E-Companion for Managing Software Component Quality with Automation: Evidence from Dependabot

Seongkyoon Jeong and Eunae Yoo

EC.1. Algorithm to Locate package.json File

JavaScript repositories contain files and directories at locations customized by developers. Although developers tend to store files in their repositories according to a typical pattern recommended by GitHub and npm, we cannot be certain of the location of a package.json file.

Therefore, we designed an algorithm to obtain the location of a package.json file in the repository for a package (hereafter, referred to as JSON). The primary data provided for the algorithm are the address of the target repository for package i (hereafter, referred to as REPO) and the name of package i (hereafter, referred to as NAME). Often, a package name includes "@" as the first letter and "/" in the middle to mention the series of the package before "/". Similarly, some packages include "-" in their package names for the same purpose. Given this tendency of package naming, we cleaned NAME by removing any special letters and creating a set of word segments split by special letter(s) (hereafter, referred to as NAME_P). We also collected the names of sub-directories under certain directories to locate a package.json file in those sub-directories. Packages tend to be stored in the directory "package". Given that understanding, we collected a set of sub-directory names under "package" (hereafter, referred to as PDIR) and of those under the root directory (hereafter, referred to as RDIR).

Algorithm 1 shows the overall flow of the search routine. The algorithm checks the potential locations of a package.json by the order of the representativeness of the location. It initially checks the root directory. If it fails to locate the file, it moves its search focus to sub-level directories, conditional upon the presence of the "packages" directory. To be specific, until a match appears, the focus keeps moving through the following directories: the directory labeled with the same name as the package, the directory that includes the package name in the full directory name, the directory labeled with a part of the package name, and any directory under the packages directory or the root directory.

We validated our algorithm by investigating historical pull requests for the located package.json files. If a located file is the correct file for the package, the vulnerable dependency should appear in the commit history, especially since even the initial creation of the file leaves a commit history. If not, the location of package.json file was dropped from the set of the identified locations. As a result, we found the locations of the package.json file for 955,081 packages, which account for 97.5% of the packages with valid GitHub links.

Data: repository address (REPO), raw package name (NAME), processed package name (NAME_P), a directory name under REPO/packages (PDIR), a directory name under REPO (RDIR)

Result: package.json address (JSON)

Process:

```
Find REPO/package.json
if JSON = ∅ then
    if REPO/packages exists then
        Find REPO/packages/NAME/package.json;
        if JSON = ∅ then
            Find REPO/packages/PDIR/package.json where NAME is nested in PDIR;
            if JSON = ∅ then
                Find REPO/packages/PDIR/package.json where PDIR is nested in NAME;
                if JSON = ∅ then
                    Find REPO/packages/NAME_P/package.json;
                    if JSON = ∅ then
                        Find REPO/packages/PDIR/NAME/package.json;
                    end
                end
            end
        end
    else
        Find REPO/NAME/package.json;
        if JSON = ∅ then
            Find REPO/RDIR/package.json where NAME is nested in RDIR;
            if JSON = ∅ then
                Find REPO/RDIR/package.json where RDIR is nested in NAME;
                if JSON = ∅ then
                    Find REPO/NAME_P/package.json;
                    if JSON = ∅ then
                        Find REPO/RDIR/NAME/package.json;
                    end
                end
            end
        end
    end
end
```

Algorithm 1: Algorithm to Locate the package.json file

EC.2. Identification of Vulnerable Dependency Instances

When identifying vulnerable dependency instances, the second step involved determining whether a security vulnerability was disclosed for any required dependencies for a package. We referred to the GitHub Advisory Database data to do so. In this process, we accounted for semantic version information. When a package has dependencies, its source code also stipulates their required versions or a range of versions. For example, 1.0.x implies any versions beginning with 1.0 (e.g., 1.0.1, 1.0.2), and ˆ2.5 implies any version between 2.5 but less than 3.0. Details about npm semantic versioning syntax are available at https://docs.npmjs.com/about-semantic-versioning/.

We defined a set of dependency j's versions that meet the version requirement stipulated by package i as V_j^i. Likewise, advisories are often published with a range of versions affected by security vulnerabilities (see Figure 1). We defined a set of j's versions that advisory k described as affected by a vulnerability as V_j^k. To determine whether a package utilizes vulnerable dependencies, we used the *semantic-version* package in Python to inspect whether there is any overlap between V_j^i and V_j^k. If an overlap existed (i.e., $\mathbb{1}[V_j^i \cap V_j^k \neq \emptyset] = 1$), then a vulnerable dependency was identified. That is, we determined direct dependency j for package i contained the security vulnerability disclosed in k and, thus, was vulnerable.

EC.3. Measurement of *ResolutionComplexity*

$ResolutionComplexity^R$ represents the number of package i's dependencies that use a vulnerable dependency as their dependency. This number includes the focal package and other dependencies belonging to the same package. An increase in $ResolutionComplexity^R$ indicates that more dependencies rely on the vulnerable dependency, increasing the complexity of resolving it. This is because developers for the focal package must ensure that the safe version of the vulnerable dependency remains compatible with all components in the focal package's network that rely on the vulnerable dependency. To illustrate how $ResolutionComplexity^R$ is operationalized, Figure EC.1 depicts various configurations of a focal package and its dependency network along with the corresponding calculation of $ResolutionComplexity^R$.

