Vibenix: An AI Assistant for Software Packaging with Nix

Martin Schwaighofer[◇]
Johannes Kepler University Linz
martin.schwaighofer@ins.jku.at

Martim Monis[◇]
INESC-ID and IST, University of Lisbon
martimrosam@tecnico.ulisboa.pt

Nuno Saavedra
INESC-ID and IST, University of Lisbon
nuno.saavedra@tecnico.ulisboa.pt

João F. Ferreira
INESC-ID and Faculty of Engineering, University of Porto
joao@joaoff.com

Rene Mayrhofer
Johannes Kepler University Linz
rm@ins.jku.at

***Abstract*—Implicit and floating dependencies, uncontrolled network access, and other impurities in build environments create intransparent supply chains. Discovering the identity and chain of custody of every dependency that could have affected the output of a build process is an error-prone forensic and reverse engineering task. Hermetically isolated build steps, linked by hashes of their inputs and outputs, make functional package management a principled alternative, which ensures that software depends only and exactly on declared inputs listed in a reproducible build recipe. Generating build recipes, which satisfy these additional constraints, however, is a complex and time-consuming task, creating a barrier to adoption. We present Vibenix, an AI-powered assistant that automatically generates such recipes as Nix expressions. Vibenix employs an agentic architecture that leverages a large language model (LLM) to iteratively refine a build recipe based on build outcomes, guided by deterministic rules and a structured feedback loop. We evaluated Vibenix on a dataset of 472 packaging tasks from the Nixpkgs repository. Vibenix successfully builds 424 of the 472 tasks (89.8%). Manual validation of a subset of 48 packages showed that 45.8% of Vibenix-built packages are functionally correct. Vibenix's post-build refinement process, based on the evaluator–optimizer pattern and leveraging a VM environment for runtime testing, results in an additional 37.5% increase in functionally correct packages, demonstrating that this refinement mechanism is a promising and important contribution to automated software packaging. Vibenix demonstrates how LLMs, as an unreliable building block, can be leveraged to generate rigorously defined, complete, and easy to audit dependency trees, for real-world software projects.**

***Index Terms*—software supply chain security, software packaging, agentic systems, hermetic builds, large language models**

I. Introduction

Modern software systems are built from complex dependency graphs, often comprising thousands of third-party components and toolchains. This complexity hinders debugging and introduces serious security risks: dependencies may rely on the ambient system state, or fetch arbitrary code from the internet during builds, leading to non-deterministic behavior across machines and time. The resulting intransparency of build systems has contributed to a wave of high-profile supply chain attacks [1], [2], prompting renewed focus on build integrity, dependency isolation, and reproducibility [3], [4].

Hermetically isolated software packaging, supported by tools such as Nix [5] and Bazel [6], have emerged as a principled defense against such threats [7]. By construction, hermetic packaging ensures that each build step is executed in a hermetically isolated environment that only contains the intended dependencies, thus segregating intended and circumstantial dependencies by design. Hermetically isolated build environments rule out reliance on global system state, uncontrolled network access, or implicit environment variables, making it possible to reason more soundly about what code is built, where it comes from, and how it behaves at runtime. Writing package recipes to build software in such an environment requires additional effort by developers and package maintainers. Moreover, packaging software with Nix is often described as a frustrating experience, particularly to newcomers.[1]

We present Vibenix, a novel AI assistant for automated Nix packaging. Vibenix analyzes a given software project and, from an initial template, iteratively refines it until it is able to build the project in a hermetically isolated environment.

Vibenix is the first agentic system that tackles hermetically isolated builds: it runs builds inside a fully isolated sandbox, interprets the resulting logs, and iteratively refines the Nix expression under progress rules. The two key techniques that we demonstrate to be key for successful automated package building are (1) a log-driven loop that uses specialized tools to incrementally repairs build failures based on concrete evidence from sandboxed execution, and (2) a post-build refinement process based on the evaluator-optimizer pattern, which inspects the runtime behaviour of the successfully built package

[◇]These authors contributed equally to this work.

[1]In the 2024 Nix Community Survey [8], among respondents who do not consider Nix part of their usual toolset, the most commonly cited barriers were "still learning" (32), "barrier to entry is too high" (27), "confusing documentation" (22), and "the Nix language is an obstacle" (20).

Workshop on LLM Assisted Security and Trust Exploration (LAST-X) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-05-3
https://dx.doi.org/10.14722/last-x.2026.23035
www.ndss-symposium.org

and applies targeted improvements to correctness, dependency declarations, and package quality.

To assess Vibenix, we conducted an empirical evaluation on a large set of packaging tasks from the Nixpkgs repository (472 tasks). The system was able to automatically build the vast majority of tasks (89.8%), and manual inspection confirmed that a substantial portion of these builds were functionally correct. Furthermore, Vibenix's post-build refinement process further improved functional correctness significantly. These results indicate that Vibenix not only automates the build process effectively but also enhances the reliability of generated packages through targeted refinements. In summary, this paper makes the following novel contributions:

- We present Vibenix, an AI assistant that automatically produces hermetic Nix expressions for real-world projects, using two key techniques: a log-driven loop that incrementally fixes build failures, and a post-build evaluator–optimizer process that improves correctness and package quality.
- We present *NixBench*, a dataset of 472 curated and recent packaging tasks from the Nixpkgs repository. We additionally provide a detailed and reproducible methodology for constructing this dataset, enabling future extensions while mitigating data-leakage risks as time progresses and the training cutoffs of large language models continue to advance.
- We evaluate Vibenix across this diverse dataset of open-source software packaging tasks, and show that it can autonomously satisfy 89.8% of these requests.
- We provide Vibenix as open source[2] and all the data and code associated with our experiments.[3]

Vibenix advances trustworthy software builds by making functional package management more accessible and automated. This in turn enables reasoning about source provenance that is independent of original suppliers, because it relies on the authoritatively enforced dependency graph constructed by Nix. Other measures to obtain strong source provenance data require participation by suppliers or package registries.

II. Software Packaging with Nix

Hermetic packaging in Nix involves writing a functional Nix expression that declaratively specifies every build step of the complete dependency graph, which is then executed in the Nix build sandbox.

