The Unresolved Clash: Systemic Issues in Golang's Dual Dependency Management Regimes

WEIJIE SUN, State Key Lab for Novel Software Technology and School of Computer Science, Nanjing University, China
YING WANG, Software College, Northeastern University, China
HUIYAN WANG*, State Key Lab for Novel Software Technology and Software Institute, Nanjing University, China
CHANG XU*, State Key Lab for Novel Software Technology and School of Computer Science, Nanjing University, China
SHING-CHI CHEUNG, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, China

The Go programming language (Golang) has gained widespread popularity since its 2009 release. As Golang evolves, it recommends a new library-referencing mode (Go Modules), which, despite offering advanced features over its original mode (GOPATH), also introduces incompatible dependency situations. Moreover, the evolving features and complex rules of Go Modules are challenging for developers to follow accurately. This coexistence of library-referencing modes, coupled with Go Modules' ongoing evolution, leads to widespread dependency management (DM) issues, incurring reference inconsistencies and even build failures. To address this, we conducted an empirical study characterizing DM issues and developed Hero, an automated tool for detecting DM issues stemming from Golang mode migration and recommending appropriate fixing solutions. Recognizing the rapid growth of the Golang's new mode and its accompanying new DM issues, we further enhanced it to Hero$^+$ with advanced DM issue diagnosis capabilities. Evaluated on 188 benchmark DM issues, Hero and Hero$^+$ achieved high detection rates of 71.3% and 97.3%. Applied to 19,000 popular Golang projects, Hero$^+$ uncovered 13,408 potential DM issues. We reported 449 issues, among which 310 (69.0%) issues have been promptly confirmed, and 297 of them (95.8%) have been fixed or are being addressed, with fixes largely implementing our recommendations.

CCS Concepts: • **Software and its engineering → Software evolution**; **Maintaining software**.

Additional Key Words and Phrases: Golang ecosystem, dependency management

*Corresponding authors.

Authors' Contact Information: Weijie Sun, State Key Lab for Novel Software Technology and School of Computer Science, Nanjing University, Nanjing, Jiangsu, China, sunweijie@smail.nju.edu.cn; Ying Wang, Software College, Northeastern University, Shenyang, Liaoning, China, wangying@swc.neu.edu.cn; Huiyan Wang, State Key Lab for Novel Software Technology and Software Institute, Nanjing University, Nanjing, Jiangsu, China, why@nju.edu.cn; Chang Xu, State Key Lab for Novel Software Technology and School of Computer Science, Nanjing University, Nanjing, Jiangsu, China, changxu@nju.edu.cn; Shing-Chi Cheung, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China, scc@cse.ust.hk.

ACM 1557-7392/2026/1-ART
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Weijie Sun, Ying Wang, Huiyan Wang, Chang Xu, and Shing-Chi Cheung. 2026. The Unresolved Clash: Systemic Issues in Golang's Dual Dependency Management Regimes. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 44 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

> ***... (Golang will) preserve basic GOPATH mode indefinitely ... GOPATH mode remains the only way to work on dependency-free legacy packages that existing module code may still depend on.***
>
> - `Russ Cox` (the tech lead of Golang),
> in *Issue #60915 [136] in Golang* (2023)

Ever since the Go programming language (Golang) was released in 2009, it has gained increasing popularity among software developers [174]. Like other modern programming languages, Golang improves software development efficiency by enabling projects to import and reuse functionalities from existing Golang projects (i.e., libraries) by specifying an *import path* [53]. There are popular sites that host modern Golang projects, with the most notable being GitHub [38], Bitbucket [4], Launchpad [14], and IBM DevOps Services [67]. Among them, GitHub hosts nearly 90% of all Golang projects [49].

Although the library-based development approach of Golang has significantly contributed to its widespread adoption, its *library-referencing mode* has undergone substantial changes as the language evolves. Before the release of Golang 1.11, the library-referencing was supported by the `GOPATH` mode. This mode does not require developers to provide any configuration file, therefore lacking the mechanism to specify dependency versions. To address this limitation, developers turned to third-party tools such as `Dep` [47] and `Glide` [97] to manage versioned libraries under a *Vendor directory*[1]. Recognizing the need for a native solution, Golang 1.11 introduced the `Go Modules` mode in August 2018 operating alongside the existing `GOPATH` mode. `Go Modules` utilizes a `go.mod` configuration file in the project root, enabling a Go *module* to reference multiple versions of a library through distinct import paths. This file explicitly specifies the module's dependencies and their versions, defines a unique *module path* for identification, and must adhere to *Semantic Import Versioning* (`SIV`) rules [50]. For example, modules with major version "*/v2*" or higher must append a version suffix (e.g., "*/v2*") to their paths. Ever since then, the two distinct modes `GOPATH` and `Go Modules` have been in use together.

With the launch of `Go Modules`, the initial expectation from the Golang community was clear: the legacy `GOPATH` mode would be phased out, and a swift, unified transition should follow. However, this anticipated future failed to materialize. Nearly seven years later, the Golang ecosystem still finds itself in a state of persistent, unintended coexistence of the two library-referencing modes. As of June 2025, nearly 80% of projects still depend on at least one project that uses a different library-referencing mode. This reality signaled the practical infeasibility of a systemic phase-out, prompting a landmark shift in Golang's policy towards `GOPATH` and a commitment to preserving basic `GOPATH` mode indefinitely [136]. Consequently, the dual-mode paradigm has become an enduring feature of the ecosystem. This persistence can be attributed to two fundamental factors. First, the cost and complexity of migrating from `GOPATH` creates a significant barrier. This is exemplified by `Kubernetes`' multi-year migration efforts and the continued `GOPATH` reliance in tightly coupled ecosystems like Docker [29, 63]. Second, `GOPATH` remains essential for building legacy libraries that

[1]The *Vendor* attribute allows a Golang project to reference a library's different versions and keep them in different folders under a vendor directory.

even fully-migrated `Go Modules` projects require. For instance, `Kubernetes` itself, after extensive migration, still depends on `GOPATH` for approximately 10% of its direct dependencies.

Adopting `Go Modules`, however, has not provided a universal solution, as the system itself introduces inherent complexity. Even for projects that have fully migrated, `Go Modules'` complex and slowly evolving features still create a new set of DM pitfalls. Our analysis of popular Golang projects in 2025 reveals that a staggering 36.6% do not fully comply with notable rules associated with `Go Modules'` key features (detailed in Section 2), e.g., `SIV` importing, `replace` directive, `multi-module` and `GOPROXY`.

Consequently, the chaotic coexistence of the two library-referencing modes, compounded by the inherent complexities of `Go Modules`, has resulted in a wide spectrum of DM issues across the Golang ecosystem. Specifically, we made the following four observations:

- *`Go Modules` is not backward compatible with `GOPATH`.* There are two typical scenarios. First, a Golang project can be referenced by its downstream projects. After it migrates to `Go Modules`, its introduced virtual import paths (with version suffixes) cannot be recognized by its downstream projects still in `GOPATH`, leading to build failures in these projects. Second, a downstream project that has migrated to `Go Modules` may fail to locate its referenced libraries in `GOPATH` or may fetch unintended library versions due to different import path interpretations between the two modes. Lacking a mechanism in `GOPATH` to declare a specific version for an import path, projects in `Go Modules` are forced to infer one—typically the highest version tag from the repository—potentially causing incompatibilities.
- *During the evolution of `Go Modules`, its introduction of numerous new features alongside corresponding rules can be challenging for developers to accurately comprehend.* For example, prior to Golang 1.13, the projects using `Go Modules` downloaded their dependencies directly from hosting sites like GitHub by default. With the release of Golang 1.13, `GOPROXY` was introduced, enabling the go command to download and authenticate modules through the official Golang infrastructure by default (e.g., the Go module mirror [51] and the Go checksum database [49]). Although this feature enhanced the security of the Golang ecosystem, it also required projects to follow the rule that released versions must remain unchanged to ensure the consistency between the host site and the checksum database.
- *DM issues can occur either when upstream and downstream projects suffer from inconsistent library-referencing modes, or when certain rules of `Go Modules` are violated even if a Golang project and its referenced upstream projects both use `Go Modules`.* Such chaos caused by the evolving library-referencing modes is unique to the Golang ecosystem, unlike existing dependency conflict and dependency bloat issues in Java [11, 33, 65, 134, 148–150, 167–170], JavaScript [125, 159], and Python [161, 166].
- *Resolving DM issues for a Golang project requires up-to-date knowledge of its upstream and downstream projects and their possible heterogeneous uses of two library-referencing modes.* However, the Golang ecosystem does not provide sufficient information to help developers assess the impact of solutions on other projects. Addressing a DM issue locally, without considering the ecosystem holistically, can easily introduce new unintended issues for downstream projects.

Figure 1(a) highlights the intractable nature of these DM issues through a long-standing conflict in `moby/moby` [110] (the open-source core of `Docker`), which remains in `GOPATH` mode. First documented in 2021 (e.g., issue #42939 [108]), this problem has persistently caused build failures, with new comments from downstream `Go Modules` users emerging as recently as 2024. The conflict arises because downstream `Go Modules` projects disregard `Moby`'s `GOPATH`-style *vendor* directory, opting instead to select a different version of its direct dependency, `docker/swarmkit`. Since the newly selected `swarmkit` is also a `GOPATH` project, the resolution process then proceeds to select

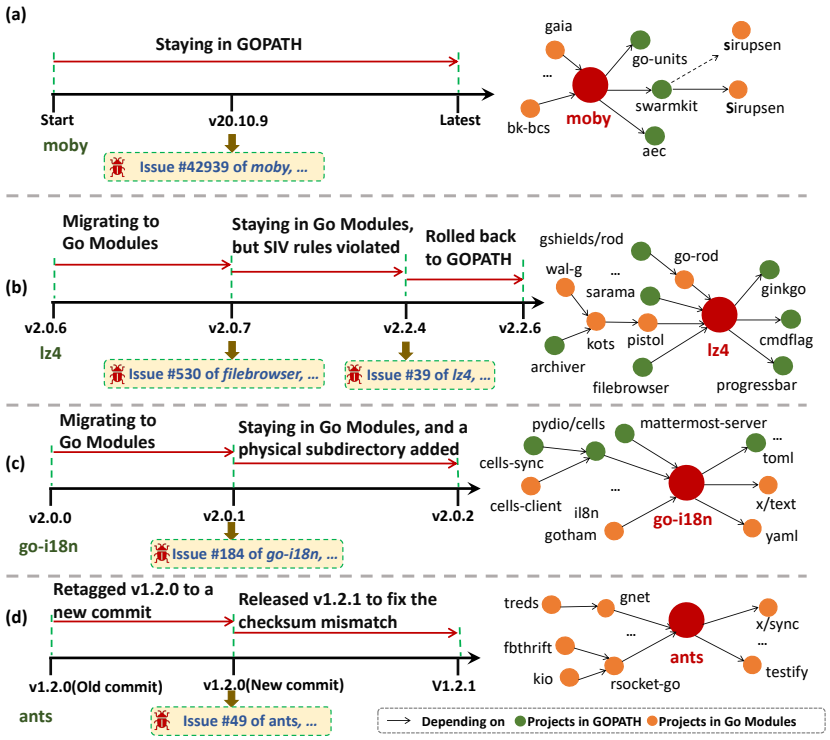

Fig. 1. DM issue examples

a modern, module-aware version of its dependency, `sirupsen/logrus`. However, the canonical import path of the new `logrus` module (*github.com/sirupsen/logrus*) subsequently clashes with the hardcoded, differently-cased path (*github.com/Sirupsen/logrus*) referenced within `swarmkit`'s source code. This path mismatch results in a build failure that is notoriously difficult to resolve, as the fault lies not with `moby` itself—which operates correctly within its `GOPATH` paradigm—but with the underlying architectural conflict that emerges when it is consumed by a `Go Modules` project.

Figure 1(b) illustrates an example DM issue from Project `lz4` [130] which migrated to `Go Modules` in version v2.0.7. In accordance with the `SIV` rules specified by `Go Modules`, the project declared module path *github.com/pierrec/lz4/v2* in its `go.mod` file with version suffix "*/v2*". Although the project can be built successfully after the migration, it induced DM issues to its downstream projects still in `GOPATH`, since the latter cannot recognize this version suffix in module paths (e.g., issue #530 of `filebrowser` [34]). To resolve the problem, `lz4` released version v2.2.4, which remained in `Go Modules` but removed version suffix "*/v2*" from its module path as a workaround. This resolved the DM issues in its downstream projects still in `GOPATH`, but induced build failures into its downstream projects that had already migrated to `Go Modules`, since this solution violated `SIV` rules (e.g., issue #39 of `lz4` [128]). Due to the lack of accurate estimation of the migration impact on downstream projects, `lz4` chose to roll back to `GOPATH` in v2.2.6 and suspended its migration until the downstream projects had completed their migrations. Such issues are common in Golang projects, imposing unforeseeable risks during the mode migration.

Figure 1(c) presents another example from `go-i18n` [114]. Its version v2.0.1 adopted `Go Modules` to provide finer control over library referencing, following the approach of its upstream projects.

However, such change induced at least five DM issues to its downstream projects in GOPATH (e.g., issue #184 [115]) due to their inability to interpret version suffix "*/v2*" in go-i18n's module path. To address the problem, go-i18n v2.0.2's repository provided an additional subdirectory *go-i18n/v2* containing a copy of the implementations to support downstream projects in GOPATH. This solution is suboptimal, as it transforms the virtual path in Go Modules into a physical one, necessitating extra maintenance with each project release. In fact, without a holistic understanding of all project dependencies and the interactions between their mixed library-referencing modes, developers often find it difficult to identify an appropriate solution that indeed resolves DM issues without affecting downstream projects.

Figure 1(d) illustrates the final example from the ants project [124], where its original v1.2.0 commit was replaced with a new one. Prior to Golang 1.13, such practices had no significant impact. However, with the introduction of GOPROXY in Golang 1.13, all downloaded Go Modules projects became subjects to checksum verification against entries stored in the checksum database, thereby enforcing the immutability of released versions. In this case, the checksum database had cached the original v1.2.0 commit, while GitHub pointed to the updated commit. This inconsistency resulted in DM issues for downstream projects, as checksum mismatch errors occurred when ants was downloaded from GitHub (e.g., issue #49 [123]). The problem persisted for over a year until version 1.2.1 was released, at which point the GOPROXY rule was finally recognized by ants developers. This example highlights how DM issues can arise unintentionally due to the evolving features of Go Modules.

To deal with such chaotic DM issues caused by the evolving library-referencing modes to the Golang ecosystem, we studied 20,000 Golang projects on GitHub, which disclosed the severity of DM issues caused by either the coexistence of the two modes or evolving nature of Go Modules. To better dig into the problem, we further sampled 500 projects from the top 1,000 ones, and collected 256 DM issues from the issue trackers for a deeper study of their characteristics and solutions. Altogether, we identified three DM issue patterns and summarized thirteen fixing solutions commonly adopted by developers. Leveraging these findings, we developed an automated tool, Hero (HEalth diagnosis tool foR the gOlang ecosystem), to detect DM issues caused by the mode migration. By taking into account the evolving features of Go Modules, we further identified the prevalence of DM issues arising from common rule violations in projects utilizing Go Modules. These results motivated the development of tool Hero$^+$, extended over Hero, that integrates the features of Go Modules. Notably, Hero and Hero$^+$ can also offer customized fixing suggestions to Golang developers, making explicit the potential benefits and consequences of the fixings for the Golang ecosystem.

To evaluate the effectiveness of Hero and Hero$^+$, we collected 188 real-world DM issues from the top 1,000 Golang projects not included in our empirical study and conducted experiments using these issues as a benchmark. Experiments demonstrated that Hero achieved a detection rate of 71.3%, while Hero$^+$ improved this rate to 97.3%. This improvement is largely attributed to Hero$^+$'s extended dependency model, which enables a more comprehensive analysis of Go Modules' features. Leveraging its capability, we further applied Hero$^+$ to the remaining 19,000 projects collected from GitHub and detected 13,408 potential DM issues. Altogether, we submitted 449 issues associated with the $1{,}001^{st}$ to $2{,}000^{th}$ most popular projects, and suggested corresponding fixing solutions. Encouragingly, 310 issues (69.0%) were quickly confirmed by the developers, among which 297 (95.8%) have been fixed or are in the process of being fixed using our suggested solutions. These fixes are expected to cause minimal or acceptable impacts on other projects in the Golang ecosystem. The confirmed issues include well-known projects, such as github/hub[39] and microsoft/presidio[103], and have also facilitated the migration of 29 projects to Go Modules.

In summary, we make the following contributions:

- To the best of our knowledge, we conducted the first empirical study involving 20,000 Golang projects to exhaustively investigate their migration status regarding library-referencing modes and violations of `Go Modules'` rules, revealing a persistent and likely permanent coexistence of both library-referencing modes nearly seven years after the introduction of `Go Modules`. Furthermore, we analyzed 256 real DM issues, providing valuable insights into the evolving landscape of the dependency management in Golang projects.
- We developed the Hero tool[2] to diagnose DM issues for the Golang ecosystem and extended it to Hero$^+$ to accommodate the evolving features introduced by `Go Modules`. Both tools can effectively detect DM issues and provide customized fixing suggestions.
- We provided a reproduction package on Hero$^+$ website for future research, which includes: (1) detailed information of the 20k subjects and 256 DM issues studied in our empirical study; (2) our benchmark dataset (188 DM issues and subjects used for evaluation).

We note that this work builds on our preliminary work [164] with these key extensions:

- We enhanced our empirical investigation to reflect the latest state of Golang's dependency management. Our updated analysis of migration dynamics (Section 3.2) quantifies both the rapid adoption of `Go Modules` and the persistent coexistence of both `Go Modules` and `GOPATH`. This observation can help contextualize Golang's recent policy shift regarding `GOPATH`. We also introduced a new research question (RQ2) examining compliance with `Go Modules`' evolving features (Section 3.3).
- We expanded our empirical study dataset to 256 DM issues and identified three new types of DM issues (Types C.2–C.4) along with five corresponding fixing solutions (Solutions 9–13), highlighting the prevalence of new DM issues arising from violations of `Go Modules'` rules (Sections 3.4 and 3.5).
- We extended the original tool Hero to develop Hero$^+$, which further models the features and infrastructure of `Go Modules` to better adapt to its evolving nature in the DM issue diagnosis (Section 4).
- We evaluated Hero$^+$ in our expanded benchmark with 188 real DM issues and observed that Hero$^+$ outperformed Hero significantly (25% improvement on the detection rate) (RQ5 in Section 5). Additionally, with the assistance of Hero$^+$, we submitted 169 new DM issues to popular Golang projects (RQ6 in Section 5).

The remainder of this article is organized as follows. Section 2 reviews Go's dependency management evolution, detailing the advanced features of `Go Modules` and the concept of module-awareness. Section 3 conducts an empirical study to explore the scale and characteristics of DM issues. Section 4 proposes Hero and Hero$^+$ techniques to diagnose DM issues for the Golang ecosystem. Section 5 evaluates Hero and Hero$^+$ experimentally with real-world projects. Sections 6 and 7 discuss the threats to validity and the related work. Finally, Section 8 concludes the article and discusses the future work.

