Coralogix

Escaping the grip:

Best practices for avoiding and overcoming APM vendor lock-in

Introduction

Application Performance Monitoring (APM) is vital to an integrated observability strategy that includes logs, metrics, and security events. It allows organizations to monitor application performance in real time, quickly identifying and resolving issues to ensure optimal user experiences and system functionality. However, the full potential of APM can be undermined by vendor lock-in, where dependency on proprietary solutions can severely restrict flexibility and innovation. This eBook guides decision-makers in DevOps, SRE, and senior engineering roles on avoiding and escaping these constraints, enhancing their technological agility and operational efficiency. We look at code examples and show how to avoid vendor lock-in by practicing clean code design and making sound architectural decisions, such as avoiding tightly coupled code, modularizing infrastructure, and using open-source solutions. We also guide you through a strategy to eliminate vendor lock-in if your system currently experiences it.

Understanding vendor lock-in

Definition and overview

Vendor lock-in occurs when a customer becomes so dependent on a vendor's products and services that switching to another provider entails substantial costs and inconveniences. This is particularly problematic in the APM space, where proprietary agents or data formats are deeply integrated into systems, complicating transitions to other solutions.

Drawbacks for organizations

Costs

The inability to switch vendors can lead to higher operational costs, as organizations might pay premium prices for services that could be obtained less expensively elsewhere.There is also increased costs due to lack of competition and price negotiation leverage.

Dependency and barrier to innovation

Dependence on a single provider for updates and system integrations can slow the adoption of newer, better technologies, hindering innovation. This dependence also limits strategic and technical flexibility, forcing companies to align their IT infrastructure decisions with a specific vendor's capabilities or product offerings.

The APM lock-in dilemma

Application Performance Monitoring allows companies to ensure their apps perform according to standards. They also enable engineers to track where slowdowns occur when performance is lacking. Since APM is a critical piece of cloud-based applications, companies must decide how to best implement it by choosing a balance between the speed of implementation, cost efficiency, and the maintainability and extensibility of the APM features.

When you purchase a new APM platform, it may come with amazing features. However, many of these tools are not vendor agnostic, meaning you could not use open-source tools when it suits you because your platform will not allow you to plug them in. This is similar to purchasing an iPhone. The phone comes with many amazing features, but you will need to use AirPods, an Apple Watch and a MacBook if you want to easily connect to accessories. Here we will discuss how vendors will often lock you into their system (and accessories) and how to avoid lock-in with coding and architecture design strategies.

Common causes and examples

Proprietary APIs, specific data formats, and custom instrumentation code are common lock-in strategies vendors use. For instance, a proprietary APM tool might use unique data tagging that doesn't translate to other systems, complicating data aggregation and analysis efforts when using other tools. Vendors may also force customers to use proprietary APIs or SDKs, requiring significant integrations to use their services. These tools often tie the developer to the vendor's ecosystem, creating a form of lock-in.

Strategies to avoid vendor lock-in

Code design

Good code design is crucial for avoiding vendor lock-in. It ensures that your applications remain flexible and adaptable to changes in technology or business needs without being overly dependent on any specific vendor's tools or platforms. Having a good code design can help to remove a vendor more easily, even if they use proprietary APIs or SDKs. However, coding alone will not suffice to get around proprietary data models or tagging.

Let's consider an example where a vendor has a proprietary SDK:

Bad code: Tightly coupled code causing lock in

In this example, any change in one place requires changes in many other places. This link drives a code smell called "shotgun surgery." In the case of vendor lock-in, this particular pattern is extremely dangerous. Now, consider the following example:

PaymentService.java

```
import vendorx.apm.APM;

public class PaymentService {
    private APM apm;

    public PaymentService() {
        this.apm = new APM();
    }

    public void processPayment(String accountId, double amount) {
        apm.startTransaction("processPayment");
        try {

            apm.endTransaction("processPayment", true);
        } catch (Exception e) {
            apm.endTransaction("processPayment", false);
            throw e;
        }
    }
}
```

