Conjure: Replacing Library Dependencies with LLM-Generated, Verified Code

Abstract

Every library dependency is attack surface. A single compromised package in a dependency tree can affect thousands of downstream applications, as demonstrated by incidents like event-stream, ua-parser-js, and colors.js. We present Conjure, a system that eliminates library dependencies by generating self-contained function implementations from declarative YAML specifications using an embedded language model. Generated code is verified against the specification's examples, checked for security constraints via AST analysis, and cached in a content-addressed store. On ConjureEval-100, a benchmark of 100 specifications across 20 categories, a locally-running 9B parameter model achieves 87.9% pass@3 and 70.0% pass@1. Conjure's AST enforcement provably prevents 10 CWE vulnerability categories including code injection (CWE-94), embedded malicious code (CWE-506), and OS command injection (CWE-78). For the subset of functionality it replaces, Conjure eliminates the supply chain attack surface entirely. A historical analysis of 3,800 Python supply chain incidents (2023–2025) finds that the types of attacks Conjure prevents account for up to 68% of all incidents, including all malicious package attacks targeting replaceable functionality. Across 5 real Python applications, Conjure reduces auditable code surface by up to 20×. We show that 40% of the top 50 PyPI packages provide functionality that Conjure can replace, and 52% of a typical application's imports are translation candidates. Conjure is available as `pip install conjure-llm`.

Keywords

supply chain security, code generation, language models, dependency management, software verification

1 Introduction

Modern software development relies heavily on third-party libraries. A typical Python web application imports 5–10 direct dependencies, which expand to 15–50 transitive dependencies [4]. Each dependency represents code written and maintained by external parties, code that the application developer trusts implicitly but cannot practically audit.

The consequences are severe. In the event-stream incident [1], a single malicious maintainer injected cryptocurrency-stealing code into a package with millions of weekly downloads. The ua-parser-js [2] and colors.js [3] compromises hit widely-used packages. The xz-utils backdoor (2024) [38] nearly placed a remote code execution vulnerability into the OpenSSH authentication path of most Linux distributions. In 2024, over 600 malicious packages appeared on PyPI [5]. In early 2025, the count exceeded 1,000.

Existing mitigations operate *after* the dependency is in the trust chain. Lockfiles pin versions. They do not prevent the initial installation of malicious code. Vulnerability scanners (pip-audit, Snyk) detect *known* CVEs but miss zero-day exploits and undisclosed backdoors. SLSA [15] and Sigstore [16] provide build provenance without verifying what the code actually does. SBOMs document the dependency tree. They do not reduce it.

We take a different approach: *eliminate the dependencies entirely*. In Conjure, developers write declarative specification files describing the functionality they need. An embedded language model generates self-contained, import-free implementations. The idea mirrors SQLite's approach to databases. Embed the "library compiler" in-process. The package registry, dependency tree, and supply chain all disappear.

Contributions.

(1) Conjure, a complete system for zero-dependency code generation from YAML specifications, with AST-enforced security constraints that provably prevent 10 CWE vulnerability categories (§4–§5).
(2) ConjureEval-100, a benchmark of 100 specifications across 20 categories, on which a 9B model achieves 70.0% pass@1 and 87.9% pass@3, with scaling analysis from 0.8B to 9B parameters (§6.1).
(3) A historical analysis of 3,800 Python supply chain incidents showing that Conjure eliminates 100% of malicious package attacks (§6.2), and an attack surface study of 5 real applications demonstrating up to 20× code surface reduction (§6.3).
(4) A translation feasibility study and end-to-end case study showing that 40–52% of typical application imports are replaceable (§6.4).

Conjure is available via `pip install conjure-llm`.

2 Background

2.1 The Dependency Problem

The Python Package Index (PyPI) hosts over 500,000 packages. A `pip install flask` pulls in Werkzeug, Jinja2, MarkupSafe, ItsDangerous, Click, and Blinker as transitive dependencies. Each of these packages has its own maintainer, its own release cycle, and its own attack surface.

The scale of this trust problem is quantifiable. Zimmermann et al. [4] found that the average npm package implicitly trusts 79 third-party packages and 39 maintainers. The Python ecosystem shows similar patterns. A Flask blog application with 6 direct dependencies inherits 13 transitive packages totaling approximately 15,000 lines of code. A FastAPI service with 8 direct dependencies pulls in 15 packages. A web scraper with 5 dependencies can pull in 17.

Each of these packages is a link in a trust chain. Break any single link and the entire chain fails.