2 Background

To establish a foundation for the subsequent discussion, this section first introduces the basic dependency management mechanisms in Go, tracing the transition from `GOPATH` to `Go Modules` and highlighting their core operational differences. Building upon this transition, the section then examines the advanced features and rules introduced by `Go Modules`, as their complexity is a primary source of contemporary dependency management issues. To fully analyze these issues, the

[2]http://www.hero-go.com/

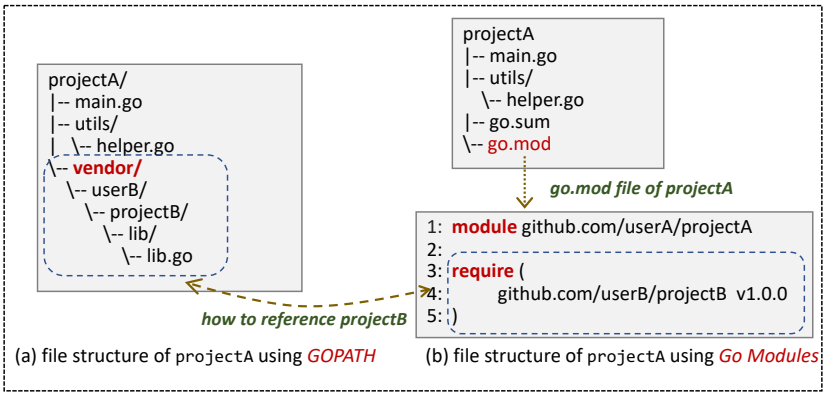

Fig. 2. Comparison between dependency management modes in Golang

section finally introduces module-awareness, a technical concept crucial for understanding how the Golang toolchain operates and why incompatibilities may arise from the interaction between these two modes.

2.1 Dependency Management in Golang

The evolution of dependency management in Golang reflects the language's continuous effort to enhance modularity, reproducibility, and interoperability within its ecosystem. Before Go 1.11, dependency management relied on the `GOPATH` mode (Figure 2(a)), which lacked a native mechanism for specifying dependency versions. To address this, developers commonly used a local *vendor* directory to store dependencies. For example, `projectA` could include `projectB` as a dependency by copying `projectB`'s source files (e.g., *lib/lib.go*) into its vendor directory (*vendoruserB/projectB/*). The Go compiler automatically prioritized dependencies in *vendor*, allowing developers to lock a project to specific library versions and prevent unexpected changes from upstream repositories. To facilitate this process, third-party tools such as `Dep` and `Glide` were introduced. These tools used configuration files (e.g., `Gopkg.toml`) to automate dependency fetching and management of the vendored libraries, but they remained external to the official Go toolchain.

To address these limitations, Go 1.11 introduced `Go Modules`, marking a transition from vendor-based to module-based dependency management (Figure 2(b)). In this mode, each project defines its own *module* through a `go.mod` file located at the root of the repository. This configuration file explicitly lists the module's own path and, in the `require` section, all its dependencies and their specific versions. For instance, `projectA` specifies its module path (*github.com/userA/projectA*) and declares its dependency on `projectB` (*github.com/userB/projectB v1.0.0*). The accompanying `go.sum` file records cryptographic checksums of downloaded modules to guarantee deterministic and reproducible builds. This design eliminates the need for manually maintained *vendor* directories and external tools by automatically resolving and versioning dependencies, providing a consistent, toolchain-integrated approach to dependency management in Golang.

2.2 Features and Rules of `Go Modules`

Go Modules introduces several key features and corresponding rules to provide a more robust and scalable dependency management system. In this paper, we focus our analysis on four of these core features: `SIV`, `replace`, `multi-module`, and `GOPROXY`. Our rationale for selecting these features

is empirically driven: as our investigation in Section 3.4 will demonstrate, violation of their rules is the primary source of modern DM issues within the Golang ecosystem.

2.2.1 Rules of SIV. `Go Modules` introduces `SIV` to support dependency management of multiple project versions. It has three related rules:

(1) Golang projects should follow a semantic versioning format (Semver)[3]. Figure 3(a) gives an example, where `projectA` tags a release with a semantic version of v2.7.0 on GitHub.
(2) The module path declared in the `go.mod` file must correspond to the major version. If a project's major version is *v0* or *v1*, its version suffix *should not* be included in its module path. Conversely, when a project's major version is *v2* or above (denoted as *v2+*), a version suffix like "*/v2*" must be included at the end of its module path. As shown in Figure 3(b), `projectA` v2.7.0's module path is "*github.com/user/projectA/v2*". To reference it, downstream projects must declare this path and import it in *require directive* attributes of the `go.mod` file, as well as in *import directive* attributes of their `.go` source files. Figures 3(c) and (d) give two examples.
(3) The module path declared in the `go.mod` file defines the module's canonical name and its prefix should correspond to the root directory of its version control repository (e.g., "*github.com/user/projectA*").

These SIV rules within `Go Modules` enable different major versions of a single library to be referenced separately through distinct paths.

For enhanced flexibility, the official Golang documentation [50] suggests two strategies for releasing v2+ projects: the *major branch* strategy and the *major subdirectory* strategy. The former involves updating a project's module and import paths to include a version suffix like */v2*". This update does not necessitate the physical creation of a new branch with the version suffix in the version control system. The latter strategy requires creating a physical subdirectory (e.g., *projectA/v2*) containing the source code and a corresponding `go.mod` file. Consequently, the module path must also end with the version suffix */v2*". Therefore, module and import paths are virtual in the major branch strategy but physical in the major subdirectory strategy. The major subdirectory approach is sometimes employed to facilitate a transition for downstream projects that use `GOPATH`, as illustrated in Figure 1(c).

Violations of these rules disrupt the contract relied upon by the Golang toolchain. The toolchain depends on a strict correspondence between a module's declared path and its version to locate and fetch the correct source code. An incorrect mapping may prevent the Go toolchain from finding the module, leading to build failures for dependent downstream projects.

2.2.2 Rule of `replace` directive. The `go.mod` file in `Go Modules` features a `replace` directive that enables developers to override module versions or redirect dependencies without altering the import paths in the source code. For instance, Figure 4(a) shows `projectA` declaring a dependency on `userB/projectB` while using the `replace` directive to substitute it with `userC/projectC`. As illustrated in Figure 4(b), the package *"github.com/userB/projectB/pkg"* is imported into `projectA`. Normally, this import would resolve to the package within `userB/projectB`. However, due to the `replace` directive in `projectA`'s `go.mod` file, the Golang toolchain instead resolves the import to the corresponding package at *"github.com/userC/projectC/pkg"*.

The primary rule for using the `replace` directive is that its use should be confined to the local development environment and should not be present in published release versions. This is because the scope of a `replace` directive is restricted to the specific module where it is defined and is not

[3]The Semver format is `MAJOR.MINOR.PATCH`, where `MAJOR`, `MINOR`, and `PATCH` denote incompatible API changes, backward compatible API changes, and backward compatible bug fixes, respectively (https://semver.org/).

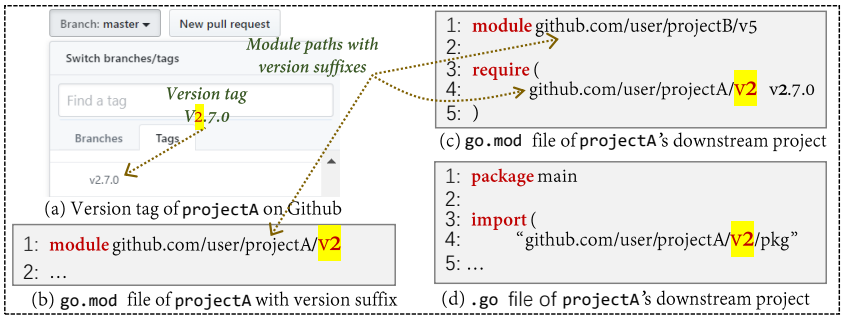

Fig. 3. SIV rules in the Go Modules mode

inherited by downstream projects. Failing to follow this rule can cause build failures for any project that depends on it. As illustrated in Figure 4, a downstream project that imports `projectA` will ignore the `replace` directive and will instead fetch the original dependency, `userB/projectB`. However, `projectA` source code contains an import for `github.com/userB/projectB/pkg`, which its author intended to be resolved from the replacement module, `userC/projectC`. If the required `pkg` directory does not exist within the original `userB/projectB` module, the downstream Golang toolchain will be unable to locate the imported package, leading to a build failure. This forces downstream maintainers to diagnose the upstream issue and manually replicate the `replace` directive, creating a fragile and unscalable workaround.

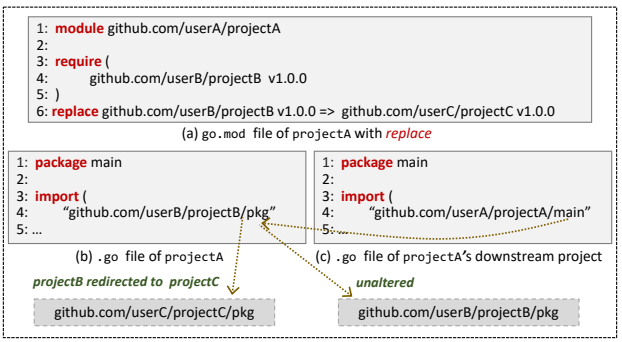

Fig. 4. `replace` directive in the Go Modules mode

2.2.3 Rule of `multi-module` project. The Go Modules multi-module feature allows a repository to contain multiple modules, each defined by its own `go.mod` file. This design enables independent dependency management and versioning, which is essential for maintaining a clean and scalable architecture in large projects.

However, managing multiple modules within the same repository requires careful coordination. A specific rule governs this structure: when a submodule is created by extracting a package from the root module, the submodule should depend on a version of the root module released after the split. As illustrated in Figure 5, consider a project *github.com/example/root* at version `v1.0.0`, which includes the package *github.com/example/root/api*. When maintainers extract this package into a new module *github.com/example/root/api*, the subsequent version of the root module (e.g.,

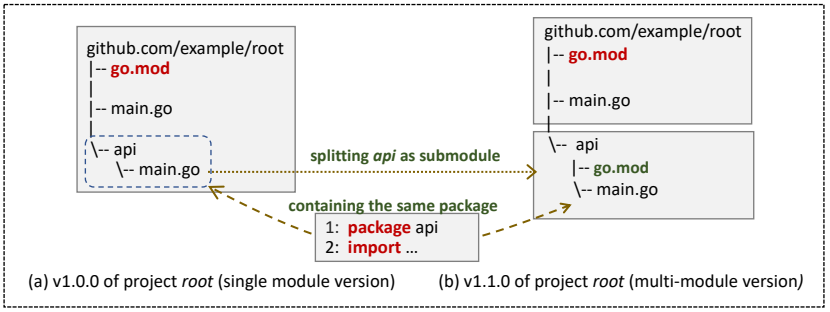

Fig. 5. A project's transition from a single-module to a multi-module architecture.

`v1.1.0`) no longer contains the *api* directory. The new submodule *github.com/example/root/api* must therefore declare a dependency on *github.com/example/root v1.1.0*.

Failing to do so can cause build errors when downstream projects depend simultaneously on the submodule (*github.com/example/root/api@v1.0.0*) and on an older version of the root module (*github.com/example/root@v1.0.0*) that still includes the same package. This duplication leads to import path conflicts and consequently build failures.

2.2.4 Rule of GOPROXY. `Go Modules` introduces `GOPROXY` to enhance the reliability and efficiency of dependency retrieval. `GOPROXY` acts as an intermediary server that caches module versions, enabling faster downloads and reducing issues caused by unavailable or slow upstream repositories. Since Go 1.13, `GOPROXY` has been enabled by default, with the Go module mirror [51] serving as the official proxy. When the Go module mirror caches modules, checksums are computed and stored in the official Go checksum database [49]. Each time a project downloads a module, the Go toolchain recalculates its checksum and verifies it against the stored value in the database, thereby ensuring security and consistency.

A key principle enforced by `GOPROXY` is the immutability of released versions. Once a version is published and cached, it must remain unchanged. This rule arises from `GOPROXY`'s caching mechanism, and violating it leads to build failures for downstream consumers. As illustrated in Figure 6, consider a project that publishes version `v1.0.0`, which `GOPROXY` then fetches and caches along with its checksum. If the developer later attempts to fix a bug by moving the `v1.0.0` tag to a new commit, `GOPROXY` will ignore this change and retain the original content for that version. When a downstream project is built, the Go toolchain will fetch the checksum for `v1.0.0` from the checksum database but may download the module's source code from the version control system where the tag was modified. Because the content of the new commit is different from the original, the calculated checksum of the downloaded code will not match the one stored in the database. This results in a build failure with a `checksum mismatch` error, effectively preventing the unexpected code from being used. Therefore, the correct practice is to release a new version (e.g., `v1.0.1`) rather than modifying an existing one.

According to the above four key features of `Go Modules` and their corresponding rules, we formally define rule violation occurring in `Go Modules` projects as follows:

Definition 1 (Rule violation): A *rule violation* occurs when a project utilizes the features of `Go Modules` but fails to comply with the corresponding rules.

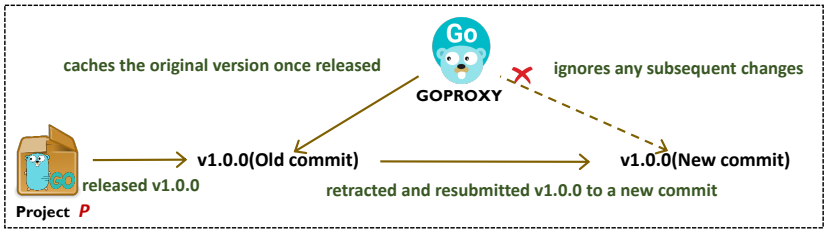

Fig. 6. Mechanism of GOPROXY caching.

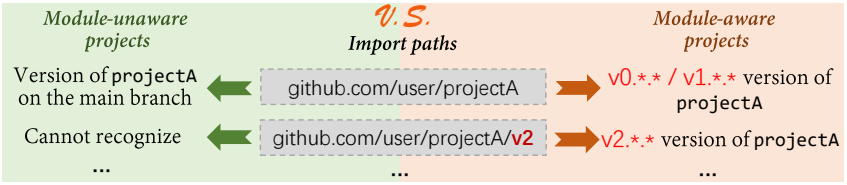

Fig. 7. Comparison of module-aware and module-unaware projects

2.3 Module-Awareness in Different Golang Versions

To ease discussion, we refer to the capability of recognizing a virtual path ended with a version suffix like "*/v2*" as *module-awareness*. This capability is important for referencing libraries in the Golang ecosystem.

As the migration from GOPATH to Go Modules has immense impact on many Golang projects, it was gradually achieved by multiple Golang versions over two years. During migration, "minimal module compatibility" was adopted since Golang 1.9.7 in the series 1.9.* and Golang 1.10.3 in the series 1.10.*, which added module-awareness to projects that had not migrated to Go Modules [48]. As such, we refer to the versions in range [1.0.1, 1.9.7) ∪ [1.10.1, 1.10.3), which manage dependencies in GOPATH without module-awareness, as *legacy Golang versions*. We refer to those in range [1.9.7, 1.10.1) ∪ [1.10.3, 1.11.1), which manage dependencies in GOPATH with module-awareness, as *compatible Golang versions*. We refer to those of 1.11.1 or above, which allow projects to adopt either GOPATH or Go Modules and support module-awareness, as *new Golang versions*. We observe that Golang projects in GOPATH often use third-party tools (e.g., Dep [47], Glide [97], etc.) to help manage dependencies. Since none of the tools supports "minimal module compatibility", their uses actually block module-awareness, messing up library-referencing (e.g., issue #878 of olivere [117] about using Dep and glide, and #103 of migrate [55] about using govendor).

Table 1 summarizes module-awareness in different Golang versions. Based on this, we give two definitions below:

Definition 2 (Module-aware project): A project is *module-aware* if and only if it uses a compatible or new Golang version and does not use any DM tool.

Definition 3 (Module-unaware project): A project is *module-unaware* if and only if it uses a legacy Golang version, or it uses a compatible or new Golang version with a DM tool.

Figure 7 shows how module-aware and module-unaware projects differ in parsing an import path with or without a *v2+* version suffix. For an import path like *github.com/user/projectA*, a module-aware project could reference a specific version *v0.*.** or *v1.*.** of projectA under *v2* (latest version under *v2*, by default), while a module-unaware project would reference the version on projectA's main branch (typically the latest version). For an import path like *github.com/user/projectA/v2*, the

Table 1. Module awareness in different Golang versions

Category	Version range	DM mode	Using DM tools	Module awareness
Legacy Golang versions	[1.0.1, 1.9.7) ∪ [1.10.1, 1.10.3)	GOPATH	Y	N
			N	
Compatible Golang versions	[1.9.7, 1.10.1) ∪ [1.10.3, 1.11.1)	GOPATH	Y	N
			N	Y
New Golang versions	≥ 1.11.1	GOPATH	Y	N
			N	Y
		Go Modules	–	Y

DM stands for dependency management. "–" means "not applicable".

former could reference a specific version *v2.*∗.∗ of `projectA` (latest version under *v3*, by default), while the latter would fail to recognize it.

According to the above background knowledge, we formally define the DM issues occurring in Golang projects as follows:

Definition 4 (Dependency management (DM) issue): If an issue is caused by the different interpretations between module-aware and module-unaware projects or rule violations within `Go Modules` projects, we refer to it as a *DM issue* in Golang ecosystem.

A project that suffers from a DM issue may fetch the unintended versions of its libraries, or may not find its referenced libraries.

3 Empirical Study

To better understand the scale and characteristics of DM issues within the Golang ecosystem, we conducted a large-scale empirical study. Specifically, we aim to answer the following four research questions:

- **RQ1 (State of Dual Library-Referencing Modes)**: *What is the current state of dual library-referencing modes (`Go Modules` v.s. `GOPATH`) and what are the primary factors contributing to it?*
- **RQ2 (Prevalence of Rule Violations in `Go Modules`)**: *Do rule violations commonly occur in projects using `Go Modules`?*
- **RQ3 (Issue Types and Causes)**: *What are common types of DM issues? What are their root causes?*
- **RQ4 (Fixing Solutions)**: *What are common practices for fixing DM issues? How do they affect the ecosystem?*

To answer RQ1/2, we collected top 20,000 popular and active open-source Golang projects from GitHub to examine their migration status and usage of `Go Modules'` features. To answer RQ3/4, we randomly selected 500 subjects (denoted as $subjectSet_1$) from top 1,000 of our collected projects. We then collected real DM issues from these projects plus some additional ones. To dig into these issues, we manually analyzed their issue descriptions, developers' discussions, code commits, and the Golang official documentation. Note that the remaining 500 projects (denoted as $subjectSet_2$) in top 1,000 of our collected projects were not used in RQ3/4. They are used to evaluate our DM issue detection techniques later in RQ5 (Sec 5.1). To better study and analyze trends in migration status and the usage of `Go Modules'` features over time, we applied the outlined procedure to collect data from 2020 as well as the most recent data from 2025. Below we present our data collection procedure and study results in detail.