Listing 1 shows a functional Nix expression generated by Vibenix that packages a Python library called `stackstac` [4]. In addition to the Nix language the listed code heavily relies on the infrastructure code and dependencies provided by Nixpkgs[5] to abstract away the details. When using Nix like this, a package definition is structured as a function that accepts a set of arguments representing other packages on which the definition depends and language-specific infrastructure code that

```
1  { lib
2  , python3Packages
3  , fetchFromGitHub
4  , gdal
5  }:
6
7  python3Packages.buildPythonPackage rec {
8    pname = "stackstac";
9    version = "0.5.1";
10
11   src = fetchFromGitHub {
12     owner = "gjoseph92";
13     repo = "stackstac";
14     rev = "v${version}";
15     hash =
16       "
     sha256-7mll2jiZEx0xPDATWCBgqle4I68pkL6BeTLHKRmXEss=
     ";
17   };
18
19   pyproject = true;
20
21   build-system = with python3Packages; [
22     pdm-pep517
23   ];
24
25   dependencies = with python3Packages; [
26     xarray
27     dask
28     rasterio
29     pystac-client
30     numpy
31     pyproj # Added missing dependency
32     pandas # Added based on evaluator feedback
33   ];
34
35   buildInputs = [
36     gdal
37   ];
38
39   doInstallCheck = true;
40   installCheckPhase = ''
41     ${python3Packages.python.interpreter} -c "import
   stackstac"
42   '';
43 }
```

Listing 1: Nix package definition for stackstac.

supports the build (line 1 to 5). Line 7 introduces a function invocation with a block that defines a Python package. Within this block, the package is assigned the name `stackstac` (line 8) and the version `0.5.1` (line 9). In Nixpkgs, the `src` attribute of a package specifies the location of its source code. It typically references a file path, or a fetcher function that retrieves the source code of the package from a Git repository, or tarball URL. In our example, a fetcher function is then used to obtain the Python package's source code (line 11 to 17).

Lines 19 to 23 ensure that the build invokes the correct build infrastructure for Python projects provided in the `nixpkgs` repository. From lines 25 to 37, project dependencies are explicitly declared. Among them, `gdal` represents a seamlessly integrated C/C++ dependency that originates outside the Python ecosystem. The accompanying comments were automatically inserted by Vibenix during various stages of the fully automated packaging process. Finally, lines 39 to 42 introduce a build-time check that verifies that the package can be successfully imported without raising errors.

Packaging software with Nix is generally more challenging

[2]https://github.com/mschwaig/vibenix
[3]https://github.com/mschwaig/vibenix-paper-experiments
[4]https://github.com/gjoseph92/stackstac
[5]https://github.com/nixos/nixpkgs

than with other tools, primarily because many build systems implicitly rely on environment dependencies or do not manage the full transitive closure of their dependency graphs. Even language-specific package managers that offer some degree of hermetic isolation, such as `cargo` for Rust, typically include escape hatches that allow unresolved system-level dependencies. In case of Rust for example, whatever gets executed inside a `build.rs` file is not covered by `Cargo.lock` and cannot be reasoned about within the corresponding dependency graph. As a result, such packages can easily evade software supply chain analyses, highlighting a fundamental limitation that our work seeks to overcome by easing the Nix packaging process.

Nix also enforces strict build-time purity by disallowing network access during builds unless all resources are accompanied by precomputed cryptographic hash values. This guarantees that all dependencies are explicitly declared, allowing users to fully understand, track and archive the set of dependencies of a package with confidence in its completeness.

From a security perspective, Adkins et al. [9] found that hermetically isolated builds facilitate the analysis and enforcement of policies on build inputs, ensure the integrity of third-party imports, and simplify the rapid deployment of critical software updates.

However, Nix also introduces unique challenges. We already mentioned disallowing network access in the build sandbox, which requires extra effort and consideration during the packaging process. The sandbox environment is also practically empty except for declared dependencies. Additionally, Nix does not adhere to the Filesystem Hierarchy Standard (FHS), which often makes it necessary to patch source code that assumes dependencies reside in usual FHS-prescribed locations. This tends to lead to runtime failures in build outputs, which have to be addressed iteratively as part of the packaging process.

III. Vibenix

Vibenix employs an agentic architecture that leverages an LLM to support the fully automated packaging process. Throughout this process, the LLM performs two primary tasks based on log output: fixing encountered errors and evaluating whether the model has made progress.

What makes Nix a good tool for applying LLMs to the problem of software packaging is that it provides a mechanism to clearly scope tasks while enabling iteration within the narrowly defined scope of a package build over the entire build process at once. Both of these properties make auditing and iteration on tasks easier than, for example, general-purpose coding tasks or imperatively setting up a build environment step by step. The roughly linear nature of build processes also makes it easier than for general-purpose coding tasks to distinguish working solutions from failing solutions for progress evaluation. The final output of the process is also easier to validate than for a general-purpose coding task, because making an existing application work is a more clearly specified goal.

A. Packaging Pipeline

In this section, we provide a step-by-step description of Vibenix's packaging process. Figure 1 presents an overview of this process.

The process begins when the user provides the URL of the target project to Vibenix.

1) Generate Fetcher: To fetch the source code of the project, Vibenix uses a Nix fetcher invocation. To generate this fetcher invocation, our tool relies on `nurl`[6], which fetches supported sources with Nix and turns them into a call to a suitable fetcher function with an associated content hash set by `nurl`. The expression assigned to the `src` attribute in Listing 1 shows an example of a fetcher invocation generated by `nurl`.

Vibenix uses the local Nix installation to execute the generated fetcher, which clones the project source code into the local Nix store. The resulting store path is recorded for later use within Vibenix, allowing it to retrieve relevant information about the project during subsequent stages of the packaging pipeline.

2) Analyze Project: Vibenix then prompts the LLM with information gathered from the project's source code, namely the project's root README file and root directory listing, to generate a summary that includes build tools, dependencies, and other details relevant to the build process. This prompt provides the model with tools to further analyze the project source at will.

3) Select Template: Rather than providing the LLM with a blank canvas, Vibenix guides its output by initializing it with a build tool or language-specific template that adheres to the style conventions of the `nixpkgs` repository. To select the appropriate template, Vibenix prompts the LLM using the previously generated summary to identify the most suitable starting point. After selecting the appropriate template, Vibenix populates its `src` attribute with the previously generated fetcher invocation.

4) Setup Nix Flake: Then Vibenix sets up a temporary project directory based on the selected template with both an additional `flake.nix` and `flake.lock` file, which provide a defined structure in terms of how the project build is invoked and which version of the existing `nixpkgs` package set it is based on. Vibenix takes advantage of existing package definitions from the `nixpkgs` repository, thereby eliminating the need to independently repackage each component required to build and run the project.

