vFunction

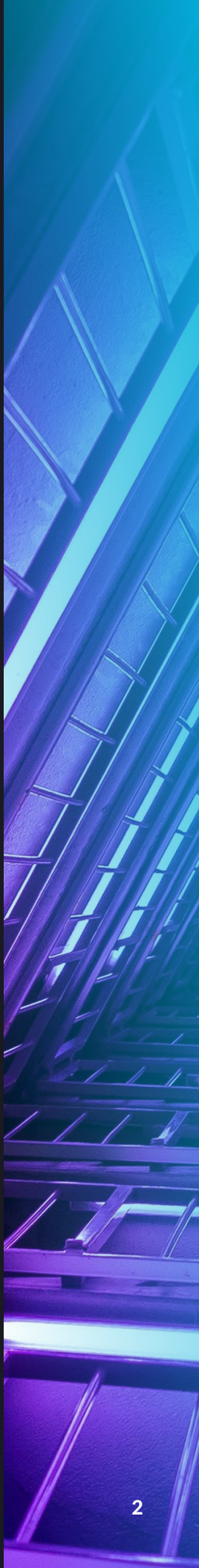
Technical Debt

A Guide for Frustrated Software Architects and CIOs

WHITEPAPER

Executive Summary

- As of 2022, application modernization has become a top priority for CIOs in the enterprise. Executives rank “investing more in application innovation” as their **top goal** for modernization efforts, while software architects list “improving engineering velocity” as their primary objective.
- Yet, according to a recent survey of 250 senior IT professionals, the majority of Application Modernization initiatives end in failure, at a cost of $1.5 million and 16 months of work hours on average.
- Among those who had started app modernization projects and failed, the top reason was a “failure to accurately set expectations,” followed by factors like a “lack of intelligent tools” and “organizational pushback.”
- Innovation is a necessity to meet business objectives, but the biggest obstacle to innovation is technical debt—a major impediment to engineering velocity.
- Technical debt accumulates when decision makers go for a short-term solution to a software development problem—instead of a more exhaustive, long-term solution—and this comes with substantial, hidden costs that organizations must pay later.
- The symptoms of technical debt ripple across your entire organization, adding complexity to your enterprise applications, delaying business goals, and frustrating your valuable engineering staff.
- Long test and release cycles make it difficult to reliably meet business requirements, ultimately leading to a poor customer experience. Technical debt also makes it harder to quickly onboard new engineering staff and has a negative impact on team morale.
- To combat the problem of technical debt, vFunction devised and validated a method to measure technical debt of an application, similarly based on an award-winning academic paper. This paper describes how an organization should plan development cycles while taking into account the effect of accumulated technical debt has on future development and releases.
- The results of these efforts are now available to architects and developers in **vFunction Assessment Hub**, which combines static analysis with AI to create a lightweight assessment tool for calculating the ongoing technical debt of your monolithic applications. Ultimately, you’re given tools to prioritize application modernization efforts and build a business case for removing architectural technical debt from your applications.



Why Application Modernization? Goals, Challenges and Failures

Application modernization is a stated priority for CIOs in the enterprise—a top-three initiative, according to CIO magazine's State of the CIO 2022, in both effort and resources. Between 2021 and 2022, industry analyst IDG witnessed the category of "Application Modernization" skyrocket from the #8 to the #3 top priority for CIOs. Additionally, IDG predicts that by 2024, the majority of legacy applications will be getting an update.

Furthermore, Gartner predicts that by 2025, 90% of applications currently used today will still be in use, and lagging due to the lack of funding for modernization initiatives and that 40% of IT budgets will be spent simply maintaining technical debt.

It's clear that application modernization is critical to business success, but at the same time we are confronted with some disturbing realities: according to 250 senior IT professionals, the majority of Application Modernization initiatives end in failure, at a cost of $1.5 million and 16 months of work hours on average.

To understand why this is happening, let's take a look behind the motivation for application modernization projects, and how the reasons they fail are connected back to the motivation in the first place.

79%

of app modernization projects fail

$1.5M

average cost of a Modernization Project

16mo

average time of a modernization ptoject

The impact of Application Modernization failures
Source: Why App Modernization Projects Fail (2022)

Modernization Goals: Engineering Velocity and Innovation

Growing your enterprise and keeping your customers happy is a continuous battle in light of competitive pressures from fast-moving digital natives—not to mention major global events like the Covid-19 pandemic.

