LIVEWYER

The HashiCorp & Terraform Risks

What is the appropriate mitigation?

The HashiCorp & Terraform Risks

Introduction

Since August 2023, there has been ongoing discourse surrounding HashiCorp›s licensing changes and the implications for Enterprise Platforms that rely on its products, particularly Terraform and Vault. This change has sparked significant concern and uncertainty among organisations that have built their Enterprise Platforms using these products.

This document adopts a Program Management perspective to evaluate the risks posed to Enterprise Platforms by these August 2023 licence changes. Through this risk and mitigation evaluation, we will aim to answer the following business questions:

» **How do we mitigate the risk of this uncertainty?**

» **Are any mitigation options valid?**

» **Will my Platform circumstances impact my mitigation choice?**

By examining the available mitigations, we seek to provide actionable insights and strategic guidance for enterprises navigating the uncertainty of HashiCorp›s newly adopted licensing framework.

Background

In the dynamic world of cloud infrastructure, HashiCorp established itself as a cornerstone since its inception in 2012 by Mitchell Hashimoto and Armon Dadgar. The company that brought us Terraform and Vault achieved significant milestones over the years that shaped the landscape of Infrastructure as Code (IaC).

In March 2020, HashiCorp›s influence and relevance in the Cloud Native space were cemented further when it joined the Cloud Native Computing Foundation (CNCF), underscoring its commitment to Open Source and community-driven development.

Terraform V1.0 was announced in June 2021, marking its general availability and affirming its maturity and stability for enterprises to use at the core of their Cloud Platforms.

However, In August 2023, HashiCorp transitioned to a Business Source License (BSL or BUSL), significantly affecting the future of Enterprise Platforms utilising Terraform and other HashiCorp products.

Examining Terraform specifically, from version 1.5.7 onwards, the BSL states that:

> ***"A 'competitive offering' is a Product that is offered to third parties on a paid basis, including through paid support arrangements, that significantly overlaps with the capabilities of HashiCorp's paid version(s) of the Licensed Work"***

Source: HashiCorp BSL

The wording of the BSL raised severe concerns within the cloud native community about Terraform's future use, concerns which were amplified further when HashiCorp announced its partnership with IBM in April 2024.

Risks & Assumptions

When HashiCorp changed its source code licence from a Mozilla Public License v2.0 (MPL 2.0) to a Business Source Licence v1.1 (BSL), it impacted the future releases of all products. While existing versions of Terraform remain unaffected, future iterations beyond 1.5.7 will see restrictions for Enterprises and their Platforms, restricting them from leveraging new versions of Terraform in their offerings.

The above actions from HashiCorp have resulted in two main risks for organisations utilising Terraform within their Enterprise Platforms. These risks are caused by the uncertainty created from both a commercial and delivery perspective.

R-001 — HashiCorp / Terraform usage in Production

There is a risk that HashiCorp will consider an Enterprise Platform utilising Terraform beyond version 1.5.7 a competitor and in violation of its BSL licence.

The material released from HashiCorp needs to clarify whether an Enterprise Platform providing services to customers can utilise Terraform for Infrastructure as Code in production or whether they would be considered a competitor by doing so.

R-002 — Terraform Commercial Uncertainty

There is a risk that HashiCorp will change its licensing costs and begin charging Terraform users from version 1.6 onwards.

While HashiCorp's current statement is that users are free to continue using Terraform, this is not guaranteed to continue, especially given IBM's recent acquisition.

Assumptions

The following high-level assumptions about your Platform will be used to contextualise the mitigation options assessed and discussed later in this document:

A-001 — The risks **R-001** and **R-002** relate to an organisation running a large Kubernetes Platform.

A-002 — This Platform comprises both Development and Production environments.

A-003 — The Platform is currently using version 1.5.7 of Terraform.

A-004 — The Platform is built using solid architecture principles. (See LiveWyer principles for building a Platform on lwy.io).

Risk Mitigations

Mitigations are available for every risk posed to a project, even if the mitigation is simply accepting the risk. This is a feasible action if the remediation effort will cost more than the impact of the risk should it materialise. That being said, it is crucial to consider all mitigation options thoroughly before remediation implementation occurs. Often, cheap short-term fixes have costly long-term consequences, particularly when the mitigations are for large and complex Enterprise Cloud Platforms.

