vFunction

Observability for Architects: A Contrarian Approach to Proactive Application Modernization

Does Architecture Still Matter?

A very controversial question - does architecture still matter? It's one that Enterprise Architects, CTOs, and Software Architects alike are struggling with everyday. Some are even asking whether the role of the architect is relevant anymore. As the majority of software development lifecycles (SDLCs) have shifted to Agile methodologies, even the Agile Manifesto poses this as a key principle:

"The best architectures, requirements, and designs emerge from self-organizing teams."

Thus, architects and their broader teams are struggling to keep up with the increasing engineering velocity as the actual state of the architecture of these apps drifts from their original desired state, taking on more and more architectural technical debt every sprint. In the absence of architectural direction and oversight, technical debt and architectural drift have worsened as the role of architecture and architects has been subsumed and overrun by pace, velocity, inattention, and dilution.

The result? Technical debt disasters are escalating - from Southwest Airlines to the FAA to Twitter - and their impact is being felt in boardrooms, stock prices, and lost customer confidence.

RELATED: Q&A with Bob Quillin - Technical Debt Risk: A Look into SWA, the FAA, and Twitter Outages

What Is Architectural Technical Debt?

Technical debt, or more specifically, architectural technical debt, is the accumulation of development shortcuts to meet increasingly complex system requirements and deadlines resulting in "big ball of mud" applications, diminishing innovation, and causing eventual downstream disasters. Gartner says that by 2026, 80% of technical debt will be architectural technical debt.[1] Not unlike financial debt, architectural technical debt continues to grow until it's finally paid in full. But you can fix architectural anomalies the right way instead of adding more debt and increasing the odds of catastrophic system failure.

Technical debt doesn't happen suddenly; it develops over time and is the accumulation of ongoing architectural drift which creates complexity and entanglement of the application domains. Until somewhat recently, technical debt hasn't been a common business term outside of IT walls. Engineers and application owners typically know it's growing in the shadows, but it hasn't been something often discussed with the business for a variety of reasons. Most notably, the debt becomes such a tangled ball of yarn with so many contributors over time, no one knows where to start or even has the resources in skill, time, and budget to even try. And if they do, they're often terrified of making a mistake. And there's no one to help decipher the mess. Those who took the shortcuts over the years have long gone.

Architectural technical debt isn't just a headache. It can and does reveal itself in devastating ways, often at the worst times when there is increased system demand. You can avoid such risk by learning from others' mistakes and taking a proactive approach to establishing a standard protocol for managing architectural technical debt.

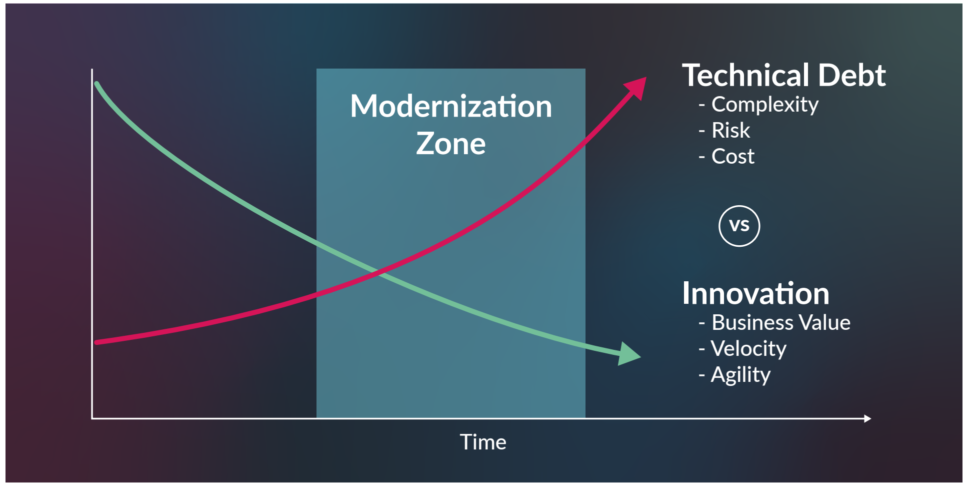
"Often when a particular symptom in a system is described as technical debt, it's not just the code quality that's bad, but it's also accumulating problems that happen in terms of architectural changes that have occurred throughout the systems development." - Carnegie Melon

