Payment Golden Catalog: A Centralized Dependency Catalog for Secure Dependency Management in Gradle Projects

Cristiano Hahn
iFood
Chapecó, Brazil
cristiano.hahn@ifood.com.br

Danilo Clemente
iFood
Lavras, Brazil
danilo.clemente@ifood.com.br

Rita Oliveira
iFood
Carnaubal, Brazil
rita.oliveira@ifood.com.br

Heitor Costa
Federal University of Lavras
Lavras, Brazil
heitor@ufla.br

Abstract

Maintaining third-party dependencies free of known vulnerabilities is a recurring challenge in software projects, particularly in microservice-oriented environments where each service independently manages its own dependency versions. At iFood's Payments division, internal security tools scan every project and open a Jira ticket for each vulnerable dependency they detect. As the number of projects grew, handling ticket volume on a per-project basis became impractical. This paper presents the Payment Golden Catalog, a centralized Gradle dependency catalog that serves as the single source of approved dependency versions for all Payments projects. It also describes the Golden-Catalog Migrator, an AI agent that automates migrating existing projects to the catalog with a single command. By centralizing dependency management, the long-term effort required for vulnerability remediation was significantly reduced, while the AI agent accelerated its adoption. After the automated migration of 74 out of 92 projects (80.43%) in approximately six months, an 88.61% reduction in open vulnerability tickets was observed. The paper discusses the catalog design, the model-agnostic migration agent, its operational challenges, and the quantitative results.

Keywords

Dependency Management, Security Vulnerabilities, Gradle, Version Catalog, AI Agent, Automated Migration

1 Target Audience

This paper is intended for software engineers and managers working in environments with many microservices, where managing external dependencies is a significant concern. In particular, it addresses organizations that rely on automated vulnerability scanning tools and face a growing volume of remediation tickets as the number of projects and dependencies increases. The content is relevant to teams that use Gradle Build Tool[1] as their build system and seek strategies to centralize dependency governance, reduce exposure to known CVEs (Common Vulnerabilities and Exposures), and automate the migration of existing projects to standardized dependency models. Professionals interested in the practical application of AI (Artificial Intelligence) agents to repetitive engineering tasks may also benefit from this experience report.

[1]https://gradle.org/

2 Presentation Report

2.1 Context

New CVEs are disclosed regularly, and any third-party library can become a security risk once a vulnerability is identified in one of its versions [6]. Prior research has shown that a significant portion of software projects rely on outdated dependencies and that developers are often unaware of vulnerabilities in their dependency trees [4]. In microservice architectures, this problem is amplified: each service typically declares its own dependencies, and the same library may appear in dozens of projects at different versions [1].

iFood[2] is the largest food-delivery platform in Latin America [3] [7]. It employs internal security tools that continuously scan project dependencies against known CVE databases. When a vulnerability is detected, a Jira[3] ticket is automatically created for each affected project-dependency pair and assigned to the responsible team for remediation.

The Payments division, responsible for iFood's financial transactions, maintains numerous Gradle-based microservices developed in parallel by multiple teams. In this context, a single new CVE affecting a widely-used library would result in dozens of tickets being generated simultaneously, one for each project that depends on the affected version. Each team was required to update the dependency individually in their respective projects. This process was repetitive, consumed considerable engineering time, and frequently competed with product development priorities. In addition, without a centralized reference for approved dependency versions, different projects often used different versions of the same library, leading to fragmentation and inconsistent security postures across the division.

2.2 Payment Golden Catalog

To address the challenges described in the previous section, the Payment Golden Catalog was created as a centralized dependency catalog for all Gradle projects within iFood's Payments division. The catalog is built on top of Gradle's native Version Catalog feature [2]. Version Catalogs allow you to declare dependencies and their versions in a centralized TOML (Tom's Obvious, Minimal Language) file (`libs.versions.toml`) that can be referenced across multiple

[2]https://www.ifood.com.br/
[3]https://www.atlassian.com/br/software/jira

projects. Instead of each project declaring its own dependency coordinates and versions directly in `build.gradle.kts`, all projects reference the catalog, ensuring that the same valid versions are used across all projects.

