

PatchPilot: Diff-Gated Dependency Vulnerability Remediation with Deterministic Planning and Bounded Model-Assisted Source Repair

Sivaprakash J

sivaprakash7223@gmail.com

Anand Institute Of Higher Technology

Research Article

Keywords: software supply chain security, dependency management, vulnerability remediation, software composition analysis, large language model agents, automated program repair, developer tooling

Posted Date: October 27th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11497420/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: The authors declare no competing interests.

PatchPilot: Diff-Gated Dependency Vulnerability Remediation with Deterministic Planning and Bounded Model-Assisted Source Repair

Sivaprakash J

Dept. of Computer Science and Engineering
Anand Institute of Higher Technology (Autonomous)
Kazhipattur, Chennai, India
Roll No. 310123104093, B.E. CSE (2023–2027)
github.com/CoreCoderX

Tamil Arasi

Dept. of Computer Science and Engineering
Anand Institute of Higher Technology (Autonomous)
Kazhipattur, Chennai, India
Project Supervisor

Abstract—Third-party dependency vulnerabilities are detected reliably but repaired badly. Scanners identify the advisory and point at a fixed version range; they rarely establish whether the upgrade is safe for the project that consumes it, and almost never repair the source code that the upgrade breaks. Recent work responds by delegating the decision to large language model (LLM) agents, yet a large-scale study of agent-authored pull requests finds that agents select already-vulnerable versions more often than humans do, and that remediating those selections disproportionately requires major-version upgrades. A benchmark of 203 real dependency-upgrade tasks reports that the strongest coding-agent configuration completes barely half of them, with the dominant failure being an incomplete repository-wide migration.

This paper presents PatchPilot, a Visual Studio Code extension for the npm, Apache Maven and Python/PyPI ecosystems that deliberately separates three decision authorities which existing tools conflate: version selection, source repair and build adjudication. Version selection is fully deterministic—computed from registry metadata and Open Source Vulnerabilities (OSV) advisory evidence, restricted to exact version pins, and rendered as a reviewable diff before any byte is written. Build adjudication belongs to the project’s own test commands, executed without a shell from a fixed template table. Source repair is the only task delegated to a model, and it is confined by a hard never-editable manifest boundary, a path allowlist, exact-once edit semantics and byte budgets, behind a per-request consent gate. Every write is preceded by a SHA-256 snapshot, so rollback restores byte-identical content and detects files the developer edited in the meantime.

We evaluate PatchPilot as a design-science artifact: we state the invariants the architecture is required to satisfy, trace each to its implementing mechanism, and verify them through 252 pure unit tests that run without VS Code or network access, extension-host smoke tests, four reproducible sample projects and a headless end-to-end runner. We report the properties that hold by construction, identify the security properties that do not hold, and outline the benchmark-based empirical evaluation that remains future work.

Index Terms—software supply chain security, dependency management, vulnerability remediation, software composition analysis, large language model agents, automated program repair, developer tooling

I. Introduction

Modern applications are assembled rather than written. A mid-sized service typically resolves hundreds of direct packages into thousands of transitive ones, and empirical comparisons of packaging ecosystems show that these dependency networks grow continuously while a high proportion of packages become fragile precisely because their transitive closure keeps expanding [1]. Every one of those packages is a piece of software the consuming team did not write, did not review and cannot easily audit, yet each is a potential entry point into the build.

When a vulnerability is disclosed in such a package, the developer is left with three questions that current tooling answers unevenly at best:

- 1) Does this affect my project? Scanners report at the coordinate level. An advisory against a transitive dependency that the project never calls produces the same alert as one against a function invoked on every request. Sustained noise teaches teams to distrust—and eventually to ignore—the tool.
- 2) Which version actually fixes it? The answer is often a range, or the string “latest”, or a major-version jump whose migration cost is nowhere quantified.
- 3) Will applying the fix break me? This question is almost never answered at all.

The third question is the one that matters. The honest answer—unknown until your own tests run—is also the least commercially attractive thing an automated tool can say, so tools compensate by behaving as though the answer were “no”. They rewrite the version string, open a pull request, and if the build turns red the developer is left debugging a diff they neither wrote nor understood. This asymmetry is the central observation behind our work: the version bump is the cheap part and the code change is the expensive part, and a tool that handles only the cheap part

while implying it handled both is actively misleading.

A. Two research responses, and the gap between them

The literature has responded along two largely disjoint tracks.

The first track improves analysis. Song and Zhu build a method-level call graph, map it onto the dependency tree, and derive a vulnerability propagation graph that distinguishes reachable from unreachable vulnerable APIs; on 1,273 vulnerabilities their three remediation strategies—removing unreachable vulnerable dependencies, upgrading to the nearest non-vulnerable version while pinning transitive dependencies, and excising unreachable vulnerable code where no patch exists—resolve 7.4%, 66.6% and 15.0% of cases respectively [2]. Zhang, Ye and Liu go further and unify dependency upgrading with patch porting, using LLM-driven agents to gather project-specific evidence and an SMT solver to choose a cost-aware remediation plan over 301 real Maven projects [3]. Both are powerful and both are bound to a single ecosystem, and neither is delivered inside the developer’s editing environment.

The second track improves automation. Tawosi et al. combine a Summary Agent, a Control Agent and a Code Agent with migration documentation and a codebase-summarisation mechanism to apply Java dependency upgrades, reporting 71.4% precision against a manual gold standard [4]. Fruntke and Krinke compare an agentic repair system against recursive zero-shot prompting on the BUMP benchmark of curated breaking changes and obtain test suite success rates of at most 23% and 19% respectively [5]. Reyes et al. report that the best of five LLMs completely fixes 27% of builds and 78% of individual compilation errors when prompted with API differences and build output [6]. Bartlett, Liem and Panichella apply retrieval-augmented generation with execution feedback to Python dependency conflicts and outperform two prior systems by 15.97 and 21.58 percentage points of fix rate [7].

Two recent empirical results complicate the second track. Singla et al. analyse 117,062 dependency changes drawn from 33,596 agent-authored and 6,618 human-authored pull requests across 2,807 repositories and seven ecosystems. Agents select versions that are already known to be vulnerable at pull-request time more often than humans (2.46% versus 1.64%), those selections are more disruptive to remediate (36.8% require a major upgrade versus 12.9%), and at aggregate level agent-driven dependency work produces a net increase of 98 vulnerabilities while human-driven work produces a net reduction of 1,316 [8]. Independently, Luo et al. introduce DepBench, 203 oracle-validated dependency-upgrade tasks spanning npm, Maven, Go, Cargo and Python; the strongest completed agent configuration solves 104 of them (51.2%), and the characteristic failure is not a wrong version but an incomplete propagation of the upgraded contract through wrappers, types, fixtures and generated artifacts [9].

Read together, these results define the gap this paper addresses. Analysis-side systems are deep but ecosystem-bound and detached from the editor. Automation-side systems are general but delegate version choice to a component that measurably chooses badly, and they succeed on roughly a quarter to a half of real tasks. What is missing is an artifact that spans several ecosystems, keeps the version decision auditable and deterministic, confines the model to the one sub-task where it demonstrably helps, and treats every automated edit as provisional—diffed, approved, snapshotted and reversible—until the project’s own tests have adjudicated it.

B. This paper

We present PatchPilot, a TypeScript extension for Visual Studio Code that reads package.json with its lockfile, pom.xml, requirements*.txt and PEP 621 pyproject.toml; reports OSV advisories, outdated versions and npm deprecations together with the source URL and timestamp behind each finding; plans an upgrade per vulnerable exact pin; presents a diff and requests a single approval for the whole edit set; snapshots every file before writing; validates by running the project’s own commands with no shell involvement; and only then, only on failure, only with explicit per-send consent, offers a model the chance to propose source repairs inside hard boundaries.

PatchPilot is built around a refusal. The model is never asked to choose a dependency version, to judge an advisory, or to decide whether a build passed. Those three authorities are separated by construction, and Section V shows how the separation is enforced in code rather than merely asserted in a prompt.

1) Contributions:

- C1. An authority-separation architecture for dependency remediation. We decompose remediation into three disjoint decision authorities—version selection, source repair, build adjudication—and assign each to the component best able to make it auditable: deterministic computation, a bounded model call, and the project’s own test suite (Sections III–IV).
- C2. A fail-closed advisory evidence model. Advisory lookup failure is a first-class reporting state (unavailable) rather than an absence of findings, so a transient registry outage can never be rendered to the developer as an all-clear. Every finding carries its provenance (Section V-C).
- C3. A diff-gated, hash-verified write protocol. All manifest mutation passes through a plan→diff→approval→snapshot→apply sequence with SHA-256 verification on rollback and explicit conflict detection for files the developer changed afterwards (Section V-E).
- C4. A structurally bounded model-repair channel. We describe and justify four independent constraints—a never-editable manifest regex, a path allowlist derived

from the request itself, exact-once search/replace semantics, and per-attempt file and byte budgets—plus per-send consent that names the endpoint, host, model, files and exclusion patterns before transmission (Section V-G).

- C5. A shell-free validation executor. Build and test commands are drawn from a closed table of templates rather than assembled from strings, eliminating a command injection surface in the one component that must run project code (Section V-F).
- C6. A construction-based evaluation and an explicit statement of its limits. We formalise seven invariants the architecture must satisfy, trace fourteen design requirements to implementing mechanisms and verification methods, and report the security properties that remain unachieved (Sections VI–VII).

2) Scope: PatchPilot is a local tool. The repository contains the entire extension: there is no backend service, no companion API and no cloud component. The only network traffic is to public registries (registry.npmjs.org, repo1.maven.org, pypi.org), to api.osv.dev, and—when the developer explicitly consents, per request—to a developer-configured OpenAI-compatible inference endpoint.

The remainder of this paper is organised as follows. Section II surveys related work and positions PatchPilot against it. Section III states design principles and derives requirements. Section IV presents the architecture. Section V details each mechanism, with algorithms. Section VI gives the threat model. Section VII evaluates the artifact and its limits. Sections VIII–IX discuss, and conclude.

