A Developer's Guide to the Amazon Disco Toolkit: Mastering Java Agent-Based Instrumentation for Distributed Systems

Part I: The Foundations of Java Instrumentation

A comprehensive understanding of the Amazon Disco Toolkit necessitates a foundational knowledge of the environment in which it operates: the Java Virtual Machine (JVM) and its powerful, yet intricate, instrumentation capabilities. This section delves into the core problems that Java Agents are designed to solve, the technologies they leverage, and the fundamental challenges inherent in modifying a running application's behavior.

Chapter 1: The World Before Agents: The Need for Runtime Insight

The journey of a Java application from human-readable source code to an executing process involves several layers of abstraction. While these layers provide portability and power, they also create challenges for observability.

The Opaque JVM

When a developer writes Java code, the javac compiler transforms it not into native machine code, but into an intermediate representation known as bytecode, stored in .class files.[1] This bytecode is platform-independent, a key tenet of Java's "Write Once, Run Anywhere" philosophy.[1] The Java Virtual Machine (JVM), a sophisticated software-based engine, then interprets and executes this bytecode.[1]

This process, while efficient, renders the running application a "black box." The internal state, method execution flows, and performance characteristics are not

readily visible. To gain insight, developers need mechanisms to peer inside the JVM's execution engine without halting the entire system.

The Problem of Cross-Cutting Concerns

In software engineering, certain functionalities are essential yet not part of the core business logic. These include logging, performance monitoring, security auditing, and transaction management. Such functionalities are known as cross-cutting concerns because they "cut across" multiple modules and layers of an application.[4]

Manually inserting code for these concerns into every relevant method is not only tedious and error-prone but also pollutes the primary business logic. It makes the code harder to read, maintain, and evolve. For example, adding performance timing to a method would require boilerplate code at the beginning and end of every method to be measured, obscuring its actual purpose. This approach is fundamentally unscalable and unsustainable in complex systems.

Instrumentation as the Solution

The solution to this dilemma is instrumentation: the technique of adding bytecode to methods to gather data or alter behavior without modifying the original source code.[6] It allows for the dynamic injection of the logic for these cross-cutting concerns at runtime. This is precisely the problem that Java Agents are designed to solve. They act as powerful tools that can intercept the class loading process and modify bytecode on the fly, enabling a wide range of applications from application performance monitoring (APM) to real-time security analysis.[9]

Chapter 2: Anatomy of a Java Agent

The ability to perform runtime instrumentation is formally exposed through the Java Instrumentation API, a standard part of the Java platform since J2SE 5.0.[6] This API

provides the core components and contracts for building agents.

Introducing java.lang.instrument

The java.lang.instrument package is the cornerstone of Java's instrumentation capabilities. It provides the necessary services that allow an agent to modify the bytecode of methods in a program running on the JVM.[10]

The Instrumentation Interface

The Instrumentation interface is the primary gateway through which an agent interacts with the JVM's instrumentation facilities. An instance of this class is passed by the JVM to the agent's entry point method, providing a powerful set of capabilities.[7] Key methods include:

- **addTransformer(ClassFileTransformer transformer, boolean canRetransform)**: This is the most fundamental method. It registers a ClassFileTransformer with the JVM. Once registered, the transformer's transform method will be invoked for every class that is loaded, redefined, or retransformed, allowing the agent to inspect and modify its bytecode.[7]
- **redefineClasses(ClassDefinition... definitions)**: This method allows an agent to completely replace the definition of one or more already-loaded classes. It takes a ClassDefinition object, which pairs a Class object with a byte array containing the new bytecode.[11] This is a powerful but blunt instrument, often used for "fix-and-continue" style debugging where a class is recompiled and hot-swapped into the running JVM. Its primary limitation is that it does not facilitate cooperation between multiple agents trying to modify the same class; the last agent to call
  redefineClasses wins, overwriting any previous modifications.
- **retransformClasses(Class<?>... classes)**: This method was introduced to address the limitations of redefineClasses in a multi-agent environment. Instead of providing new bytecode directly, retransformClasses triggers a re-application of all registered transformers on the specified classes. Crucially, the transformation process starts from the *original* class file bytes, not the currently

active ones.[13] This allows each registered transformer to apply its changes in a chain, enabling multiple agents to instrument the same class without conflict. This cooperative model is essential for frameworks like Disco that rely on a pluggable architecture.[12]
- **getAllLoadedClasses()**: This utility method returns an array of all classes currently loaded by the JVM, which can be useful for introspection or for identifying classes to retransform.[7]

The ClassFileTransformer Interface

This interface is the workhorse of instrumentation. Any class that implements ClassFileTransformer can be registered with the Instrumentation instance to intercept class loading.[15] It defines a single primary method:

byte transform(ClassLoader loader, String className, Class<?> classBeingRedefined, ProtectionDomain protectionDomain, byte classfileBuffer) throws IllegalClassFormatException

Each parameter provides critical context:

- loader: The classloader that is loading the class. This can be null if the class is being loaded by the bootstrap classloader.[17]
- className: The fully qualified name of the class in its internal format (e.g., java/util/List).[17]
- classBeingRedefined: The Class object if the transformation is triggered by a redefinition or retransformation; otherwise, it is null for an initial class load.[17]
- protectionDomain: The security protection domain of the class.
- classfileBuffer: The raw bytecode of the class as a byte array.

The implementation of this method must return a new byte array containing the transformed bytecode. If no transformation is performed, it must return null. It is critical to note that the input classfileBuffer must not be modified in place.[17] Furthermore, if the

transform method throws an exception, the JVM will typically ignore it and proceed with the original, untransformed bytecode, which can make debugging agents challenging.[18]

The Agent's Identity: MANIFEST.MF

A Java Agent is not just a collection of classes; it is a specially packaged JAR file with a specific MANIFEST.MF file. This manifest acts as an identity card, telling the JVM how to load and initialize the agent.[19] Several attributes are critical for an agent's operation.

