Open Source Migration Practices and Patterns

NUWAN DIAS
DEPUTY CTO, WSO2

CONTENTS

The software landscape has undergone a significant shift in recent years, witnessing the burgeoning popularity of **open-source software (OSS)**. This collaborative development model — where the source code is freely accessible and modifiable — has revolutionized how software is built, distributed, and utilized. From powering critical infrastructure to fueling cutting-edge innovations, OSS has become a significant force in the modern world.

This Refcard delves into the key characteristics and advantages of open-source software, exploring its impact on various aspects of software development and deployment. We'll examine the benefits of cost-effectiveness, flexibility, and security offered by OSS, while also acknowledging potential challenges and best practices for responsible adoption.

BENEFITS OF MIGRATING TO OPEN SOURCE

In today's technology landscape, organizations are constantly seeking ways to optimize efficiency, agility, and innovation. Open source is a collaborative development model, offering a powerful alternative to traditional, closed-source software. Migrating to open source presents a wealth of benefits, from significant cost savings and enhanced security to increased flexibility and a vibrant community for support and collaboration. In this section, we discuss some of the key benefits of migrating to open source.

COST-EFFECTIVENESS

Open-source software allows your organization to be significantly cost-effective. Specifically, OSS:

- Eliminates licensing fees and allows customization, reducing reliance on expensive vendors.
- Often has lower hardware requirements, saving on infrastructure costs.

FLEXIBILITY

Open-source software offers organizations a flexible and adaptable foundation for their technology needs, empowering them to take control, solve problems faster, and scale efficiently.

OSS makes organizations much more flexible by enabling:

- Customizable code, which allows organizations to tailor the software to their specific needs.
- Seamless integration with other systems, thanks to open standards.
- Freedom from vendor lock-in, which fosters better negotiating power and agility.

SECURITY AND TRANSPARENCY

In today's digital world, security and transparency are paramount for any organization. OSS steps up to the challenge by offering a unique

instaclustr
by NetApp

HARNESS THE POWER OF OPEN SOURCE

Build and scale reliable applications faster

Expert consulting and global support

- Migration to open source
- Open source strategies
- Heath checks
- Technology kickstarter package
- Solution architecture
- Operational review

Managed platform for open source technologies

Deploy, manage, and monitor all components of your data layer and related infrastructure




perspective on both. It helps organizations become more secure and transparent by:

- **"Many Eyes" principle:** Large developer communities lead to quicker identification and resolution of vulnerabilities.
- **Community-driven security:** Proactive vulnerability tracking and knowledge sharing.
- **Customizable security:** Customizable security and code transparency enable organizations to take control of their security posture.

INNOVATION

Another benefit of adopting open source is the level of innovation opportunities it offers for organizations. These opportunities open doors for higher levels of creativity in solutions offered by organizations. Collaboration within the open-source community fuels continuous learning and innovation, keeping organizations at the forefront by:

- Access to a vast pool of talent and expertise through the global open-source community.
- Rapid prototyping and experimentation with new technologies and ideas.
- A shared codebase, which serves as a springboard for innovation.
- Fostering a culture of openness and knowledge sharing, leading to groundbreaking innovations.

While open source benefits organizations in the various forms mentioned above, it also benefits organizations to navigate through key challenges that the modern software industry faces, which will be further explored below.

COMMON SOFTWARE CHALLENGES ADDRESSED BY OPEN SOURCE

Modern enterprises face a lot of challenges with regards to proprietary software. These come in the flavors of cost, sociocultural and geographic impacts, and market conditions that make it challenging to run a viable business.

UNPREDICTABLE COSTS

Cloud services can have complex pricing and unexpected usage spikes, hindering budgeting and planning. OSS offers cost stability with transparent pricing and customizable solutions.

SOCIOCULTURAL AND GEOGRAPHIC IMPACTS

Reliance on proprietary software can be disrupted by sociocultural and/or geographic requirements. OSS offers:

- **Sovereignty:** Control over deployment and maintenance, reducing reliance on specific countries.
- **Continuity:** Ability to "fork" projects to maintain critical software, even if original projects are sanctioned.
- **Vendor independence:** Freedom from vendor limitations through globally accessible code.
- **Community support:** Access to a global support network even if vendor support is unavailable.

MARKET CONDITIONS

Open source helps navigate economic challenges through:

- **Cost control:** Eliminates licensing fees and promotes flexibility, freeing resources for innovation.
- **Agility and adaptability:** Allows rapid adaptation to changing market demands.
- **Resilience:** Reduces dependencies on specific vendors or regions, ensuring technology availability during disruptions.