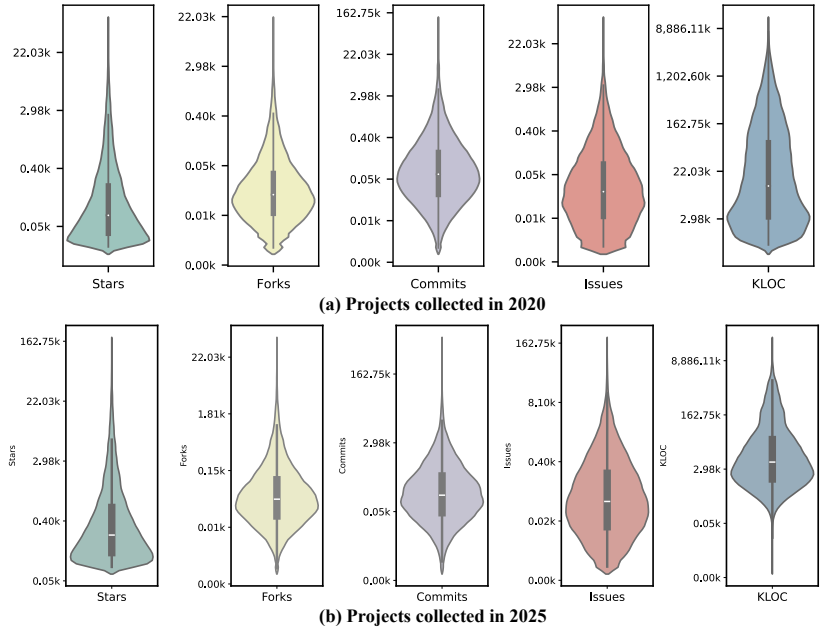

Fig. 8. Statistics of collected 20,000 Golang projects in 2020 and 2025 (log scale)

3.1 Data Collection

Step 1: Collecting Golang projects. We collected the top 20,000 most popular Golang projects from GitHub, which hosts over 90% of Golang repositories. The *popularity* of a project was determined by its number of stars.

We repeated the aforementioned procedure in 2020 and 2025. The projects collected in these two years exhibit only a 51.0% overlap, ensuring the diversity of our dataset. Figure 8 presents the demographics of the projects collected during these two periods. They are: (1) popular (60.3% and 91.3% having 100+ stars or forks in 2020 and 2025 separately), (2) well-maintained (on average having 339 code commits and 136 issues in 2020 and 758 code commits and 458 issues in 2025), and (3) large-sized (on average having 72.3 KLOC in 2020 and 107.6 KLOC in 2025). We used these projects for RQ1/2.

Step 2: Collecting DM issues. For the 500 projects in $subjectSet_1$, after filtering the ones that have no issue trackers or code repositories, we considered the remaining projects as subjects. We then added to the seed subjects Golang's official project `golang/go` [52] and two most popular dependency management tools `Dep` [47] and `Glide` [97], for better studying DM issues from the perspective of the ecosystem. In total, we obtained 487 projects in 2020 and 498 projects in 2025 for RQ3/4.

As these projects contain many issue reports, we filtered using keywords "go modules" and "go.mod" (case insensitive) to locate potential DM issues for manual analysis ("go.mod" configuration file is a notable new feature in the `Go Modules` mode). Keyword "go modules" returned 1,342 and 10,702 issue reports in 2020 and 2025, and "go.mod" returned 2,421 and 13,363 ones, respectively. After merging overlapping reports, we removed noise through a two-step process. First, we excluded issue reports that did not discuss DM issues (e.g., issue #5559 [46] of project `gogs` [45] only

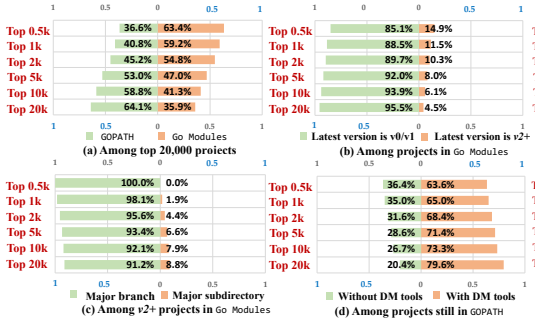

Fig. 9. Investigation statistics for RQ1 in 2020

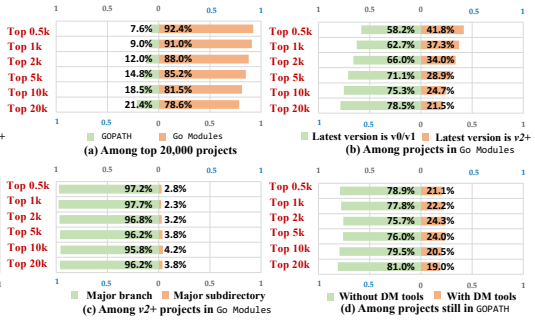

Fig. 10. Investigation statistics for RQ1 in 2025

documented developers' plan to migrate to `Go Modules`). Second, we excluded issue reports that discussed nothing about root causes of DM issues.

Three co-authors cross-checked all collected issue reports and finalized a collection of 151 well-documented DM issues in 2020 and 179 in 2025, involving 127 and 129 distinct Golang projects respectively. These issues contain sufficient details for studying RQ3/4.

3.2 RQ1: State of Dual Library-Referencing Modes

To answer RQ1, we investigate the state of dual library-referencing modes from two distinct perspectives: the macro-level of the entire Golang ecosystem and the micro-level within the dependency graph of individual projects.

3.2.1 Ecosystem-Level Perspective. At the ecosystem level, we quantify the overall adoption rates of each dependency mode across our 20,000-project dataset to understand the broader landscape.

- For all 20,000 projects, we counted the number of projects that have migrated to `Go Modules` by checking whether `go.mod` files exist in their latest versions' repositories.
- For projects that have migrated to `Go Modules`, we checked whether their major version numbers of latest releases are *v2+*. If so, we further checked their adopted strategies (i.e., major branch/subdirectory) in the code repositories.
- For projects still in `GOPATH`, we checked whether they use third-party tools to manage dependencies by the presence of their configuration files. For example, using the `Dep` [47] or `Glide` [97] tool requires a `Gopkg.toml` or `glide.yaml` configuration file, respectively.

Results. Figure 9 and Figure 10 show analysis results for subjects in 2020 and 2025. To see trends, we divided all projects into six (overlapping) groups based on their popularities: top 0.5k, 1k, 5k, 10k, and 20k (1k = 1,000).

Figures 9(a) and 10(a) reveal that the proportion of projects migrating to `Go Modules` increases with their popularity, indicating that this migration is considered a good practice within the ecosystem. A comparison of the two figures shows a significant rise in the adoption of `Go Modules` over the past five years, increasing from 35.9% in 2020 to 78.6% in 2025. This positive trend highlights the growing preference for `Go Modules` in the Golang ecosystem, reflecting a community-wide shift towards more robust and standardized dependency management. Nonetheless, 21.4% of the top 20,000 projects continued to use `GOPATH` in 2025, including prominent projects like `Moby` (the upstream project for Docker)[110], which boasts 70k stars.

Figures 9(b) and 10(b) show that by 2025, 21.5% of projects that migrated to `Go Modules` had released *v2+* versions, compared to just 4.5% in 2020. This indicates a positive trend, with an increasing number of projects achieving the maturity required for major version updates (*v2+*).

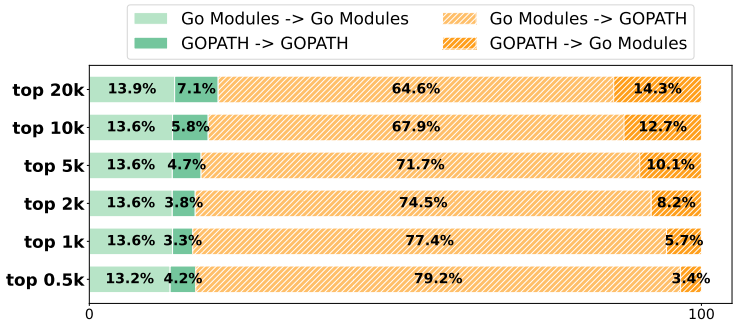

Fig. 11. Rate of module-awareness among dependencies

However, as depicted in Figures 9(c) and 10(c), only 8.8% in 2020 and 3.8% in 2025 of these *v2+* projects were managed using the major branch strategy. This finding suggests that the majority of *v2+* projects rely on virtual module paths with version suffixes, such as "*/v2*". Consequently, these projects are likely to induce build failures in downstream projects that are not module-aware. As the release of *v2+* versions becomes more widespread, the current 78.5% of projects still on *v0/v1* are likely to eventually follow suit and DM issues will easily occur.

Figure 10(d) shows that currently 19.0% of projects in `GOPATH` still use third-party tools to manage dependencies. This reliance on third-party tools hinders module awareness for projects that adopt compatible Golang versions. Figures 9(d) and 10(d) indicate a decrease in the proportion of `GOPATH` projects using third-party tools (from 79.6% to 19.0%), exhibiting a positive trend of reducing reliance on third-party tools within `GOPATH`. However, nearly one-quarter of these projects still depend on such tools in 2025, indicating the still existing lack of module awareness within the Golang ecosystem.

3.2.2 Project-Level Perspective. While our latest ecosystem-level data reveals broad adoption trends for `Go Modules`, this macro view does not fully capture the complexities of dependency management. Specifically, downstream projects may not always adopt the latest versions of their dependencies. In such cases, while projects continue to release `Go Modules` versions, downstream projects may still rely on older `GOPATH` versions. Additionally, some `GOPATH` projects remain functional and continue to be utilized by various projects. To further understand the status of mode coexistence more delicately, we analyze the library-referencing modes of dependencies from a project-level perspective across our latest 2025 dataset.

We check the library-referencing modes of the top 20,000 projects for the presence of `go.mod` files in the repositories of their latest versions. For `Go Modules` projects, we extract the dependencies declared in `go.mod` and analyze their library-referencing modes in relation to the declared versions. For `GOPATH` projects, we extract their dependencies and analyze the library-referencing modes based on the latest available versions. This approach enables us to assess whether certain projects depend on others that use different library-referencing modes, either directly or transitively.

Results. We categorize all the 20,000 projects into four categories according to their dependency modes: (1) `Go Modules` projects which depend solely on `Go Modules` project; (2) `Go Modules` projects which depend on at least one `GOPATH` project; (3) `GOPATH` projects which depend solely on `GOPATH` project; (4) `GOPATH` projects which depend on at least one `Go Modules` project.

As shown in Figure 11, 78.9% of the projects exhibit conflicting library-referencing modes with their dependencies. Specifically, 66.8% of `GOPATH` projects depend on at least one `Go Modules` project, either directly or transitively, while 82.3% of `Go Modules` projects rely on at least one

GOPATH project. Additionally, the prevalence of conflicting library-referencing modes increases with project popularity. For example, although the kubernetes/kubernetes project [86], which has received 116k stars and ranks third in popularity, has migrated to Go Modules, approximately 10% of its direct dependencies still use GOPATH. This demonstrates that the coexistence of the two library-referencing modes creates a complex DM status in the Golang ecosystem.

This widespread coexistence is a significant source of DM issues in the Golang ecosystem, arising from fundamental incompatibilities in how each mode resolves dependencies. The two systems operate on conflicting principles. Specifically, the module-unaware GOPATH projects lack a native versioning mechanism and cannot interpret the versioned module path suffixes (e.g., /v2) that are central to modern library design in Go Modules. Conversely, when a Go Modules project depends on a legacy GOPATH project, its toolchain fetches dependencies directly from source repositories, bypassing the curated versions stored within the GOPATH project's vendor directory, which leads to dependency resolution inconsistencies. The concrete impacts of these mechanistic incompatibilities, such as build failures caused by version mismatches and import path conflicts, are the core focus of our analysis in Section 3.4, where they are investigated as *Type A* and *Type B* issues.

3.2.3 Challenges and Motivations of Migration. The previous findings reveal a complex and evolving migration landscape. On one hand, the persistence of GOPATH nearly seven years after the introduction of Go Modules warrants an investigation into the significant barriers that hinder a complete transition. On the other hand, the clear acceleration in Go Modules adoption over the past five years calls for a complementary analysis of the drivers now sufficiently powerful to overcome this inertia. This section, therefore, presents a qualitative analysis of both the barriers to and drivers of migration, drawing on developer discussions to explain these dual trends.

Challenges of migration. We investigate how developers consider this problem from projects still in GOPATH. We focused on the GOPATH part of top 500 out of the 20,000 projects both in 2020 and 2025, and analyzed their issue reports that discuss migration to study reasons for holding the migration. We obtained 70 issue reports discussing unsuccessful migration, and observed five common reasons:

- ***Existing versioning scheme incompatible with SIV rules in Go Modules (28/70).*** Some projects have their own versioning schemes, different from SIV rules in Go Modules. To avoid incompatibility (e.g., issue #328 of go-tools [31]), developers chose to stay with GOPATH.
- ***Third-party DM tools hindering the migration plan (16/70).*** Some projects heavily rely on third-party tools for dependency management. As the tools do not work with Go Modules, developers chose to live with the tools instead of migration (e.g., issue #61 [44] of uuid).
- ***Causing problems to downstream projects in GOPATH (11/70).*** Many projects are still in GOPATH, inconvenient to reference upstream projects in Go Modules. For continuous support for downstream projects, developers chose to stay with GOPATH (e.g., issue #103 [55] of migrate).
- ***Reliance on the legacy toolchain (7/70).*** Some projects are deeply integrated with complex build systems, continuous integration pipelines, and other toolchains designed specifically for GOPATH. Migrating to Go Modules requires prohibitive efforts, thus deterring adoption (e.g., issue #46471 [111] of moby).
- ***Lack of project maintenance (8/70).*** Some projects are no longer under active development, even though they remain functional and widely used. Without dedicated maintainers to migrate them, these projects remain on GOPATH due to inertia (e.g., issue #93 [20] of go-spew).

These challenges can be classified into two distinct types based on the maintenance status of the affected projects. The first type involves active projects facing high-effort migration barriers,

such as technical incompatibility or ecosystem disruption. While significant, these challenges are potentially surmountable as long as efforts are made. The second category, however, pertains to projects that are no longer actively maintained. This creates a large backlog of legacy libraries for which migration is largely unfeasible, as there are typically no developers available to perform the work. The persistence of this latter group is a critical factor, as it implies that a complete, ecosystem-wide transition to `Go Modules` is improbable. This insight helps explain Golang's 2023 policy reversal to preserve `GOPATH` [136], effectively accepting the dual library-referencing modes as an enduring feature of the ecosystem.

Motivations of migration: We also analyze the projects collected in both datasets, which have indeed migrated from `GOPATH` to `Go Modules` in the past five years. We examined their issue reports, particularly those discussing migration processes. In total, we identified 31 issue reports that explicitly address the motivations behind these migrations. From these reports, we observed four common motivations driving the migration to `Go Modules`:

- ***`Go Modules` as the new standard (12/31):*** With successive updates to the Go language, `Go Modules` became more stable and established itself as the default mode for dependency management. To maintain compatibility with the latest versions, many projects chose to migrate to `Go Modules` (e.g., issue #169 [10] of `scc`).
- ***New features of `Go Modules` (8/31):*** `Go Modules` introduced useful features which significantly improved development efficiency. To leverage these advantages, many projects chose to migrate to `Go Modules` (e.g., issue #408 of [59] `microservices-Demo`).
- ***Decline of third-party dependency management tools (6/31):*** After the introduction of `Go Modules`, many third-party dependency management tools gradually ceased to be maintained. As `GOPATH` projects lost support from these tools, developers opted to migrate to `Go Modules` (e.g., issue #109 of [119] `faasd`). This trend also explains the observed decrease in the usage of dependency management tools within `GOPATH` projects in the 2025 dataset.
- ***Pressure from downstream projects (5/31):*** To align with downstream projects that had transitioned to `Go Modules`, many upstream projects felt pressured to migrate as well (e.g., issue #9306 [90] of `kyma`).

The interplay between migration challenges and motivations is notable, as some factors transformed from impediments to catalysts. For instance, reliance on legacy DM tools, once a challenge, became a motivation of migration as those same tools were deprecated. Similarly, pressure from downstream projects acted as a bidirectional force: initially a challenge to protect GOPATH consumers, but reversed into a motivation as the ecosystem's center of gravity shifted towards Go Modules.

However, the primary drivers for migration are intrinsic to `Go Modules` itself. Its establishment as the default standard and its powerful features provided compelling incentives for adoption. Yet, these features introduce new rules and complexities, motivating our investigation in Section 3.3 on how well projects adhere to `Go Modules` post-migration.

Answer to RQ1: *The Golang ecosystem has taken a great step in migrating to `Go Modules`, with 78.6% of top 20,000 projects on GitHub migrated to `Go Modules` in 2025. Despite this positive ecosystem-wide trend, a deeper look inside individual projects reveals a persistent coexistence, where 82.3% of `Go Modules` projects still rely on at least one `GOPATH` library. This coexistence is not a transient condition but an enduring reality, sustained by unmaintained legacy `GOPATH` projects that `Go Modules` projects still depend on.*

Table 2. Violations of Features in `Go Modules` Projects (2020 vs. 2025)

Year	Violations of Features					Multi-Module Projects	Projects with Violation	Total Projects
	SIV	`replace`		`multi-module`	`GOPROXY`			
		adopted	inherited					
2020	1,195 (28.5%)	563 (13.4%)	75 (1.8%)	40 (1.0%)	46 (1.1%)	183 (4.4%)	1,640 (39.1%)	4,195
2025	3,333 (26.8%)	1,468 (11.8%)	419 (3.4%)	202 (1.6%)	187 (1.5%)	1,274 (10.2%)	4,553 (36.6%)	12,456

3.3 RQ2: Prevalence of Rule Violations in `Go Modules`

As discussed in Section 3.2.1, advanced features are a significant driver for migrating to `Go Modules`. However, the promise of a more robust dependency management system is undermined if its features are widely misused. To assess the situation, we investigated the adoption of four key `Go Modules` features—`SIV`, `replace`, `multi-module`, and `GOPROXY`—and the corresponding rule compliance across the latest release versions of the collected projects. The procedure was as follows:

- We selected projects from the top 20,000 that utilize `Go Modules` in their latest release versions.
- For these projects, we analyzed each project's latest version and module path to identify potential `SIV` rule violations. We also examined the `go.mod` files of the projects and their dependencies to count occurrences of the `replace` directive. Additionally, we verified the consistency of commit timestamps for the latest release versions from GitHub and `GOPROXY`.
- We classified projects with multiple `go.mod` files as `multi-module` projects and then identified submodules that were carved out from a root module by examining their history on the official package search site and checking their go.mod files to determine if they correctly depend on the root module.

Results. Our analysis indicates a modest decrease in overall rule violations, which contrasts with the significant increase in the adoption of Go Modules, from 4,195 projects in 2020 to 12,456 in 2025. As shown in Table 2, the percentage of projects with at least one violation shifted from 39.1% in 2020 to 36.6% in 2025. This slight improvement, when viewed against the substantial growth of the ecosystem, suggests that achieving widespread adherence to best practices remains a persistent challenge.

This persistence is observable across different features. The violation rate for `SIV`, for example, affected 26.8% of projects in 2025, a slight decrease from 28.5% in 2020. In a similar vein, while the direct adoption of the `replace` directive decreased, the issue of inherited directives became more prominent. The proportion of projects inheriting the directive from upstream dependencies increased from 1.8% to 3.4%, indicating that downstream projects are increasingly impacted by the dependency configurations of upstream modules. This highlights a systemic challenge in the ecosystem where module configurations can propagate and create constraints for consumers.

The use of the `multi-module` feature also grew considerably, with the number of such projects increasing nearly sevenfold from 183 in 2020 to 1,274 in 2025. This reflects its growing role in managing projects with complex structural layouts. While the rate of compliance within this specific cohort improved, the rapid adoption of the feature meant that the absolute number of projects with submodule violations increased fivefold, from 40 to 202. This growth in the total number of non-compliant projects underscores the expanding challenge of correctly applying the intricate dependency rules demanded by such complex structures.