The Payment Golden Catalog is published as a versioned artifact that Payments projects can consume. The curation process works as follows: dependency versions in the catalog are validated by internal security-scanning tools before being included. Only versions without known vulnerabilities are approved and published. When a new CVE is disclosed for a dependency already in the catalog, the affected version is updated in the catalog, and a new artifact version is released. Projects that adopt the catalog only need to update the catalog version to receive all dependency fixes at once. This approach provides three main benefits.

- **Centralized updates:** a vulnerability fix is applied once in the catalog and propagated to all projects that consume it, eliminating the need to update each project individually;
- **Version uniformity:** all projects within the division use the same approved dependency versions, reducing fragmentation and simplifying compatibility analysis;
- **Governance:** the catalog provides full visibility over which dependencies and versions are in use across the division, enabling better control and compliance with security policies.

Figure 1 illustrates the lifecycle of a security vulnerability detection and its simultaneous resolution across multiple projects using the centralized catalog architecture. When a new CVE is disclosed for a widely used dependency (Step 1), instead of raising alerts and requiring manual intervention across dozens of microservices, the security team updates the dependency coordinates to a secure version in the centralized catalog itself (Step 2). A new version of the catalog is published as an artifact to the internal registry (Step 3). To ensure compliance and rapid remediation, projects must use the latest secure version of the catalog. Each microservice's CI/CD (Continuous Integration/Continuous Delivery) pipeline includes validation gates that automatically block new deployments if an outdated catalog version is detected. Consequently, when following projects initiate a deployment, they must pull the latest catalog to propagate the fix. When the continuous security scanning tool runs its next analysis (Step 4), it verifies that the vulnerability has been resolved across all active codebases, automatically closing the corresponding Jira tickets and eliminating manual engineering overhead per repository.

The operational maintenance of the catalog is controlled strictly. Dependency updates are approved via Merge Requests containing proof of vulnerability resolution (e.g., local *Snyk*[4] CLI execution logs). For cases where a secure version causes incompatibilities or temporary exceptions are needed (often with transitive dependencies), the strategy is to segment dependencies by module, allowing the project to import specific functional versions already present in the catalog. At the time of writing, the Payment Golden Catalog is scoped to the Payments division. Its adoption by other iFood divisions is being evaluated as a potential next step.

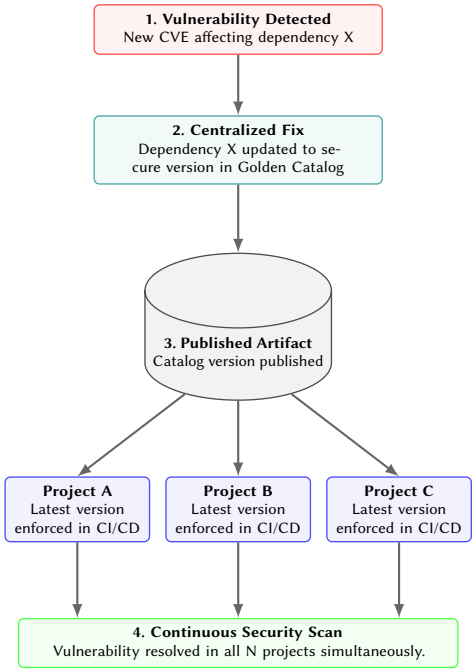

Figure 1: Centralized vulnerability detection, resolution, and pipeline-enforced propagation flow across multiple microservice projects.

2.3 Golden-Catalog Migrator

Migrating existing projects to adopt the Payment Golden Catalog is not a trivial task. Each project has its own `build.gradle.kts` file with dependencies declared in different formats and conventions. While centralized catalogs are a known architectural pattern, their large-scale adoption is often hindered by the complexity of migrating legacy configurations across numerous microservices. The Golden-Catalog Migrator addresses this practical challenge, reducing manual effort. It was developed as a model-agnostic AI agent (allowing developers to use the LLM of their choice, for example, Claude[5] Sonnet or Opus) that migrates a project to the catalog conventions with a single command.