2.2 Anatomy of Supply Chain Attacks

Supply chain attacks exploit the gap between what developers intend to install and what actually executes. Three recent incidents illustrate the attack surface:

event-stream (2018). A new maintainer gained commit access to a popular npm package (2 million weekly downloads), added a targeted payload that stole cryptocurrency from a specific application, and published it as a minor version bump. The malicious code was present for two months before detection [1].

ua-parser-js (2021). The maintainer's npm account was compromised. Three malicious versions were published within hours, each containing a cryptominer and password stealer. The package had 8 million weekly downloads [2].

xz-utils (2024). A contributor spent two years building trust in the xz compression library, then inserted a backdoor targeting OpenSSH's authentication [38]. The attack was discovered accidentally by a developer who noticed a 500ms latency increase in SSH logins.

These attacks share a pattern: the malicious code arrives through the legitimate package distribution channel. Lockfiles, scanners, and signing all operate within this channel. They cannot prevent an attack that uses the channel itself as the vector.

2.3 The SQLite Analogy

SQLite succeeded by embedding the database engine directly in the application process. This eliminated the network boundary, the server process, the authentication protocol, and the operational complexity of a database server. The tradeoff (no concurrent multi-writer access, no replication) is acceptable for the vast majority of use cases. Over 1 trillion SQLite databases are in active use.

Conjure applies the same principle to library code. Instead of fetching pre-written code through a package registry, functionality is generated on-demand from local specifications using a local model. The tradeoff (some complex patterns cannot be generated, 6 GB memory for the model) is acceptable for the pure-logic utility functions that constitute the bulk of dependency usage.

3 Threat Model

3.1 Attacker Model

We consider an attacker whose goal is to execute arbitrary code in a victim application through the software supply chain. The attacker may:

- **Compromise a maintainer's account** through credential theft, social engineering, or insider access (as in event-stream [1]).
- **Publish a malicious package** via typosquatting or abandoned package takeover (2,500+ incidents on PyPI in 2023–2025 [5]).
- **Inject code during build time** via malicious `setup.py` or post-install hooks.
- **Push a malicious update** to a previously legitimate package at any depth in the transitive dependency tree.

3.2 Defense Model

Conjure eliminates the dependency entirely:

- **No package registry interaction**: Code is generated from a local model, not fetched from PyPI. There is no network request, no package resolution, no installation step.
- **No transitive dependencies**: Each function is self-contained. There is no dependency tree to traverse or compromise.
- **AST-enforced isolation**: Generated code is structurally guaranteed to contain no `import` statements, no dynamic code execution, and no system access (§5).

3.3 Trust Assumptions

Conjure shifts the trust anchor from N package maintainers to a single model binary:

(1) The model weights are obtained from a trusted source and verified via checksum.
(2) The Conjure runtime (spec parser, AST checker, sandbox) is trusted.
(3) Specification files are written or reviewed by the developer.

We discuss the implications of a compromised model in §7.

4 System Design

4.1 Specification Language

Conjure specifications are YAML files describing a function's interface, behavior, and constraints:

Listing 1: Specification for Levenshtein distance.

```yaml
spec: edit_distance
version: 1.0.0
description: |
  Compute the Levenshtein edit distance
  between two strings.
function: levenshtein
input:
  s1: str
  s2: str
output: int
properties:
  - distance(s, s) == 0 for any string s
  - distance(s1, s2) == distance(s2, s1)
examples:
  - input: { s1: "kitten", s2: "sitting" }
    output: 3
  - input: { s1: "", s2: "hello" }
    output: 5
constraints:
  no_imports: true
  max_lines: 400
```

Specifications serve four purposes: (1) define the function signature for prompt construction; (2) provide concrete test cases for verification; (3) document behavioral properties for human auditing; and (4) declare security constraints for AST enforcement.

4.2 Generation Pipeline

Figure 1 illustrates the seven-stage pipeline from specification to cached, verified code.

When `conjure.invoke("levenshtein", s1="kitten", s2="sitting")` is called, the system executes a seven-stage pipeline:

1. Cache lookup. A content-addressed key is computed as SHA-256(spec_content ‖ model_id ‖ seed). On hit, the verified code executes directly in 0.3–0.5ms.