This document will examine each mitigation, evaluating the positives, negatives, and effort involved to provide a detailed recommendation, ultimately answering the questions at the beginning of this document.

M-001 — OpenTofu

OpenTofu is a Terraform fork created in response to HashiCorp's switching to its BSL license. The codebases are extremely similar, making this a like-for-like alternative to Terraform.

You can find more information about its history on its website, particularly its FAQ section: FAQ | OpenTofu.

Positives

Transitioning from Terraform to OpenTofu offers several significant advantages, primarily by completely mitigating risks **R-001** and **R-002** without requiring additional remedial actions. OpenTofu, a current like-for-like product, encompasses all essential features needed for your platform's Infrastructure as Code (IaC) requirements. The active community support and ongoing feature development for OpenTofu, often addressing previously unmet requests for Terraform, ensure continuous enhancement and innovation of the Product. The close similarity between the codebases of Terraform and OpenTofu further simplifies the implementation process, minimising the need for extensive retraining of engineers. According to OpenTofu, it serves as a drop-in replacement for Terraform, compatible with versions 1.5.x and most of 1.6.x, requiring no code changes for compatibility and is ready for production use. Thus, moving to OpenTofu from Terraform is a strategic step that maintains operational efficiency while enhancing the platform's capabilities.

Negatives

Moving from Terraform to OpenTofu does come with several challenges. One significant drawback is the urgency with which your platform's management and leadership team must decide. Being "under the gun" to choose is seldom ideal, but a swift resolution minimises the associated risks. In large-scale enterprises, decisions typically take time to navigate the necessary approval forums. This delay could complicate the transition as the codebases of OpenTofu and Terraform are expected to diverge over time. As the differences between the two products grow, the complexity of migration will increase, potentially leading to greater operational difficulties and the need for additional resources to manage the transition effectively.

Effort

Transitioning from Terraform to OpenTofu will demand considerable effort from various teams across multiple sprints. Although the current similar codebases

make the implementation relatively straightforward, the high stakes and potential consequences of any errors ensure that the effort involved is substantial. Significant architectural effort is also necessary, as Infrastructure as Code (IaC) is a critical component and best practice for any platform, necessitating thorough discussions and approvals within the architecture team. Additionally, despite the similarities between the codebases, comprehensive updates to your documentation are essential to reflect the changes, ensuring that all stakeholders are informed and aligned with the new architecture.

Summary

Transitioning from Terraform to OpenTofu is a viable solution that offers significant advantages, such as complete mitigation of risks **R-001** and **R-002**, active community support, and minimal retraining due to similar codebases. However, the urgency of a swift decision is crucial, as delays could lead to increased complexity and operational difficulties due to the diverging codebases. While the effort required will span multiple sprints and involve significant architectural input and documentation updates, acting quickly will make the transition much smoother. Therefore, this mitigation option becomes less appealing as time progresses.

If your team has bandwidth, and the decision can be made quickly, OpenTofu is a fantastic mitigation option.