5) Execute Build: After the Nix flake is set up, Vibenix enters an agentic loop. This agentic loop makes multiple calls to the LLM until a successful build is reached or Vibenix reaches one of its stopping conditions. First, Vibenix uses the local Nix installation to build the current Nix expression and collect the build results. If the build succeeds, the package goes through a refinement process (see Section III-A10). Afterwards, the finished package is presented to the user, saved to disk, and the pipeline ends.

[6]https://github.com/nix-community/nurl

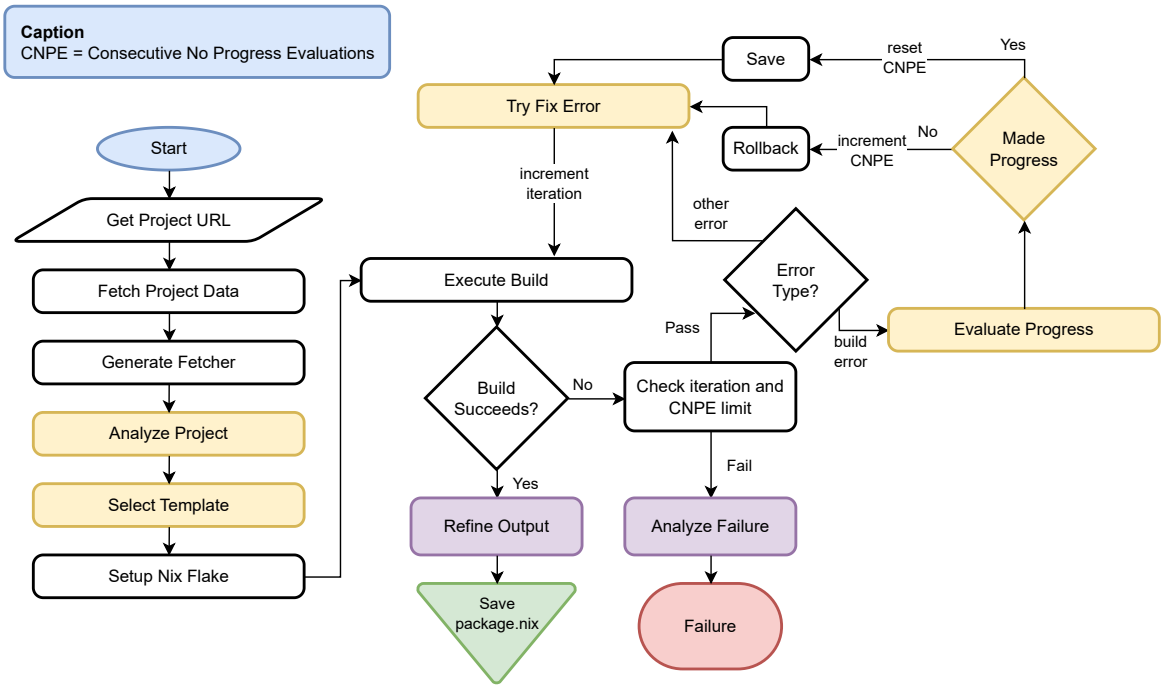

Fig. 1: Overview of Vibenix's packaging pipeline. Yellow and purple boxes indicate use of an LLM in these steps.

6) Stopping Conditions: If the build fails, Vibenix evaluates whether one of the following stopping conditions has been met:

1) the maximum number of allowed iterations has been exceeded;
2) the maximum number of consecutive evaluations without observed progress has been exceeded.

If one of these conditions is met, Vibenix stops its agentic cycle and outputs a failure report (see Section III-A11). Otherwise, the iterative process continues.

7) Handling Errors: Subsequently, Vibenix inspects the nature of the error that caused the build to fail. If the LLM managed to write a valid Nix expression that successfully evaluates into a dependency graph but fails during the build step, Vibenix proceeds to evaluate if progress has been made (See Section III-A8).

Otherwise, Vibenix continues, by attempting to fix the encountered error (See Section III-A9). Errors that lead to this path include hash-mismatches and invalid Nix expressions.

8) Evaluate Progress: For build errors, Vibenix evaluates whether progress was made in the packaging process. During the first evaluation, Vibenix always assumes that progress has been made, as there is no prior run available for comparison. From the second evaluation onward, Vibenix compares the log files of the current and previous builds to determine whether progress has been made.

To do so, Vibenix starts by preprocessing the logs. When the log size does not exceed 100 lines, the complete two sets of logs are included in the comparison. Otherwise, Vibenix identifies the earliest line in the current attempt that does not appear in the log from the previous attempt. All log content preceding the index of this divergent line is discarded in both sets of logs. This heuristic is grounded in the observation that, due to the parallel nature of operations, log entries may not be sequentially ordered. Therefore, detecting the first unmatched line offers a practical approximation of the divergence point between both attempts. Some additional truncation may also be required in this case.

Subsequently, Vibenix supplies the LLM with the two truncated logs, accompanied by a prompt instructing it to assess which build progressed further. The LLM outputs `PROGRESS` if it determines that the new attempt advanced beyond the previous one. In this case, Vibenix resets the number of consecutive evaluations without observed progress, and saves the current attempt. Otherwise, the LLM outputs a different status like `REGRESS` or `STAGNATION`, in which case Vibenix increments the number of consecutive evaluations without observed progress, and rolls back to the last saved attempt.

Tracking the number of consecutive evaluations without observed progress serves as a safeguard against unnecessary consumption of computational resources when the model fails to advance within the packaging process.

9) Try Fix Error: In this step, Vibenix prompts the LLM to fix the error in the current Nix expression. Vibenix provides the LLM with an input consisting of the current Nix expression, the error logs, the summary generated in Section III-A2, and supplementary framework or language-specific context based on the selected template. Vibenix also provides the model with the following tools:

1) **Edit tools**: Directly returning the entire packaging expression, or even snippets, at the end of error fixing prompts can often introduce noise, especially surrounding

long strings like source hashes, and increased costs. For this reason, the model has access to standardized tools to perform localized edits to the packaging expression and view its current state.
2) **Package Search**: Given a query, the function retrieves relevant packages from the `nixpkgs` repository by performing a search on all the packages available.
3) **Find Nix Package Output**: Given a relative file path, the function retrieves packages in Nixpkgs that provide this file path in their output. This function leverages the `nix-locate` command.
4) **File-centric tools**: The agent has access to tools that interact with the file system, including listing directory contents, identifying file types and sizes, reading file contents, and performing pattern-based searches. These operations are supported in both the project source directory and a commit of the `nixpkgs` repository.