OSS provides a valuable alternative for organizations seeking cost-effective, flexible, and secure software solutions in today's unpredictable environment.

CORE PRACTICES FOR OPEN-SOURCE MIGRATIONS

In this section, we will be looking at the key practices to consider when migrating to using open-source software. We will be evaluating data migration strategies, open-source software dependencies, security vulnerabilities, and open-source licensing.

MIGRATING TO OPEN-SOURCE DATABASES

Migrating to open-source databases presents a compelling option for organizations seeking several key benefits such as reduced costs, enhanced security, scalability, flexibility, and so on.

SELECTING THE RIGHT DATABASE

Selecting the proper database for your application needs to be the first and most important decision to make before considering a migration to an open-source database. There are several factors to consider when selecting the right database.

Table 1: Key considerations for selecting the right database

KEY CONSIDERATIONS	DETAILS
Nature of processed data	Ensure the database adheres to these ramifications: • Structured data: Postgres, MySQL • Document-oriented data: MongoDB, CouchDB • Column-oriented data: Apache Cassandra, Apache HBase • Key-value pairs: Redis, Memcached • Graph data: Neo4j, Dgraph (for social media sites, recommendation systems, etc.)

TABLE 1 CONTINUES ON NEXT PAGE

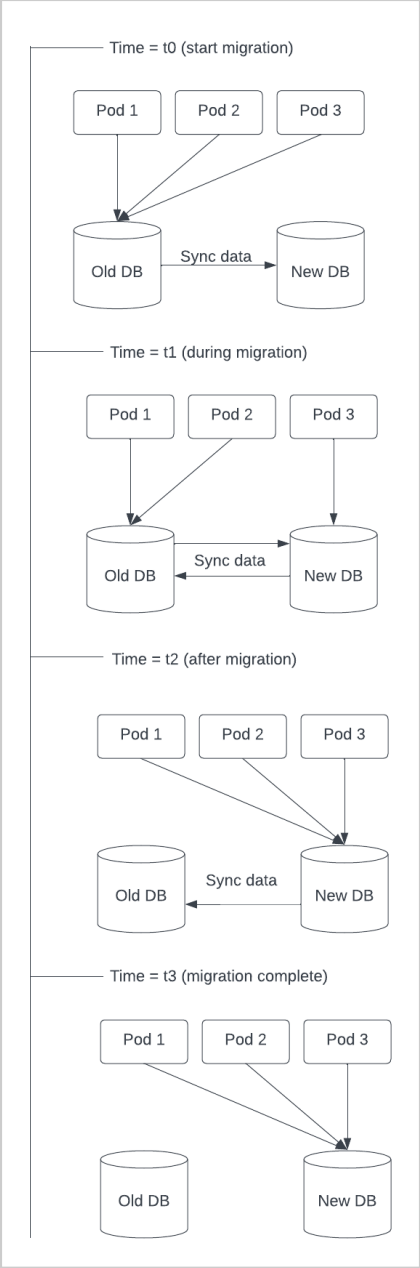
KEY CONSIDERATIONS	DETAILS
Read-write operations	Consider your application's read-write needs. Relational databases might not be ideal for massive write rates due to their slower nature, a side effect of the data being structural. Explore alternatives like Redis or Cassandra for high-speed write requirements.
Replication requirements	Evaluate data replication requirements before migrating to an open-source database. The number and type of replicas (read-write vs. read-only) are crucial factors: • All replicas read-write: Cassandra might be suitable. • A single read-write replica: Postgres or MySQL with master-slave replication can suffice.
Consistency models	Open-source databases offer different consistency models: • Eventual consistency (Cassandra): Acceptable delay for data synchronization across all nodes. Prioritizes availability and scale. • Strong consistency (Postgres/MySQL): Synchronous replication ensures data consistency across nodes. Prioritizes data integrity and accuracy.
Data requirements	• Data availability: Choose Cassandra if availability is crucial. • Data accuracy: Opt for Postgres/MySQL for high accuracy requirements (e.g., financial transactions).
Master-Master replication	Postgres and MySQL offer global read-write (master-master replication) but require careful data design to avoid conflicts.
Back-ups and recovery	Regardless of the database type, understand the back-up and recovery process for your open-source solution. Consider acquiring or developing the necessary skills. If you are using a database for caching purposes (e.g., Redis) consider the following: • Caching: Consider if you need to persist the cache when using the database primarily for caching purposes. • In-memory: Running your database completely in-memory can be sufficient and cost-effective for specific systems that rely solely on in-memory data. • Disk-level guarantees: Some systems require data to persist on disk to function correctly, even after the database restarts and loses in-memory data.