Graph Summary

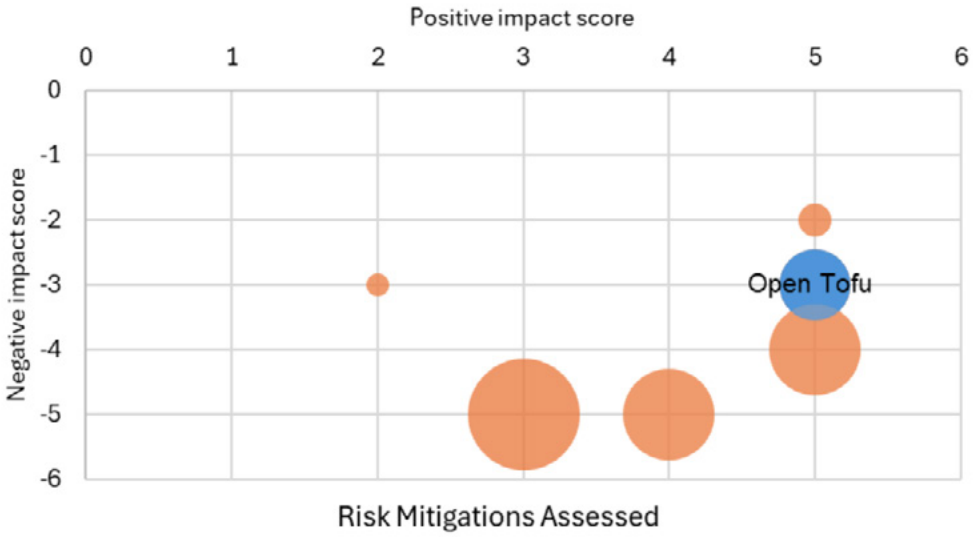


Table Summary

OpenTofu		
Positives	Negatives	Effort
EXTRA LARGE	MEDIUM	LARGE

M-002 — Pin Terraform version at 1.5.7

Given that both risks can only materialise in future versions of Terraform, it is possible to pin your Platform's Terraform version at V1.5.7 as a valid mitigation to **R-001** and **R-002**.

Positives

Pinning Terraform at version 1.5.7 offers significant advantages, primarily by effectively mitigating risks **R-001** and **R-002**. This version operates under the Mozilla Public License v2.0 (MPL 2.0), ensuring compliance and stability for your Platform. Unlike other methods, such as **M-001** (OpenTofu), this mitigation approach provides flexibility, allowing your organisation to wait for further clarifications from HashiCorp regarding the implications of the Business Source License (BSL). Therefore, this strategic pause provides the option to later pivot to alternative mitigations like **M-001** (OpenTofu) or **M-004** (Accept Risk), depending on future information released from HashiCorp. Additionally, the risk associated with this approach is minimal, as the Platform will maintain its current functionality by continuing to use a familiar version of Terraform. Thus, starting with this **M-002** mitigation provides both time and flexibility, presenting a robust initial position for ongoing risk management.

Negatives

Pinning Terraform at version 1.5.7 does come with some drawbacks. The primary negative aspect is that your Platform will only be able to utilise features available up to version 1.5.7. Although this version includes necessary patches under the existing Mozilla Public License v2.0 (MPL 2.0), it will miss out on new features that could benefit both your Platform and your engineering team. Additionally, while this mitigation effectively addresses the immediate risks of **R-001** and **R-002**, it is less proactive than other options, and the limitation on new features might necessitate further engineering actions down the line. Lastly, the longer the Platform remains pinned at version 1.5.7, the more it will lag behind the latest versions of Terraform and OpenTofu. This would complicate future updates or migrations, in addition to the compliance risk of security enhancements being included in these Product releases your Platform will not be able to utilise.

Effort

Implementing this mitigation requires minimal effort from the engineering, architecture and documentation teams. The engineering effort needed to pin Terraform at this version is relatively small, making the process straightforward. Additionally, the documentation team will only need to make minor updates to

reflect this change, ensuring that the overall effort involved in implementing this mitigation strategy remains low.

Summary

M-002 offers significant benefits both by fully mitigating risks with minimal short-term impact on the Platform and providing flexibility for future adjustments based on HashiCorp's updates. Although it has minor negatives, mainly due to the availability of more proactive mitigations, and not being able to take advantage of any Terraform new features, the minimal engineering and documentation effort required makes **M-002** an attractive option, mainly if your engineering teams are occupied with other urgent matters.