In the case of hash mismatch errors, Vibenix limits the input of the LLM to the current Nix expression and the error logs. For such scenarios, it employs a specialized prompt that includes guidance tailored to resolve this specific class of issues. Hash mismatches occur when using an unknown content hash of a network resource, such as a GitHub repository. Since Vibenix does not know correct hashes in advance, it fills in a dummy hash value (`lib.fakeHash`). Subsequently, Nix will return a hash mismatch error, which includes the correct hash value, which Vibenix will then fix. This trust-on-first-use workflow mirrors best practice in manual Nix packaging [10].

10) Refinement: Once an initial build is successfully produced, either through the first build attempt or during the agentic loop, the core goal is considered to be reached. However, Vibenix also includes an optional refinement step. This step is specifically designed to improve package quality, presentation, and detect and correct errors not detectable during build. Its implementation draws inspiration from the agentic workflow pattern of the Evaluator-optimizer [11].

Each iteration of the refinement process primarily involves two prompts to the LLM. The first, the *Evaluator* prompt, is provided with the Nix expression and relevant project information. Its task is to identify and describe a single specific improvement in packaging. Subsequently, the *Optimizer* prompt receives the *Evaluator*'s feedback and is tasked with implementing the suggested change within the expression.

If the code produced by the *Optimizer* is built successfully, it replaces the previous expression; otherwise, it goes through the same packaging loop as before refinement for a short number of iterations, in an attempt to fix the errors introduced. From this inner loop, the package is either built successfully once more, or the maximum number of error fixing iterations is reached and the changes are discarded. Regardless of success, the refinement process continues until a predefined maximum number of iterations is reached. Once refinement ends, the final packaging code is one produced during this process or the one initially provided to it.

Given that the packaging expression received by the *Evaluator* prompt always builds successfully, this prompt can leverage additional tools that interact with the build process results. One such tool is `run_in_vm`, which allows the LLM to specify a shell script to run in an isolated NixOS virtual-machine containing the built package, generic useful commands, and other packages optionally specified by the model (for example, Python with the built package in its scope).

This tool allows Vibenix to inspect build outputs, giving further context for diagnosing issues, and to correct errors that only appear during program execution, and so further increasing the likelihood that the final packaging expression is sound.

11) Failure Analysis: If, by the end of the agentic loop, Vibenix fails to produce a successfully building Nix expression, it goes through a last failure analysis step. In the first prompt, it receives the packaging code, the most recent build error that led to progress, and relevant project context. Then a concise explanation for the failure is asked. In the second prompt, the LLM is tasked with classifying the generated explanation into one of several predefined categories, such as *dependency not in Nixpkgs*, *missing lock file*, or *build downloads from network*.

The failure analysis description and the failure type are presented to the user. This feedback provides insight into the root cause of the packaging issue, enabling the user to better understand the problem and then manually intervene.

B. Technical Implementation

Vibenix and its algorithm are written in Python, and exposed to the user as a command line application. The LLM integration is written using the `pydantic-ai`[7] framework. As mentioned in Section III-A1 it uses the local Nix installation. See the `README.md` file for how to invoke the project. For desktop usage, Vibenix prompts users to select a model provider, a model, and to provide the relevant API keys. Usage as part of our CI setup is described in Section IV-B1.

Due to space constraints we cannot incorporate prompts into the descriptions of this work. All prompts are available as `jinja2` templates in Markdown format in the Vibenix repository. The build tool and language-specific templates that are incorporated into some of the prompts were generated ahead of time by prompting the proprietary tool `claude-code`[8] with access to the Nixpkgs repository. This was done for nine languages based on popularity. Vibenix falls back to a less effective generic template otherwise. All templates can be found in the Vibenix repository.

IV. Evaluation

In this section, we first describe the dataset used to evaluate Vibenix, followed by an explanation of the evaluation methodology, and conclude by presenting our results.

[7]https://github.com/pydantic/pydantic-ai
[8]https://github.com/anthropics/claude-code

A. Dataset

To assess the effectiveness of our approach, we require a dataset comprising software projects that can be packaged using Nix. One of the cornerstones of the Nix community is the Nixpkgs package repository[9]. It is a monorepo, and the largest repository of open source software, as tracked by repology[10], with, at the time of writing, around 129,000 packages maintained by a community of 4,257 maintainers.

To construct our dataset, we start by collecting all commits from the Nixpkgs repository with a commit author date between August 1st, 2025 and November 20th, 2025 that introduced either a *package.nix* or *default.nix* file. The addition of these files signals the creation of a new package. In total, this process yielded 1,146 commits. We selected a commit author date of August 1st, 2025 as the starting point to prevent data leakage, as the models used in our evaluation were trained on data with a cutoff prior to this date. The end date, November 20th, 2025, corresponds to the point in time at which the dataset was collected. Future studies can replicate our dataset construction procedure by adjusting the date range to fit their requirements accordingly.

Next, we filtered commits to retain only those that introduced packages with at least six lines of code, a defined `meta` attribute, and an explicit Nix fetcher. These criteria restrict the dataset to real package definitions that comply with the Nixpkgs requirement that all packages provide a meta attribute [12], while excluding aliases, wrappers, and module definitions. This filtering yielded 1,014 commits. We then examined the commit messages to identify those that follow the Nixpkgs conventions for introducing new packages (e.g., *"init at X.Y.Z"*). We retained only the commits whose messages matched these patterns. After this step, 761 commits remained.

TABLE I: Summary of number of commits removed from our dataset for each exclusion criterion.