Attribute	Description	Required?	Relevant Method(s)
Premain-Class	Specifies the agent class to load at JVM startup. The class must contain a premain method. [21]	Yes (for static load)	premain
Agent-Class	Specifies the agent class to load when attaching to a running JVM. The class must contain an agentmain method. [21]	Yes (for dynamic load)	agentmain
Can-Redefine-Classes	A boolean (true/false) indicating if the agent requires the ability to redefine classes using Instrumentation.redefineClasses. [22]	No (default false)	redefineClasses
Can-Retransform-Classes	A boolean (true/false) indicating if the agent requires the ability to retransform classes using Instrumentation.retransformClasses. [22]	No (default false)	retransformClasses
Boot-Class-Path	A space-separated list of JARs to be added to the	No	N/A

	bootstrap classloader's search path. Crucial for instrumenting core JDK classes. [23]		

Misconfiguration of these manifest attributes is a common source of errors, such as the JVM aborting with a "Failed to load Premain-Class manifest attribute" message.[21]

Chapter 3: Agent Loading Mechanisms: Static vs. Dynamic

The Java Instrumentation API provides two distinct mechanisms for loading an agent into a JVM, each serving different use cases and operational needs.[6]

Static Loading (premain)

Static loading is the most straightforward and common method for deploying a Java Agent. The agent is loaded as part of the JVM's initialization sequence, before the application's main method is ever invoked.[6]

This is accomplished by adding the -javaagent command-line flag when starting the application.[19] The syntax is as follows:

java -javaagent:/path/to/my-agent.jar=someOptions -jar /path/to/my-app.jar

When the JVM sees this flag, it locates the agent class specified by the Premain-Class attribute in the agent JAR's manifest. It then invokes one of two premain methods on that class [6]:

1. public static void premain(String agentArgs, Instrumentation inst)
2. public static void premain(String agentArgs)

The JVM will first attempt to call the signature with the Instrumentation parameter. The agentArgs string contains whatever was passed after the equals sign in the -javaagent flag (e.g., "someOptions" in the example above), allowing for runtime

configuration of the agent.[10] Because

premain runs before the application's main method, it guarantees that the agent's transformers are in place to instrument every class as it is loaded.[21]

Dynamic Loading (agentmain)

Dynamic loading is a more advanced technique that allows an agent to be attached to a JVM that is already running.[6] This capability is immensely powerful for live troubleshooting, profiling, and managing production systems without requiring a service restart.[22]

The entry point for a dynamically attached agent is the agentmain method, which also has two possible signatures [6]:

1. public static void agentmain(String agentArgs, Instrumentation inst)
2. public static void agentmain(String agentArgs)

The mechanism for triggering this dynamic load is the Java Attach API.[6] This API, found in the

com.sun.tools.attach package (and jdk.attach module in modern JDKs), allows a separate Java process (an "injector") to connect to a target JVM process and instruct it to load an agent.[29] The typical workflow is:

1. The injector application identifies the process ID (PID) of the target JVM.
2. It calls VirtualMachine.attach(pid) to establish a connection to the target JVM.[31]
3. It then calls vm.loadAgent(agentJarPath, options) to command the target JVM to load the specified agent JAR.[22]
4. The target JVM receives this command, loads the agent JAR, finds the class specified by the Agent-Class manifest attribute, and invokes its agentmain method.[21]

This ability to instrument a running system is a cornerstone of modern APM tools and advanced debugging utilities.

The choice between static and dynamic loading is not merely technical but strategic. Static loading is simpler and ensures comprehensive instrumentation from the very start of the application's lifecycle. It is ideal for baseline observability and

performance monitoring that should always be active. Dynamic loading, while more complex to orchestrate, is indispensable for on-demand diagnostics, enabling engineers to attach profiling or debugging tools to a live production environment to investigate issues without incurring the cost of a restart. A robust agent framework like Disco is designed to support both paradigms, allowing developers to choose the appropriate strategy for their needs.

Chapter 4: The Classloader Conundrum: A Critical Challenge for Agents

While the Instrumentation API provides the *means* to modify bytecode, the JVM's class loading architecture presents significant challenges that any non-trivial agent must overcome. These challenges revolve around class visibility and dependency management.

The JVM Classloader Hierarchy

The JVM does not load all classes from a single location. Instead, it employs a hierarchical system of classloaders, each responsible for a specific set of classes. This system follows a strict delegation model. The three primary built-in classloaders are:

1. **Bootstrap Classloader**: The root of the hierarchy. It is typically implemented in native code and is responsible for loading the core Java API classes (from rt.jar in older JDKs, or core modules in modern JDKs).[1] In Java code, it is represented by null.
2. **Platform (or Extension) Classloader**: The child of the Bootstrap Classloader. It loads classes from the JDK's extension directories.[1]
3. **System (or Application) Classloader**: The child of the Platform Classloader. It is responsible for loading the classes from the application's classpath, which is specified via the -cp command-line option or the CLASSPATH environment variable.[1]

The Parent-First Delegation Model

When a request to load a class (e.g., com.example.MyClass) is made, the process follows a strict "parent-first" delegation model.

1. The request typically starts at the System Classloader.
2. The System Classloader does not immediately search its own path. Instead, it first **delegates** the request to its parent, the Platform Classloader.
3. The Platform Classloader, in turn, delegates the request to *its* parent, the Bootstrap Classloader.
4. The Bootstrap Classloader searches its path. If it finds the class, it loads it, and the process ends.
5. If the Bootstrap Classloader fails, the request returns to the Platform Classloader, which now searches its own path. If it succeeds, it loads the class.
6. If the Platform Classloader also fails, the request finally returns to the System Classloader, which searches its application classpath.
7. If the System Classloader cannot find the class, a ClassNotFoundException is thrown.[33]

This model ensures that core Java classes are always loaded by the Bootstrap Classloader, preventing applications from accidentally or maliciously replacing them. It also ensures that a class is loaded only once by the highest-level loader in the hierarchy that can find it.

The Agent's Dilemma

This strict hierarchy creates two major problems for Java Agents.

1. **The Visibility Problem**: By default, an agent's classes are loaded by the System Classloader.[21] Now, consider an agent that needs to instrument a core JDK class like java.net.HttpURLConnection, which is loaded by the Bootstrap Classloader. The agent's ClassFileTransformer can successfully intercept the bytecode of HttpURLConnection. However, if the instrumentation code it injects tries to call a helper class from the agent's own JAR (e.g., com.myagent.HttpLogger), a NoClassDefFoundError will occur at runtime. This is because classes loaded by the Bootstrap Classloader cannot "see" or reference classes loaded by its children, like the System Classloader.[33] The agent's helper classes are invisible to

the core JDK classes it is trying to modify. The traditional solution is to use the Boot-Class-Path manifest attribute to force the agent's helper JARs onto the bootstrap classpath, but this is a blunt and often problematic approach.[24]

2. **The Dependency Conflict Problem ("JAR Hell")**: An agent is a guest in a host application. What happens if the agent requires a specific version of a library, say Guava v30, for its internal logic, but the host application it is attached to depends on Guava v25? Since both the agent's JAR and the application's JARs are typically on the System Classloader's path, the JVM will load only one version of the Guava classes. This can lead to subtle and difficult-to-diagnose runtime errors like NoSuchMethodError if the agent's code calls a method that exists in v30 but not in v25.[34] This is a classic example of the "diamond dependency problem," and it is particularly acute for agents, which must coexist peacefully with an unknown and potentially conflicting set of application dependencies.