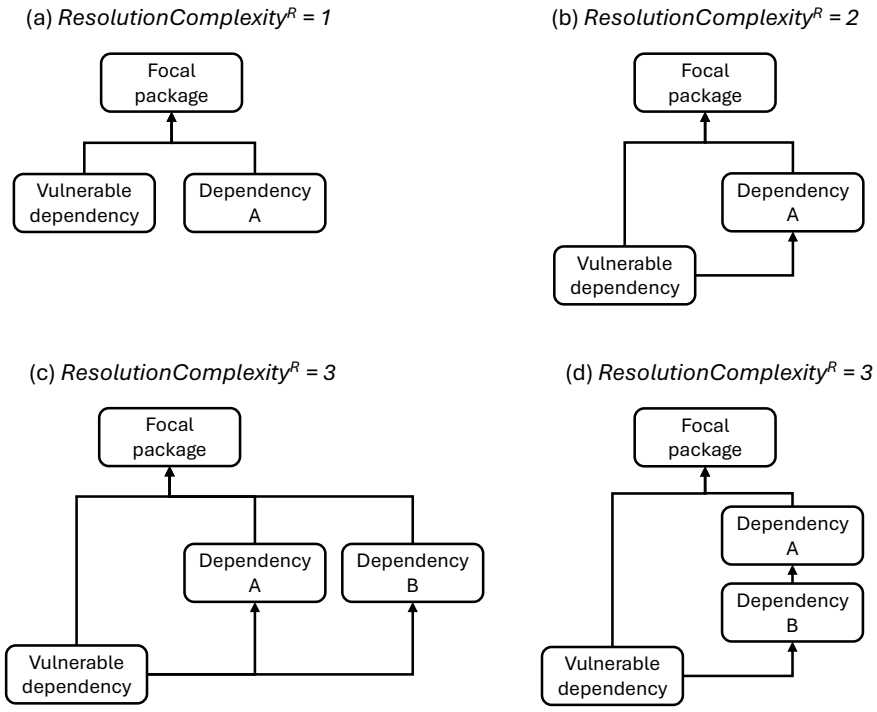


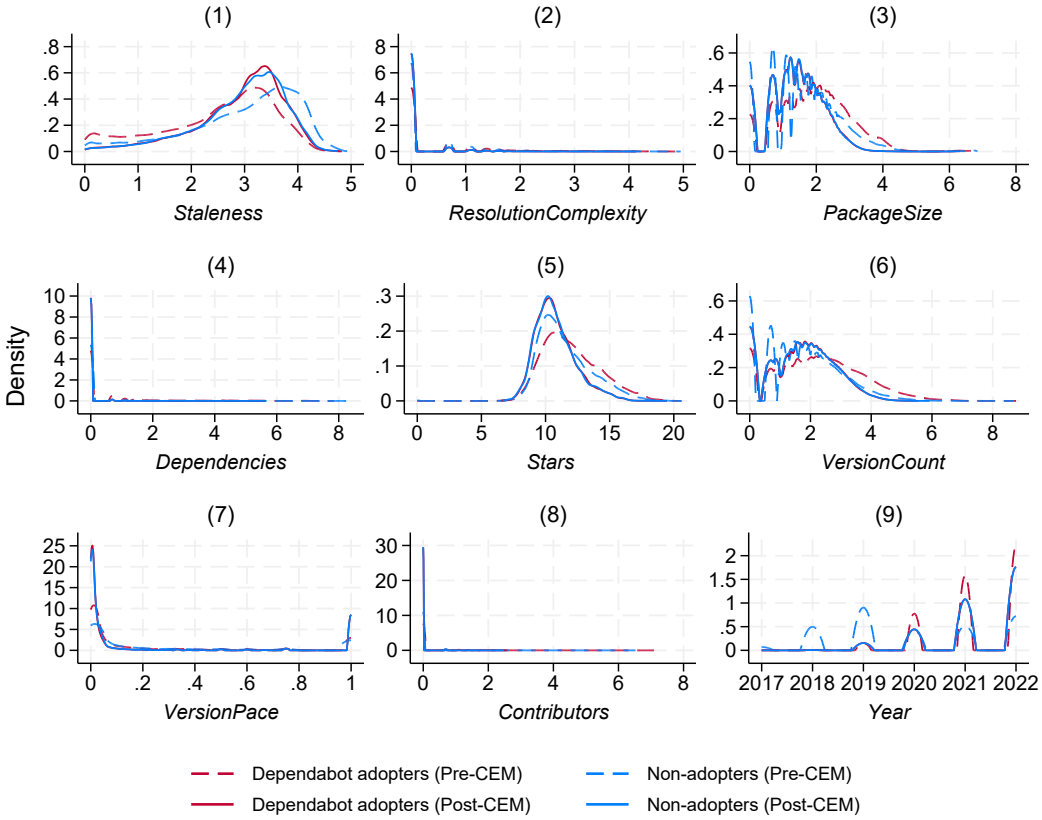
Figure EC.1 Resolution Complexity by the Structure of Dependency

Panel (a) depicts a scenario where the focal package has two independent dependencies: $Vulnerable\,dependency$ and $Dependency\,A$. Resolving this vulnerable dependency instance requires the developers of the focal package to test that updating to the safe version does not impact the compatibility of only one package, which is the focal package itself. Thus, $ResolutionComplexity^R$ is equal to 1. Panel (b) illustrates a case where $Vulnerable\,dependency$ serves as both a direct dependency as well as a transitive dependency through $DependencyA$. Here, resolving this vulnerable dependency instance requires developers to check that updating to a safe version would not negatively impact the focal package nor $Dependency\,A$, resulting in $ResolutionComplexity^R$ to be 2. In Panel (c), $Vulnerable\,dependency$ is also a transitive dependency through $Dependency\,B$. As a result, three interfaces need to be verified. To reflect this, $ResolutionComplexity^R$ is equal to 3. Finally, Panel (d) shows a scenario where $Vulnerable\,dependency$ is not only a direct dependency but also a transitive dependency through $DependencyB$, which serves as a dependency for $Dependency\,A$. Since the functionality of the focal package, $Dependency\,A$, and $Dependency\,B$ depends on $Vulnerable\,dependency$, $ResolutionComplexity^R$ is equal to 3.