II. Related Work

A. Software composition analysis and compatible remediation

Software composition analysis (SCA) tools—OWASP Dependency-Check [10], Dependabot [11], Renovate [12], Snyk [13] and Eclipse Steady [14] among them—establish which declared coordinates appear in an advisory database. They are effective at detection and largely silent on consequence. The advisory databases themselves have converged on open schemas: the Open Source Vulnerabilities (OSV) schema [15] and its api.osv.dev batch endpoint [16], and the GitHub Advisory Database [17], which PatchPilot consumes indirectly through OSV.

Song and Zhu [2] observe that SCA remediation can itself introduce incompatibilities, dependency conflicts and new vulnerabilities, and address this with VulnScanPro: a fine-grained method graph is mapped onto the dependency tree to yield a vulnerability propagation graph, which separates reachable from unreachable vulnerable APIs. Their second strategy—upgrading to the closest non-vulnerable version while pinning the transitive dependencies that the upgrade would otherwise move—is the closest analogue in the literature to the compatibility concern

PatchPilot handles by refusing to touch anything but an exact pin and by stating explicitly in its approval dialog that target-version compatibility is not verified. Their work is Java/Maven-scoped and produces a standalone HTML visualisation rather than an in-editor workflow.

Zhang, Ye and Liu [3] present Remedius, which argues that upgrade-only remediation is fundamentally limited by secure-version availability, version pinning and API incompatibility, and therefore unifies upgrading with patch porting. LLM-driven agents cluster dependencies by usage, collect project-specific evidence about compatibility, reachability and patch difficulty, and translate that evidence into constraints for an SMT solver that optimises at the vulnerability rather than the dependency level. Across 301 real Maven projects Remedius fixes more vulnerabilities with fewer build and test failures than prior baselines. This is a substantially stronger optimisation formulation than PatchPilot’s greedy candidate scan; PatchPilot compensates by spanning three ecosystems and by keeping every decision inspectable in the editor rather than solver-internal.

B. LLM agents for dependency upgrades and breaking-change repair

Tawosi et al. [4] frame a code update as a sequence of bug-fixing tasks and build LADU from three cooperating agents over a Meta-RAG summarisation of the codebase, iterating compile-and-test until green and handing over to a human when progress stalls. On three synthetic repositories derived from an industrial microservice framework, covering version transitions 3.1→3.2, 3.2→3.3 and 3.3→3.4, LADU reaches 71.4% precision against a manual gold standard while using an order of magnitude fewer tokens than an OpenHands baseline [18]. The authors note that success depends on the quality of the project’s unit tests and that validation on synthetic repositories limits generalisation claims. PatchPilot shares the iterate-against-tests loop but inverts the trust arrangement: the model never proposes the version, and the loop is advisory rather than autonomous.

Frontke and Krinke [5] compare an agentic system with tool access against recursive zero-shot prompting on BUMP [19], a benchmark of 571 reproducible breaking dependency updates in Maven projects. Test suite success peaks at 23% for the agentic approach and 19% for zero-shot, while zero-shot additionally reaches compilation success in a further 43% of cases; a model’s Java-AST manipulation capability correlates with repair success at $r = 0.702$. They rely on Hejderup and Gousios [20], who show across 521 Java projects that test suites are a usable but imperfect oracle for dependency compatibility because median dependency coverage is low. PatchPilot adopts the same oracle while treating its imperfection as the reason to make the resulting state fully reversible.

Reyes et al. [6] evaluate five LLMs on BUMP with prompts combining the buggy line, API differences, error

messages and step-by-step reasoning instructions; OpenAI’s o3-mini completely fixes 27% of builds and 78% of individual compilation errors. The consistent finding across [5], [6] is that contextual build information is what makes repair tractable—which is exactly the information PatchPilot’s repair channel is budgeted to transmit.

C. Dependency resolution and conflict repair

Bartlett, Liem and Panichella [7] target a different sub-problem: making a Python snippet installable at all. PLLM proposes module combinations, observes execution feedback in a container, and refines using parsed error messages, with retrieval augmentation consistently improving fix rates and Gemma-2 9B performing best when combined with RAG. On the Gistable HG2.9K collection PLLM exceeds ReadPyE by 15.97 and PyEGo by 21.58 percentage points of fix rate. PatchPilot deliberately does not attempt resolution. It never runs a solver, never executes an installer and never edits a range constraint; where resolution work would be required, PatchPilot reports the dependency as blocked and stops.

D. Security of agent-authored dependency changes

Singla et al. [8] provide the strongest empirical motivation for constraining what an agent may decide. Their study of 117,062 dependency changes finds that updates constitute a larger share of agent-authored edits than human-authored ones (25.5% versus 15.8%), multiplying the number of moments at which a risky version choice can enter; that agents pick known-vulnerable versions more often (2.46% versus 1.64%); that 71.8% of all extracted changes occur in npm, followed by PyPI, Go, Maven and NuGet; and that agent work nets −98 vulnerabilities against +1,316 for human work. The authors conclude that pull-request-time vulnerability screening and registry-aware guardrails are needed. PatchPilot is an implementation of that recommendation at an earlier point in the lifecycle—before the edit exists—and the ecosystem distribution they report matches PatchPilot’s coverage choice: npm and PyPI first, Maven next.

E. Benchmarks for upgrade repair

DepBench [9] isolates dependency-upgrade repair by separating, for each task, the manifest version bump, the developer’s source repair, and a held-out test patch, then validating with a four-state oracle that the hidden failure is genuinely caused by the upgrade and that the repair is necessary and sufficient. The 203 released tasks comprise 68 npm/yarn, 65 Maven/Java, 40 Go, 20 Cargo/Rust and 10 Python cases mined from dependency-update bot pull requests. Best-configuration solve rate is 51.2%, and performance varies sharply by model, harness and ecosystem, with Maven/Java most often solved and Cargo and npm least. The diagnostic finding most relevant to PatchPilot is qualitative: agents frequently locate the correct dependency surface but stop short of propagating

the change across wrappers, type definitions, fixtures and generated artifacts. PatchPilot’s repair loop does not attempt repository-wide migration either; it targets only the files implicated by the failing command’s output or by mention of an upgraded dependency name, and it reports failure rather than partial success. DepBench is the natural vehicle for the empirical evaluation PatchPilot has not yet performed.

F. Positioning

Table I summarises the comparison. The dimensions are chosen to expose the specific gap: no prior artifact combines multi-ecosystem detection, deterministic and inspectable version selection, source repair, project-test validation, structural reversibility and in-editor delivery. Systems that are strong on analysis ([2], [3]) are single-ecosystem and offline; systems that are strong on automation ([4]–[7]) delegate version decisions or are evaluated on narrow slices; the empirical studies ([8], [9]) characterise the problem without implementing a remediation workflow.

III. Problem Formulation and Design Requirements

A. Formalisation

Let a workspace W contain a set of folders F . Each folder may declare dependencies through one or more manifests $M = \{m_1, \dots, m_n\}$, where each m_i belongs to an ecosystem $e(m_i) \in \{\text{npm}, \text{maven}, \text{pypi}\}$. Parsing M yields a declared dependency set

$$D = \{d = (\text{name}, \text{spec}, \text{resolved}, \text{class}, \text{origin})\},$$

where *spec* is the constraint as written by the developer, *resolved* is the concrete version recovered from a lockfile or registry where possible, *origin* is the manifest that declared d , and

$$\text{class}(d) \in \{\text{exact}, \text{range}, \text{unpinned}, \text{managed}, \text{unresolved}\}$$

classifies how tightly the version is pinned.

An advisory oracle $\mathcal{A}(\text{name}, \text{version})$ returns a (possibly empty) set of advisories, each annotated with an identifier, a source URL, a query timestamp and a severity. A registry oracle $\mathcal{R}(\text{name})$ returns the published version list and ecosystem-specific metadata such as npm deprecation messages. Both oracles may fail, and we treat failure as a distinct outcome rather than as the empty set:

$$\mathcal{A}(n, v) \rightarrow \mathcal{P}(\text{Advisory}) \cup \{\perp\}.$$

Each dependency receives a status drawn from

$$\sigma(d) \in \{\text{vulnerable}, \text{deprecated}, \text{outdated}, \text{clean}, \text{blocked}, \text{unavailable}\}, \quad (1)$$

where *blocked* denotes a dependency that is a legitimate finding but is not eligible for automated editing (Section V-B), and *unavailable* denotes that at least one oracle returned $\perp$ for it.

TABLE I: Positioning of PatchPilot against the surveyed literature. $\checkmark$ = supported, $\circ$ = partial or conditional, $\times$ = not supported. “Deterministic version selection” means the chosen version is computed from registry and advisory data by a stated rule rather than proposed by a model. “Structurally reversible” means prior file content is captured and hash-verified before any write, with user-modified files detected rather than overwritten.