How Most Enterprises Manage Architecture Drift

There is a cost to technical debt. As technical debt increases, innovation decreases, engineering velocity slows to a crawl, and agility goes out the window. The product team is working with one hand tied behind their back, unable to execute key initiatives because the application is too slow, too hard to change, and costs too much to run. The only way to speed up innovation is to add more shortcut "fixes" — bandaids that are only patching things temporarily and yes, adding more technical debt that someone at some point will have to deal with but "not now because we simply don't have time."

For most organizations, it's next to impossible to tackle technical debt while simultaneously focusing on innovation. Because technical debt doesn't always sound the alarm bells until it's really bad, many organizations opt to dedicate limited resources to sexier innovation that brings high fives and sometimes, a nice bump to revenue.

You can leave technical debt to fester or leave it for the next leadership team to tackle, but make no mistake, it doesn't go away on its own. It has to be dealt with at some point, preferably when the business isn't in full crisis mode. We can look at at least three prominent organizations that hadn't yet dealt with their technical debt to see how ignorance and deferral can lead to a corporate, customer, and PR nightmare.



Case Study: Southwest Airlines

Southwest

Southwest Airlines' network was hit hard by an architectural technical debt issue, only to be gut punched shortly after with yet another similar debacle. The first issue came during the airline's busiest season — Christmas. A legacy software problem led to nearly 17,000 flight cancellations for two million passengers at a cost of more than $1 billion.

The second was due to a firewall failure that prevented thousands of flights from taking off. The network outage lasted only an hour, but, as often happens with technical debt, an issue in one part of the system had a cascading effect on multiple other systems because of the interdependencies.

Both failures led the vice president of the Southwest Airlines Pilots Association to admit the airline problems were due to "chronic under-investment in tech infrastructure." The New York Times called it "shameful" and said, "This problem — relying on older or deficient software that needs updating — is known as incurring technical debt, meaning there is a gap between what the software needs to be and what it is. While aging code is a common cause of technical debt in older companies — such as with airlines, which started automating early — it can also be found in newer systems, because software can be written in a rapid and shoddy way, rather than in a more resilient manner that makes it more dependable and easier to fix as you expand. As you might expect, quicker is cheaper."

Southwest knew they had technical debt problems for years, with various employee groups writing open letters, picketing, sending emails to head honchos, and going to the media. Those pleas were ignored or at least shelved.

The airline is finally taking technical debt seriously, saying they are addressing it, but it isn't an easy fix at this point. The problem is so extensive, there is no telling how long it will take to modernize properly. And beyond the system upgrade costs, there has been immeasurable damage to the brand, with some swearing they will never fly the airline again. Only time will tell whether Southwest will regain its favor and market share, but it could have avoided much of these catastrophes had it addressed its technical debt risk years ago.

Case Study: FAA

Federal Aviation Administration

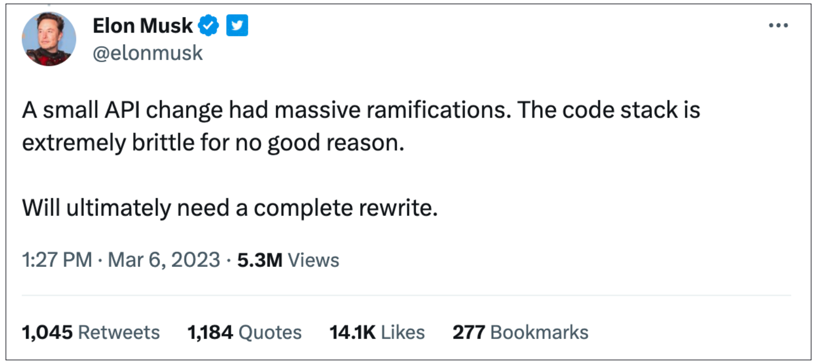
The FAA suffered a computer failure just weeks after the Southwest Airlines fiasco, canceling 1,200 flights and delaying 7,800 flights. This time, the issue was due to contractors introducing errors in the core data used in a key system. As soon as technicians detected a problem, they were able to move to a redundant system, but that system was trying to access the same corrupted data, so they couldn't immediately recover.