As a result, aging system architectures are forced into the spotlight as IT teams struggle with engineering velocity and application scalability. This in turn has driven board members and executive teams to initiate a mandate to modernize applications and infrastructure to reduce costs and improve application scalability. Many organizations look to cloud platforms like AWS, Azure, Google Cloud, Red Hat OpenShift, and others to achieve these benefits.

Executives rank investing more in application "innovation" as their top priority, while app architects list "improving engineering velocity" as their main goal. This speaks directly to the unique pressures each of these IT stakeholders may feel.

Top Goals of App Modernization Projects

Top Goals of Application Modernization
Source: Why App Modernization Projects Fail (2022)

This difference also speaks to different sides of the same coin in terms of overall organizational goals: innovation is a necessity to meet business objectives, but the biggest obstacle to innovation is technical debt—a major impediment to engineering velocity. Related to engineering velocity is the "ramp time for new developers," another concern mentioned by architects that directly connects to application modernization.

Why Modernization Efforts Fail So Often

Factors like technical debt (in the form of app complexity and risk) and overall cost have caused the majority of modernization projects to fail. Yet before that even happens, internal struggles put app modernization efforts in peril: 97% predict someone in their organization would push back on a proposed project before it even starts. This difference also speaks to different sides of the same coin in terms of overall organizational goals: innovation is a necessity to meet business objectives, but the biggest obstacle to innovation is technical debt—a major impediment to engineering velocity. Related to engineering velocity is the "ramp time for new developers," another concern mentioned by architects that directly connects to application modernization

97%

predict someone in their organization would push back on a proposed project

Top Reasons for Pushback

Reasons for Organizational Pushback
Source: Why App Modernization Projects Fail (2022)

In terms of organizational pushback, "risk" is the number one reason for both Executives and Architects. Other concerns include cost, fear of change, and the lack of clear ROI.

In looking at what causes failed modernization efforts, there are some noteworthy differences between what executives and architects call out as top reasons for failure, which point to the intertwined—but sometimes opposing—pressures that leaders and architects experience.

Among those who had started app modernization projects and failed, the top reason cited across all stakeholders was "failure to accurately set expectations." Yet in looking at architects alone, they note a "lack of intelligent tools" as the number one reason for failure.

Failure to Accurately Set Expectations

Everyone can agree on one thing: if you haven't set and confirmed expectations and goals for a modernization initiative, things are likely to go wrong from the start. No one can confidently execute a project that hasn't quantified the expected business outcome, project scope and timeline, and the various costs associated with modernization.

To do this right, we need a combination of executive sponsorship, strong project management, and tools and technologies to accurately assess and quantify the ROI of the effort.

Lack of Intelligent Tooling

Among architects, the need for tooling ranked high when asked why modernization projects fail, with responses to the survey such as: "We still lack the tools to do it properly." "Having more automation tools means faster release times and better modernization processes."

Reasons for Organizational Pushback
Source: Why App Modernization Projects Fail (2022)

This points to a lack of intelligent tools to help architects and engineers reduce the time and risk of these modernization projects, with many architects using static analysis tools and/or application performance management tools—neither of which analyze each application for architectural complexity, technical debt, and identifying aging frameworks, nor assist in the transformation to microservices.

The combination of poorly set expectations and the lack of tooling are that the primary goals of application modernization cannot be met: no increase in engineering velocity, and slow innovation cycles.

Technical Debt Is a Major Blocker to Application Modernization Goals

As we stated earlier: innovation is a necessity to meet business objectives, but the biggest obstacle to innovation is technical debt—a major impediment to engineering velocity.

Let's see what technical debt is really made up of, how it accumulates, and why it's so detrimental to engineering velocity and the ability to innovate.

Technical Debt: The Silent Killer of Engineering Velocity and Innovation

What is Technical Debt?