Graph Summary

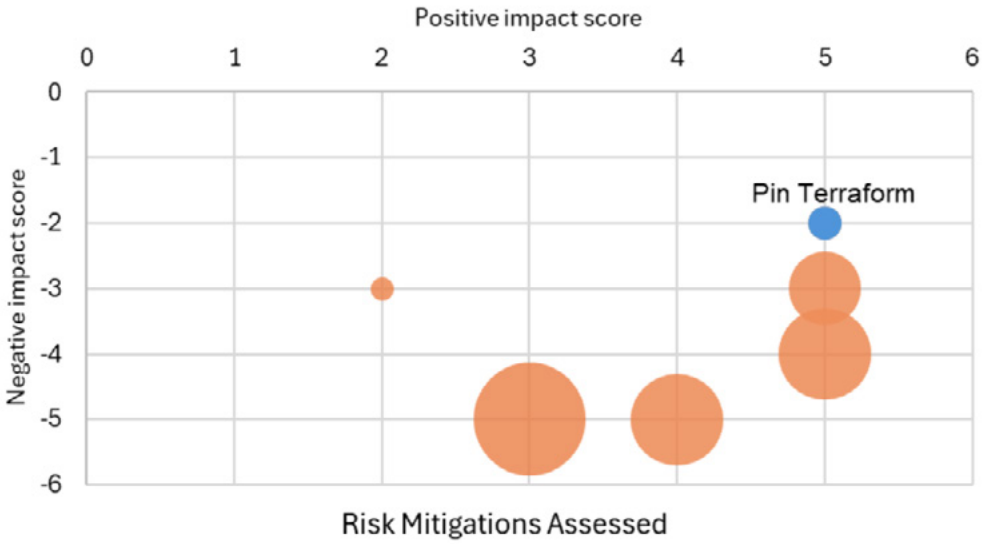


Table Summary

Pin Terraform		
Positives	**Negatives**	**Effort**
EXTRA LARGE	**SMALL**	**EXTRA SMALL**

M-003 — Alternative products

Given the wide variety of "Alternative products" available to replace Terraform, we have split this section to focus on three separate examples. These examples may not cover a complete list of available products, but they cover a broad range of options.

We have chosen the following three "alternative products" to focus on:

M-003a — Cloud Provider Products

M-003b — Pulumi

M-003c — In-House Development

M-003a — Cloud Provider Products

Depending on your chosen cloud provider, you may want to consider the respective product(s) to align your cloud infrastructure with your IaC product.

Below, we look at three of the most popular Cloud providers and their respective IaC products:

AWS → AWS CloudFormation

Azure → Azure ARM Templates

GCP → Google Deployment Manager

Positives

Transitioning from Terraform to a cloud provider's native Infrastructure as Code (IaC) does have advantages and will effectively mitigate the risks associated with HashiCorp's shift to a BSL license (**R-001** and **R-002**). By adopting a cloud provider's IaC solution, organisations can remove the uncertainties and constraints imposed by the new licensing model. In addition, aligning with a chosen Hyperscaler grants access to dedicated support channels, improving operational efficiency and reliability. Lastly, cloud-native IaC tools are often optimised for their respective platforms, providing improved performance, stability, and support tailored to the specific needs of your Platform infrastructure.

Negatives

While transitioning from Terraform to a cloud provider's native Infrastructure as Code (IaC) product can mitigate risks associated with HashiCorp's BSL license change, it also introduces several significant challenges. Firstly, the migration

process would be lengthy and complex. A phased and gradual roll-out is recommended over a big bang migration to mitigate implementation risks. Still, this approach is far slower and would necessitate parallel running of both Terraform and the new cloud product. This dramatically impacts engineering, documentation, and architecture teams, as they manage two codebases, documentation sets, and designs simultaneously until the migration is fully completed. Furthermore, this move would sacrifice the vendor-agnostic nature of IaC, potentially resulting in vendor lock-in and making it unsuitable for current or future multi-cloud platforms.

The complexity and risk associated with migrating such a critical component as IaC are significant. Your engineers would need to undergo training to support the new product, adding to the transition costs and time. Lastly, maintaining your platform becomes significantly more complex as the new codebase differs from Terraform, requiring management of both systems until the migration is completed.

Effort

Transitioning from Terraform to a cloud provider's native Infrastructure as Code (IaC) product is a significant and high-risk endeavour that requires extensive effort across multiple teams. Key teams involved include Architecture, Engineering, Product Owners, Project Managers, and the Documentation Team. Each team must invest substantial time and effort to manage the platform during the migration period, as they will need to parallel run two codebases until the transition is fully completed. This parallel operation adds complexity and increases the workload for all involved, necessitating careful planning and coordination to ensure a successful migration.

Summary

Implementing **M-003a** is a viable mitigation. This would fully remediate the key risks, and alignment with cloud-specific support is particularly advantageous for Platforms on a single cloud. However, this transition comes with considerable technical limitations. One of which is the potential of vendor lock-in. Whilst this is a deviation from architecture best practices and could cause resiliency issues, this may be an acceptable risk if your Platform is only ever hosted on a single cloud. The second technical complexity is the lengthy and complex migration process. This will involve managing dual codebases during this period, and impose substantial burdens on architecture, engineering, documentation, and support teams, with existing Terraform efforts lost and new training required. The extensive effort involved necessitates significant resources across multiple teams and continuous complexity in platform operations and support. Therefore, whilst this mitigation is viable, other mitigations are more appealing.