A key challenge in constructing $ResolutionComplexity$ is determining the appropriate how far to search within a package's network of dependencies to identify those that are connected to a vulnerable dependency. In general, a dependency with a path length of n from package i supports the functionality of a package through a chain of dependencies with path lengths ranging from $n-1$ to 1. The search process for vulnerable direct dependency j disclosed in advisory k as a transitive dependency with a path length of n to package i in the package's dependency network is as follows: 1) search for dependencies that package i's other dependencies rely on, 2) search for dependencies that the dependencies identified in the previous step rely on, ..., and n) search for cases in which the dependencies identified in the previous step rely on vulnerable dependency j. Since the total number of packages scales to 10^6 in our sample, the combinatorial complexity of mapping dependency networks increases exponentially with each additional path length, reaching 10^{6n} for a path length of n. We address this computational challenge by setting a threshold for the path length between the dependencies considered for the calculation of $ResolutionComplexity$ and package i. Specifically, for each path length n, we calculate the ratio of packages with a path length n dependency relying on a vulnerable dependency to the number of packages directly relying on a vulnerable dependency (i.e., a path length of 1). We limit our analysis to path lengths where this ratio exceeds 5%. Of all packages directly relying on vulnerable dependencies, the proportion of packages relying on vulnerable dependencies at path lengths of 2, 3, 4, and 5 are 26.3%, 16.5%, 10.6%, and 6.7%, respectively. At the level of individual vulnerable dependency instances, the corresponding ratios are 20.0%, 12.1%, 7.6%, and 4.9%. This demonstrates that dependencies decreasingly rely on a vulnerable dependency as the number of path lengths increases. Given this pattern, we set the maximum path length for the calculation of $ResolutionComplexity$ to 5.

EC.4. Variable Distributions by Coarsened Exact Matching

Figure EC.2 shows the distributions of Dependabot adopters and non-adopters across the matching variables. Prior to matching, notable distributional gaps exist between adopters and non-adopters, and CEM substantially reduces these gaps. Moreover, the distribution of the matched observations in the post-CEM sample are similar to most of the distributions from the pre-CEM sample, alleviating concerns regarding the representativeness of the matched sample.



Notes: While Panel (9) depicts the distribution of publication years as a continuous measure, our main model accounts for publication year using year-specific dummy variables. The density values may exceed 1 because the chosen kernel bandwidth can be smaller than 1.

Figure EC.2 Distributions of Matching Variables Before and After CEM

EC.5. Goodness-of-Fit by Baseline Hazard Distribution

To select the best-fitting specification for our parametric proportional hazards model, we demonstrate the goodness-of-fit statistics by the baseline hazard distribution. The following five distributions are tested: Weibull, Gompertz, Log Logistics, Exponential, and Log Normal. Table EC.1 shows the results, sorted by AIC statistics. Overall, the Weibull hazard distribution performs the best. Specifically, this specification has the highest log-likelihood and the lowest AIC and BIC statistics. We, thus, use a Weibull baseline hazard for our main model.

Table EC.1 Goodness-of-Fit by Baseline Hazard Distributions

Hazard distribution	Log-likelihood	AIC	BIC
Weibull	-12,471	24,968	25,076
Exponential	-12,552	25,129	25,229
Gompertz	-12,552	25,131	25,239
Log Logistic	-12,563	25,152	25,260
Log Normal	-12,681	25,388	25,496

EC.6. Robustness Checks with Alternative Functional Forms

First, we evaluate the sensitivity of our results to the shared frailty specification. Table EC.2 shows the results. In column (1), we assume that every vulnerable dependency instance is independent and do not include shared frailty by package. Then, we specify inverse Gaussian distributions for the shared frailty instead of the gamma distribution in column (2). Second, instead of the Weibull function, column (3) assigns the baseline hazard function to follow a Gompertz distribution. Third, we apply the Cox proportional hazards model to mitigate concerns associated with the parametrization of the baseline hazard rate in column (4). Reassuringly, the results of these robustness checks are consistent with the original results presented in column (4) of Table 3. In addition, we run a choice model in which the dependent variable equals 1 if the resolution of a vulnerable dependency instance occurs by the date of data collection and 0 otherwise. Column (5) presents the results, which remain consistent with the survival analysis results.

Table EC.2 Robustness Check with Other Functional Forms

	(1) Independent	(2) Inverse Gaussian	(3) Gompertz	(4) Cox	(5) Logit
Dependabot	0.636^{***}	1.029^{***}	1.083^{***}	0.625^{***}	0.729^{***}
	(0.044)	(0.067)	(0.071)	(0.044)	(0.050)
ResolutionComplexity	-0.303^{**}	-0.271	-0.074	-0.293^{**}	-0.243^{*}
	(0.126)	(0.173)	(0.184)	(0.125)	(0.140)
Staleness	-0.689^{***}	-1.115^{***}	-1.287^{***}	-0.686^{***}	-0.815^{***}
	(0.022)	(0.037)	(0.042)	(0.022)	(0.026)
PackageSize	0.010	0.008	0.012	0.010	0.008
	(0.015)	(0.023)	(0.025)	(0.015)	(0.017)
Dependencies	-0.050	-0.107^{**}	-0.138^{***}	-0.051^{*}	-0.053
	(0.031)	(0.046)	(0.050)	(0.031)	(0.034)
Stars	0.411^{***}	0.740^{***}	0.987^{***}	0.398^{***}	0.633^{***}
	(0.067)	(0.078)	(0.123)	(0.068)	(0.074)
VersionCount	0.101^{***}	0.181^{***}	0.181^{***}	0.101^{***}	0.128^{***}
	(0.021)	(0.032)	(0.035)	(0.021)	(0.023)
VersionPace	-0.545^{***}	-0.804^{***}	-0.765^{***}	-0.545^{***}	-0.598^{***}
	(0.077)	(0.108)	(0.109)	(0.076)	(0.082)
Contributors	0.593^{***}	1.161^{***}	1.426^{***}	0.581^{***}	0.733^{***}
	(0.140)	(0.270)	(0.412)	(0.140)	(0.203)
Severity	-0.106^{***}	-0.065	-0.025	-0.110^{***}	-0.157^{***}
	(0.032)	(0.043)	(0.046)	(0.032)	(0.037)
Constant	-1.798^{***}	-3.159^{*}	-4.845^{***}		3.265^{**}
	(0.687)	(1.650)	(1.406)		(1.269)
Observations	30,328	30,328	30,328	30,328	30,328
Log-likelihood	-13,222	-12,388	-12,486	-28,666	-8,061
Year FE	Yes	Yes	Yes	Yes	Yes
Shared frailty	No	Yes	Yes	No	No