Splunk **describes Technical Debt** in the following way:

> **Tech debt, also known as technical debt or code debt, suggests that a simplistic, poorly understood, or "quick and dirty" solution to a software development problem (instead of a more thorough, robust solution) comes with substantial, hidden costs that organizations must pay later. Programmer Ward Cunningham originated the definition of technical debt as we know it in a 1992 article articulating that while the enterprise may save money in the short term by coding in this fashion, in the long run, "interest" from the tech debt, as with monetary debt, will accumulate, making the initial problem increasingly costly to fix as time goes on.**

Technical debt, in plain words, is an accumulation over time of lots of little compromises that consequently hamper your coding efforts. Sometimes, you (or your manager) choose to handle these challenges "next time" because of the urgency of the current release.

This is a cycle that continues for many organizations until a true breaking point or crisis occurs. If software teams decide to confront technical debt head-on, these software engineers may discover that the situation has become so complex that they do not know where to start.

The difficult part is that any decision we make needs to strike a balance between short-term benefits (e.g. releasing a version faster) and long-term costs of technical debt (e.g. paying that debt later on). This emphasizes the need to properly assess and address it when planning development cycles.

How Technical Debt Affects Your Applications, Business, and Engineers

Your monolithic application is hindering innovation due to its compounding technical debt, which is stifling engineering velocity and leading to the inability to rapidly innovate. The symptoms of technical debt make their way across your entire organization, including your enterprise applications, business goals, and valued engineering staff.

Technical Debt Leads to Slow Test and Release Cycles

Monolithic applications—which may have millions of lines of code and thousands of classes—that have accumulated technical debt are extremely difficult to update because a change to any part of the code can ripple through the application, causing unintended operational changes or failures in seemingly unrelated parts of the codebase.

Because monoliths are a single entity with functionalities and dependencies interwoven throughout, they are inflexible, brittle, and difficult to update with new features or functions. This makes testing cycles long, manually driven, and unpredictable, which naturally leads to slower release cycles.

Technical Debt Leads to the Inability to Meet Business Goals

If your test and release cycles are slow and prone to delays, then it becomes significantly more difficult to meet deadlines and business requirements. If it takes weeks to successfully build and deploy a new version of your product—namely for bug fixes—then you're likely not meeting modern customer expectations for software services.

When delayed releases and bug fixes impact the user experience, then customer satisfaction and retention are at risk–two fairly critical business metrics.

Technical Debt Impacts Team Onboarding and Morale

It may not be at the top of mind for some executives, but battling with difficult, brittle, and unpredictable monolithic systems can—and often does—negatively impact team morale.

A monolithic system often parallels a monolithic engineering organization. The lack of autonomy, clearly defined business domains, and the chance to make a personal impact on the success of the company are frequent anecdotal complaints by architects and developers.

Additionally, it is becoming increasingly difficult to find, hire, and onboard developers who pursue maintaining monoliths as a career path. Onboarding new staff members to the point where they understand what the overall system looks like can take 3-6 months.

Clearly, technical debt is a major liability for your ability to raise development velocity and increase speed on innovation. In order to tackle this problem, the first step is to understand and quantify it.

Can Technical Debt Be Measured?

In their seminal article from 2012, "In Search of a Metric for Managing Architectural Technical Debt", authors Robert L. Nord, Ipek Ozkaya, Philippe Kruchten, and Marco Gonzalez-Rojas offer a metric to measure technical debt based on dependencies between architectural elements.

This method shows how an organization should plan development cycles while taking into account the effect that accumulating technical debt will have on the overall resources required for each subsequent version released.

Though this article was published nearly 10 years ago, its relevance today is hard to overstate–in March 2022, it was awarded the "Most Influential Paper" Award at the 19th IEEE International Conference on Software Architecture (ICSA 2022).

Image source: Twitter

The scope of this paper does not include, however, what constitutes an architectural element or how architects and developers working with monolithic applications can actually calculate technical debt proactively.

What is needed are technologies purpose-built for application modernization to help organizations understand technical debt and make informed decisions about prioritizing projects to refactor critical monolithic services into decoupled, independent microservices in the cloud. In the next section, we describe how we've taken these academic concepts and applied them to build a technology platform for architects and developers to assess and measure technical debt.

Tackling Technical Debt with Math, Machine Learning, and Data-driven Business Cases: The vFunction Method

Using Data Science and Machine Learning to Determine Technical Debt

Determining technical debt is key to making decisions regarding any specific application and prioritizing modernization initiatives between multiple applications. As such, vFunction has created a method that can be used to not only compare the performance of different design paths for a single application, but also compare the technical debt levels of multiple applications at an arbitrary point in their development life cycle.

