vFunction

WHITE PAPER

Enhancing System Reliability

How architectural events support shifting left for application resiliency, scalability, and engineering velocity

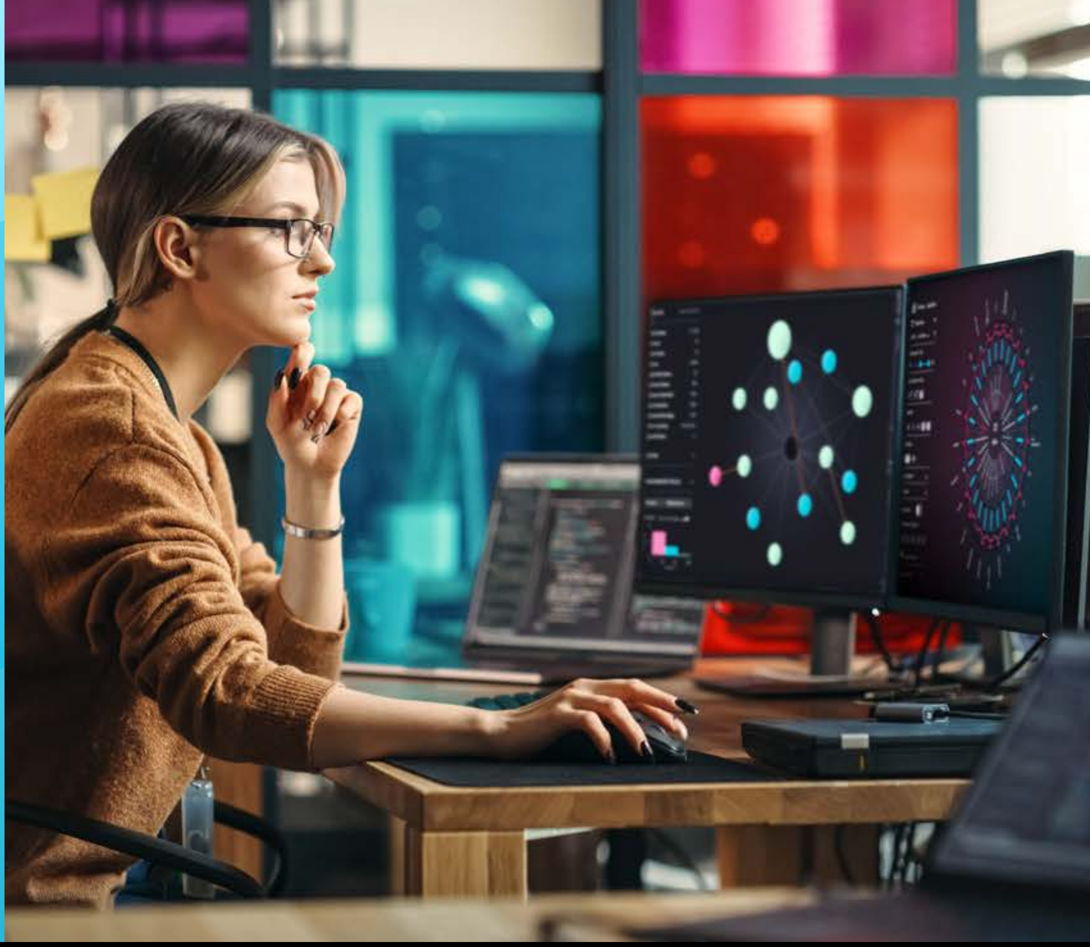
Resiliency

Scalability

Velocity

Table of Contents

Introduction



Engineering organizations constantly search for ways to increase application release velocity, improve reliability, and address scalability challenges. While there are many ways that teams try to address these issues, they are primarily chasing symptoms.

The root cause for most applications — from monoliths to complex microservice deployments — is the underlying software architecture. Addressing architectural drift and detecting technical debt problems as they accumulate empowers application teams to proactively address the core problems affecting their key performance indicators (KPIs) and metrics.

Architectural events enable engineering teams to detect when software architecture changes occur, assess their impact on critical engineering KPIs such as resilience, scalability, release frequency, and cloud readiness, and pinpoint where, how, and why to fix them.

What are architectural events?

Architectural events provide crucial insights into the changes and issues that impact application architectures. They identify specific areas of high technical debt, monitor architectural quality, and catch potential issues before they escalate into significant problems or help remediate them if they already impact application resiliency. These events are vital indicators of an application's architectural health and are essential to raising software engineering quality and increasing business innovation.

Architectural observability detects architectural drift and the events that caused it by providing real-time visibility into how applications are structured. It addresses sources of technical debt and connects the dots between architectural decisions and business priorities. Engineering leaders and software architects use it to increase developer productivity, continuously modernize applications, and maximize cloud benefits.

By continually analyzing application architecture, architectural observability identifies architectural drift and technical debt and makes intelligent recommendations to fix them. Continuous observation and analysis are necessary for accurate visibility into architectural health and technical debt.

Architectural events are detected by routinely analyzing applications' binaries statically in the CI/CD and dynamically in production to understand architecture, observe drift, and find and fix architectural technical debt. They detect changes in software architecture variables, including:

Domains
The addition or deletion of business domains in the architecture

Dependencies
Cross-domain, circular, and high densities of dependencies

Complexity
A composite measurement reflecting the effort required to maintain and refactor the architecture

Modularity
A critical factor in enabling scalability and adoption of cloud-native container services

Dead code
Detection of reachable but unused or obsolete code through runtime analysis

Resources
Cross-domain use of resources such as database tables, files, or sockets

Classes
Dynamic, static, and common class dependencies identification

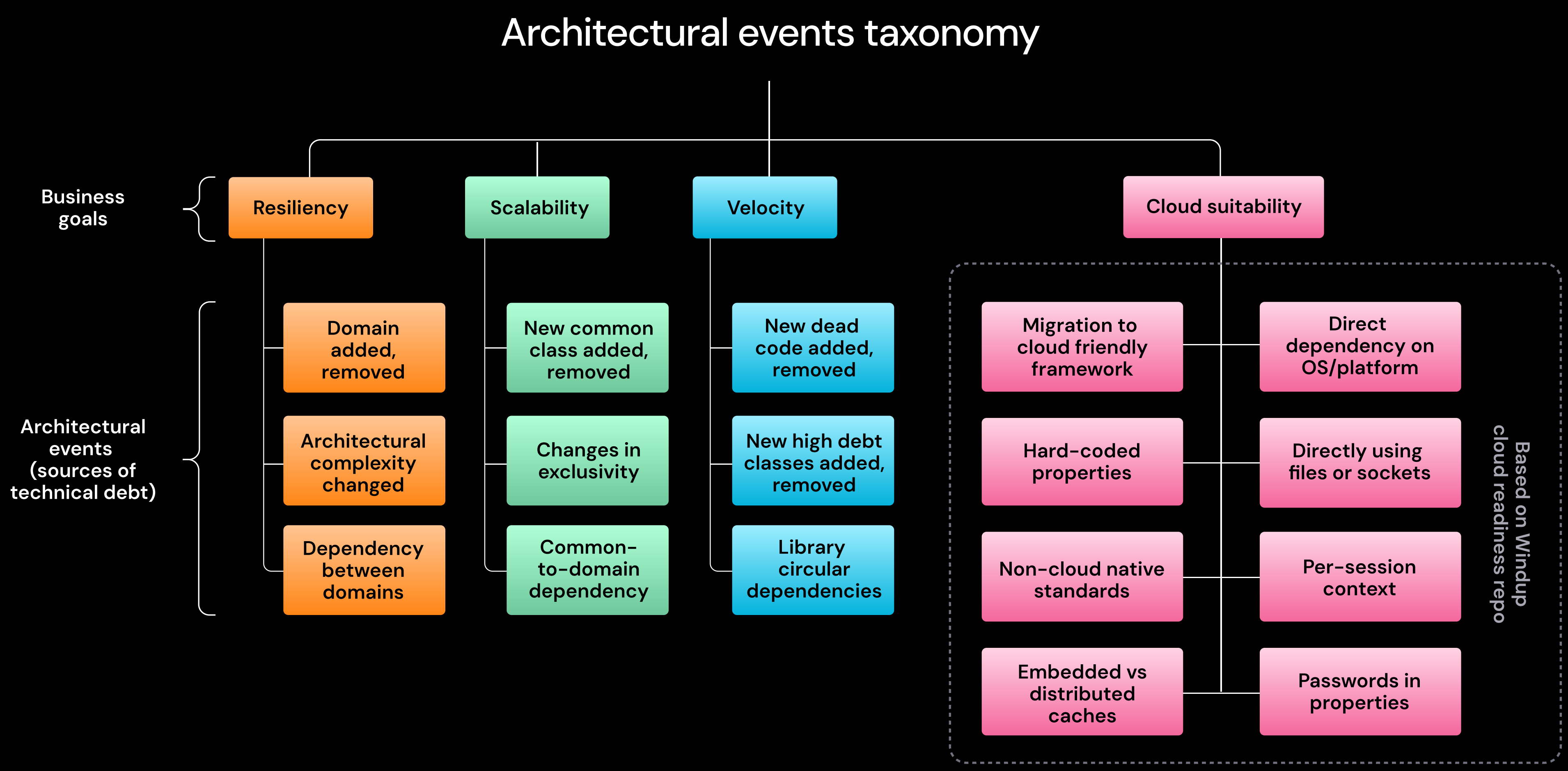
Frameworks
Aging frameworks have a significant impact on cloud readiness, security, and reliability