For any piece of tooling designed to be universally applicable, such as a monitoring or tracing agent, operating as a well-behaved guest is paramount. It must not interfere with the host application's own dependencies or classloading behavior. This necessity for robust isolation reveals why simple agent implementations are often fragile in real-world environments. It also provides the critical context for understanding the sophisticated engineering decisions behind frameworks like the Amazon Disco Toolkit. The heavy investment in custom classloaders and dependency shading techniques is not an optional enhancement; it is a fundamental requirement to solve the core challenges of the Java Agent ecosystem and create a tool that is safe and reliable enough for production use.[35]

Chapter 5: The Art of Bytecode Manipulation

At the heart of a ClassFileTransformer lies the task of altering bytecode. This is not done by manipulating Java source code, but by directly modifying the low-level instruction set that the JVM executes.[8] Several libraries have been developed to abstract away the raw complexity of the class file format, each offering different trade-offs between performance, flexibility, and ease of use.

A High-Level View

When a transform method receives the classfileBuffer byte array, it is effectively looking at the compiled representation of a Java class. To modify this, a developer almost always uses a specialized library. These libraries parse the byte array into a structured representation, allow for modifications, and then serialize the structure back into a valid byte array for the JVM to consume.[38]

Comparative Analysis of Libraries

The landscape of bytecode manipulation is dominated by three major libraries. The choice of which library to use has significant implications for the development of an agent, and understanding their differences helps to justify why modern frameworks like Disco and Mockito have standardized on Byte Buddy.

Library	API Level	Key Characteristics	Common Use Cases
ASM	Low-Level (Visitor API)	Extremely fast with minimal memory overhead. It requires a deep understanding of JVM bytecode instructions and the class file format. ASM is the de-facto standard low-level tool and is used as the underlying engine for many other bytecode libraries and compilers. [39]	High-performance frameworks (Spring), compilers (Kotlin, Groovy), and as the engine for higher-level libraries like Byte Buddy and CGLIB. [39]
Javassist	High-Level (Source-level)	Allows developers to write Java code as String literals, which Javassist then compiles and injects into the target class. This makes it much easier to learn than	Rapid prototyping, simpler instrumentation tasks where performance is not the primary concern, and for educational purposes to understand

		ASM for simple tasks. However, it can be slower, and its internal compiler has historically lagged behind the latest Java language features. [42]	bytecode modification concepts without the steep learning curve of ASM. [45]
Byte Buddy	High-Level (Fluent API)	Provides a modern, type-safe, and highly expressive domain-specific language (DSL) for creating and modifying classes. It uses a "pluggable" interception model based on delegation rather than string-based code. Built on top of ASM for performance, it is actively maintained and offers a powerful balance of ease-of-use and flexibility. [46]	Modern Application Performance Monitoring (APM) tools, mocking frameworks (Mockito), and agent development frameworks like the Amazon Disco Toolkit. [39]

Byte Buddy's approach represents a significant evolution in bytecode manipulation. By providing a type-safe, fluent API, it mitigates the risks of runtime errors from malformed string-based code (a common issue with Javassist) while abstracting away the verbosity and complexity of the raw ASM visitor API. This combination of safety, power, and developer productivity makes it the logical choice for a modern instrumentation framework like Disco.

Part II: The Amazon Disco Toolkit: Architecture and Core Concepts

Having established the foundational principles of Java instrumentation, we can now

analyze the specific architecture of the Amazon Disco Toolkit. Disco is not merely a single tool but a comprehensive framework designed to address the unique observability challenges posed by modern, distributed, service-oriented architectures.

Chapter 6: DiSCo's Philosophy: Comprehending Distributed Systems

The name "Disco" is an acronym for **Di**stributed **S**ystems **Co**mprehension.[49] This name encapsulates its core mission: to provide the necessary tools and runtime primitives to understand the complex, asynchronous, and multi-threaded behavior of applications built as a collection of microservices. It is important to note that Disco is officially designated as pre-release software (with versions below 1.0), which implies that its APIs and features are subject to change and should be used with this consideration in mind.[49]

The Core Problem

In a monolithic application, tracing a request is relatively straightforward as it typically executes within a single process and often on a single thread. In a distributed system, a single user action can initiate a complex cascade of events. For example, a call to an API gateway might trigger an authentication service, which then calls a user profile service and an order service. The order service, in turn, might submit multiple tasks to a thread pool to query inventory and shipping services in parallel.[49]

Traditional logging and tracing tools often fail in this environment. Log entries from different threads within the same service appear interleaved and uncorrelated. When an error occurs in a worker thread from a thread pool, its log entries are "orphaned," with no clear link back to the original request that initiated the work. This makes debugging and performance analysis exceedingly difficult.[49]

Disco's Solution

Disco addresses this fundamental challenge by providing two key runtime primitives, implemented as extensions on top of the JVM:

1. An **in-process Event Bus** to publish significant lifecycle events.
2. A **Transactionally Scoped data dictionary**, the TransactionContext, to carry state and a consistent identity across thread and service boundaries.

Together, these primitives aim to transform the chaotic, interleaved stream of operations in a distributed system into a coherent, traceable narrative for each individual transaction.[49]

Chapter 7: Core Primitive I: The In-Process Event Bus

At its heart, Disco employs an event-driven architecture. Instead of embedding monitoring logic directly at points of interest, Disco's instrumentation publishes generic, decoupled events to an in-process event bus. This bus serves to "advertise moments in the lifetime of a transaction or activity".[49]

Key Event Types

The standard Disco plugins are designed to instrument common frameworks and libraries, publishing a standardized set of events that describe the flow of a transaction. These events provide the raw data for any observability tool built on top of Disco. Key event types include:

- **ServiceRequestEvent / ServiceResponseEvent**: Published when a service receives an incoming request and when it sends the final response. For example, a WebRequestEvent is published by the web plugin when a servlet's service() method is invoked.[49]
- **ServiceDownstreamRequestEvent / ServiceDownstreamResponseEvent**: Published when the service makes an outgoing call to another service (a "downstream" dependency). This could be an HTTP call made with an Apache client, an AWS SDK request, or a JDBC query.[50]
- **ThreadEnterEvent / ThreadExitEvent**: Published to track the lifecycle of threads

within the JVM.
- **ForkEvent / JoinEvent**: These are crucial for tracking asynchronous operations. A ForkEvent is published when work is submitted to a different thread (e.g., via ExecutorService.submit()), and a JoinEvent is published when the result is retrieved.[49]

The Listener Model

Tooling is constructed by creating Listener classes. These classes contain methods annotated with @Listener that subscribe to specific event types on the bus. For instance, a logging tool would implement a listener that subscribes to all the above events and writes them to a file. This model elegantly decouples the act of *instrumentation* (publishing events) from the act of *analysis* (consuming events), making the system highly modular and extensible.

Chapter 8: Core Primitive II: The TransactionContext

While the Event Bus provides the raw data, the TransactionContext provides the essential "glue" that makes this data meaningful in a concurrent and distributed environment.