Our technology platform measures the technical debt of an application based on the dependency graph between its classes. Next, the platform performs multi-faceted analysis on the graph to determine a score that describes the technical debt of the application. Here are some of the metrics extracted from the raw graph:

1. Average/median outdegree of the vertices on the graph
2. Top N outdegree of any node in the graph
3. Longest paths between classes

Using standard clustering algorithms on the graph, our platform identifies communities of classes within the graph and measures additional metrics on them, such as the average outdegree of the identified communities and the longest paths between communities.

By using these generic metrics on the dependency graphs, our platform uncovers architectural issues which represent real technical debt in the original code base. Moreover, by analyzing dependencies on these two levels, class and community, we give a broad interpretation of what an architectural element is in the real world–helping to better quantify technical debt and prioritize which applications should be modernized first.

This is an intelligent approach to identifying technical debt and building a data-driven business case for prioritizing specific applications for modernization. With this approach, the top reasons for organizational pushback (i.e. risk, cost, fear of change) can be directly addressed based on real numbers, removing the opportunity for biases or political decisions to hobble modernization initiatives before they even start.

Identify and Assess Technical Debt in Real World

To test this method, we created a data set of over fifty real-world applications from a variety of domains (financial services, eCommerce, automotive, and others and extracted the aforementioned metrics from them. We used this data set in two ways.

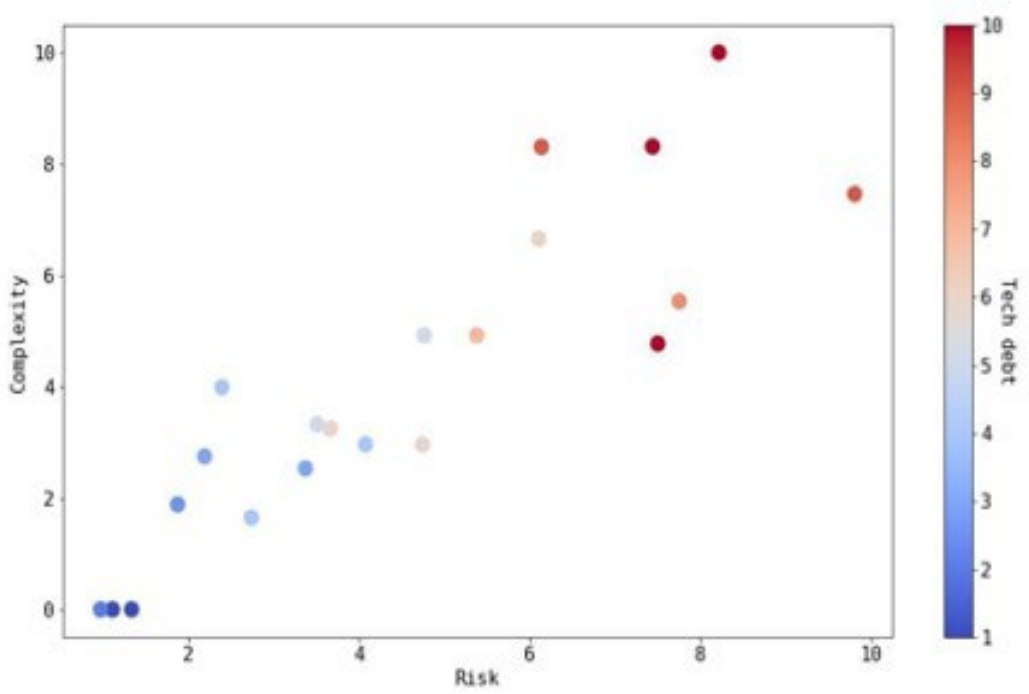
Firstly, we correlated specific instances of high-ranking occurrences of outdegrees and long paths with local issues in the code. For example, identifying god classes by their high outdegree. This proved efficient and increased our confidence level that this approach is valid in identifying local technical debt issues.

Secondly, we attempted to provide a high-level score that can be used not only to identify technical debt in a single application but also be able to compare technical debt between applications and to use it to help prioritize which should be addressed and how. To do that we introduced three indexes:

1. **Complexity -** represents the effort required to add new features to the application
2. **Risk -** represents the potential risk that adding new features will have downstream impacts on other parts of the application.
3. **Overall Debt -** represents the overall amount of extra work required when attempting to add new features

The graph depicts a sample of applications demonstrating the relationship between the aforementioned metrics.