How does a coding error point to technical debt? In this case, the FAA system was an older monolithic system, where code changes cannot be decoupled from other code. One change can result in a system crash. Had the system been modernized into microservices, they could have made a single change to a single microservice that wouldn't have caused issues elsewhere.

Case Study: Twitter

A similar situation occurred at Twitter, where a single coding error broke the social media platform for a couple of hours. Twitter had been offering free access to its API but sought to eliminate that privilege and tasked a single engineer to change the code. And that they did, but the change had what the Verge calls "cascading consequences inside the company, bringing down much of Twitter's internal tools along with the public-facing APIs." Elon Musk tweeted later that "The code stack is extremely brittle for no good reason. Will ultimately need a complete rewrite."

For Twitter, technical debt was making the entire architecture vulnerable to breakage. With new management came code change requests, as often happens with leadership changes. And instead of fixing the architectural technical debt, they focused on adding requested capabilities, or in this case, removing one. Again, interdependencies in a monolithic architecture caused unforeseen downstream effects. If we know anything about Musk, it's that he's an innovator. But if Twitter's applications are old monoliths, code can't change quickly for the cloud, impeding innovation.

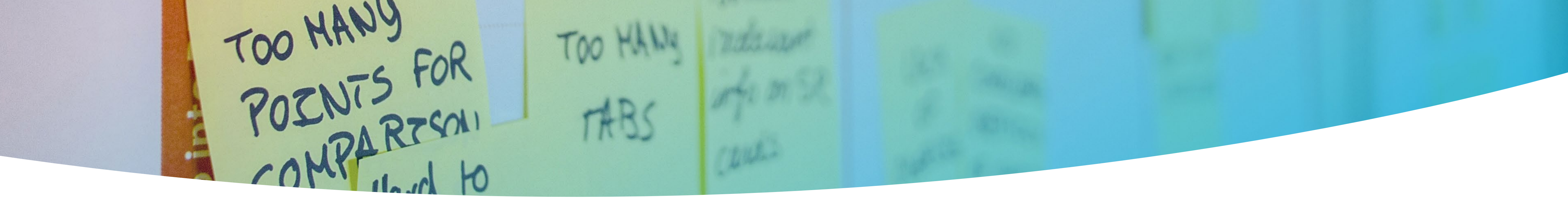

Elon Musk
@elonmusk

A small API change had massive ramifications. The code stack is extremely brittle for no good reason.

Will ultimately need a complete rewrite.

1:27 PM · Mar 6, 2023 · **5.3M** Views

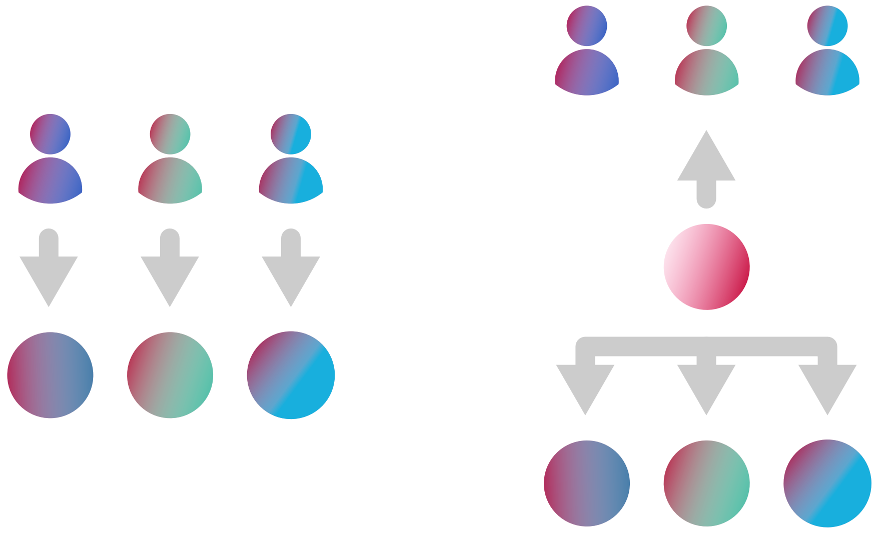
1,045 Retweets **1,184** Quotes **14.1K** Likes **277** Bookmarks



Do These Red Flags Sounds Familiar?