MIGRATING DATA SUCCESSFULLY

If you are already using a database in production and thinking of switching to an open-source database, it is important to thoroughly think of how you are going to manage the cut-over to the new system (database). Failing to think through data migration strategies could result in challenges such as data loss and corruption, application downtimes, performance issues, and so on. Whatever data migration strategy is put in place, it is critically important to have a roll-back plan in case something goes wrong.

A typical data migration process is depicted in Figure 1 below.

Figure 1: Typical data migration process



For high availability and scalability, an application usually consists of more than one running application instance. These application instances are depicted as Pod 1, Pod 2, and Pod 3 in Figure 1.

The migration process is described in greater detail below:

1. **t0**: The migration starts by copying all data from the old database to the new database.

2. **t1**: Once data copying is complete, start switching the application pods to point to the new database.
   - Start the bi-directional data sync just before the application pods start switching. This way, new data created by application pods that are still pointing to the old database (Pod 1 and Pod 2) is copied to the new database, and data created in the new database is copied to the old.
3. **t2**: Once all the application pods have switched to the new database, start tests to ensure the migration is successful.
   - Keep syncing data from the new database to the old. This ensures that the application pods can switch back (roll-back) to the old database without any data loss in case something goes wrong.
4. **t3**: Once all tests are complete, stop the data sync and remove the old database.

CORE DATA MIGRATION PRACTICES

In this subsection, we will consider different data migration strategies, applications, and suitability.

CORE PRACTICES FOR DATA MIGRATION	DESCRIPTION
Big Bang	This involves migrating all data from one database to another in a single, coordinated operation. A "rip-and-replace" approach, the old system is swapped entirely for the new one in a short timeframe. Per Figure 1, this approach would mean that you go directly from t0 to t3, thus skipping t1 and t2. While Big Bang is the simplest and easiest for migrating data, however, it is only practical in situations where the time difference between t3 and t0 is considerably low and the application can experience downtime during that period. A slight deviation of this can be to run the application in read-only mode during t0 and t3. This way, no new data is created during the migration period, resulting in partial uptime while still eliminating data sync (t1 and t2) processes.
Trickle migration	Also known as *phased migration* or *incremental migration*, this is an approach that breaks down the data migration process into smaller, manageable stages. Instead of transferring all data in one go like the Big Bang method, trickle migration transfers data in multiple phases, often spread across a longer timeframe. If the data in your organization is complex and large, trickle migration can help reduce or eliminate downtimes by phasing out the migration process. This approach requires being able to run the application in small silos where certain parts of the application are working with the new database while some other parts are working with the old.
Hybrid migration	Hybrid migration offers the best of both Big Bang and trickle migration techniques. Hybrid migration is divided into two distinct phases: • **Phase 1**: Initial Bulk Transfer (Big Bang element) — A significant portion of the data (often the most critical or time-sensitive) is transferred in a single operation, similar to the Big Bang method. This phase aims to quickly migrate the core data and establish basic functionality on the new system. • **Phase 2**: Trickle Migration of Remaining Data — The remaining data, often less critical or easier to manage, is migrated in smaller, controlled batches, similar to the trickle approach. This phase allows for more granular control, validation, and adjustments during the migration process.
Change Data Capture (CDC)	The CDC-based data migration approach is what we discussed in Figure 1. For syncing data from one database to the other, a CDC approach is used. Compared to Big Bang or trickle, the CDC approach is complicated and requires careful planning. The advantage it provides over the rest is that it offers a seamless, zero-downtime migration possibility. For scenarios where the application availability is crucial, a CDC-based approach is the best suited.
Golden record migration	This approach focuses on migrating the most accurate and complete representation of each data point from the source system to the target system. This approach leverages the principles of master data management (MDM), specifically the concept of a "golden record." The application needs to be aware of both the source and target databases at the same time. When the application is requested for a particular record, it fetches it from the source database, transforms it, and inserts it into the target database. From there onwards, the application looks up that particular record in the target database only. This approach is suitable when the organization needs to cleanse and/or reformat data in the source database.