How architectural events improve engineering excellence

Bad architecture doesn't just happen. It accumulates over time and requires active management, akin to handling any critical health, fitness, or financial debt responsibility. To prevent the buildup of poor architecture choices — be they unintended or purposeful — architectural events detect and pinpoint the source of the problem and identify how they impact your engineering goals and business objectives. Figure 1 below illustrates an architectural events taxonomy organized by business goals:

- **Resiliency:** increase application stability, reliability, and recovery
- **Scalability:** support increased or dynamic workloads to accommodate growth and demand without compromising performance
- **Velocity:** accelerate release frequency and decrease friction
- **Cloud readiness:** streamline migration to and adoption of modern cloud-native services

By focusing on architectural events affecting their engineering and business KPIs, leaders can determine what to fix first.



Architectural events taxonomy
Business goals
Resiliency
Scalability
Velocity
Cloud suitability
Architectural events (sources of technical debt)
Domain added, removed
Architectural complexity changed
Dependency between domains
New common class added, removed
Changes in exclusivity
Common-to-domain dependency
New dead code added, removed
New high debt classes added, removed
Library circular dependencies
Migration to cloud friendly framework
Direct dependency on OS/platform
Hard-coded properties
Directly using files or sockets
Non-cloud native standards
Per-session context
Embedded vs distributed caches
Passwords in properties
Based on Windup cloud readiness repo

Figure 1: Architectural events taxonomy

Resiliency

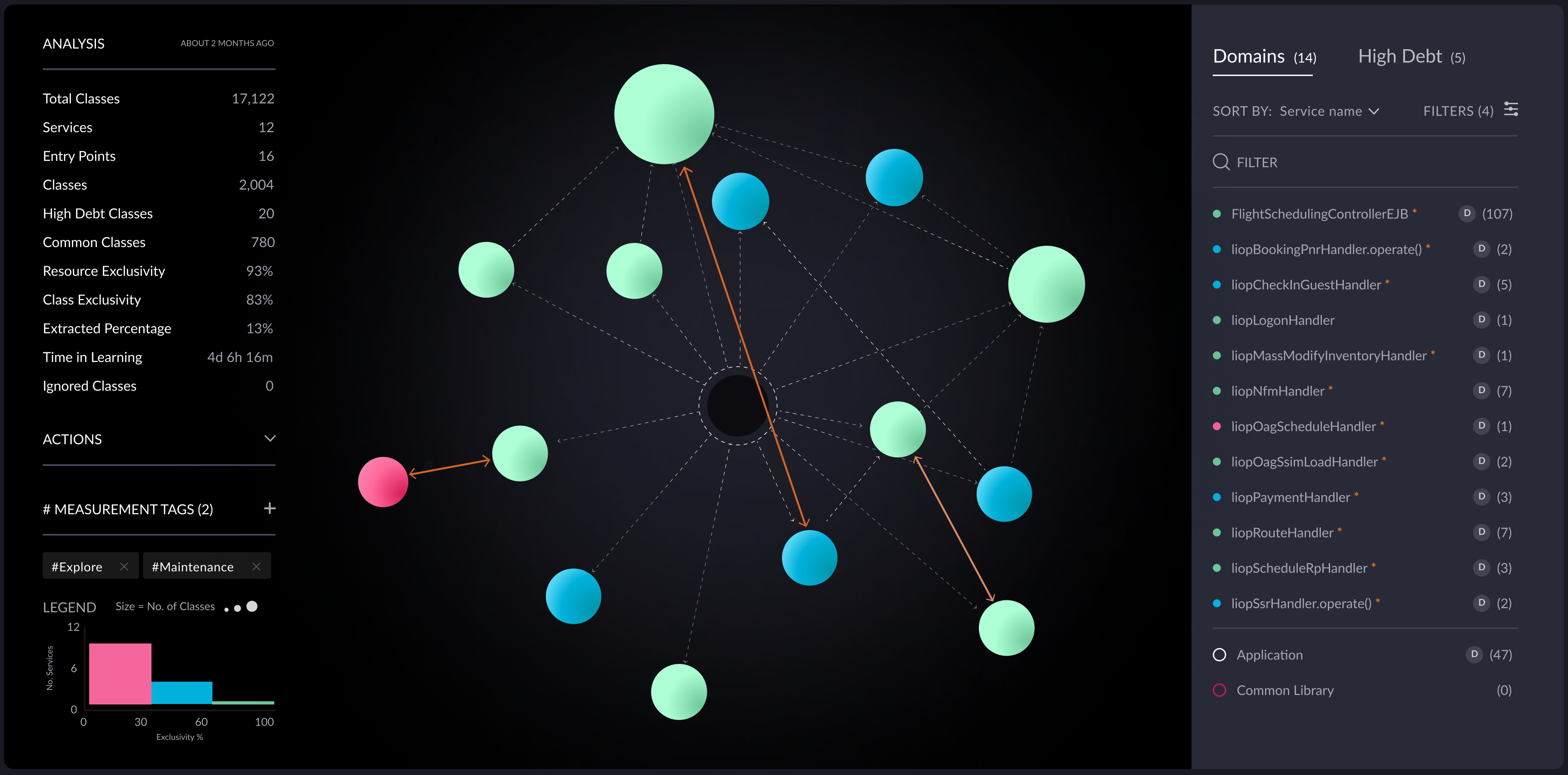
Architectural events that impact application resiliency

Application resiliency entails the ability of a workload to recover from infrastructure or service disruptions, dynamically acquire computing resources to meet demand and mitigate disruptions, such as misconfigurations or transient network issues. In a modern cloud environment, applications use high-availability services, such as auto-scaling, load balancing, microservice and container services, and DevOps methodologies to increase resiliency.