Highlighting these case studies illustrates how different system problems can be traced back to unchecked technical debt. There were early risk signs in each case, plenty of them, but they were either ignored, pushed off to a rainy day, or completely missed because no one was paying attention or had the historical technical know-how to fix them properly.

These are obviously large entities with publicized incidents, but organizations of all sizes are at risk because of their dependence on ever-evolving software, shifting to the cloud, and the constant pressure to innovate. These driving factors require incremental changes to code within monoliths, which lead to architectural debt.

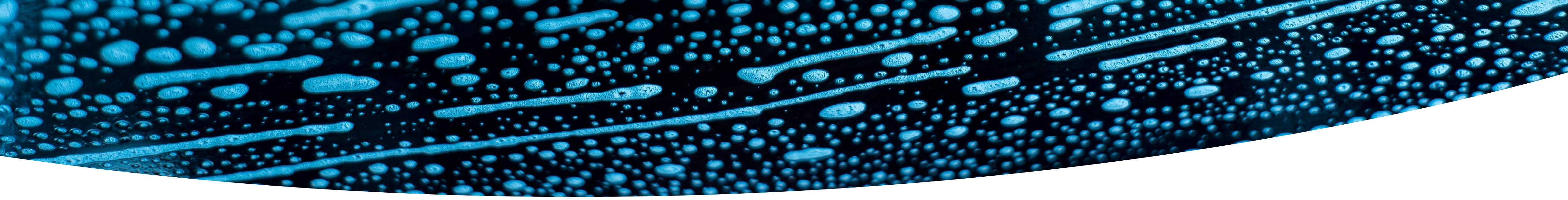


The warning signs vary, but some of the biggest red flags you might see are:

- Key requested features are consistently delayed or stalled
- Your feature backlog continues to grow while modernization updates are deferred
- You're losing market share to more agile competitors
- Your operational costs continue to spiral up
- You are slow to respond to change — your engineering agility and velocity are slow and frustrating

Another red flag is architectural drift and erosion, where there is a significant difference between the desired architectural state and the actual architecture itself. When increasingly more features are added, for example, the code and app become brittle. Issues can arise because of the lack of good processes. Developers come and go, constantly adding, tweaking, or trying to fix something without much forethought as to how their partial work impacts the whole.

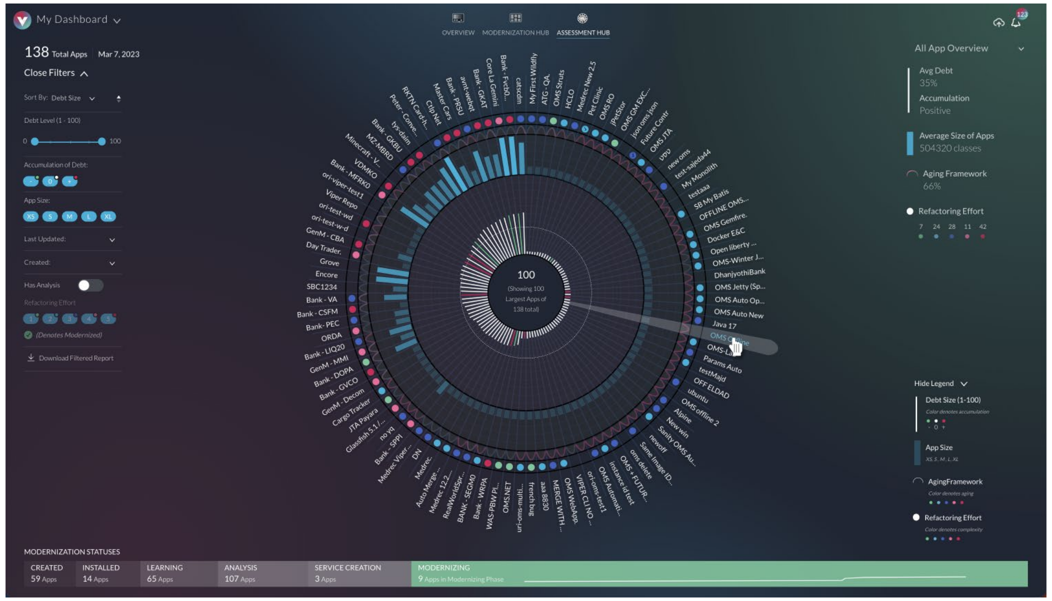
All of these scenarios are common and tied to your inability to innovate to keep up with innovation and competitors. Instead, there should be a cultural shift that prioritizes transforming certain monoliths into microservices that are much easier to manage and manipulate without causing collateral damage. With microservices, you can isolate issues and minimize the cascading effects of changes.