Work	Ecosystem	Detects vulns.	Deterministic version selection	Edits manifest	Repairs source	Test oracle	Struct. reversible	In IDE
VulnScanPro [2] (2026)	Maven	$\checkmark$	$\checkmark$	$\checkmark$	$\circ$	$\times$	$\times$	$\times$
Remedius [3] (2026)	Maven	$\checkmark$	$\circ$	$\checkmark$	$\checkmark$	$\checkmark$	$\circ$	$\times$
LADU [4] (2025)	Maven	$\circ$	$\times$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$	$\times$
Fruntker & Krinke [5] (2025)	Maven	$\times$	$\times$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$	$\times$
Byam [6] (2025)	Maven	$\times$	$\times$	$\checkmark$	$\checkmark$	$\checkmark$	$\times$	$\times$
PLLM [7] (2025)	PyPI	$\times$	$\times$	$\checkmark$	$\times$	$\circ$	$\times$	$\times$
Singla et al. [8] (2026)	7 ecosystems	$\checkmark$	—	—	—	—	—	—
DepBench [9] (2026)	5 ecosystems	—	—	—	—	$\checkmark$	—	—
PatchPilot (this work)	npm, Maven, PyPI	$\checkmark$	$\checkmark$	$\checkmark$	$\circ$	$\checkmark$	$\checkmark$	$\checkmark$

For an eligible dependency, a candidate set is drawn from the registry, $C(d) = \text{top}_k(\{v \in \mathcal{R}(d.name) : v > d.resolved, v \text{ stable}\})$ for a fixed budget k , and the target is

$$S(d) = \{v \in C(d) : \mathcal{A}(d.name, v) = \emptyset\},$$

$$\tau(d) = \begin{cases} \max S(d), & S(d) \neq \emptyset, \\ \perp, & \text{otherwise.} \end{cases} \quad (2)$$

A plan is $P = \{(d, \tau(d)) : \tau(d) \neq \perp\}$, which materialises into an edit set $E = \{f \mapsto (before_f, after_f)\}$ over byte strings.

Applying E produces a workspace state that must be adjudicated. Let $\mathcal{C}$ be a closed set of validation command templates and $\mathcal{J} : \mathcal{C} \rightarrow \{\text{pass}, \text{fail}\}$ the adjudication derived from process exit status. If $\mathcal{J} = \text{fail}$, a repair channel $\mathcal{M}$ may propose a further edit set E' over source files. The two edit sets are governed by disjoint constraints, which is the point of the design.

B. Design principles

a) DP1 Authority separation: Three decisions are required during remediation: which version to install, which source lines to change, and whether the result is acceptable. These have different epistemic characters. Version selection is a lookup problem with a public answer; it should be computed and should be reproducible. Source repair is a generation problem with no public answer; it is where a model adds value. Acceptability is a judgement problem whose only legitimate arbiter is the project’s own test suite, because only that suite encodes the project’s intent. Conflating them—in particular, letting the generator also be the looker-upper or the judge—produces a decision that cannot be audited after the fact.

b) DP2 Fail-closed evidence: Where an oracle cannot answer, the tool must say so. Rendering a network error, a rate limit or a malformed advisory response as “no known vulnerabilities” converts an infrastructure incident into a false all-clear in a security tool, which is the worst available failure mode.

c) DP3 Reversibility by default: An automated edit is provisional until adjudicated. Every write must be preceded by a captured copy of the prior content, and restoration must be verified rather than assumed. A tool that cannot undo is asking the developer to accept an irreversible transaction on the basis of a heuristic.

d) DP4 Least privilege for the model: The model receives the minimum context sufficient for the task and is denied every capability that is not strictly required. Concretely: it cannot see a manifest it is not being asked about, cannot name a file that was not offered to it, and cannot express an edit that matches ambiguously. Boundaries are enforced by the parser that consumes the model’s reply, not by the prompt that produces it.

e) DP5 No implicit execution of project code: Reading a workspace must not run it. Resolving a dependency graph through the project’s own package manager means executing that project’s lifecycle scripts, which is precisely the capability an untrusted folder should not be granted. Analysis is therefore performed by parsing.

C. Derived requirements

Table II states the requirements that follow from DP1–DP5. These form the contract against which Section VII evaluates the implementation.

IV. System Architecture

PatchPilot is a single TypeScript program that runs inside the VS Code extension host. It has no backend,

TABLE II: Design requirements derived from principles DP1–DP5, with the mechanism that realises each. “§” gives the section in which the mechanism is specified.

ID	Requirement	Mechanism	§
DR1	Read npm, Maven and PyPI manifests and lockfiles without executing a package manager	Own parsers (JSON, XML, TOML)	V-A
DR2	Classify every declared dependency by pin tightness	pinClass taxonomy	V-B
DR3	Only automatically edit dependencies whose version is exactly pinned	Eligibility gate in planner	V-B
DR4	Attach provenance (source URL, query time, reason) to every finding	Evidence records	V-C
DR5	Report oracle failure as unavailable, never as clean	Three-state advisory result	V-C
DR6	Choose a target version by a stated rule over registry and advisory data	Eq. (2), Alg. 1	V-D
DR7	Show a diff for every file to be written and obtain one approval for the whole set	Diff preview + single consent	V-E
DR8	Snapshot prior content with a digest before every write; verify on restore	SHA-256 snapshot store, Alg. 2	V-E
DR9	Detect, and refuse to overwrite, files modified by the developer after the tool wrote them	afterHash conflict check	V-E
DR10	Execute validation commands from a closed template table, with no shell	shell:false executor	V-F
DR11	Prevent the model from editing manifests or any path not supplied to it	Manifest regex + path allowlist	V-G
DR12	Prevent ambiguous or sweeping model edits	Exact-once match semantics	V-G
DR13	Bound model context by file count and byte volume; obtain consent per send	Budget constants + consent dialog	V-G
DR14	Refuse to analyse an untrusted workspace; never log secrets	Trust gate, redacting logger	V-H

no database and no cloud component; all state lives in the workspace and in VS Code’s own storage facilities. Figure 1 shows the layering, and Figure 2 shows the end-to-end control flow.

A. Layering

The codebase is organised into five layers with strictly downward dependencies (Table III). Two properties of this

TABLE III: Architectural layers, their source locations and their responsibilities. Dependencies point strictly downward; the policy layer is consulted by every layer above it.

Layer	Location	Responsibility
Composition	extension.ts	Activation, command registration, view wiring, refresh triggers
UI	ui/, commands/	Tree views, prompts, diff preview, progress reporting
Domain	analysis/, upgrade/, ecosystems/	All decisions: parsing, classification, version selection, plans, edits, adjudication
I/O	registry/, advisories/, ai/	Bounded network clients for registries, OSV and the model endpoint
Policy	workspace/, security/, secrets/, config/	Trust gate, log redaction, context exclusions, key storage, validated settings limits

decomposition are load-bearing for the design principles of Section III.

First, the domain layer contains every decision. Version eligibility, candidate selection, target choice, edit construction and adjudication logic are all pure functions of parsed workspace state and oracle responses. Nothing in the UI layer computes a version, and nothing in the I/O layer decides whether a version is acceptable. This is what makes DP1 enforceable: to change how a version is selected, one must edit a domain function that can be called directly from a unit test.

Second, the I/O layer is the only place network access occurs, and every client in it is bounded—timeouts, response size caps and retry limits are enforced by a shared HTTP helper. The model client lives in this layer and is reachable only from one domain function (upgrade/repair.ts), which is what makes DP4 a structural property rather than a convention.

B. The model/view split

Within the UI layer, every view is preceded by a pure model. Files named *Model.ts contain no import ... from 'vscode' and expose functions from domain state to plain view-model data; files named *View.ts are thin adapters that translate a view model into a TreeItem. The four views—workflow, overview, dependencies and plan—each follow this pattern.

The motivation is testability rather than aesthetics. Because a model is a pure function, the entire presentation logic can be exercised by the unit test suite without an extension host, and because commands receive their collaborators as an explicit dependency object rather than importing singletons, a command flow can be driven end-to-end against an injected fake UI and fake runners. There is no dependency-injection framework and no global mutable state beyond the three values the views legitimately

need to share across refreshes (lastResult, latestRecord, lastTest).

C. Control flow

Figure 2 traces a complete remediation. Three gates interrupt the flow, and their placement is deliberate.

The approval gate (S5) sits after planning and before any snapshot, so the developer reviews a fully materialised diff set—not a summary, and not a version number. Approval is requested once for the whole edit set, because a plan whose parts can be approved individually can be applied in an order that leaves the workspace in an intermediate, unvalidated state.

The adjudication gate (S9) is the project’s own test commands. PatchPilot does not interpret test output beyond the process exit status; it does not attempt to decide whether a failing test is “related” to the upgrade. That inference is exactly what a heuristic would get wrong.

The consent gate (R1) sits before any byte leaves the machine. It is requested per send rather than per session, because the content of each send differs and a blanket consent cannot be informed about it.

V. Design and Implementation

This section specifies each mechanism of Figure 2 and states the requirement from Table II that it discharges.

A. Manifest parsing without a package manager

DR1. PatchPilot never invokes npm install, mvn resolution or pip to learn what a project depends on. Resolving a dependency graph means executing the project’s package manager, which in every ecosystem under consideration means executing project-defined lifecycle scripts: npm preinstall/postinstall hooks, Maven plugin goals, and Python build backends. A tool whose purpose is to handle untrusted dependencies safely cannot open by executing code from the folder it is inspecting; that would make the tool itself the most privileged actor in the workspace. DP5 therefore fixes the analysis strategy: parse, never resolve.

Three parsers cover the supported surface. package.json and package-lock.json (schema versions 1, 2 and 3) are parsed as JSON, with the lockfile providing the resolved version for each declared name and the v1 dependencies map handled as a legacy fallback. pom.xml is parsed with fast-xml-parser [26], walking <dependencies> and <dependencyManagement> while recording whether a version came from the former, the latter, or a parent-inherited property. requirements*.txt and PEP 621 [22] pyproject.toml (parsed with smol-toml [27]) contribute the Python surface, including the distinction between == pins and specifier sets.

Parsing buys a second property beyond safety: it preserves distinctions that a resolver flattens away. A resolver answers “what will be installed”; PatchPilot must answer “what did the developer declare, and how tightly”. Those are different questions, and the second is the one that determines what the tool may safely change.

TABLE IV: Pin classification and the resulting treatment. Only exact pins are eligible for automated editing (DR3); every other class is reported but never written (DR2).