While the cloud provider is responsible for **providing** these services, the application organization is accountable for **building** a resilient architecture to take advantage of them.

Here are three architectural events that directly identify and impact application resiliency. Consider these when building or rearchitecting your applications.

Event: Domain added or removed



ANALYSIS	ABOUT 2 MONTHS AGO
Total Classes	17,122
Services	12
Entry Points	16
Classes	2,004
High Debt Classes	20
Common Classes	780
Resource Exclusivity	93%
Class Exclusivity	83%
Extracted Percentage	13%
Time in Learning	4d 6h 16m
Ignored Classes	0

● FlightSchedulingControllerEJB *	D	(107)
● liopBookingPnrHandler.operate() *	D	(2)
● liopCheckInGuestHandler *	D	(5)
● liopLogonHandler	D	(1)
● liopMassModifyInventoryHandler *	D	(1)
● liopNfmHandler *	D	(7)
● liopOagScheduleHandler *	D	(1)
● liopOagSsimLoadHandler *	D	(2)
● liopPaymentHandler *	D	(3)
● liopRouteHandler *	D	(7)
● liopScheduleRpHandler *	D	(3)
● liopSsrHandler.operate() *	D	(2)
○ Application	D	(47)
○ Common Library		(0)

Figure 2: vFunction domain topology view

DEFINITION	Introducing new domains or removing an existing domain leads to application instability.
DESCRIPTION	The discovery of a new domain or the removal of an existing one triggers this event, shown in Figure 2. This event affects the overall organization and system structure. Understanding when and why domains are added or removed helps architects and engineering teams assess the system's evolving needs and complexity.

IMPACT AND REMEDIATION

When adding a new domain, architects should evaluate its role and ensure proper encapsulation to maintain the system's modularity.

It's important to verify that the new domain doesn't duplicate existing functionality to prevent redundancy and maintain the system's cohesion and efficiency.

Removing a domain signifies a shift in the application architecture. This change could be due to various reasons, such as evolving business needs, optimization efforts, or refactoring. Regardless of the cause, architects must review the architecture to confirm the removal hasn't adversely affected the coherence of the application. They should validate the boundaries of the remaining domains to ensure they're intact, verify that they have adequately reassigned the responsibilities of the removed domain, and evaluate if the removal has introduced any orphaned elements or dependencies. This analysis helps maintain the application architecture's integrity, maintainability, and resiliency.

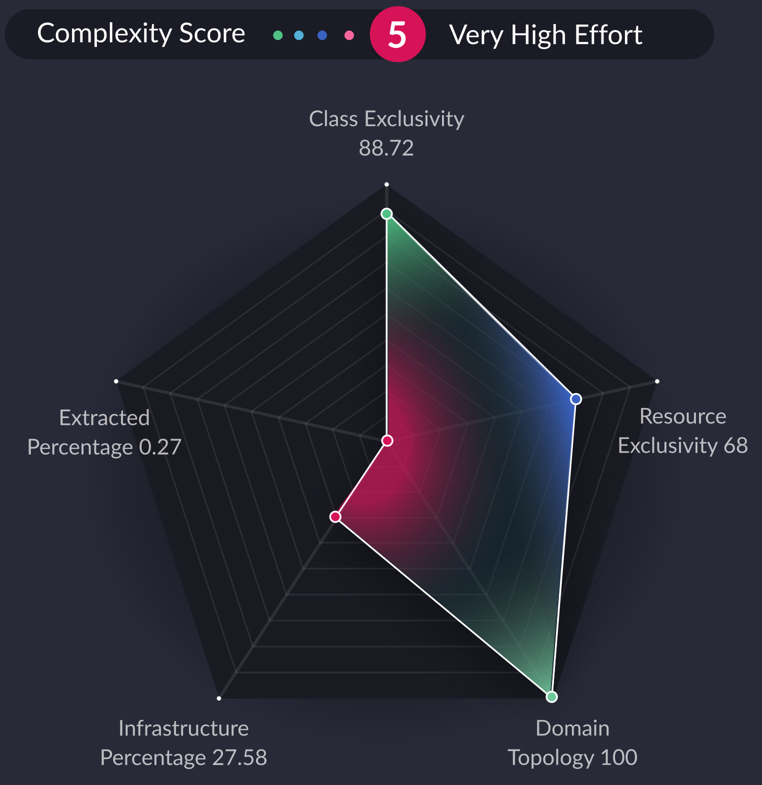
Event: Architectural complexity changed

DEFINITION

An application's architectural complexity reflects the effort required to maintain and refactor its architecture.

DESCRIPTION

As shown in Figure 3, architectural complexity is a weighted average of five metrics. The event triggers whenever there is a shift in this computed value, either an increase or a decrease in the composition of these five metrics:



Extracted percentage: The percentage of classes not required in the application after extracting the domains.

Class exclusivity: The percentage of classes exclusive to the domains.

Resource exclusivity: The percentage of resources exclusive to the domains.

Domain topology: The number of domain-to-domain calls required to call any domain.

Common percentage: The level of common classes found.

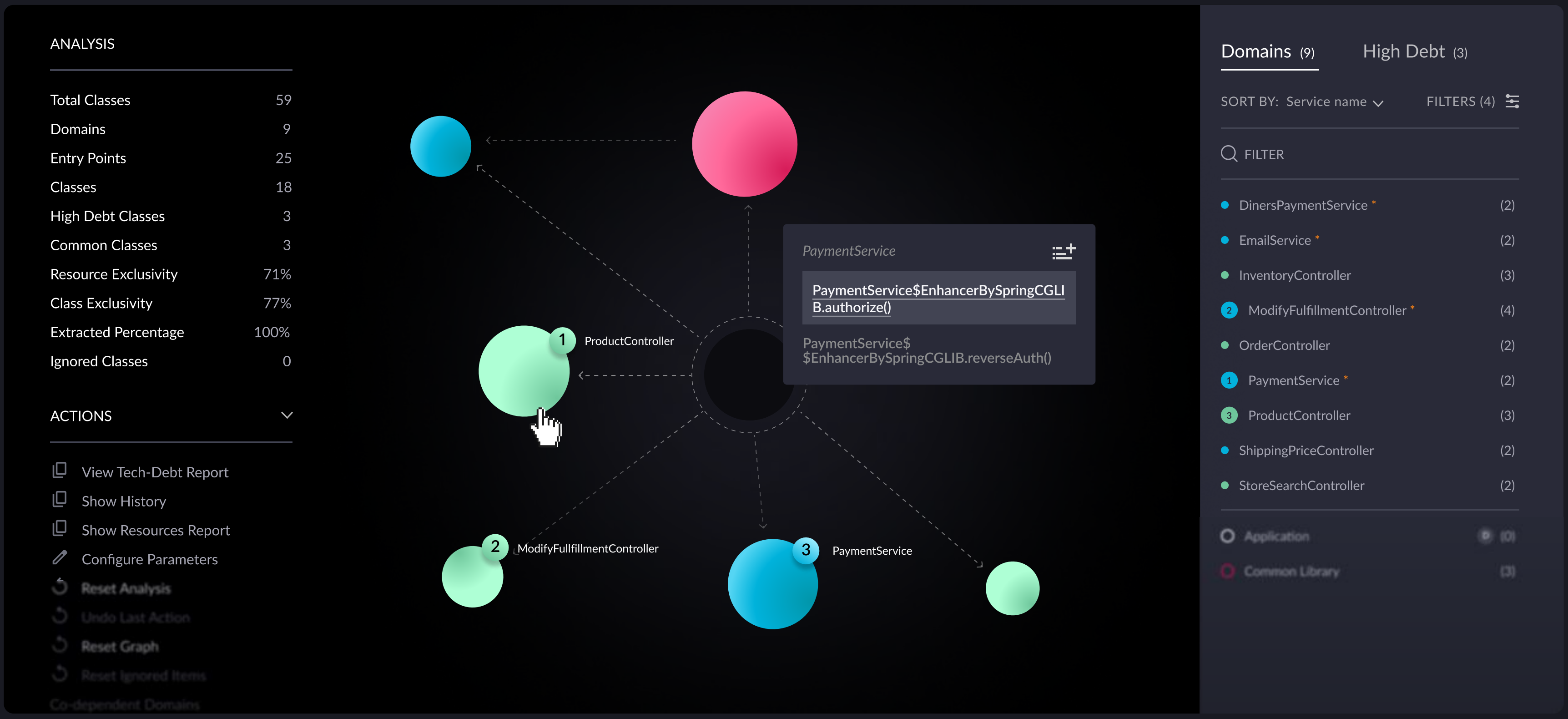
Figure 3: Architectural complexity as calculated and show in vFunction