Regarding the `GOPROXY` rule, violations in the latest project versions increased from 1.1% to 1.5%. It is important to note that this figure represents a single point in time for each project. The

Table 3. DM Issues Collected in 2020 and 2025

Issue Type	2020	2025	Merged
A	41	34	49
B	40	25	47
C	70	120	160
Total	151	179	256

cumulative number of violations across a project's entire release lifecycle is likely to be higher, posing a larger potential threat to the immutability and trustworthiness of the Go ecosystem's dependency graph.

A crucial aspect of these rule violations is that they often introduce latent risks that are not immediately visible within the violating project itself. A project with such a violation may build successfully in isolation, producing no errors or warnings for its own maintainers. The resulting problems, however, tend to manifest as build failures only when a downstream project attempts to consume the project. This delayed impact means that upstream developers may be unaware that they are causing significant issues for their users. A detailed analysis of these consequences is presented in Section 3.4 as part of our investigation into *Type C* issues.

Answer to RQ2: *The widespread adoption of `Go Modules` has not translated into a corresponding improvement in dependency management discipline, with the overall rule violation rate only decreasing from 39.1% in 2020 to 36.6% in 2025. This slight improvement, contrasted with massive project growth, suggests the ecosystem's persistent DM challenges stem from the complexities of applying advanced features at scale, indicating that resolution requires a focus on community best practices in addition to adopting the advanced dependency management mode.*

3.4 RQ3: Issue Types and Root Causes

We observed three common types of DM issues in issue reports collected in 2020 and 2025. Table 3 provides a summary of these DM issues, with a total of 256 unique issues after removing overlapping entries. Although there have been slight modifications in the names and definitions of these issues, they remain consistent and comparable between the 2020 and 2025 datasets. In the following sections, we introduce each type of issue and analyze their root causes, supported by relevant examples.

Type A. *DM issues can occur when projects in `GOPATH` depend on projects in `Go Modules` (49/256 =19.1%).* The former are typically module-unaware. Build failures may arise when such projects directly or transitively depend on the latter but are unable to recognize their virtual paths with version suffixes, e.g., issue #1017 of `glide` [98].

Among the 49 *Type A* issues, 39 occurred in module-unaware projects when they upgraded upstream dependencies that introduced virtual import paths in their newer versions. This highlights that version upgrades of libraries in `Go Modules` can pose risks to module-unaware downstream projects, and developers should assess these risks before upgrading. The remaining 10 issues occurred when new upstream projects were introduced that transitively depend on virtual import paths.

Type B. *DM issues can occur when projects in `Go Modules` depend on projects in `GOPATH` (47/256 =18.4%).* There are two cases. The first (*Type B.1*) is due to the different import path interpretations between `GOPATH` and `Go Modules`, and the second (*Type B.2*) is due to the interference of *Vendor* attribute in `GOPATH`.

Type B.1 *(19/47).* Let project P_A in `Go Modules` depend on project P_B in `GOPATH`, and P_B further depend on P_C in `Go Modules` with import path *github.com/user/PC*. Suppose that P_C has released a *v2+* version with the major branch strategy. From P_B's perspective, it interprets the import path as P_C's latest version (i.e., *v2+* version on P_C's main branch). However, in P_A's build environment, the import path is interpreted as a *v0*/*v1* version of P_C (no version suffix in the path). As a result, P_A fails to fetch P_C's correct version and can encounter errors when building with P_B.

Type B.1 issues are difficult to notice, and can easily cause build failures. For example, issue #47246 of `cockroach` [17] reported that a client project in `Go Modules` depends on `cockroach` v19.5.2 in `GOPATH`, and `cockroach` further depends on project `apd` [16] in `Go Modules` (with a *v2+* version). Although `cockroach` itself correctly referenced `apd` v2.0.0 (latest version) by interpreting import path *github.com/cockroachdb/apd*, the client project instead fetched `apd` v1.1.0 based on its interpretation of this import path. As a result, the client project's building failed due to missing an important field (not in `apd` v1.1.0 but in v2.0.0).

Type B.2 *(28/47).* Let project P_A in `Go Modules` depend on project P_B in `GOPATH`, and P_B further depend on project P_C, which is managed in P_B's *Vendor* directory. A *Vendor* directory is a major feature of `GOPATH`, which localizes the maintenance of remote dependencies' specific versions. We note that P_A references P_C by import path *github.com/user/PC* declared in P_B's source files rather than from P_B's *Vendor* directory. Although the build may work for the time being, P_A can fail to fetch P_C if P_C is deleted or moved to another repository (e.g., renaming). Even if the fetching is successful, the version on P_C's hosting site could be different from the one in P_B's *Vendor* directory, causing potential build failures due to the inconsistency.

Such situations often occur, since there are essentially two versions of a library at two different sites and their consistency is not guaranteed. We witnessed a *Type B.2* issue in project `moby` [110], which has received 68.4k stars on GitHub and ranked the eighth in popularity. To support its large number of downstream projects still in `GOPATH`, `moby` has not migrated to `Go Modules`. Its issue #39302 [109] reported that `moby` referenced project `logrus` [146] from its *Vendor* directory, and `logrus` had been relocated from *github.com/**S**irupsen/logrus* to *github.com/**s**irupsen/logrus* (case sensitive) on GitHub. This incurred DM issues to many of `moby`'s downstream projects in `Go Modules` (e.g., issues #127 of `testcontainers` [153] and issue #2 of `shnorky` [145]), as they could not fetch `logrus` by the import path in `moby`'s source files.

Type C. *DM issues can occur when projects in `Go Modules` depend on projects in `Go Modules` but contain certain rule violations (160/256=62.5%).* There are four cases of rule violations (C.1–4) in consideration, each corresponding to the misuse of one of the following `Go Modules` features: `SIV`, `replace`, `multi-module`, and `GOPROXY`. Among these, violations related to `SIV` and `replace`, which were also identified as the most prevalent in our large-scale analysis (Section 3.3), are the most frequent cause of Type C issues, indicating their high impact level.

Type C.1 *(83/160).* We identified three types of `SIV` rule violations that caused build failures to downstream projects: (1) lacking version suffixes like "*/v2*" in module paths or import paths, although the versions of concerned projects are *v2+* (41/83) (e.g., issue #1355 [79] of `iris`); (2) version tags not following the `MAJOR.MINOR.PATCH` format (25/83) (e.g., issue #1848 [120] of `gobgp`); (3) module paths in `go.mod` files are inconsistent with URLs associated with concerned projects on their hosting sites (17/83) (e.g., issue #9 [76] of `jwplayer`).

While downstream projects can encounter build failures, the projects violating `SIV` rules do not produce warnings or errors themselves when building. Currently, there is no diagnosis technique to detect the three `SIV` rule violation types, or mechanism to enforce `SIV` rules, as discussed in issues #1355 of `iris` [79] and #32695 of `golang/go` [54] (by `lz4`'s [129] users). As a result, projects violating `SIV` rules can "safely" stay in the Golang ecosystem, despite the unexpected consequences

to their downstream projects. Regarding such risk, `lz4`'s developers commented its severity on issue #32695 that *"we need to fix this issue and figure out how big the crater it brings to the ecosystem."*

Type C.2 *(37/160).* Let project P_A in `Go Modules` depend on project P_B, with project P_B using the `replace` directive to substitute dependency P_{C1} with P_{C2}. From project P_B's perspective, the dependency on P_{C1} is successfully replaced with P_{C2}. However, since the `replace` directive is not inheritable, project P_A continues to depend on P_{C1} rather than the intended P_{C2}. This discrepancy can lead to build failures or inconsistencies when P_A is built with P_B.

Type C.2 issues may have significant repercussions on the Golang ecosystem. In issue #20421 [85] of `test-infra`, `test-infra` imports `client-go@v11` but uses the `replace` directive to substitute it with another package, creating a ripple effect. Specifically, any downstream project that imports `test-infra` must also apply the same `replace` directive in its `go.mod` file. This practice propagates the `replace` directive throughout the Golang ecosystem, potentially forcing projects to adopt unnecessary or incompatible dependencies. Moreover, in Golang 1.16, this issue becomes even more problematic, as the `go install` command fails when the `go.mod` file contains a `replace` directive, adding further inconvenience and frustration for developers.

Type C.3 *(25/160).* Let project P_A in `Go Modules` depend on the `single-module` version (v1.0.0) of the `Go Modules` project P_B and utilize the functionality in P_B's `api` package. Subsequently, project P_B separates the `api` package into a standalone submodule P_{sub} in version v2.0.0. When project P_A updates its `go.mod` file to include the new P_{sub} submodule, it ends up with two different modules that provide the same `api` package: the old `single-module` version of P_B and the new P_{sub}. This situation leads to build failures, as the Golang compiler encounters ambiguity and cannot determine which module should be finally selected.

Type C.3 issues are becoming increasingly prevalent as the Golang ecosystem evolves and projects expand in size and complexity. Developers are increasingly adopting a `multi-module` approach to separate distinct functional components within their projects as discussed in Section 3.3, which can inadvertently give rise to *Type C.3* issues if rules of `multi-module` are not enforced. This issue is exemplified by case #2543 [58] in the `google-api-go-client` project, where a downstream project intending to use only the `androidpublisher/v3` package encountered ambiguous import errors related to the `cloud.google.com/go/compute/metadata` package. The root cause of this conflict lies in the fact that `cloud.google.com/go@v0.26.0` had not yet been refactored into multiple modules. Consequently, it included its own version of `cloud.google.com/go/compute/metadata`, which conflicted with another instance of the same package introduced via transitive dependencies.

Type C.4 *(15/160).* Let project P_A initially release version v0.0.1. Following this release, the developer of project P_A identifies a minor issue in the version. Instead of issuing a new version, the developer opts to resolve the problem by removing the original v0.0.1 tag, committing the fix, and re-tagging it as v0.0.1. However, due to the caching mechanism of `GOPROXY`, the mirrored copy of project P_A's v0.0.1 version will remain unchanged, still containing the erroneous code. As a result, projects relying on project P_A will continue to use the outdated and faulty v0.0.1 version with `GOPROXY`.

Downstream projects that fetch dependencies directly from hosting sites may experience build failures if *type C.4* issues arise in their upstream projects. For instance, issue #814 [118] in the `ginkgo` project illustrates this scenario, where the version tag v1.16.3 was initially associated with an incorrect commit and later reassigned to the correct one. However, `GOPROXY` continued to reference the original incorrect commit. Consequently, when a downstream project attempts to download `ginkgo@v1.16.3` directly from the source, it encounters a build failure due to a checksum mismatch between the downloaded project and the version cached in `GOPROXY`.

Answer to RQ3: *DM issues commonly occur due to heterogeneous uses of GOPATH and Go Modules as well as developers' violations of Go Modules' rules. Their manifestations can be summarized into three types and there are two common root causes: (1) GOPATH and Go Modules interpret import paths in different ways, and (2) the lack of strict enforcement of Go Modules' rules across the Golang ecosystem.*

3.5 RQ4: Fixing Solutions

Out of the 256 DM issues, 245 issues have fixing patches or fixing plans that developers have agreed on. We studied them and observed thirteen common fixing solutions, which demonstrate different trade-offs. To validate that these solutions represent widespread practices rather than project-specific patterns, we analyzed the origins of the 245 fixed issues. The results confirm high cross-project diversity. We found that instances of the same fixing strategy being applied multiple times within the same project are exceptionally rare, occurring in 16 cases within three of the most frequently observed solutions. This extremely low project-level overlap indicates that the thirteen fixing strategies we identified are not localized workarounds but represent broadly adopted, independent practices across the Golang ecosystem. In the following, we present each of these thirteen solutions in detail.

Solution 1: *Projects in GOPATH migrate to Go Modules (30/245=12.2%).* Migrating from GOPATH to Go Modules can help fix *Type A* issues, since these issues are caused by projects still in GOPATH, which are unable to recognize import paths with version suffixes. For example, in issue #454 [74], redis [43] migrated to Go Modules, but its downstream project benthos was still in GOPATH. Then, benthos was suggested to migrate to Go Modules to avoid build failures. This solved benthos's problem, but caused incompatibility to benthos's module-unaware downstream projects. As a result, new *Type A* issues (e.g., issue #232 [73]) arose.

Solution 2: *Projects in Go Modules roll back to GOPATH (13/245=5.3%).* Some projects rolled back to GOPATH after migrating to Go Modules for fixing *Types A* and *C* issues. For example, in issue #61 [44] (*Type A*), project uuid's [46] migration to Go Modules broke the building of many downstream projects in GOPATH. As a compromise, uuid rolled back to GOPATH, waiting for downstream projects to migrate first. In issue #663 [144] (*Type C*), gopsutil and its downstream projects were all in Go Modules, but gopsutil violated SIV rules (lacking a version suffix in its module path of *v2+* release), causing build failures to downstream projects. As such, gopsutil chose to roll back to GOPATH to make downstream projects work again. This solution solves the problem, but hinders the migration status of the ecosystem.

Solution 3: *Changing the strategy of releasing v2+ projects in Go Modules from major branch to subdirectory (5/245=2.0%).* It helps resolve *Type A* issues, where module-unaware projects cannot recognize virtual import paths for *v2+* libraries in Go Modules. The new strategy creates physical paths by code clone, so that libraries can be referenced by module-unaware projects. However, this is just a workaround and needs extra maintenance in subsequent releases (e.g., issue of go-i18n [115] as discussed in Sec 1).

Solution 4: *Maintaining v2+ libraries in Go Modules in downstream projects' Vendor directories rather than referencing them by virtual import paths (6/245=2.4%).* Similar to *solution 3*, this solution also helps resolve *Type A* issues. By making a copy of libraries in downstream projects' repositories, it avoids fetching the libraries by virtual import paths. For example, in issue #141 [100], radix [101] refused to use the major subdirectory strategy for its *v2+* project release in Go Modules. Its downstream projects had to make a copy of radix's code in their *Vendor* directories, which requires extra maintenance and potentially causes *Type B.2* issues in future.

Issue Type	Solution's Impact(B/C)	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13
Type A	Benefits	ab1	ab2	ab2	-	-	-	-	-	-	-	-	-	-
	Consequences	uc1	uc2	uc3	uc3|4									
Type B.1	Benefits	-	-	-	-	-	-	-	-	-	-	-	-	-
	Consequences					uc3|4								
Type B.2	Benefits	-	-	-	-	-	ab3	-	-	-	-	-	-	-
	Consequences						-							
Type C.1	Benefits	-	ab2	-	-	-	-	ab3	-	-	-	-	-	-
	Consequences		uc2			uc3|4		uc1	uc3					
Type C.2	Benefits	-	-	-	-	-	-	-	-	ab3	-	-	-	-
	Consequences					uc3|4				-				
Type C.3	Benefits	-	-	-	-	-	-	-	-	-	ab3	-	-	-
	Consequences										-	uc4		
Type C.4	Benefits	-	-	-	-	-	-	-	-	-	-	-	ab3	-
	Consequences											uc4	-	uc4

Additional benefits (ab):

ab1:Promoting the migration to Go Modules
ab2:Supporting downstream projects without module-awareness
ab3:Supporting downstream projects in Go Modules(module-aware)

"S1-13" denote fixing solutions 1-13.

Undesired consequences (uc):

uc1:Breaking compatibility with downstream projects without module-awareness
uc2:Hindering the migration to the ecosystem
uc3:Increasing maintenance effort
uc4:Introducing potential DM issues in future

Fig. 12. Benefits and consequences of the thirteen fixing solutions

Solution 5: *Using a* `replace` *directive with version information to avoid using import paths in referencing libraries (37/245=15.1%).* It addresses *Type B.1* (problematic import path interpretations) and *Type C.1* (import path violating `SIV` rules) issues. For example, in issue #12 [3], a client project used a directive to `replace` the original import path: `replace github.com/andrewstuart/goq => astuart.co/goq v1.0.0`, to reference its expected project `goq`'s [2] version. However, this would make developers no longer able to use the `go get` command to automatically fetch upgraded libraries and may introduce *Type C.3* issue in future.

Solution 6: *Updating import paths for libraries that have changed their repositories (27/245=11.0%).* It fixes *Type B.2* issues, where libraries in a project's *Vendor* directory may be inconsistent with the ones referenced by their import paths. It updates import paths to help a project's downstream projects in `Go Modules` fetch consistent library versions. For example, in issue #429 [57], `go-cloud` [56] managed library `etcd` in its *Vendor* directory, `etcd` later changed its hosting repository from *github.com/coreos/etcd* to *go.etcd.io/etcd*. To fix build failures for its downstream projects in `Go Modules`, `go-cloud` updated `etcd`'s import path to the latest one for the consistency. This fixes the issue and benefits all affected downstream projects without impacting others in the ecosystem.

Solution 7: *Projects in* `Go Modules` *fix configuration items to strictly follow SIV rules (58/245=23.7%).* Projects that have migrated to `Go Modules` are suggested to follow Golang's official `SIV` rules to fix their induced *Type C* issues. For example, in #1149 [41], project `redis` [43] added a version suffix "*/v7*" at the end of its module path to follow `SIV` rules. However, we noticed that while the issues are fixed, the project's downstream projects in `GOPATH` may be impacted (unable to recognize the version suffixes, e.g., issue #1151 [42] reported for `redis`).

Solution 8: *Using a hash commit ID for a specific version to* `replace` *a problematic version number in library referencing (11/245=4.5%).* It fixes *Type C* issues, where some projects in `Go Modules` violate `SIV` rules in version numbers and cause build failures to downstream projects that are also in `Go Modules`. It avoids referencing problematic version numbers, by a *require directive* with a specific

hash commit ID. For example, in issue #6048 [137], one of `prometheus`'s downstream projects in `Go Modules` chose to use directive `require github.com/prometheus/prometheus 43acd0e` to reference its expected version v2.12.0. Similar to *Solution 5*, this solution would also make developers unable to automatically fetch upgraded libraries using command `go get`.

Solution 9: *Project removes the use of the `replace` directive in release versions (21/245=8.6%).* To address *Type C.2* issues, it is advisable for developers to eliminate the use of the `replace` directive when releasing stable versions intended for downstream use. For example, in issue #10032 [105], `minio` removed the `replace` directive and directly modified the version declared in `go.mod`. This solution benefits all downstream projects in `Go Modules`, as no `replace` directive needs to be inherited.

Solution 10: *Submodules declare root module as dependency (9/245=3.7%).* To address *Type C.3* issues, developers should declare a specific dependency on the root module in newly added submodules and the version of the root module should be after the version when the submodules were split. Thus whenever downstream projects depend on the submodule, Golang will select the newer version of root module which is after submodules are split (e.g, issue #481 [5] of `go-autorest`).

Solution 11: *Downstream projects which use `Go Modules` adjust the version of dependency (13/245=5.3%).* This solution addresses *Type C.3* and *Type C.4* issues by upgrading or downgrading the dependency to a version free of DM issues. For example, when a downstream project depends on a version of the root module that predates its split into multiple modules, the issue can be resolved by updating the dependency version in the downstream project's `go.mod` file. By adjusting the version constraint to a newer release where the `multi-module` structure has been implemented, the downstream project can avoid issues related to ambiguous references caused by the upstream projects' DM issues (e.g., issue #278 [102] of `micro`). However, DM issues may recur if the Golang toolchain ultimately selects problematic versions.

Solution 12: *Upstream project releases a new version tag (11/245=4.5%).* This solution addresses *Type C.4* issues. Due to the caching mechanism of `GOPROXY`, when a developer releases a version containing an error, it is advisable to release a new patch version to correct the issue rather than deleting the original tag and republishing. This approach helps prevent versioning inconsistencies in downstream projects and avoids complications arising from cached versions in `GOPROXY` (e.g., issue #1014 of `dns`).