Graph Summary

Table Summary

Cloud Provider Products		
Positives	**Negatives**	**Effort**
LARGE	**EXTRA LARGE**	**EXTRA LARGE**

M-003b — Pulumi

In this section, we examine the potential of Pulumi as an alternative to Terraform so that we can mitigate the risks associated with HashiCorp's BSL license. By detailing the positives, negatives, and implementation efforts, we aim to provide you with a comprehensive assessment. This can form the basis of any decision regarding Pulumi as a viable mitigation for your Platform.

Overview

From Pulumi's website:

> ***"Pulumi is a modern infrastructure as code Platform. It leverages existing programming languages—TypeScript, JavaScript, Python, Go, .NET, Java, and markup languages like YAML—and their native ecosystems to interact with cloud resources".***

Based on the description above, Pulumi seems to be a modern alternative to Terraform. When dealing with IaC, we want it to be declarative as a key architectural principle. Multiple threads online discuss whether Pulumi is imperative or declarative. For the sake of this report, we will assess it as a declarative tool, as stated by Pulumi.

Positives

Similar to **M-001** and **M-002**, the primary benefit of implementing Pulumi to replace Terraform would be that the Pulumi mitigation completely removes the risks posed by **R-001** and **R-002**. The second positive of note is Pulumi's use of common programming languages, such as Python, Java, Node.js and more. This greatly benefits engineering teams, allowing them flexibility and comfort when coding in their preferred language. The final benefit is that Pulumi is Open Source; therefore, there are support options available from both Pulumi and the Open Source community.

Negatives

The significance of this first negative against using Pulumi for an existing Platform cannot be stressed enough. Engineering Teams, Documentation Teams, Architecture Teams, etc, will have to manage two codebases, documentation and designs, respectively, as the migration occurs. This is known as "parallel running". Not only does this drastically increase effort for the teams mentioned above, but the complexity of rolling out new features, addressing CVEs, maintaining the Platform and dealing with customer incidents skyrockets. The second significant

risk in performing an X-Large Migration with something as important and ingrained in your Platform as IaC is the ramifications of any errors would be vast and severe and, therefore, a significant risk in itself to be approved by the business. Finally, all existing code would be thrown away, there would be no re-use of code, and all previous efforts to develop IaC through Terraform would be lost.

Effort

Our calculations consider the effort across multiple teams and skill sets X-Large. The engineering teams would need to spend significant time implementing and managing both codebases simultaneously, already mentioned as a substantial negative to this mitigation.

In addition to Engineering teams, architecture would require an equally large amount of effort, given the importance of designing the transition and final architectures for a pivotal component like IaC.

The last team we will discuss in detail before the final impacted teams are listed is the effort required from both Operations and Incident Management teams to support implementation. Again, this was touched upon in the negatives section above. Operating and supporting a platform is complex in nature, and managing a platform with two codebases is not far short of doubling the effort and knowledge required by these teams.

Other teams to mention are documentation and project management. Our calculations and estimates consider that both teams' efforts are significant in their design and implementation activities.

Summary

Using Pulumi to replace Terraform is a viable option and mitigates risks **R-001** and **R-002** in their entirety. That being said, the negatives involved in implementation are vast for an existing platform, and we do not consider this a pragmatic option, primarily because of the additional effort, complexity, and cost the Platform would undertake.

If you were building a Platform from scratch, this option would become much more appealing, but we believe it should be considered only in this scenario.