OrderService.java

```java
import vendorx.apm.APM;

public class OrderService {
    private APM apm;

    public OrderService() {
        this.apm = new APM();
    }

    public void createOrder(String accountId, double amount) {
        apm.startTransaction("createOrder");
        try {
            // Perform some order creation
            apm.endTransaction("createOrder", true);
        } catch (Exception e) {
            apm.endTransaction("createOrder", false);
            throw e;
        }
    }
}
```

SubscriptionService.java

```java
import vendorx.apm.APM;

public class SubscriptionService {
    private APM apm;

    public SubscriptionService() {
        this.apm = new APM();
    }

    public void renewSubscription(String accountId, double amount) {
        apm.startTransaction("renewSubscription");
        try {
            // Perform some subscription renewal
            apm.endTransaction("renewSubscription", true);
        } catch (Exception e) {
            apm.endTransaction("renewSubscription", false);
            throw e;
        }
    }
}
```

In the above example, each application depends on VendorX's APM class. Changes or upgrades to that APM class will also require updates to the payment, order, and subscription classes. Changing to a different APM tool would also require updates to every class in which the tool is embedded, leading to significant engineering costs.

Good code: Same code decoupled via interfaces

If one class's inner workings change, the other classes do not care because their interface has been *abstracted*. This is the fundamental principle of good code design when tackling vendor lock-in. By adhering to principles of modularity, using open standards, and abstracting dependencies away from proprietary APIs and libraries, developers can build systems that are easier to maintain and upgrade. This approach facilitates easier integration with different tools and services and empowers organizations to switch vendors with minimal disruption.

Ultimately, well-designed code enhances the organization's ability to innovate and scale and respond to market changes more rapidly, maintaining a competitive edge while controlling costs and reducing risks associated with vendor dependency.

Java

APM Interface

```
public interface APMObserver {
    void startTransaction(String name);
    void endTransaction(String name, boolean success);
}
```

Vendor X APM Observer Implementation

```
import vendorx.apm.APM;

public class VendorXAPMObserver implements APMObserver {
    private APM apm;

    public VendorXAPMObserver() {
        this.apm = new APM();
    }

    @Override
    public void startTransaction(String name) {
        apm.startTransaction(name);
    }

    @Override
    public void endTransaction(String name, boolean success) {
        apm.endTransaction(name, success);
    }
}
```

PaymentService.java

```
public class PaymentService {
    private APMObserver apmObserver;

    public PaymentService(APMObserver apmObserver) {
        this.apmObserver = apmObserver;
    }

    public void processPayment(String accountId, double amount) {
        apmObserver.startTransaction("processPayment");
        try {
            // Perform some payment processing
            apmObserver.endTransaction("processPayment", true);
        } catch (Exception e) {
            apmObserver.endTransaction("processPayment", false);
            throw e;
        }
    }
}
```

OrderService.java

```
public class OrderService {
    private APMObserver apmObserver;

    public OrderService(APMObserver apmObserver) {
        this.apmObserver = apmObserver;
    }

    public void createOrder(String accountId, double amount) {
        apmObserver.startTransaction("createOrder");
        try {
            // Perform some order creation
            apmObserver.endTransaction("createOrder", true);
        } catch (Exception e) {
            apmObserver.endTransaction("createOrder", false);
            throw e;
        }
    }
}
```