To train the ML model, we manually analyzed the applications in our data set, employing expert knowledge of the individual architects and developers in charge of product development. From there, we then scored each application's complexity, risk, and



Source: vFunction, Inc. 2022

overall debt on a scale of 1-5 (where a score of 1 represents little effort required and 5 represents high effort). We used these benchmarks to train a machine learning model which correlates the values of the extracted metrics with the indexes and normalizes them to a score of 0-100.

This allows us to use this ML model to issue a score per index for any new application we encounter, enabling us to analyze entire portfolios of applications and compare them to one another and to our pre-calculated benchmarks.

vFunction Assessment Hub: Calculate Technical Debt

To increase innovation velocity and scalability, you must directly address accumulated technical debt across your application estate. Current manual assessment approaches are slow, complex, costly, and prone to failure—this makes building an accurate, data-driven application modernization plan extremely difficult.

Without the proper AI-driven assessment tools, it's nearly impossible for decision-makers to prioritize application modernization projects based on real data. To build a business case that reduces risk, improves project efficiency, and reduced cost, an analysis of the technical debt of monolithic applications, the accurate identification of the source of that debt, and a way to measure its negative impact on innovation are critical elements.

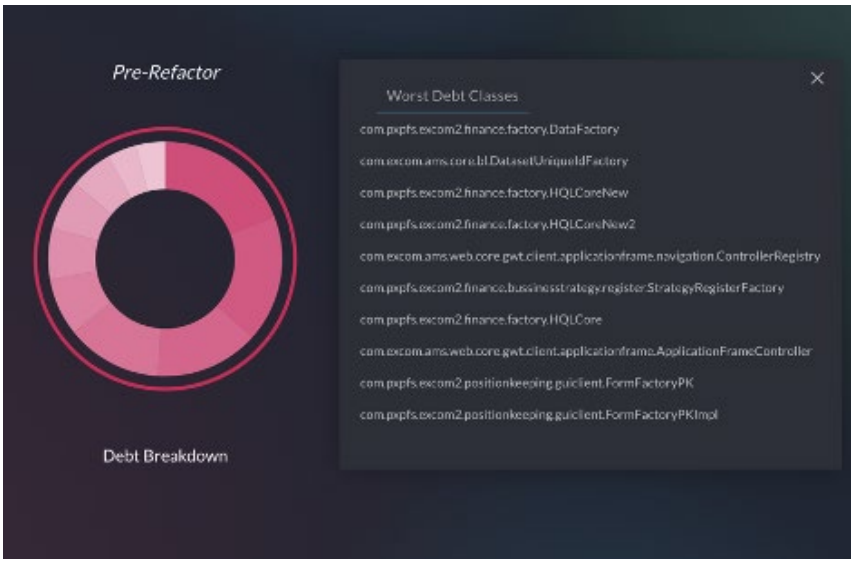
vFunction Assessment Hub leverages the data science and machine learning mentioned in the previous section and turns them into a lightweight assessment tool for calculating the technical debt of your monolithic applications. The goal is to help your organization build a business case and share a downloadable assessment report that offers concrete data needed to remove architectural technical debt from your applications.

1. Calculate

AI-trained algorithms calculate the technical debt of your monolithic applications, accurately identify the source of that debt, and measure its negative impact on innovation.

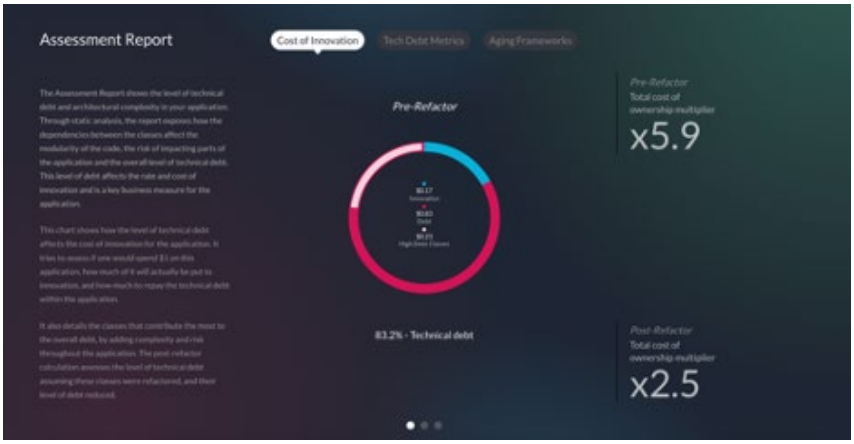
2. Prioritize

Clearly assess the benefits of modernization by understanding the cost of technical debt versus innovation, identifying the top 10 debt classes, and stack-ranking which apps to modernize first.