The agent's internal architecture is designed around a structured Chain-of-Thought (CoT) approach [8], in which the migration process is decomposed into explicitly defined phases. Instead of a single-step code rewrite, the agent operates as a stateful reasoner that executes iterative feedback loops, utilizes external information, and maintains high reliability through local validation gates and interactive developer fallback mechanisms. Figure 2 illustrates the internal architecture, decision pipelines, and interactive loops of the Golden-Catalog Migrator. To ensure the robustness of the migration agent, its execution pipeline incorporates several prompting, grounding, and reasoning techniques:

(1) **Legacy build.gradle.kts Processing**. The agent begins by parsing the legacy project to target the *Elimination of Legacy Bad Practices*. It specifically extracts and removes

[4]https://snyk.io/

[5]https://www.anthropic.com/claude

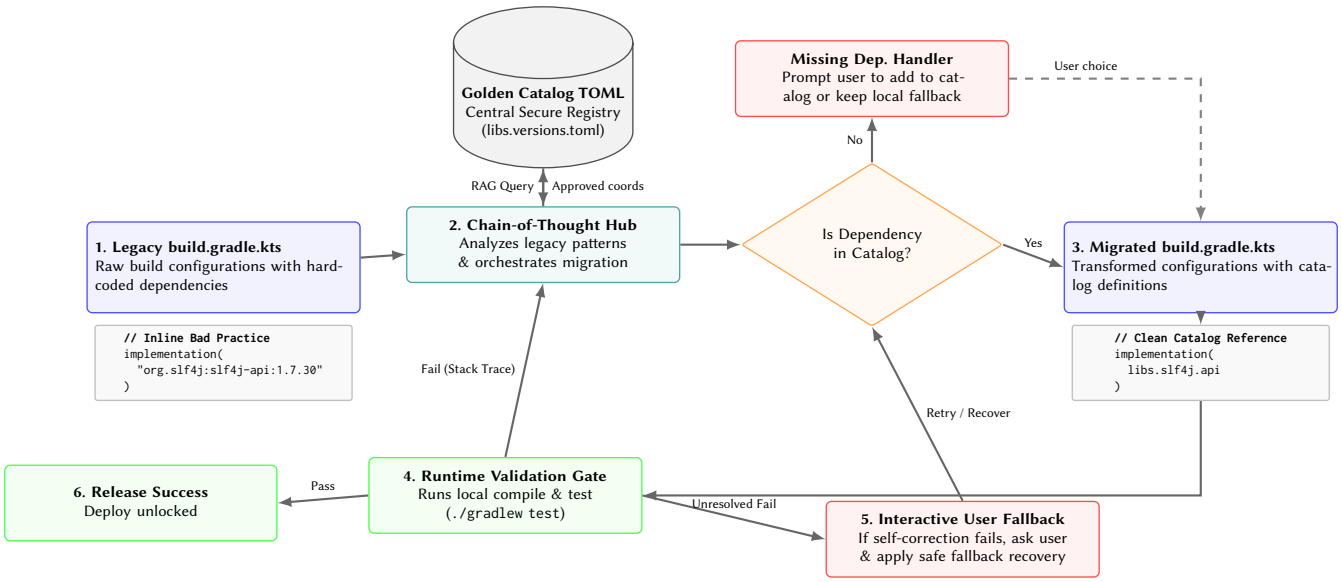

Figure 2: Internal architecture and execution pipeline of the Golden-Catalog Migrator agent, illustrating RAG database lookups, Chain-of-Thought reasoning, bad practice elimination, missing dependency resolution, runtime validation, and interactive fallback loops.

common undesirable dependency declaration patterns from the build files, including hardcoded inline coordinate strings (e.g., `"org.slf4j:slf4j-api:1.7.30"`), inconsistent local variables used for versioning scattered throughout the build files, and ad-hoc repository URLs;

(2) **Chain-of-Thought Hub**. The agent's core reasoning is executed through *Structured Chain-of-Thought Phases* (e.g., intent detection, structural parsing, and transformation planning). To avoid LLM hallucinations about library coordinate strings or version availability, it employs *RAG via Shell Grounding* [5], querying the central Golden Catalog TOML repository via specialized shell skills to ensure that dependencies are mapped only to approved versions. Furthermore, it provides *Unmapped and Missing Dependency Handling*: if a dependency is not in the catalog, it asks the developer to add it. Because the projects of the division follow a well-defined standard and a folder structure, operational challenges and manual interventions during migration were mostly limited to resolving missing dependencies. As the catalog grew, this manual intervention progressively decreased;