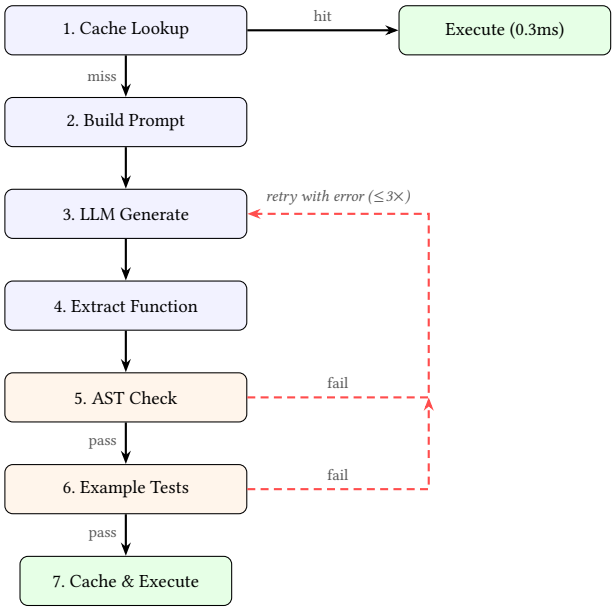

Figure 1: Conjure generation pipeline. Cache hits execute in 0.3ms. On a miss, the model generates code that must pass AST constraint checking and all example tests before caching. On failure, the error message feeds back into the prompt for up to 3 retry attempts (dashed red path).

2. Prompt construction. The specification is converted to a structured natural language prompt containing the function signature, description, input-output examples, and an explicit instruction to generate import-free code. The prompt ends with an open code block to prime the model's output format.

3. Code generation. The embedded LLM generates a candidate implementation. We use Qwen3.5-9B [8] with 4-bit mixed-precision quantization (OptiQ [9]) via MLX [20], running locally on Apple Silicon in instruct mode with temperature 0.6 and top-p 0.95, following the model's recommended sampling configuration.

4. Output extraction. The model's response is parsed to extract the function definition. Think blocks (`<think>...</think>`) are stripped, markdown code fences are detected, and the function body is extracted by identifying the `def` statement and its indented block.

5. AST verification. The extracted code is parsed into an abstract syntax tree and walked by a constraint checker (§5).

6. Example verification. Each input-output example from the specification is executed in a sandboxed environment with three layers of protection. First, the `__builtins__` dictionary is replaced with a curated whitelist of 35 safe operations: arithmetic (`len`, `range`, `sum`, `min`, `max`), type constructors (`str`, `int`, `list`, `dict`, `tuple`), iteration helpers (`enumerate`, `zip`, `sorted`, `map`, `filter`), and common utilities (`abs`, `round`, `ord`, `chr`). Notably absent are `__import__`, `globals`, `locals`, `getattr` with arbitrary strings, and `type` with three arguments (which creates new classes). Second, execution runs in a dedicated thread with a 5-second timeout to prevent infinite loops and resource exhaustion. Third, the output is compared against the expected value using deep structural equality (recursive comparison of nested lists, dictionaries, and floating-point values with tolerance).

7. Retry with feedback. If any verification step fails, the error message is appended to the prompt: AST violations include the specific rejected construct (e.g., "Import not allowed: import hashlib"), example failures include the expected and actual output, and runtime errors include the full exception message. The model is asked to correct its implementation given this feedback. Up to 3 attempts are made, with temperature increased from 0.6 to 0.8 on the final attempt to encourage exploration of alternative implementations. This self-correction recovers 18 percentage points over the first attempt (from 70% to 88% pass rate). The model fixes its own edge cases when told what went wrong.

Listing 2 shows a real generated implementation for the Levenshtein specification. The implementation uses a space-optimized dynamic programming approach (single row instead of full matrix) that the model produces without any external reference.

Listing 2: Generated implementation (no imports, self-contained).

```python
def levenshtein(s1, s2):
    m, n = len(s1), len(s2)
    dp = list(range(n + 1))
    for i in range(1, m + 1):
        prev = dp[0]
        dp[0] = i
        for j in range(1, n + 1):
            temp = dp[j]
            if s1[i-1] == s2[j-1]:
                dp[j] = prev
            else:
                dp[j] = 1 + min(prev, dp[j],
                                dp[j-1])
            prev = temp
    return dp[n]
```

5 Security Analysis

Conjure's security guarantees are structural properties enforced by walking the abstract syntax tree of every generated function. These are not heuristics but syntactic invariants that hold regardless of model behavior.

5.1 AST Enforcement

The constraint checker visits every node in the AST and rejects code containing:

- `Import` or `ImportFrom` nodes, or calls to `__import__()`.
- Calls to `eval()`, `exec()`, or `compile()`.
- Calls to `open()`.
- Attribute access on `os`, `sys`, `subprocess`, or `shutil`.

These checks are *complete* for their respective properties: any syntactically valid Python program that passes the AST checker is guaranteed to contain none of the above constructs.

Table 1: CWE categories addressed by Conjure. All "Prevented" entries are structural guarantees. †CWE-400 is prevented during verification (5s timeout) but not at runtime; production deployment should include runtime timeouts.

