Smart Manufacturing Data Hub (SMDH)

**Architecture Design Options**

Comprehensive Analysis of Four Architecture Approaches

|  |  |
| --- | --- |
| Document Version | 2.0 FINAL |
| Date | October 2025 |
| Status | For Review |
| Classification | Internal Use |
| Author | AI Applied Technical Team |
| Intended Audience | Technical Architects and Business Stakeholders |

# 1. Executive Summary

This document presents a comprehensive analysis of four distinct architectural approaches for the Smart Manufacturing Data Hub (SMDH) platform. The SMDH is designed as a cloud-native, multi-tenant Internet of Things (IoT) platform that empowers small and medium-sized manufacturing enterprises with real-time visibility into their operations.

## 1.1 Document Purpose

The purpose of this document is to provide both technical architects and business stakeholders with a clear understanding of four viable architecture options for the SMDH platform. Each option has been analysed in detail, considering factors such as:

* Cost implications and total cost of ownership
* Technical complexity and required expertise
* Performance characteristics and latency requirements
* Scalability and multi-tenancy capabilities
* Implementation timelines and resource requirements
* Operational considerations and maintenance overhead

## 1.2 System Context

The SMDH platform serves manufacturing companies by collecting, processing, and visualising data from various industrial sensors and equipment. The system must support:

* Multiple manufacturing companies (tenants) using a shared platform infrastructure
* Processing 2.6 to 3.9 million data points per day from diverse sensor types
* Real-time monitoring with latency under 1 second for critical alerts
* Self-service company onboarding and device registration
* Automated dashboard provisioning and data visualisation
* Secure data isolation between different companies
* Support for 20-40 concurrent users across 30-40 companies in Year 1

## 1.3 The Four Architecture Options

This document evaluates four distinct architectural approaches, each with unique strengths and trade-offs:

**Option A: Flink-Based AWS-Heavy Architecture**

This option leverages Apache Flink for advanced stream processing, providing sub-second latency and sophisticated data transformation capabilities. It represents the most powerful but also most complex approach, suitable for scenarios requiring advanced real-time processing.  
Monthly cost: £2,500-4,200 (30 tenants) | Complexity: Very High | Timeline: 24 weeks

**Option B: Snowflake-Leveraged Architecture**

This option centralises data storage and processing in Snowflake, a cloud data platform. It offers the simplest architecture with SQL-first development, making it ideal for teams with traditional database skills. This is the recommended option for most scenarios.  
Monthly cost: £2,170-3,450 (30 tenants) | Complexity: Low | Timeline: 20 weeks

**Option C: AWS IoT SiteWise-Based Architecture**

This option uses AWS IoT SiteWise, a managed service specifically designed for industrial IoT. It provides built-in asset modelling and time-series capabilities but requires custom development for multi-tenant isolation and dashboards.  
Monthly cost: £6,334 (30 tenants) | Complexity: Medium-High | Timeline: 28 weeks

**Option D: Pure AWS-Native Architecture (Timestream + Grafana)**

This option uses only AWS services, combining Amazon Timestream for time-series data storage with Grafana for dashboards. It provides a fully AWS-native solution with excellent multi-tenancy support and industry-standard visualisation.  
Monthly cost: £3,965 (30 tenants) | Complexity: Medium | Timeline: 22 weeks

## 1.4 Key Finding: All Options Are Viable

An important finding from our analysis is that all four options meet the budget constraint of £200-300 per tenant per month. A previous cost calculation error for Option C has been corrected, reducing its estimated monthly cost by 70% (from £21,150 to £6,334 for 30 tenants). This means the decision should be based on factors other than basic affordability, such as technical requirements, team capabilities, and strategic preferences.