Notes: Robust standard errors clustered at the package level in parentheses; * $p<0.10$, ** $p<0.05$, *** $p<0.01$

EC.7. Robustness Checks with Package-Level Fixed Effects

While the shared frailty model used in our main analysis mitigates concerns about unobserved package-level heterogeneity, it relies on distributional assumptions at the package level. Therefore, we assess the robustness of our findings to alternative model specifications related to package-level fixed effects. Table EC.3 shows the results.

Table EC.3 Robustness Check with Package Fixed Effect

Model	(1)	(2)	(3)	(4)
Model description	First Instance	Stratified CPH	FE Logit	Stratified CPH
Dependabot	0.728^{***}	0.754^{***}	1.093^{**}	0.779^{**}
	(0.067)	(0.203)	(0.452)	(0.335)
ResolutionComplexity	0.118	-0.515	-2.890^{**}	-1.328
	(0.265)	(0.503)	(1.205)	(0.851)
Staleness	-0.663^{***}	0.171	-0.708	0.122
	(0.049)	(0.121)	(0.444)	(0.289)
PackageSize	-0.029	-0.172	-0.757	0.027
	(0.023)	(0.308)	(0.959)	(0.742)
Dependencies	-0.125^{***}	0.133	-8.344	-3.007
	(0.045)	(0.781)	(5.664)	(3.760)
Stars	0.654^{***}	1.464^{**}	0.274	1.935
	(0.096)	(0.589)	(1.130)	(1.305)
VersionCount	0.224^{***}	-0.603	-9.885^{**}	-8.298^{***}
	(0.032)	(0.384)	(3.886)	(2.598)
VersionPace	-0.620^{***}	0.152	16.857^{**}	8.731
	(0.095)	(0.504)	(7.424)	(14.939)
Contributors	-2.126			
	(1.388)			
Severity	-0.131^{***}	0.103	0.156	0.149
	(0.047)	(0.077)	(0.223)	(0.155)
Constant	-2.230^{**}			
	(0.886)			
Observations	8,498	30,328	842	842
Log-likelihood	-4,958	-756	-99	-210
Year FE	Yes	Yes	Yes	Yes
Package FE	No	Yes	Yes	Yes

Notes: Robust standard errors clustered at the package level in parentheses; * $p<0.10$, ** $p<0.05$, *** $p<0.01$

Column (1) of presents results from a sample of 8,498 observations that represent only the first vulnerable dependency instance for each package. This approach eliminates the need for a shared frailty term. To implement this approach, we identified the first vulnerable dependency instances and applied the matching approach outlined in Section 5.1, yielding a total of 8,978 observations. Column (2) reports results from a model where the baseline hazard rate is stratified at the package level. This analysis employs a Cox proportional hazards (CPH) model, which is computationally feasible due to the absence of a parameterized baseline hazard function. Incorporating the additional 22,022 package-level non-linear terms required for a parametric function would otherwise render the computation infeasible. In this specification, the variable *Contributors* is dropped due to a lack of within-package variation. In column (3), we display results from a logit regression model with package fixed effects. Here, the binary dependent variable indicates whether a vulnerable dependency instance was resolved by our data collection time. Including the package fixed

effect leads to the exclusion of any observations from packages where all or none of the vulnerable dependency instances have been resolved. Due to the low within-package variation in the dependent variable, the sample size is reduced to 842 observations. Lastly, column (4) applies the stratified CPH model as in column (2), but uses the same sample as column (3) to ensure consistency across various specifications. Overall, all models consistently show a positive and statistically significant effect of Dependabot adoption on reducing time-to-resolution, reinforcing our main findings.

EC.8. Sensitivity Checks on Matching Variables

We assess the sensitivity of the CEM procedure by sequentially dropping each matching variable and re-estimating Eq. 1. Table EC.4 reports the results. Columns (1)–(9) correspond to specifications in which one matching variable is excluded at a time. Across all specifications, the estimated effect of Dependabot adoption remains positive and statistically significant, with magnitudes comparable to the baseline. These findings indicate that our results are not primarily driven by any single matching variable and reinforce the robustness of the main analysis.