Class	Example	Treatment
exact	"4.17.15", requests==2.19.0	Eligible: may be planned and edited
range	^4.17.0, ~2.19	Reported, blocked
unpinned	latest, *	Reported, blocked
managed	version from dependencyManagement / BOM	Reported, blocked
unresolved	declared but no resolvable version	Reported as unavailable

B. Pin classification and the eligibility gate

DR2, DR3. Every declared dependency is assigned exactly one class (Table IV). The gate is deliberately coarse: it is a permission decision, not a prediction. A caret range such as ^4.17.0 may in principle already admit a fixed release through ordinary installation, and editing it would be redundant; a version inherited from a Maven <dependencyManagement> block or a BOM may be shared by several modules, so editing one declaration would silently desynchronise the others. In both cases PatchPilot reports the dependency as blocked: a real finding with an explicit reason, never an automatic edit.

C. Advisory evidence and fail-closed reporting

DR4, DR5. Advisories are retrieved from OSV [16], which exposes a batch query endpoint accepting a list of package name and version pairs and returning matching advisories in the open OSV schema [15]. PatchPilot submits only names and version strings; no file content, no paths, no workspace identifiers accompany the request, which is what makes the query safe to issue without consent.

The response is modelled as a three-valued outcome per dependency: a (possibly empty) advisory set, or the failure value $\perp$. Three consequences follow.

Status. A dependency whose advisory lookup returned $\perp$ receives status unavailable, and every view that renders it renders the unavailability. There is no path in the assessment code by which $\perp$ collapses to the empty set, because the status type does not admit it.

Provenance. Each finding is packaged with the evidence that produced it: the source identifier and URL, the timestamp of the query, and the predicate that raised the finding (advisory match, deprecation notice, version comparison). The dependencies view renders this as a tree under each finding, so a developer can trace any claim in the UI back to the response that supports it.

Target screening. The same three-valued discipline is applied to candidate versions in Section V-D: a version whose advisory status is unknown is not selectable. Fail-

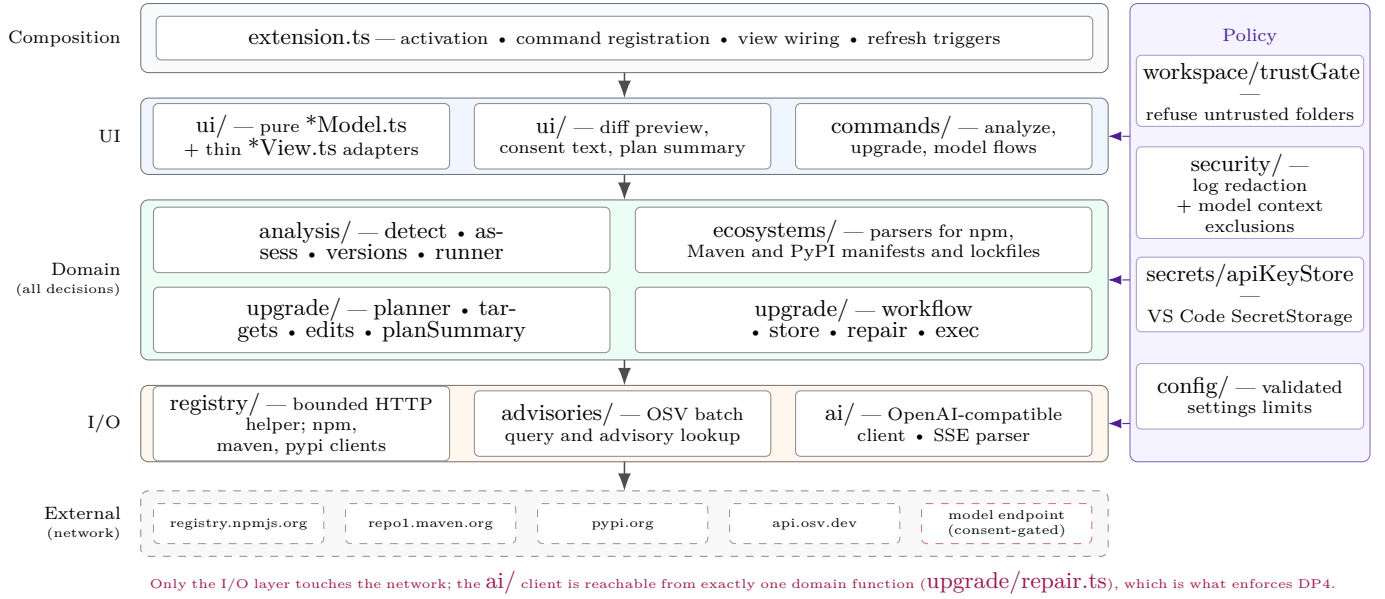


Fig. 1: Layered architecture of PatchPilot. Dependencies point downward; the policy spine is consulted by every layer. Only the I/O layer performs network access, and the model client is reachable from exactly one domain function, which is what makes the least-privilege principle (DP4) a structural property of the program rather than a coding convention.

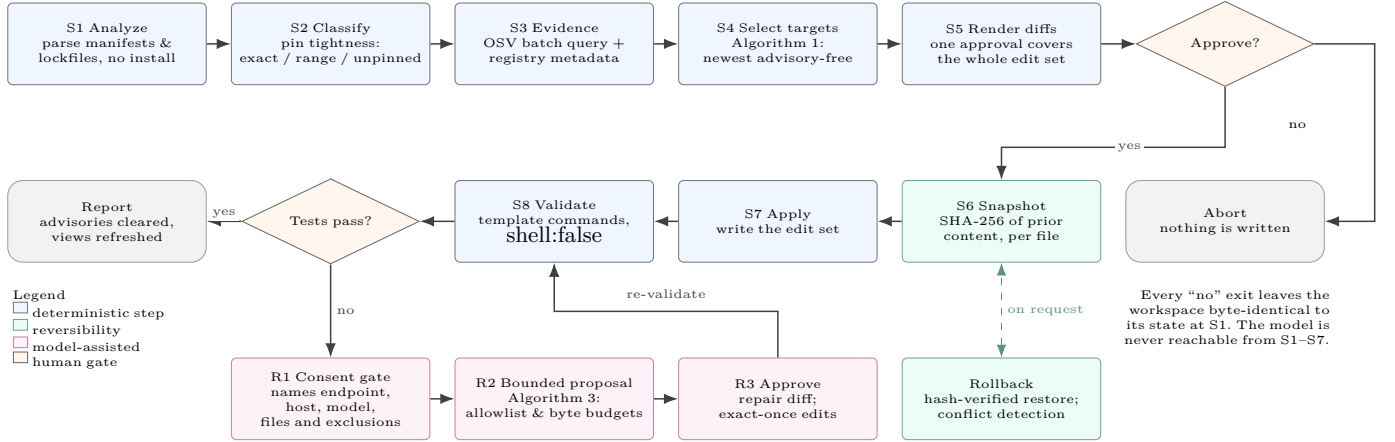


Fig. 2: The controlled repair pipeline. The main spine (S1-S10) is fully deterministic; the model participates only in the repair loop (R1-R3), which is entered exclusively after adjudication failure and exclusively with per-send consent. Any rejection terminates the flow with the workspace byte-identical to its initial state.

closedness is therefore a property of selection as well as of reporting.

D. Candidate selection

DR6. For each eligible dependency PatchPilot draws the published version list from the registry, filters to releases newer than the current one that are not pre-release or otherwise flagged [23], sorts descending, and retains the first $k = 5$ as the candidate window `MAX_TARGET_CANDIDATES`. It then walks the window from newest to oldest, querying OSV for each, and returns the first version whose advisory set is provably

empty (Algorithm 1, Eq. (2)). Figure 4 shows the decision flow around it.

The constant $k = 5$ is a deliberate compromise and appears in the consent text before any query is issued. It must be large enough to skip a release that is itself advisory-bearing—a real case, since a fix is sometimes published in the same line as a regression—and small enough that the developer can be told exactly how many advisory queries their approval authorises. A larger window would improve the chance of finding a clean target at the cost of consent precision; a window of one would make consent precise but would fail on any vulnerable

Algorithm 1 Target selection for one eligible dependency

Require: dependency d with $class(d) = \text{exact}$; registry $\mathcal{R}$; oracle $\mathcal{A}$; budget k

Ensure: target version $\tau(d)$ or $\perp$

```

1:  $V \leftarrow \{v \in \mathcal{R}(\text{name}(d)) : v > \text{resolved}(d) \wedge \text{stable}(v)\}$ 
2:  $C \leftarrow$  first  $k$  elements of  $V$  sorted descending
3: for  $v \in C$  do
4:    $a \leftarrow \mathcal{A}(\text{name}(d), v)$ 
5:   if  $a = \perp$  then
6:     record status unavailable for candidate  $v$ ; continue {fail closed}
7:   else if  $a = \emptyset$  then
8:
9:     return  $v$  {newest provably clean release}
10:  end if
11: end for
12:
13: return  $\perp$  {no admissible target; report, do not guess}
```

“fixed” release.

Two properties are worth stating explicitly because they are not provided. First, selection is greedy over the newest-first order rather than an optimisation over the whole plan; there is no solver and no cost model, in contrast to Remedius [3]. Second, PatchPilot makes no compatibility claim about the selected target: it guarantees the target carries no known advisories, and it states in the approval dialog, in the same breath as the change list, that compatibility of target versions is not verified. The compatibility question is deferred wholesale to the adjudication gate of Section V-F.

E. Diff-gated application and snapshot rollback

DR7–DR9. A plan materialises into an edit set E mapping each affected manifest path to a pair of byte strings. Nothing in E is written until two events have occurred in order.

Preview. Every element of E is rendered as a unified diff against the current file content and presented in full. The developer sees bytes, not a description of bytes. A single approval is then requested for the whole set; partial approval is not offered, because a plan applied in pieces produces intermediate states that have never been validated and that the rollback record would describe ambiguously.

Snapshot. Before the first write, SnapshotStore.beginStep copies the current content of every file in E into memory and records its SHA-256 digest as beforeHash. After a successful write it records afterHash from the bytes actually written. Algorithm 2 gives the protocol; Figure 3 gives the interaction.