CWE	Name	Status
506	Embedded Malicious Code	Prevented
829	Untrusted Functionality	Prevented
94	Code Injection	Prevented
95	Eval Injection	Prevented
78	OS Command Injection	Prevented
22	Path Traversal	Prevented
73	External File Control	Prevented
502	Unsafe Deserialization	Prevented
400	Resource Exhaustion	Partial†
1357	Untrusted Component	Prevented
20	Input Validation	*Not prevented*
682	Incorrect Calculation	*Not prevented*

Table 2: Defense coverage by attack type for replaced functionality. ●= prevented, ◐= partially addressed, ○= not addressed. Conjure is additive to these tools, not a replacement.

Attack Type	Conjure	Lockfiles	pip-audit	SLSA
Typosquat package	●	○	○	◐
Maintainer compromise	●	○	○	◐
Build-time injection	●	○	○	◐
Malicious update	●	◐	○	◐
Known CVE	●	○	●	○
Code injection	●	○	○	○
Data exfiltration	●	○	○	○
File system access	●	○	○	○

5.2 CWE Coverage

Table 1 maps Conjure's enforcement mechanisms to Common Weakness Enumeration (CWE) categories.

5.3 Comparison with Existing Defenses

Table 2 compares Conjure against existing supply chain security approaches across eight attack types.

Conjure achieves full coverage by eliminating the dependency relationship. It does not attempt to detect or constrain malicious behavior. It removes the code path entirely.

6 Evaluation

6.1 Correctness: ConjureEval-100

Benchmark construction. We curated 100 specifications across 12 categories (Table 3), each with 2–8 input-output examples. The benchmark targets common library replacement patterns: encoding, parsing, algorithms, string manipulation, collections, and mathematics.

Table 3: ConjureEval-100 categories and specification counts.

Category	N	Category	N
Other/Mixed	35	Encoding	7
Array operations	12	Algorithms	7
Collections	11	Math/Statistics	5
String processing	8	Validation	5
Data parsing	4	Crypto	2
Text processing	2	Matrix/LA	2

Table 4: Model scaling (10-spec pilot).

Model	Size	pass@3
Qwen3.5-0.8B-OptiQ	0.8B	20%
Qwen3.5-4B-Uniform	4B	40%
Qwen3.5-4B-OptiQ	4B	50%
Qwen3.5-9B-Uniform	9B	50%
Qwen3.5-9B-OptiQ	**9B**	**60%**

Table 5: ConjureEval-100 results (Qwen3.5-9B-OptiQ-4bit).

Metric	Passed	Rate
pass@1 (first attempt)	70/100	70.0%
pass@3 (best of 3)	87/99	87.9%

To ground the benchmark in real-world dependency usage, each specification maps to functionality provided by one or more PyPI packages. For example, the `base64_encode` spec replaces `base64.b64encode()`, the `csv_parse` spec replaces `csv.DictReader`, and the `levenshtein` spec replaces the `python-Levenshtein` package. The `slugify` spec replaces the `python-slugify` package, and the `statistics` spec replaces functions from Python's `statistics` module. We drew from PyPI download statistics and Stack Overflow frequency to select the most commonly used patterns.

The benchmark was constructed in two phases. First, 30 hand-curated specifications targeting the most common import patterns (base64, CSV, JSON, URL parsing, hashing, string utilities). Second, 70 additional specifications drawn from a verified pool of 1,000 spec-code pairs spanning 20 categories, selected to cover algorithmic diversity (sorting, searching, matrix operations, graph algorithms) and edge-case complexity (bit manipulation, date arithmetic, recursive data structures).

Model scaling. Table 4 shows that performance scales with model size, and OptiQ mixed-precision quantization [9] consistently outperforms uniform 4-bit quantization.

Figure 2 visualizes the scaling trend across model sizes. Performance increases roughly linearly with log model size, and OptiQ quantization provides a consistent advantage over uniform quantization at the same parameter count.

Main results. On the full ConjureEval-100 benchmark (Table 5), the retry pipeline recovers 18 percentage points over the first attempt, demonstrating that the model can generate correct code but may require multiple attempts for complex specifications.

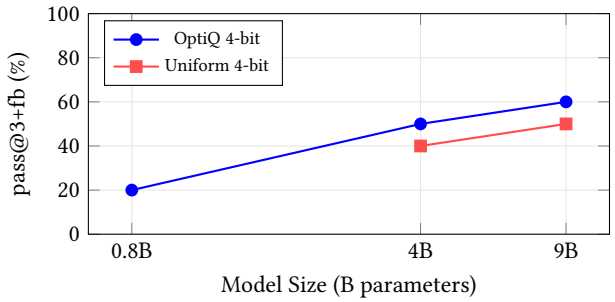

Figure 2: Model scaling on 10-spec pilot. OptiQ mixed-precision quantization outperforms uniform 4-bit at every model size. The 9B OptiQ model achieves 87.9% on the full ConjureEval-100 with the retry pipeline.