Table EC.4 Sensitivity Checks on Matching Variables

	(1)	(2)	(3)
Dependabot	1.044***	0.908***	0.883***
	(0.040)	(0.053)	(0.045)
Observations	75,688	40,088	54,630
Log-likelihood	-31,440	-16,576	-24,174
Dropped matching variable	*Staleness*	*ResolutionComplexity*	*PackageSize*
Control variables	Yes	Yes	Yes
Year FE	Yes	Yes	Yes
Shared frailty	Yes	Yes	Yes
	(4)	(5)	(6)
Dependabot	0.831***	0.883***	0.900***
	(0.049)	(0.046)	(0.044)
Observations	40,326	52,512	56,986
Log-likelihood	-19,501	-23,119	-25,677
Dropped matching variable	*Dependencies*	*Stars*	*VersionCount*
Control variables	Yes	Yes	Yes
Year FE	Yes	Yes	Yes
Shared frailty	Yes	Yes	Yes
	(7)	(8)	(9)
Dependabot	0.642***	0.970***	0.809***
	(0.037)	(0.059)	(0.046)
Observations	48,308	33,274	63,490
Log-likelihood	-21,125	-13,862	-32,800
Dropped matching variable	*VersionPace*	*Severity*	Year dummies
Control variables	Yes	Yes	Yes
Year FE	Yes	Yes	Yes
Shared frailty	Yes	Yes	Yes

Notes: Results on control variables are available upon request; robust standard errors clustered at the package level in parentheses; * $p < 0.10$, ** $p < 0.05$, *** $p < 0.01$

EC.9. Robustness Checks with Alternative Matching Methods

We run a set of robustness checks using other matching techniques to address potential selection on Dependabot adoption. Table EC.5 reports the results.

Table EC.5 Robustness Checks with Alternative Matching Methods

Matching method	(1) CEM Weighted	(2) PSM 1-Nearest	(3) PSM 2-Nearest	(4) PSM 3-Nearest
Dependabot	0.541^{***}	0.875^{***}	0.540^{***}	0.535^{***}
	(0.052)	(0.031)	(0.029)	(0.032)
ResolutionComplexity	-0.070	-0.156^{***}	-0.169^{***}	-0.168^{***}
	(0.116)	(0.019)	(0.017)	(0.017)
Staleness	-0.722^{***}	-0.715^{***}	-0.504^{***}	-0.499^{***}
	(0.024)	(0.012)	(0.011)	(0.011)
PackageSize	0.010	0.023^{***}	0.008	0.011^{*}
	(0.018)	(0.007)	(0.006)	(0.006)
Dependencies	-0.040	-0.048^{***}	0.029^{**}	0.031^{**}
	(0.034)	(0.016)	(0.013)	(0.014)
Stars	0.490^{***}	0.339^{***}	0.229^{***}	0.225^{***}
	(0.068)	(0.010)	(0.007)	(0.007)
VersionCount	0.130^{***}	0.064^{***}	-0.009	-0.009
	(0.029)	(0.011)	(0.008)	(0.008)
VersionPace	-0.650^{***}	-0.355^{***}	-0.216^{***}	-0.220^{***}
	(0.084)	(0.038)	(0.034)	(0.035)
Contributors	0.519^{***}	0.226^{***}	0.156^{***}	0.147^{***}
	(0.143)	(0.033)	(0.021)	(0.021)
Severity	-0.091^{**}	-0.038^{**}	-0.122^{***}	-0.123^{***}
	(0.046)	(0.015)	(0.015)	(0.015)
Constant	-2.330^{***}	-4.561^{***}	-3.879^{***}	-3.798^{***}
	(0.721)	(0.411)	(0.251)	(0.347)
Observations	58,963	103,455	91,584	87,430
Log-likelihood	-24,447	-65,069	-66,247	-64,375
Year dummies	Yes	Yes	Yes	Yes
Shared frailty	No	Yes	No	No

Notes: Robust standard errors clustered at the package level in parentheses; * $p<0.10$, ** $p<0.05$, *** $p<0.01$

In column (1), we first check the robustness of the main finding using a matched sample obtained without the k2k restriction. Since matching weights differ within packages, we cannot include shared frailty by package. We also conduct matching using propensity score matching (PSM), which is another commonly used matching method. Column (2) uses a sample generated by the nearest neighbor PSM with the same specification as the original model (i.e., Weibull hazard model with gamma shared frailty). Columns (3) and (4) expand our test of the sensitivity of our results to matching with PSM by using samples based on PSM with two and three nearest neighbors with weights, respectively.

The results for the models in Table EC.5 show that the impact of Dependabot adoption on time-to-resolution is positive and statistically significant across all the different matching techniques. Interestingly, the effect size of *Dependabot* from the nearest neighbor PSM sample (0.875, $p<0.01$) is very close to that of our original sample derived from CEM with the k2k restriction (0.916, $p<0.01$). The consistency of the results demonstrates that our findings, especially related to the effect of Dependabot adoption, are not sensitive to the choice of matching method.

EC.10. Robustness Checks with Control Function Approach

As discussed in Section 5.1, the adoption of Dependabot is endogenously determined by package developers. Our analysis addresses this endogeneity issue using coarsened exact matching. However, although the matching approach effectively deals with a significant portion of endogenous factors, it is possible that some remaining endogenous issues (e.g., omitted variable bias from unobserved heterogeneity) may have affected the precision of our estimates.

To further validate our finding with respect to the effect of Dependabot, we check the robustness of our finding using an alternative method for addressing endogeneity: the control function (CF) approach (Wooldridge 2013). This approach, also known as two-stage residual inclusion approach, adjusts for endogeneity using endogenous variations in the variable of interest. The CF approach is particularly suitable for dealing with non-linear regression models (Wooldridge 2013). In particular, previous studies suggest using the CF approach to address endogeneity issues in survival analysis instead of the conventional two-stage least squares regression approach (2SLS) (Terza et al. 2008). Thus, we employ the CF method.

The CF approach requires an instrumental variable (IV) in the estimation. We endogenize *Dependabot* by adopting the commonly practiced group-level instrumental variable approach (Hausman 1996, Petrin and Train 2010). To be specific, we formulate an instrumental variable, *AdoptionRate*, as the average level of Dependabot adoption for the other packages that use the same vulnerable dependency at the point of the advisory publication. The instrumental variable for package i and advisory k can be represented as follows:

$$AdoptionRate_{ik} = \frac{\sum_{\tilde{i} \in I_k, \tilde{i} \neq i} Dependabot_{\tilde{i}k}}{n(I_k) - 1}, \tag{EC.1}$$

where I_k is a set of package i that uses vulnerable dependency j associated with advisory k. In this calculation, we drop 135 vulnerable dependency instances (0.04% of the full sample) where only a single instance is found in the group (i.e., $n(I_k) = 1$) because we cannot obtain the aforementioned average for those instances; thus, the sample size decreases to 369,287.