## 1.5 How to Use This Document

This document is structured to support both detailed technical review and high-level business decision-making:

* Technical Architects should review Sections 4-7 for detailed component descriptions and data flows
* Business Stakeholders may focus on Section 1 (this summary) and Section 8 (comparisons)
* Decision-makers should review Section 9 for specific recommendations and decision frameworks
* All readers can refer to Section 10 (Glossary) for explanations of technical terms

# 2. Document Purpose and Audience

## 2.1 Intended Audience

This document has been prepared for a diverse audience with varying levels of technical expertise:

|  |  |
| --- | --- |
| Audience | What to Focus On |
| Senior Leadership | Executive Summary (Section 1), Recommendations (Section 9), Cost Comparison (Section 8.1) |
| Product Managers | Architecture overviews (Sections 4-7), Decision Framework (Section 8.5) |
| Technical Architects | Detailed component descriptions (Sections 4-7), Data flows, Technical comparisons |
| Engineering Teams | Component descriptions, Technology stacks, Implementation considerations |

## 2.2 Reading Guide for Non-Technical Readers

For readers without extensive cloud computing or AWS knowledge, this document includes:

* Plain English explanations of technical concepts where possible
* Visual architecture diagrams showing how components connect
* Analogies to familiar concepts (e.g., "like a sorting office for data")
* A comprehensive glossary defining all technical terms
* Summary boxes highlighting key points in each section
* Cost and complexity indicators for quick comparison

Whenever you encounter an unfamiliar term, please refer to the Glossary in Section 10. Terms defined in the glossary appear in this format on first use: Internet of Things (IoT).

# 3. System Overview and Requirements

## 3.1 What is the Smart Manufacturing Data Hub?

The Smart Manufacturing Data Hub (SMDH) is a cloud-based software platform that helps manufacturing companies monitor and analyse their operations in real-time. Think of it as a central "command centre" where data from factory equipment, environmental sensors, and tracking systems all flows together to provide insights.

### How It Works (Simplified)

1. Manufacturing companies register on the platform and create accounts

2. They register their factory sites and the equipment/sensors they want to monitor

3. Sensors send data continuously (e.g., "Machine A is running", "Temperature is 22°C")

4. The platform receives, validates, and stores this data securely

5. Dashboards automatically update to show real-time status and historical trends

6. Alerts notify users when something needs attention (e.g., machine breakdown, poor air quality)

## 3.2 Key System Requirements

The architecture options must all satisfy the following essential requirements:

### Data Volume and Performance

|  |  |
| --- | --- |
| Requirement | Target |
| Daily data points processed | 2.6 - 3.9 million rows per day |
| Real-time monitoring latency | Under 1 second for critical data |
| Alert notification time | Under 10 seconds from trigger event |
| System availability | 99.9% uptime (less than 9 hours downtime per year) |

### Multi-Tenancy (Shared Platform)

The platform must support multiple manufacturing companies simultaneously whilst ensuring complete data isolation. This means:

* Company A cannot see Company B's data under any circumstances
* Each company has their own dashboards, users, and configurations
* The platform must efficiently share resources (like servers) whilst maintaining security
* Support for 30-40 companies initially, scaling to 100+ companies over time

### Budget Constraints

The target cost is £200-300 per company (tenant) per month. For 30 companies, this means:

**Target monthly cost: £6,000 - £9,000**

This budget must cover all cloud infrastructure, data storage, processing, and dashboard services. Good news: all four architecture options fall within or close to this budget.

## 3.3 Use Cases Supported

The platform must support diverse manufacturing monitoring scenarios:

**Machine Utilisation Monitoring**

Tracks how efficiently manufacturing equipment is being used. Sensors collect data every second about whether machines are running, idle, or experiencing downtime. This helps identify bottlenecks and improve productivity.

**Air Quality Management**

Monitors environmental conditions in manufacturing facilities. Sensors measure CO₂, volatile organic compounds (VOCs), particulate matter, temperature, and humidity. Ensures compliance with health and safety regulations and worker wellbeing.

**Energy Monitoring**

Tracks electrical consumption of equipment and facilities. Monitors voltage, current, power factor, and kilowatt-hour usage. Helps identify energy waste and reduce costs.

**Job Location Tracking**

Uses RFID tags or barcodes to track work-in-progress through the factory. Provides real-time visibility of where jobs are located and helps prevent lost or delayed orders.

# 4. Architecture Option A: Flink-Based AWS-Heavy Architecture

## 4.1 Overview

|  |  |
| --- | --- |
| Monthly Cost (30 tenants) | £2,500 - 4,200 |
| Cost Per Tenant | £83 - 140 |
| Complexity Level | Very High (15+ services) |
| Implementation Timeline | 24 weeks |
| Team Size Required | 3-4 engineers |
| Best Suited For | Scenarios requiring sub-second latency and advanced ML |

### What is This Architecture?

Option A represents the most technically sophisticated approach, using Apache Flink—an advanced stream processing framework—to handle data in real-time. Imagine a highly efficient assembly line where data flows through multiple specialised stations, each performing specific transformations and checks at extremely high speed.