Observability for Architects: A Contrarian Approach

Most monoliths are struggling to keep up in today's high-velocity, agile environments. They're too brittle, their features are too interdependent, and the infrastructure required to run them is expensive and can't keep up with the constant elasticity requirements, all of which further drives up costs and slows delivery. The typical approach is to defer modernization by lifting and shifting to the cloud or continue to muddle through with a monolith, crossing your fingers and hoping you can outlive it.

Architecture observability is a contrarian approach. It challenges the assumption that technical debt is unmanageable and just a fact of life. Instead of letting your software portfolio drift along without any oversight or control, observability for architects measures your baseline architectures, detects drift, and alerts you to major changes that can prevent the technical debt disasters described above.



Observability for Architects: Puts the Architect Back into the SDLC

With Observability for Architects, architects can detect specific drift issues when they happen and proactively address them. This approaches the modernization problem **incrementally** by decomposing the process into stories or work items that can be placed into sprints and backlogs. Architects can then **actively engage** in the agile SDLC process, detecting architectural drift early and addressing it as they go.

The first step is to actually measure your technical debt risk by analyzing each app across your application estate.

- How complex is your architecture and how long are the dependency chains?
- What is the risk associated with making a change?
- Which applications do you need to modernize first?
- How close are you to a tech debt disaster?
- What is the most effective way to modernize?
- How much will modernization cost?
- What are the costs and risks of NOT modernizing?

To answer these questions, architects need visibility and data, both of which will be required to incrementally fix architecture drift issues early and gradually build better modernization business cases. Eliminating technical debt and modernizing your architecture takes time, money, and people, so you must justify the investment. But building a business case is often the biggest roadblock to reducing your technical debt backlog. It can be tremendously time-consuming and frustrating without the right tools.

Fortunately, vFunction offers the automation, visualization, and data that architects need to finally shed light on the risks and opportunities in terms the rest of the business can understand — how such an investment will fuel innovation, agility, and resilience.

vFunction: Observability for Architects

Baselining Technical Debt Across Your Software Portfolio

The vFunction Continuous Modernization Platform automatically analyzes your technical debt risk among your monolithic applications, providing you with its source and its impact on innovation, engineering velocity, and agility. You'll finally understand your application estate complexity and the technical debt that's hamstringing growth and profitability, plus a prioritized list of what to modernize.

With vFunction, you can build a modernization plan and a complete modernization strategy that is built on data-driven measurements and realistic time estimates. Depending on the number and complexity of your apps, you can get the visibility you need in less than an hour or just a few weeks so you can move on to fixing issues.