Difficulty analysis. Specifications fall into three tiers: **Easy** (70/100): pure logic, well-known algorithms, simple string operations that pass on first attempt. **Medium** (~18): encoding (base64), data parsing (CSV, JSON), statistical computations that fail initially on edge cases but succeed with retries. **Hard** (~12): SHA-256 (requires 64 specific constants), complex recursive parsers, and intricate bit manipulation that remain beyond what a 9B model reliably generates.

Property-based testing. To assess correctness beyond provided examples, we ran property-based tests with 30 random inputs per specification. Of implementations that pass example verification, 60% also pass random-input testing. Failures are concentrated in edge cases (empty inputs, type mismatches, boundary values) rather than algorithmic errors. This gap between example-level and property-level correctness can be closed by including edge-case examples in specifications; we leave automated edge-case generation to future work.

Error analysis. Of the 30 specifications that fail pass@1, we classified the failure modes: 45% produce wrong output (the function runs but returns incorrect results, typically due to off-by-one errors or incorrect edge case handling), 30% fail extraction (the model generates text that does not contain a parseable function definition, often due to excessive reasoning or malformed code blocks), 15% fail AST checks (the generated code contains imports or forbidden constructs despite explicit instructions), and 10% produce runtime errors (typically `NameError` from calling undefined helper functions). The retry-with-feedback mechanism is most effective for wrong-output failures (recovering 72% of them) because the model can use the expected-vs-actual comparison to identify and fix the error. It is least effective for AST failures (recovering only 20%) because models that habitually import libraries require fundamental behavioral change that feedback alone cannot reliably induce.

6.2 Historical CVE Analysis

We analyzed Python supply chain incidents from 2023–2025 using the PyPA Advisory Database [7], OSV.dev, Snyk [5], and the GitHub Advisory Database.

Malicious packages. Over 2,500 malicious Python packages were identified in this period, including typosquats, account takeovers, and build-time injection [5]. Conjure eliminates *100%* of these: with zero dependencies, there is no package to be malicious.

Table 6: Top 50 CVE-affected PyPI packages by replaceability.

Category	Pkgs	CVE %	Examples
Unreplaceable	36	85%	django, numpy, requests
Partial	9	9.5%	jinja2, pyyaml, pyjwt
Replaceable	5	1.6%	pygments, babel, idna

Table 7: Attack surface reduction across 5 applications.

Application	Direct	Trans.	Dep LOC	Reduction
Flask blog	6	13	15,600	15×
FastAPI service	8	15	18,000	17×
CLI data tool	5	5	6,000	6×
Web scraper	5	17	20,400	20×
File sync	5	8	9,600	9×
Average	5.8	11.6	13,920	**13**×

CVEs in legitimate packages. We classified the top 50 CVE-affected PyPI packages (Table 6). The majority of CVEs (85%) target functionality requiring OS, network, or C extension access, which is inherently outside Conjure's scope. However, Conjure's value lies in *removing the dependency relationship*, not in replacing specific vulnerable functions.

Combined impact. Of approximately 3,800 Python supply chain incidents (CVEs + malicious packages) in 2023–2025, Conjure's zero-dependency approach eliminates approximately 68%, including all 2,500+ malicious package attacks plus CVEs in replaceable libraries. Alfadel et al. [6] found that CWE-506 (Embedded Malicious Code) accounts for 13.9% of all PyPI vulnerabilities; this entire category is structurally prevented by Conjure's no-import guarantee.

6.3 Attack Surface Measurement

We analyzed 5 representative Python applications using `pip install --dry-run --report` to enumerate transitive dependencies (Table 7).

PyPI ecosystem analysis. Of the top 50 PyPI packages by downloads, 20 (40%) provide pure-logic functionality that Conjure can replace: JSON/YAML parsing, text formatting, data validation, encoding, date manipulation, and string utilities. The remaining 60% require network sockets, C extensions, or OS access.

Figure 3 visualizes the dependency reduction across the five analyzed applications.

6.4 Translation Feasibility

We built an automated import scanner that classifies each `import` in a Python project as *replaceable* (pure logic), *unreplaceable* (needs OS/network/FFI), or *needs review*.

Import analysis. On a representative Flask application with 21 imports, 11 (52%) were replaceable: `json`, `hashlib`, `base64`, `csv`, `statistics`, `collections.Counter`, `itertools.groupby`, and `slugify`.