This architecture is called "AWS-Heavy" because it uses many different Amazon Web Services (AWS) components, each specialised for a particular task. Whilst this provides maximum flexibility and power, it also creates significant complexity in managing all these interconnected services.

## 4.2 Architecture Diagram

*[INSERT: SMDH-Option-A-Flink-Architecture.drawio diagram here]*

The diagram above shows how data flows from sensors through multiple processing stages before reaching users. Each box represents a different AWS service or component.

## 4.3 Detailed Component Description

This section explains each major component of the architecture in plain English, with the technical details that architects need to evaluate the design.

### Layer 1: Data Sources and Ingestion

**AWS IoT Core (MQTT Broker)**

What it does: Acts as the "front door" for all sensor data entering the system. Sensors connect to IoT Core using MQTT (a lightweight messaging protocol designed for IoT devices) and publish their data.  
  
Why it matters: Handles device authentication using X.509 certificates, ensuring only authorised sensors can send data. Supports millions of concurrent connections and can receive data from 30-45 sensors per manufacturing site.  
  
Technical details: Provides Quality of Service (QoS) levels 0 and 1 for message delivery. Includes device shadows for storing last-known state and offline capabilities.

**IoT Rules Engine**

What it does: Acts like a "sorting office" that examines each incoming message and routes it to the appropriate destination based on rules.  
  
Why it matters: Enables intelligent message routing without writing custom code. Can enrich messages with additional metadata (like tenant IDs) and filter out invalid data.  
  
Technical details: SQL-based rules engine. Can route to multiple destinations simultaneously. Supports transformations using built-in functions.

**Kinesis Data Firehose**

What it does: Buffers incoming data and groups it into larger batches before passing it onward. Think of it like a truck that waits until it's full before making a delivery, rather than making individual trips for each package.  
  
Why it matters: Improves efficiency by batching data. Handles automatic compression and format conversion. Provides built-in retry logic if downstream systems are temporarily unavailable.  
  
Technical details: Buffers data up to 10MB or 60 seconds (whichever comes first). Supports Parquet, ORC, and JSON formats. Can invoke Lambda functions for transformation.

**Lambda Transformation Functions**

What it does: Small programs that validate, clean, and transform individual data records. Examples include converting temperature from Fahrenheit to Celsius, validating sensor ranges, or adding calculated fields.  
  
Why it matters: Ensures data quality before storage. Enables custom business logic without managing servers. Highly scalable—automatically handles processing for one sensor or one thousand.  
  
Technical details: Serverless functions written in Python or Node.js. Invoked synchronously during data flow. Timeout: 5 minutes maximum. Can access other AWS services like DynamoDB for lookups.

**S3 Staging Storage**

What it does: Stores data in structured folders (like an organised filing cabinet) in Parquet format (an efficient columnar storage format).  
  
Why it matters: Provides durable, low-cost storage. Acts as a "data lake" where historical data can be kept indefinitely. Enables recovery if downstream processes fail.  
  
Technical details: S3 Standard storage with intelligent-tiering for cost optimisation. Partitioned by tenant\_id/device\_id/year/month/day for efficient queries. Lifecycle policies move data to Glacier after 90 days for long-term archival.

### Layer 2: Stream Processing (The Heart of Option A)

**Apache Flink on Amazon EMR**

What it does: This is the defining feature of Option A. Apache Flink is an advanced stream processing framework that processes data in real-time with sub-second latency. It performs complex operations like windowing (grouping data into time periods), pattern detection (e.g., "detect if a machine cycles on and off three times in 5 minutes"), and sophisticated aggregations.  
  
Why it matters: Enables the most advanced real-time analytics capabilities. Can handle complex event processing that simpler systems cannot. Provides exactly-once processing guarantees, meaning no data is lost or processed twice.  
  
Technical details: Runs on Amazon EMR (Elastic MapReduce) cluster with 3-5 nodes. Uses RocksDB for state management with checkpoints to S3 every 5 minutes. Supports SQL queries (Flink SQL) and custom Java/Scala/Python code. Autoscaling based on data velocity.  
  