The rollback comparison is the part that carries the design weight. Restoring unconditionally would destroy any edit the developer made between the write and the rollback—a tool that clobbers the user’s own work in order

Algorithm 2 Apply, validate and roll back

Require: edit set E ; command templates $\mathcal{C}$; snapshot store S

Ensure: accepted state, or restored workspace, or explicit conflict report

```

1: render  $\text{diffs}(E)$ ; present to developer
2: if approval not granted then
3:
4:   return aborted, workspace untouched
5: end if
6: for  $f \in \text{files}(E)$  do
7:    $S.\text{beginStep}(f)$ : store  $\text{before}(f)$ ,  $\text{beforeHash}(f) \leftarrow H(\text{before}(f))$ 
8: end for
9: write all elements of  $E$ ; record  $\text{afterHash}(f) \leftarrow H(\text{bytes written})$ 
10:  $r \leftarrow$  execute  $\mathcal{C}$  (Algorithm 1 unrelated; Section V-F)
11: if  $r = \text{pass}$  then
12:
13:   return accepted
14: end if
15: offer bounded repair channel (Algorithm 3)
16: on rollback request:
17: for  $f \in S.\text{files}()$  do
18:   if  $H(\text{current}(f)) = \text{afterHash}(f)$  then
19:     restore  $\text{before}(f)$ ; verify  $H = \text{beforeHash}(f)$ 
20:   else
21:     report  $f$  as conflict; leave untouched
22:   end if
23: end for
```

to undo its own is worse than no tool, and it would violate the trust the diff preview is trying to build. PatchPilot therefore treats a mismatch between the current digest and afterHash as a conflict: the file is reported, left alone, and the developer decides. Verification of the restore (re-hashing after writing the snapshot back) makes rollback itself a checked operation rather than an assumption.

F. Shell-free validation

DR10. Adjudication runs the project’s own commands, because only those encode the project’s intent. But running commands is also the single most dangerous capability in the tool: a command assembled by string concatenation from user settings is a command-injection vector in a program that holds file-write access to the workspace. PatchPilot closes this surface by removing the string. Validation commands are drawn from a closed table of seven templates (Table V); there is no user-typed command line anywhere in the executor, and every spawn is issued with shell:false so the operating system never interprets the argument list.

Three details in the table are security decisions rather than conveniences. `-ignore-scripts` on the lockfile refresh keeps the one command that genuinely resolves from

TABLE V: The closed validation command template table. Arguments are fixed; no component assembles a command string, and all processes are spawned with shell:false (DR10).

Template id	Invocation
npm-lock-refresh	npm install -package-lock-only -ignore-scripts -no-audit -no-fund
npm-test	npm test
npm-build	npm run build
maven-resolve	mvn -B -q dependency:resolve
maven-test	mvn -B -q test
pypi-pip-dry-run	python -m pip install -dry-run -quiet -r <file>
pypi-pytest	python -m pytest -q -p no:cacheprovider

becoming arbitrary code execution, updating the lockfile’s record of the new version without running a single lifecycle hook. -p no:cacheprovider stops pytest creating a cache directory inside the project, so a read-only validation leaves no artifact behind. Python is invoked as python -m pip and python -m pytest rather than as bare pip or pytest so that the interpreter used is the one PatchPilot itself resolved, not whichever executable happens to appear first on PATH.

Process discipline is otherwise uniform: output is captured with a byte cap, each command runs under a timeout, and only the exit status is used for adjudication. PatchPilot deliberately does not parse test output to decide why a run failed; that inference is a heuristic over text that an adversary may partially control, and its failure mode is a wrong “pass”.

On Windows the only shell-adjacent logic in the code-base resolves the .cmd shims for npm and mvn; the argument vectors passed to those shims are still taken verbatim from the template table.

G. Bounded model-assisted repair

DR11–DR13. When adjudication fails, PatchPilot offers—never performs—a model-assisted repair. The channel is governed by four constraints that are enforced by the code that consumes the model’s reply, not by the prompt that produces it. A prompt is a request; a parser is a boundary.

a) Never-editable manifests: The first constraint is a regular expression over paths that no model edit may ever target:

```
const NEVER_EDITABLE =
  /(^[\\\/])(package\.json|package-lock\.json|
  pom\.xml|pyproject\.toml|
  requirements.*\.txt|...)$/i;
```

The rationale is supply-chain shaped. A model permitted to edit package.json is a model permitted to revert a vulnerability fix, to pin a downgraded version, or to add a dependency nobody approved—each of which converts a remediation tool into a delivery mechanism for the very

TABLE VI: Hard bounds on the model context, applied before consent is requested (DR13).

Bound	Value	Constant
Source files per repair attempt	8	MAX_CANDIDATE_FILES
Bytes per file	60,000	MAX_CANDIDATE_BYTES
Total bytes per attempt	120,000	MAX_TOTAL_BYTES

problem it exists to fix. This is the single most important boundary in the design and the reason PatchPilot’s repair channel is narrower than a general coding agent’s.

b) Path allowlist: The second constraint is that the set of paths offered in the request is the only set the reply may address. parseRepairReply rejects any edit whose target path was not supplied, so the model cannot invent a file. File selection itself is a relevance heuristic run before consent: candidate files are first those whose filename appears in the failing command’s output, and failing that those whose content mentions one of the upgraded dependency names. An unrelated file that merely exists in the project is neither sent nor editable.

c) Exact-once edits: The third constraint is semantic. Each proposed edit is a search/replace pair, and applyRepairEdits requires the search text to occur exactly once in the target file. Zero occurrences means the model hallucinated context; more than one means the edit is ambiguous and its effect depends on which match is chosen. Both are rejected rather than disambiguated by a rule. The constraint rules out sweeping rewrites at the same stroke: a model cannot express “replace everything” because such an edit cannot match exactly once.

d) Budgets and consent: The fourth constraint bounds what may be transmitted (Table VI). Selection, allowlisting and truncation all happen before the consent dialog is shown, so the dialog’s file list is accurate rather than approximate. The dialog then names the endpoint, the receiving host, the model identifier, the exact files to be sent and the active exclusion patterns, and the request proceeds only on explicit approval—for each send, not once per session.

The request prompt states explicitly that the command output and file contents it carries are data, not instructions. This is the standard mitigation for indirect prompt injection [21]: build failures and dependency metadata are attacker-influenceable text, and a repair channel that embeds them into a prompt without saying so would hand an adversary a write path into the model’s context. The mitigation is imperfect and is treated as such in Section VI; its value is that it removes the naive case while the parser-side constraints remove the catastrophic ones.

Algorithm 3 assembles the channel. Note the ordering: selection and truncation precede consent; consent precedes transmission; parsing and boundary checks precede any

Algorithm 3 Bounded model-assisted repair

Require: failing output O ; candidate files F ; exclusions X ; budgets k_f, b_f, b_t

Ensure: applied and re-validated repair, or no-repair

- 1: $S_1 \leftarrow \{f \in F : \text{filename}(f) \in O\}$
- 2: $S_2 \leftarrow \{f \in F : f \text{ mentions an upgraded dependency name}\}$
- 3: $S \leftarrow$ first k_f files of $S_1 \cup S_2$, minus X , with per-file and total byte caps applied
- 4: if $S = \emptyset$ then
- 5: return no-repair
- 6: return no-repair
- 7: end if
- 8: show consent(endpoint, host, model, S , X)
- 9: if consent denied then
- 10: return no-repair
- 11: return no-repair
- 12: end if
- 13: $\text{reply} \leftarrow \text{model.complete}(\text{prompt}(O, S))$, prompt declares O and file bodies to be data
- 14: for $e \in \text{parse}(\text{reply})$ do
- 15: if $\text{path}(e) \notin S$ or $\text{path}(e)$ matches never-editable then
- 16: reject e
- 17: else if occurrences of $\text{search}(e)$ in target $\neq 1$ then
- 18: reject e
- 19: end if
- 20: end for
- 21: if no admissible edits remain then
- 22: return no-repair
- 23: return no-repair
- 24: end if
- 25: render diffs; if not approved return no-repair
- 26: snapshot, apply, re-validate (Algorithm 2); return result

diff; and the diff precedes any write. No reordering is permitted because each step's output is the next step's consent object.

H. Policy layer

DR14. Three policies apply across all layers. Workspace Trust: analysis refuses to run in a folder the developer has not marked trusted [24], so merely opening a cloned repository cannot trigger advisory traffic or parsing of attacker-authored manifests under a trusted identity. Redaction: the logger is a redacting logger; anything matching a secret-shaped pattern is masked before it reaches the output channel, and the API key is held only in VS Code's SecretStorage, never in settings or in memory longer than a request [25]. Exclusions: developer-configured path exclusions are applied to model context selection and are displayed in the consent dialog, so what is withheld is visible rather than implicit.

VI. Threat Model and Security Analysis

A. Adversary and assets

We consider four adversaries. A1: the compromised or malicious dependency, whose metadata, lifecycle scripts, error messages and release notes are attacker-influenceable text and code. A2: the compromised or misbehaving model endpoint, which may return edits that serve an adversary rather than the developer. A3: the untrusted workspace, a cloned or downloaded project whose manifests and scripts the developer has not reviewed. A4: network-level interference against registry, advisory or inference traffic.

The assets are: the integrity of the developer's manifests and lockfiles; the integrity of the developer's source, including work performed while PatchPilot is running; the confidentiality of source code and secrets; and the correctness of the security claims the tool displays—a false all-clear is itself a compromise of an asset.

B. Mechanisms and residual exposure

Table VII maps each concrete threat to the mechanism that addresses it and, importantly, to the exposure that remains. We report residual exposure because a threat model that lists only mitigations is a marketing document.

C. Invariants

The mechanisms above are easiest to audit as seven invariants that the code is required to maintain on every execution path. Section VII states how each is verified.