IMPACT AND REMEDIATION

Architects should simplify areas identified as highly complex:

1. **Extracted percentage:** To improve extracted percentage, add entry points closer to the application's root.
2. **Class exclusivity:** To address class exclusivity problems, review and eliminate non-exclusive classes of domains where possible.
3. **Resource exclusivity:** Identify and reduce non-exclusive resources in the identified domains.
4. **Domain topology:** A high volume of domain-to-domain calls creates topological complexities that can be managed by reducing unnecessary inter-domain communication.
5. **Common percentage:** To increase, review the common classes, mark business logic classes as non-common and mark infra JARs (Java ARchive).

Event: Dependency between domains



ANALYSIS

Total Classes	59
Domains	9
Entry Points	25
Classes	18
High Debt Classes	3
Common Classes	3
Resource Exclusivity	71%
Class Exclusivity	77%
Extracted Percentage	100%
Ignored Classes	0

ACTIONS

- View Tech-Debt Report
- Show History
- Show Resources Report
- Configure Parameters
- Reset Analysis
- Undo Last Action
- Reset Graph
- Reset Ignored Items

Co-dependent Domains

PaymentService

PaymentService$EnhancerBySpringCGLIB.authorize()

PaymentService$$EnhancerBySpringCGLIB.reverseAuth()

Domains (9) High Debt (3)

SORT BY: Service name FILTERS (4)

FILTER

DinersPaymentService *	(2)
EmailService *	(2)
InventoryController	(3)
2 ModifyFulfillmentController *	(4)
OrderController	(2)
1 PaymentService *	(2)
3 ProductController	(3)
ShippingPriceController	(2)
StoreSearchController	(2)
Application	(0)
Common Library	(3)

Figure 4: Dependencies visualization

DEFINITION Detection of new dependencies between domains, as visualized in Figure 4, enables architects to address potential undesired domain dependencies that reduce resilience and stability.

DESCRIPTION Architects can simplify interactions between domains by highlighting new, changed, or unnecessary dependencies, leading to improved domain exclusivity and a more robust and reliable system.

IMPACT AND REMEDIATION Maintain architectural health and manage the growth of technical debt by identifying and addressing the introduction of new domain dependencies.

Scalability

Architectural events that impact application scalability

Gartner defines scalability as “the measure of a system’s ability to increase or decrease in performance and cost in response to changes in application and system processing demands.”

Software architecture significantly impacts how an application can scale based on the modularity of the system and its ability to take advantage of scaling technologies such as auto-scaling, load balancing, horizontal scaling, and containerization and orchestration services utilizing Kubernetes or serverless platforms.

To improve scalability in applications, address these architectural events first.

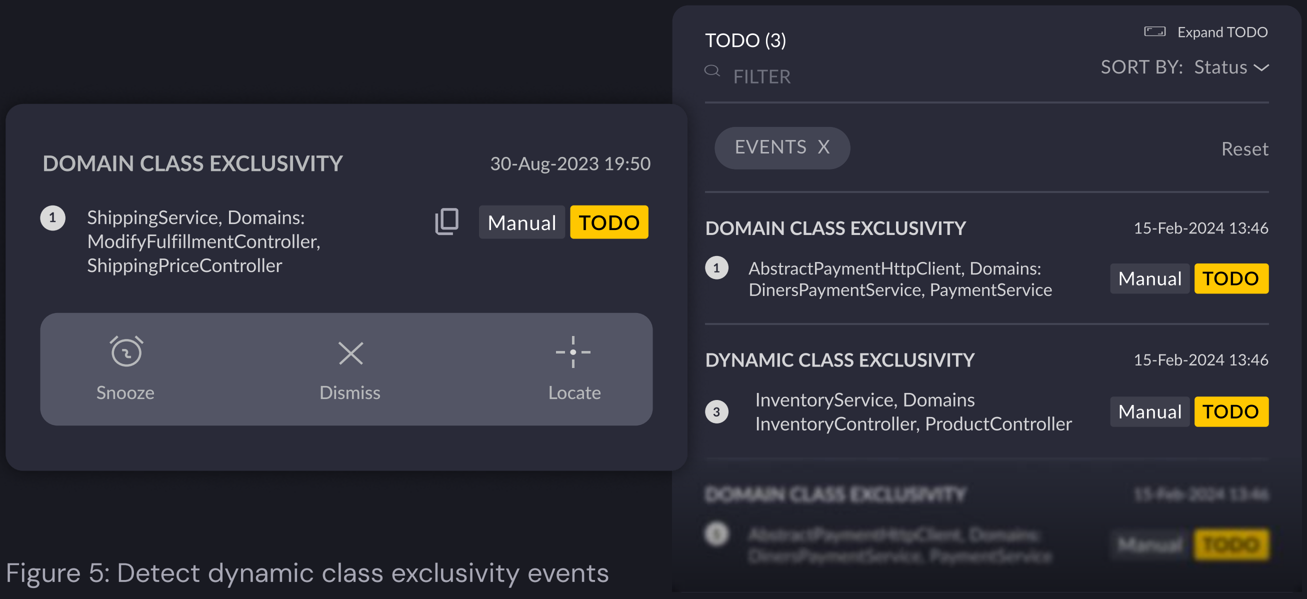
Event: New common classes added/removed

DEFINITION	This event monitors the introduction of new common classes or the removal of existing ones.
DESCRIPTION	Common classes are those used across multiple domains or functions. The misuse or overuse of common classes can lead to tight coupling and decreased modularity.
IMPACT AND REMEDIATION	Architects should consider adding an extensively used common class to its JAR or infra JAR. Tracking how different domains use these classes is crucial to avoid unnecessary dependencies and tight couplings.

Event: Common-to-domain dependency

DEFINITION	This event focuses on enhancing the management of compile-time dependencies by identifying and addressing dependencies involving common classes at the application entry points.
DESCRIPTION	This event aims to ensure that compile-time dependencies are tracked and managed. Continual tracking maintains the application architecture's modular structure and cohesiveness, which is crucial for its overall health and efficiency.
IMPACT AND REMEDIATION	In scenarios where an entry point establishes a dependency on a common class at compile time, highlighting these dependencies becomes critical. Undetected and unresolved dependencies lead to architectural inefficiencies and, subsequently, an increase in technical debt. This event's automatic detection capability plays a role in the early intervention of technical debt and keeps the application architecture efficient and aligned with best practices.