TABLE 1 CONTINUES ON NEXT PAGE

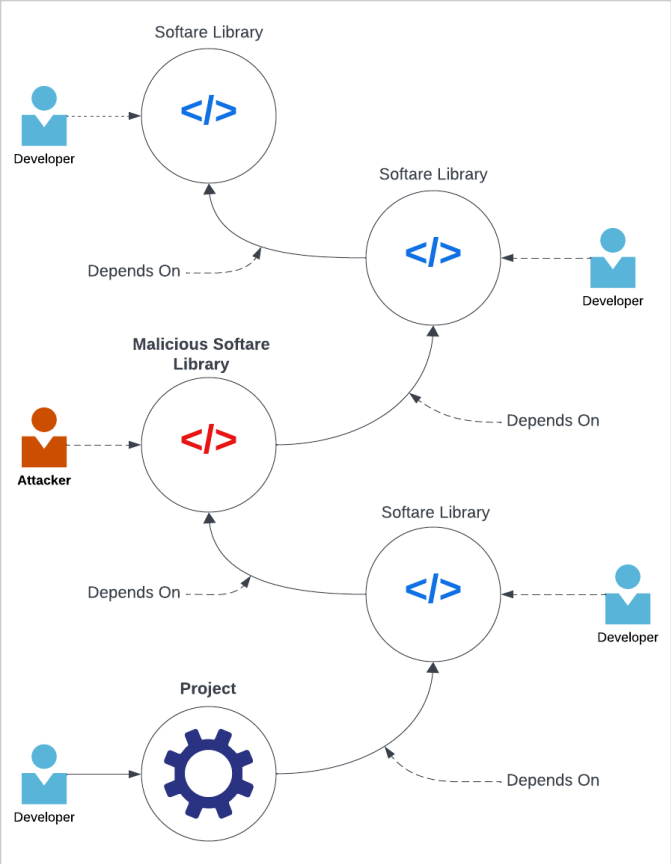
MANAGING SOFTWARE DEPENDENCIES

In today's world, no software stands alone. The software we build will always depend upon components built and maintained by others. Similarly, the software we produce will be used by someone else either via integration or inclusion (libraries). This is called the **software supply chain**.

While publicly accessible for anyone to contribute, software repositories like npm or Maven Central for JavaScript and Java packages respectively can harbor malicious code despite scanning efforts. These attacks, where harmful code is snuck into public repositories, are known as *software supply chain attacks*.

As illustrated in Figure 2, the project depends on a software library, which, in turn, depends on other software libraries, out of which one of them is malicious. This makes the project vulnerable to the attack through the malicious software dependency.

Figure 2: Illustration of a software supply chain attack



MAINTENANCE

Using outdated software is a key contributor to technical debt. It also increases your risk of being attacked since old software may contain vulnerabilities that have been identified and fixed in newer versions. To be kept safe from outdated software, you always need to be up to date.

DEPENDENCY MANAGEMENT, TOOLS, AND TECHNIQUES

Managing dependencies in a software project is no trivial task. Any reasonably sized software project will have over a hundred software dependencies. Keeping track of all these dependencies and making sure they are up to date can only be done through automations. These automations should scan your code for dependencies and verify if they are up to date. The automations should run on a periodic basis (e.g., once a week) and check for the availability of newer versions of your dependencies. In the event it finds newer versions, it should either auto-upgrade or notify the relevant stakeholders. Here are two popular free and open-source tools that you can use for this purpose:

- Dependabot: From GitHub, this is a well known open-source tool that helps keep your dependencies up to date. It automates dependency management within your repositories and helps developers identify outdated dependencies, suggest updates, and even create pull requests to incorporate the latest versions. This not only simplifies maintenance but also enhances the security and functionality of your projects.
- Renovate: This is a similar open-source tool that also assists in keeping dependencies up to date. However, it supports a wider range of package managers and programming languages compared to Dependabot.

Both tools can be integrated within CI/CD pipelines. This makes it possible to integrate them in your build processes so that you don't have to attend to it manually. Apart from these two, there are many other tools as well.