What makes it complex: Requires expertise in Java or Scala programming. Cluster management and tuning is non-trivial. Debugging distributed stream processing applications requires specialised skills. State management and checkpointing must be carefully configured to prevent data loss.

**RocksDB State Backend**

What it does: Stores the "working memory" of Flink jobs. For example, if you're calculating a 5-minute average, it needs to remember the last 5 minutes of data.  
  
Why it matters: Enables stateful stream processing. Provides fault tolerance—if a server crashes, processing can resume from the last checkpoint without data loss.  
  
Technical details: Embedded database within each Flink task. Checkpoints to S3 for durability. Incremental checkpointing reduces overhead. Recovery time typically under 2 minutes.

### Layer 3: Data Platform (Snowflake)

**Snowpipe Auto-Ingestion**

What it does: Automatically detects when new data files arrive in S3 and loads them into Snowflake (the data warehouse) within seconds.  
  
Why it matters: No manual intervention required. Enables near-real-time analytics on batch-loaded data. Automatically scales based on workload.  
  
Technical details: Event-driven using S3 notifications → SQS → Snowpipe. Loads micro-batches continuously. Charges based on warehouse compute time (typically seconds per file).

**Snowflake Tables and Views**

What it does: Stores data in structured tables (like Excel spreadsheets but for billions of rows). Views provide pre-calculated aggregations and enforce row-level security for multi-tenancy.  
  
Why it matters: SQL interface familiar to most analysts. Native support for VARIANT (JSON) data types. Row-Level Security (RLS) ensures tenants only see their own data.  
  
Technical details: Tables partitioned by tenant\_id. VARIANT columns for flexible schema. Clustering on timestamp columns for query performance. Materialised views for common aggregations.

**Snowflake Streams and Tasks**

What it does: Streams track changes to tables (like a change log), and Tasks are scheduled SQL statements that run periodically to process those changes.  
  
Why it matters: Enables incremental processing—only new or changed data is processed, reducing costs. Provides a native ETL (Extract, Transform, Load) framework within Snowflake.  
  
Technical details: Change Data Capture (CDC) using streams. DAG-based task orchestration. Serverless compute—only pay when tasks run.

### Layer 4: Application and Analytics

**API Gateway and Lambda API Handlers**

What it does: Provides REST APIs that applications call to retrieve data. API Gateway acts as the "front desk," routing requests to Lambda functions that execute the business logic.  
  
Why it matters: Enables mobile and web applications to access data securely. Enforces authentication and rate limiting. Scales automatically with demand.  
  
Technical details: API Gateway with JWT token authentication. Lambda functions query Snowflake or DynamoDB. Tenant isolation enforced via Cognito user attributes.

**Amazon QuickSight (Dashboards)**

What it does: Creates interactive dashboards and visualisations. Users can see charts, graphs, and tables showing their manufacturing data.  
  
Why it matters: Embedded dashboards can be integrated into custom applications. Pay-per-session pricing model is cost-effective for occasional users. Native connector to Snowflake.  
  
Technical details: Embedded dashboards with Row-Level Security. Supports scheduled reports and email alerts. SPICE in-memory cache for fast queries.

**Amazon SageMaker (Machine Learning)**

What it does: Enables building, training, and deploying custom machine learning models. Examples include predictive maintenance (predicting when equipment will fail) or anomaly detection.  
  
Why it matters: Provides unlimited flexibility for advanced analytics. Supports all major ML frameworks (TensorFlow, PyTorch, scikit-learn). GPU-accelerated training for deep learning.  
  
Technical details: SageMaker training jobs on ml.p3 instances. Model hosting on auto-scaling endpoints. Feature store for reusable ML features.

## 4.4 Data Flow and Processing

### Real-Time Data Flow (Sub-Second Path)

For critical data requiring immediate processing (e.g., machine failure alerts):

1. Sensor publishes MQTT message to IoT Core (latency: <100ms)

2. IoT Rules Engine routes to Kinesis Data Streams (latency: <200ms)

3. Flink consumes from Kinesis and processes with sub-second windowing (latency: <500ms)

4. Flink detects alert condition and publishes to SNS/SES (latency: <200ms)

5. User receives email/SMS notification (total latency: <2 seconds)

### Batch Processing Flow (Historical Analytics)

For historical reporting and trend analysis (less time-sensitive):

1. Kinesis Firehose buffers data in 60-second batches

2. Lambda function transforms and validates each record

3. Data written to S3 in Parquet format (partitioned by date and tenant)

4. Snowpipe detects new S3 files and loads into Snowflake (<5 minutes)

5. Snowflake Tasks run hourly to calculate aggregations

6. QuickSight dashboards refresh automatically showing updated metrics

## 4.5 Strengths and Limitations

### Strengths

**Sub-Second Latency:** Achieves <1 second end-to-end latency for critical data. Essential if regulatory requirements mandate immediate alerting.