The key rationale for the formulation of our instrumental variable, which follows the approach practiced in previous studies (e.g., Jeong and Lee 2022, Zou et al. 2023), is that packages with the same dependency share similar properties and, accordingly, are likely to be exposed to a similar level of external pressure regarding Dependabot adoption. This instrumental variable meets the exclusion restriction condition (Wooldridge 2013). Because this variable reflects only other packages with the same dependency and advisory, it does not directly explain the resolution of the focal vulnerable dependency instance while it predicts the focal package's Dependabot adoption at the time of advisory publication. In addition, this instrument also meets the instrument relevancy condition (Wooldridge 2013).

Table EC.6 Robustness Check with Control Function Approach

Model specification	(1) First-stage	(2) Second-stage
Dependabot		2.774^{***}
		(0.128)
ResolutionComplexity	-0.001	-0.081^{***}
	(0.001)	(0.014)
Staleness	-0.052^{***}	-0.675^{***}
	(0.001)	(0.011)
PackageSize	0.002^{***}	0.047^{***}
	(0.000)	(0.005)
Dependencies	0.007^{***}	-0.058^{***}
	(0.001)	(0.012)
Stars	0.070^{***}	0.214^{***}
	(0.001)	(0.012)
VersionCount	0.026^{***}	0.053^{***}
	(0.000)	(0.009)
VersionPace	-0.057^{***}	-0.226^{***}
	(0.002)	(0.027)
Contributors	-0.018^{***}	0.221^{***}
	(0.001)	(0.026)
Severity	0.006^{***}	-0.058^{***}
	(0.001)	(0.009)
AdoptionRate	0.700^{***}	
	(0.007)	
Residuals		-2.466^{***}
		(0.130)
Constant	0.056^{***}	-5.368^{***}
	(0.006)	(0.085)
Observations	369,287	369,287
Log-likelihood	-134,010	-163,631
R-squared	0.248	N/A
Year FE	Yes	Yes
Shared frailty	N/A	Yes

Notes: Robust standard errors clustered at the package level in parentheses; * $p<0.10$, ** $p<0.05$, *** $p<0.01$

Our first stage results (column (1) in Table EC.6), where the dependent variable is *Dependabot*, show a statistically significant coefficient for *AdoptionRate* ($p<0.01$). The Anderson canonical correlation LM statistic is 62574.9 ($p<0.01$), and Cragg-Donald Wald F-statistic is 75339.2. These results indicate that the instrumental variable is neither under-identified nor weakly-identified. The second-stage results in column (2) of Table EC.6 are consistent with our main analysis (i.e., Table 3). The coefficient of *Dependabot* is positive and statistically significant ($p<0.01$), corroborating our findings. This ensures the validity of the analysis based on coarsened exact matching, suggesting that our matching approach sufficiently addresses endogeneity issues and that the remaining endogeneity issues after matching are fairly minimal.

EC.11. Placebo Test on the Effect of Dependabot Adoption

We run a placebo test to check for the possibility of any spurious results associated with Dependabot adoption. If the estimated effect of Dependabot adoption truly occurs because of actual Dependabot adoption (hereafter, we refer to this case as *actual assignment*), similar effects should not appear when adoption is randomly reassigned. Following previous studies (Bertrand et al. 2004, Wang et al. 2023), we randomly assign a placebo adoption date to each package that adopted Dependabot. The placebo dates are drawn from the interval between the package's first observed adoption date

and the last observable date in the dataset. Hereafter, we refer to this case as *placebo assignment*. Using the placebo adoption date, we determine each package's Dependabot adoption status for each vulnerable dependency instance. We set *Dependabot* equal to 1 if the placebo adoption timing precedes the advisory publication date for that vulnerable dependency instance and 0 otherwise. Because a single package can experience multiple vulnerable dependency instances with different advisory publication dates, the same placebo adoption date can produce different adoption-status values across instances belonging to the same package.

Next, using the placebo assignment and the corresponding value of *Dependabot*, we estimate its coefficient as well as the moderation coefficient for the interaction term with *Staleness* using the full sample (i.e., Eq. 1 and 2). We repeat this process 50 times, following previous studies that run a placebo test (Bertrand et al. 2004, Wang et al. 2023).

Panel (a) in Figure EC.3 shows the results of the placebo test. The coefficients are sorted by size along the horizontal axis and include 95% confidence intervals. The coefficient under the actual assignment (i.e., from Equation 1) is also included for comparison with the coefficients derived from the placebo assignment. The results indicate that the effect of *Dependabot* with the placebo assignment is generally not significant at the 95% level. Moreover, the effect size itself is notably different from the effect size under the actual assignment.