Event: Changes in exclusivity


DOMAIN CLASS EXCLUSIVITY 30-Aug-2023 19:50

1 ShippingService, Domains: ModifyFulfillmentController, ShippingPriceController — Manual TODO

TODO (3) — Expand TODO

FILTER — SORT BY: Status

EVENTS X — Reset

DOMAIN CLASS EXCLUSIVITY 15-Feb-2024 13:46

1 AbstractPaymentHttpClient, Domains: DinersPaymentService, PaymentService — Manual TODO

DYNAMIC CLASS EXCLUSIVITY 15-Feb-2024 13:46

3 InventoryService, Domains InventoryController, ProductController — Manual TODO

DOMAIN CLASS EXCLUSIVITY 15-Feb-2024 13:46

1 AbstractPaymentHttpClient, Domains: DinersPaymentService, PaymentService — Manual TODO

Figure 5: Detect dynamic class exclusivity events

DEFINITION

This event monitors changes in the exclusivity of specific classes, both dynamic and static, as well as resources like database tables, transactions, Spring beans, sockets, and files. In this context, exclusivity refers to whether a class or resource is confined to a single domain or dispersed across multiple domains.

DESCRIPTION

Changes in exclusivity reflect the shifting distribution of these elements across domains.

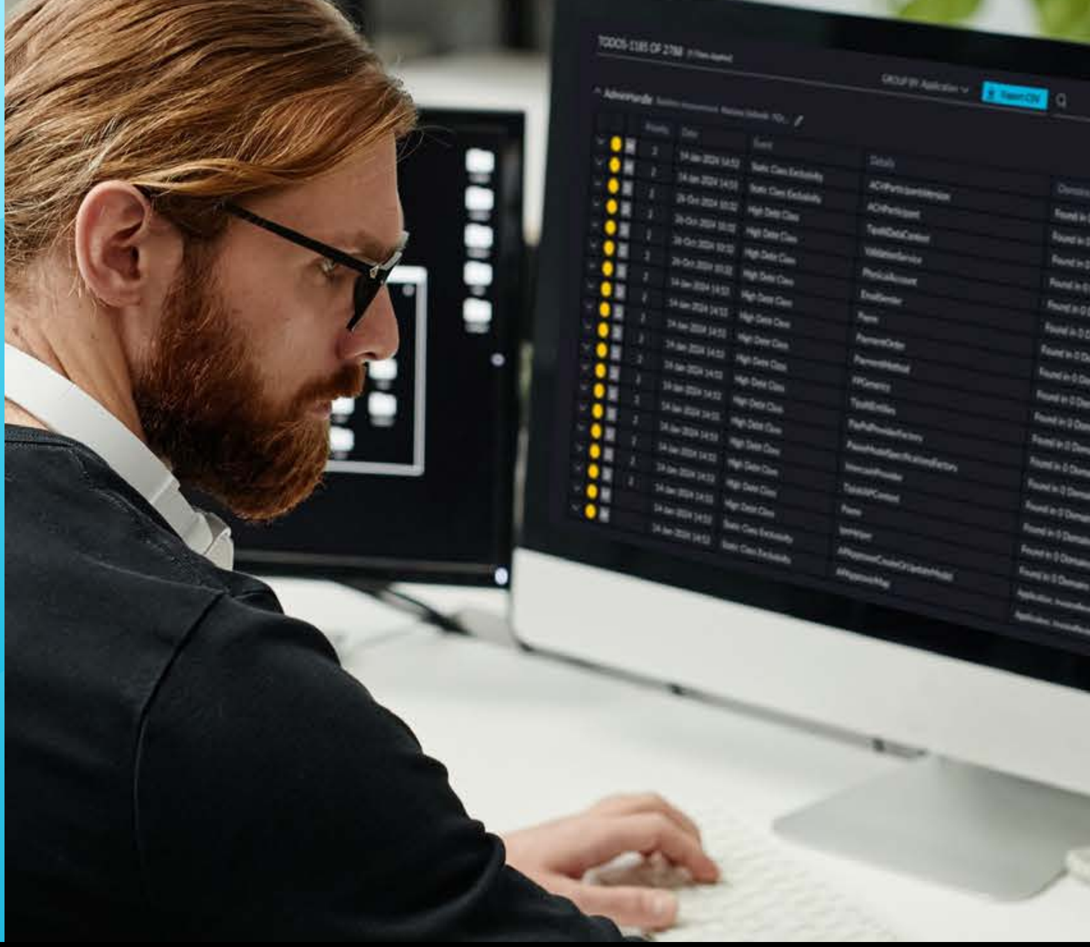
- **Dynamic class exclusivity changes**
  Monitoring exclusivity changes in dynamic classes as seen in Figure 5 above allows teams to identify and respond to runtime modifications that could introduce instability or security risks to the system. Understanding these changes enables the timely mitigation of potential issues and ensures the application's consistent and secure performance.
- **Static class exclusivity changes**
  Monitoring exclusivity changes in static classes is invaluable for maintaining a coherent architecture. It helps track unintentional dependencies and coupling introduced during the development phase, promotes cleaner code, and facilitates easier maintenance and debugging processes.
- **Resource exclusivity changes**
  Resources, such as database tables and files, play a pivotal role in the system's functionality. Monitoring their exclusivity ensures data integrity and system performance. When a typically exclusive resource starts appearing in multiple domains, it could indicate data redundancy, potential data conflicts, or increased load on the resource. Understanding and addressing these exclusivity resource changes is vital for preventing data-related issues and maintaining the system's overall health and efficiency.

IMPACT AND REMEDIATION

A class or resource appearing in multiple domains indicates reduced modularity and increased complexity. Architects should carefully examine this expanded distribution to discern whether it is intentional and advantageous or a harbinger of architectural pitfalls like heightened coupling or encapsulation loss.

Consider refactoring when a class or resource loses its exclusivity. Targeted refactoring restores the exclusive nature and singular responsibility of the resource or class, minimizing the unnecessary creation of dependencies and safeguarding the integrity and efficiency of the application's architecture.