CORE PRACTICES FOR DEPENDENCY GOVERNANCE

Dependency management is not a one-time task. It requires regular attention to ensure your dependencies are up to date. Below are some useful steps for how to govern your dependencies:

- Establish clear guidelines and policies for selecting, approving, and managing dependencies within your organization.
- Specify acceptable sources for dependencies (e.g., official repositories, trusted vendors, etc.).
- Define a process for reviewing and approving new dependencies before adding them to projects.
- Outline guidelines for updating dependencies, including frequency, risk assessment, and potential breaking change considerations.
- Regularly scan your dependencies for known vulnerabilities. Utilize automated tools and incorporate security scanning into your CI/CD pipeline.
- When updating dependencies, prioritize addressing security vulnerabilities and mitigate potential risks before integrating new versions.
- Stay updated on the security landscape and emerging vulnerabilities to proactively address threats associated with your dependencies.
- Monitor dependency usage by tracking how your dependencies are used within your codebase and identify unused or rarely used dependencies.
- Regularly evaluate the need for each dependency. Consider if a particular dependency still aligns with your project's requirements or if alternatives exist.
- Remove unused or outdated dependencies to minimize potential security risks and streamline your project's codebase.

VERSIONING AND UPDATES

Encourage the adoption of semantic versioning for all dependencies. This standardizes versioning practices, making it easier to understand the nature of changes in each new version:

- **Major version:** Introduces breaking changes.
- **Minor version:** Introduces new features but doesn't break existing functionality.
- **Patch version:** Fixes bugs or security vulnerabilities without introducing new features.

By following semantic versioning, you can make informed decisions about updating dependencies based on the nature of changes introduced in new versions. Continuous testing is required to make sure you don't break anything when you update dependencies.

SECURITY VULNERABILITIES AND SCANNING

Security vulnerability scanning plays a critical role in safeguarding the integrity and reliability of software used by organizations. It helps you keep your systems safe through:

- **Proactive identification:** By proactively scanning open-source components for vulnerabilities, organizations can identify potential weaknesses before they are exploited by attackers.
- **Early mitigation:** Early detection allows for timely patching or addressing vulnerabilities, minimizing the window of opportunity for attackers and mitigating potential damage.
- **Improved software quality:** Regular vulnerability scanning contributes to an overall improvement in the security posture of software applications, fostering trust and reliability.
- **Compliance and risk management:** In regulated industries or security-conscious organizations, vulnerability scanning demonstrates proactive security measures and helps meet compliance requirements.

SECURITY VULNERABILITIES AND REPORTS

With open-source software, all users have access to your code; therefore, users are also able to identify vulnerabilities in your code and dependencies. This should be seen as a blessing rather than a curse. Being able to identify vulnerabilities before an attacker exploits them is, indeed, a good thing.

The following can be used as guidelines to handle security vulnerabilities and reports:

1. Establish clear and accessible channels for reporting vulnerabilities in your open-source code or dependencies.
2. Maintain a well-defined security policy document outlining the process for reporting vulnerabilities.
   - Define what constitutes a vulnerability and what information should be included in a vulnerability report.
   - Specify preferred communication channels for reporting vulnerabilities, such as the email address, issue reporting system, etc.
   - Clearly document the public acknowledgement/appreciation process.
3. Define a process for dealing with a reported vulnerability.
   - Accessing the vulnerability, categorizing based on severity, acknowledgements, alerting customers, fixing the issue, maintaining secrecy until the issue is fixed, etc.
4. Once an issue is fixed, deliver the software patch and notify customers. Provide them a reasonable period to apply the fix.
5. Once the customers have applied the fix, notify and issue the fix to the community as well.
6. Proceed with the public acknowledgement and reward for the reporter.

RESPONDING TO VULNERABILITIES

When a vulnerability is reported in your software, it is important to maintain a stringent response process. A high-level response process is outlined below:

1. Initial acknowledgement and triage

- Acknowledge receipt promptly: Respond to the vulnerability report within a short timeframe (ideally, within 24 hours) to acknowledge its receipt and thank the reporter for their responsible disclosure.
- Confidentiality: Maintain confidentiality unless a public disclosure is necessary or agreed upon with the reporter.
- Maintain an internal database of reported vulnerabilities and check against that to see if the same issue has been reported before.
- Triage the vulnerability: Analyze the report to assess the severity of the vulnerability, potential impact on users, and feasibility of exploitation.

2. Communication and collaboration

- Communicate with the reporter: Keep the reporter informed of the progress made in addressing the vulnerability.
- Seek clarification: If necessary, ask clarifying questions to understand the vulnerability report better and ensure accurate assessment.

3. Fix development and verification

- Develop a fix: Prioritize and dedicate resources to fixing the vulnerability promptly, considering the severity and potential impact.
- Internal testing: Thoroughly test the fix within your development environment to ensure it effectively addresses the vulnerability and doesn't introduce new issues.