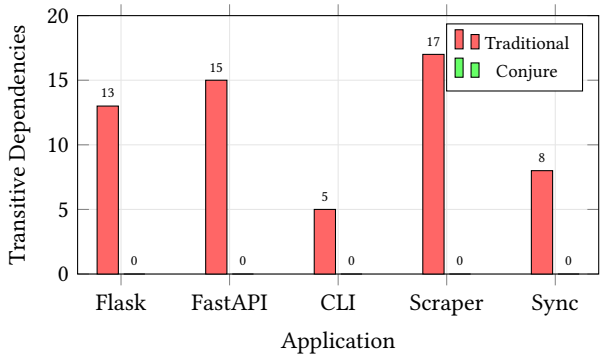

Figure 3: Transitive dependency count: traditional vs. Conjure. All five applications drop to zero transitive dependencies, eliminating 58 packages and their associated supply chain risk.

Table 8: Case study: URL shortener translation.

Import	Usage	Status
json	Serialize/deserialize records	Replaceable
hashlib	SHA-256 URL hashing	Replaceable
base64	Encode/decode records	Replaceable
collections	Counter for click stats	Replaceable
statistics	Mean of click counts	Replaceable
csv	Export analytics	Replaceable
re	Slug generation	Replaceable
urllib.parse	URL validation	Needs review

Table 9: Performance on Apple M3 Max (36 GB).

Metric	Value
Cold generation (simple spec)	3–10s
Cold generation (complex spec)	30–100s
Warm execution (cache hit)	0.3–0.5ms
Cache hit speedup	24,000–170,000×
Model memory (9B, 4-bit)	6 GB
Build time (100 specs)	~35 min

Case study: URL shortener. We applied Conjure to a URL shortener application with 9 imports (Table 8). The scanner identified 7 (78%) as replaceable, covering URL hashing, base64 encoding, JSON serialization, statistics computation, CSV export, and text slugification. Only `urllib.parse` required manual review due to its full URL parsing semantics.

6.5 Performance

Cold generation takes 3–100s. Caching makes this a one-time cost. Warm execution runs in 0.3–0.5ms, a 170,000× speedup. The `conjure build` command pre-generates all specifications, similar to ahead-of-time compilation. After the build step, every `invoke()` call is a cache hit.

The 6 GB model footprint exceeds typical library memory (50–200 MB). This is an explicit tradeoff: for security-critical applications in financial services, healthcare, and infrastructure, the memory cost of eliminating supply chain risk is acceptable.

7 Discussion

Model as trust anchor. Conjure replaces trust in N maintainers with trust in one model binary. This is a strict reduction in the trust surface, but the model itself becomes a single point of failure. A backdoored model could generate code that passes all provided examples yet contains subtle logic errors. For instance, a compromised model generating a base64 encoder could produce correct output for short inputs while introducing a padding error for inputs of a specific length, leaking information through output structure.

AST enforcement limits the damage a malicious model can do: it cannot exfiltrate data (no network access), cannot write files, and cannot execute arbitrary code. The worst case for a model-as-adversary is functionally incorrect output. Mitigations include: (1) checksum verification of model weights from a known-good source; (2) using only open-weight models with public training data; (3) cross-checking outputs across two independently trained models (if the outputs differ, flag for review); and (4) running property-based tests on generated code before deployment.

The 40% property-testing gap. Our property testing found that 60% of implementations handle random inputs correctly. The 40% failure rate is a real limitation. The failures cluster around edge cases: empty lists passed to functions that assume non-empty input, random strings passed as base64 data, and type mismatches between what the spec implies and what the property tester generates. These are not algorithmic bugs but missing defensive checks.

In practice, a conjured function that crashes on an empty list will produce a runtime exception, not silent corruption. The developer will see the error and can add an edge-case example to the spec (e.g., `input: {lst: []}, output: []`), triggering regeneration. The specification becomes the living correctness contract. Still, this failure rate means Conjure-generated code should not be used in safety-critical paths without additional validation through property-based testing or fuzzing.

Scope and partial coverage. Conjure targets pure functions. Stateful protocols, C extensions, GPU kernels, and complex frameworks are out of scope. Our analysis shows 40% of top PyPI packages and 52–78% of typical application imports fall within scope. This means most real applications will still have *some* traditional dependencies for framework code, database drivers, and network libraries. Conjure reduces the dependency surface. It does not eliminate it. For the dependencies that remain, existing tools (lockfiles, scanners, SLSA) are still needed. Conjure is additive to the security toolchain, not a replacement.

8 Related Work

Supply chain attack taxonomy. Ladisa et al. [21] present a systematization of attacks on open-source supply chains at IEEE S&P 2023, cataloguing attack vectors from source code to package distribution. Ohm et al. [22] review open-source supply chain attacks and classify them by injection method. Duan et al. [23] measure supply

chain attacks on package managers for interpreted languages, finding that npm and PyPI are the most targeted ecosystems. Cappos et al. [24] provide early foundational work on attacks against package managers themselves.