Beyond ThreadLocal

Java developers have long used the ThreadLocal class to store data that is specific to a single thread. However, ThreadLocal is fundamentally inadequate for modern, asynchronous applications. When a task is submitted to a thread pool, the work is executed on a different thread. The ThreadLocal context of the submitting thread is lost, breaking the chain of causality.[49]

The Solution: A Transactionally-Scoped Dictionary

The TransactionContext is Disco's solution to this problem. It is a thread-safe, map-like data store that is created at the beginning of a transaction's lifecycle (e.g., upon receiving an HTTP request) and is designed to persist until the transaction concludes.[49]

The most powerful feature of the TransactionContext is its automatic propagation across thread handoffs. Disco achieves this by instrumenting core Java concurrency classes, such as java.lang.Thread, java.util.concurrent.ExecutorService, and java.util.concurrent.ForkJoinPool. When a "forking" thread passes work to a "forked" thread, Disco's instrumentation ensures that the forked thread inherits a reference to the parent's TransactionContext.[49]

The Transaction ID

By default, every new TransactionContext is automatically populated with a unique 96-bit random hexadecimal string, which serves as the **Transaction ID**. This ID remains immutable and consistent throughout the entire lifecycle of that specific transaction, no matter how many threads it crosses within a single service.[49] If this ID is propagated to downstream services (e.g., via HTTP headers), it can be used to trace an entire request flow across a distributed system.

The TransactionContext is the central and most innovative element of Disco's architecture. While an event bus is a common design pattern, the automatic, transparent propagation of a consistent context across asynchronous boundaries is what truly distinguishes the toolkit. It directly solves the "orphaned thread" problem by ensuring that every event, regardless of which thread it originated from, can be correlated back to a single, unique transaction ID. This transforms a potentially chaotic stream of unrelated events into a coherent, queryable narrative for each activity, which is the foundational requirement for effective distributed systems comprehension.

Chapter 9: A Modular Ecosystem: The Power of Plugins

To promote flexibility and prevent monolithic bloat, Disco is designed as a modular, pluggable system. The core agent itself is lightweight and acts as a substrate for discovering and loading plugins that provide specific instrumentation capabilities.[49]

Decoupled Instrumentation

This architecture allows users to tailor their monitoring solution to their specific needs. An application that doesn't use a database has no need to load the SQL instrumentation plugin, reducing overhead and potential conflicts. The core agent, disco-java-agent.jar, is responsible for the fundamental agent lifecycle and plugin loading, while the actual bytecode transformations are delegated to the plugins themselves.[51]

Official Plugins

The Disco project provides several official plugins that serve as both production-ready tools and canonical examples for developers wanting to build their own instrumentation. These include:

- **disco-java-agent-web-plugin**: Provides instrumentation for standard Java Servlets and the widely used Apache HTTP client libraries.[49]
- **disco-java-agent-sql-plugin**: Instruments Java Database Connectivity (JDBC) calls, allowing for the tracing of database queries.[49]
- **disco-java-agent-aws-plugin**: Instruments the AWS SDK for Java (both v1 and v2), publishing events for downstream calls to AWS services.[50]
- **disco-java-agent-kotlin-plugin**: Adds support for context propagation across Kotlin Coroutines, demonstrating the framework's extensibility to other JVM languages and concurrency models.[36]

The pluginPath Argument

The plugin discovery mechanism is configured via an argument to the -javaagent flag. The user specifies a pluginPath that points to a directory on the filesystem. The agent will scan this directory at startup and load any valid plugin JARs it finds.[51]

Example startup command:
-javaagent:/path/to/disco-java-agent.jar=pluginPath=/path/to/disco-plugins
This design allows for a high degree of customization. Users can mix and match official and custom-built plugins simply by placing their respective JAR files into the designated plugin directory.

Chapter 10: Taming Dependencies: How Disco Manages Classloader Isolation

As established in Chapter 4, classloader visibility and dependency conflicts are among the most severe challenges in building robust Java Agents. Disco employs a sophisticated, multi-layered strategy to overcome these issues, prioritizing runtime stability and ease of deployment.

Revisiting the Problem

An agent must coexist with its host application without causing NoClassDefFoundError due to class visibility issues or NoSuchMethodError due to conflicting dependency versions. A failure in the agent must not bring down the application it is meant to be monitoring.

Disco's Multi-Layered Solution

Disco's approach to this problem demonstrates a deep understanding of JVM internals and a commitment to production-grade engineering.

1. **JAR Shading/Shadowing**: The Disco documentation explicitly states that its build process uses JAR shading to manipulate package namespaces.[53] This is a

technique where a build tool, such as the Maven Shade Plugin or Gradle Shadow Plugin, bundles an agent's dependencies *inside* the agent's final JAR file. Critically, it also renames the packages of these bundled dependencies. For example, if the agent uses com.google.common.collect.ImmutableList, the shading process renames it to something like software.amazon.disco.shaded.com.google.common.collect.ImmutableList and rewrites the agent's own bytecode to call the renamed class. This effectively creates a private, internal copy of the dependency, making it impossible to conflict with any version of the same library that the host application might be using.[54]

2. **PluginClassLoader**: To further enhance isolation, particularly between different plugins, Disco introduced a custom PluginClassLoader.[36] Each plugin is loaded into its own classloader instance, creating a strong boundary that prevents one plugin's dependencies from interfering with another's. This addresses a potential "JAR hell" scenario *within the agent itself* if multiple plugins were to depend on conflicting versions of the same library.
3. **ResourcesClassInjector**: This component solves the most difficult visibility problem. If a plugin needs to instrument a class loaded by the Bootstrap Classloader (e.g., a core JDK class) but also needs to use its own helper classes, the ResourcesClassInjector provides a mechanism to solve this. It works by packaging the plugin's helper classes as resources within the plugin JAR and then, at runtime, programmatically injecting those classes into the same classloader as the target class (e.g., the Bootstrap Classloader). This makes the helper classes visible to the instrumented code, resolving the NoClassDefFoundError issue without polluting the global bootstrap classpath.[53]

The combination of these techniques represents a pragmatic and robust engineering trade-off. The complexity of the agent's build process is intentionally increased to ensure that its deployment and runtime behavior are as simple and stable as possible. For the end-user, the result is a single, self-contained agent JAR that can be attached to an application with a high degree of confidence that it will not cause classloading or dependency-related failures. This design choice prioritizes runtime safety and ease of use, which are the correct priorities for any tool intended for production environments.

Part III: A Practical Tutorial: Building and Using a Disco Agent

This section transitions from architectural theory to hands-on application. It provides a step-by-step guide to building, packaging, and deploying a custom Java Agent using the Amazon Disco Toolkit. The goal is to create a simple yet illustrative agent that logs transaction lifecycles, demonstrating the core value of the TransactionContext.

Chapter 11: Setting Up Your Development Environment

A successful build requires a correctly configured environment. The Disco toolkit has specific dependencies that must be met.

Prerequisites

- **Java 8 JDK**: The Disco GitHub repository explicitly states that the project must be built with a Java 8 JDK. Using a newer version can lead to build failures related to dependency resolution. It is crucial to ensure that the JAVA_HOME environment variable points to a valid Java 8 installation.[49]
- **Gradle**: The project uses Gradle as its build tool. A recent version of Gradle should be installed and available on the system's PATH.