4. Public disclosure and updates

- Release a patch or update: Provide a patch or update that addresses the vulnerability. Issue it to your customers first.
- Coordinate public disclosure: If a public disclosure is necessary, collaborate with the reporter to determine an appropriate timeframe and communication strategy.

MANAGING SOFTWARE LICENSES

When building open-source software, it is critical to understand working with software licensing. Just because a software is open source doesn't mean you can use it at will. You need to check if the license associated with the software permits you to use the software in the way in which you intend to. The same applies to the open-source software that you produce as well. The license you attach to your own software will determine how others can use, modify, and redistribute it.

HOW TO CHOOSE A SUITABLE SOFTWARE FOR USE BASED ON ITS LICENSE

Here are some common factors to check before you choose an open-source software for your project.

1. **Copyright and distribution:**
   - Does the license allow you to freely distribute the software, even if you modify it?
   - Are there any restrictions on how you can distribute it (e.g., requiring that source code be included)?
2. **Modification:**
   - Can you modify the original code to fit your project's needs?
   - Are there any limitations on how you can modify it (i.e., preserving certain functionalities)?
3. **Commercial use:**
   - Can you use the software in a commercial product or service?
   - Some licenses might restrict commercial use or require additional steps (i.e., disclosing modifications).
4. **Attribution:**
   - Does the license require you to credit the original authors in your project?
   - How should attribution be provided (i.e., copyright notice)?
5. **Warranty and liability:**
   - Open-source software typically comes with no warranty.
   - Does the license clarify who is liable for any issues arising from the software?

The Open Source Initiative (OSI) License Reference provides explanations of various licenses. Here are some common open-source licenses:

1. **Apache License 2.0** – A popular permissive license that strikes a good balance between openness and protecting the original code, the Apache License 2.0 requires keeping copyright notices, providing a copy of the license with the distributed code, and following certain patent terms, but allows for free use, modification, and distribution, even for commercial purposes.
2. **MIT License** – Another very permissive license, the **MIT License** allows you to keep the copyright and license notice intact, but otherwise, it allows for free use, modification, and distribution of the code, even for commercial purposes.
3. **BSD Licenses** – BSD is a family of permissive licenses, with the most popular ones being 2-Clause BSD and 3-Clause BSD. Similar to the MIT License, this requires keeping copyright notices and disclaimers, but otherwise, it allows for free use, modification, and distribution.

In case of any doubt on the suitability of a software license, it is always recommended to consult a lawyer. Apart from the suitability of the software and its license, you also need to do a general health check on it to make sure the community around it is active. There are many open-source projects and versions of projects that have been abandoned by the community. Make sure that the software you select for your project has active community involvement. In case a critical bug or security vulnerability is reported within that software, an active community will increase the chances of finding a resolution faster rather than on your own.

PREVENTING ACCIDENTAL MISUSE OF A SOFTWARE LICENSE

In a project with many contributors, it is quite possible that someone unknowingly uses a software dependency that does not bear a license friendly for your purpose. As a general rule of thumb, it is important to establish a process where you mandate an approval process and a security scan report for any software dependency in your organization. It is also a best practice to establish a rule saying that you can only use software bearing a particular license. (e.g., Apache 2.0 and MIT only).

However, as they say: "trust but verify." Make sure you have an automated process that scans through your software dependencies and performs relevant checks. This can be integrated with your software CI pipelines so that these scans never get missed. Maintaining an internal database of all software dependencies and their licenses will help in this cause. When a new dependency is introduced, the checks will fail due to it not being available in the database yet — at which point, the new dependency can be added to the database after review and approval.

CONCLUSION

In conclusion, open-source software presents a compelling alternative to traditional proprietary solutions. It offers cost-effectiveness, flexibility, security, and a collaborative environment that fosters innovation. However, it's crucial to approach open source with awareness and implement best practices to mitigate potential risks. By carefully evaluating dependencies, maintaining security hygiene, and contributing responsibly to the open-source ecosystem, individuals and organizations can harness the full potential of open-source software while safeguarding their digital assets and fostering a more secure and collaborative technological landscape.

WRITTEN BY NUWAN DIAS,
DEPUTY CTO, WSO2

Nuwan is a Product Manager at WSO2. He comes from a technical background, contributing to software products in various business domains. He's an author, speaker and technical specialist on APIs, integration, and security.

3343 Perimeter Hill Dr, Suite 100
Nashville, TN 37211
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.