SubscriptionService.java

```
public class SubscriptionService {
    private APMObserver apmObserver;

    public SubscriptionService(APMObserver apmObserver) {
        this.apmObserver = apmObserver;
    }

    public void renewSubscription(String accountId, double amount) {
        apmObserver.startTransaction("renewSubscription");
        try {
            // Perform some subscription renewal
            apmObserver.endTransaction("renewSubscription", true);
        } catch (Exception e) {
            apmObserver.endTransaction("renewSubscription", false);
            throw e;
        }
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        APMObserver apmObserver = new VendorXAPMObserver();

        PaymentService paymentService = new PaymentService(apmObserver);
        OrderService orderService = new OrderService(apmObserver);
        SubscriptionService subscriptionService = new SubscriptionService(apmObserver);

        paymentService.processPayment("account1", 100.0);
        orderService.createOrder("account2", 200.0);
        subscriptionService.renewSubscription("account3", 300.0);
    }
}
```

Node.JS

APMObserver.js

```
class APMObserver {
    startTransaction(name) {
        throw new Error('Method not implemented.');
    }

    endTransaction(name, success) {
        throw new Error('Method not implemented.');
    }
}

module.exports = APMObserver;
```

VendorY_APM_Observer.js

```
const VendorYAPM = require('vendory-apm');
const APMObserver = require('./APMObserver');

class VendorYAPMObserver extends APMObserver {
    constructor() {
        super();
        this.apm = new VendorYAPM();
    }

    startTransaction(name) {
        this.apm.startTransaction(name);
    }

    endTransaction(name, success) {
        this.apm.endTransaction(name, success);
    }
}

module.exports = VendorYAPMObserver;
```

PaymentService.js

```
class PaymentService {
    constructor(apmObserver) {
        this.apmObserver = apmObserver;
    }

    processPayment(accountId, amount) {
        this.apmObserver.startTransaction('processPayment');
        try {
            // Perform some payment processing
            this.apmObserver.endTransaction('processPayment', true);
        } catch (error) {
            this.apmObserver.endTransaction('processPayment', false);
            throw error;
        }
    }
}

module.exports = PaymentService;
```

OrderService

```
class OrderService {
    constructor(apmObserver) {
        this.apmObserver = apmObserver;
    }

    createOrder(accountId, amount) {
        this.apmObserver.startTransaction('createOrder');
        try {
            // Perform some order creation
            this.apmObserver.endTransaction('createOrder', true);
        } catch (error) {
            this.apmObserver.endTransaction('createOrder', false);
            throw error;
        }
    }
}

module.exports = OrderService;
```

App.js

```
const VendorYAPMObserver = require('./VendorYAPMObserver');
const PaymentService = require('./PaymentService');
const OrderService = require('./OrderService');

const apmObserver = new VendorYAPMObserver();

const paymentService = new PaymentService(apmObserver);
const orderService = new OrderService(apmObserver);

paymentService.processPayment('account1', 100.0);
orderService.createOrder('account2', 200.0);
```

Architectural decisions

When integrating APM and observability tools, ensuring that proprietary tools within your environment are contained and their impact is minimized is vital. Here's how you can achieve this:

Containerization and abstraction of proprietary tools

Use containerization technologies like Docker and orchestration platforms like Kubernetes to encapsulate proprietary tools. This makes it easier to replace them if necessary and limits their interaction with other parts of your infrastructure. The applications will also be portable across different environments without significant refactoring.

Use of labels and annotations

Minimize the use of vendor-specific labels and annotations on resources. Instead, opt for generic labels that describe the resource's role or purpose, not its vendor-specific configuration or role. This approach reduces dependency on any vendor's tooling and facilitates easier migration to other tools if needed.

Modular infrastructure design

Design your infrastructure to be modular where possible using interfaces and abstraction layers like OpenTelemetry. This approach allows you to swap out components with minimal disruption to the overall system.

Governance and documentation

Establish clear governance around proprietary versus open solutions, including detailed documentation of where and why proprietary solutions are used. This governance should also outline the process for evaluating and potentially replacing these solutions with more open alternatives.

Breaking free: Escaping existing lock-in

Assessing and refactoring code dependency

Review and identify any proprietary SDKs or APIs integrated into your codebase. Wherever possible, replace these with open standards. For example, a proprietary logging SDK can be replaced with SLF4J in Java, an open-source, simple facade for various logging frameworks. SLF4J allows the end user to plug in the desired logging framework at deployment time.