Project Setup with Gradle

This tutorial will use Gradle with the Kotlin DSL (build.gradle.kts) for project setup.

1. **Create a New Gradle Project**: Initialize a new Java application project using your IDE or the Gradle command line: gradle init. Select 'application', 'Java', and 'Kotlin' for the build script DSL.
2. **Configure build.gradle.kts**: Open the generated build.gradle.kts file and configure it for agent development.

3. **Add the Shadow Plugin**: The key to packaging a self-contained agent is creating a "fat jar" or "uber jar" that includes all dependencies. The Gradle Shadow plugin is the standard tool for this.[57] Add the plugin to the plugins block:

```
Kotlin
plugins {
    java
    application
    id("com.github.johnrengelman.shadow") version "7.1.2" // Or a more recent version
}
```

4. **Add Disco Dependencies**: To ensure consistent versions across all Disco modules, it is best practice to use the Bill of Materials (BOM) package. This is done by declaring a platform dependency. Then, add the specific Disco APIs needed for the agent.[49]

```
Kotlin
dependencies {
    // Import the Disco Bill of Materials (BOM) to manage dependency versions
    implementation(platform("software.amazon.disco:disco-toolkit-bom:0.13.0"))

    // Add the core Disco agent API
    implementation("software.amazon.disco:disco-java-agent-api")

    // Add the web interceptor to generate events for web requests
    implementation("software.amazon.disco:disco-java-agent-web")
}
```

Chapter 12: Walkthrough: Creating a Custom Transaction Logging Agent

The goal is to build an agent that listens for the beginning and end of web requests and prints a log message prefixed with the unique Transaction ID, demonstrating Disco's context propagation.

Step 1: The Listener (MyListener.java)

The listener class contains the logic that reacts to events published on the EventBus.

Java

```java
package com.example.disco.agent.example;

import software.amazon.disco.agent.event.Event;
import software.amazon.disco.agent.event.Listener;
import software.amazon.disco.agent.event.ServiceRequestEvent;
import software.amazon.disco.agent.event.ServiceResponseEvent;
import software.amazon.disco.agent.transaction.Transaction;

public class MyListener {
    /**
     * Listens for the start of a service request.
     * @param event the event published by the Disco Event Bus.
     */
    @Listener
    public void onServiceRequest(ServiceRequestEvent event) {
        String txId = Transaction.get().getTransactionId();
        System.out.printf(" Service request received. Origin: %s, Method: %s%n",
                txId, event.getOrigin(), event.getOperation());
    }

    /**
     * Listens for the end of a service request.
     * @param event the event published by the Disco Event Bus.
     */
    @Listener
    public void onServiceResponse(ServiceResponseEvent event) {
        String txId = Transaction.get().getTransactionId();
        System.out.printf(" Service response sent. Origin: %s%n",
                txId, event.getOrigin());
    }
}
```

In this class, methods are annotated with @Listener. Disco's event bus scans for this annotation and registers the method to be called when an event of the corresponding parameter type (e.g., ServiceRequestEvent) is published. Inside the method, Transaction.get() retrieves the current TransactionContext, from which the consistent transactionId is obtained.[49]

Step 2: The Agent (MyAgent.java)

The agent class is the entry point that the JVM will call. Its responsibility is to initialize and register the necessary components.

Java

```
package com.example.disco.agent.example;

import software.amazon.disco.agent.DiscoAgentTemplate;
import software.amazon.disco.agent.interception.Installable;
import software.amazon.disco.agent.web.WebInterceptor;

import java.lang.instrument.Instrumentation;
import java.util.Collections;
import java.util.List;

public class MyAgent extends DiscoAgentTemplate {
    /**
     * The premain entry point, called by the JVM at startup.
     * @param agentArgs arguments passed to the agent on the command line
     * @param instrumentation the Instrumentation instance provided by the JVM
     */
    public static void premain(String agentArgs, Instrumentation instrumentation) {
        new MyAgent(agentArgs).agentmain(agentArgs, instrumentation);
    }

    public MyAgent(String agentArgs) {
        super(agentArgs);
```

```
    }

    /**
     * Overridden to supply custom listeners.
     * @return a list of listeners to add to the event bus.
     */
    @Override
    public List<software.amazon.disco.agent.event.Listener> getListeners() {
        return Collections.singletonList(new MyListener());
    }

    /**
     * Overridden to supply the Installables that generate events.
     * @return a list of Installables to apply to the JVM.
     */
    @Override
    public List<Installable> getInstallables() {
        // This interceptor is what generates the ServiceRequestEvent and ServiceResponseEvent
        return Collections.singletonList(new WebInterceptor());
    }
}
```

This class uses DiscoAgentTemplate for convenience. The most critical steps are:

1. Defining the static premain method, which is the required entry point for the JVM.[19]
2. Registering our MyListener with the EventBus by returning it from getListeners().
3. **Crucially**, installing the WebInterceptor. Without this, no ServiceRequestEvent or ServiceResponseEvent objects would ever be published, and our listener would never be called. The interceptor contains the ClassFileTransformer logic that actually instruments the servlet classes.

Step 3: The Manifest (src/main/resources/META-INF/MANIFEST.MF)

A manifest file is required to tell the JVM which class contains the premain method.

```
Manifest-Version: 1.0
Premain-Class: com.example.disco.agent.example.MyAgent
Can-Retransform-Classes: true
```

This file must be placed in the src/main/resources/META-INF directory to be included correctly in the final JAR.[25]

Step 4: The Build Script (build.gradle.kts)

Finally, the Gradle Shadow plugin must be configured to create the fat jar and embed the manifest attributes.

Kotlin

```
tasks.shadowJar {
    archiveBaseName.set("my-disco-agent")
    archiveClassifier.set("all")
    archiveVersion.set("1.0.0")

    manifest {
        attributes(
            "Premain-Class" to "com.example.disco.agent.example.MyAgent",
            "Can-Retransform-Classes" to "true"
        )
    }
}
```

With this configuration, running gradle shadowJar will produce a single, self-contained JAR file named my-disco-agent-1.0.0-all.jar in the build/libs directory.

Chapter 13: Walkthrough: Deploying and Observing the Agent

With the agent JAR built, the final step is to run it against a target application to observe its behavior.

Creating a Target Application

A simple Spring Boot web application serves as an excellent target. The following controller defines a single endpoint:

Java

```
// In a separate Spring Boot project
package com.example.webapp;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("/")
    public String index() {
        System.out.println("Processing request for /");
        return "Greetings from Spring Boot!";
    }
}
```

This application should be packaged into an executable JAR file (e.g., my-web-app.jar).