Graph Summary

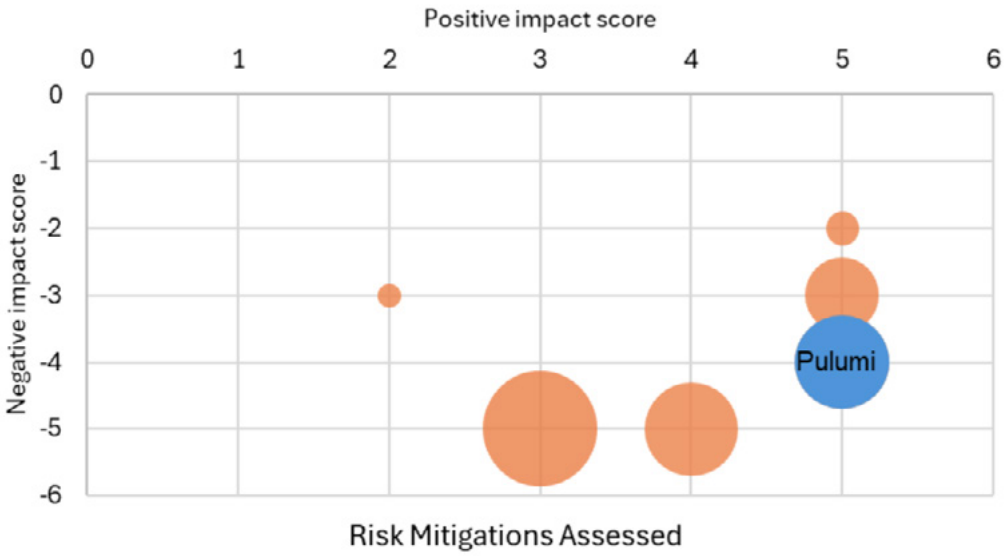


Table Summary

Pulumi		
Positives	**Negatives**	**Effort**
EXTRA LARGE	**LARGE**	**EXTRA LARGE**

M-003c — In-House Developed Product

Below, we assess the validity of developing an IaC tool In-House with your development team. On the face of it, this seems completely viable. Negating the principal risks and giving ownership and control of the product would prevent a licensing change in the future, as it has with Terraform.

But delving deeper may have other implications, so let's explore this mitigation in more detail.

Positives

As mentioned above, the primary benefit of implementing an In-House solution is the full mitigation and removal of **R-001** and **R-002**. This option does not require any future work and is a proactive form of mitigation. The secondary benefit of developing an In-House product is the complete ownership of the product and its roadmap. The product can be specifically tailored to meet the needs of your Platform and organisation. In the past, criticism has been made of HashiCorp and Terraform as requests for new features have gone unactioned for considerable periods. With the control over your product, new features that deliver significant value to your Platform can be prioritised accordingly.

Negatives

Unfortunately, this mitigation has many negative implications should you pursue it. The most sizable negative which should be considered is the migration process and the impact on your team. During this phased migration, your Engineering, Documentation, Architecture, and other teams must handle two codebases, documentation, and designs. This adds extreme complexity to maintaining a Platform which already comes with an array of complexities without adding additional where it is not needed.

In addition to maintenance and management, the migration itself is complex and involves additional risk. IaC is a critical component of the Platform. Performing a significant change to the codebase is a considerable risk that needs careful and close management over a long time for a large production platform. It is a considerable risk to perform a large Migration with something as important and ingrained in your Platform as IaC.

The third negative for an In-House developed product is the same as for any product: It will require considerable ongoing maintenance to ensure it continues to add value. This point highlights the importance of continued support and

sponsorship from senior stakeholders.

The final point we will discuss in detail is the impact this has on recruitment and training. As your Platform scales and new services are added, it is natural that you will need to increase your engineering capability. Having an In-House solution for IaC means recruiting an engineer with experience in the tool is impossible. Therefore, the newly hired engineer will require significant training before they can manage the Platform, regardless of their previous experience.

Considering the large number of negative implications associated with this mitigation, we have focused on describing the most impactful four. However, other negative factors to consider would be:

1. In-house knowledge only provides a risk that the product may have a “shelf life” equivalent to the developer’s time in the company.
2. Existing code would be thrown away, and no code would be reused. All previous efforts to develop IaC through Terraform would be lost.
3. No external support from vendors would be available.

Effort

The effort involved with this mitigation action is extreme and the most sizable of all mitigations examined in this report. We have taken into account the effort involved in the initial Product Development, Migration, Training of current and new engineers, and finally, the development of new features alongside the ongoing maintenance of the product.

This mitigation would most impact the engineers. It would require significant effort in all the stages mentioned above. The architecture and Documentation teams would also be significantly affected, predominantly with the initial Product design and migration stages, but they would also be involved with Operations and maintenance.

From a Project Management perspective, this would require close management and oversight in all phases, particularly the complex and lengthy migration process.

Finally, your incident teams would require initial training in your new product, which again introduces risk and complexity when your Platform uses two different codebases throughout the transition from Terraform to your In-House developed alternative.