**Advanced Stream Processing:** Flink provides capabilities that simpler systems lack: complex event processing, pattern matching across multiple streams, sophisticated windowing strategies.

**Unlimited Flexibility:** Can implement any custom logic in Java/Python/Scala. No limitations on data transformations or ML algorithms.

**Best-of-Breed Components:** Each component is industry-leading in its category. Flink for streaming, Snowflake for analytics, SageMaker for ML.

**Proven at Scale:** Flink powers stream processing at companies like Netflix, Uber, and Alibaba. Known to handle billions of events per day.

### Limitations and Challenges

**Very High Complexity:** Managing 15+ interconnected services requires deep expertise. Each component has its own configuration, monitoring, and troubleshooting requirements.

**Steep Learning Curve:** Flink expertise is rare and expensive. Teams need proficiency in multiple languages (Java for Flink, Python for Lambda, SQL for Snowflake).

**Longest Implementation Timeline:** Estimated 24 weeks to production. Building and testing distributed streaming applications takes time.

**Operational Overhead:** EMR cluster management, Flink job tuning, checkpoint monitoring, state recovery testing all require ongoing attention.

**Highest Debugging Complexity:** Troubleshooting issues across distributed Flink jobs, multiple data pipelines, and async processing is challenging.

**Resource Intensive:** Requires 3-4 senior engineers with diverse skill sets: Flink specialists, AWS architects, data engineers.

### When to Choose Option A

Select this architecture ONLY if:

* Sub-second latency is legally or regulatorily required (not just "nice to have")
* Complex event processing (CEP) is essential for the business case
* Team already has Flink expertise or budget to hire specialists
* Willing to accept 24-week timeline and higher operational complexity
* Budget supports £2,500-4,200/month for infrastructure alone

**Recommendation:** For most manufacturing IoT scenarios, the complexity and cost of Option A are not justified. Consider Option B (Snowflake) for simplicity or Option D (AWS-native) for fast alerting without Flink complexity.

# 5. Architecture Option B: Snowflake-Leveraged Architecture

## 5.1 Overview

|  |  |
| --- | --- |
| Monthly Cost (30 tenants) | £2,170 - 3,450 (CHEAPEST) |
| Cost Per Tenant | £72 - 115 |
| Complexity Level | Low (5 core services) |
| Implementation Timeline | 20 weeks (FASTEST) |
| Team Size Required | 2-3 engineers (SQL skills) |
| Best Suited For | Most scenarios - Recommended default choice |

### What is This Architecture?

Option B represents the simplest and most cost-effective approach, centralising nearly all data processing and storage within Snowflake—a cloud data platform optimised for analytics. Imagine a "one-stop shop" where data arrives, gets processed, stored, and analysed all in one unified system, rather than moving between multiple separate services.

This architecture is called "Snowflake-Leveraged" because it maximises the use of Snowflake's built-in capabilities (Streams for change tracking, Tasks for scheduling, native SQL for transformations), minimising the need for external processing engines like Flink or complex Lambda functions.

**🏆 RECOMMENDED OPTION FOR MOST SCENARIOS**

## 5.2 Architecture Diagram

*[INSERT: SMDH-Option-B-Snowflake-Architecture.drawio diagram here]*

The diagram above shows a much simpler data flow compared to Option A. Notice how data reaches Snowflake quickly and most processing happens within a single platform.

## 5.3 Detailed Component Description

### Layer 1: Simplified Ingestion Path

**AWS IoT Core (Same as Option A)**

Function identical to Option A—receives MQTT messages from sensors with X.509 certificate authentication. The difference in Option B is what happens next: data flows directly to Snowflake with minimal intermediate processing.

**Kinesis Data Streams (Minimal Buffering)**

What it does: Provides a lightweight buffer between IoT Core and Snowflake. Unlike Option A which uses Kinesis Firehose with heavy batching, Option B uses Kinesis Streams for near-real-time delivery.  
  