Solution 13: *The downstream projects use `GOPROXY` to fetch dependencies (4/245=1.6%).* This solution effectively addresses *Type C.4* issues. These issues stem from checksum discrepancies when dependencies are fetched directly from source repositories rather than through `GOPROXY`. By using `GOPROXY` to fetch dependencies, downstream projects ensure that both the checksum and the dependency are consistently accessed from the same source, thereby eliminating these discrepancies (e.g., issue #1169 [87] in `kubebuilder`). However, this is only a temporary workaround, as DM issues may recur if `GOPROXY` needs to be disabled in the future.

As summarized in Figure 12, these solutions fix their targeted DM issues, but at the same time they may bring additional benefits ($ab1$–$ab3$) or undesired consequences ($uc1$–$uc4$). When there are multiple fixing solutions for a specific DM issue, developers are suggested to carefully consider the relevant dependencies and minimize the impact on other projects in the ecosystem, by weighing consequences against benefits.

Answer to RQ4: *We observed thirteen common fixing solutions for DM issues, covering 95.7% of the studied issues. Most solutions could affect other projects in the ecosystem. When fixing a DM issue, developers should find a tradeoff between the benefits and the possible consequences.*

4 HERO+: DM Issue Diagnosis

Our empirical study reveals a high prevalence of DM issues within the Golang ecosystem, primarily due to the mixed usage of library-referencing modes and the evolving features of `Go Modules`. This led to the development of a tool named HERO, designed to automatically detect DM issues and provide tailored solutions. Additionally, we extended HERO into HERO+ to integrate new features and changes to existing functionalities in `Go Modules`.

HERO+ operates in two primary steps. First, it extracts dependencies among Golang projects and their library-referencing modes. Second, it detects DM issues within these projects based on the issue types and root causes identified in RQ3. Furthermore, it provides customized fixing suggestions based on the findings from RQ4. The following sections explain how HERO+ models project dependencies and detects DM issues.

4.1 Constructing Dependency Model

First, we construct a dependency model for the Golang project under analysis. To capture the evolving features of `Go Modules`, the HERO+ model includes the following key components:

- $vr = (v, st, pt)$ stores information of a version, where v represents the exact version number. Fields st and pt denote the checksum of v on its hosting site and `GOPROXY`, respectively.

The vr component captures the interaction between the current project and repositories commonly used for fetching dependencies, namely hosting sites and `GOPROXY`. This component enables HERO+ to identify whether a version has been modified since its initial release, thereby providing insights into potential inconsistencies across different repositories.

- $mo = (mp, S, rp, sub)$ records information about a module, where mp represents the module path (for the module to be referenced by downstream projects), and rp is a collection of `replace` directives for the module. Field S specifies whether the module is released using the major branch strategy (`yes` or `no`), indicating whether mp is a virtual path. Field sub is an enumeration-type variable that represents the module's role within a `multi-module` project and is set to 0 if the module is the root module. For submodules, sub is assigned a value of 1 or 2, depending on whether the submodule correctly depends on the root module.

The mo component ensures accurate tracking of submodules within `multi-module` projects. Without mo, submodules may be overlooked in the analysis. Additionally, the rp field captures the `replace` directive, which records instances where dependencies are redirected to alternative sources or versions. Neglecting rp would result in tracking pre-replacement dependencies, causing inconsistencies with the Golang toolchain's behavior.

- $pj = (mp, t, vd, ip)$ component records information about a `GOPATH` project, where mp denotes the module path. Field t indicates whether pj relies on any DM tools (either `yes` or `no`), and vd is a collection of import paths (a set of URLs) that reference upstream libraries maintained in pj's *Vendor* directory but cannot be found at the specified URLs (e.g., due to removal or renaming). Field ip represents the import paths for libraries within the project's source files.

The pj component is crucial for modeling of `GOPATH` project dependencies. Since such projects lack a `go.mod` file to explicitly declare their dependencies, the ip field provides the necessary data to reconstruct the dependency graph by analyzing all imported packages.

Based on the three components, we formally define the dependency model below.

Definition 5 (Dependency model): The dependency model $\mathcal{D}(P_v)$ for version v of a project P is a 3-tuple (Pr, Ds, Us).

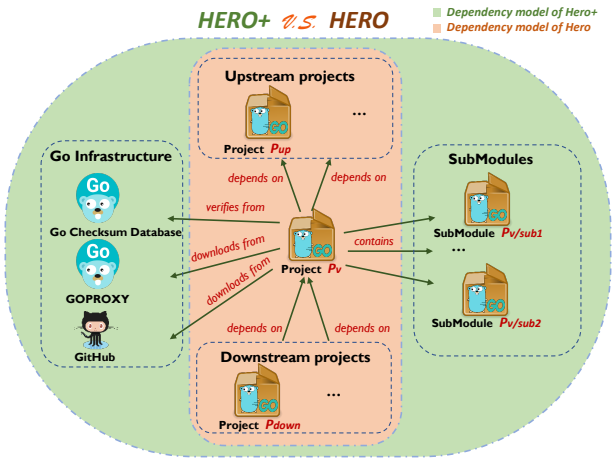

Fig. 13. Dependency model of Hero and Hero$^+$

- $Pr = (vr, md, mos, pj)$ records the information of the **current project**, where vr records the information of P_v's version v and md is P_v's library-referencing mode (`GOPATH` or `Go Modules`). Field mos contains all the modules' information mo if Pr uses `Go Modules`, and is empty otherwise. Conversely, if Pr is in `GOPATH`, pj holds the project's relevant information.
- $Ds = \{dp_1, dp_2, \cdots, dp_n\}$ represents a collection of P_v's **downstream projects** dp_i, where each $dp_i = (v_i, md_i, mo_i, pj_i)$. Field v_i denotes the latest version of dp_i. Fields md_i and pj_i record information about downstream projects in `Go Modules` and `GOPATH`, respectively.
- $Us = \{up_1, up_2, \cdots, up_n\}$ represents a collection of P_v's **upstream projects** up_i, where each $up_i = (vr_i, md_i, mo_i, pj_i, I_i)$. Field md_i denotes the library-referencing mode of up_i, and fields mo_i and pj_i contain information specific to projects in `Go Modules` and `GOPATH`, respectively. If P_v uses `Go Modules`, field vr_i represents the specific version of up_i as declared in P_v's configuration file. Otherwise, if P_v uses `GOPATH`, vr_i refers to the latest version of up_i. Additionally, if both up_i and P_v use `Go Modules`, field I_i indicates whether up_i is transitively included in P_v by any project in `GOPATH` (`yes` or `no`).

Below, we explain how to obtain the field values of Hero$^+$'s dependency model.

Step 1: Collecting Pr information. Leveraging GitHub's REST API "*repository_url*" [40], Hero$^+$ queries the repository name of P_v to determine its library-referencing mode md by checking for the presence of a `go.mod` file in the repository. The remaining information is collected in two ways, depending on the library-referencing mode of the project.

- P_v in `Go Modules`: Hero$^+$ builds vr by fetching the version v's commit checksum from GitHub (st) using REST API endpoints for git tags [40], as well as from the `GOPROXY` endpoint (pt) [51]. Next, mos is obtained by traversing the project to identify all submodules containing their own `go.mod` files and collecting their module information mo. The mp and rp fields of mo are obtained by parsing the `go.mod` file to identify the module path and any `replace` directives, respectively. The sub field is determined based on the location of the `go.mod` file and whether the root module's module path is correctly specified within it. Hero$^+$ identifies the release strategy S by checking for the presence of subdirectories such as $up_i/v2$.
- P_v in `GOPATH`: Hero$^+$ uses the version v of P_v for vr and leaves st and pt empty. The project information is recorded in fields of pj. The mp field of pj is obtained from the query results,

and the t field is determined by the presence of any configuration files for DM tools. The ip field is determined by parsing the source files of P_v to collect import paths for libraries maintained in the *Vendor* directory, and vd is further obtained by querying the "*repository_url*" API with ip to verify whether the corresponding libraries have been deleted or relocated (e.g., by returning `HTTP 404: Not Found` errors [40]).

Step 2: Collecting** Ds **information. Leveraging GitHub's REST API "*code_search_url*" [40], Hero$^+$ queries with P_v's repository name to check which projects depend on it. This information is from the *require directives* of a project's `go.mod` file, *import directives* of its source files, or a DM tool's configuration file. Each found project corresponds to an item dp_i in the collection Ds. Note that Hero$^+$ collects the latest version v_i for dp_i, and decides its library-referencing mode md_i (by checking whether P_v's repository name is declared in its `go.mod` file). Fields mo_i and pj_i can be obtained in a similar way as described in *Step 1* via fetching source files. These collected downstream projects depend on P_v and can also reference its earlier versions.

Step 3: Collecting** Us **information. Project P_v's upstream projects information is collected in two ways, depending on the library referencing mode of the project:

- P_v in `Go Modules`: Hero$^+$ collects P_v's upstream projects up_i by parsing the `go.mod` file, which configures a project's direct and transitive dependencies with import paths mp and specific version numbers v_i. Based on mp and v_i, Hero$^+$ checks whether the corresponding directory of the relevant project contains a `go.mod` file to determine field md_i of up_i. Field I is determined by checking whether md_i is included in the import paths of any upstream projects that use `GOPATH`. Fields vr_i, mo_i, and pj_i are obtained similarly to *Step 1*.
- P_v in `GOPATH`: Hero$^+$ collects P_v's direct dependencies up_i with import paths mp from its source files. With the import paths, Hero$^+$ leverages GitHub's "*repository_url*" API to look into these dependencies' repositories to collect their latest versions, from which it decides the corresponding version numbers v_i and library-referencing modes md_i. Then Hero$^+$ recursively collects the information of P_v's transitive dependencies declared in `go.mod` or sources files in concerned repositories, and identifies version numbers, import paths, library-referencing modes in a similar way.

Hero$^+$ versus Hero. The dependency model $\mathcal{D}(P_v)$, populated by the three steps above, represents the core of Hero$^+$'s analytic capabilities. Its key advancements over our previous work, Hero, are a direct result of incorporating the vr, mo, and pj components. As visualized in Figure 13, while Hero focuses on upstream and downstream projects, the dependency model of Hero$^+$ delves deeper into these projects and also offers a broader perspective that extends beyond them. First, Hero$^+$ integrates additional `Go Modules'` features, such as `multi-module` support, enabling comprehensive analysis across all modules in `multi-module` projects and offering deeper insights into dependencies within complex project structures. Second, Hero$^+$ extends its analysis beyond specific release versions by interacting with Golang infrastructure components, such as the Go Checksum Database. Differing from Hero, which centers on a release version's source code, Hero$^+$ is now able to monitor the lifecycle of the version, from its inception to subsequent updates. This offers a holistic perspective on a project's evolution within the Golang ecosystem.

4.2 Diagnosing DM Issues

The dependency model built by Hero$^+$ contains sufficient information for detecting DM issues and suggesting solutions.

Detecting DM issues. Our study disclosed that most DM issues caused build errors, already observable. Therefore, Hero$^+$ focuses on identifying latent DM issues that have not yet manifested but would cause build failures under common scenarios. For instance, in the docker/compose

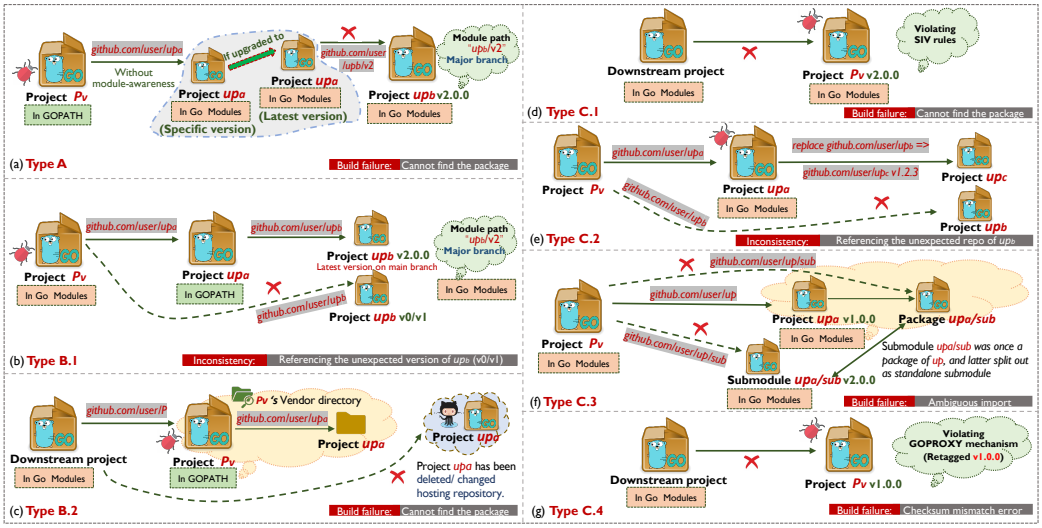

Fig. 14. Three types of DM issues Hero+ detects

project (PR #11268 [30]), a routine, automated dependency upgrade raised by an automated bot triggered a latent Type B.2 DM issue, finally resulting in a build failure. This proactive detection strategy of Hero+ aims to shift the maintenance process from a reactive stance, where failures are fixed after they occur, to a proactive one focused on prevention. This shift enables developers to address potential problems before they impact the broader ecosystem. Due to page limit, we explain scenarios for which Hero+ reports issues in this paper with algorithm details on our website.

Type A. Figure 14(a) shows a scenario, where a module-unaware project P_v references a specific version of its upstream project up_a in `Go Modules`. This version is older than up_a's latest version, which newly introduces another upstream project up_b in `Go Modules` with a *v2+* version released using the major branch strategy. No build failures occur in P_v when referencing the older version of up_a. However, if P_v updates to reference the latest version of up_a, it will fail to recognize up_b's virtual import path. When seeing such a possibility, Hero+ reports a warning of *Type A* issue for P_v.

Type B.1. Figure 14(b) shows a scenario, where project P_v in `Go Modules` transitively references a *v2+* upstream project up_b in `Go Modules` (released by the major branch strategy) through another module-unaware project up_a in `GOPATH`. Since `GOPATH` and `Go Modules` interpret import paths differently, up_a would use up_b's latest version (e.g., v2.0.0), while P_v would use up_b's old *v0/v1* version, causing inconsistencies. Thus, Hero+ reports a warning of *Type B.1* issue for P_v.

Type B.2. Figure 14(c) shows a scenario, where project P_v in `GOPATH` references an upstream project up_a maintained only in its *Vendor* directory (i.e., up_a has already been deleted or relocated). No build failures occur when P_v has no downstream projects in `Go Modules`. However, if P_v has such downstream projects, the latter would fetch up_a via its import path (i.e., hosting repository) rather than from P_v's *Vendor* directory, causing build failures due to failing to fetch up_a. Thus, Hero+ reports a warning of *Type B.2* issue for P_v.

Type C.1. Figure 14(d) shows a scenario, where project P_v in `Go Modules` violates `SIV` rules (as discussed in Sec 3.4). The violation may not introduce build failures when P_v has no downstream projects in `Go Modules`. However, build failures would occur if such projects exist in future. Thus, Hero+ reports a warning of *Type C* issue for P_v.

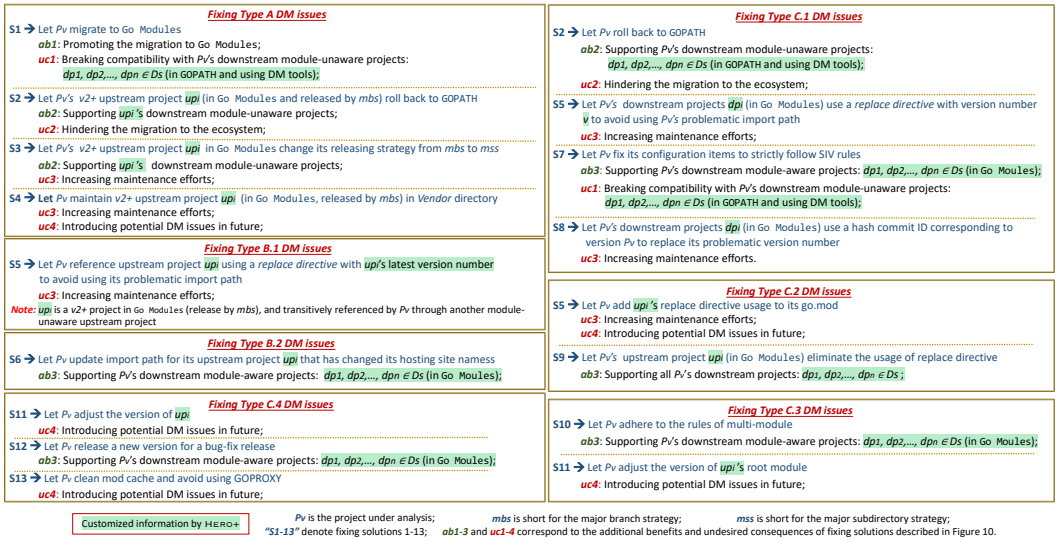

Fig. 15. Templates of customized fixing suggestions for three types of DM issues

Type C.2. Figure 14(e) presents a case where the project up_a in `Go Modules` violates the rules of `replace` directive. Specifically, up_a declares a dependency on up_c through a `replace` directive. However, due to the non-inheritance of `replace` directives, P_v erroneously depends on up_b. While this violation may not immediately trigger build failures when P_v interacts with the dependency, such errors are likely to manifest as up_a undergoes updates. Consequently, HERO$^+$ identifies and classifies this as a *Type C.2* issue for P_v.

Type C.3. Figure 14(f) illustrates a scenario where project up_a was initially a single-module project in version v1.0.0, with up_a/sub as a package within up_a. In version v2.0.0, up_a splits up_a/sub into a standalone submodule. As a result, P_v, using `Go Modules`, simultaneously depends on up_a at version v1.0.0 and on the split submodule up_a/api at version v2.0.0. As a result, up_a/sub is included both as part of up_a in v1.0.0 and as a separate submodule in v2.0.0. This duplication introduces potential build failure because of ambiguous import. Therefore, HERO$^+$ flags this situation as a *Type C.3* issue for P_v.

Type C.4. Figure 14(g) depicts a scenario where project P_v re-releases a tag, resulting in inconsistencies between version v1.0.0 of P_v on `GitHub` and `GOPROXY`. If downstream projects fetch dependencies directly from `GitHub`, this inconsistency may lead to build failures in those downstream projects. Consequently, HERO$^+$ raises a *Type C.4* issue warning for P_v.

Customized fixing suggestions. Our empirical study has identified applicable fixing solutions for each issue type (Figure 12). We summarize the impacts of these solutions as templates in Figure 15. For each detected DM issue, HERO$^+$ suggests the applicable solutions to developers by customizing the template with potential impact analysis based on the associated dependency model.

