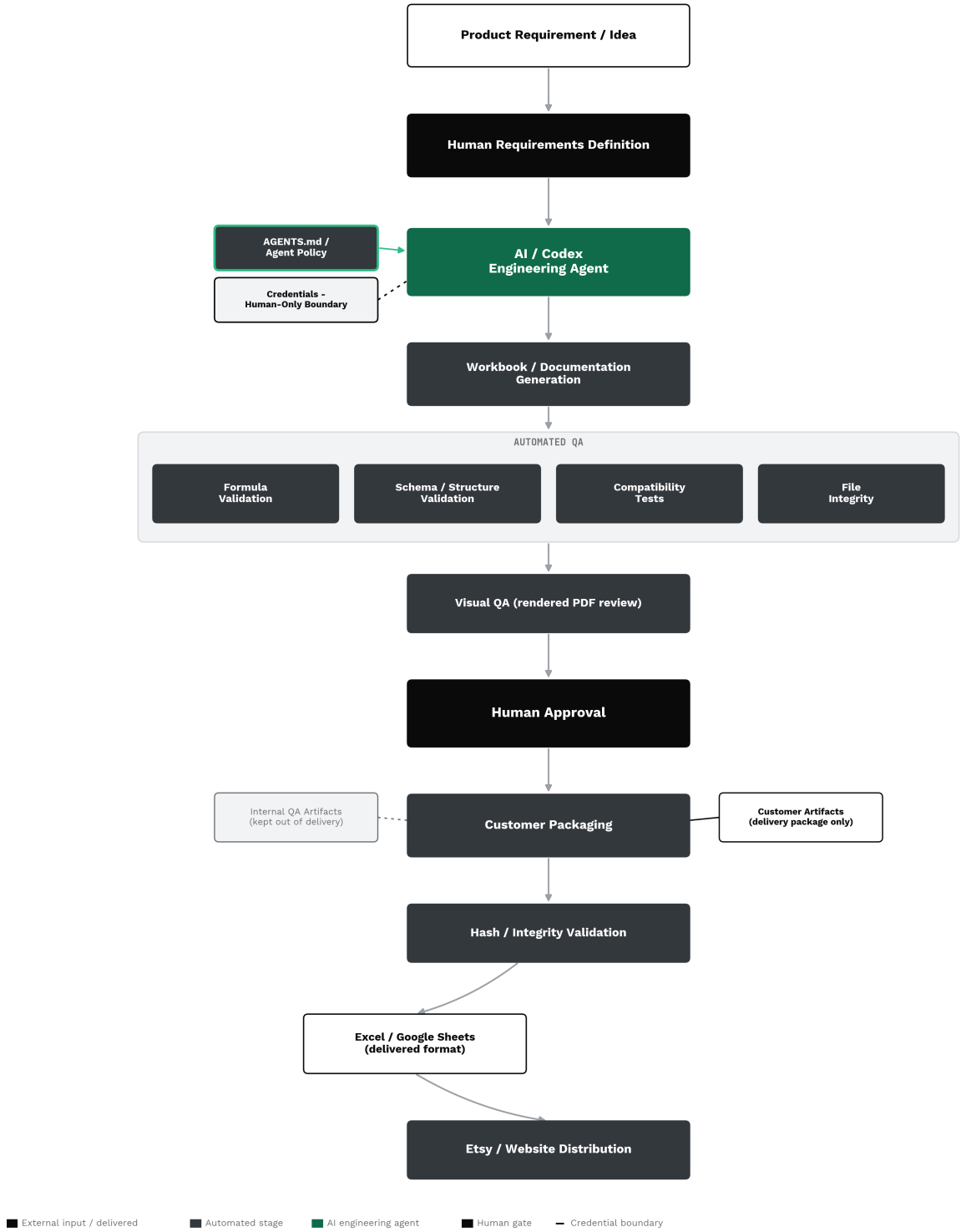
- Never ask for passwords, recovery codes, payment-card data, or MFA secrets
- Pause for human interaction on CAPTCHA, MFA, or payment confirmation
- Never claim success without reopening and verifying the resulting state
- Never upload internal QA or source files as customer-facing products
- Preserve original release archives and validate archive integrity
- Record package hashes for every release
- Keep production artifacts strictly separate from internal engineering artifacts

The direct cybersecurity lesson here: an AI agent should operate under an explicit authorization model, exactly like any other privileged automation - least privilege, explicit boundaries, mandatory human takeover on ambiguity, and verification instead of self-reported success.