Why it matters: Reduces latency whilst still providing ordering guarantees and replay capabilities. Simpler configuration than Firehose.  
  
Technical details: 10-20 shards based on throughput. 24-hour retention for replay. Partition key: tenant\_id for tenant-level ordering.

**Snowpipe Streaming**

What it does: Snowflake's streaming ingestion service that continuously loads data from Kinesis directly into Snowflake tables in near-real-time (typically 5-10 seconds).  
  
Why it matters: Eliminates the need for S3 staging, Lambda transformations, and batch loading. Data arrives in Snowflake faster and with fewer moving parts. This is a key simplification compared to Option A.  
  
Technical details: Serverless—no infrastructure to manage. Automatically handles schema evolution. Exactly-once delivery semantics. Uses Snowflake's VARIANT type for flexible JSON ingestion.

### Layer 2: Unified Data Platform (Snowflake)

**Snowflake: The Core of Option B**

In Option B, Snowflake performs the role of multiple services from Option A combined:

• Data Lake (replaces S3)

• Stream Processing (replaces Flink)

• Data Warehouse (same as Option A)

• ETL Engine (replaces Lambda + AWS Glue)

• Query Engine (same as Option A but simpler)

**Raw Tables with VARIANT Columns**

What it does: Stores incoming JSON data in Snowflake's VARIANT type, which preserves the original structure whilst allowing SQL queries on nested fields.  
  
Why it matters: No need to pre-define rigid schemas. If sensors start sending new fields, they're automatically captured. Queries can still extract specific values: SELECT data:temperature::float FROM raw\_data.  
  
Technical details: Tables partitioned by TENANT\_ID and DEVICE\_ID for query performance. Clustering on TIMESTAMP column. Automatic compression reduces storage costs by 75-85%.

**Snowflake Streams (Change Data Capture)**

What it does: Automatically tracks what data has been added, updated, or deleted in tables. Think of it like a "changelog" that remembers what's new since you last looked.  
  
Why it matters: Enables incremental processing—transformations only operate on new data, not the entire table every time. Drastically reduces processing costs and time.  
  
Technical details: Streams use hidden metadata columns (METADATA$ACTION, METADATA$ISUPDATE). Zero-copy—doesn't duplicate data. Multiple consumers can read from the same stream independently.

**Snowflake Tasks (Orchestration)**

What it does: Scheduled SQL statements that run automatically. For example, "Every 5 minutes, calculate hourly aggregations for any new data."  
  
Why it matters: Replaces external orchestration tools like Apache Airflow. Enables building complex data pipelines entirely in SQL. DAG (Directed Acyclic Graph) support for task dependencies.  
  
Technical details: Serverless execution using dedicated warehouses. Cron-like scheduling or stream-triggered execution. Built-in retry logic and alerting on failure.

**Curated Views with Row-Level Security (RLS)**

What it does: Pre-calculated views that automatically filter data based on who's querying. A user from Company A sees only Company A's data; Company B sees only theirs.  
  
Why it matters: Multi-tenancy security enforced at the database level—cannot be bypassed by applications. Simpler than application-level filtering. Snowflake's RLS is mature and battle-tested.  
  
Technical details: Secure views with SESSION\_CONTEXT('CURRENT\_TENANT') filtering. Mapping tables define user→tenant relationships. Policy objects for reusable security logic across multiple tables.

**Snowflake Time-Travel and Zero-Copy Cloning**

What it does: Time-Travel allows querying historical data ("show me this table as it was 2 days ago"). Zero-copy cloning creates instant copies of databases for testing without duplicating storage.  
  
Why it matters: Time-Travel enables easy recovery from mistakes (accidental deletions, bad updates). Cloning enables safe testing of schema changes or ETL modifications.  
  
Technical details: Time-Travel retention configurable 1-90 days (1 day default). Cloning uses copy-on-write (only changes consume additional storage). Fail-safe provides 7 additional days for Snowflake support to recover data.

### Layer 3: Machine Learning in Snowflake

**Snowflake Cortex ML**

What it does: Enables building machine learning models using SQL commands within Snowflake. Example: CREATE MODEL predict\_machine\_failure AS SELECT \* FROM training\_data;  
  
Why it matters: No need to export data to external ML platforms. Analysts with SQL skills can build models without Python expertise. Predictions run as SQL queries.  
  