5 Evaluation

We study two research questions in our evaluation:

- **RQ5 (Effectiveness)**: *How effective is HERO and HERO$^+$ in detecting DM issues for Golang projects?*

Table 4. HERO and HERO+'s effectiveness on DM issue detection

Types \ Results	Hero				Hero+			
	Ground Truth	*Detected*	*Missed*	*Detection Rate*	*Ground Truth*	*Detected*	*Missed*	*Detection Rate*
Type A	41	39	2	95.1%	41	41	0	100.0%
Type B.1	15	15	0	100.0%	15	15	0	100.0%
Type B.2	29	29	0	100.0%	29	29	0	100.0%
Type C.1	54	51	3	94.4%	54	51	3	94.4%
Type C.2	18	0	18	0.0%	18	18	0	100.0%
Type C.3	7	0	7	0.0%	7	7	0	100.0%
Type C.4	24	0	24	0.0%	24	22	2	91.7%
Summary	**188**	**134**	**54**	**71.3%**	**188**	**183**	**5**	**97.3%**

- **RQ6 (Usefulness)**: *Can HERO and HERO+ detect new DM issues for real-world Golang projects and assist the developers in fixing the detected issues?*

For RQ5, we conducted experiments using the DM issues associated with projects in $subjectSet_2$. These issues were collected following the procedure described in Section 3.1. After removing overlaps between the 132 DM issues from the 2020 collection and the 90 from the 2025 collection, the final benchmark contains 188 unique DM issues and their corresponding project versions. Note that none of these issues overlap with those used in our empirical study. This dataset was used to evaluate whether HERO and HERO+ could detect these issues in buggy versions or predict them in earlier versions. To the best of our knowledge, no existing tool targets the specific DM issue types addressed in this work. Consequently, our evaluation compares HERO+ with its predecessor, HERO, which serves as the only available baseline. It is also worth noting that issue-fixing versions are not necessarily issue-free, since new DM issues may be introduced after the original ones are fixed, as discussed earlier.

For RQ6, we applied HERO and HERO+ to the rest 19,000 of the top 20,000 Golang projects (i.e., excluding those used for RQs 3–5). We reported the detected issues together with root cause analyses and fixing suggestions to respective developers. In our issue reports, we also highlighted the preferred solutions based on their impact on other projects.

5.1 RQ5: Effectiveness

Experimental setup. The benchmark dataset contains 188 DM issues, including 41 *Type A* (21.8%), 15 *Type B.1* (8.0%), 29 *Type B.2* (15.4%), 54 *Type C.1* (28.7%), 18 *Type C.2* (9.6%), 7 *Type C.3* (3.7%), 24 *Type C.4* (12.8%) issues. We collected their corresponding project versions to evaluate HERO and HERO+'s capability of detecting or predicting DM issues:

- *Type A:* These issues occurred when module-unaware projects in `GOPATH` referenced *v2+* dependencies in `Go Modules` by virtual import paths. Since issue occurrences would already cause build failures, we ran HERO and HERO+ on the previous project versions where such issues had not occurred.
- *Type B.1:* These issues occurred when projects in `Go Modules` referenced dependencies in `GOPATH`, with different import path interpretations to *v2+* projects released by the major branch strategy. The inconsistency may not lead to immediate build failures or functional failures, but is indeed risky. Thus, we ran HERO and HERO+ on the current project versions to check whether it can detect potential issues.
- *Types B.2, C.1, C.2, C.3 and C.4: Type B.2* issues occurred when the dependencies maintained in the current projects' *Vendor* directories were deleted or relocated remotely. The others occurred when the current projects in `Go Modules` violated rules of `Go Modules'` features. In both cases, the current projects would not have symptoms like build failures, but their

downstream projects in `Go Modules` would when referencing them in future. Thus, we ran Hero and Hero$^+$ on current project versions to check whether it can detect potential issues.

Results. Table 4 presents our experimental results. Hero and Hero$^+$ identified 134 and 183 DM issues (all true positives), respectively, covering 71.3% and 97.3% of the 188 issues in the ground truth. As shown, Hero$^+$ consistently outperformed or matched Hero across all categories of DM issues. Notably, Hero$^+$ successfully detected all instances of DM issues in Types *C.2–C.3*. This improvement is attributed to the advanced dependency model used by Hero$^+$, which incorporates additional information essential for detecting a broader range of DM issues. The only undetected issues by Hero$^+$ belong to *Type C.1* and *Type C.4*, as the problematic versions in the affected projects were either resubmitted or deleted. By default, both Hero and Hero$^+$ analyze only the latest available versions of releases for *Type C* issues.

5.2 RQ6: Usefulness

For the 19,000 Golang projects in consideration, Hero and Hero$^+$ reported 2,433 and 13,408 new issues after analyzing them altogether. Although the key information of root causes and fixing suggestions can be automatically generated, reporting these issues to developers involves substantial manual work, such as communicating with developers, helping them submit PRs, etc. As such, we choose to report issues for the top 1001–2000 popular projects (top 1–1000 already used for RQs 3–5) in the projects' issue trackers. Until July 2025, we reported a total of 449 issues, summarized in Table 5. Encouragingly, 310 issues (69.0%) were promptly confirmed by developers, among which 297 issues (95.8%) were subsequently fixed or are currently being addressed. For all but two fixed issues, developers implemented our suggested fixes. The remaining issues are still pending, likely due to project inactivity or other project-specific considerations from the developers.

Feedback on issue detection. While confirmation rates vary across different types of DM issues (52.9%–80.5%), most confirmed issues received positive feedback from developers. We give some examples below. In issue #2922 (*Type A*) of `kiali` [81], the developer confirmed the issue and mentioned "*I have found the same issue as you describe via the commit `c453e89` [80]. I just stuck in an older version of this library*". In issue #256 (*Type B.1*) of `flamingo-commerce` [66], developers were previously unaware of the risk and commented "*I guess the inconsistency of library version was imported by accident. We will create a PR to remove the occurrence*". In issue #114 (*Type B.2*) of `tomato` [156], a developer commented "*Nice catch! I think it is nice to clean up our vendor directory, since library `bitly/go-nsq` repository is not existed anymore*". We also reported issue #16381 (*Type C.1*) [132] to project `tidb` [133] that violated `SIV` rules and this specific issue could even affect 341 downstream projects!

Our reports also facilitate a better understanding of `Go Modules` features and their corresponding rules. For instance, in issue #4593 (*Type C.1* of `steampipe` [158]), developers initially overlooked the `SIV` rule requiring a `/v2` suffix for major version updates since its their first v2 release just in June 2025. After our report highlighted the omission ,they understood the rule and immediately fixed the problem. In issue #517 (*Type C.3* of `graphjin` [32]), a developer previously knew little about rules of `multi-module` but learned about it with our help and commented "*Thanks! Spent hours did not find this doc*". Similarly, in issue #1164 (Type C.4) of fq [160], the developers moved a version tag to a new commit because they were unaware that `GOPROXY` caches the original commit and that tags should therefore be immutable—the rule that our report clarified.

Our detection results also identified issues that have remained latent within a codebase for extended periods. In issue #1574 (*Type C.2* of `doctl` [27]), we reported that its upstream project `godo` [26] has used `replace` directive in release versions. The issue was later noticed by the maintainer of the upstream project and found that the `replace` directives were put in place 4

Table 5. Statistics of 449 DM issues reported by HERO and HERO+

Type	Issue reports (Issue report ID, Project name)
Type A	#1647,postgres-operator▲; #249,cells▲; #3840,teleport▲; #315,fathom▲; #1008,factomd▲; #11,webkubectl▲; #115,terway▲; #1020,quorum▲; #50036,cockroach▲; #25105,origin▲; #519,jaeger-client-go▲; #288,RedisShake▲; #475,pgweb▲; #1741,heketi▲; #787,veneur▲; #77,git-chglog▲; #178,vearch▲; #1640,openstorage▲; #97,manifest-tool▲; #82,wave▲; #319,operator-marketplace▲; #4323,bk-cmdb▲; #2922,kiali; #345,go-carbon; #21,gowebsocket; #10,nging; #8,Hands-On-Software-Engineering-with-Golang; #53,kafka-proxy; #456,istio-operator; #48,gke-managed-certs; #23,render; #2488,amazon-ecs-agent; #51,core; #180,standup-raven▲; #93,kubergrunt▲; #1852,metrictank; #141,pupernetes; #222,balena-engine; #491,go-vite; #44,gh-polls; #106,integram; #1068,amazon-ecs-cli; #4835,trafficcontrol; #2786,runtime; #342,presidio; #18383,snapd; #52,sqoop; #94,acyl; #385,dns; #729,bitrise; #18,kube-iptables-tailer; #3460,minishift; #447,mu; #1545,faas; #330,arena; #609,fossa-cli; #1,tepleton; #277,redis-operator; #2962,swarmkit; #72,k8s-spot-rescheduler; #741,open-service-broker-azure; #1007,appsody; #13,nginx-clickhouse; #94,acyl; #534,kubernikus; #125,core; #974,GoSublime; #713,functions; #1293,ansible-service-broker; #658,stork; #21,aliyun-log-jaeger; #209,boleto-api; #411,postgres-exporter; #129,gitkube; #3016,pouch;
Type B.1	#1411,signalfx-agent; #2,findgs; #151,block-explorer; #284,watchman; #256,flamingo-commerce; #3,scifgif; #12,ntcr; #488,benthos; #182,go-geom; #5366,cds; #55,vault-pki-backend-venafi; #37,dataframe-go; #136,dsk; #82,konfigadm; #170,rabbitmq-exporter; #1,foxtrot; #1220,weave; #663,yorc; #1186,blockatlas; #3843,weave; #295,serial-vault; #3970,sensu-go; #208,bosun; #12,vaultenvporter-go; #12,stashvision; #71,isopod; #719,sops; #48,awsu; #23,terraform-provider-pingaccess; #21,pivot; #310,memberlist; #11258,cluster-api;
Type B.2	#114,tomato; #20,kube-cluster-sample; #1,ovpn-tool; #1,cache; #10,rankdb; #4,go-workshops; #1306,neo-go; #2,chat; #232,saferwall; #7,hcunit; #49,examples; #499,cost-model; #1,go-universal-network-adapter; #9215,kyma; #12,rboot; #1,video-stream-recorder; #347,server; #50,CPU-Pooler; #76,honeyaws; #190,envman; #2401,paas-cf; #7978,telegraf; #2395,kubernetes-client; #732,bitrise; #63,albiondata-client; #83,remp; #438,snmpcollector; #27,chrly; #15,pike; #104,service-mesh-extensions; #1512,skygear-server; #69,go-scm; #37,aur-out-of-date; #36,rai; #12559,lotus; #584,sentinel-golang; #234,dcs-bios; #985,assetto-server-manager; #678,louketo-proxy; #770,terraform-provider-libvirt; #1,subs; #31,logrus-influxdb; #61,mlmodelscope; #17,dns; #107,multiaddr; #1,goDistributedCron; #4,field-services; #27,amanar; #20,sailfish; #143,training; #29,airflow-on-k8s-operator; #17,telegraf-lotus; #212,cronsun; #6448,erda; #3748,zadig; #213,gleam;
Type C.1	#34,java-buildpack-memory-calculator; #54,gormt; #162,gocron; #8,generic; #26,go-sessions; #13,keystore-go; #49,go-sdk; #21,gokiteconnect; #2517,hub; #265,cameradar; #309,server; #1638,micro; #317,marketstore; #28,media-sort; #114,mmock; #833,chain33; #3,artifex; #17,accounting; #29,checkmail; #7138,jx; #5,goDoH; #77,gin-admin; #158,gosparkpost; #4,lsleases; #11,bhugo; #13,mcwss; #15,grpc-json-proxy; #5,wifi-password-qr; #2,transcoder; #2,pipe-to-me; #42,restruct; #141,gots; #23,huego; #8,math-engine; #6,iso9660; #6,raft-badger; #27,tenus; #27,go-bitcoind; #7,gotime; #22,ADtoLDAP; #80,lenses-go; #113,STNS; #504,multus-cni; #90,tank; #4118,git-lfs; #203,vale; #25,echo-session; #118,mmark; #481,chirpstack-network-server; #1255,ceph-csi; #284,aliyun-cli; #5268,singularity; #6306,terraform-provider-google; #933,cli; #2305,felix; #501,aws-nuke; #2126,calicoctl; #91,goblin; #3,sparkzstd; #121,email; #24,columnize; #43,nes; #6,ring; #279,goczmq; #239,pongo2; #42,ccli; #644,rqlite; #629,direnv; #581,gost; #181,cloud-game; #313,gedcom; #26,healthcheck; #15,go-web-app; #26,go-corona; #22,license; #2274,gobgp; #1147,go-iost; #9,cuid; #43,jsonrpc; #32,jsonrpc; #90,go-rest-api; #59,go-arty; #4,skl-go; #77,terminal-to-html; #27,gnark; #16,gotypist; #1,Goid; #4593,steampipe; #804,xuperchain; #255,qor; #780,ttn; #334,bblfshd; #333,sealos; #3754,sensu-go; #475,logspout; #103,sdk; #335,gostatsd; #394,goproxy; #23,dque; #72,goffmpeg; #1272,go-algorand; #16381,tidb; #25,hyperfox; #195,vaulted; #561,moira; #990,tidb-binlog; #1747,vpp; #95,hashi-helper; #333,goim; #4,chive; #214,manba; #4,openssl; #22,ynab.go; #21,libgrin; #727,bettercap; #293,sso; #222,linx-server; #306,k8s-prometheus-adapter; #212,go-nebulas; #43,uiprogress; #45,roger; #37,gann; #7,recaptcha; #13,kratos-demo; #1,metrics; #25,echo-session; #226,coze-discord-proxy; #159,checkup;
Type C.2	#32,hackpadfs; #67,nemo-go; #383,identifo; #139,metro2; #417,nginx-kubernetes-gateway; #115,herodot; #341,guardian; #456,network-operator; #1147,kubernetes-nmstate; #1177,azurefile-csi-driver; #2465,operator; #45,kube-events; #671,tile38; #545,ContainerSSH; #95,likecoin-chain; #966,celestia-core; #134,jwt-to-rbac; #1088,idena-go; #240,coil; #13,prometheus-example-app; #410,csi-driver-nfs; #752,metacontroller; #525,tf-controller; #1102,clickhouse-operator; #728,kured; #252,podinfo; #1061,splunk-operator; #6746,eventing; #415,oci-cloud-controller-manager; #3112,dnscontrol; #1574,docth; #2567,listmonk; #1890,zos; #28,embedshim; #483,image-automation-controller; #4225,keda; #11335,kuma; #99,thanos-receive-controller; #41,goNfCollector; #295,fetchd; #833,mindoc; #4,poco; #65,labca; #6,FedLCM; #5355,caddy; #58,cdp-cache; #15,streamingfast-client; #5,ascode; #13,gocat; #1430,arktos; #53,earlybird; #800,minter-go-node; #602,shentu; #871,schemahero; #8872,dapr; #3686,kratos; #7310,jaeger;
Type C.3	#232,CycleTLS; #856,flow-cli; #711,storage; #106,rollbar-go; #27,go-dbf; #137,fhir; #394,goproxy; #16,gq; #138,prom-label-proxy; #8029,deepflow; #716,tdl; #223,system-upgrade-controller; #1961,hive; #41,airflow-client-go; #492,amazon-ecr-credential-helper; #360,grule-rule-engine; #13,yql; #517,graphjin; #3530,aries-framework-go; #11,go-tsne; #294,frameworks; #56,xds; #9,regex; #18,tools; #17,gorm-paginator; #7,go-disk-usage; #79,bchwallet; #8,go-opengraph; #769,catalyst; #60,gengine; #1222,headscale; #1493,skywire; #10490,gardener; #2449,skydive;
Type C.4	#340,test-infra; #117,go-template; #10,jobor; #5,go; #92,wait4it; #57,ethtool; #39,location; #54,whris; #521,mink; #22,merlin-agent; #13,JDC-Monitor; #377,kpt-config-sync; #1456,indexer; #27,gh-grep; #28,refunc; #1366,cilium-cli; #75,container-web-tty; #431,kubegems; #758,radondb-mysql-kubernetes; #73,hola-proxy; #133,tensor; #259,quorum; #62,upgit; #2114,tiup; #188,quickshare; #27,k8spacket; #252,certify; #2005,celo-blockchain; #280,admiral; #926,koperator; #862,layotto; #247,ipfs-search; #4952,sensu-go; #4959,mx-chain-go; #166,IOC-golang; #1147,hoverfly; #2751,testcontainers-go; #1164,fq; #26,note-maps; #20,surf; #1179,oathkeeper; #3246,gotosocial; #835,compliance-operator; #1,gopractise-demo; #253,tkeel; #137,creatly-backend; #5,hellogolang; #209,applicationset-progressive-sync; #2,toolset; #230,application; #771,cluster-api-provider-bringyourownhost; #121,cfrpki; #4,go-ddd; #29,pryrite; #134,vorteil; #1,goprogressbar; #783,open-monitor; #38,ptc-go; #44,objectbox-go; #2360,ziti; #3263,argo-events; #512,faktory; #876,jira-cli; #4828,tracee; #228,aistore; #2525,koordinator; #3054,apptainer;

Status 1: Issues fixed using our suggestions; Status 2: Issues under fixing using our suggestions; Status 3: Issues confirmed, but fixing not decided; Status 4: Issues fixed using other suggestions; Status 5: Issues pending; Issue ID▲: Migration to `Go Modules` conducted (desired); Due to page limit, the detailed information of reported issues is provided on our homepage (**http://www.hero-go.com/**)

Table 6. Statistics of potentially impacted downstream projects for the confirmed DM issues

Type	Type A	Type B.1	Type B.2	Type C.1	Type C.2	Type C.3	Type C.4	Total
Total Potential Affected Downstream	2,495	1,102	9,533	50,931	434	17,438	10,843	92,776
Average Potential Affected Downstream	53.1	50.1	238.3	494.5	11.7	968.8	252.2	299.3

years ago and were finally found with our help, a developer commented "*Thanks so much for the help cleaning this up!*". Similarly another developer commented in issue #2567 (*Type C.2* of `listmonk` [83]) that, "*This was a few years ago. Thanks for bringing this to notice. It's now removed.*"

Feedback on fixing suggestions. To ease the discussion, we divide the 297 DM issues that have been fixed or are under fixing into three categories: (1) 282 taking our highlighted preferred solutions (with minimal impacts to other projects), (2) 13 taking one of our suggestions (impacting some projects), and (3) the remaining two not taking our suggestions.

As an example for category (1), issue #3754 [140] was induced by project `sensu-go`'s [142] `SIV` rule violations. We warned the potential build failures for `sensu-go`'s 89 downstream projects. This was confirmed by developers' comments "*We are aware of this issue, but the way you have summarized it, including the paths forward and impact analyses, is very valuable.*" Although developers were unable to immediately follow `SIV` rules due to internal restrictions, they nevertheless adopted our recommended solution by extracting part of the project code into a new module that complies with `SIV` rules for downstream use. For category (2), the developers did not take our highlighted preferred fixing solutions. Instead, with the information of impacted downstream projects reported in the issue, developers chose to add notes in their projects' documentations to suggest the concerned downstream projects work around potential DM issues by using `replace` directives (*Solution 5*) or hash commit ID (*Solution 8*) (e.g., issues #16381 of `tidb` [132]). For category (3), only developers of the two issues (#3970 of `sensu-go` [141] and #770 of `libvirt` [28]) did not take our fixing suggestions. Still, they actually used other similar libraries for substitution, to avoid possible trouble.

The feedback above indicates that Hero and Hero$^+$ are effective tools for detecting and predicting DM issues in Golang projects, as well as for suggesting appropriate fixing solutions and providing valuable impact analysis. Developers also showed interests in our tool. For example, one developer commented "*I found that you sent many contributions on GitHub for this kind of subjects on many repositories. How do you detect the problems with `Go Modules`? Do you plan to share a tool or something to manage `Go Modules` issues?*" (`ovh/cds`'s [122] issue #5366 [121]). Another commented "*It is a good bot!*" (`TheThingsNetwork`'s issue #780 [155]). Encouraged by such comments, we are planning to release our tool for public use to help build a healthy Golang ecosystem.