Release Architecture

CLEARWARD AI-ASSISTED PRODUCT ENGINEERING & RELEASE PIPELINE

Project 6 - Clearward Release Architecture



Every release moves from requirement through an AI engineering agent, automated and visual QA, and an explicit human approval gate before packaging, hashing, and distribution.

Skills Demonstrated

AI / AUTOMATION

Codex · ChatGPT · Agent Instruction Design · Prompt Engineering

Workflow Decomposition · Human-in-the-Loop Control · Browser Automation Planning

ENGINEERING

Excel · Google Sheets · Structured Data · Formula Logic · Data Validation

QA Automation · File Packaging · Hash Validation · Release Management

SECURITY / GOVERNANCE

Secrets Handling · Authorization Boundaries · Least Privilege

Human Approval Gates · Artifact Separation · Change Validation

Spending / Transaction Authority · Auditability

PRODUCT

Requirements Engineering · User Experience · Product Documentation

Digital Distribution · Compatibility Design · Customer-Support Boundaries

PROJECT 07

Basaltborne

Know Your Gear. Run What Works. - AI-Assisted Publishing Platform

Basaltborne is an AI-assisted outdoor, automotive, and gear publishing platform - initial content domains include Jeep and automotive, camping, outdoor gear, garage and tools, fitness, technology, gear intelligence, field notes, deals, and trail information.

Its real purpose in this portfolio is to demonstrate directing AI coding agents through a realistic software-engineering lifecycle - architecture, implementation, testing, security review, and release - rather than a single prompt-and-done exercise.

DEVELOPMENT MODEL

Development is deliberately phase-gated. AI coding agents can research, propose architecture, implement code, write tests, produce documentation, and troubleshoot failures - but advancing a phase requires defined acceptance criteria, passing tests, human review, source authorization, and explicit security boundaries. The operating principle throughout: AI output is treated as untrusted until validated.

Technical Stack

Next.js / React / TypeScript	application framework and type safety for the public site and admin surfaces
PostgreSQL / Supabase	managed relational database with built-in auth and Row-Level Security
Row-Level Security (RLS)	database-enforced access control that doesn't depend on the app layer getting it right
Database migrations	versioned, replayable schema changes instead of hand-edited production state
API / RPC boundaries	a defined surface of server-controlled operations instead of direct table access
Zod	runtime schema validation for anything crossing a trust boundary
Git / GitHub / GitHub Actions	version control and automated validation on every change
Playwright	end-to-end tests exercising real user-visible flows
Vitest	unit tests for application logic

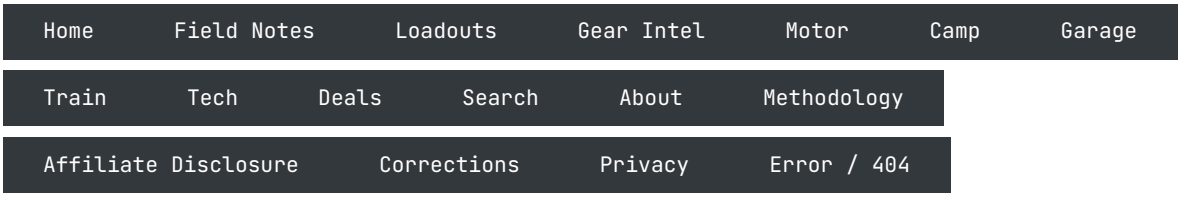
pgTAP	tests written and run inside Postgres to verify database rules and RLS behavior directly
ESLint / TypeScript compiler	static analysis and type-checking as a merge gate
PowerShell / Docker	local tooling and environment consistency

Phase 1 - Foundation

Repository structure, database architecture, authentication and authorization foundations, approval and audit concepts, an admin scaffold, migration discipline, security boundaries, and a testing baseline - the groundwork every later phase depends on. The core lessons were in schema design, migration discipline, authorization, RLS, auditability, and change control.

Phase 2 - Public Experience

The public-facing site: frontend architecture, navigation, responsive design, content architecture, SEO considerations, accessibility, branding, error states, and disclosure/trust content (methodology, affiliate disclosure, corrections policy) built in as first-class pages, not afterthoughts.