(3) **Migrated build.gradle.kts Generation**. Based on the planned transformations, the agent produces the migrated Kotlin DSL build file. The identified legacy bad practices are cleanly replaced by unified, type-safe Version Catalog accessors (e.g., `libs.slf4j.api`);

(4) **Runtime Validation Gate**. The agent performs *Self-Correction via Compiler Logs* by integrating with local build tools (e.g., running `./gradlew test` or `./gradlew compileKotlin`). If the generated build configuration fails the local compilation gate, the agent captures the detailed compiler error logs and stack traces from the terminal, feeding them back into its reasoning hub to rewrite the file and fix the issue in an iterative correction loop;

(5) **Interactive User Fallback**. If test errors persist after multiple self-correction cycles, or if the agent encounters ambiguous Gradle build structures, it employs *Interactive User Prompting and Fallback Recovery*. The agent pauses its execution, displays the current build failure to the developer, and prompts them for guidance. If a path cannot be determined or if the user rejects a transformation, the agent initiates fallback recovery mechanisms, including reverting specific file modifications to the last known stable state, using wildcard dependency configurations, or applying standard preconfigured fallback dependencies;

(6) **Release Success**. Once the migrated code successfully passes all local validation gates without unresolved errors, the deployment is unlocked, and the automated migration of the project is considered complete.

This structured, multi-layered approach makes the agent resilient in enterprise environments, where microservice builds are complex and follow diverse structural conventions.

2.4 Success Indicators

The adoption of the Payment Golden Catalog, combined with the automated migration provided by the Golden-Catalog Migrator, produced measurable improvements in the Payments division's dependency management process. It is important to note that the centralized catalog primarily enables long-term reductions in vulnerability management effort. In contrast, the AI agent was the key enabler, accelerating its adoption within a feasible timeframe. The following indicators summarize the results obtained:

- The number of active vulnerability reports in the Payments division was reduced by 88.61% following the adoption of the Payment Golden Catalog;
- Using the automated migration strategy, 74 out of 92 projects (80.43%) were successfully migrated to the Payment Golden Catalog in approximately six months. The remaining 18 projects are pending due to business prioritization, as the most critical and central services have been migrated.

These results indicate that centralizing dependency management at the division level, supported by an automated migration agent, is an effective approach to reducing the operational overhead associated with vulnerability remediation. The solution enabled better control over external dependencies, improved the division's overall security posture, and significantly reduced the effort required to migrate pre-existing projects to the new model.

Artifact Availability

The artifacts associated with this work, including the Payment Golden Catalog and the Golden-Catalog Migrator, are proprietary internal tools of iFood and are not publicly available. Although the tools themselves are not shared, the intent of this paper is to describe the implementation approach in sufficient detail for software engineers facing similar challenges to adapt it to their own Gradle-based projects. The concepts discussed, such as centralized version catalogs and AI-assisted migration, are not specific to iFood's context and can be replicated using Gradle's native Version Catalog feature or analogous mechanisms available in other dependency managers.

References

[1] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM. doi:10.1145/3196398.3196401

[2] Gradle Inc. 2024. Version Catalogs. https://docs.gradle.org/current/userguide/version_catalogs.html. Accessed: 2026-05-02.

[3] Kevel. 2023. iFood case study: 20x ad revenue with Kevel ad server. https://www.kevel.com/customers/ifood Accessed: 2026-05-20.

[4] 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 (2018), 384–417. doi:10.1007/s10664-017-9521-5

[5] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, Vol. 33. 9459–9474.

[6] Gede Artha Azriadi Prana, Abhishek Sharma, Loh Kang Shar, David 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 (2021). doi:10.1007/s10664-021-09959-3

[7] Sensor Tower. 2024. Fragmented LatAm Food Delivery Market Evolves Amidst Uber's Exit. https://sensortower.com/blog/fragmented-lat-am-food-delivery-market-evolves-amidst-ubers-exit Accessed: 2026-05-20.

[8] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Vol. 35. 24824–24837.