3. Analyze

Analyze and download key metrics that provide ROI and TCO measurements critical for more effective and compelling application modernization business cases.



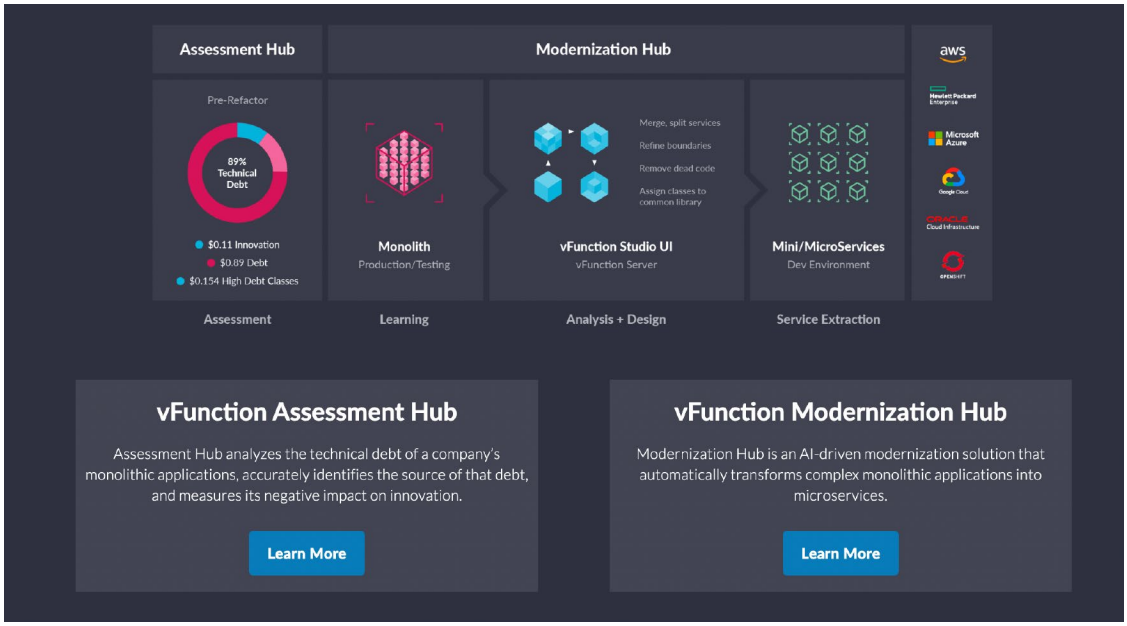
4. Modernize Immediately

Once your application modernization priorities have been determined, you can directly move to refactoring, rearchitecting, and rewriting applications with the **vFunction Modernization Hub**.

Conclusion

Technical debt is a major barrier to achieving engineering velocity and a high pace of innovation, the primary goals of both executives and architects for application modernization. As we've seen, 79% of modernization projects fail at high cost and extended timelines due to a lack of data for accurately setting expectations and intelligent tooling.

Using data science and machine learning, vFunction offers a data-driven solution to eliminating technical debt that includes the ability to identify, quantify, prioritize, and make a clear business case with ROI for a modernization initiative.



vFunction

Request a Demo

About vFunction

vFunction is the first and only AI-driven platform for developers and architects that intelligently and automatically transforms complex monolithic Java applications into microservices, restoring engineering velocity and optimizing the benefits of the cloud. Designed to eliminate the time, risk and cost constraints of manually modernizing business applications, vFunction delivers a scalable, repeatable factory model purpose-built for cloud native modernization. With vFunction, leading companies around the world are accelerating the journey to cloud-native architecture and gaining a competitive edge. vFunction is headquartered in Palo Alto, CA, with offices in Israel. To learn more, visit vFunction.com.