Example: Before & after – Datadog vs SLF4J

The first example shows a class instrumented for logs using the Datadog Java library, while the second example uses SLF4J.

PaymentService.java (Datadog APM)

```
import datadog.trace.api.Trace;
import datadog.trace.api.DDTags;
import datadog.trace.api.DDSpan;

public class PaymentService {

    @Trace(operationName = “processPayment”)
    public void processPayment(String accountId, double amount) {
        DDSpan span = datadog.trace.api.GlobalTracer.get()
                          .buildSpan(“processPayment”)
                          .withTag(DDTags.RESOURCE_NAME, “payment”)
                          .start();
        try {
            // Perform some payment processing
            span.setTag(“amount”, amount);
        } catch (Exception e) {
            span.setTag(DDTags.ERROR, true);
            span.setTag(“error.msg”, e.getMessage());
            throw e;
        } finally {
            span.finish();
        }
    }
}
```

PaymentService.java (SLF4J)

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class PaymentService {

    private static final Logger logger = LoggerFactory.getLogger(PaymentService.class);

    public void processPayment(String accountId, double amount) {
        logger.info("Starting transaction: processPayment");
        try {
            // Perform some payment processing
            logger.info("Processed payment for account: {}, amount: {}", accountId,
amount);
        } catch (Exception e) {
            logger.error("Error processing payment for account: {}, amount: {}",
accountId, amount, e);
            throw e;
        } finally {
            logger.info("Ending transaction: processPayment");
        }
    }
}
```

Migrating from proprietary solutions

Develop a phased migration plan that allows you to gradually replace proprietary components with open-standard-based alternatives without disrupting ongoing operations. Here we show how you can move from Datadog to SLF4J. Starting with a class that uses Datadog's implementation only, and with the Datadog logic present in each class.

PaymentService.java

```
import datadog.trace.api.Trace;
import datadog.trace.api.DDTags;
import datadog.trace.api.DDSpan;

public class PaymentService {

    @Trace(operationName = "processPayment")
    public void processPayment(String accountId, double amount) {
        DDSpan span = datadog.trace.api.GlobalTracer.get()
                          .buildSpan("processPayment")
                          .withTag(DDTags.RESOURCE_NAME, "payment")
                          .start();
        try {
            // Perform some payment processing
            span.setTag("amount", amount);
        } catch (Exception e) {
            span.setTag(DDTags.ERROR, true);
            span.setTag("error.msg", e.getMessage());
            throw e;
        } finally {
            span.finish();
        }
    }
}
```

1. Add SLF4J logging alongside Datadog. By adding this logic alongside Datadog, you can verify functionality with little risk to the existing system.

PaymentService.java

```
import datadog.trace.api.Trace;
import datadog.trace.api.DDTags;
import datadog.trace.api.DDSpan;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class PaymentService {

    private static final Logger logger = LoggerFactory.getLogger(PaymentService.class);

    @Trace(operationName = “processPayment”)
    public void processPayment(String accountId, double amount) {
        DDSpan span = datadog.trace.api.GlobalTracer.get()
                          .buildSpan(“processPayment”)
                          .withTag(DDTags.RESOURCE_NAME, “payment”)
                          .start();
        try {
            logger.info(“Starting transaction: processPayment for account: {}”,
accountId);
            // Perform some payment processing
            span.setTag(“amount”, amount);
            logger.info(“Processed payment for account: {}, amount: {}”, accountId,
amount);
        } catch (Exception e) {
            span.setTag(DDTags.ERROR, true);
            span.setTag(“error.msg”, e.getMessage());
            logger.error(“Error processing payment for account: {}, amount: {}”,
accountId, amount, e);
            throw e;
        } finally {
            span.finish();
            logger.info(“Ending transaction: processPayment for account: {}”, accountId);
        }
    }
}
```

2. Extract all APM code to a separate method-handling APM instrumentation. This will make it easier to remove code later.

PaymentService.java