Technical details: Supports classification, regression, time-series forecasting. AutoML automatically selects best algorithm. Model versioning and lineage tracking built-in.  
  
Limitations: Less flexible than SageMaker (Option A). Cannot use custom algorithms or deep learning frameworks. Good for 80% of ML use cases; advanced scenarios may require SageMaker integration.

### Layer 4: Application and Analytics (Simplified)

**API Gateway + Lambda (Minimal)**

What it does: Same role as Option A but much simpler. Lambda functions primarily just execute SQL queries against Snowflake—no complex transformation logic.  
  
Why it matters: Less custom code means fewer bugs and faster development. Snowflake handles the heavy lifting; Lambda is just a thin API layer.  
  
Technical details: Snowflake Node.js or Python connector. Connection pooling for performance. API responses cached in CloudFront for frequently-accessed data.

**QuickSight or Power BI (User Choice)**

What it does: Dashboarding identical to Option A. Option B offers flexibility: use QuickSight (AWS-native) or Power BI (popular in manufacturing sector).  
  
Why it matters: Many manufacturing companies already have Power BI licences and expertise. Snowflake's native connectors for both tools enable easy integration.  
  
Technical details: Direct query mode (live connection to Snowflake) or import mode (cached extracts). Row-Level Security passes through from Snowflake. Embedded dashboards for white-labelling.

## 5.4 Data Flow and Processing

### End-to-End Data Flow (Simplified)

The beauty of Option B is the dramatically simpler data path:

1. Sensor publishes MQTT message to IoT Core (latency: <100ms)

2. IoT Core routes to Kinesis Data Streams via Rules Engine (latency: <200ms)

3. Snowpipe Streaming ingests directly into Snowflake raw table (latency: 5-10 seconds)

4. Snowflake Stream detects new rows immediately

5. Snowflake Task (runs every 1 minute) processes new rows and updates curated views (latency: <60 seconds)

6. QuickSight dashboard refreshes showing latest data (total latency: 60-65 seconds for dashboards)

**Important Note on Latency:**

The default configuration provides 60-65 second latency for dashboard updates. This violates the <10 second alert requirement. HOWEVER, this is easily fixed by adding a Lambda "fast-path" for alerts (see Section 8 for details). The fast-path adds £100-150/month and 2 weeks to timeline but achieves <5 second alert latency whilst keeping dashboard processing in Snowflake.

## 5.5 Strengths and Limitations

### Strengths

**Lowest Total Cost:** £2,170-3,450/month for 30 tenants (£72-115 per tenant). Cheapest option by 15-20%. Cost scales linearly with usage—no expensive clusters idling overnight.

**Lowest Complexity:** Only 5 core services vs 15+ in Option A. Single platform (Snowflake) handles most processing. Less to configure, monitor, and troubleshoot.

**Fastest Time-to-Market:** 20-week timeline (fastest of all options). SQL-first development means faster iteration. Large pool of SQL-skilled developers vs rare Flink specialists.

**SQL-First Development:** Transformations, aggregations, and ML all use SQL. Familiar to data analysts and engineers. Easier to review and maintain than Java/Scala code.

**Native Multi-Tenancy:** Snowflake's Row-Level Security is production-proven. Policy-based security reduces risk of data leakage. Tenant isolation enforced at database level, cannot be bypassed.

**Unified Data Governance:** All data in one platform simplifies governance. Single place to audit access, track lineage, enforce retention policies. Reduces compliance overhead.

**Automatic Scaling:** Snowflake warehouses auto-suspend when idle (no wasted cost) and auto-resume on query (no manual intervention). Multi-cluster warehouses handle query concurrency spikes.

**Proven at Manufacturing Scale:** Snowflake used by major manufacturers (Toyota, Siemens) for industrial IoT. Handles 100+ petabytes of data in production.

### Limitations and Considerations

**Alert Latency Requires Fix:** Default 60-65 second latency violates <10s requirement. FIX: Add Lambda fast-path for alerts (+£100-150/month, +2 weeks). After fix, alert latency <5 seconds whilst batch processing stays in Snowflake.

**Snowflake Licensing Cost:** Snowflake charges separately from AWS bill. Requires contract negotiation. Some organisations have policies against third-party data platforms (prefer AWS-native Option D).