Summary

Whilst the in-house developed product is a viable option for mitigating risks **R-001** & **R-002**, the negative implications and vast development, migration and maintenance effort make this option extremely unappealing. Developing the product will take a significant amount of time. Therefore, you would be managing risks **R-001** and **R-002** for a lengthy period before you can even begin the migration process.

If you were designing a Platform on a greenfield site, the nature of this being a Product for your engineering team to maintain and develop alongside the Platform would make other options much more viable and appealing while still managing to mitigate the risks fully.

Control over the Product and the roadmap, while a positive aspect of this option, does little to negate the large negatives and effort involved with this mitigation.

Overall, as a mitigation, we strongly advise against pursuing the development of an In-House tool.

Graph Summary

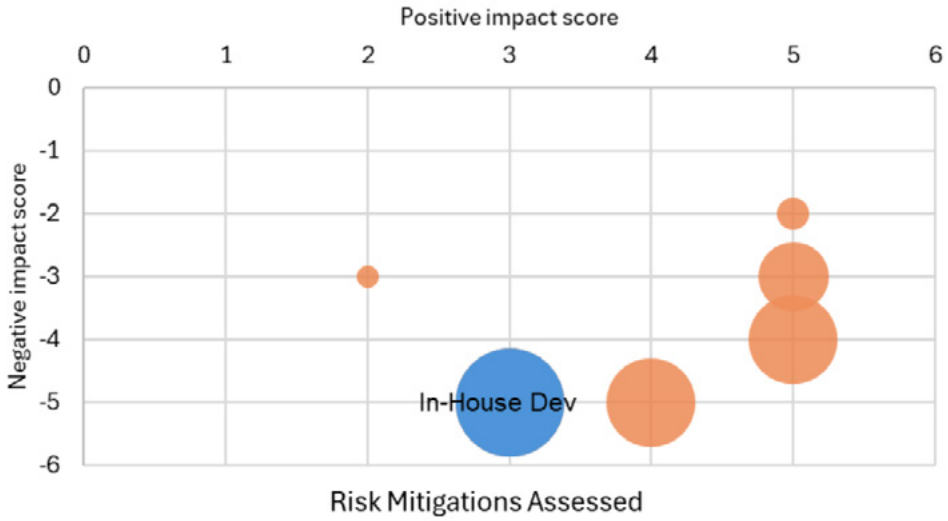


Table Summary

In-House Developed Product		
Positives	**Negatives**	**Effort**
MEDIUM	**EXTRA LARGE**	**EXTRA EXTRA LARGE**

M-004 — Accept the risk and continue with Terraform

We have explored a number of mitigations, but the one remaining is to purely accept the risk. In essence, this is the "do nothing" approach, as no mitigations are appropriate, or you believe the cost and impact of mitigating are more significant than the combined likelihood, cost, and effort should the risks materialise. Let's delve into the positives, negatives, and efforts of accepting these risks below.

Positives

Although mitigation **M-004** does not have a long list of positives, there are a couple. Firstly, and obviously, by accepting Risks **R-001** and **R-002**, your Platform will continue to work without disruption. There will be no impact on customers or your teams who support and maintain your Platform. In addition, your Platform can take advantage of all new features Terraform releases in versions beyond 1.5.7.

Negatives

Similar to the positives, accepting the risks has few negatives, but they are significant. Firstly, the risks are not mitigated. Therefore, from a risk management perspective, the situation is being monitored indefinitely, and the potential impact continues to loom over your team and Platform.

Effort

The effort involved in **M-004** is minimal. You may need to engage your architecture team and risk manager to gain approval for the acceptance.

Summary

Given how significant the impact could be if they materialise, we would not advise accepting the risk. Whilst it could be an attractive option given how little effort is involved and the fact you can continue to use Terraform and any new features it may release in the future, the fact that neither risk has been mitigated results, in our opinion, this should not be an option for your Platform.

Graph Summary

Table Summary

Accept the risk		
Positives	**Negatives**	**Effort**
SMALL	**MEDIUM**	**EXTRA SMALL**

Conclusion

Whilst we have assessed several mitigations available, three stand out as potential options depending on your circumstances.

Scenario 1 – Existing Platform with Engineering bandwidth

In this scenario, there is value in making the move from Terraform to OpenTofu (**M-001**). While this option becomes less appealing over time, if your engineering team currently has the bandwidth, mitigating these risks while the codebases are similar is a very appealing option.