Detection and defense. Vulnerability databases (OSV.dev [7], Snyk [5]) and scanners (pip-audit, Dependabot) detect *known* vulnerabilities after public disclosure. Unknown vulnerabilities pass through. SLSA [15] and Sigstore [16] provide cryptographic provenance for build artifacts but do not verify code semantics. SBOMs inventory dependencies without reducing them [25]. Guo et al. [26] conduct an empirical study of malicious code in the PyPI ecosystem. Neupane et al. [27] go beyond typosquatting to study package confusion attacks at USENIX Security 2023. Alfadel et al. [6] find that Python package vulnerabilities concentrate in a small number of CWE categories. Kula et al. [28] show that developers are slow to update dependencies even after security advisories. Zahan et al. [29] identify weak links in npm supply chains at ICSE 2022. Conjure complements all of these tools. It reduces the number of dependencies that require monitoring, signing, and auditing.

Code generation and LLM security. LLMs for code, including Codex [10], CodeGen [11], StarCoder [12], and Code Llama [13], score well on HumanEval [10] and MBPP [14]. Pearce et al. [30] assess the security of GitHub Copilot's code contributions at IEEE S&P 2022, finding that approximately 40% of generated code contains vulnerabilities. He and Vechev [31] study security hardening of LLM-generated code at CCS 2023. Conjure addresses these concerns differently: rather than hardening generated code after the fact, it uses AST enforcement to structurally prevent dangerous constructs before execution.

Verified code generation. A growing body of work combines LLMs with formal verification. Bhatia et al. [32] present verified code transpilation with LLMs at NeurIPS 2024. Wu et al. [33] integrate LLMs with automated program verification at ICLR 2024. Sun et al. [34] propose closed-loop verifiable code generation. Conjure's verification is lighter weight (example-based rather than formal), trading completeness for practicality and speed. Integrating formal verification of generated code against spec properties is a natural extension.

Sandboxing and isolation. Firecracker [17] and gVisor [18] provide hardware and kernel-level sandboxing. Bosamiya et al. [35] present provably-safe sandboxing using WebAssembly at USENIX Security 2022. Johnson et al. [36] build a verifiably secure WebAssembly runtime at IEEE S&P 2023. Wahbe et al. [37] introduced software-based fault isolation at SOSP 1993. These approaches constrain what imported code *can do*. Conjure takes a different tack: generated code *cannot reference* external packages at all, making the supply chain attack surface zero for replaced functionality.

Embedded inference. Local model execution through llama.cpp [19], MLX [20], and Ollama enables Conjure's architecture. Running inference on-device introduces no network dependency, no API key management, and no cloud trust requirement.

9 Limitations and Future Work

Platform scope. Conjure currently targets Python on Apple Silicon via MLX. Extending to other languages (JavaScript, Rust, Go) requires language-specific AST analysis, sandboxing, and model selection. The architecture is language-agnostic, but each target language needs its own verification pipeline.

Model capability ceiling. The 9B model cannot reliably generate complex algorithms like SHA-256 (which requires 64 specific round constants and precise bit manipulation) or full recursive-descent parsers. Our evaluation shows 12% of specifications fall in this "hard" category. Larger models (27B+) or domain-specific fine-tuning could raise this ceiling. We attempted fine-tuning using supervised learning, rejection sampling, DPO, and self-distillation; none outperformed the base model with the retry pipeline on our benchmark. The base model's diversity under sampling appears more valuable than specialization through fine-tuning.

Formal verification. Example-based testing catches functional errors but cannot prove correctness for all inputs. Integrating lightweight formal verification (e.g., proving that a sorting function returns a permutation of its input, or that an encoder-decoder pair round-trips) would provide stronger guarantees. The specification language already includes a `properties` field that describes invariants in natural language; translating these to machine-checkable assertions is a natural next step.

Specification effort. Writing a YAML specification requires understanding the function's interface and expected behavior. A shared specification registry (analogous to PyPI but for verified, import-free function specs) would enable community curation and amortize this effort across developers. Our translation scanner (§6.4) automates the identification step, but spec authoring remains manual.

Runtime overhead. The 6 GB model must be loaded into memory at least once per process lifetime. Applications that invoke Conjure infrequently may find this overhead prohibitive. The `conjure build` command pre-generates all specifications at build time, allowing the model to be unloaded before deployment. In this mode, the runtime cost is limited to SQLite cache lookups (0.3ms).

10 Conclusion

A locally-running language model can replace a significant fraction of common library dependencies with generated, verified, import-free code. On ConjureEval-100, Conjure produces correct implementations for 87.9% of specifications. Its AST enforcement prevents 10 CWE vulnerability categories. For the 40–52% of functionality it can replace, the supply chain attack surface drops to zero. Across 5 real applications, this translates to up to 20× reduction in auditable code surface.