Phase 3 - Controlled Published Content

The core distinction in this phase is between draft content and approved/published content: immutable, versioned publication; an explicit approval workflow; public-read boundaries; source and provenance controls; and full auditability. Publication is treated as a privileged state transition enforced server-side - not a boolean flag flipped in the UI.

Database Security

Security concepts implemented directly in the data layer, not left to the application to get right every time:

Row-Level Security	Application access can't rely solely on client-side filtering - the database itself enforces who can read or write which rows.
Private tables	Sensitive or internal data stays behind controlled database access, never exposed as a general-purpose readable table.
Public APIs / RPCs	Only specifically designed public operations are exposed - not direct table access from the client.
Server-only privilege	Privileged service-role credentials are never shipped to the browser.
Input validation	Every untrusted parameter is validated before use, at the boundary.
Fail closed	Invalid or unauthorized states do not silently fall back to insecure behavior.
Migration discipline	Approved migrations are immutable; changes are introduced through additive migrations, never hand-edited after the fact.
Least privilege	Consumers receive only the minimum data actually needed for the operation.
Information minimization	Internal publisher and approval identity information does not leak through public content APIs.

Validation & CI

Testing combined static analysis, unit tests, database-level tests, and end-to-end browser tests, all run independently in GitHub Actions:

97 / 97	**120 / 120**	**28 / 28**	**109**	**35 / 35**
Vitest tests - core Phase 3 validation	pgTAP database tests	Playwright end-to-end tests	Vitest tests after later Trail work expanded the suite	Playwright tests after the same expansion

Test numbers aren't marketing - each layer proves something different:

Unit tests (Vitest)	application logic behaves as specified
pgTAP	database rules and RLS/security behavior hold at the data layer, not just the app layer
Playwright	user-visible, end-to-end behavior actually works

Accessibility checks	keyboard navigation and WCAG-related issues are caught before release
Build / type / lint gates	static and compilation quality, enforced before merge
Migration replay	the database can be rebuilt from migrations alone - deployment is reproducible
GitHub Actions	every check above runs as independent, remote verification, not just “works on my machine”