Category	Count
Add multiple packages	3
Modify existing packages	5
Require patches	26
Add lock files	13
Change the *all-packages.nix*	6
Marked as broken	1
Marked as unfree	21
Evaluation phase (more than 30 seconds)	3
More than one criterion	7

Out of these 761 commits, we excluded those that:

1) **Add multiple packages.** Such commits introduce several packages at once. Including these commits would require disentangling their relationships to isolate the packaging intent for each package, an effort that would significantly complicate dataset construction without yielding insights relevant to our tool. By focusing on single-package introductions, we ensure clean, self-contained examples.
2) **Modify existing packages.** Our tool is designed to introduce new packages, not update or refactor existing ones. Commits that modify other packages would require the tool to reason about and safely change those other packages. Excluding such commits ensures that each datapoint corresponds to a scenario that our tool can meaningfully address.
3) **Require patches.** Some packages need source-code patches, which our tool does not attempt to infer or generate. Excluding these cases prevents confounding factors that are not related to the synthesis of Nix expressions and keeps the evaluation focused on the packaging logic rather than the modification of upstream code.
4) **Add lock files.** Commits that introduce lock files (e.g., for language-specific dependency managers) designate a specific set of additional packages as trustworthy. Doing this independently of the original author and in an automated manner has trust implications, which have yet to be addressed. Currently, Vibenix cannot generate such files or access the internet to do so.
5) **Change the *all-packages.nix*.** Commits modifying this central index often involve structural reorganizations or multi-package edits. Removing these commits keeps the dataset focused on isolated packaging tasks.
6) **Are marked as *broken* or *unfree*** Packages carrying these metadata flags indicate missing dependencies or licensing restrictions. Our tool currently assumes buildability under standard Nixpkgs conditions, and unfree packages introduce licensing restrictions that prevent their redistribution and would hinder the reproducibility and open availability of our dataset.
7) **Required more than 30 seconds to complete the Nix evaluation phase.** This timeout refers to the evaluation phase of Nix itself during which Nix expands the package expression and computes its derivation. We impose this limit to keep our data-processing scripts fast. Although some of these slow evaluations may simply be outliers that could be optimized or handled with additional engineering effort, they represent a very small fraction of packages. Given time constraints and their limited impact on the dataset, we opted to exclude them rather than tune our pipeline for these uncommon cases.

After applying these exclusion criteria, we obtain 676 commits.

Then, we attempted to build the packages introduced in each commit. For each target commit, we traversed the history of the most recent 150 tip of branch commits of the *nixpkgs-unstable* branch in the Nixpkgs repository, which correspond to the 150 most recent channel states of *nixpkgs-unstable* from early July to the end of the experiment period, to identify the tip of branch commit immediately preceding the addition of the new package. These recent channel states serve as a reference window that ensures all evaluated changes are based on a reasonably up-to-date version of the repository. This constraint avoids including package additions whose base

[9]https://github.com/nixos/nixpkgs
[10]https://repology.org/

commits are substantially older and may have been circulating publicly for an extended period, which could create data leakage issues in our evaluation. Five commits were excluded because none of these channel states was an ancestor of theirs. We then attempted to build each package on top of its corresponding prior commit, imposing a 15-minute time limit on each build attempt. If the build succeeded, we recorded that specific *nixpkgs-unstable* commit to represent the repository state on which Vibenix should be evaluated. If the build failed, we discarded the package. In total, we excluded 122 commits due to build failures, resulting in 549 commits after this step.

Of these commits, we retained only those from which a single fetcher could be reliably extracted. Vibenix accepts as input a single fetcher responsible for retrieving the software to be packaged. Restricting our dataset in this way keeps the focus on the packaging logic itself, rather than on resolving multi-fetcher workflows, which would require substantial additional engineering effort and are not representative of common practice in Nixpkgs. Consequently, 14 commits were excluded for introducing packages with multiple fetchers, and an additional two were removed due to malformed fetchers or the absence of a detectable fetcher.

Finally, we excluded 52 commits in which the upstream repository already contained Nix expressions. In such cases, Vibenix could extract and reuse existing packaging logic through its project-analysis tools, which would compromise the validity of our evaluation by allowing the system to locate a pre-existing solution rather than synthesizing one or having it in the training data. Additionally, eight commits were removed for using nonstandard attribute names, such as *finalAttr.version*, instead of the conventional Nixpkgs form *finalAttrs.version*. Supporting these rare packages would require substantial engineering effort without yielding meaningful insights. We also excluded one package that was identified as a duplicate.

In the end, our final evaluation dataset comprises 472 packaging tasks, each associated with the corresponding software to be packaged, its version, the specific Nixpkgs commit on top of which it should be packaged, and a Nix fetcher. We refer to this dataset as *NixBench*.

B. Evaluation Methodology

1) Experimental Setup: We run our experiments in public inside dedicated GitHub Actions workflow runs. Each individual packaging request was run in isolation as a dedicated job.

a) Model comparison: We compare the performance of Vibenix when instantiated with different language models. Our evaluation includes models from multiple providers that span a range of sizes, costs, and licensing regimes (both closed-source and open-source). This analysis allows us to assess the extent to which Vibenix generalizes across model families and to evaluate how effectively different models perform the hermetically isolated software packaging task. For this comparison, we randomly selected a subset of 54 packaging tasks from NixBench. We used a subset of NixBench because running multiple experiments on the entire dataset would incur prohibitive computational costs, both in terms of time and monetary resources.

b) Assessing the Contribution of Vibenix's Components: For each model, we also conducted experiments in which all Vibenix tools and the progress-evaluation mechanism were disabled, allowing us to quantify the impact of these components on overall performance.

c) Evaluating Runtime Correctness: Even when a package builds successfully, the resulting software may still fail at runtime. In such cases, Vibenix has not achieved its intended goal, as the generated package is functionally incorrect. To assess this aspect, we manually tested successfully built packages in a VM-based environment. We focused on the packages produced by *Claude Haiku 4.5*, which demonstrated the strongest overall performance in our experiments. We then compared the number of correct packages before and after applying the refinement step, as the refinement process is designed to identify and address runtime errors.

d) Large-scale evaluation: Finally, we conduct a large-scale evaluation on NixBench using *Claude Haiku 4.5* as the deployed model. This enables us to assess with greater certainty how well Vibenix generalizes. Additionally, for this experiment, we analyze the success rate across iterations to assess the importance of the iterative loop in Vibenix.

The following time limits were applied in all experiments:

- a time limit of seven minutes for an individual build attempt configured in Nix, via Vibenix;
- a soft time limit of 30 minutes per package request in Python (30 plus seven extra minutes for pending builds);
- a hard 45 minutes time limit per package request configured in GitHub Actions.

Table IV in Appendix A reports the training cutoff dates for each model used in our experiments. Importantly, all training cutoffs precede the dates of the commits in NixBench.

V. Results

In this section, we describe the results of the experiments described in Section IV-B.