Velocity

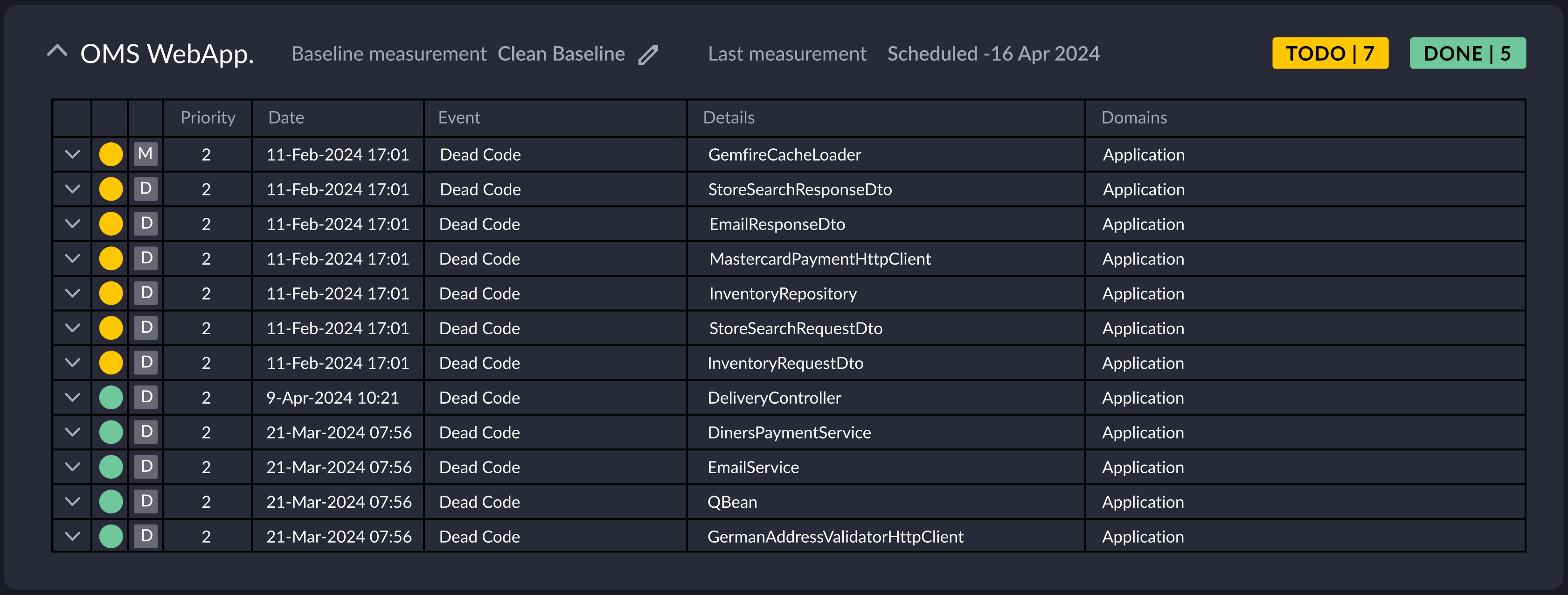


Architectural events that impact engineering velocity

DORA metrics by the DevOps Research and Assessment Group are standard measures application teams use for continuous improvement and quarterly and annual goals.

Complex software architecture increases test times, complicates adding new features, and increases deployment friction due to the plethora of cross-domain dependencies. Architectural events help engineering leaders get to the root of their velocity challenges.

Event: New dead code added/removed



OMS WebApp. Baseline measurement Clean Baseline Last measurement Scheduled -16 Apr 2024 TODO | 7 DONE | 5

			Priority	Date	Event	Details	Domains
⌄	🟡	M	2	11-Feb-2024 17:01	Dead Code	GemfireCacheLoader	Application
⌄	🟡	D	2	11-Feb-2024 17:01	Dead Code	StoreSearchResponseDto	Application
⌄	🟡	D	2	11-Feb-2024 17:01	Dead Code	EmailResponseDto	Application
⌄	🟡	D	2	11-Feb-2024 17:01	Dead Code	MastercardPaymentHttpClient	Application
⌄	🟡	D	2	11-Feb-2024 17:01	Dead Code	InventoryRepository	Application
⌄	🟡	D	2	11-Feb-2024 17:01	Dead Code	StoreSearchRequestDto	Application
⌄	🟡	D	2	11-Feb-2024 17:01	Dead Code	InventoryRequestDto	Application
⌄	🟢	D	2	9-Apr-2024 10:21	Dead Code	DeliveryController	Application
⌄	🟢	D	2	21-Mar-2024 07:56	Dead Code	DinersPaymentService	Application
⌄	🟢	D	2	21-Mar-2024 07:56	Dead Code	EmailService	Application
⌄	🟢	D	2	21-Mar-2024 07:56	Dead Code	QBean	Application
⌄	🟢	D	2	21-Mar-2024 07:56	Dead Code	GermanAddressValidatorHttpClient	Application

Figure 6: Pinpoint dead code

DEFINITION

Signals the detection of dead flows, obsolete, or dead code within an application.

DESCRIPTION

Dead code, as seen in Figure 6, comprises inactive or non-executable sections that serve no functional purpose within the application's source code. This includes unused functions/methods defined but not invoked and reachable code that is logically inaccessible during execution. Engineers can prune such code without jeopardizing the program's operational integrity.

Traditional static analysis tools often miss dead code that dynamic analysis detects using techniques such as:

- **Adaptive dead code recognition:** Dead code that dynamically emerges from changes in operational logic and user requirements.
- **Sensitive unreachable code detection:** This event accurately flags unreachable code, which is often mistaken as necessary due to complex dependencies or obfuscated execution paths.
- **Redundant code blocks detection within active functions:** this event finds obsolete code snippets within active functions.
- **In-depth analysis of runtime data:** This event discovers dead code that is not detectable during compile time or through static analysis, ensuring comprehensive detection.

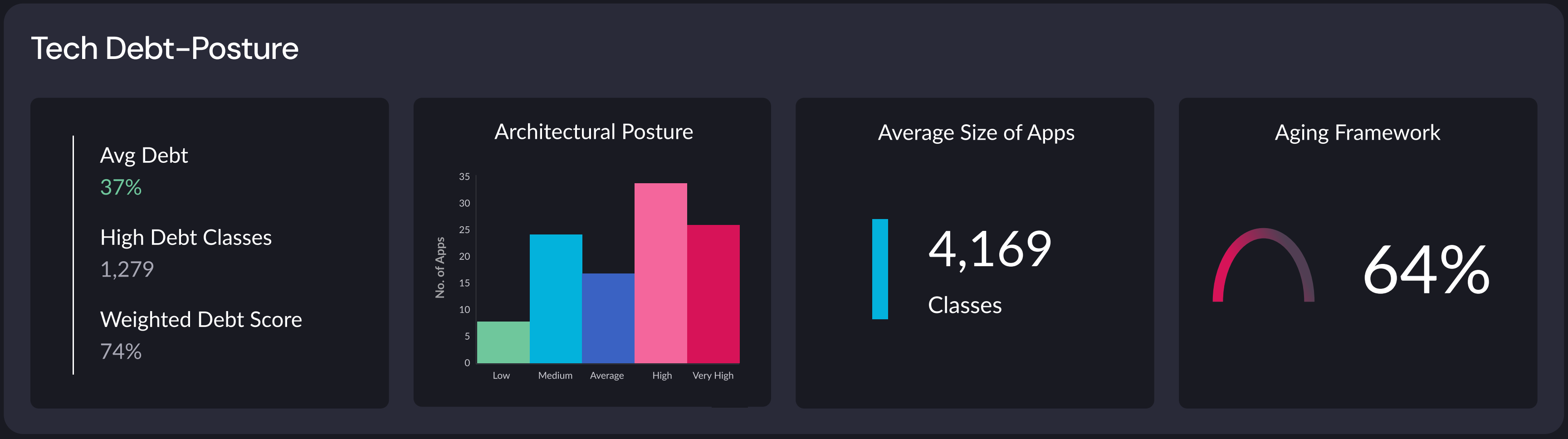
Any dynamic analysis tool used for dead code detection will require safety mechanisms to ensure that code not executed is dead code, not just code of an infrequent flow. Therefore, a combination of static and dynamic analysis is advised.

IMPACT AND REMEDIATION

Engineering teams should implement a strategy to periodically review and clean up the codebase to keep it clean, maintainable, and efficient.

While traditional methodologies include static code analysis tools, code coverage tools, and manual code review, it is important to include dynamic analysis and static analysis of the binaries, which offer real-time and deeply analytical insights, providing thorough and accurate dead code detection and removal.

Event: New high debt classes added, removed


Tech Debt-Posture

Avg Debt 37%

High Debt Classes 1,279

Weighted Debt Score 74%

Average Size of Apps: 4,169 Classes

Aging Framework: 64%

Figure 7: Detect high debt classes

DEFINITION

This event tracks the introduction and removal of high-debt classes. 'Debt' refers to architectural technical debt (ATD), which signifies the level of dependence between domains and architectural complexity created by the specific class.

DESCRIPTION

"High debt class" events do not solely rely on the usual suspects of coding flaws or design shortcuts. These events focus on dynamic analysis of parameters to evaluate and identify a class's architectural debt:

- **Dependencies:** The class should have many dependencies, indicating dependencies with other classes or components within the application.
- **Dependents:** Multiple dependents rely on this high debt class, making it a crucial node whose failure or malfunction might have cascading effects.
- **Size:** This indicates the absolute size of the class, not relative to the application. A high debt class of substantial size indicates its encompassing complexity and potential difficulty in maintenance and modification.