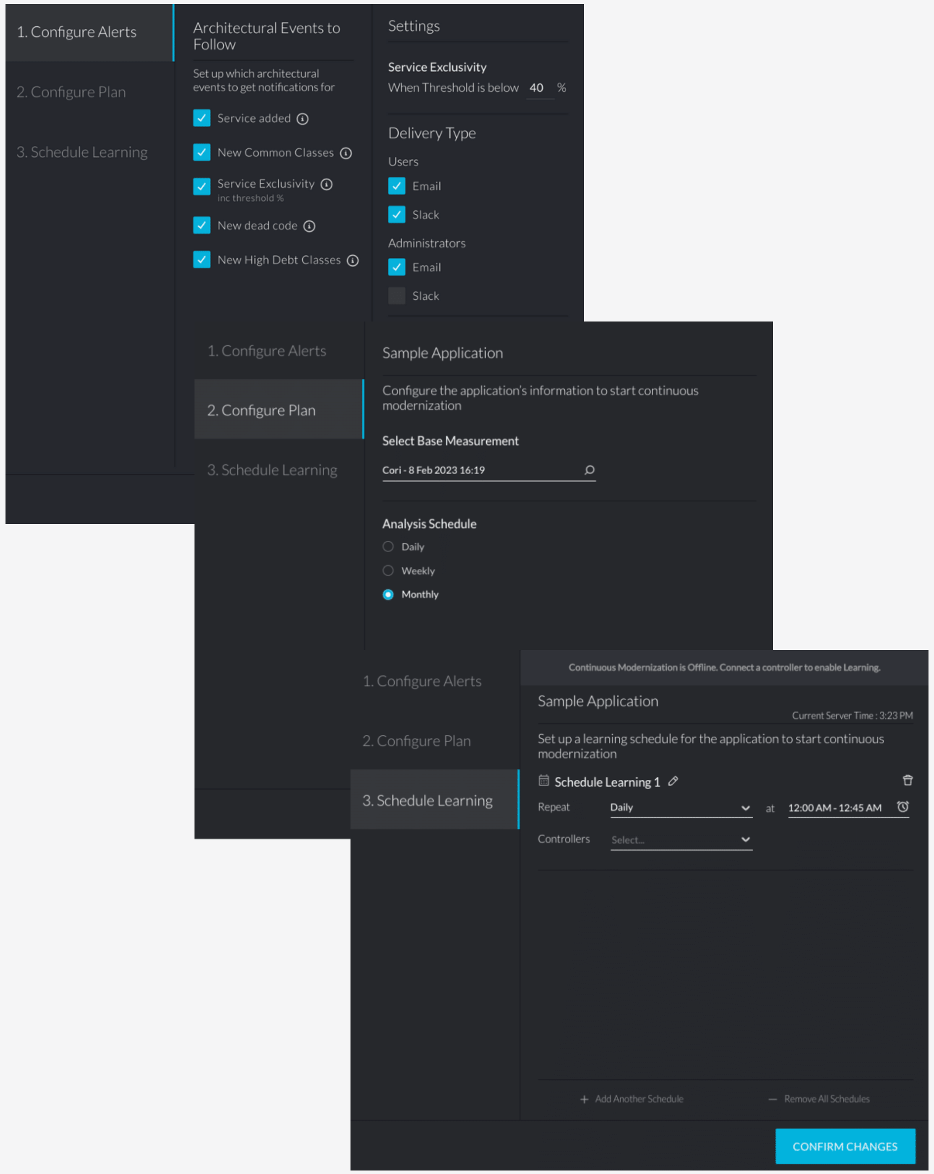
The beauty of vFunction is we go beyond analysis to provide the platform to take the next step: converting those high-risk monoliths into microservices.

Architectural Observability to Proactively Detect Drift

Adopting a culture of continuous modernization is key to incrementally identifying architectural technical drift from sprint to sprint, release to release.

The vFunction Architectural Observability Manager is a powerful observability solution for architects to manage debt, providing visibility into anomalies before they cause more serious consequences. As the first shift-left solution for architects, it is ideal for monitoring, finding, and quickly fixing application modernization and architectural drift issues yourself or with the vFunction Refactoring Engine as they arise.

Incremental and progressive modernization can be an effective tool in the battle against architectural drift and stagnant modernization initiatives. By using architectural observability to chip away at the problem, agile modernization can become a reality versus an oxymoron.



Architectural Events to Follow

Set up which architectural events to get notifications for

- [✓] Service added
- [✓] New Common Classes
- [✓] Service Exclusivity
  inc threshold %
- [✓] New dead code
- [✓] New High Debt Classes

Settings

Service Exclusivity
When Threshold is below 40 %

Delivery Type

Users
- [✓] Email
- [✓] Slack

Administrators
- [✓] Email
- [ ] Slack

Sample Application

Configure the application's information to start continuous modernization

Select Base Measurement

Cori - 8 Feb 2023 16:19

Analysis Schedule
- ○ Daily
- ○ Weekly
- ● Monthly

Continuous Modernization is Offline. Connect a controller to enable Learning.