A. Model comparison

Table II reports the results of deploying Vibenix with different language models. Depending on the model used, Vibenix successfully produces buildable packages in between 44.4% and 90.7% of cases. *GPT-OSS-20B* yields the lowest performance; however, it is also by far the smallest model in our comparison. In contrast, *Claude Haiku 4.5* achieves the highest success rate, although it is also the most expensive model, costing nearly five times more than any other model we evaluated.

Interestingly, *GPT-OSS-120B*, an open weights model designed to fit into 80 GB of memory, achieves strong results with a success rate of 79.6%, only 11.1% lower than *Claude Haiku 4.5*. This suggests that it is feasible to achieve competitive performance using self-deployed models for the hermetically isolated packaging task.

TABLE II: Model performance comparison for a subset of 54 randomly selected packaging tasks.

Model	Success Rate	Avg Cost ($)		Avg Iter.	Timeouts
		All	Succ/Fail		
Claude Haiku 4.5					
Vibenix	90.7%	0.34	0.22/1.50	3.1	1.9%
Disabled	74.1%	0.05	0.03/0.11	7.0	1.9%
Gemini Flash 2.5					
Vibenix	81.5%	0.07	0.05/0.17	6.6	1.9%
Disabled	72.2%	0.06	0.03/0.16	8.1	1.9%
GPT-5-mini					
Vibenix	88.9%	0.06	0.04/0.20	3.1	11.1%
Disabled	88.9%	0.03	0.02/0.10	5.4	0.0%
Vibenix (1h timeout)	96.3%	0.07	0.06/0.12	3.0	0.0%
GPT-5-nano					
Vibenix	51.9%	0.04	0.02/0.07	4.4	46.3%
Disabled	63.0%	0.02	0.01/0.04	9.0	1.9%
Vibenix (1h timeout)	55.6%	0.09	0.03/0.15	6.8	35.2%
GPT-OSS-120B					
Vibenix	79.6%	N/A	N/A	6.9	1.9%
Disabled	70.4%	N/A	N/A	4.0	0.0%
GPT-OSS-20B					
Vibenix	44.4%	N/A	N/A	3.2	0.0%
Disabled	59.3%	N/A	N/A	8.8	0.0%

GPT-5-mini and *GPT-5-nano* exhibited noticeably slower token-generation speeds compared to the other models, resulting in a higher rate of packaging runs exceeding the time limit. This issue has been previously reported by the community and appears to be a known limitation of the GPT-5 family of models [13]–[15]. To better understand how these models perform when given additional time, we conducted an additional run for each model with the timeout increased from 30 minutes to 1 hour. Under this extended timeout, *GPT-5-mini* no longer experienced timeouts and became our best-performing model, producing buildable packages in 96.3% of the cases. In contrast, *GPT-5-nano* continued to struggle, still timing out in 35.2% of the runs even with the extended limit.

B. Assessing the Contribution of Vibenix's Components

When we disable Vibenix's tools and progress-evaluation, the success rate decreases by up to 16.6%. The largest drop occurs for *Claude Haiku 4.5*, which we hypothesize is due to this model making the most effective use of the tools provided.

For *GPT-5-mini*, the disabled configuration achieves the same success rate as Vibenix under the 30-minute timeout, but its performance increases by 7.4% when using the extended one-hour limit. Because the disabled version issues fewer tool calls and omits the progress-evaluation prompts, it consumes fewer tokens overall, allowing each iteration to run more quickly; as a result, it is less affected by the shorter timeout. A similar pattern is observed for *GPT-5-nano*. We believe that with a sufficiently longer timeout, Vibenix would also surpass the disabled configuration for *GPT-5-nano*.

Finally, *GPT-OSS-20B* actually shows a 14.9% increase in performance when using the disabled configuration. A closer inspection of the Vibenix runs with this model reveals that *GPT-OSS-20B* frequently struggles with correct tool invocation, often failing to adhere to the required formats, for example, by over-escaping JSON structures or producing malformed tool-call arguments. This behavior suggests that the model lacks sufficient reasoning and planning capabilities to reliably execute structured tool calls. As a result, disabling tool usage and progress evaluation removes a major source of errors for this model, thus improving its overall success rate.

C. Evaluating Runtime Correctness

After the model successfully produces a package, it performs three refinement iterations aimed at making targeted improvements. During this process, the model gains access to an additional tool that enables it to invoke a Bash script inside a VM containing the freshly built package. This allows the model to critically examine its own output and correct runtime defects, most notably missing or incorrectly specified runtime dependencies. Table III shows our results before and after refinement. Out of the randomly selected subset of 54 packages built by *Claude Haiku 4.5* for the model comparison, Vibenix failed to build six packages; these were therefore excluded from the manual validation. Overall, the refinement stage results in a 37.5% improvement in functional correctness, increasing the number of functionally correct packages from 22 to 40, as determined through a combination of code review, comparison of build outputs using diffing tools, and manual runtime inspection in a test VM by the authors.

TABLE III: Manual validation results for functional correctness.

Category	Correct	Incorrect
Without refinement	22 / 48 (45.8%)	26 / 48 (54.2%)
With refinement	40 / 48 (83.3%)	8 / 48 (16.7%)

Many corrected packages were misusing `buildPythonApplication` for Python libraries, which prevents the package from being imported. This is a practically meaningful distinction that refinement corrected 14 times.

D. Large-scale evaluation

For the large-scale evaluation, Vibenix successfully produced 424 buildable packages out of 472 packaging tasks, achieving a success rate of 89.8%. The average cost per package was $0.34; however, this value decreases to $0.23 when considering only the packages that succeeded, indicating a relatively low cost per package, especially when compared to the developer time required to perform the same task manually. The average time per package was 4 minutes and 14 seconds. These results indicate that Vibenix is a practical solution for real-world scenarios where reproducibility and supply chain security are essential, demonstrating that fully automated packaging is both feasible and beneficial. Future iterations of Vibenix, could potentially be deployed to automatically create

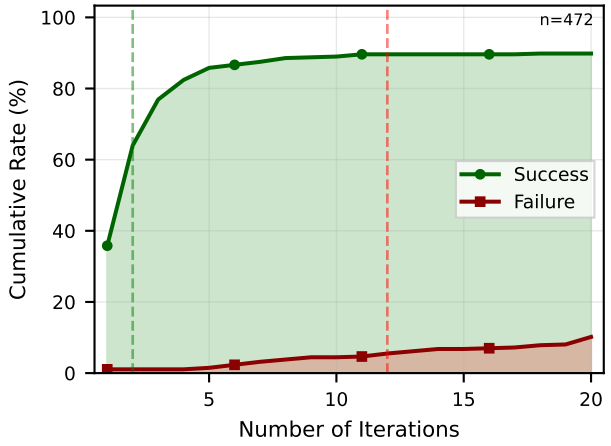

Fig. 2: Success and failure rates by iteration in a large-scale evaluation of 473 packaging tasks.

or extend a package sets like Nixpkgs based on user-provided packaging requests.