This approach delves deeper into the relationships within the application, offering a nuanced understanding of the debt a class introduces.

IMPACT AND REMEDIATION

To efficiently manage and mitigate technical debt, architects must prioritize refactoring classes identified as high debt under this refined criteria. Strategies include:

- Reducing the class's complexity and dependencies.
- Regular architectural reviews to prevent the introduction of new high debt classes.

Prioritize these classes in the development backlog for refactoring. By focusing on architecturally significant and entangled classes, this approach provides a targeted strategy to control and reduce the systemic risk and maintenance overhead introduced by high debt classes, thereby promoting a healthier, more sustainable codebase in the long term.

Event: Library circular dependencies

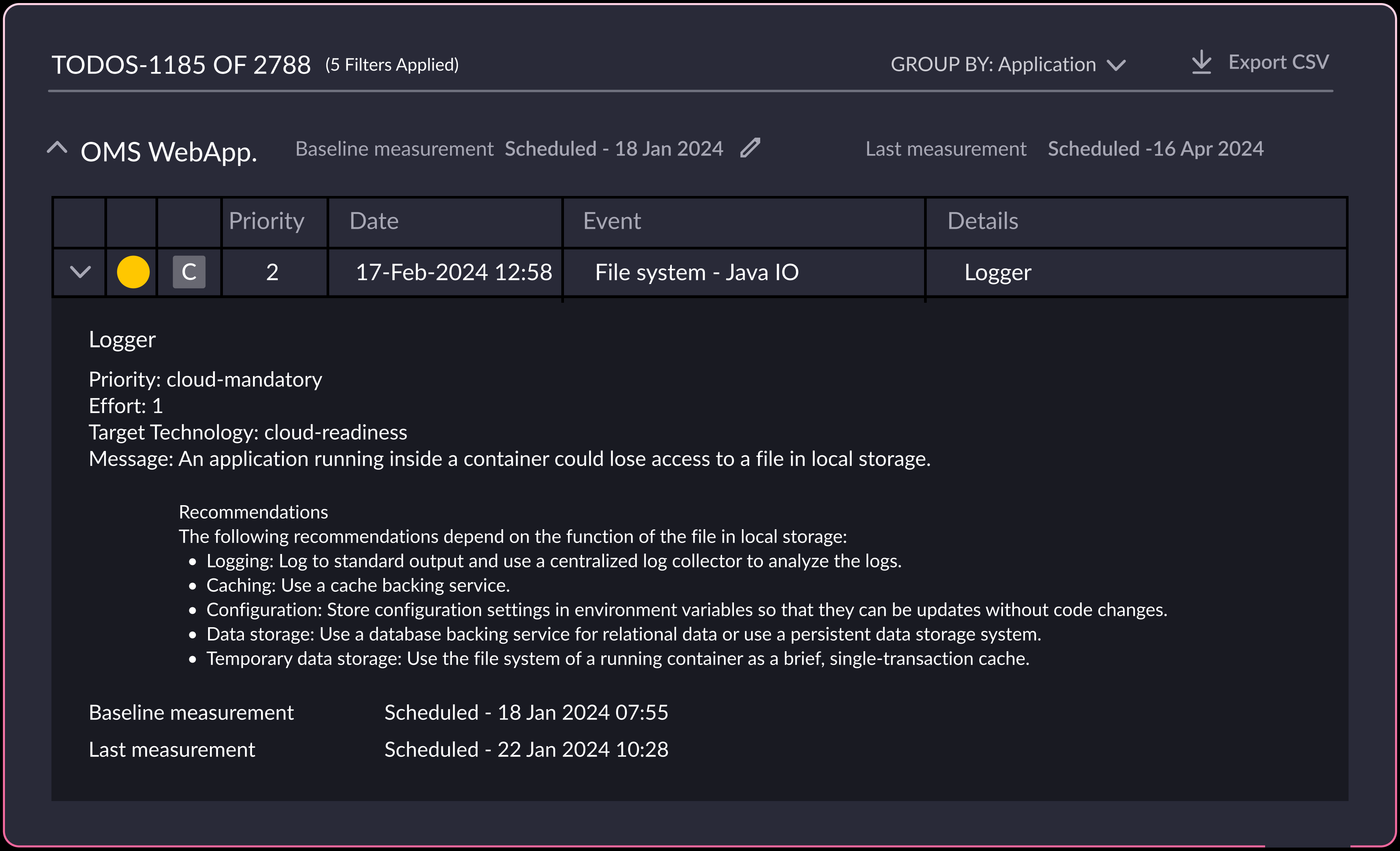
DEFINITION	This event detects library circular dependencies in internal JARs.
DESCRIPTION	• Circular dependencies can manifest as bi-directional arrows in the dependency graphs, indicating that two or more JARs are directly interdependent or dependent through a series of intermediaries. • Circular dependencies can lead to initialization problems, ambiguous designs, and complicate the module upgrade or replacement processes. • When a circular dependency is detected, architects should delve deep to understand the reasons behind it. Often, it's an unintentional result of incremental changes or rapid development without adequate architectural oversight.
IMPACT AND REMEDIATION	These dependencies can affect the startup order or cause runtime failures, particularly in dynamic module systems, impacting the system's resiliency. Assess whether this dependency introduces redundant functionality or manifests domain overlap. These insights can guide refactoring efforts to break the circular chain. Beyond just identification, it's beneficial to have a visual representation of these dependencies. An isolated graph, highlighted with an orange dashed line and focused solely on the circular chain allows teams to grasp the extent of the issue and strategize a resolution without being overwhelmed by the entirety of the dependency graph. Addressing these dependencies head-on ensures that the application's architecture remains modular, maintainable, and free from intricate entanglements that can stifle innovation.

Cloud Readiness

Architectural events to detect cloud readiness issues and streamline cloud migration

Cloud readiness architectural events highlight an application's preparedness for cloud environments. Given the increasing reliance on cloud platforms for scalability, flexibility, efficiency, and cost reduction, it is a critical aspect of modern software architecture. Applications should follow the fundamental principles of cloud-native design, such as microservices architecture, containerization, and DevOps practices.

These events start with the existing technologies and frameworks in the application and then provide actionable insights into the cloud readiness of a specific domain. Alerting engineers to the readiness of their application domains for cloud environments, as shown in Figure 8 on the next page, identifies areas that require changes or improvements to leverage cloud capabilities effectively.


TODOS-1185 OF 2788 (5 Filters Applied)

GROUP BY: Application

Export CSV

OMS WebApp. Baseline measurement Scheduled - 18 Jan 2024 Last measurement Scheduled -16 Apr 2024

			Priority	Date	Event	Details
	●	C	2	17-Feb-2024 12:58	File system - Java IO	Logger

Logger

Priority: cloud-mandatory
Effort: 1
Target Technology: cloud-readiness
Message: An application running inside a container could lose access to a file in local storage.

Recommendations
The following recommendations depend on the function of the file in local storage:

- Logging: Log to standard output and use a centralized log collector to analyze the logs.
- Caching: Use a cache backing service.
- Configuration: Store configuration settings in environment variables so that they can be updates without code changes.
- Data storage: Use a database backing service for relational data or use a persistent data storage system.
- Temporary data storage: Use the file system of a running container as a brief, single-transaction cache.