Running With the Agent

To run the web application with the custom Disco agent, use the -javaagent flag, pointing to the fat jar created in the previous chapter.

java -javaagent:/path/to/my-disco-agent-1.0.0-all.jar -jar /path/to/my-web-app.jar

Analyzing the Output

First, run the web application *without* the agent and send a few concurrent requests (e.g., using curl http://localhost:8080 in multiple terminals). The log output will be simple and uncorrelated:

```
Processing request for /
Processing request for /
Processing request for /
```

Now, stop the application and restart it *with* the -javaagent flag. Send concurrent requests again. The output will be dramatically different:

```
 Service request received. Origin: WebRequest, Method: GET
Processing request for /
 Service response sent. Origin: WebRequest
 Service request received. Origin: WebRequest, Method: GET
Processing request for /
 Service response sent. Origin: WebRequest
 Service request received. Origin: WebRequest, Method: GET
Processing request for /
 Service response sent. Origin: WebRequest
```

This output clearly demonstrates the power of the Disco toolkit. Each request is assigned a unique transaction ID (TX:...) that persists from the ServiceRequestEvent to the ServiceResponseEvent. Even though the requests are processed concurrently, their log entries are now perfectly correlated, solving the core problem of observability in multi-threaded environments.

Part IV: Advanced Topics and Real-World Application

While the tutorial demonstrates the core functionality of Disco, deploying agents in production environments requires consideration of more advanced topics, from integration with existing systems to security and performance. This section explores these aspects, using a real-world product as a case study.

Chapter 14: Case Study: Deconstructing the AWS X-Ray Agent

The most prominent and instructive example of the Amazon Disco Toolkit in practice is the official AWS X-Ray auto-instrumentation agent for Java. This agent provides automatic tracing for a variety of frameworks with minimal configuration, and its architecture is built directly on Disco's principles.[52]

Disco in the Wild

The X-Ray agent is designed to instrument Java web applications to capture and send trace data to the AWS X-Ray service. It automatically traces incoming servlet requests and downstream calls made with the AWS SDK, Apache HTTP clients, and JDBC drivers.[52] The fact that AWS chose Disco as the foundation for this official product is a strong testament to the framework's robustness and suitability for production-grade observability tooling.

The Power of pluginPath

An analysis of the X-Ray agent's installation instructions reveals its reliance on Disco's plugin architecture. The required JVM argument is [52]:

-javaagent:/<path-to-disco>/disco-java-agent.jar=pluginPath=/<path-to-disco>/disco-plugins

This command line breaks down as follows:

1. It uses the canonical, pluggable disco-java-agent.jar as the main agent entry point. This JAR contains the core agent logic, including the Event Bus, Transaction Context management, and the plugin discovery system.
2. It uses the pluginPath argument to point to a separate directory, disco-plugins. This directory contains a set of specialized JARs, such as aws-xray-agent-plugin.jar, disco-java-agent-aws-plugin.jar, and disco-java-agent-sql-plugin.jar.

This separation is a direct application of Disco's modular design. The core agent provides the "bus," while the plugins in the pluginPath directory provide the specific instrumentation logic. The aws-xray-agent-plugin.jar likely contains the listeners that subscribe to Disco events and translate them into X-Ray segments and subsegments.

Customization and Troubleshooting

This plugin-based architecture offers a powerful mechanism for customization. As noted in the AWS documentation, users can toggle tracing for different event types by simply adding or removing plugin JARs from the pluginPath directory. For example, to disable the tracing of SQL queries, a user can remove the disco-java-agent-sql-plugin.jar from the directory and restart the application.[51] This allows users to fine-tune the agent's behavior and reduce overhead without needing to recompile any code.

Furthermore, troubleshooting the X-Ray agent often involves enabling Disco's own internal logging. This is achieved by passing additional arguments to the -javaagent flag, such as loggerfactory and verbose, which are parsed by Disco's core agent

configuration system.[51]

The architecture of the AWS X-Ray agent validates Disco's entire approach. It proves that the separation of a core agent substrate from a collection of independent, swappable plugins is not just a theoretical concept but a practical and flexible model for building a complex, feature-rich observability product.

Chapter 15: Beyond the Basics: Further Considerations

Deploying any agent in a production environment requires careful planning around integration, performance, and security.

Building Custom Plugins

While Disco provides plugins for common libraries, developers will often need to instrument custom or unsupported frameworks. The process involves:

1. Identifying the key methods in the target library to intercept (e.g., the entry point of a request handler).
2. Creating a new Installable class that uses a bytecode manipulation library like Byte Buddy to define the interception.
3. The interception logic should not contain the analysis code itself but should instead publish a custom event to the EventBus.
4. A corresponding Listener is then created to subscribe to this new event and perform the desired analysis. This maintains the decoupled architecture.

Managed Runtimes (AWS Lambda)

In environments like AWS Lambda, developers do not have direct control over the java command-line arguments. The Disco documentation notes that agents can be "injected" into such runtimes.[49] This typically involves adding the agent JAR to the application's dependencies and then programmatically invoking the agent's loading

mechanism as the very first step in the application's

main method or handler constructor. This must be done before any of the classes to be instrumented are loaded or used by the application, as Disco's bytecode manipulation may not work correctly on classes that have already been initialized.

Performance and Overhead

Instrumentation is not free. Adding code to methods, even via bytecode manipulation, introduces CPU and memory overhead.[60] While Disco is designed to be lightweight, heavily instrumented applications can experience performance degradation. Developers should:

- Be selective about which plugins are enabled.
- Consider the performance impact of their listener logic.
- Leverage features like Disco's instrumentation-preprocess module, which aims to mitigate startup latency by performing instrumentation at build-time rather than at runtime.[36] This shifts the performance cost from the application's startup to the CI/CD pipeline.

Security Considerations

Attaching an agent to a JVM is a privileged operation that requires careful security management.

- **File System Permissions**: The operating system user that owns the running Java process must have read permissions on the agent JAR file and its plugin JARs. It also requires write and execute permissions for the directory where the agent will write its log files.[62]
- **Java Security Manager**: If the target application is running with a Java SecurityManager enabled, the agent will be subject to its policies. This often requires granting explicit permissions to the agent's code base in a java.policy file. A common, though broad, permission is to grant java.security.AllPermission to the agent's JAR file location.[63] Without the correct permissions, the agent may fail with

AccessControlException when trying to perform operations like accessing files or getting classloaders.
- **JMX Communication**: For agents that expose management interfaces via Java Management Extensions (JMX), securing these endpoints is critical in production. Unsecured JMX endpoints can allow unauthorized users to monitor and control the application. Best practices include enabling password authentication and using SSL/TLS to encrypt JMX communication, as well as using firewalls to restrict network access to JMX ports.[66]

Conclusion: The Future of Observability with Disco

The Amazon Disco Toolkit emerges as a powerful and thoughtfully engineered framework for building sophisticated observability tools for distributed Java applications. Its design directly confronts the most difficult challenges in this domain: maintaining transactional context across asynchronous and multi-threaded boundaries, and managing the complexities of classloading and dependency conflicts in a multi-agent environment.