Figure 2 shows the cumulative success and failure rates across iterations. Almost all successful builds occur within the first five iterations, after which the success rate grows more slowly and plateaus only at iteration 18. This suggests that the iterative approach of Vibenix contributes to a higher overall success rate compared to a single attempt.

VI. Related Work

We focus on hermetic builds and their role in software supply chain security, which highlight the importance of secure and reproducible development practices; and on automated software packaging, which discusses previous work to streamline the creation of development and deployment environments.

A. Software Supply Chain Security and Hermetic Builds

Hermetically isolated builds are widely recognized as a stepping stone towards reproducible builds [16] and are supported by tools such as Bazel [6], Guix [17], Nix [5] and Gitian[11]. Both concepts strengthen software supply chain security [18]–[21]. For example, security researchers have explored hermetic builds to mitigate software supply chain security issues [22], [23] and to maintain an indexed archive of binary software packages [23].

Beyond security, Cloud-based Build Systems [24], which in addition to hermetically isolated builds provide output lookups by hash, like Nix, Guix and Bazel, have demonstrated their ability to enable reproducible builds at scale [7], [20], [25], and facilitate the creation of indexed archives of source code and binary packages to facilitate the reproducibility of science [26] and aid practical debugging. However, despite these advantages, recent findings by Alfadel and McIntosh [27] reveal a trend of organizations migrating away from Bazel due to unexpectedly high maintenance costs. This highlights a critical barrier to broader adoption and positions our work as a means to address this challenge by lowering the maintenance burden associated with hermetic build systems.

B. Automated Software Packaging

There have been some efforts to facilitate or automate software packaging with Nix. `nix-ai-help`[12] is an LLM-powered CLI tool providing various functionalities for Nix users, including automated project packaging through its command `package-repo` and build failure analysis through its command `build`. In comparison to Vibenix, `nix-ai-help` employs a single-shot LLM inference for package generation, lacking iterative improvements based on build output or progress evaluation. Moreover, `nix-ai-help` does not integrate build failure analysis with the project packaging process.

Furthermore, the `nix-init`[13] tool also offers automated packaging of projects with Nix, but does not leverage LLM or implement any iterative approach. It is also limited to projects in Rust, Go, and Python.

Recent work has explored automated pipelines to execute test suites at scale, often to construct benchmarks, such as BugSwarm [28] and GitBug-Java [29]. These efforts typically focus on a single programming language and assume specific CI/CD platforms. Complementary to this, Eliseeva et al. [30] propose EnvBench, a benchmark focused specifically on environment setup, covering 329 Python and 665 JVM-based repositories. They evaluate several LLM-based strategies and find that even the best agent successfully configures only 6.69% of Python and 29.47% of JVM repositories, highlighting the difficulty of the task and the limitations of current techniques. ExecutionAgent [31] introduces a general-purpose LLM-based agent capable of setting up environments and executing test suites across diverse languages and build systems. Distinct from existing solutions outside of the Nix ecosystem, Vibenix advances supply chain security by automatically enforcing hermetic builds, closing knowledge gaps that arise from non-deterministic and environment-dependent builds.

VII. Conclusion

In this work, we introduced Vibenix, an AI assistant for automated Nix packaging, addressing a critical challenge in software supply chain security. By combining functional package management with an agentic LLM-based architecture, Vibenix demonstrates that fully automated packaging is both feasible and practical. Our large-scale evaluation on real-world Nixpkgs packaging tasks shows promising results: Vibenix produces buildable packages in 89.8% of the cases, of which 83.3% are functionally correct. Many AI-for-security efforts repeatedly tackle higher-level tasks without addressing the solvable but fundamental challenge of linking artifacts and source code. Vibenix closes this gap by providing a structured,

[11] https://github.com/devrandom/gitian-builder

[12] https://github.com/olafkfreund/nix-ai-help

[13] https://github.com/nix-community/nix-init

verifiable and automated approach to packaging projects, creating a solid foundation to build scalable and trustworthy tools and workflows on.

APPENDIX

TABLE IV: Knowledge cutoff and release dates.

Model	Knowledge Cutoff	Release Date
Claude Haiku 4.5	Feb 2025[a]	Oct 1, 2025[b]
Gemini Flash 2.5	Jan 2025[c]	Jun 17, 2025[d]
GPT-5-mini	May 31, 2024[e]	Aug 7, 2025[g]
GPT-5-nano	May 31, 2024[f]	Aug 7, 2025
GPT-OSS-120B	Jun 1, 2024[h]	Aug 5, 2025[j]
GPT-OSS-20B	Jun 1, 2024[i]	Aug 5, 2025

[a] https://docs.anthropic.com/en/docs/about-claude/models/overview
[b] https://www.anthropic.com/news/claude-haiku-4-5
[c] https://deepmind.google/models/gemini/flash/
[d] https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-5-flash
[e] https://platform.openai.com/docs/models/gpt-5-mini
[f] https://platform.openai.com/docs/models/gpt-5-nano
[g] https://openai.com/index/introducing-gpt-5/
[h] https://platform.openai.com/docs/models/gpt-oss-120b
[i] https://platform.openai.com/docs/models/gpt-oss-20b
[j] https://openai.com/index/introducing-gpt-oss/

A. Data Contamination Prevention

All knowledge cutoff dates presented in Table IV precede the commits in NixBench, preventing training data contamination. Anthropic is the only provider publishing a post-training and fine-tuning cutoff date (July 2025), providing additional assurance against contamination. We deemed the one-week overlap between OpenAI model releases (August 5-7, 2025) and our dataset start (August 1, 2025) acceptable, assuming fine-tuning datasets would not change in a relevant manner one week before release. The final experiments presented in this work were performed between December 8 and December 15, 2025.