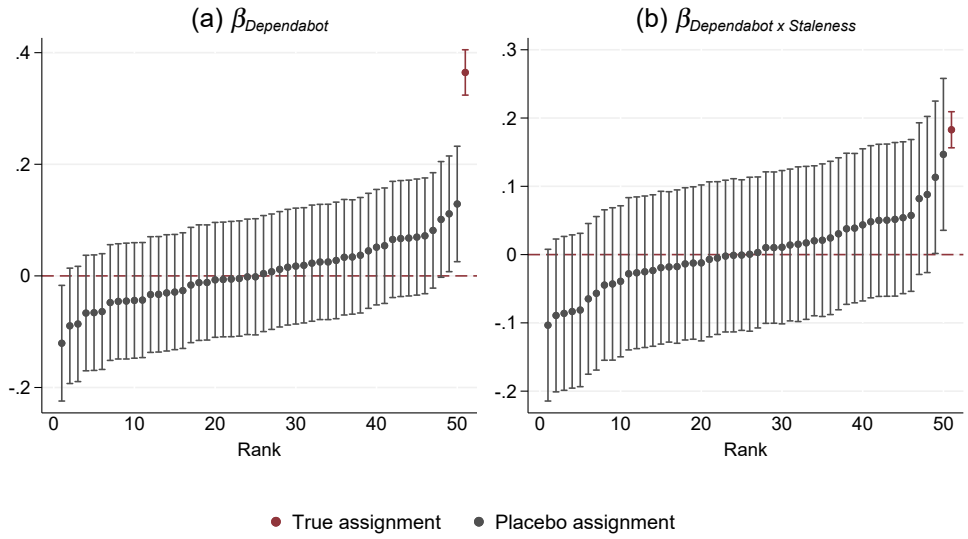


Figure EC.3 Results of Placebo Test

Likewise, the placebo test results in Panel (b) indicates that the moderating effect of Dependabot adoption with the staleness of the package is likely to appear only when actual Dependabot adoption status is assigned to instances. Consequently, we conclude that our findings are likely to result from the true values of Dependabot adoption. Overall, these results bolster our findings.

EC.12. Alternative Explanation

An alternative explanation for the observed impact of Dependabot on time-to-resolution comes from the endogenous adoption of the tool. Developers with a heightened sensitivity to software component security may be more inclined to use Dependabot for their packages. Such a self-selection process may result in the spurious effect of Dependabot adoption. In other words, security-conscious developers may inherently tend to be more responsive to resolving vulnerabilities in their packages' dependency chains and, simultaneously, possess a stronger preference for using Dependabot given its expected benefits for dependency security. As such, the identified positive effect of Dependabot adoption on how quickly vulnerable dependency instances are resolved may merely reflect this inherent sensitivity instead of the causal effect of the adoption itself.

In addition to our primary approach to this issue, discussed in Section 5.1, we additionally validate our findings using the disclosure of a notably significant vulnerable dependency that worked as a "wake-up" call across many industries. In December 2021, the disclosure of a critical security vulnerability in the Apache Log4j package elicited intense concern within the software industry. Log4j is extensively used as a dependency by numerous enterprise-level products, including those by Microsoft, Google, Oracle, and IBM. Due to its widespread integration, the vulnerability was estimated to compromise the security of hundreds of millions of devices (Henriquez 2022). Consequently, the disclosure of the Log4j vulnerability sparked an immediate response by government and business organizations to patch vulnerable versions, and it was considered the most significant vulnerability event in history. As such, the Log4j vulnerability served as a wake-up call for the software industry to prioritize software component security and stay vigilant in identifying and resolving security vulnerabilities in dependencies (Tung 2021).

Given the impact of the Log4j vulnerability, we expect that developers would be more sensitive to vulnerable dependency instances after its disclosure than before. If developers' sensitivity truly matters for how quickly they resolve vulnerable dependency instances, we should be able to observe changes in the effect of Dependabot over time. To make a fair comparison, we use the subsample of vulnerable dependency instances within ± 1 year from the Log4j vulnerability disclosure and test whether the effect of Dependabot adoption becomes different by the disclosure. Table EC.7 shows the results. Column (1) presents the results that validate the generalizability of our main finding in this subsample. The positively significant effect of Dependabot is also present (0.945, $p < 0.01$). Next, the results shown in column (2) indicate the interaction term between *Dependabot* and the indicator for whether a vulnerable dependency instance took place after the Log4j vulnerability disclosure (i.e., $PostLog4j$) is not statistically significant (0.141, $p > 0.1$). This finding indicates that the effect of Dependabot on the resolution of vulnerable dependency instances does not change after the Log4j vulnerability disclosure. Hence, the alternative explanation related to

developers' sensitivity to software security appears to be negligible, corroborating our key result that Dependabot indeed improves the prompt and effective response to vulnerabilities in dependencies. Finally, column (3) shows the results from excluding observations since November 2021 to check the robustness of our findings to potential right-censoring. We find consistent results with our main analysis (0.952, $p < 0.01$).

Table EC.7 Robustness Checks for Alternative Explanations

Model description	(1) Post-2020	(2) Interaction with Post-log4j	(3) Pre Oct. 2021
Dependabot	0.945^{***}	0.873^{***}	0.952^{***}
	(0.081)	(0.107)	(0.075)
Dependabot × *PostLog4j*		0.141	
		(0.138)	
ResolutionComplexity	0.054	0.056	-0.440^{*}
	(0.184)	(0.184)	(0.226)
Staleness	-1.309^{***}	-1.309^{***}	-0.965^{***}
	(0.050)	(0.050)	(0.048)
PackageSize	−0.024	−0.024	0.046^{*}
	(0.027)	(0.027)	(0.026)
Dependencies	-0.123^{**}	-0.123^{**}	-0.127^{**}
	(0.056)	(0.056)	(0.051)
Stars	0.907^{***}	0.909^{***}	0.894^{***}
	(0.127)	(0.127)	(0.121)
VersionCount	0.154^{***}	0.154^{***}	0.191^{***}
	(0.037)	(0.037)	(0.036)
VersionPace	-0.815^{***}	-0.814^{***}	-0.786^{***}
	(0.126)	(0.126)	(0.115)
Contributors	1.669^{***}	1.670^{***}	0.544
	(0.422)	(0.423)	(0.366)
Severity	-0.127^{**}	-0.125^{**}	-0.236^{***}
	(0.057)	(0.057)	(0.052)
Constant	-3.075^{***}	-3.043^{***}	-3.480^{**}
	(0.381)	(0.382)	(1.370)
Observations	25,000	25,000	13,811
Log-likelihood	-8,267	-8,267	-8,167
Year FE	Yes	Yes	Yes
Shared frailty	Yes	Yes	Yes