Contribution to Golang ecosystem. To evaluate the impact of Hero$^+$ on the broader Go ecosystem, we investigated the downstream dependents of projects in which we identified DM issues. The results, summarized in Table 6, indicate that our tools effectively mitigate significant potential risks throughout the ecosystem.

The DM issues we reported may affect a total of 92,776 downstream projects, with an average of 299.3 projects impacted per issue. Among the various types, *Type C.3* issues have the most extensive impact, affecting an average of 968.8 projects each. Other types also pose substantial risks; for instance, issue #2751, a single *Type C.4* issue in the `testcontainers-go` [154] project, had the potential to affect 8,437 downstream projects. This underscores the importance of Hero$^+$ in mitigating the widespread consequences of these vulnerabilities.

6 Discussions

6.1 Threats to Validity

6.1.1 External Validity. One possible threat is the representativeness of the studied Golang projects and DM issues. To reduce the threat, we selected the top 20,000 projects on GitHub for migration status analysis (RQ1), and randomly chose 500 from the top 1,000 projects to investigate DM issues' characteristics (RQs 3-4). These projects are popular, large-sized, and well-maintained. To further ensure their representativeness beyond popularity, we conducted a systematic categorization of the

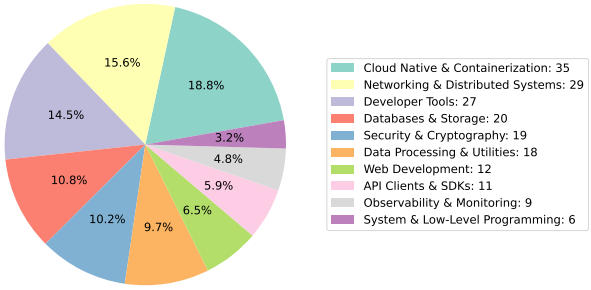

Fig. 16. Distribution of Project Categories for the Analyzed DM Issues.

projects associated with our 256 manually analyzed DM issues. We collected project descriptions and tags on GitHub, utilized an LLM to synthesize categories and classify the projects, and manually verified the results. This analysis confirmed that our dataset is well-distributed across a wide range of critical categories in the Golang ecosystem, as shown in Figure 16. It revealed a strong presence in key areas such as cloud-native infrastructure, popular web frameworks, and developer tools, among others. This demonstrates that our dataset covers a diverse and representative sample.

A related threat concerns the generality of the issue types that Hero$^+$ detects, since they were derived from studying 500 Golang projects. To mitigate this, we used a different set of DM issues to evaluate Hero$^+$ (RQ5) and found that it can detect 97.3% of these issues, which suggests that our findings on issue characteristics are generalizable. Furthermore, Hero$^+$ detected a large number of potential DM issues after analyzing 19,000 Golang projects, many of which have been submitted and confirmed by developers. This further strengthens the generality of the findings in this paper.

Finally, a threat to external validity is whether our findings, derived from open-source projects on GitHub, generalize to industrial projects outside GitHub. This threat is mitigated by two key factors. First, our dataset of popular projects naturally includes many foundational projects maintained by large technology companies such as Google, which reflect industrial best practices and challenges. Second, the fundamental mechanisms of Golang's dependency management are universal to the language itself, not the hosting platform. For instance, even if a company builds a proprietary, self-hosted code repository, it must still conform to the protocols and rules dictated by the Golang toolchain for its projects to be buildable. Therefore, the types of DM issues we identify, their root causes, and the fixing patterns we observe are highly likely to be applicable in industrial settings.

6.1.2 Internal Validity. One possible threat concerns the scope of the Go Modules features investigated (`SIV`, `replace`, `multi-module`, and `GOPROXY`). Our focus on these four features was not predetermined; rather, it was empirically derived from a comprehensive investigation of real-world problems. We conducted a multi-year collection of DM issues, employing rigorous search keywords that yielded 256 distinct issues. Our root-cause analysis revealed that the overwhelming majority of modern, intra-module DM issues stemmed directly from these four features. This data-driven approach ensures our study is grounded in the dominant challenges developers face. Furthermore, these features are fundamental to `Go Modules`. `SIV` and `GOPROXY` features, for example, apply by default to nearly every project using this mode. Our large-scale analysis in Section 3.3 confirms that violations are not isolated, affecting 36.6% of `Go Modules` projects in 2025, which underscores the widespread relevance of our focus. While it is conceivable that other, less common features

could contribute to DM issues, our extensive data collection confirms that our focus on these four features encapsulates the most prevalent and impactful problems.

A separate threat relates to the analysis scope of Hero$^+$. Our analysis is limited to public GitHub repositories, as Hero$^+$ cannot directly analyze private or internal corporate codebases. However, GitHub is the dominant hosting platform, hosting nearly 90% of all Golang projects. Therefore, while this limitation exists, the scenario of missing issues in private repositories is not common within the context of our study. Still, to mitigate this limitation for practical application, Hero$^+$ provides a feature for users to directly upload their projects for local analysis.

Another threat is the potential for false positives of Hero$^+$, which arises from two primary sources related to versioning ambiguity. First, the `GOPATH` mode utilizes the `vendor` directory, which lacks explicit library version declarations. To mitigate this, Hero$^+$ attempts to extract versions by supporting the configuration files of common custom DM tools (e.g., `Dep`, `Glide`). In the absence of such a tool, Hero$^+$ reverts to mimicking the Golang toolchain's own inference mechanisms. Second, we evaluate all projects against the complete set of modern `Go Modules` rules, regardless of their Golang toolchain version. For instance, we flag modifications to released version tags as violations of the `GOPROXY` immutability rule (default since Golang 1.13), which might be perceived as a false positive for a project with older Golang versions. However, this is a deliberate design choice. Due to the Golang toolchain's backward compatibility, a modern project can still depend on such an older library. In this scenario, the modern toolchain will enforce the rule, leading to a build failure from a checksum mismatch. Therefore, flagging this issue at its source identifies a latent conflict that would manifest in a modern build environment.

Finally, our study involves significant manual work, such as identifying and analyzing issue reports. To reduce the risk of human error, three co-authors have cross-validated all manual results for consistency.

6.2 Practical Application of Hero$^+$

Two key factors influence the practical adoption of Hero$^+$: its integration into standard development workflows and its scalability to large projects. We discuss both aspects below.

Integration into Development Workflows. Hero$^+$ can be seamlessly integrated into the daily practices of development teams. Currently deployed as a web-based tool, its functionality can be further exposed via a Web API, enabling integration into CI/CD pipelines (e.g., GitHub Actions) to automatically detect DM issues before code is merged.

Performance and Scalability. Hero$^+$ incorporates a caching mechanism to avoid redundant model reconstruction during analysis. Building a dependency model requires collecting information from multiple sources, such as GitHub and the Go Checksum Database. To reduce repeated data retrieval, dependency models of previously analyzed projects and commonly used upstream projects are stored in the cache. When a project depends on a cached project, Hero$^+$ reuses the existing dependency model instead of rebuilding it. This mechanism improves analysis efficiency when examining a large number of projects.

6.3 Hero$^+$'s Generalizability Beyond the Golang Ecosystem

Two aspects of our methodology are generalizable to the DM issues induced by incompatible library-referencing modes at other ecosystems:

- The scenarios of issue types and their causes: (1) projects in the legacy library-referencing mode depend on projects in the new library-referencing mode, (2) projects in the new mode depend on those in the legacy mode, and (3) projects in the new mode depend on others also in the new mode, can be generalized to analyze similar situations.

- The formulation of issue fixing patterns. The methodology to construct the dependency model by collecting information about its upstream and downstream projects can be adapted to other ecosystems. With the aid of such a dependency model, fixing suggestions can be structurally formulated based on applicable solutions and their potential impacts. The generalization of our methodology needs to consider the unique characteristics of the studied programming languages, since our work focuses only on the Golang ecosystem (one of the most influential and fastest growing open-source ecosystems).

7 Related Work

Software dependency management. Software dependency management is a well-recognized challenge, and a significant body of literature has emerged to address its complexities. A primary focus of existing literature is the challenge of keeping libraries up to date. Specifically, researchers have extensively investigated the practices of upgrading dependency versions to minimize technical lag and ensure freshness [6, 19, 21–23, 89, 112, 163, 176]. Complementing this, others have focused on the complexities of migrating client code to adapt to breaking changes or API evolution in dependent libraries [25, 35, 61, 68, 78, 96, 99, 113, 138, 139]. For instance, Islam et al. [68] provided a comprehensive empirical study on Python library migrations, introducing PyMigTax to classify migration-related code changes and identifying shortcomings in existing tooling.

However, the process of upgrading versions or integrating new libraries frequently introduces dependency conflicts. To mitigate these risks, prior works have developed techniques for detecting and resolving conflicts across languages like JavaScript [125, 131], Java [65, 167–169], C# [93], and Python [161, 166]. Notably, Patra et al. [125] proposed ConflictJS to analyze conflicts in client-side Web libraries. Addressing the algorithmic limitations of package managers, Pinckney et al. [131] introduced Pacsolve, which replaces Npm's greedy resolution strategy with a Max-SMT approach to ensure optimal and flexible dependency installation. In the Maven ecosystem, Wang et al. [167–169] established a systematic approach with tools like DECCA and RIDDLE to assess the severity of conflicts and generate test cases that trigger crashes.

Beyond version conflicts, a growing body of research has focused on the structural integrity and quality of dependency configurations [15, 69], ranging from redundancy to insufficiency. On one end of the spectrum, the accumulation of unused or redundant libraries—known as dependency bloat—has been extensively analyzed [11, 33, 134, 148–150, 159, 170]. Soto-Valero et al. [148–150] developed tools like DEPCLEAN and DEPTRIM to detect and eliminate bloated dependencies in Maven. On the other end, the absence of necessary dependencies or environmental configurations remains a critical cause of build failures [8, 143, 171, 175]. Bezemer et al. [8] highlighted the risks of unspecified dependencies in Make-based systems. Addressing similar issues in C/C++, Wu et al. [171] proposed PackHunter to automate the recovery of missing packages.

Different from these general issues related to conflict resolution, bloat reduction, or build recovery, our studied DM issues stem specifically from incompatible library-referencing modes and the evolving features of `Go Modules`. Ghorbani et al.'s work [37] is closely related to our Hero$^+$; they formally defined inconsistent modular dependencies for Java-9 applications (JPMS) and proposed Darcy to repair them. However, their focus remained on architecture-implementation mapping, whereas our work addresses the systemic clash between Golang's dual management regimes.

Software ecosystem analysis. Prior studies have investigated a broad range of software ecosystems, including `Java`, `JavaScript`, `Golang`, `Ruby`, `Python`, and others. These studies generally fall into three categories: (1) Ecosystem modeling and analysis, which attempts to analyze dependency network structure [7, 18, 24, 62, 82, 91], and retrieve package information across projects [1, 36, 162]. For example, Benelallam et al. [7] constructed the Maven Dependency Graph to study the evolution of `Java` dependencies. (2) Socio-technical theories, investigating community structure [77, 84, 152],

and human factors such as developer behavior [9, 75, 94, 104, 135, 157, 180]. For example, Blincoe et al. [9] proposed coupling references to model technical dependencies between projects, and explored characteristics of open-source or commercial software ecosystems. (3) Diagnosis and monitoring, focusing on the ecosystem's evolution [70, 88, 147, 173, 176], security risks [60, 107, 151, 181], and quality assurance [13, 95, 126, 127, 166]. For instance, Zimmermann et al. [181] investigated how a small number of compromised maintainer accounts could impact the majority of packages.

Researchers have also extensively investigated the cascading effects of upstream projects. Regarding security, researchers have examined how upstream vulnerabilities impact downstream projects [64, 106, 165, 172, 177, 179]. For example, Wu et al. [172] analyzed 50 million invocations in `Maven` to assess upstream vulnerability risks, while Hu et al. [64] conducted a similar lifecycle study for `Golang`. Regarding functionality, other studies focus on how upstream API evolution affects downstream compatibility [12, 71, 72, 92, 116, 178]. For instance, Jayasuriya et al. [71, 72] studied the downstream impact of API changes, and Li et al. [92] performed a large-scale study on API evolution in Golang. To the best of our knowledge, our work is the first attempt to study the health of the Golang ecosystem specifically from the perspective of DM issues.

8 Conclusions and Future Work

In this paper, we conducted a comprehensive study of DM issues in Golang projects caused by Golang's coexisting dependency management modes, which posed significant challenges to the health and sustainability of the ecosystem. Specifically, we investigated the characteristics of 256 DM issues, analyzed their root causes, and identified common fixing solutions. Based on these insights, we proposed detection algorithms integrated with customizable fixing templates to facilitate the systematic and efficient resolution of DM issues. These algorithms were implemented in the Hero$^+$ tool, which demonstrated strong performance in detecting and diagnosing 97.3% issues on a benchmark dataset of 188 issues.

For future work, we plan to extend the capabilities of Hero$^+$ by strategically integrating Large Language Models (LLMs). While the current deterministic, rule-based analysis performed by Hero$^+$ ensures high-quality and certain diagnosis, a task less suited to the probabilistic nature of LLMs, these models can offer powerful complementary capabilities. We identify two key directions for our future work. The first direction is automated patch generation. The structured, detailed diagnostic reports from Hero$^+$ could serve as high-quality, context-rich prompts for an LLM, enabling the automatic generation of complete code patches or pull requests that implement the recommended fixes. This would advance our workflow from automated diagnosis to automated repair. The second direction is the discovery of new issue patterns. As the Golang ecosystem evolves, new types of DM issues may emerge. LLMs could be employed to mine and analyze large corpora of unstructured, natural language data from developer issue reports to identify these emerging patterns, thereby enriching the knowledge base of rule-based tools like Hero$^+$.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant Nos. 92582201 and 62302209) and the Hong Kong SAR Research Grants Council/General Research Fund (Ref No: 16206524). The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

[1] Ahmad Abdellatif, Yi Zeng, Mohamed Elshafei, Emad Shihab, and Weiyi Shang. 2020. Simplifying the Search of Npm Packages. *Information and Software Technology* 126 (Oct. 2020), 106365. doi:10.1016/j.infsof.2020.106365

[2] andrewstuart. 2024. goq. https://github.com/andrewstuart/goq

[3] andrewstuart. 2024. Issue #12 of project goq. https://github.com/andrewstuart/goq/issues/12

[4] Atlassian. 2024. Bitbucket. https://bitbucket.org/

[5] Azure. 2024. Issue #481 of project go-autorest. https://github.com/Azure/go-autorest/issues/481

[6] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Software Engineering* 20, 5 (Oct. 2015), 1275–1317. doi:10.1007/s10664-014-9325-9

[7] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The Maven Dependency Graph: A Temporal Graph-Based Representation of Maven Central. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 344–348. doi:10.1109/MSR.2019.00060

[8] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An Empirical Study of Unspecified Dependencies in Make-Based Build Systems. *Empirical Software Engineering* 22, 6 (Dec. 2017), 3117–3148. doi:10.1007/s10664-017-9510-8

[9] Kelly Blincoe, Francis Harrison, Navpreet Kaur, and Daniela Damian. 2019. Reference Coupling: An Exploration of Inter-Project Technical Dependencies and Their Characteristics within Large Software Ecosystems. *Information and Software Technology* 110 (2019), 174–189. doi:10.1016/j.infsof.2019.03.005

[10] boyter. 2024. Issue #169 of project scc. https://github.com/boyter/scc/issues/169

[11] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event, USA, 135–146. doi:10.1145/3368089.3409738

[12] John Businge, Alexander Serebrenik, and Mark van den Brand. 2012. Survival of Eclipse Third-Party Plug-Ins. In *Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, Riva del Garda, Trento, Italy, 368–377. doi:10.1109/ICSM.2012.6405295

[13] Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. 2024. Understanding Misconfigurations in ROS: An Empirical Study and Current Approaches. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Vienna Austria, 1161–1173. doi:10.1145/3650212.3680350

[14] Canonical. 2024. Launchpad. https://launchpad.net/

[15] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2023. Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1741–1765. doi:10.1109/TSE.2022.3191353

[16] cockroachdb. 2024. apd. https://github.com/cockroachdb/apd

[17] cockroachdb. 2024. Issue #47246 of project cockroach. https://github.com/cockroachdb/cockroach/issues/47246

[18] Filipe Roseiro Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. 2021. An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2457–2470. doi:10.1109/TSE.2019.2952130

[19] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. 2015. Measuring dependency freshness in software systems. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, Florence, Italy, 109–118.

[20] davecgh. 2025. No Longer Go Gettable with Go 1.11 and 'GO111MODULE=on' inside $GOPATH · Issue #93 · Davecgh/Go-Spew. https://github.com/davecgh/go-spew/issues/93

[21] Fernando López de la Mora and Sarah Nadi. 2018. Which library should I use?: a metric-based comparison of software libraries. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, Gothenburg, Sweden, 37–40.

[22] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Madrid, Spain, 404–414.

[23] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Dallas, TX, USA, 2187–2200.

[24] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *Proceedings of the 16th International Conference on Mining Software Repositories*. ACM, Montreal, QC, Canada, 349–359.

[25] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring: Research Articles. *Journal of Software Maintenance and Evolution: Research and Practice* 18, 2 (March 2006), 83–107.