Baseline measurement	Scheduled - 18 Jan 2024 07:55
Last measurement	Scheduled - 22 Jan 2024 10:28

Figure 8: Cloud readiness event example

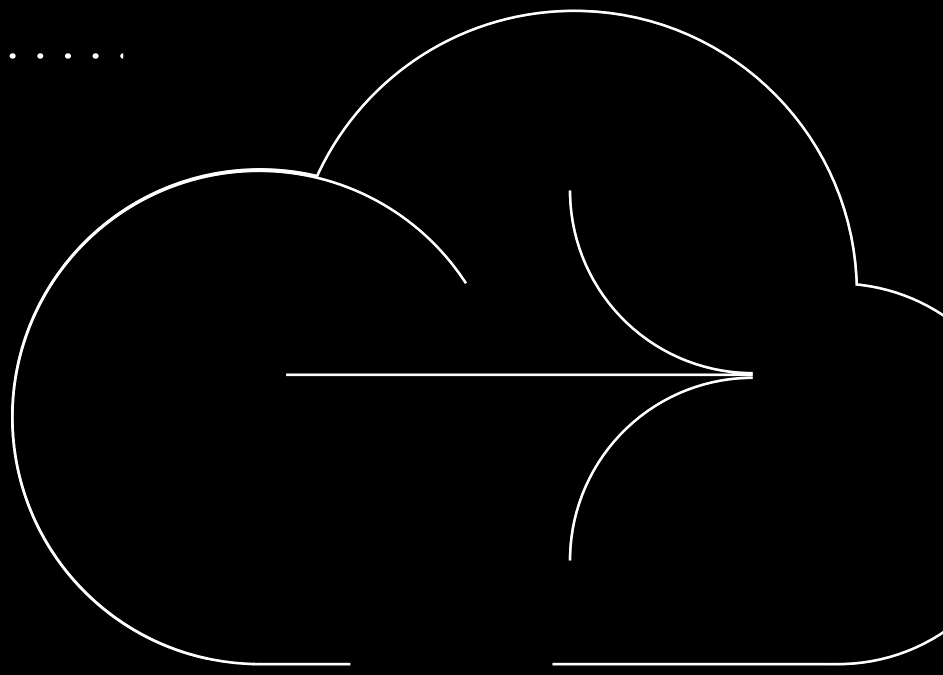
Examples of cloud readiness events based on Windup Cloud Readiness rulesets and applied dynamically based on the detected application configuration include:

- Migration to cloud friendly frameworks
- Direct dependency on an O/S platform
- Hard-coded properties
- Non-cloud native standards
- Per session context
- Embedded versus distributed caches
- Passwords in properties

Cloud compatibility events create tasks based on the source technology used by the application and the target technology the user intends to migrate to. JAVA target technologies for compatibility evaluation include:

Azure AKS	Azure App Service	Azure Container Apps	Azure Spring Apps
Camel	Drools	EAP	Fuse
Hibernate	Hibernate Search	Jakarta	jBPM
JEE	JWS	Linux	Open Liberty
OpenJDK 11	OpenJDK 17	Quarkus	RESTEasy

Cloud readiness is also available for .NET applications. Compatibility readiness is not a one-time event but a continuous process. Regular evaluations and updates ensure that applications align with the latest cloud technologies and methodologies.



Distributed Applications

Events for distributed applications

In distributed applications, monitoring architectural events is crucial due to the inherent complexity and interconnectedness of multiple services and components. Unlike monolithic applications, where the codebase is centralized, distributed systems involve intricate dependencies and communication patterns among various services.

Architectural events provide visibility into these interactions, enabling proactive technical debt management, maintainability, and overall system health. This visibility empowers informed decision-making regarding refactoring, optimization, or architectural adjustments, ensuring that the distributed application adheres to established principles, maintains modularity, and avoids uncontrolled complexity.

Events that impact resiliency

Event: New service added

This event indicates that a new service has been detected within the application, potentially signaling unplanned expansion or architectural drift. Uncontrolled growth of services can lead to increased complexity, reduced maintainability, and potential performance issues.

Event: Multi-hop trace

This event alerts users to multi-hop traces that exceed a certain threshold, indicating potentially complex service interactions that could impact performance. Excessive hops or overly complex service interactions can lead to latency, inefficiency, and increased potential for failures or bottlenecks.

Event: Introduce service dependency

While introducing dependencies can enable new functionality or improve integration, it should be done judiciously to avoid creating tight coupling or excessive inter-service communication. Users should carefully assess the proposed dependencies and ensure they align with the architectural principles and design goals.

Events that impact application scalability

Event: Resource exclusivity between services

This event informs users about changes in resource exclusivity among services, which could signal potential conflicts or inefficiencies. Resource sharing can lead to contention, performance issues, or data integrity problems if not managed properly.

Events that impact engineering velocity

Event: Service dependency added

This event notifies the users of newly added dependencies between services, which affect complexity and can potentially affect the application's performance. Service dependencies can introduce coupling, making it more difficult to maintain, scale, or refactor individual components.

Event: Circular traces

This event warns users about the detection of circular traces within the application, which can lead to performance issues or infinite loops. Circular dependencies can create complex and hard-to-maintain code, as well as introduce potential stability and scalability issues.

Event: Merge services

This event highlights opportunities to consolidate multiple services into a single service, potentially simplifying the application's structure. Excessive service granularity can lead to increased complexity, overhead, and potentially redundant functionality.

In MONOLITHIC APPLICATIONS, where the codebase is centralized, architectural events play a vital role in maintaining a well-structured and maintainable codebase. Unlike distributed systems, where the focus is on service interactions, architectural events in monolithic applications provide insights into the evolving complexity within the codebase itself.

By monitoring events, architects gain a comprehensive understanding of the application's evolving complexity. Events allow teams to address potential issues before they escalate, ensuring the codebase remains modular, cohesive, and free from unnecessary complexity.

vFunction architectural observability

Architectural observability is the key to detecting architectural events on a continuous basis. With the ability to analyze an application statically and dynamically, understand its architecture, observe drift, and find and fix architectural technical debt, architects and engineering teams can directly address application resiliency and scalability while improving engineering velocity and streamlining cloud readiness.

vFunction's AI-driven architectural observability platform manages architectural drift and detects architectural technical debt problems as they are introduced, so application teams can take proactive steps to specifically address the core problems affecting their highest priority KPIs.

The vFunction architectural observability platform is scalable, secure, and designed to routinely analyze and observe architecture in your regular development cycles to:

- Discover the real architecture of your applications
- Prevent architectural drift
- Manage and remediate technical debt
- Increase application resiliency
- Transform monoliths to microservices

Learn more about how vFunction can help you identify the architectural events impacting your business.

Backed by

citi VENTURES · Engineering Capital venture capital for engineers · Hewlett Packard Enterprise · PRESIDIO VENTURES · SHASTA · wipro · ZEEV VENTURES

Partnered with

aws · Hewlett Packard Enterprise · Microsoft Azure · ORACLE Cloud Infrastructure · wipro

Awards & Recognition

aws PARTNER Migration & Modernization Software · STRATUS 2023 AWARD · DEJ 20 TOP EMERGING VENDOR FOR MANAGING PERFORMANCE · CRN CLOUD 100 2024 · Intellyx DIGITAL INNOVATOR 2023

About vFunction

vFunction, the pioneer of AI-driven architectural observability, delivers a platform that increases application resiliency, scalability and engineering velocity by continuously identifying and recommending ways to reduce technical debt and complexity in applications. Global system integrators and top cloud providers partner with vFunction to assist leading companies like Intesa Sanpaolo and Trend Micro in discovering their architecture and transforming applications to innovate faster and change their business trajectory. vFunction is headquartered in Menlo Park, CA, with offices in Israel, London, and Austin, TX.

To learn more, visit www.vfunction.com.