Notes: Robust standard errors clustered at the package level in parentheses; * $p < 0.10$, ** $p < 0.05$, *** $p < 0.01$

EC.13. Conditional Effects by the Level of Staleness

To better understand how Dependabot mitigates staleness-related delays, we relax the log-linearity assumption on *Stalensss* and instead model it using indicators for six-month staleness bins. The hazard function is specified as:

$$h_{ijk}(t|\alpha_i, X) = h_0(t)\alpha_i \exp\left(\begin{array}{c}\beta_0 + \beta_1 Dependabot_{ijk} \\ + \sum_{s=1}^{S} \beta_{2s}\, Dependabot_{ijk} \times \mathbf{1}[Staleness_{ijk} \in \mathcal{B}_s] \\ + \sum_{s=1}^{S} \gamma_s\, \mathbf{1}[Staleness_{ijk} \in \mathcal{B}_s] + \boldsymbol{\gamma}\mathbf{W}_{ijk} + \eta_k\end{array}\right), \tag{EC.2}$$

where $s = 1, \ldots, S$ indexes each six-month staleness bin, and $\mathcal{B}_s$ denotes the s-th six-month interval of the continuous staleness value ($\mathcal{B}_1 = (0.5, 1]$ years, $\mathcal{B}_2 = (1, 1.5]$ years, ...). The 0–6 month bin is omitted as the reference category. The terms γ_s capture the main effect of staleness at each bin, and the terms $\beta 2s$ capture the attenuating effect of Dependabot adoption at each bin. All other notation follows Equation 2. Table EC.8 reports the full coefficient estimates.

Table EC.8 Conditional Effects of Dependabot Adoption by Staleness Bin

	(1)
Dependabot=1	0.569***
	(0.115)
Staleness Bin=1	−1.006***
	(0.129)
Staleness Bin=2	−1.640***
	(0.141)
Staleness Bin=3	−2.213***
	(0.161)
Staleness Bin=4	−2.599***
	(0.183)
Staleness Bin=5	−2.951***
	(0.209)
Staleness Bin=6	−3.142***
	(0.278)
Staleness Bin=7	−4.172***
	(0.446)
Staleness Bin=8	−3.705***
	(0.548)
Staleness Bin=9	−4.510***
	(0.760)
Staleness Bin=10	−5.090***
	(1.019)
Dependabot=1 × *Staleness Bin*=1	0.349**
	(0.165)
Dependabot=1 × *Staleness Bin*=2	0.278
	(0.182)
Dependabot=1 × *Staleness Bin*=3	0.161
	(0.203)
Dependabot=1 × *Staleness Bin*=4	0.464**
	(0.229)
Dependabot=1 × *Staleness Bin*=5	0.791***
	(0.258)
Dependabot=1 × *Staleness Bin*=6	1.163***
	(0.327)
Dependabot=1 × *Staleness Bin*=7	1.331***
	(0.505)
Dependabot=1 × *Staleness Bin*=8	1.492**
	(0.602)
Dependabot=1 × *Staleness Bin*=9	1.489*
	(0.862)
Dependabot=1 × *Staleness Bin*=10	2.369**
	(1.069)
ResolutionComplexity	−0.147
	(0.168)
PackageSize	0.020
	(0.022)
Dependencies	−0.097**
	(0.044)
Stars	0.838***
	(0.104)
VersionCount	0.188***
	(0.030)
VersionPace	−0.698***
	(0.098)
Contributors	1.151***
	(0.335)
Severity	−0.036
	(0.042)
Constant	−4.841***
	(1.395)
Observations	30,328
Log-likelihood	-12,382
Year FE	Yes
Shared frailty	Yes

Standard errors in parentheses; * $p < 0.10$, ** $p < 0.05$, *** $p < 0.01$

References

Bertrand, Marianne, Esther Duflo, Sendhil Mullainathan. 2004. How Much Should We Trust Differences-In-Differences Estimates?*. *The Quarterly Journal of Economics* **119**(1) 249–275.

Hausman, Jerry A. 1996. Valuation of new goods under perfect and imperfect competition. *The Economics of New Goods*. University of Chicago Press, 207–248.

Henriquez, Maria. 2022. Log4Shell a huge wake-up call for 95% of security leaders. URL `https://www.securitymagazine.com/articles/97517-log4shell-a-huge-wake-up-call-for-95-of-security-leaders`.

Jeong, Seongkyoon, Jaeseok Lee. 2022. Environment and energy? the impact of environmental management systems on energy efficiency. *Manufacturing & Service Operations Management* **24**(3) 1311–1328.

Petrin, Amil, Kenneth Train. 2010. A control function approach to endogeneity in consumer choice models. *Journal of Marketing Research* **47**(1) 3–13.

Terza, Joseph V, Anirban Basu, Paul J Rathouz. 2008. Two-stage residual inclusion estimation: addressing endogeneity in health econometric modeling. *Journal of Health Economics* **27**(3) 531–543.

Tung, Liam. 2021. US warns Log4j flaw puts hundreds of millions of devices at risk. URL `https://www.zdnet.com/article/log4j-flaw-puts-hundreds-of-millions-of-devices-at-risk-says-us-cybersecurity-agency/`.

Wang, Lina, Elliot Rabinovich, Harish Guda. 2023. An analysis of operating efficiency and policy implications in last-mile transportation following Amazon's integration. *Journal of Operations Management* **69**(1) 9–35.

Wooldridge, Jeffrey M. 2013. *Introductory Econometrics: a Modern Approach*. South-Western, Cengage Learning.

Zou, Fan, Yan Dong, Sining Song, Manus Rungtusanatham. 2023. Product recalls and supply base innovation. *Manufacturing & Service Operations Management* **25**(5) 1931–1946.