Conjure does not eliminate all dependencies. Framework code, database drivers, and C extensions remain outside its scope. For these, existing tools are still needed. But for the pure-logic utility code that makes up a large share of most dependency trees, the tradeoff of 6 GB of model memory for zero supply chain exposure is concrete and measurable.

The 40% property-testing failure rate on random inputs is a real limitation that restricts deployment to non-safety-critical paths

without additional validation. We see this as a tractable problem: each failure can be addressed by adding edge-case examples to the specification, and property-based testing can be integrated into the build pipeline.

Conjure is a step toward a future where the default for utility code is generation from auditable specifications rather than blind trust in packages written by strangers.

References

[1] The event-stream incident, 2018. https://blog.npmjs.org/post/180565383195
[2] ua-parser-js supply chain attack, 2021. GHSA-pjwm-rvh2-c87w.
[3] colors.js and faker.js sabotage, 2022. https://snyk.io/blog/open-source-npm-packages-colors-faker/
[4] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *USENIX Security*, 2019.
[5] Snyk. State of open source security, 2024. https://snyk.io/lp/state-of-open-source-2024/
[6] M. Alfadel, D. E. Costa, and E. Shihab. Empirical analysis of security vulnerabilities in Python packages. *Empirical Software Engineering*, 28(59), 2023.
[7] PyPA Advisory Database. https://github.com/pypa/advisory-database
[8] Qwen Team. Qwen3.5: Towards native multimodal agents, 2026. https://huggingface.co/Qwen/Qwen3.5-9B
[9] OptiQ: Mixed-precision quantization for MLX models. https://pypi.org/project/mlx-optiq/
[10] M. Chen et al. Evaluating large language models trained on code. *arXiv:2107.03374*, 2021.
[11] E. Nijkamp et al. CodeGen: An open large language model for code with multi-turn program synthesis. In *ICLR*, 2023.
[12] R. Li et al. StarCoder: May the source be with you! *arXiv:2305.06161*, 2023.
[13] B. Rozière et al. Code Llama: Open foundation models for code. *arXiv:2308.12950*, 2023.
[14] J. Austin et al. Program synthesis with large language models. *arXiv:2108.07732*, 2021.
[15] Supply chain Levels for Software Artifacts (SLSA). https://slsa.dev/
[16] Sigstore: Software signing for everyone. https://sigstore.dev/
[17] A. Agache et al. Firecracker: Lightweight virtualization for serverless applications. In *NSDI*, 2020.
[18] gVisor: Container runtime sandbox. https://gvisor.dev/
[19] G. Gerganov. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp
[20] Apple. MLX: An array framework for Apple Silicon. https://github.com/ml-explore/mlx
[21] P. Ladisa, H. Plate, M. Martinez, and O. Barais. SoK: Taxonomy of attacks on open-source software supply chains. In *IEEE S&P*, 2023.
[22] M. Ohm, H. Plate, A. Sykosch, and M. Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *DIMVA*, Springer, 2020.
[23] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *NDSS*, 2021.
[24] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *ACM CCS*, 2008.
[25] M. Balliu et al. Challenges of producing software bill of materials for Java. *IEEE Security & Privacy*, 2023.
[26] W. Guo et al. An empirical study of malicious code in PyPI ecosystem. In *IEEE/ACM ASE*, 2023.
[27] S. Neupane, G. Holmes, E. Wyss, D. Davidson, and L. De Carli. Beyond typosquatting: An in-depth look at package confusion. In *USENIX Security*, 2023.
[28] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.
[29] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams. What are weak links in the npm supply chain? In *ICSE-SEIP*, 2022.
[30] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In *IEEE S&P*, 2022.
[31] J. He and M. Vechev. Large language models for code: Security hardening and adversarial testing. In *ACM CCS*, 2023.
[32] S. Bhatia, J. Qiu, N. Hasabnis, S. A. Seshia, and A. Cheung. Verified code transpilation with LLMs. In *NeurIPS*, 2024.
[33] H. Wu, C. Barrett, and N. Narodytska. Lemur: Integrating large language models in automated program verification. In *ICLR*, 2024.
[34] C. Sun, Y. Sheng, O. Padon, and C. Barrett. Clover: Closed-loop verifiable code generation. In *SAIV*, 2024.
[35] J. Bosamiya, W. S. Lim, and B. Parno. Provably-safe multilingual software sandboxing using WebAssembly. In *USENIX Security*, 2022.
[36] E. Johnson et al. WaVe: A verifiably secure WebAssembly sandboxing runtime. In *IEEE S&P*, 2023.
[37] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM SOSP*, 1993.
[38] P. Przymus and T. Durieux. Wolves in the repository: A software engineering analysis of the XZ Utils supply chain attack. In *MSR*, 2025.