```
import datadog.trace.api.Trace;
import datadog.trace.api.DDTags;
import datadog.trace.api.DDSpan;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class PaymentService {

    private static final Logger logger = LoggerFactory.getLogger(PaymentService.class);

    @Trace(operationName = “processPayment”)
    public void processPayment(String accountId, double amount) {
        DDSpan span = startTransaction(“processPayment”, “payment”);
        try {
            logger.info(“Starting transaction: processPayment for account: {}”,
accountId);
            // Perform some payment processing
            span.setTag(“amount”, amount);
            logger.info(“Processed payment for account: {}, amount: {}”, accountId,
amount);
        } catch (Exception e) {
            span.setTag(DDTags.ERROR, true);
            span.setTag(“error.msg”, e.getMessage());
            logger.error(“Error processing payment for account: {}, amount: {}”,
accountId, amount, e);
            throw e;
        } finally {
            finishTransaction(span);
            logger.info(“Ending transaction: processPayment for account: {}”, accountId);
        }
    }

    private DDSpan startTransaction(String operationName, String resourceName) {
        return datadog.trace.api.GlobalTracer.get()
                        .buildSpan(operationName)
                        .withTag(DDTags.RESOURCE_NAME, resourceName)
                        .start();
    }

    private void finishTransaction(DDSpan span) {
        span.finish();
    }
}
```

3. Define an interface for APM operations and create an implementation for Datadog. This will allow you to switch implementations easily.

APMObserver.java

```
public interface APMObserver {
    void startTransaction(String operationName, String resourceName);
    void endTransaction(String operationName, boolean success, String errorMessage);
}
```

DatadogObserver.java

```
import datadog.trace.api.DDTags;
import datadog.trace.api.DDSpan;
import datadog.trace.api.GlobalTracer;

public class DataDogAPMObserver implements APMObserver {

    private DDSpan span;

    @Override
    public void startTransaction(String operationName, String resourceName) {
        span = GlobalTracer.get()
                .buildSpan(operationName)
                .withTag(DDTags.RESOURCE_NAME, resourceName)
                .start();
    }

    @Override
    public void endTransaction(String operationName, boolean success, String
errorMessage) {
        if (!success) {
            span.setTag(DDTags.ERROR, true);
            span.setTag(“error.msg”, errorMessage);
        }
        span.finish();
    }
}
```

PaymentService.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class PaymentService {

    private static final Logger logger = LoggerFactory.getLogger(PaymentService.class);
    private final APMObserver apmObserver;

    public PaymentService(APMObserver apmObserver) {
        this.apmObserver = apmObserver;
    }

    public void processPayment(String accountId, double amount) {
        apmObserver.startTransaction("processPayment", "payment");
        try {
            logger.info("Starting transaction: processPayment for account: {}", accountId);
            // Perform some payment processing
            logger.info("Processed payment for account: {}, amount: {}", accountId,
amount);
            apmObserver.endTransaction("processPayment", true, null);
        } catch (Exception e) {
            logger.error("Error processing payment for account: {}, amount: {}", accountId,
amount, e);
            apmObserver.endTransaction("processPayment", false, e.getMessage());
            throw e;
        } finally {
            logger.info("Ending transaction: processPayment for account: {}", accountId);
        }
    }
}
```

4. Replace Datadog with SLF4J only by switching the observer implementation and without changing the core application logic.