Git & Release Engineering

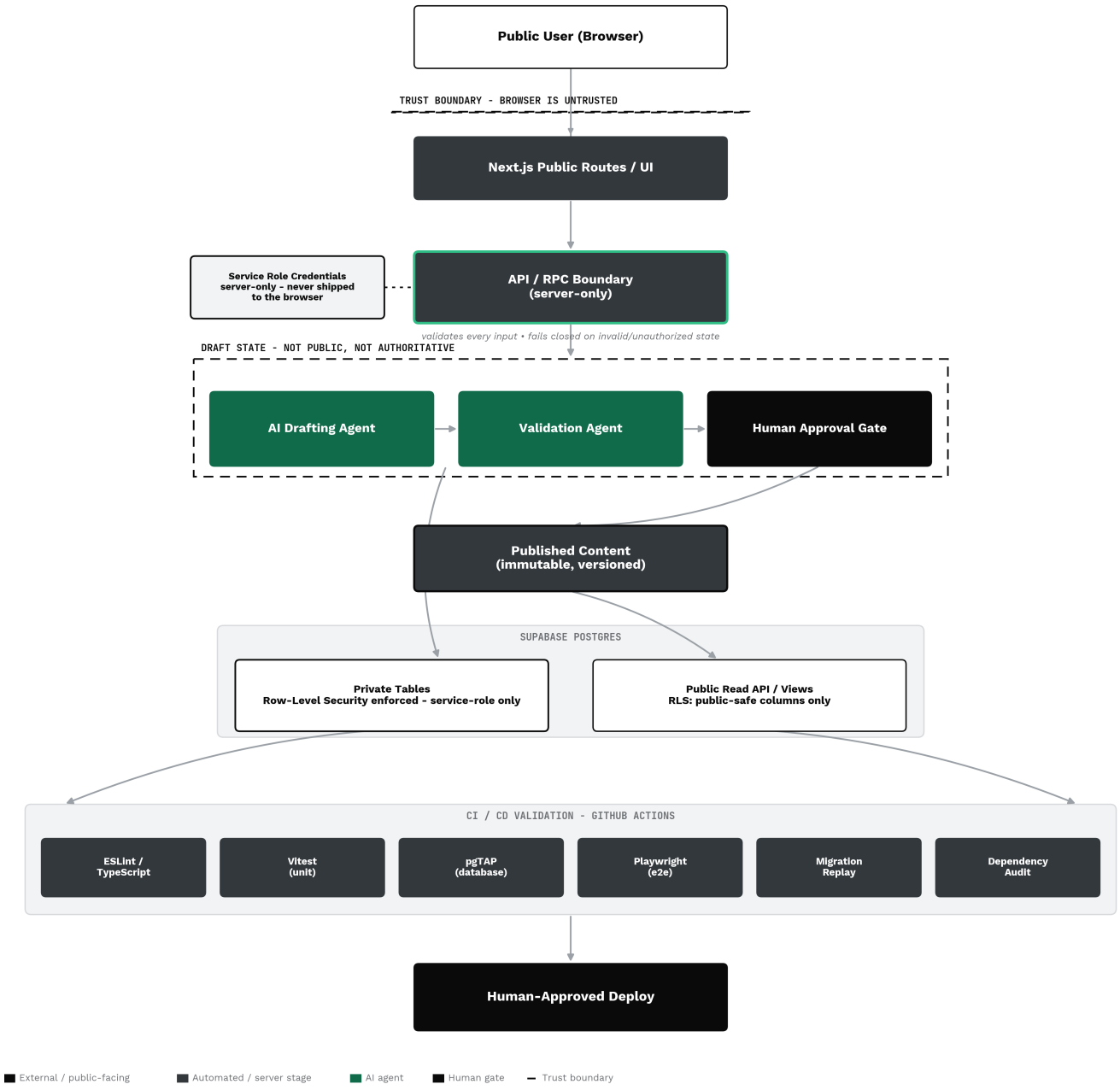
- Feature work happens on branches off main; nothing beyond trivial documentation fixes is committed directly to main.
- Commits are kept small and scoped, with messages tied to the acceptance criteria they satisfy, so the history reads as a record of reviewed, validated changes - not just a save mechanism.
- Every change goes through pull-request review, including AI-agent-authored changes: nothing merges without an explicit review pass against acceptance criteria and test results.
- Main is protected behind the CI/CD validation suite - lint, type-check, unit tests, pgTAP, Playwright, and migration replay all have to pass before a merge is even possible.
- Releases are tagged and tied to a changelog, and every migration and deploy is considered in terms of “how would I undo this” before it ships - rollback is planned for, not improvised.

The pull request is where automated validation and human judgment both have to sign off before anything reaches main - branch protection is the technical backstop for the same human-approval principle used everywhere else in this project.

Secure Publishing Architecture

BASALTBORNE SECURE PUBLISHING & APPLICATION ARCHITECTURE

Project 7 - Basaltborne Trust Boundaries, RLS & Release Validation



The browser is treated as untrusted by default; server-side RPCs, Row-Level Security, and a human approval gate stand between a draft and anything the public can read, with an independent CI/CD suite validating every change before deploy.

Secure Agent Design Principles

These principles are the throughline across every agent design in this portfolio. They exist because an AI agent with tool access is privileged automation, and it should be governed like any other privileged automation.

Least privilege	The agent receives only the permissions required for the current task. Collection, analysis, and remediation should use separate authority where practical.
Human-in-the-loop	Consequential actions such as isolation, process termination, account disablement, or publication require explicit approval.
Deterministic validation	Use code and rules for checks that should not depend on probabilistic model judgment: schema validation, required fields, hashes, and allowlists.
Untrusted-input handling	Logs, documents, websites, and command output may contain instructions intended to manipulate the model. Tool policy must override content.
Observable operation	Log tool calls, inputs, decisions, errors, retries, confidence, approvals, and stopping reasons.
Safe failure	Stop and escalate when evidence is insufficient, tools fail repeatedly, permissions are exceeded, or actions conflict with policy.
Separation of facts and inference	Reports must clearly distinguish collected evidence from model interpretation and analyst conclusions.
Rollback and validation	Any recommended change includes expected results, validation criteria, operational risks, and a rollback procedure.