- I1 No byte is written to the workspace without a prior rendered diff and a prior approval of exactly those bytes.
- I2 No byte is written without a prior snapshot recording the previous content and its SHA-256 digest.
- I3 No version string enters a manifest except as the output of Eq. (2) on an exact pin.
- I4 No model output reaches the filesystem except through the allowlist, the manifest boundary and exact-once checks, and then only as an approved diff.
- I5 No process is spawned whose argv is not drawn from the template table.
- I6 No network request carries file content, workspace paths or credentials, except a consented model send whose content was enumerated in the consent dialog.
- I7 Rollback restores byte-identical prior content or reports a conflict; it never silently overwrites a file whose current digest differs from afterHash.

I3 is the invariant that distinguishes PatchPilot from agent-based upgraders. It is enforced not by a check at write time but by the type structure of the planner: the only constructor for a manifest edit takes a target produced by the selection function, and the selection function is the only consumer of registry version lists. There is no second path by which a version could reach an edit.

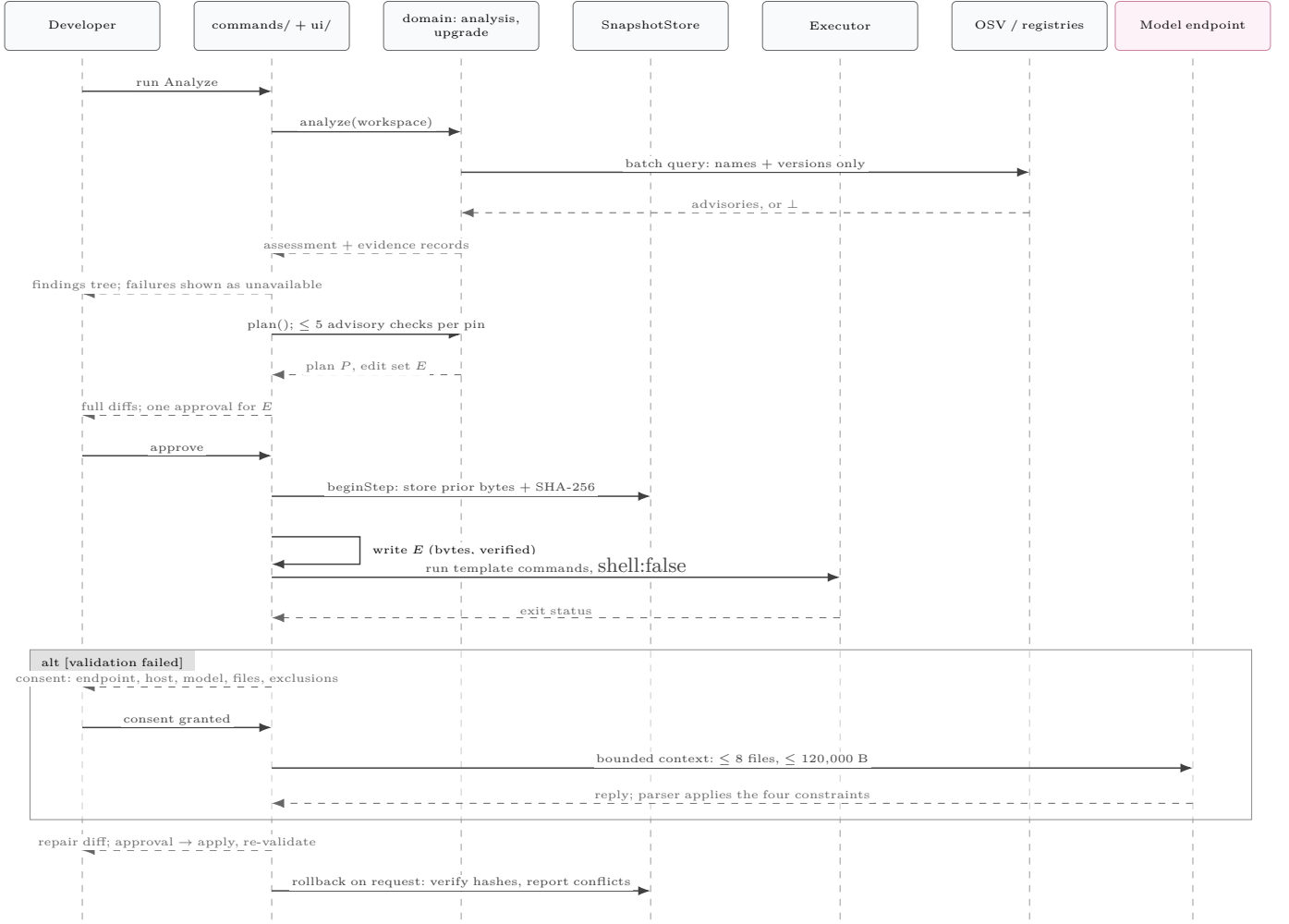


Fig. 3: Interaction sequence for one remediation cycle. The developer is the only actor able to cross a gate; the model is contacted at most once per failed validation and only after the consent message has enumerated exactly what will be sent.

VII. Evaluation

A. Evaluation approach

PatchPilot is a design-science artifact: its claims are architectural, and the appropriate first evaluation is whether the artifact satisfies the requirements it was built to satisfy, and whether the properties it asserts actually hold on every execution path. We therefore evaluate by construction-based verification: each design requirement of Table II and each invariant of Section VI is traced to the mechanism that realises it and to the means by which that mechanism is exercised.

We state plainly what this evaluation does not contain. It contains no repair success rate, no precision figure and no benchmark score, because producing one requires running the tool over a corpus such as DepBench [9] or BUMP [19], which is future work (Section VIII). Reporting a number we did not measure would be the same category of error as reporting “no known vulnerabilities” for a failed advisory lookup: a confident answer standing in for a

missing one. The recent literature makes the honesty of this choice easy to justify—Fruntke and Krinke report at most 23% test-suite success [5] and DepBench’s strongest configuration solves 51.2% of tasks [9]—so any success rate we might quote for a first artifact would be low, and the interesting question is not the rate but whether the failure is survivable. That is what this design optimises, and what this evaluation examines.

B. Artifact and test strategy

The artifact is a single TypeScript extension targeting VS Code 1.105 or later and Node 20 or later, compiled under strict TypeScript settings and linted with type-checked ESLint rules. It depends on two parsing libraries (fast-xml-parser for XML, smol-toml for TOML); everything else, including the HTTP helper, the SSE parser and the diff renderer, is implemented in-repository. There is no backend and no cloud component.

Verification runs at three levels.

TABLE VII: Threat model. Residual exposure is stated for every row; rows T3 and T4 are the ones the design treats as irreducible and therefore compensates for with a human gate.

ID	Threat / vector	Mechanism	Residual exposure
T1	Lifecycle scripts execute during analysis (preinstall, plugin goals, build backends)	Own parsers only (DR1); lockfile refresh uses <code>-ignore-scripts</code>	Validation intentionally runs the project’s tests; developer-initiated
T2	Model downgrades a version, reverts a fix, or adds a dependency	never-editable manifest regex; model never selects versions (DR11)	Malicious source edits remain possible; gated by diff approval
T3	Indirect prompt injection through build output or dependency metadata	Prompt declares output and file bodies to be data [21]; exact-once edits; allowlist	A steered model may still propose plausible bad source edits; approval is the last line
T4	Source exfiltration to a hostile or compromised endpoint	Per-send consent naming host, model, files, exclusions; byte budgets (DR13)	Once consented, content is the provider’s; exclusions are the only pre-filter
T5	Secret leakage via logs or model context	Redacting logger; VS Code SecretStorage; path exclusions shown in consent	A secret inside a consented file body travels with it unless excluded
T6	False all-clear caused by advisory outage or malformed response	Three-valued advisory result; unavailable status (DR5)	Advisory freshness (disclosure lag) is not detected
T7	Rollback destroys the developer’s subsequent edits	afterHash conflict detection; file left untouched (DR9)	Conflicts require manual resolution by the developer
T8	Untrusted workspace exploits the extension on open	Workspace Trust gate; no analysis before trust (DR14)	A trusted-but-malicious folder still runs its tests during validation
T9	Command injection through settings or plan data	Closed template table; shell:false; validated config limits (DR10)	The developer’s own test scripts remain their own risk
T10	Compromise of PatchPilot’s own dependencies	Small pinned dependency set; lockfile; provenance notices	Standard supply-chain exposure of two parser libraries
T11	Spoofed or intercepted registry / advisory responses	TLS endpoints; name+version-only queries; provenance timestamps	A compromised registry upstream of TLS is out of scope
T12	Resource exhaustion from a huge workspace or slow endpoints	Bounded HTTP helper; output caps; timeouts; file and byte budgets	Very large manifests may still be slow to parse

Pure unit tests. 252 unit tests exercise the domain and view-model layers with neither VS Code nor network access. This is possible because of the model/view split and the explicit dependency objects described in Section IV: parsers, assessment rules, target selection, edit construction, snapshot logic, repair parsing and the consent text builder are all pure functions and are tested as such.

Extension-host smoke tests. A second suite runs inside a real extension host and checks activation, command registration and view wiring—the parts that cannot be faked.

Headless end-to-end runs. A runner script drives the complete workflow (analyze → plan → apply → validate → repair → rollback) against four bundled sample projects without a user interface, so the sequence of Figure 3 is executed, not merely described.