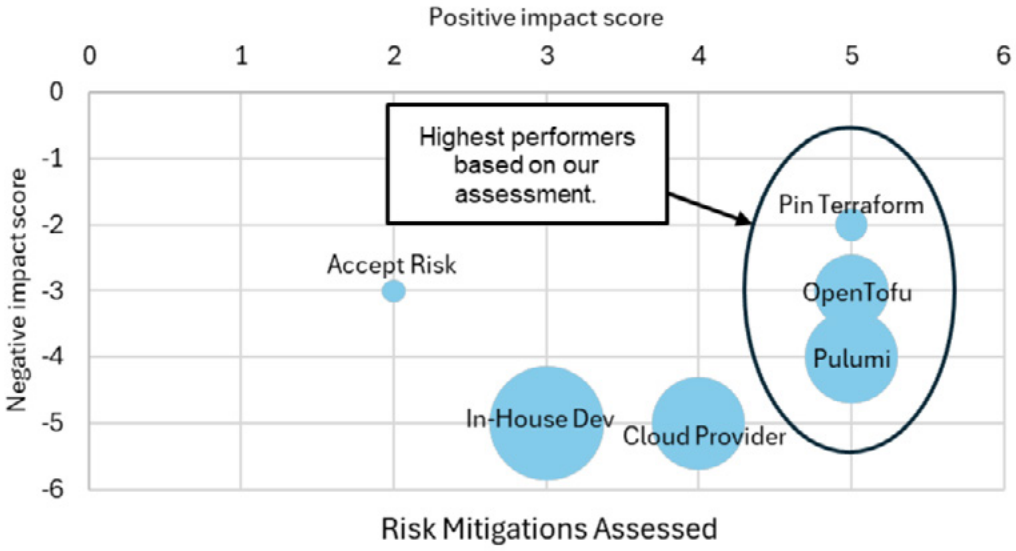
Scenario 2 – Existing Platform with no Engineering bandwidth

If your existing team is fully utilised with pressing new features and addressing critical CVE's amongst other important tasks, our recommendation is to pin Terraform at version 1.5.7 (**M-002**). The benefit of this is you give yourself and your team time to see if HashiCorp releases any new information in the future. With little engineering effort, you can remove the risks and have the flexibility to pivot in the future should you need to.

Scenario 3 – Building a New Platform

Whether you are in the design stages or having early discussions about creating a Platform, both Pulumi (**M-003b**) and OpenTofu (**M-001**) are the standout candidates. Pulumi will be an attractive candidate for your engineering teams, given the flexibility in languages you can use. On the other hand, OpenTofu is the closest direct replacement for Terraform, which will remain Open Source for its lifetime, given the nature in which it was created. These mitigation options will allow you to design and build a platform without the potential impacts of **R-001** and **R-002**.

Full Graph Summary



The above summary shows the top-performing mitigations in our assessment. The size of the bubbles indicates varying degrees of effort involved in each option, with Pinning Terraform requiring the least effort and migrating to Pulumi requiring the most effort of the three.

Omissions

This section explains whether any factors, both positive and negative, were excluded from this Risk mitigation study and documents the logic behind the decision.

Quantifying future product development

Many articles across the Cloud Native community discuss whether OpenTofu or Terraform is more "proactive" in dealing with issues and new feature requests from the Cloud Native community.

On the one hand, OpenTofu states:

> ***"Anyone who has used Terraform in the last eight years has probably come across issues that took some time to be resolved. The large community involved in developing OpenTofu means this will no longer be the case."***

Conversely, if you look at GitHub, the level of activity and commits on Terraform in May and June is far higher for Terraform than OpenTofu.

Therefore, measuring the health of a project and its "proactiveness" is challenging. The Cloud Native Native community has discussed this topic at length in numerous articles, such as Measuring the Health of Git Repositories | by Augmentable Software.

Given that there is no accurate way of monitoring and quantifying progressive development, never mind estimating and quantifying future development activities, we have purposefully omitted this factor from this document.

Contributing Documents

You can find a full Project Management Risk Log, which contains the details of how each risk and mitigation's values were assessed and calculated, here: PM Risk Log.

This document references the LiveWyer architectural best practices and design principles. The full details are available on our website: LiveWyer Platform Design Principles