SLF4JOberver.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SLF4JAPMObserver implements APMObserver {

    private static final Logger logger = LoggerFactory.getLogger(SLF4JAPMObserver.class);

    @Override
    public void startTransaction(String operationName, String resourceName) {
        logger.info("Starting transaction: {}, resource: {}", operationName,
resourceName);
    }

    @Override
    public void endTransaction(String operationName, boolean success, String
errorMessage) {
        if (!success) {
            logger.error("Error in transaction: {}, error: {}", operationName,
errorMessage);
        }
        logger.info("Ending transaction: {}", operationName);
    }
}
```

PaymentService.java

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class PaymentService {

    private static final Logger logger = LoggerFactory.getLogger(PaymentService.class);
    private final APMObserver apmObserver;

    public PaymentService(APMObserver apmObserver) {
        this.apmObserver = apmObserver;
    }

    public void processPayment(String accountId, double amount) {
        apmObserver.startTransaction("processPayment", "payment");
        try {
            logger.info("Starting transaction: processPayment for account: {}",
accountId);
            // Perform some payment processing
            logger.info("Processed payment for account: {}, amount: {}", accountId,
amount);
            apmObserver.endTransaction("processPayment", true, null);
        } catch (Exception e) {
            logger.error("Error processing payment for account: {}, amount: {}",
accountId, amount, e);
            apmObserver.endTransaction("processPayment", false, e.getMessage());
            throw e;
        } finally {
            logger.info("Ending transaction: processPayment for account: {}", accountId);
        }
    }
}
```

Choosing the right observability platform

Choosing the right observability provider is a crucial decision that can significantly impact your organization's monitoring capabilities' flexibility, scalability, and effectiveness. The ideal provider should meet your needs and accommodate future growth and technological changes. Selecting a platform that supports open standards like OpenTelemetry, offers extensive data portability, and allows integration with a broad range of tools and systems is essential.

Using open standards ensures that your observability infrastructure remains adaptable and can evolve without being hindered by vendor lock-in. By prioritizing these features, organizations can maintain control over their monitoring environments and make strategic decisions that align with long-term business goals.

Support for OpenTelemetry

Ensures compatibility and future-proofing of instrumentation. A vendor that does not support OpenTelemetry makes them the exception, not the rule.

Data portability

Ensures that data can be exported in open, non-proprietary formats like CSV or Parquet for use within other systems.

Ability to affordably scale

The constraints on scaling are more than just cost. Some platforms were designed with data volumes from a decade ago in mind. Any platform you choose needs to be designed for modern scale, and that means the ability to affordably ingest and analyze vast amounts of traces, logs and metrics, visualize large amounts of data in a digestible way, and maintain query performance, even when there are terabytes to scan.

Staying free with Coralogix

While many observability vendors create lock-in with proprietary agents, storage formats and the like, Coralogix gives customers a full SaaS experience for APM while being completely open source friendly. With OpenTelemetry being used for shipping data, Parquet for our storage format and customers actually storing (and querying) all their data in their own S3 bucket, you are completely free to migrate your data anywhere you wish. In addition to this, Coralogix's in-stream analysis eliminates reliance on indexing and hot storage so you can affordably analyze all your data, logs, metrics, and traces for a fraction of what other vendors are charging.

In conclusion, while APM is essential for modern IT operations, being tied to a single vendor can severely limit an organization's agility and cost-efficiency. Organizations can mitigate the risks of vendor lock-in and maintain their competitive edge by adopting strategies that promote open standards and flexibility. This eBook has laid out foundational strategies and practical tips to help decision-makers navigate the complex landscape of APM tools and vendors, empowering them to make informed decisions that align with their long-term technological and business goals.

About Coralogix

We are rebuilding the path to observability using a real-time streaming analytics pipeline that provides monitoring, visualization, and alerting capabilities without the burden of indexing.

By enabling users to define different data pipelines per use case, we provide deep insights for less than half the cost.

In short, we are streaming the future of data.

Built for tomorrow's data scale

2K+
Global customers

10K+
DevOps and engineering Users

500K+
Applications monitored

3M+
Events processed per second

Modern architecture. Disruptive value.
Observability and Security that scale with you.

Get a Demo

Coralogix