The toolkit's primary strengths lie in its core architectural decisions. The combination of the **in-process Event Bus** and the automatically propagated **TransactionContext** provides a robust mechanism for transforming disparate runtime events into a coherent, traceable narrative for each transaction. This directly solves the "orphaned thread" problem that plagues traditional logging and tracing in asynchronous systems. Furthermore, the **pluggable architecture**, exemplified by the AWS X-Ray agent, allows for a high degree of customization and modularity, enabling developers to build tailored monitoring solutions by simply adding or removing instrumentation plugins. Finally, Disco's sophisticated use of **JAR shading and custom classloaders** demonstrates a commitment to production-readiness, ensuring that agents can be deployed safely as well-behaved guests within a host application without causing dependency conflicts.

The Amazon Disco Toolkit is not an out-of-the-box APM solution but rather a framework for the *builders* of such solutions. It is ideally suited for organizations and teams that have unique observability requirements not fully met by off-the-shelf products. For developers who need to create custom, low-level instrumentation for their specific distributed architecture—whether for performance profiling, detailed

logging, or security auditing—Disco provides a powerful and well-designed foundation.

It is essential for potential adopters to remember that Disco is currently designated as **pre-release software**.[49] As such, its APIs and internal mechanisms may evolve. Developers should engage with the official

awslabs/disco GitHub repository to stay informed of the latest changes, report issues, and contribute to its development before deploying agents built with the toolkit into business-critical production environments.[49] By providing a solid foundation for understanding and manipulating the behavior of complex distributed systems, the Amazon Disco Toolkit represents a significant contribution to the field of Java-based observability.

Works cited

1. How JVM Works - JVM Architecture - GeeksforGeeks, accessed June 17, 2025, https://www.geeksforgeeks.org/java/how-jvm-works-jvm-architecture/
2. Compilation and Execution of a Java Program - GeeksforGeeks, accessed June 17, 2025, https://www.geeksforgeeks.org/compilation-execution-java-program/
3. The Execution Lifecycle of a Java Application, accessed June 17, 2025, https://www.cesarsotovalero.net/blog/how-the-jvm-executes-java-code.html
4. Introduction to Java Agent Programming | GeeksforGeeks, accessed June 17, 2025, https://www.geeksforgeeks.org/introduction-to-java-agent-programming/
5. Exploring Java Agent Programming - DEV Community, accessed June 17, 2025, https://dev.to/adaumircosta/exploring-java-agent-programming-o77
6. What Are Java Agents and How to Profile With Them - Stackify, accessed June 17, 2025, https://stackify.com/what-are-java-agents-and-how-to-profile-with-them/
7. Instrumentation (Java Platform SE 8) - Oracle Help Center, accessed June 17, 2025, https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html
8. Java Instrumentation — A Simple Working Example in Java - DEV Community, accessed June 17, 2025, https://dev.to/rubyshiv/java-instrumentation-a-simple-working-example-in-java-4adm
9. Understanding Java Agents - DZone, accessed June 17, 2025, https://dzone.com/articles/java-agent-1
10. java.lang.instrument (Java SE 11 & JDK 11) - Oracle Help Center, accessed June 17, 2025, https://docs.oracle.com/en/java/javase/11/docs/api/java.instrument/java/lang/instrument/package-summary.html
11. How can i use redefineClasses() method in javaagents - Stack Overflow, accessed June 17, 2025,

https://stackoverflow.com/questions/33034001/how-can-i-use-redefineclasses-method-in-javaagents
12. Instrumentation (Java SE 21 & JDK 21) - IGM, accessed June 17, 2025, https://igm.univ-mlv.fr/~juge/javadoc-21/api/java.instrument/java/lang/instrument/Instrumentation.html
13. Difference between redefineClasses and retransformClasses? - OpenJDK mailing lists, accessed June 17, 2025, https://mail.openjdk.org/pipermail/serviceability-dev/2008-May/000131.html
14. redefine VS. retransform | lsieun, accessed June 17, 2025, https://lsieun.github.io/java-agent/s01ch03/redefine-vs-retransform.html
15. WritingYourOwnJavaAgent | Schuchert Wikispaces, accessed June 17, 2025, https://schuchert.github.io/wikispaces/pages/WritingYourOwnJavaAgent
16. Java Instrumentation - Javapapers, accessed June 17, 2025, https://javapapers.com/core-java/java-instrumentation/
17. ClassFileTransformer (Java SE 17 & JDK 17) - Oracle Help Center, accessed June 17, 2025, https://docs.oracle.com/en/java/javase/17/docs/api/java.instrument/java/lang/instrument/ClassFileTransformer.html
18. Java ClassFileTransformer fails to throw exception - Stack Overflow, accessed June 17, 2025, https://stackoverflow.com/questions/78421704/java-classfiletransformer-fails-to-throw-exception
19. A beginner's guide to Java agents - JVM Advent, accessed June 17, 2025, https://www.javaadvent.com/2019/12/a-beginners-guide-to-java-agents.html
20. Writing a Profiler in 240 Lines of Pure Java | Foojay.io, accessed June 17, 2025, https://foojay.io/today/writing-a-profiler-in-240-lines-of-pure-java/
21. Package java.lang.instrument - Oracle Help Center, accessed June 17, 2025, https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html
22. Java Instrumentation — Java Repositories 1.0 documentation - Read the Docs, accessed June 17, 2025, https://jse.readthedocs.io/en/latest/jdk8/instrumentation.html
23. CLassFileTransformer - udaniweeraratne - WordPress.com, accessed June 17, 2025, https://udaniweeraratne.wordpress.com/tag/classfiletransformer/
24. Using Javassist to log method calls and argument values, how to make a logger class visible in every instrumented class? - Stack Overflow, accessed June 17, 2025, https://stackoverflow.com/questions/29451704/using-javassist-to-log-method-calls-and-argument-values-how-to-make-a-logger-cl
25. How should I specify my Premain-class in manifest.mf when starting JVM with -javaagent, accessed June 17, 2025, https://stackoverflow.com/questions/33730305/how-should-i-specify-my-premain-class-in-manifest-mf-when-starting-jvm-with-jav
26. Guide to Java Instrumentation | Baeldung, accessed June 17, 2025, https://www.baeldung.com/java-instrumentation

