TECHNICAL PROGRAM DELIVERY • PORTFOLIO CASE STUDY

Leadership Agent

Building an Autonomous Content Operations Pipeline

WHAT THIS DEMONSTRATES

Full lifecycle program delivery, integration management, risk and issue control, quality governance, technical communication, and hands-on technical program ownership.

The Problem

Most AI content tools stop at generation. They can draft text, but they do not govern the work. They do not reliably filter inputs, prevent repetition, enforce brand standards, capture human edits, or improve from operational feedback.

The objective was to build an autonomous content operations pipeline that could research, draft, evaluate, refine, and prepare executive-level LinkedIn content while running fully on-premises.

The goal was not to prove that AI could write. The goal was to prove that an AI workflow could be governed, sequenced, monitored, corrected, and improved like a real delivery system. That became the Leadership Agent.

The Solution and Architecture

The Leadership Agent is a nine-stage autonomous content pipeline with a separate feedback loop for continuous improvement. It runs on an Ubuntu host, uses Ollama with qwen2.5:7b for local inference, ChromaDB for vector storage and semantic similarity, and FastAPI for the service layer.

> The architecture was intentionally local-first. No third-party LLM calls. No per-token billing. No operational data leaving the machine.

1 Researcher › 2 Gatekeeper › 3 Scorer › 4 Strategist › 5 Writer › 6 Critic › 7 Editor › 8 Formatter › 9 Publisher QC

↩ **Feedback Ingester** closes the loop. After human review, edits are compared against the agent baseline and recurring patterns are promoted into the brand guide.

How the Pipeline Works

The **Researcher** pulls articles from 25+ RSS sources. The **Gatekeeper** filters items for brand relevance. The **Scorer** ranks the remaining items and supports deduplication by URL, theme, and semantic similarity. The **Strategist** selects the strongest items and determines the content angle and format. The **Writer** produces the first draft. The **Critic** evaluates the draft across seven quality dimensions and can trigger up to three rewrite cycles. The **Editor** and **Formatter** polish the output for LinkedIn. **Publisher QC** checks the final content against brand and publishing rules. After human review, the **Feedback Ingester** compares the agent output against the edited version, logs the differences, and promotes recurring patterns into the brand guide.

This made the system more than a content generator. It became a governed content delivery workflow.

Delivery Approach

I delivered the work in four phases.

Phase 1: Establish the End-to-End Pipeline

The first priority was a working path from source material to draft output. That meant sequencing ingestion, relevance filtering, scoring, strategy, writing, and output generation before optimizing any individual component. The delivery decision was deliberate: a complete but imperfect workflow would expose integration risk faster than isolated agent development.

Phase 2: Harden Quality Governance

Once the pipeline could generate content, the next priority was quality control. Basic filtering became a stronger governance layer. I added the Gatekeeper, critic scoring, rewrite logic, publisher checks, and brand-rule enforcement so the system could reject weak outputs instead of passing everything forward. This shifted the system from "generate a draft" to "manage content through acceptance criteria."

Phase 3: Improve Reliability and Operational Control

The next set of issues came from real operation. Model calls were slow. Some stages could hang. Duplicate topics surfaced. Some failures were too quiet. I added timeout handling, async-safe execution wrappers, run logging, staged fallbacks, deduplication controls, and clearer failure paths. This was the point where the project moved from prototype behavior to operational discipline.

Phase 4: Close the Human Feedback Loop

The final phase connected human review back into the system. The Feedback Ingester compares the agent baseline against the final human-edited version, identifies recurring changes, and promotes repeated patterns into the brand guide. That turned editorial judgment into structured improvement data.

Risks and Issues Encountered

ISSUE	IMPACT	RESOLUTION
Broken reflexion loop	Drafts could repeat weak approaches after rejection.	Reworked the rewrite flow so rejection reasons and failed-attempt history were passed into later attempts.
Async/sync mismatch	Synchronous model calls blocked async orchestration and created stalled pipeline behavior.	Added async-safe wrappers, executor-based execution, and timeout controls.
Silent credential failure in cron job	Scheduled monitoring appeared healthy while a git-based heartbeat failed silently.	Added explicit success validation and failure alerting to the scheduled job, so a silent failure surfaces instead of reporting healthy.
Duplicate articles and repeated themes	Content output risked becoming repetitive across runs.	Added deduplication by used item, blocked theme, duplicate title, and semantic similarity.
Scoring reliability	Generated content needed a consistent quality bar before human review.	Added seven-axis critic scoring, minimum thresholds, rewrite loops, and Publisher QC.
Human edits were not feeding system improvement	The system could generate content but could not learn from editorial changes.	Added immutable agent baselines and a Feedback Ingester to compare human edits against agent output.

Engineering-to-Governance Translation

TECHNICAL CAPABILITY	PM / GOVERNANCE COMPETENCY DEMONSTRATED
Multi-agent pipeline orchestration	Integration management across dependent workstreams
Local-first inference	Privacy, cost, and deployment constraint management
Gatekeeper filtering	Intake control and quality governance
Semantic scoring and ChromaDB deduplication	Portfolio-level content control and repetition risk reduction
Critic scoring across seven axes	Acceptance criteria and quality assurance
Rewrite loop with failed-attempt history	Defect handling and corrective action
Timeout handling and staged fallback behavior	Operational resilience and issue control
Run logging	Delivery transparency and troubleshooting support
Publisher QC	Final quality gate before release
Feedback Ingester	Lessons learned capture and continuous improvement

What I Would Do Differently

1. Design Observability Earlier

The system became easier to operate after logging, timeout handling, and failure visibility improved. Next time, I would define observability requirements at the start rather than after operational issues surfaced.

2. Separate Evaluation Logic Earlier

The Critic became a core governance component. I would isolate evaluation logic sooner so scoring, thresholds, and rewrite behavior could be tested and tuned independently.

3. Treat Scheduled Jobs as Production Dependencies

The cron heartbeat issue reinforced that background automation needs explicit validation. A scheduled job that fails silently is not automation. It is hidden risk.

4. Define Operational Metrics Before Scaling Usage

The system had functional success before it had a full measurement model. Future iterations would define throughput, latency, pass rate, rewrite rate, and duplicate reduction targets earlier.

5. Add Automated Tests Before Expanding Features

Several issues were discovered through live operation. A stronger test suite around orchestration, scoring, file outputs, and feedback ingestion would reduce future rework.

Outcome

I designed and built a working autonomous content pipeline, then operated it through real delivery issues. The work included architecture, sequencing, local inference, vector-based deduplication, async execution fixes, scoring governance, quality gates, human feedback capture, and operational reliability improvements.

This case study matters because it is not a slideware AI concept. It is a real system with real integration points, real failure modes, and real governance decisions behind it.