Sample Application

Current Server Time : 3:23 PM

Set up a learning schedule for the application to start continuous modernization

Schedule Learning 1

Repeat: Daily at 12:00 AM - 12:45 AM

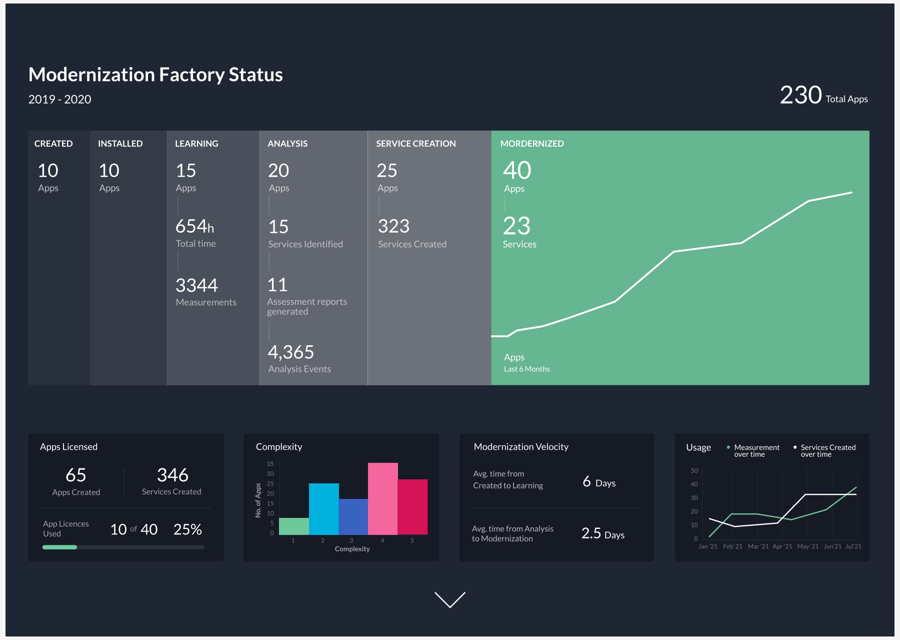
Controllers: Select...

Transforming the Architecture

Once it's time to refactor, vFunction Refactoring Engine helps you re-architect and break down the monolith into microservices.

Instead of the common lift-and-shift approach that fails to leverage the true value of cloud-native architectures and doesn't fix any problems, our platform offers the ability to refactor, re-architect, or rewrite your complex monolithic applications into cloud-native architectures built for the cloud. The Refactoring Engine is comprehensive, providing invaluable intelligence and AI to speed transformation while reducing the risk of errors that can cause downstream problems.

Finally, architects can iteratively design and deploy architectural improvements by untangling dependencies at the business logic level. Applications are now flexible instead of brittle. The ultimate result? Increased engineering velocity to keep innovation humming.



¹ Gartner, Measure and Monitor Technical Debt With 5 Types of Tools, Tigran Egiazarov, Thomas Murphy, 27 February 2023

GETTING STARTED

The old way of just playing defense is stressful and tests agility. With the rapid pace of application evolution, taking a contrarian approach to managing technical debt through continuous modernization efforts is a more proactive approach that directly attacks the problem let alone a better use of resources. Easier said than done. But with a solution like vFunction, you finally have what you need to fully understand the situation, incrementally chip away at technical debt and modernization, build a strong business case for apps needing modernization, and transform your organization from vulnerable to safeguarded.

If you are ready to take the first step, we invite you to try our AO Manager Express. With this rapid, cloud-based solution, you can assess up to three Java monolithic applications for free to calculate their complexity and your technical debt risk and modernization opportunities. Technical debt is there. The question is, will it stop with you? Architects have a unique opportunity to engage more actively in an agile SDLC, make architecture matter again, and in turn provide significant value to the business. Contact us for more information or request a demo today.