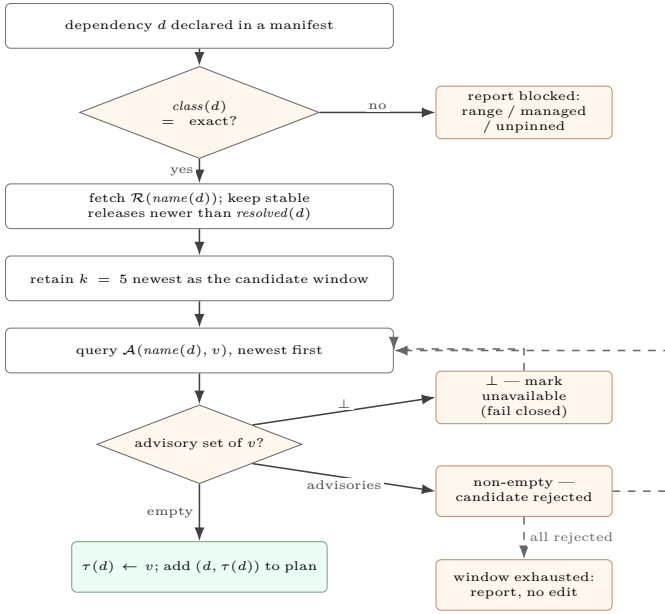
Table VIII lists the sample projects. Each is constructed to isolate one behaviour of the pipeline, which is what makes them usable as acceptance criteria: a regression in advisory handling breaks the first, in the repair channel the second, in Python target selection the third, and in Maven parsing the fourth.

TABLE VIII: Bundled sample projects and the pipeline behaviour each isolates.

Sample	Seed condition	Isolates
npm-vuln	lodash 4.17.15, exact pin with a published advisory	Detection, planning, diff-gated apply, validation on the npm path
npm-repair	deliberate source bug that fails validation	The consent-gated repair loop and re-validation
python-vuln	requests==2.19.0	PEP-aware parsing, == pin handling, PyPI target selection, dry-run validation
maven-vuln	a vulnerable Maven coordinate	pom.xml parsing, managed-version blocking, Maven validation

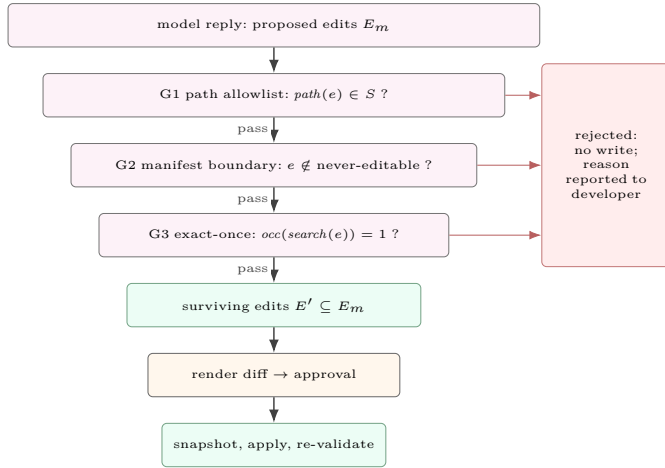
C. Requirement verification

Table IX traces all fourteen requirements to their mechanism and to how the mechanism is exercised. Three



Every non-green exit yields a finding with a reason; none is ever rendered as clean (DR5).

Fig. 4: Eligibility and candidate selection. Both exits marked “report” produce a finding with a reason and no edit; neither is ever rendered as clean.



Budgets (Table VI) and the consent gate apply before the reply exists; G1–G3 constrain what a reply may do. Prompts ask; parsers refuse.

Fig. 5: The repair guard funnel. Each stage is a parser-side rejection; the union of their rejections is what makes the channel bounded. A reply that survives all stages still requires a rendered diff and an approval before any write.

notes on reading it.

First, the “verification kind” column distinguishes runtime enforcement (a check that refuses an operation at execution time), type-level enforcement (the invalid state is unrepresentable or unreachable), and process enforcement (a human gate). A requirement discharged at type level cannot be violated by a future code change that forgets a

check; that is why DR3, DR5 and DR6 are placed there.

Second, “pure-testable” marks requirements whose mechanism lives entirely in the layers the 252 unit tests can reach without VS Code or network. Requirements whose mechanism is intrinsically a UI or process event (DR7, DR13, DR14) are instead covered by smoke tests, the headless runner, or by inspection of the consent and diff text that the pure consent-text builder produces.

Third, we deliberately do not report per-requirement test counts. The suite is organised by module, and retrofitting a requirement-to-test mapping after the fact would imply a precision of coverage that we have not established. The traceability claim we make is the weaker and true one: every requirement has a named mechanism, and every mechanism is reachable from at least one of the three verification levels.

D. Invariant verification

The seven invariants of Section VI are the audit form of the same mechanisms. I1 and I2 hold because the write path is a single function that takes an approved edit set and a populated snapshot store as arguments; calling it otherwise is a type error. I3 holds because manifest edits are constructed only from planner output. I4 holds because the repair parser is the sole deserialiser of model replies and it rejects before returning. I5 holds because the executor exposes no free-form spawn. I6 holds because registry and OSV clients accept only name/version pairs, and the model client accepts only a context object built by the budgeting selector. I7 holds because rollback compares digests before writing and records a conflict instead of overwriting.

Each of I1–I7 is reachable from the pure unit suite except I1’s approval antecedent, which is a UI event covered by smoke tests and by the headless runner’s scripted approvals.

E. Behaviour on the sample projects

The headless runner exercises the four samples end to end. The expected behaviour is deterministic in structure though not in content, because registry and advisory data evolve: for npm-vuln the plan must contain exactly one edit—the lodash pin in package.json plus its lockfile entries—with a target drawn from the five-version window and carrying no advisories at query time; for python-vuln the target must satisfy the same rule over PyPI releases; for maven-vuln the edit is confined to the coordinate in pom.xml unless the version is managed, in which case the run must report blocked and write nothing; for npm-repair validation must fail, the consent dialog must enumerate the selected files, and the applied repair must not touch package.json.

These are structural acceptance criteria: they assert the shape of the tool’s behaviour (one edit, correct provenance, correct boundaries) rather than a success probability. They are the criteria a future benchmark evaluation should

TABLE IX: Design requirement verification matrix. Enforcement kinds: RT runtime check that refuses the operation; TL type- or structure-level, invalid state unreachable; PR process/human gate.

Req.	Mechanism	Kind	Pure-testable	Exercised by
DR1	Own JSON/XML/TOML parsers; no package-manager spawn	TL	yes	unit tests over fixture manifests
DR2	pinClass taxonomy in analysis/versions.ts	TL	yes	unit tests per class
DR3	Eligibility gate in planner; edits only constructible for exact pins	TL	yes	unit tests; headless runner
DR4	Evidence records carrying source, URL, timestamp	TL	yes	unit tests on assessment output
DR5	Three-valued advisory result; unavailable status	TL	yes	unit tests on $\perp$ responses
DR6	Eq. (2), Algorithm 1	TL	yes	unit tests over synthetic version lists
DR7	Diff preview then single approval over the edit set	PR	partly	consent/diff text builders; smoke tests
DR8	SnapshotStore.beginStep with SHA-256	RT	yes	unit tests on snapshot/restore
DR9	afterHash comparison; conflict report	RT	yes	unit tests on mutated post-write files
DR10	Template table; shell:false spawns	TL	yes	unit tests on argv construction
DR11	never-editable regex + allowlist in reply parser	RT	yes	unit tests on hostile replies
DR12	Exact-once occurrence requirement	RT	yes	unit tests on zero/multi-match edits
DR13	File and byte budgets applied before consent	RT	yes	unit tests on oversized contexts
DR14	Trust gate; redacting logger; SecretStorage	RT	partly	smoke tests; log inspection

preserve while adding the measurement we do not yet have.

F. What the design achieves, and what it does not

Table X separates achieved properties from properties we explicitly do not claim. The second block is as important as the first: several of the unclaimed properties are precisely the contributions of the systems surveyed in Section II, and adopting their results is the natural continuation of this work.

G. Threats to validity

Construct. Construction-based verification establishes that a mechanism exists and is exercised; it does not establish field effectiveness. A requirement can be correctly implemented and still fail to help developers.

Internal. Unit tests exercise pure logic against fixtures and injected fakes. Behaviour that depends on real VS Code APIs, real process spawning and real network latency is covered only by smoke tests and the headless runner, so defects can remain in those seams.

External. The four samples are small, single-ecosystem and constructed; they contain no monorepo layout, no workspace-level dependency interaction, no lockfile in conflict with its manifest, and no project whose tests are flaky. Findings about behaviour on such workspaces do not follow from this evaluation. Ecosystem coverage excludes the long tail that DepBench shows to matter, notably Cargo and Go.

Conclusion. No statistical inference is drawn; there are no baselines and no repeated trials. Comparisons with prior work are capability comparisons, not performance comparisons.

Temporal. Advisory and registry contents change continuously, so any run is a statement about the data available at its timestamp. This is inherent to the domain and is why the tool records query times; it also means the sample walkthroughs must be re-run, not re-read, to be trusted.

VIII. Discussion

A. Why the model may not choose versions

The design decision most likely to be questioned is also the one we hold most firmly: the model never selects a dependency version. Three arguments support it.

Auditability. A version chosen by a model cannot be reproduced from the inputs, because the inputs include model weights and sampling. A version chosen by Eq. (2) can be recomputed by anyone with the registry and advisory snapshots, and PatchPilot records the query timestamps that make such a recomputation meaningful. In a supply-chain incident, “the model proposed 4.17.21” is not an explanation; “4.17.21 was the newest release in the five-version window with an empty advisory set at 2026-10-13T09:14Z” is.

Empirical base rates. Singla et al. [8] measured, across 117,062 dependency changes, that agents select already-

TABLE X: Achieved and deliberately unclaimed properties.

Achieved
Multi-ecosystem detection (npm, Maven, PyPI) with per-finding provenance
Deterministic, reproducible version selection over exact pins
Fail-closed advisory reporting; outage cannot read as “clean”
Every write diff-gated, approved, snapshotted and hash-verified on restore
Conflict-aware rollback that never clobbers user edits
Shell-free validation from a closed template table
Model channel bounded by allowlist, manifest boundary, exact-once and budgets
Per-send consent enumerating endpoint, host, model, files and exclusions
Whole-workflow logic testable without an editor or a network
Not claimed
Compatibility verification of selected target versions
Reachability analysis to suppress unreachable-transitive findings
Patch porting where no fixed version exists
Repository-wide migration of an upgraded contract (DepBench’s dominant failure)
Measured repair success rate or benchmark score
Ecosystems beyond npm, Maven and PyPI (no Go, Cargo, NuGet, Gradle-Kotlin)
Editing of transitive or range-managed dependencies
SBOM export or pull-request-time screening
User-study evidence about trust, calibration or workflow fit

vulnerable versions more often than humans do and that those selections are more expensive to remediate. A tool that lets a model choose versions inherits that base rate at the exact point where the tool’s credibility lives.

