Stakeholder Report
AMI Patch Evidence Tracker (Synthetic Data)
Student: Euris Garcia
Date: January 2026

1. Background and Objectives

In many organizations, promoting operating system images (such as AMIs) from DEV to STAGE and PROD requires convincing evidence that security vulnerabilities have been addressed. Today, this "DEV patch evidence" is often compiled manually, based on vulnerability scanner exports and ticketing systems.

This manual process is:

- Time-consuming and error-prone.
- Difficult to reproduce consistently across different teams and sprints.
- Hard to audit, because evidence is scattered across spreadsheets, screenshots,
  and emails.

The objective of this project is to demonstrate, in a safe and synthetic way, how a small internal tool can:

- Track patch events for specific services and environments.
- Generate synthetic BEFORE and AFTER vulnerability snapshots.
- Automatically compute which vulnerabilities were fixed.
- Produce change-request-ready text for STAGE and PROD.
- Enforce a clear, auditable lifecycle for patch promotion.

2. Solution Overview

The "AMI Patch Evidence Tracker (Synthetic Data)" is a small web application built in Python using FastAPI, SQLAlchemy, and a local SQLite database. It provides two main screens:

- Dashboard:
- Lists patch events with filters by service, environment (DEV, STAGE, PROD),
- and lifecycle state.

- Displays summary statistics (counts by environment and lifecycle phase) to
- give a quick overview of patch activity.

- Patch Event Detail:
- Shows metadata (service, environment, AMI ID, patch date, notes).
- Allows the user to generate synthetic BEFORE and AFTER scan snapshots.
- Computes fixed vulnerabilities to serve as DEV evidence.
- Manages the lifecycle using a State Pattern.
- Generates synthetic STAGE and PROD change-request summaries.

All data is synthetic; the application never connects to production tools or environments. CVEs, plugin IDs, hostnames, and AMI IDs are fake-but-realistic identifiers generated for demonstration and training purposes.

3. How It Works – DEV Evidence Workflow

For each patch event, the user follows a simple three-step process:

1) Generate BEFORE snapshot
- The application creates a synthetic BEFORE scan with a set of vulnerabilities of varying severities (Critical, High, Medium, Low).

2) Generate AFTER snapshot
- The application generates an AFTER snapshot that reuses a subset of the BEFORE vulnerabilities to represent the remaining issues.
- The system enforces that AFTER can only be requested after BEFORE exists, via both the user interface (disabled button) and backend checks.

3) Compute fixed vulnerabilities
- The application compares BEFORE and AFTER and identifies vulnerabilities that disappeared between the two snapshots.
- These “fixed vulnerabilities” are presented in a table along with a severity breakdown.
- The patch event is marked as having DEV evidence available.

The interface clearly shows BEFORE/AFTER counts, the list of fixed vulnerabilities, and the severity breakdown. This simulates the evidence patch

teams need to justify promoting an AMI beyond DEV.

4. Lifecycle Management and CR Summaries

The patch lifecycle is implemented with a State Pattern and follows a sequence from DEV to STAGE to PROD and finally CLOSED. Each patch event has a current state, and only valid transitions are allowed (for example, an event cannot be closed until the PROD patching step has been completed).

Once DEV evidence is available and both BEFORE and AFTER snapshots exist, the user can generate:

- A STAGE CR summary:
- Text that explains which synthetic vulnerabilities were fixed and at what severities.
- Intended to be pasted into a change request form for promoting from DEV to STAGE.

- A PROD CR summary:
- Similar text reflecting the same DEV evidence, but written from the perspective of promoting to PROD after STAGE validation.
- Available only in appropriate lifecycle states.

These summaries are generated programmatically based on the computed diff and severity counts, which reduces the risk of manual errors and inconsistent descriptions.

5. Benefits, Limitations, and Future Extensions

Benefits:

- Demonstrates how a lightweight internal tool can standardize and streamline DEV patch evidence capture.

- Encourages a more structured and auditable approach to patch lifecycle management.

- Provides a safe environment to explore these ideas using only synthetic data.

Limitations:

- The current implementation uses synthetic data only and does not integrate with real scanners, ticketing systems, or cloud APIs.

- It focuses on a single type of patch event (AMI-based server images) and a simplified set of lifecycle states and rules.

Potential future extensions:

- Integrating with real vulnerability scanners once appropriate security and data handling measures are in place.

- Extending the domain model to cover additional asset types or patch methods.
- Enhancing reporting, including trend charts, SLA tracking, and exportable evidence packages.

6. Conclusion

The AMI Patch Evidence Tracker (Synthetic Data) shows how Python, FastAPI, and an object-oriented design can be combined to create a practical, safe prototype that automates DEV patch evidence capture. Even without connecting to real systems, it highlights the value of:

- Clear lifecycle rules,
- Automated evidence generation,
- And consistent CR-ready summaries.

These ideas can be adopted and expanded within a production environment using real data sources, while maintaining the same core workflows demonstrated in this project.