Testing Strategy

Testing an agent means testing its failure behavior, not just its happy path. The strategy below applies to both proposed agent builds:

- Happy-path functional testing with known-good inputs
- Missing, malformed, stale, and contradictory input testing
- Tool timeout, permission failure, and unavailable-source testing
- Prompt-injection and malicious-content testing
- Retry-limit and safe-stop verification
- False-positive and false-negative review using controlled lab scenarios
- Replayable test cases with versioned prompts and configuration
- Human review of whether the system's reasoning is supported by collected evidence

Cybersecurity Application & Roadmap

Secure system administration, disciplined troubleshooting, evidence handling, and risk-aware change planning are the foundation. AI and agentic systems extend those fundamentals; they do not replace them. Applied to security operations, the practical value is to:

- Accelerate evidence gathering while preserving analyst control.
- Standardize repeatable triage and documentation processes.
- Reduce time spent on low-value formatting and summarization.
- Improve consistency of validation, rollback, and audit documentation.
- Help analysts identify missing evidence and unresolved questions.
- Create safer automation through explicit permissions, controls, and stopping conditions.

NEAR-TERM DEVELOPMENT ROADMAP

Phase	Objective	Deliverable	Success Measure
1. Foundation	Build secure local orchestration and structured logging	Minimal agent framework with approved tools	Every action logged; permissions constrained
2. Publishing Agent	Implement state classification, tool selection, validation, and approval	Working demo using fictional Self Help guides	Correct package generation and safe failure behavior
3. Triage Collector	Build read-only Windows evidence collection	Structured endpoint evidence package	COMPLETE - repeatable output, no destructive actions
4. Triage Reasoning	Add correlation, confidence, and reporting	Analyst-ready triage report	COMPLETE - findings trace directly to evidence
5. Adversarial Testing	Test prompt injection, malformed inputs, and tool failures	Test report and mitigations	COMPLETE - documented findings and fixes
6. Portfolio Demo	Record concise walkthrough and publish sanitized artifacts	Demo video, diagrams, code samples, lessons learned	A reviewer can understand design and controls quickly

Cross-Project Skills Index

The individual project sections show skills in context. This is the same competency set collapsed into one index, grouped by category rather than by project.

AI & AUTOMATION

Prompt Engineering · Context Engineering · AI Task Decomposition · Agent Supervision · Human-in-the-Loop Systems · Model / Tool Selection · Agentic Workflow Design · Multi-Stage LLM Workflows · Agent State Modeling · AI Maturity Classification · Domain-Specific Assistants · Scheduled Automation · Confidence Modeling

SOFTWARE & SYSTEMS ENGINEERING

Requirements Engineering · Systems Architecture · Software Architecture · Data Architecture · Next.js · TypeScript · React · PostgreSQL · Supabase · API Design · Schema Validation (Zod) · Git · GitHub · Branching Strategy · CI/CD · Database Migrations

SECURITY & TESTING

Secure-by-Design Development · Authentication & Authorization · Row-Level Security · Least Privilege · Secret Management · Input Validation · Trust Boundaries · Fail-Closed Behavior · Data Provenance · Prompt-Injection Resistance · Tool Allowlists · Untrusted-Input Handling · Deterministic Validation · Safe Stopping & Retry Limits · Audit Logging · Adversarial Testing · Unit / Integration / E2E Testing (Vitest, Playwright) · Database-Security Testing (pgTAP) · Accessibility Testing · Dependency Auditing

PLATFORM, OPS & PRODUCT

PowerShell · Windows Administration · Hyper-V · Security Event Analysis · Endpoint Telemetry · Microsoft Defender · Sysmon · QA / QC · Release Gates · Rollback Awareness · Runbooks · Excel & Google Sheets Systems · Data Validation & Formula Engineering · Product Documentation

---

CLOSING NOTE

These seven projects sit at different scales - a weekend experiment, a home lab, and two platforms built for real customers - but the underlying discipline doesn't change: define the architecture and the boundaries first, let AI agents do the drafting and the heavy lifting inside those boundaries, validate what comes back with evidence rather than trust, and keep a human decision at every point where the cost of being wrong is real.

The applied examples in this portfolio are real, running workflows, configured assistants, and shipped products. The Knowledge Publishing Agent and Endpoint Triage Agent are proposed builds intended to demonstrate deeper agent engineering. That distinction is intentional and reflects an accurate assessment of current maturity.

Casey Knott - Portfolio compiled September 2026