**Vendor Lock-in (Medium):** Snowflake-specific features (Streams, Tasks, VARIANT) not portable to other databases. However, SQL code is mostly standard and data can be exported. Less lock-in than Option A (Flink expertise lost if switching).

**Limited Flexibility for Complex ML:** Cortex ML suitable for standard use cases (regression, classification). Custom deep learning or specialised algorithms require SageMaker integration (adds complexity). Not as flexible as Option A.

**Real-Time Latency Not Sub-Second:** Snowpipe Streaming provides 5-10 second latency, not <1 second. If sub-second requirement is legally mandated for ALL data (not just alerts), Option A or D required.

### When to Choose Option B

Select this architecture if:

* Cost and simplicity are top priorities (most SME scenarios)
* Team has SQL skills or can train quickly (common)
* 60-second dashboard latency acceptable, or willing to add Lambda fast-path for <5s alerts
* Want fastest time to market (20 weeks baseline, 22 weeks with fast-path)
* Need strong data governance and compliance (unified platform simplifies)
* Snowflake licensing not a blocker (or already have Snowflake contract)
* Prefer proven technology over cutting-edge (Snowflake widely adopted in manufacturing)

**✅ RECOMMENDATION:** Option B is the recommended default choice for the SMDH platform. It offers the best balance of cost, simplicity, and capabilities for 80% of manufacturing IoT scenarios. The alert latency issue is easily addressed with a Lambda fast-path. Only choose Option A if sub-second latency is legally mandated, or Option D if AWS-native is required.

# 6. Architecture Option C: AWS IoT SiteWise-Based Architecture

## 6.1 Overview

|  |  |
| --- | --- |
| Monthly Cost (30 tenants) | £6,334 ✅ CORRECTED (was £21,150) |
| Cost Per Tenant | £211 |
| Complexity Level | Medium-High (8-10 services) |
| Implementation Timeline | 28 weeks (longest) |
| Team Size Required | 3-4 engineers (SiteWise + AWS IoT) |
| Best Suited For | Teams with deep SiteWise expertise; <10 tenants |

### What is This Architecture?

Option C leverages AWS IoT SiteWise, a managed service specifically designed for industrial equipment monitoring. Imagine a platform purpose-built for factories—it "speaks" the language of industrial sensors and equipment, with built-in concepts like asset hierarchies (e.g., "Factory → Production Line → Machine → Sensor"), time-series storage, and automatic metric calculations.

Unlike general-purpose databases, SiteWise understands manufacturing concepts natively. You define an "asset model" (like a template for a CNC machine), and SiteWise automatically organises data, calculates metrics like Overall Equipment Effectiveness (OEE), and provides specialised dashboards.

**⚠️ IMPORTANT COST CORRECTION**

A previous analysis incorrectly calculated Option C cost at £21,150/month due to a 7.3x data volume error. The CORRECTED cost is £6,334/month (70% reduction), making this option economically viable. However, it remains the most expensive option and has multi-tenancy limitations.

## 6.2 Architecture Diagram

*[INSERT: SMDH-Option-C-SiteWise-Architecture.drawio diagram here]*

The diagram shows a dual-storage approach: SiteWise for continuous sensor data and Timestream for discrete events, with a custom React application providing white-labelled dashboards.

## 6.3 Detailed Component Description

### Layer 1: Dual Ingestion Path

**AWS IoT SiteWise Gateway (On-Premises)**

What it does: A software appliance (runs on a local server at the factory) that connects to industrial protocols like OPC-UA (common in manufacturing) and Modbus (used by energy metres). It "translates" these protocols into AWS IoT format.  
  
Why it matters: Many manufacturing sensors don't speak MQTT directly. The gateway bridges legacy equipment to the cloud without replacing existing infrastructure.  
  
Technical details: Runs as Docker containers on EC2 instances or on-premises hardware. Supports OPC-UA, Modbus TCP, Ethernet/IP. Includes local caching for offline operation and automatic retry.  
  
Operational consideration: Requires managing on-premises software at each manufacturing site. Updates, security patches, and troubleshooting must be performed remotely or on-site.

**Split Routing: Time-Series vs Events**

What it does: IoT Rules Engine routes data to different destinations based on type:  
 • Continuous data (machine status, temperature) → SiteWise  
 • