**ILEVEL PROJECT**

1. BUILD and DEPLOY PROCESS
   1. Ilevel backend Build Process

Step1: Gitlab CI -> Trigger the build pipeline which creates an external TFS build jobs using a python script .

Code Snippet from CI template

==========================

.tfs\_ci\_job\_template:

extends: .non\_git\_job\_template

stage: build

script:

- $SCRIPTS\_DIR/scripts/run\_build.py --buildDefinitions "$BUILD\_DEFINITION\_ID"

rules:

- if: '$CI\_COMMIT\_REF\_NAME == "master"'

- if: '$CI\_PIPELINE\_SOURCE == "merge\_request\_event"'

- if: '$CI\_PIPELINE\_SOURCE == "web"'

Code Snippet from python File

def trigger\_build(self, definitionId, sourceBranch, parameters):

build = Build(definitionId, sourceBranch, parameters)

tfs\_endpoint = f"{global\_vars.TFS\_URL}/{global\_vars.VIEWPOINT\_PROJECT}/\_apis/build/builds"

payload = self.\_\_get\_build\_payload(build)

logging.info(f"Going to trigger TFS build with the following payload {payload}")

response = self.\_\_session.post(tfs\_endpoint, data=payload )

response.raise\_for\_status()

build.update(response.json())

buildWebUrl = f"{global\_vars.TFS\_URL}/{global\_vars.VIEWPOINT\_PROJECT}/\_build/results?buildId={build.id}"

logging.info(f"Triggered build url: {buildWebUrl}")

build.set\_tfs\_web\_url(buildWebUrl)

return build

Step2 – Now TFS build job triggers docker build pipeline in gitlab again using below bash script .

/TFSDeployment/Scripts/DockerBuild/StartGitlabDockerBuild.sh

Snippet of the code .

pipeline\_state=`curl -s --compressed \

-F "ref=${Docker\_Build\_Branch}" \

-F token=${GITLAB\_DOCKER\_BUILD\_TOKEN} \

-F "variables[DockerComposeFile]=${DOCKERCOMPOSEFILE}" \

-F "variables[DockerComposePrimaryFile]=${DOCKERCOMPOSEPRIMARYFILE}" \

-F "variables[DockerComposeAdditionalFile]=${DOCKERCOMPOSEADDITIONALFILE}" \

-F "variables[DockerComposeSonarFile]=${DOCKERCOMPOSESONARFILE}" \

-F "variables[DockerComposeTargetService]=${DOCKERCOMPOSETARGETSERVICE}" \

-F "variables[DockerComposeTestService]=${DOCKERCOMPOSETESTSERVICE}" \

-F "variables[DockerRegistryHostName]=${DOCKERREGISTRYHOSTNAME}" \

-F "variables[DockerImageName]=${DOCKERIMAGENAME}" \

-F "variables[AppName]=${APPNAME}" \

-F "variables[BUILD\_BUILDID]=${BUILD\_BUILDID}" \

-F "variables[BUILD\_REPOSITORY\_URI]=$BUILD\_REPOSITORY\_URI" \

-F "variables[MERGE\_REQUEST\_IID]=${MERGE\_REQUEST\_IID}" \

-F "variables[TARGET\_BRANCH]=${TARGET\_BRANCH}" \

-F "variables[BranchName]=${branch}" \

-F "variables[ProjectID]=${GITLABPROJECTID}" \

-F "variables[SONARPROJECTKEY]=${SONARPROJECTKEY}" \

-F "variables[SonarExcludeCoverage]=${SONAREXCLUDECOVERAGE}" \

-F "variables[SonarExclusions]=${SONAREXCLUSIONS}" \

-F "variables[SonarInclusions]=${SONARINCLUSIONS}" \

-F "variables[SONAR\_START]=$SONARSTART" \

-F "variables[SonarLinksCi]=${SYSTEM\_TASKDEFINITIONSURI}${SYSTEM\_TEAMPROJECT}/\_build?definitionId=$SYSTEM\_DEFINITIONID" \

-F "variables[SonarFile]=$SONARFILE" \

-F "variables[CsprojFilePath]=${CSPROJFILEPATH}" \

-F "variables[ReleaseVersion]=$RELEASEVERSION" \

-F "variables[CsprojFileEdit]=${CSPROJFILEEDIT}" \

-F "variables[CODE\_IPREO\_COM\_READ\_TOKEN]=${CODE\_IPREO\_COM\_READ\_TOKEN}" \

-F "variables[TestResultsFolder]=${TESTRESULTSFOLDER}" \

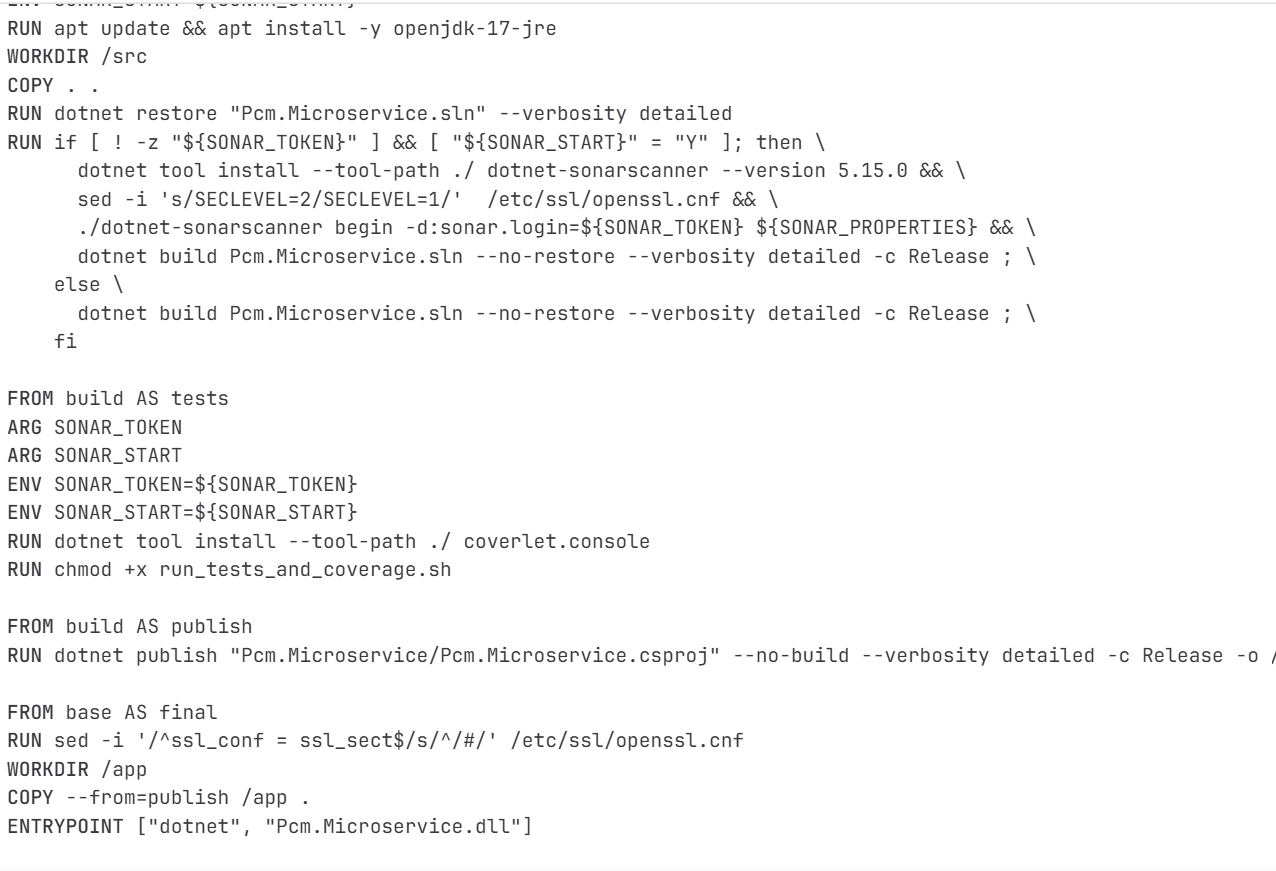
-F "variables[BuiltAssetsFolder]=${BUILTASSETSFOLDER}" \

-F "variables[DockerComposeTargetServiceUp]=${DOCKERCOMPOSETARGETSERVICEUP}" \

-F "variables[BUILD\_WITH\_TESTS]=${BUILDWITHTESTS}" \

-X POST "${GITLAB\_BASE\_URL}/api/v4/projects/${GITLAB\_DOCKER\_BUILD\_PROJECT}/trigger/pipeline"`

Step3 : Docker Build -> Docker Build pipeline initiates the main application build and perform test and analyse sonarqube coverage and pushes images to ECR with the build version.



* 1. ILEVEL backend DEPLOY

Step 1 : TFS Deploy Job initiates octopus deployment for specific component deployment on the specific environment [ DEV, PROD LIKE and PROD].

Step 2. TFS pushes necessary deployment packages to the octopus .

Step3 . Octopus deploy jobs handles the main deployment of the microservice on the ECS cluster

Step4 . Octopus deployment step registers the task definition and update the ECS service and monitor the ecs service deployments and also check the health of the ecs services.

1. **What is a pulsar application ?**

A **Pulsar application** refers to any system, service, or data pipeline built using **Apache Pulsar** — a **cloud-native, distributed messaging and streaming platform**.

Pulsar can handle **real-time event streaming**, **message queuing**, and **pub/sub messaging** — all in one system.

Apache Pulsar is designed as a **multi-layered system**, with **brokers**, **bookies**, and **ZooKeeper** (or metadata store) working together.



**1. Producers**

* Applications that publish (send) messages to a Pulsar **topic**.
* Messages can be:
  + **Persistent** (stored durably)
  + **Non-persistent** (in-memory for low latency)

**2. Consumers**

* Applications that subscribe to topics.
* Supports multiple **subscription modes**:
  + **Exclusive** – only one consumer per subscription
  + **Shared** – multiple consumers share messages (load-balanced)
  + **Failover** – one active, others standby
  + **Key\_Shared** – partitioned delivery based on message keys

**3. Brokers**

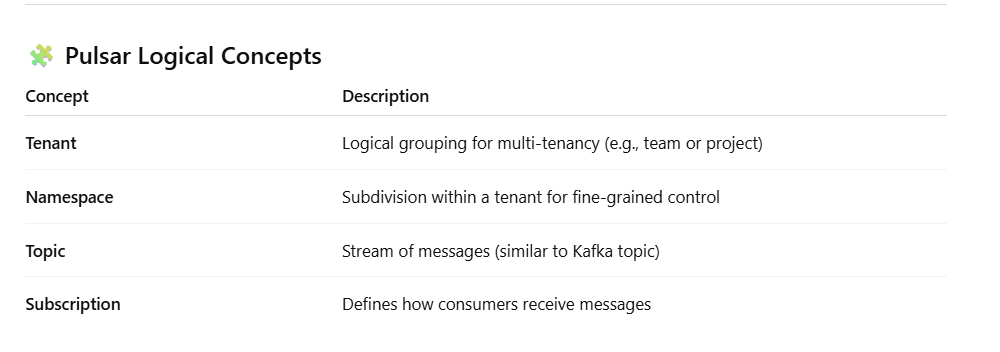
* Stateless servers that:
  + Manage connections between producers and consumers
  + Coordinate message delivery
  + Forward messages to BookKeeper for storage

**4. BookKeeper (Bookies)**

* The **storage layer** that ensures:
  + Durable persistence
  + Replication of messages
  + Fast recovery if a node fails

**5. ZooKeeper / Metadata Store**

* Maintains metadata such as:
  + Topic ownership
  + Cluster configuration
  + Broker assignments  
    *(Newer versions support metadata in etcd as well.)*

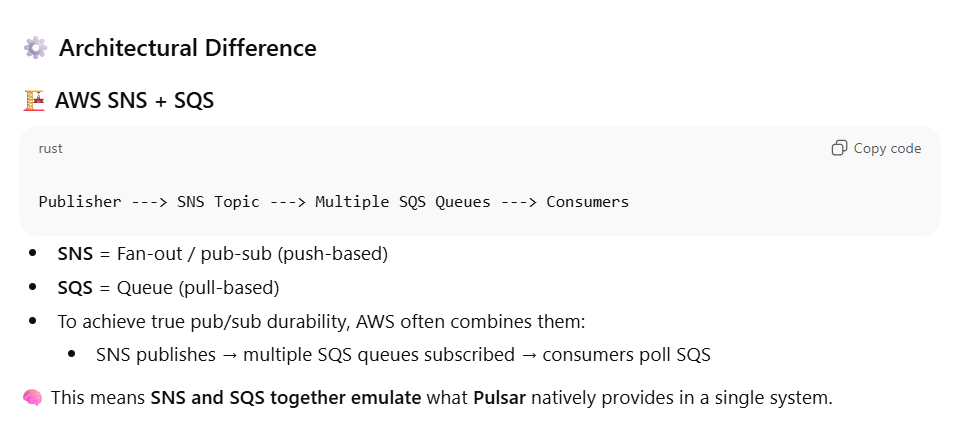


1. **How Apache pulsar is different from AWS SNS and SQS?**

**⚔️ Apache Pulsar vs AWS SNS + SQS**

Both **Apache Pulsar** and **AWS SNS/SQS** provide **messaging** and **pub/sub** capabilities, but they’re **architecturally and functionally quite different**.

| **Feature** | **Apache Pulsar** | **AWS SNS + SQS** |
| --- | --- | --- |
| **Type** | Open-source distributed messaging & streaming platform | Managed message queue (SQS) + pub/sub service (SNS) |
| **Supports both queue + stream semantics** | ✅ (built-in) | ❌ (SNS = pub/sub, SQS = queue; must integrate manually) |
| **Message storage** | Durable, replicated via Apache BookKeeper | SQS persists messages temporarily (up to 14 days) |
| **Scalability** | Horizontally scalable (multi-broker, multi-cluster) | Scales automatically within AWS region |
| **Multi-tenancy** | ✅ Built-in tenants & namespaces | ❌ Each SNS/SQS setup per account |
| **Geo-replication** | ✅ Native, multi-region replication | ⚙️ Needs manual setup (cross-region SNS topics) |
| **Streaming (ordered, replayable)** | ✅ Supports message replay, partitioned topics | ❌ Messages are deleted after consumption |
| **Message retention** | Configurable (can store forever) | Max 14 days retention in SQS |
| **Subscription modes** | Exclusive, Shared, Failover, Key\_Shared | SNS fan-out only (push), SQS poll only |
| **Protocol** | Custom Pulsar protocol + REST/WebSocket | HTTP/S, Lambda, SQS endpoints |
| **Deployment** | Self-managed or via Pulsar Cloud | Fully managed by AWS |
| **Integration with external systems** | Pulsar IO connectors (Kafka, DBs, ES, etc.) | SNS/SQS integrates well with AWS ecosystem only |
| **Latency** | Low (real-time streaming capable) | Low to moderate (depends on delivery mode) |
| **Cost** | Open-source (infra cost only) | Pay per request/message in AWS |
| **Use cases** | Enterprise-grade event streaming, hybrid cloud, multi-region data pipelines | AWS-native asynchronous messaging, decoupling microservices |



A screenshot of a computer

AI-generated content may be incorrect.

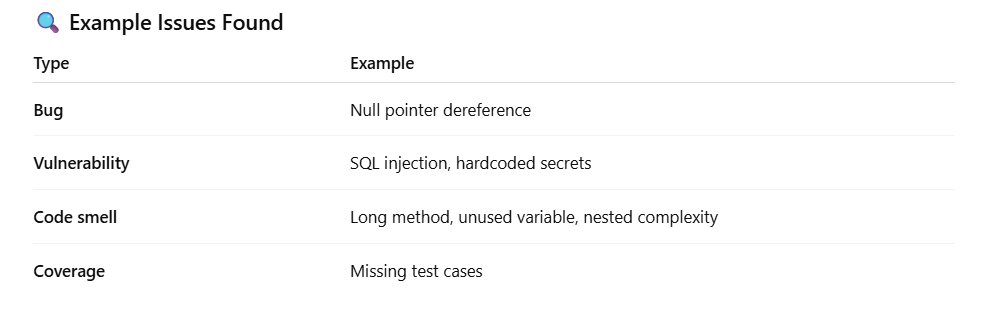
1. Sonarqube Vs Mend Analysis

**🧩 1️⃣ SonarQube Analysis**

**🎯 Purpose**

SonarQube focuses on **your own source code** — written by developers — to detect:

* Code quality issues
* Security vulnerabilities (SAST)
* Code smells and maintainability issues
* Technical debt and test coverage



**🧩 2️⃣ Mend (WhiteSource) Analysis**

**🎯 Purpose**

Mend focuses on **third-party and open-source components** in your project — dependencies you import via:

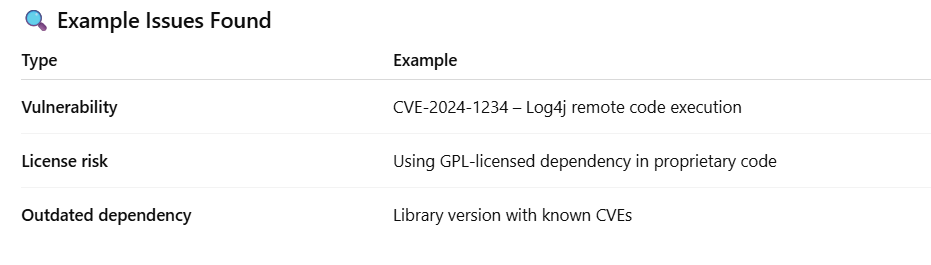
* Maven, npm, pip, NuGet, Go modules, etc.

It performs **Software Composition Analysis (SCA)** to:

* Identify **open-source libraries** used
* Detect **known CVEs (vulnerabilities)**
* Check **license compliance** (GPL, MIT, Apache, etc.)
* Suggest **safe versions or patches**

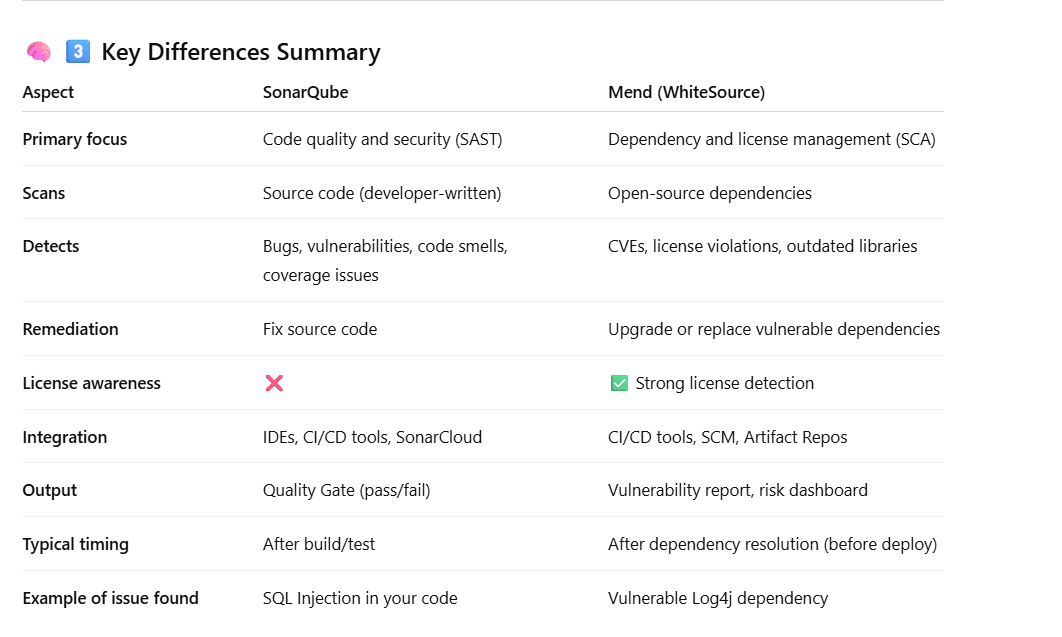
**🧠 How It Works**

1. Mend scans your dependency manifests (e.g., pom.xml, package.json, requirements.txt).
2. It compares each library’s version against its vulnerability and license database.
3. It reports:
   * Vulnerabilities (CVE IDs)
   * Risk levels (High/Medium/Low)
   * Suggested remediations
   * License violations



A screen shot of a computer

AI-generated content may be incorrect.



1. How ECS logs are sent to Splunk ?



**⚙️ 2️⃣ Step-by-Step Integration Methods**

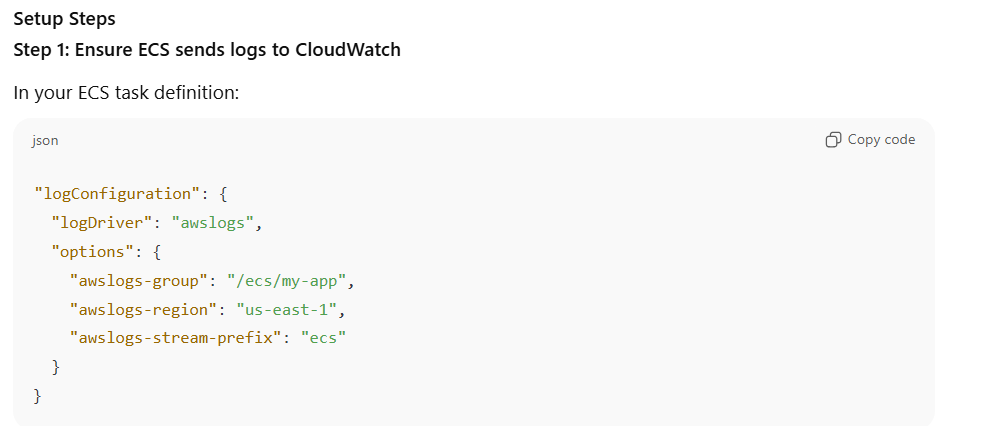
You have **three main approaches** depending on your environment 👇

**🅰️ Option 1 — Using Kinesis Firehose → Splunk HEC (Recommended)**

✅ **Best for production** — scalable, reliable, managed buffering.

**How it works**

1. ECS logs go to CloudWatch log groups.
2. Create a **Kinesis Data Firehose** delivery stream with **destination = Splunk**.
3. Configure **CloudWatch Logs subscription filter** to stream data to Firehose.
4. Firehose delivers logs to **Splunk HEC endpoint**.



**Step 2: Create a Splunk HTTP Event Collector (HEC)**

* Go to **Splunk > Settings > Data Inputs > HTTP Event Collector**
* Create a new token
* Example endpoint:  
  https://splunk.company.com:8088/services/collector
* Note down:
  + HEC token
  + HEC endpoint URL

**Step 3: Create a Kinesis Firehose Delivery Stream**

In AWS Console:

* Destination: **Splunk**
* Splunk endpoint: your HEC URL
* HEC token: from previous step
* Backup: **S3** (recommended)
* Choose **compression, buffering, and retry** options as needed

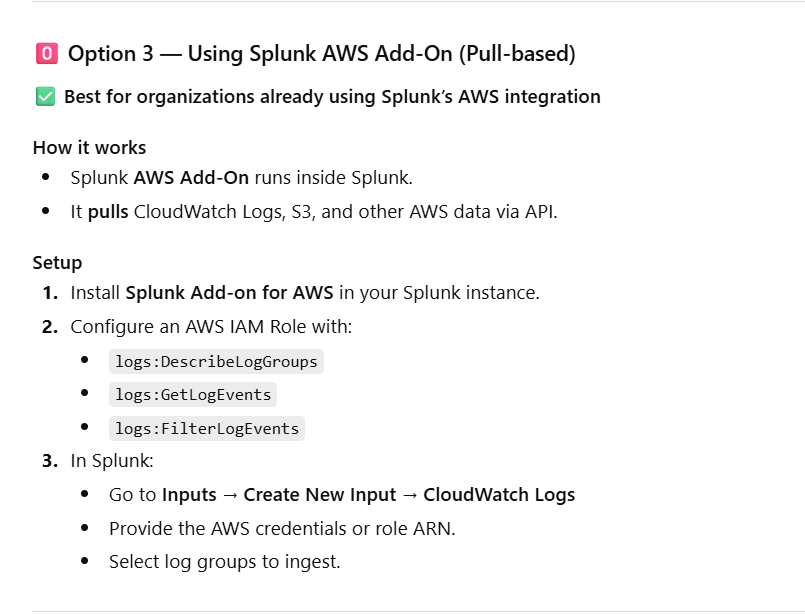


**🅱️ Option 2 — Using AWS Lambda Subscription Filter (Lightweight)**

✅ **Best for smaller setups or testing**

**How it works**

1. ECS → CloudWatch Logs
2. CloudWatch → triggers a **Lambda function [Add a CloudWatch Logs subscription filter to trigger this Lambda.]**
3. Lambda → sends data to Splunk HEC endpoint via HTTPS POST



1. How cloudwatch metrics or Ec2 custom data metrics are sent to Splunk observability ?

**Option 1: Native AWS Integration in Splunk Observability**

✅ **Recommended & Easiest Method**

Splunk Observability provides a **native integration** with AWS CloudWatch using **AWS CloudWatch Metric Streams + Kinesis Data Firehose**.  
This integration continuously streams metrics in near real-time to Splunk Observability.

AWS CloudWatch → CloudWatch Metric Stream → Kinesis Firehose → Splunk Observability Cloud

**Steps**

1. **Enable AWS Integration in Splunk Observability:**
   * Go to **Settings → Data Setup → AWS Integration** in Splunk Observability.
   * Choose **Metric Streams** option (preferred over polling).
2. **CloudFormation Template Setup:**
   * Splunk provides a CloudFormation template that sets up:
     + CloudWatch Metric Stream
     + Kinesis Data Firehose delivery stream
     + IAM roles & permissions
3. **Delivery to Splunk:**
   * The Firehose stream sends metrics directly to:

https://ingest.<realm>.signalfx.com/v1/cloudwatch-metrics

**Option 2: Using Splunk OpenTelemetry Collector (Agent-based)**

This method uses an **agent (collector)** running on EC2 instances to capture and forward metrics/logs directly to Splunk Observability.

EC2 Instance → Splunk OpenTelemetry Collector → Splunk Observability (Metrics + Logs)

**Option 3: Using Splunk Add-on for AWS (for Splunk Enterprise/Cloud, not Observability)**

If your target is **Splunk Enterprise / Splunk Cloud** (not Observability), you can use:

* **Splunk Add-on for AWS**
* It polls CloudWatch Metrics and Logs using AWS APIs
* Less efficient than Metric Streams (polling-based)
* Suitable for traditional Splunk indexers

**DEVOPS [ GENERIC and LEADERSHIP BASED]**

1. What is Devops?

DevOps is not a specific tool or technology — it’s a **mindset, culture, and set of practices** aimed at breaking down the silos between software development and IT operations.  
Normally , developers focused on building features, while operations focused on stability and uptime. DevOps bridges this gap by encouraging **shared responsibility**, **continuous feedback**, and **automation** across all stages of software delivery.

In a DevOps model:

* **Developers and Ops engineers work together** right from the design phase.
* **Automation** is heavily used for building, testing, deployment, and infrastructure management or any kind of repetitive tasks.
* **Continuous Integration (CI)** and **Continuous Deployment (CD)** pipelines ensure code changes are build, tested and deployed quickly. Small, frequent, reliable releases.
* **Monitoring and feedback loops** help teams learn from production and improve continuously. Use tools like Prometheus, CloudWatch, or Splunk to measure performance and get insights.

“DevOps is a cultural and technical approach that unifies development and operations teams to deliver software faster and more reliably. It focuses on collaboration, automation, and continuous delivery. By using practices like CI/CD, infrastructure as code, and continuous monitoring, DevOps enables organizations to respond to business needs quickly while maintaining high quality and stability. In short, DevOps transforms software delivery into a continuous, automated, and feedback-driven process.”

1. **What are the key principles of designing a scalable CI/CD pipeline?**

Designing such a scalabe pipeline requires focusing on **architecture, automation, modularity, observability, and security**.

**⚙️ 1. Modular and Reusable Architecture**

* **Design pipelines as modular building blocks** — separate stages for build, test, deploy, and monitoring.
* Each stage should be **independent and reusable** across projects or microservices.
* Use **pipeline-as-code** (e.g., GitHub Actions YAML, Jenkinsfile, GitLab CI YAML) for version control and consistency.

**Example:**  
Instead of one monolithic Jenkins pipeline, create reusable shared libraries or templates for build/test/deploy stages.

**🔁 2. Parallelism and Concurrency**

* Allow **parallel execution** of tests or builds to reduce overall pipeline time.
* Use **distributed runners/agents** or **auto-scaling build infrastructure** (like GitHub Actions runners, Jenkins agents on Kubernetes).
* Adopt **micro-pipelines per microservice** instead of one giant pipeline.

**Example:**  
Run unit tests, linting, and security scans in parallel instead of sequentially

**🧩 3. Environment Consistency via IaC & Containers**

* Use **Infrastructure as Code (IaC)** tools like Terraform or CloudFormation to ensure consistent infrastructure provisioning.
* Package builds and dependencies into **containers (Docker)** so they run identically across environments.
* Avoid “it works on my machine” problems.

**Example:**  
A Dockerized build environment ensures the same Python or Node version across dev, QA, and prod.

**🚀 4. Continuous Testing & Quality Gates**

* Integrate automated **unit, integration, and security tests** early in the pipeline.
* Use **quality gates** (e.g., SonarQube, Snyk, Checkov) to block poor-quality code or vulnerabilities before deployment.
* Make testing **fast, automated, and parallelizable**.

**Example:**  
Fail the pipeline if code coverage drops below threshold or if dependency scans find critical vulnerabilities.

**🧠 5. Scalable Infrastructure for CI/CD Runners**

* Implement **auto-scaling runners** or **build agents** using cloud-native solutions (EKS, ECS, Kubernetes).
* Use **ephemeral agents** that spin up per build and terminate after completion — ensuring scalability and isolation.

**Example:**  
A Kubernetes-based Jenkins cluster that automatically provisions pods for each build.

**🔍 6. Observability and Feedback Loops**

* Build **visibility** into the pipeline: metrics, logs, and dashboards (e.g., Grafana, Splunk, Prometheus).
* Monitor **pipeline duration, failure rates, test coverage**, and **deployment success**.
* Provide **real-time feedback** to developers via Slack, Teams, or email integrations.

**Example:**  
Alert when deployment fails or test coverage drops below a defined threshold.

**🛡️ 7. Security and Compliance (DevSecOps)**

* Embed **security scans** (SAST, DAST, dependency scanning, IaC scanning) directly into the pipeline.
* Use **secrets management systems** like AWS Secrets Manager, Vault, or GitHub Encrypted Secrets.
* Ensure **role-based access control (RBAC)** for pipelines and deployment credentials.

**Example:**  
Before deploying to production, automatically scan Docker images with Trivy or Aqua.

**🔄 8. Progressive Delivery & Rollback Mechanisms**

* Support **progressive deployment strategies** like Blue-Green, Canary, or Feature Flags.
* Enable **automated rollback** in case of failure (using health checks or monitoring alerts).

**Example:**  
If a new version increases error rate, pipeline triggers rollback to the previous stable build automatically.

1. **How do you ensure cost efficiency and resource governance in AWS or Azure?**

✅ **How to Ensure Cost Efficiency and Resource Governance in AWS**

**🎯 1. Right-Sizing and Auto-Scaling**

* Continuously monitor resource utilization (CPU, memory, IOPS, network).
* **Right-size** instances, storage, and databases — downgrade or terminate underused resources.
* Implement **auto-scaling** groups or **Kubernetes Horizontal Pod Autoscalers (HPA)** to match workload demand dynamically.

**Example:**  
Use **AWS Compute Optimizer** or **Azure Advisor** recommendations to resize EC2, RDS, or VM instances based on usage patterns.

**💰 2. Use Cost-Effective Pricing Models**

* Choose the right **pricing plan** for workloads:
  + **Reserved Instances (RIs)** or **Savings Plans** for steady-state workloads.
  + **Spot Instances / Spot VMs** for non-critical or batch jobs.
  + **Serverless compute** (AWS Lambda, Azure Functions) for event-driven workloads.
* Prefer **S3 lifecycle policies** or **Azure Blob lifecycle management** to automatically move infrequently accessed data to cheaper storage classes.

**Example:**  
Move old logs to **S3 Glacier** or **Azure Archive Storage**.

**🧠 3. Implement Resource Tagging and Cost Allocation**

* Apply **mandatory tagging policies** for cost visibility and accountability (e.g., Environment, Owner, Project, CostCenter).
* Use **AWS Tag Editor** or **Azure Resource Graph** to enforce and audit tags.
* Enable **cost allocation reports** by tags to attribute spend to specific teams or projects.

**🧾 4. Governance Through Policies and Guardrails**

* Use **Service Control Policies (SCPs)** in AWS Organizations or **Azure Policy** to enforce guardrails:
  + Restrict creation of resources in non-approved regions.
  + Enforce encryption, tagging, and size limits.
  + Prevent use of expensive instance types unless approved.
* Implement **resource quotas and budgets** to prevent accidental overspending.

**Example:**  
Disallow provisioning of GPU instances unless in a specific account or resource group.

**🧱 6. Infrastructure as Code (IaC) for Consistency and Lifecycle Management**

* Use **Terraform**, **CloudFormation**, or **Bicep** to define and version infrastructure.
* Apply IaC to **destroy unused resources automatically** in dev/test environments.

**Example:**  
Automatically tear down QA environments after 12 hours using pipeline automation.

**🧩 8. Optimize Storage and Data Transfer**

* Use **S3 Intelligent-Tiering** or **Azure Blob Cool/Archive tiers** for cost savings.
* Compress, archive, or delete obsolete data regularly.
* Use **VPC endpoints / Private Links** to reduce data transfer costs between services.

**Example:**  
Compress build artifacts before uploading to S3 or Azure Storage.

**🔄 9. Schedule and Automate Resource Shutdown**

* Automatically shut down non-production workloads during off-hours (e.g., nights/weekends).
* Use **AWS Instance Scheduler**, **Azure Automation Runbooks**, or **EventBridge / Logic Apps** for scheduling.

**Example:**  
Turn off dev EC2 instances after 7 PM and restart at 8 AM.

1. **How do you mentor DevOps engineers?**

**🎯 1. Understand Each Engineer’s Strengths and Goals**

Mentorship starts with understanding the individual.

* I begin by identifying each engineer’s current skill level, career goals, and learning preferences (hands-on, reading, pair programming, etc.).
* I discuss what areas they want to grow in — for example: CI/CD, cloud infrastructure, observability, or security.
* Then I set personalized growth plans that align both with team goals and individual aspirations.

Example:  
If someone is strong in scripting but weak in cloud architecture, I assign them to design a Terraform module under my guidance.

**🧭 2. Lead by Example (Hands-on Mentorship)**

DevOps is best taught through practice and collaboration.

* I pair-program or co-own tasks during early learning phases — e.g., writing a new CI/CD pipeline together or debugging an IaC deployment.
* I narrate my thought process — explaining why I’m making certain architectural or tooling choices, not just how.
* I make sure every engineer gets hands-on exposure to real-world issues — deployments, incidents, and optimization.

Example:  
While implementing a GitHub Actions pipeline, I’ll show them how to modularize workflows for reusability and security.

**🧩 3. Build a Learning Culture**

DevOps evolves fast, so continuous learning must be part of the team DNA.

* I organize weekly knowledge-sharing sessions (e.g., “DevOps Fridays”) where team members demo new tools, pipelines, or postmortems.
* Encourage certifications (AWS, Azure, Terraform) and provide learning resources like Pluralsight, A Cloud Guru, or internal wikis.
* Celebrate learning — not just delivery — to foster curiosity.

Example:  
After a failed deployment, we hold a blameless postmortem, document the lessons learned, and use it as a teaching moment.

**🧠 4. Encourage Ownership and Autonomy**

**True mentorship empowers, not micromanages.**

* I assign end-to-end ownership of small projects — for example, designing a monitoring dashboard or building a reusable CI/CD template.
* I’m available for guidance but let them make design decisions and mistakes safely.
* This helps them build confidence and accountability.

Example:  
An engineer leads a migration from Jenkins to GitHub Actions while I guide them on best practices and review architecture decisions.

**📊 5. Create Structured Feedback Loops**

Constructive feedback is vital for growth.

* I conduct 1:1 feedback sessions focusing on strengths, areas of improvement, and measurable goals.
* I use real project examples to provide actionable feedback.
* I also encourage peer feedback so team members learn from each other.

Example:  
Instead of saying “your Terraform code needs improvement,” I’d say “try using Terraform modules for reusability — it’ll reduce code duplication by 40%.”

**🤝 6. Promote DevOps Culture and Collaboration**

Mentorship isn’t just technical — it’s cultural.

* I reinforce **DevOps values**: collaboration, automation, transparency, and shared responsibility between Dev, Ops, and QA.
* I make sure DevOps engineers understand **why culture matters** as much as tools.
* I encourage cross-functional collaboration — e.g., pairing with developers or SREs on deployment design.

**Example:**  
I’ll have DevOps engineers join sprint planning to understand the “why” behind product features — not just build pipelines in isolation.

1. **What’s the biggest DevOps challenge you faced?**

**✅ Challenge Context**

You have:

* ~100 microservices
* Code in **GitLab**, pipelines in **TFS + Octopus** (legacy)
* Each service has complex build/release processes
* Octopus currently handles deployments and environment tracking
* The goal is to **migrate all CI/CD workflows to GitHub Actions** (and possibly GitHub Environments)

This is a **large-scale CI/CD modernization program**, not a simple migration — so it must be handled **phased, modular, and governed**.

**⚙️ Step-by-Step Strategy to Handle This Efficiently**

**1. Discovery and Assessment Phase**

You can’t migrate what you don’t fully understand.

* **Inventory all pipelines** in TFS and Octopus:
  + Build triggers, dependencies, variables, deployment targets, and approval steps.
* Classify pipelines based on **complexity and criticality** (e.g., high/medium/low).
* Identify **common patterns** — e.g., build, test, deploy templates reused across services.
* Identify what’s **Octopus-specific** (release versioning, variable substitution, step templates) — these will need redesign.

**Deliverable:**  
A migration assessment report with:

* Pipeline inventory
* Common reusable components
* Service dependencies
* Migration difficulty estimate per service

**2. Define the Target Architecture**

Design before you migrate — otherwise chaos multiplies.

* Define a **standardized CI/CD pipeline architecture** in GitHub Actions:
  + Use **reusable workflows** for build/test/deploy.
  + Use **GitHub Environments** for dev, QA, stage, and prod with **approvals and secrets**.
  + Implement **OIDC-based authentication** with cloud providers (AWS/Azure) to eliminate credential storage.
* Choose a **deployment mechanism**:
  + GitHub Actions → directly deploy (if your infra supports it).
  + OR integrate GitHub Actions with Octopus (temporarily) during transition.
* Decide on **artifact storage** — GitHub Packages, AWS S3, or Azure Artifacts.
* Integrate **SonarQube, security scanning, and observability** natively in workflows.

**Deliverable:**  
A **GitHub Actions reference architecture** + governance model.

**3. Build a Reusable CI/CD Framework**

Don’t rewrite 100 pipelines — **standardize and reuse**.

* Create **modular workflow templates** (build, test, deploy) under a centralized repo, e.g., .github/workflows-templates.
* Each microservice just **inherits and configures** these templates using inputs.
* Define **naming conventions** and **repository structure** to maintain uniformity.
* Integrate:
  + **SonarQube** for quality gates
  + **Snyk/Trivy** for security scans
  + **Slack/MS Teams** notifications
  + **Environment approvals** for controlled release

A screenshot of a computer

AI-generated content may be incorrect.

**4. Phased Migration Approach**

Migrate incrementally — not in a “big bang.”

1. **Pilot phase (5–10 microservices)**
   * Choose non-critical, low-dependency services.
   * Test new GitHub Actions templates end-to-end.
   * Validate performance, secrets, approvals, and rollback.
2. **Parallel run phase**
   * Run old TFS/Octopus and GitHub Actions in parallel.
   * Compare outcomes and stability.
3. **Progressive migration**
   * Migrate services batch-wise (e.g., per domain or business capability).
   * Automate migration scripts if possible (using GitHub CLI or APIs).
4. **Decommissioning phase**
   * Once stable, shut down old pipelines in TFS/Octopus systematically.
   * Archive configurations for audit.