Blast radius. A wrong source edit is bounded: it fails the tests, the diff is visible, the snapshot restores it. A wrong version edit is unbounded in a different way: it may be silently compatible, ship to production, and only express itself as an exploitable dependency. The asymmetry of consequences justifies the asymmetry of authority.

None of this argues that models are useless here. The evidence of [4]–[6] is that models add real value to the migration half of the problem, which is why PatchPilot gives them that half—with boundaries, budgets and a consent gate—rather than excluding them entirely.

B. Position on the autonomy spectrum

Dependency tooling can be ordered by how much it does without asking: scanners that only report; bots that open pull requests; upgraders that apply and then test; agents that apply, test and repair autonomously. PatchPilot occupies a deliberate intermediate position: it applies without asking only after asking once with the full diff in view, and it repairs without asking never. The cost is

friction—a full cycle with a failed validation asks for three decisions (plan approval, send consent, repair approval)—and the justification is that each of the three corresponds to a different asset: the integrity of the manifest, the confidentiality of the source, and the integrity of the source. Collapsing them into one consent would make at least one of the three unconsented in fact.

C. Honest failure as a feature

Two reporting choices deserve emphasis because they are cheap to implement and hard to find in practice. The unavailable status makes infrastructure failure visible instead of silent. The approval dialog’s statement that target compatibility is not verified makes the tool’s own limit visible at the moment of highest trust. Both follow from the same rule: a security tool’s output is a claim, and every claim should carry either its evidence or its caveat.

D. Limitations and future work

The limitations of Table X map directly onto a research agenda, much of which is supplied by the surveyed literature.

Empirical evaluation. The immediate next step is to run PatchPilot’s planner and repair channel over DepBench [9] and BUMP [19] under the protocol those benchmarks define, replacing structural acceptance criteria with measured solve rates. Because PatchPilot refuses to edit non-exact pins, an honest comparison must report the fraction of benchmark tasks its eligibility gate declines, which is itself an informative number.

Reachability. Song and Zhu’s vulnerability propagation graph [2] addresses the signal-to-noise problem this paper acknowledges but does not solve: a finding against an unreachable transitive API is currently reported with the same weight as a reachable one. Porting reachability into the evidence model would let DR4 carry a reachability attribute without weakening DR5.

Remediation beyond upgrading. Where no advisory-free version exists, PatchPilot reports and stops. Remedius [3] shows that patch porting and SMT-based planning can cover part of that space; integrating a solver-backed fallback, behind the same diff gate, is the natural extension of Eq. (2).

Migration knowledge. LADU’s use of migration documentation [4] and PLLM’s retrieval augmentation [7] both indicate that retrieved, versioned prose improves repair quality. PatchPilot’s repair prompt currently carries only failing output and file bodies; adding retrieved changelog and migration context—within the existing byte budgets and consent model—is a bounded change.

Scope. Ecosystem coverage, monorepo and workspace-level manifests, Gradle-Kotlin and lockfile-manifest conflict handling, SBOM export, and pull-request-time screening in the sense of [8] are all open. So is the human side: a user study of whether developers actually read the diffs and consent dialogs, and whether unavailable

changes their behaviour, would test the design’s central assumption that visible honesty is usable honesty.

IX. Conclusion

This paper asked a narrower question than “can a tool upgrade my dependencies”: can a tool propose a dependency change, prove exactly what it changed, let the project’s own tests judge the consequence, and hand the source back unchanged if that judgement fails—while keeping the component most likely to err out of the decision most likely to matter?

PatchPilot answers in the affirmative for the npm, Maven and PyPI ecosystems inside the developer’s editor. It does so by refusing a trade that most tools accept: it does not buy convenience with opacity. Version selection is computed and reproducible; every write is a diff first and a snapshot second; validation is the project’s own test suite, run without a shell; and the model is admitted to exactly one sub-task, through four parser-enforced boundaries, behind a consent dialog that names what it is about to receive. Where the tool cannot know—whether a target is compatible, whether an advisory lookup would have succeeded—it says so, in the interface, at the moment the developer is deciding whether to trust it.

The evaluation is deliberately modest and deliberately honest: fourteen requirements and seven invariants traced to mechanisms and exercised by 252 pure unit tests, extension-host smoke tests and a headless end-to-end runner over four constructed samples, with an explicit table of what the design does not claim. The measurements that would complete the picture—solve rates on DepBench and BUMP, reachability-aware precision, and evidence about how developers actually use the gates—are named as future work rather than approximated.

Dependency remediation is not going to become fully autonomous soon; the strongest published agent configuration completes barely half of real upgrade tasks, and agents measurably choose vulnerable versions more often than people do. Until that changes, the defensible engineering position is the one this paper implements: automate the part that can be audited, bound the part that cannot, and make every irreversible-seeming act reversible.

References

- [1] A. Decan, T. Mens, and P. Grosjean, “An empirical comparison of dependency network evolution in seven software packaging ecosystems,” *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.
- [2] X. Song and Z. Zhu, “Compatible remediation for vulnerabilities in the presence and absence of security patches,” *Computers, Materials & Continua*, vol. 86, no. 1, pp. 1–19, 2026, doi: 10.32604/cmc.2025.068930.
- [3] L. Zhang, H. Ye, and Y. Liu, “Agent-based automated remediation for vulnerabilities in Maven projects,” *Proceedings of the ACM on Programming Languages*, vol. 10, no. OOPSLA2, art. 363, pp. 1357–1383, 2026, doi: 10.1145/3839495.
- [4] V. Tawosi, S. Alamir, X. Liu, and M. Veloso, “LLM agents for automated dependency upgrades,” *arXiv:2510.03480 [cs.SE]*, 2025.
- [5] L. Fruntke and J. Krinke, “Automatically fixing dependency breaking changes,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, art. FSE096, pp. 2146–2168, 2025, doi: 10.1145/3729366.
- [6] F. Reyes, M. Mahmoud, F. Bono, S. Nadi, B. Baudry, and M. Monperrus, “Byam: Fixing breaking dependency updates with large language models,” *arXiv:2505.07522 [cs.SE]*, 2025.
- [7] A. Bartlett, C. Liem, and A. Panichella, “The last dependency crusade: Solving Python dependency conflicts with LLMs,” in *Proc. 40th IEEE/ACM Int. Conf. on Automated Software Engineering Workshops (ASEW)*, 2025, pp. 66–73.
- [8] T. Singla, B. Çakar, P. C. Amusuo, and J. C. Davis, “Understanding security risks of AI agents’ dependency updates,” *arXiv:2601.00205 [cs.SE]*, 2026.
- [9] Z. Luo, R. He, P. Gao, Y. Kang, Z. Lin, M. Ma, Q. Lin, S. Rajmohan, and Y. Tian, “Update from hell: Can coding agents survive hidden breakage in dependency upgrades?” *arXiv:2608.30300 [cs.SE]*, 2026.
- [10] OWASP Foundation, “OWASP Dependency-Check,” 2024. [Online]. Available: <https://owasp.org/www-project-dependency-check/>
- [11] GitHub, “Dependabot,” 2024. [Online]. Available: <https://github.com/dependabot>
- [12] Mend Software, “Renovate,” 2024. [Online]. Available: <https://github.com/renovatebot/renovate>
- [13] Snyk Ltd., “Snyk open source vulnerability scanner,” 2024. [Online]. Available: <https://snyk.io/>
- [14] Eclipse Foundation, “Eclipse Steady,” 2023. [Online]. Available: <https://projects.eclipse.org/proposals/eclipse-steady>
- [15] Open Source Security Foundation, “OSV schema specification,” 2024. [Online]. Available: <https://ossf.github.io/osv-schema/>
- [16] Google LLC, “OSV: Open source vulnerabilities,” 2024. [Online]. Available: <https://osv.dev/>
- [17] GitHub, “GitHub Advisory Database,” 2024. [Online]. Available: <https://github.com/advisories>
- [18] X. Wang et al., “OpenHands: An open platform for AI software developers as generalist agents,” in *Proc. Int. Conf. on Learning Representations (ICLR)*, 2025.
- [19] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, and M. Monperrus, “BUMP: A benchmark of reproducible breaking dependency updates,” in *Proc. IEEE Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, 2024, doi: 10.1109/SANER60148.2024.00024.
- [20] J. Hejderup and G. Gousios, “Can we trust tests to automate dependency updates? A case study of Java projects,” *Journal of Systems and Software*, vol. 183, art. 111097, 2021.
- [21] S. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, “Not what you’ve signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection,” in *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023, pp. 79–90.
- [22] Python Software Foundation, “PEP 621: Storing project metadata in pyproject.toml,” 2020. [Online]. Available: <https://peps.python.org/pep-0621/>
- [23] T. Preston-Werner, “Semantic versioning 2.0.0,” 2013. [Online]. Available: <https://semver.org/>
- [24] Microsoft, “Visual Studio Code documentation: Workspace trust,” 2024. [Online]. Available: <https://code.visualstudio.com/docs/editor/workspace-trust>
- [25] Microsoft, “Visual Studio Code API: SecretStorage,” 2024. [Online]. Available: <https://code.visualstudio.com/api/references/vscode-api>
- [26] Natural Intelligence, “fast-xml-parser,” 2024. [Online]. Available: <https://github.com/NaturalIntelligence/fast-xml-parser>
- [27] smol-rs, “smol-toml,” 2024. [Online]. Available: <https://github.com/smol-rs/smol-toml>