B. Author Contributions

Martin Schwaighofer and Martim Monis contributed equally to this work as first authors. Martin Schwaighofer initiated the project, provided the overall vision and direction, and led the implementation. Martim Monis designed and implemented the refinement process, implemented tools and other features, improved model prompts, and debugged and fixed numerous bugs throughout the codebase. Both collaborated extensively on the analysis of results. Nuno Saavedra led the writing effort, and critically shaped the scientific presentation, in terms of targeted goals and presented results. He also contributed to the implementation, introducing function calling and simplifying the agentic loop. João F. Ferreira and René Mayrhofer contributed as advisors, with João F. Ferreira additionally contributing to the writing and literature review.

Detailed attribution of code changes can be found in the commit history of the project repositories.

ACKNOWLEDGMENT

This work has been supported by the JKU LIT Secure and Correct Systems Lab and Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, Österreichische Staatsdruckerei GmbH, and the State of Upper Austria.

This work was supported by Fundação para a Ciência e a Tecnologia (FCT): N. Saavedra by grant BD/04736/2023; M. Monis, N. Saavedra, and J. F. Ferreira by projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025), UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025) and the 'InfraGov' project, with ref. n. 2024.07411.IACDC (DOI: 10.54499/2024.07411.IACDC), funded by the 'Plano de Recuperação e Resiliência (PRR)' under the investment 'RE-C05-i08 - Ciência Mais Digital', measure 'RE-C05-i08.M04' (in accordance with the FCT Notice No. 04/C05 i08/2024), framed within the financing agreement signed between the 'Estrutura de Missão Recuperar Portugal (EMRP)' and the FCT as an intermediary beneficiary.

REFERENCES

[1] S. Peisert, B. Schneier, H. Okhravi, F. Massacci, T. Benzel, C. Landwehr, M. Mannan, J. Mirkovic, A. Prakash, and J. B. Michael, "Perspectives on the solarwinds incident," *IEEE Security & Privacy*, vol. 19, no. 2, pp. 7–13, 2021.

[2] P. Przymus and T. Durieux, "Wolves in the repository: A software engineering analysis of the XZ utils supply chain attack," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 91–102, IEEE, 2025.

[3] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1509–1526, IEEE, 2023.

[4] L. Williams, G. Benedetti, S. Hamer, R. Paramitha, I. Rahman, M. Tamanna, G. Tystahl, N. Zahan, P. Morrison, Y. Acar, *et al.*, "Research directions in software supply chain security," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–38, 2025.

[5] E. Dolstra, M. De Jonge, E. Visser, *et al.*, "Nix: A safe and policy-free system for software deployment.," in *LISA*, vol. 4, pp. 79–92, 2004.

[6] Google, "Bazel." https://bazel.build/, 2024. Accessed: 2025-07-14.

[7] J. Malka, S. Zacchiroli, and T. Zimmermann, "Does functional package management enable reproducible builds at scale? yes.," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pp. 775–787, IEEE, 2025.

[8] Nix Marketing Team, "Nix community survey 2024 results." https://discourse.nixos.org/t/nix-community-survey-2024-results/55403, 2024. 2290 respondents; analysis at https://github.com/GuillaumeDesforges/nix-survey-analysis-2024.

[9] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield, *Building secure and reliable systems: best practices for designing, implementing, and maintaining systems*, ch. 14. O'Reilly Media, 2020.

[10] "Updating source hashes." https://nixos.org/manual/nixpkgs/stable/#sec-pkgs-fetchers-updating-source-hashes, 2025. [Online; accessed 11-December-2025].

[11] Anthropic, "Building Effective Agents," 2024. Accessed 14-07-2025.

[12] NixOS Project, "Contributing to nixpkgs packages: Meta attributes," 2025. Accessed: 2025-02-09.

[13] R. DiBona, "Gpt-5 api: Unreliable and slow compared to gpt-4.1," September 2025.
[14] OpenAI Developer Community, "Gpt-5 is very slow on the api," 2025.
[15] OpenAI Developer Community, "Gpt-5 is very slow compared to 4.1 (responses api)," 2025.
[16] S. Zheng, B. Adams, and A. E. Hassan, "On build hermeticity in bazel-based build systems," *IEEE Software*, 2024.
[17] L. Courtès, "Functional package management with guix," *arXiv preprint arXiv:1305.4584*, 2013.
[18] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022.
[19] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2021.
[20] J. Malka, S. Zacchiroli, and T. Zimmermann, "Reproducibility of build environments through space and time," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pp. 97–101, 2024.
[21] M. Fourné, D. Wermke, W. Enck, S. Fahl, and Y. Acar, "It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1527–1544, IEEE, 2023.
[22] M. Lins, R. Mayrhofer, M. Roland, D. Hofer, and M. Schwaighofer, "On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: Early learnings from xz," *arXiv preprint arXiv:2404.08987*, 2024.
[23] M. Schwaighofer, M. Roland, and R. Mayrhofer, "Extending cloud build systems to eliminate transitive trust," in *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pp. 45–55, 2023.
[24] A. Mokhov, N. Mitchell, and S. Peyton Jones, "Build systems à la carte," *Proc. ACM Program. Lang.*, vol. 2, jul 2018.
[25] S. Zheng, B. Adams, and A. E. Hassan, "Does using bazel help speed up continuous integration builds?," *Empirical Software Engineering*, vol. 29, no. 5, p. 110, 2024.
[26] L. Courtès, T. Sample, S. Zacchiroli, and S. Tournier, "Source code archiving to the rescue of reproducible deployment," in *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability*, ACM REP '24, (New York, NY, USA), p. 36–45, Association for Computing Machinery, 2024.
[27] M. Alfadel and S. McIntosh, "The classics never go out of style: An empirical study of downgrades from the bazel build technology," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–12, 2024.
[28] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 339–349, IEEE, 2019.
[29] A. Silva, N. Saavedra, and M. Monperrus, "Gitbug-java: A reproducible benchmark of recent java bugs," in *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 118–122, 2024.
[30] A. Eliseeva, A. Kovrigin, I. Kholkin, E. Bogomolov, and Y. Zharov, "Envbench: A benchmark for automated environment setup," in *ICLR 2025 Third Workshop on Deep Learning for Code*.
[31] I. Bouzenia and M. Pradel, "You name it, I run it: An LLM agent to execute tests of arbitrary projects," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1054–1076, 2025.