27. What is the use of agentmain method in java instrumentation - Stack Overflow, accessed June 17, 2025, https://stackoverflow.com/questions/19786078/what-is-the-use-of-agentmain-method-in-java-instrumentation
28. nazmulidris/java-agent: Example of a JVM instrumentation agent - GitHub, accessed June 17, 2025, https://github.com/nazmulidris/java-agent/
29. Java attach API - attach to JVM of different user - GitHub Gist, accessed June 17, 2025, https://gist.github.com/SubOptimal/516d8dfba07fd12ecc19
30. Using the attach api in Java - Stack Overflow, accessed June 17, 2025, https://stackoverflow.com/questions/62971569/using-the-attach-api-in-java
31. VirtualMachine (Attach API) - Oracle Help Center, accessed June 17, 2025, https://docs.oracle.com/javase/8/docs/jdk/api/attach/spec/com/sun/tools/attach/VirtualMachine.html
32. Class Loaders in Java - Baeldung, accessed June 17, 2025, https://www.baeldung.com/java-classloaders
33. ClassLoader in Java - GeeksforGeeks, accessed June 17, 2025, https://www.geeksforgeeks.org/java/classloader-in-java/
34. Isolation using Java's ClassLoader - Farid Zakaria's Blog, accessed June 17, 2025, https://fzakaria.com/2020/11/10/isolation-using-java-s-classloader
35. opentelemetry-java-instrumentation/docs/contributing/javaagent-structure.md at main - GitHub, accessed June 17, 2025, https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/docs/contributing/javaagent-structure.md
36. disco/CHANGELOG.md at master · awslabs/disco - GitHub, accessed June 17, 2025, https://github.com/awslabs/disco/blob/master/CHANGELOG.md
37. Unleashing the Power of Java's Bytecode Instrumentation: Profiling and Monitoring Applications - 30 Days Coding, accessed June 17, 2025, https://30dayscoding.com/blog/java-bytecode-instrumentation-profiling-monitoring-applications
38. Java vs. Native Agents – And How It Affects Your Code - Harness, accessed June 17, 2025, https://www.harness.io/blog/java-vs-native-agents
39. ASM, accessed June 17, 2025, https://asm.ow2.io/
40. Real-World Bytecode Handling with ASM - Oracle Blogs, accessed June 17, 2025, https://blogs.oracle.com/javamagazine/post/real-world-bytecode-handling-with-asm
41. A Guide to Java Bytecode Manipulation with ASM - Baeldung, accessed June 17, 2025, https://www.baeldung.com/java-asm
42. Javassist/ASM Audit Log — Java Repositories 1.0 documentation, accessed June 17, 2025, https://jse.readthedocs.io/en/latest/jdk8/javassistLog.html
43. Introduction to Javassist | Baeldung, accessed June 17, 2025, https://www.baeldung.com/javassist
44. Hibernate has migrated from Javassist to Byte Buddy for its Proxy implementation - Reddit, accessed June 17, 2025, https://www.reddit.com/r/java/comments/5dm586/hibernate_has_migrated_from_javassist_to_byte/

45. Java Bytecode: Journey to the Wonderland (Part 3) | Foojay Today, accessed June 17, 2025, https://foojay.io/today/java-bytecode-simplified-journey-to-the-wonderland-part-3/
46. byte-buddy/README.md at master - GitHub, accessed June 17, 2025, https://github.com/raphw/byte-buddy/blob/master/README.md
47. Byte Buddy - runtime code generation for the Java virtual machine, accessed June 17, 2025, https://bytebuddy.net/
48. A Guide to Byte Buddy | Baeldung, accessed June 17, 2025, https://www.baeldung.com/byte-buddy
49. awslabs/disco: A suite of tools including a framework for creating Java Agents, for aspect-oriented tooling for distributed systems. - GitHub, accessed June 17, 2025, https://github.com/awslabs/disco
50. disco/disco-java-agent-aws/README.md at master - GitHub, accessed June 17, 2025, https://github.com/awslabs/disco/blob/master/disco-java-agent-aws/README.md
51. The official AWS X-Ray Auto Instrumentation Agent for Java. - GitHub, accessed June 17, 2025, https://github.com/aws/aws-xray-java-agent
52. AWS X-Ray auto-instrumentation agent for Java, accessed June 17, 2025, https://docs.aws.amazon.com/xray/latest/devguide/aws-x-ray-auto-instrumentation-agent-for-java.html
53. disco/disco-java-agent/disco-java-agent-core/README.md at master - GitHub, accessed June 17, 2025, https://github.com/awslabs/disco/blob/master/disco-java-agent/disco-java-agent-core/README.md
54. What is a "shaded" Java dependency? - Software Engineering Stack Exchange, accessed June 17, 2025, https://softwareengineering.stackexchange.com/questions/297276/what-is-a-shaded-java-dependency
55. What is the maven-shade-plugin used for, and why would you want to relocate Java packages? - Stack Overflow, accessed June 17, 2025, https://stackoverflow.com/questions/13620281/what-is-the-maven-shade-plugin-used-for-and-why-would-you-want-to-relocate-java
56. How to Run Github Code Online - YouTube, accessed June 17, 2025, https://www.youtube.com/watch?v=qnPKoCl6Gcw&pp=0gcJCdgAo7VqN5tD
57. Dependencies - Shadow Gradle Plugin - GradleUp, accessed June 17, 2025, https://gradleup.com/shadow/configuration/dependencies/
58. How do I make gradle's build task generate the shadow jar _instead_ of the "regular" jar?, accessed June 17, 2025, https://stackoverflow.com/questions/30309290/how-do-i-make-gradles-build-task-generate-the-shadow-jar-instead-of-the-regu
59. How to Write a Javaagent | JRebel & XRebel by Perforce, accessed June 17, 2025, https://www.jrebel.com/blog/how-write-javaagent
60. Should you use Java Agents to instrument your application? - DevelOtters.com, accessed June 17, 2025,

https://develotters.com/posts/should-you-use-java-agents-to-instrument-your-application/
61. software.amazon.disco:disco-java-agent-instrumentation-preprocess - Maven Central, accessed June 17, 2025, https://central.sonatype.com/artifact/software.amazon.disco/disco-java-agent-instrumentation-preprocess
62. Determine permissions requirements (Java) - New Relic Documentation, accessed June 17, 2025, https://docs.newrelic.com/docs/apm/agents/java-agent/troubleshooting/determine-permissions-requirements-java/
63. Security exceptions when running Java agents on HCL Notes / HCL Domino, accessed June 17, 2025, https://support.hcl-software.com/csm?id=kb_article&sysparm_article=KB0034343
64. Java Security Manager Configuration - Splunk AppDynamics Documentation, accessed June 17, 2025, https://docs.appdynamics.com/display/SaaS24/Java+Security+Manager+Configuration
65. How to identify which permissions to add under server.policy - Splunk Community, accessed June 17, 2025, https://community.splunk.com/t5/AppDynamics-Knowledge-Base/How-to-identify-which-permissions-to-add-under-server-policy/ta-p/738191
66. Deep Dive into Java Management Extensions (JMX) - Stackify, accessed June 17, 2025, https://stackify.com/jmx/
67. JMX Monitoring - A Beginner's Guide to Java Performance | SigNoz, accessed June 17, 2025, https://signoz.io/guides/jmx-monitoring/
68. JMX Monitoring: Your Go-To Guide for Java Application Management - Last9, accessed June 17, 2025, https://last9.io/blog/jmx-monitoring/
69. Issues · awslabs/disco - GitHub, accessed June 17, 2025, https://github.com/awslabs/disco/issues