[26] DigitalOcean. 2024. Digitalocean/Godo. https://github.com/digitalocean/godo

[27] digitalocean. 2024. Issue #1574 of project doctl. https://github.com/digitalocean/doctl/issues/1574

[28] dmacvicar. 2024. Issue #770 of project libvirt. https://github.com/dmacvicar/terraform-provider-libvirt/issues/770

[29] docker. 2025. Switch to Go Module by Crazy-Max · Pull Request #4116 · Docker/Cli. https://github.com/docker/cli/pull/4116
[30] Docker/Compose. 2025. Build(Deps): Bump Github.Com/Docker/Docker from 25.0.0-Beta.1+incompatible to 25.0.0-Beta.2+incompatible by Dependabot[Bot] · Pull Request #11268 · Docker/Compose. https://github.com/docker/compose/pull/11268.
[31] dominikh. 2024. Issue #328 of project go-tools. https://github.com/dominikh/go-tools/issues/328
[32] dosco. 2024. Issue #517 of project graphjin. https://github.com/dosco/graphjin/issues/517
[33] Georgios-Petros Drosos, Thodoris Sotiropoulos, Diomidis Spinellis, and Dimitris Mitropoulos. 2024. Bloat beneath Python's Scales: A Fine-Grained Inter-Project Dependency Analysis. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 2584–2607. doi:10.1145/3660821
[34] filebrowser. 2024. Issue #530 of project filebrowser. https://github.com/filebrowser/filebrowser/issues/530
[35] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient static checking of library updates. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Lake Buena Vista, FL, USA, 791–796.
[36] Kai Gao, Weiwei Xu, Wenhao Yang, and Minghui Zhou. 2024. PyRadar: Towards Automatically Retrieving and Validating Source Code Repository Information for PyPI Packages. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 2608–2631. doi:10.1145/3660822
[37] Negar Ghorbani, Joshua Garcia, and Sam Malek. 2019. Detection and repair of architectural inconsistencies in Java. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, Montréal, QC, Canada, 560–571.
[38] GitHub. 2024. GitHub. https://github.com/
[39] github. 2024. github/hub. https://github.com/github/hub
[40] GitHub. 2024. REST API v3 standards. https://developer.github.com/v3/
[41] go-redis. 2024. Issue #1149 of project redis. https://github.com/go-redis/redis/issues/1149
[42] go-redis. 2024. Issue #1151 of project redis. https://github.com/go-redis/redis/issues/1151
[43] go-redis. 2024. redis. https://github.com/go-redis/redis
[44] gofrs. 2024. Issue #61 of project uuid. https://github.com/gofrs/uuid/issues/61
[45] gogs. 2024. gogs. https://github.com/gogs/gogs
[46] gogs. 2024. Issue #5559 of project gogs. https://github.com/gogs/gogs/issues/5559
[47] golang. 2024. Dep. https://github.com/golang/dep
[48] Golang. 2024. Explanations for minimal module compatibility in Go Wiki. https://github.com/golang/go/wiki/Modules
[49] Golang. 2024. Go Checksum. https://proxy.golang.org/
[50] Golang. 2024. Go Modules explained in Golang Wiki. https://github.com/golang/go/wiki/Modules
[51] Golang. 2024. Go Proxy. https://sum.golang.org/
[52] golang. 2024. golang/go. https://github.com/golang/go
[53] Golang. 2024. Import path syntax described in Golang documentation. https://golang.org/cmd/go/#hdr-Import_path_syntax
[54] golang. 2024. Issue #32695 of project golang/go. https://github.com/golang/go/issues/32695
[55] golang-migrate. 2024. Issue #103 of project golang-migrate. https://github.com/golang-migrate/migrate/issues/103
[56] google. 2024. go-cloud. https://github.com/google/go-cloud
[57] google. 2024. Issue #429 of project go-cloud. https://github.com/google/go-cloud/issues/429
[58] googleapis. 2024. Issue #2543 of project google-api-go-client. https://github.com/googleapis/google-api-go-client/issues/2543
[59] GoogleCloudPlatform. 2024. Issue #408 of project microservices-demo. https://github.com/GoogleCloudPlatform/microservices-demo/issues/408
[60] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Kirchberg, Luxembourg, 166–177. doi:10.1109/ASE56229.2023.00135
[61] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, St. Louis, MO, USA, 274–283.
[62] Luisa Hernández and Heitor Costa. 2015. Identifying Similarity of Software in Apache Ecosystem – An Exploratory Study. In *Proceedings of the 2015 12th International Conference on Information Technology - New Generations*. IEEE, Las Vegas, NV, USA, 397–402. doi:10.1109/ITNG.2015.70
[63] Tim Hockin|. 2024. Using Go Workspaces in Kubernetes. https://www.kubernetes.dev/blog/2024/03/19/go-workspaces-in-kubernetes/.
[64] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2024. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon, Portugal, 1–13. doi:10.1145/3597503.3639230

[65] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, effort-aware library version harmonization. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 518–529. doi:10.1145/3368089.3409689

[66] i-love-flamingo. 2024. Issue #256 of project flamingo-commerce. https://github.com/i-love-flamingo/flamingo-commerce/issues/256

[67] IBM. 2024. IBM DevOps Services. https://hub.jazz.net/git

[68] Mohayeminul Islam, Ajay Kumar Jha, Ildar Akhmetov, and Sarah Nadi. 2024. Characterizing Python Library Migrations. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 92–114. doi:10.1145/3643731

[69] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2022. Dependency Smells in JavaScript Projects. *IEEE Transactions on Software Engineering* 48, 10 (Oct. 2022), 3790–3807. doi:10.1109/TSE.2021.3106247

[70] Slinger Jansen. 2014. Measuring the Health of Open Source Software Ecosystems: Beyond the Scope of Project Health. *Information and Software Technology* 56, 11 (2014), 1508–1519. doi:10.1016/j.infsof.2014.04.006

[71] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 1238–1261. doi:10.1145/3643782

[72] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1433–1444. doi:10.1145/3597926.3598147

[73] Jeffail. 2024. Issue #232 of project benthos. https://github.com/Jeffail/benthos/issues/232

[74] Jeffail. 2024. Issue #454 of project benthos. https://github.com/Jeffail/benthos/pull/454

[75] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. 2011. The onion patch: migration in open source ecosystems. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011)*. ACM, Szeged, Hungary, 70–80.

[76] jwplayer. 2024. Issue #9 of project jwplayer. https://github.com/jwplayer/jwplatform-go/issues/9

[77] Jaap Kabbedijk and Slinger Jansen. 2011. Steering insight: An exploration of the ruby software ecosystem. In *Proceedings of the International Conference of Software Business*. Springer, Berlin, Heidelberg, 44–55.

[78] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E Hassan. 2016. Logging library migrations: A case study for the apache software foundation projects. In *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR)*. ACM, Austin, TX, USA, 154–164.

[79] kataras. 2024. Issue #1355 of project iris. https://github.com/kataras/iris/issues/1355

[80] kiali. 2024. commit c453e89. https://github.com/kiali/kiali/commit/c453e89dbd76de161930e2996bdc1303c4d22187

[81] kiali. 2024. Issue #2922 of project kiali. https://github.com/kiali/kiali/issues/2922

[82] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*. IEEE, Buenos Aires, Argentina, 102–112.

[83] knadh. 2025. Drop the Replace Usage in Go.Mod. · Issue #2567 · Knadh/Listmonk. https://github.com/knadh/listmonk/issues/2567.

[84] Sophia Kolak, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. 2020. It Takes a Village to Build a Robot: An Empirical Study of The ROS Ecosystem. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Adelaide, Australia, 430–440. doi:10.1109/ICSME46990.2020.00048

[85] kubernetes. 2024. Issue #20421 of project test-infra. https://github.com/kubernetes/test-infra/issues/20421

[86] Kubernetes. 2024. Kubernetes/Kubernetes: Production-Grade Container Scheduling and Management. https://github.com/kubernetes/kubernetes

[87] kubernetes-sigs. 2024. Issue #1169 of project kubebuilder. https://github.com/kubernetes-sigs/kubebuilder/issues/1169

[88] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, Ali Ouni, and Katsuro Inoue. 2017. An exploratory study on library aging by monitoring client usage in a software ecosystem. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Klagenfurt, Austria, 407–411.

[89] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (Feb. 2018), 384–417. doi:10.1007/s10664-017-9521-5

[90] kyma-project. 2024. Issue #9306 kyma. https://github.com/kyma-project/kyma/issues/9306

[91] Nuttapon Lertwittayatrai, Raula Gaikovina Kula, Saya Onoue, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. 2017. Extracting insights from the topology of the javascript package ecosystem. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference*. IEEE, Nanjing, China, 298–307.

[92] Wenke Li, Feng Wu, Cai Fu, and Fan Zhou. 2023. A Large-Scale Empirical Study on Semantic Versioning in Golang Ecosystem. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering*

(ASE). IEEE, Kirchberg, Luxembourg, 1604–1614. doi:10.1109/ASE56229.2023.00140

[93] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. 2022. Nufix: Escape from NuGet Dependency Maze. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1545–1557. doi:10.1145/3510003.3510118

[94] Xin Liu, Hang Su, Shuo Wang, Xuesong Lu, and Aoying Zhou. 2026. MDJOSC: matching digital talents and job titles in open source communities. *Frontiers of Computer Science* 20 (2026), 2008614–. https://journal.hep.com.cn/fcs/EN/10.1007/s11704-025-50084-x

[95] Xiang-Jun Liu, Ping Yu, and Xiao-Xing Ma. 2024. An Empirical Study on Automated Test Generation Tools for Java: Effectiveness and Challenges. *Journal of Computer Science and Technology* 39, 3 (2024), 715–736. doi:10.1007/s11390-023-1935-5

[96] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, Campobasso, Italy, 106–117.

[97] Masterminds. 2024. Glide. https://github.com/Masterminds/glide

[98] Masterminds. 2024. Issue #1017 of project glide. https://github.com/Masterminds/glide/issues/1017

[99] Stephen McCamant and Michael D Ernst. 2003. Predicting problems caused by component upgrades. In *Proceedings of the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, Helsinki, Finland, 287–296.

[100] mediocregopher. 2024. Issue #141 of project radix. https://github.com/mediocregopher/radix/issues/141

[101] mediocregopher. 2024. radix. https://github.com/mediocregopher/radix

[102] micro. 2024. Issue #278 of project micro. https://github.com/micro/micro/issues/278

[103] microsoft. 2024. microsoft/presidio. https://github.com/microsoft/presidio

[104] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. 2023. "We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco, CA, USA, 1281–1293. doi:10.1145/3611643.3616293

[105] minio. 2024. Issue #10031 of project minio. https://github.com/minio/minio/issues/10031

[106] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *Proceedings of the 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Macao, China, 201–211. doi:10.1109/SANER56733.2023.00028

[107] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. 2014. The bug catalog of the maven ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, Hyderabad, India, 372–375.

[108] Moby. 2021. Build fails due to swarmkit version mismatch when moby is used as a library. GitHub Issue. https://github.com/moby/moby/issues/42939.

[109] moby. 2024. Issue #39302 of project moby. https://github.com/moby/moby/issues/39302

[110] moby. 2024. moby. https://github.com/moby/moby

[111] moby. 2025. Convert to Go Modules (Go.Mod/Go.Sum) · Issue #46761 · Moby/Moby. https://github.com/moby/moby/issues/46761

[112] Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, Thorsten Berger, Steffen Dienst, and Ahmed E. Hassan. 2016. Analyzing Ad Library Updates in Android Apps. *IEEE Software* 33, 2 (March 2016), 74–80. doi:10.1109/MS.2014.81

[113] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. A Study on Behavioral Backward Incompatibilities of Java software Libraries. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Santa Barbara, CA, USA, 215–225.

[114] nicksnyder. 2024. go-i18n. https://github.com/nicksnyder/go-i18n

[115] nicksnyder. 2024. Issue #184 of project go-i18n. https://github.com/nicksnyder/go-i18n/issues/184

[116] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2022. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central: An External and Differentiated Replication Study. *Empirical Software Engineering* 27, 3 (May 2022), 61. doi:10.1007/s10664-021-10052-y

[117] olivere. 2024. Issue #878 of project Elastic. https://github.com/olivere/elastic/issues/878

[118] onsi. 2024. Issue #814 of project ginkgo. https://github.com/onsi/ginkgo/issues/814

[119] openfaas. 2024. Issue #109 of project faasd. https://github.com/openfaas/faasd/issues/109

[120] osrg. 2024. Issue #1848 of project gobgp. https://github.com/osrg/gobgp/issues/1848

[121] ovh. 2024. Issue #5366 of project cds. https://github.com/ovh/cds/issues/5366

[122] ovh. 2024. ovh/cds. https://github.com/ovh/cds

[123] panjf2000. 2024. Issue #49 of project ants. https://github.com/panjf2000/ants/issues/49

[124] panjf2000. 2024. Panjf2000/Ants: Ants Is the Most Powerful and Reliable Pooling Solution for Go. https://github.com/panjf2000/ants

[125] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between Javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 741–751.
[126] Yun Peng, Ruida Hu, Ruoke Wang, Cuiyun Gao, Shuqing Li, and Michael R. Lyu. 2024. Less Is More? An Empirical Study on Configuration Issues in Python PyPI Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon, Portugal, 1–12. doi:10.1145/3597503.3639077
[127] Marc Pichler, Bernhard Dieber, and Martin Pinzger. 2019. Can I Depend on You? Mapping the Dependency and Quality Landscape of ROS Packages. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE, Naples, Italy, 78–85. doi:10.1109/IRC.2019.00020
[128] pierrec. 2024. Issue #39 of project pierrec/lz4. https://github.com/pierrec/lz4/issues/39
[129] pierrec. 2024. lz4. https://github.com/pierrec/lz4
[130] pierrec. 2024. pierrec/lz4. https://github.com/pierrec/lz4
[131] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culpo, and Todd Gamblin. 2023. Flexible and Optimal Dependency Management via Max-SMT. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 1418–1429. doi:10.1109/ICSE48619.2023.00124
[132] pingcap. 2024. Issue #16381 of project tidb. https://github.com/pingcap/tidb/issues/16381
[133] pingcap. 2024. tidb. https://github.com/pingcap/tidb
[134] Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. 2021. The Used, the Bloated, and the Vulnerable: Reducing the Attack Surface of an Industrial Application. In *Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Luxembourg, 555–558. doi:10.1109/ICSME52107.2021.00056
[135] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E. Santosa, Asankhaya Sharma, and David Lo. 2021. Out of Sight, out of Mind? How Vulnerable Dependencies Affect Open-Source Projects. *Empirical Software Engineering* 26, 4 (April 2021), 59. doi:10.1007/s10664-021-09959-3
[136] The Go programming language. 2025. Cmd/Go: Preserve Basic GOPATH Mode Indefinitely. https://github.com/golang/go/issues/60915.
[137] prometheus. 2024. Issue #6048 of project prometheus. https://github.com/prometheus/prometheus/issues/6048
[138] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*. IEEE, Riva del Garda, Trento, Italy, 378–387.
[139] S. Raemaekers, A. van Deursen, and J. Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (July 2017), 140–158. doi:10.1016/j.jss.2016.04.008
[140] sensu. 2024. Issue #3754 of project sensu-go. https://github.com/sensu/sensu-go/issues/3754
[141] sensu. 2024. Issue #3970 of project sensu-go. https://github.com/sensu/sensu-go/issues/3970
[142] sensu. 2024. sensu-go. https://github.com/sensu/sensu-go
[143] Taha Shabani, Noor Nashid, Parsa Alian, and Ali Mesbah. 2025. Dockerfile Flakiness: Characterization and Repair . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1793–1805. doi:10.1109/ICSE55347.2025.00238
[144] shirou. 2024. Issue #663 of project gopsutil. https://github.com/shirou/gopsutil/issues/663
[145] simiotics. 2024. Issue #2 of project shnorky. https://github.com/simiotics/shnorky/issues/2
[146] Sirupsen. 2024. logrus. https://github.com/Sirupsen/logrus
[147] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The emergence of software diversity in maven central. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 333–343.
[148] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens, Greece, 1021–1031. doi:10.1145/3468264.3468589
[149] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *Empirical Software Engineering* 26, 3 (March 2021), 45. doi:10.1007/s10664-020-09914-8
[150] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. 2023. Automatic Specialization of Third-Party Java Dependencies. *IEEE Transactions on Software Engineering* 49, 11 (November 2023), 5027–5045. doi:10.1109/TSE.2023.3324950
[151] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting taint specifications for JavaScript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 198–209. doi:10.1145/3377811.3380390

[152] M. M. Mahbubul Syeed, Klaus Marius Hansen, Imed Hammouda, and Konstantinos Manikas. 2014. Socio-Technical Congruence in the Ruby Ecosystem. In *Proceedings of The International Symposium on Open Collaboration* (Berlin, Germany) *(OpenSym '14)*. Association for Computing Machinery, New York, NY, USA, 1–9. doi:10.1145/2641580.2641586

[153] testcontainers. 2024. Issue #127 of project testcontainers. https://github.com/testcontainers/testcontainers-go/issues/127

[154] testcontainers. 2024. Issue #2751 of project testcontainers-go. https://github.com/testcontainers/testcontainers-go/issues/2751

[155] TheThingsNetwork. 2024. Issue #780 of project TheThingsNetwork. https://github.com/TheThingsNetwork/ttn/issues/780

[156] tomatool. 2024. Issue #114 of project tomatool. https://github.com/tomatool/tomato/issues/114

[157] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 511–522.

[158] turbot. 2025. Missing 'v2' in Module Path · Issue #4593 · Turbot/Steampipe. https://github.com/turbot/steampipe/issues/4593.

[159] H.C. Vázquez, A. Bergel, S. Vidal, J.A. Díaz Pace, and C. Marcos. 2019. Slimming Javascript Applications: An Approach for Removing Unused Functions from Javascript Libraries. *Information and Software Technology* 107 (March 2019), 18–29. doi:10.1016/j.infsof.2018.10.009

[160] wader. 2025. Checksum Mismatch When Trying to Download Fq. · Issue #1164 · Wader/Fq. https://github.com/wader/fq/issues/1164.

[161] Huiyan Wang, Shuguan Liu, Lingyu Zhang, and Chang Xu. 2023. Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco, CA, USA, 198–210. doi:10.1145/3611643.3616264

[162] Shuo Wang, Xinjun Mao, Shuo Yang, Menghan Wu, and Zhang Zhang. 2024. ROS Package Search for Robot Software Development: A Knowledge Graph-Based Approach. *Frontiers of Computer Science* 19, 6 (Dec. 2024), 196320. doi:10.1007/s11704-024-3660-9

[163] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Adelaide, Australia, 35–45. doi:10.1109/ICSME46990.2020.00014

[164] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. 2021. Hero: On the Chaos When PATH Meets Modules. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, Spain, 99–111. doi:10.1109/ICSE43902.2021.00022

[165] Ying Wang, Peng Sun, Lin Pei, Yue Yu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2023. Plumber: Boosting the Propagation of Vulnerability Fixes in the *Npm* Ecosystem. *IEEE Transactions on Software Engineering* 49, 5 (May 2023), 3155–3181. doi:10.1109/TSE.2023.3243262

[166] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: monitoring dependency conflicts for Python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 125–135. doi:10.1145/3377811.3380426

[167] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Lake Buena Vista, FL, USA, 319–330.

[168] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing Chi Cheung. 2019. Could I have a stack trace to examine the dependency conflict issue?. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, Montréal, QC, Canada, 572–583.

[169] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2022. Will Dependency Conflicts Affect My Program's Semantics? *IEEE Transactions on Software Engineering* 48, 7 (July 2022), 2295–2316. doi:10.1109/TSE.2021.3057767

[170] Nimmi Rashinika Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the Npm Ecosystem. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 2632–2655. doi:10.1145/3660823

[171] Rongxin Wu, Zhiling Huang, Zige Tian, Chengpeng Wang, and Xiangyu Zhang. 2025. PackHunter: Recovering Missing Packages for C/C++ Projects. *IEEE Trans. Softw. Eng.* 51, 1 (Jan. 2025), 206–219. doi:10.1109/TSE.2024.3506629

[172] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia, 1046–1058. doi:10.1109/ICSE48619.2023.00095

[173] Chang Xu, Yi Qin, Ping Yu, Chun Cao, and Jian Lv. 2020. Theories and techniques for growing software: paradigm and beyond. *SCIENTIA SINICA Informationis* 50 (2020), 1595–1611. doi:10.1360/SSI-2020-0079

[174] Rafed Muhammad Yasir, Moumita Asad, Asadullah Hill Galib, Kishan Kumar Ganguly, and Md Saeed Siddik. 2019. GodExpo: an automated god structure detection tool for Golang. In *Proceedings of the 3rd International Workshop on Refactoring*. IEEE, Montréal, QC, Canada, 47–50.

[175] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proceedings of the ACM on Software Engineering* 2, FSE (June 2025), 2618–2640. doi:10.1145/3729386

[176] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *Proceedings of the International Conference on Software Reuse*. Springer, Berlin, Heidelberg, 95–110.

[177] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating Persistence of Open-Source Vulnerabilities in Maven Ecosystem. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Kirchberg, Luxembourg, 191–203. doi:10.1109/ASE56229.2023.00058

[178] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2023. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 51, 12 pages. doi:10.1145/3551349.3556956

[179] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. 2023. Compatible Remediation on Vulnerabilities from Third-Party Libraries for Java Projects. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE, Melbourne, VIC, Australia, 2540–2552. doi:10.1109/ICSE48619.2023.00212

[180] Yu-Qian Zhuang, Liang Wang, Ke-Xin Sun, Hong-Yu Kuang, and Xian-Ping Tao. 2025. Understanding Users' Affective States during Issue Resolution in Open Source Software ProjectsGolang. *Journal of Computer Science and Technology* (2025).

[181] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, Santa Clara, CA, USA, 995–1010.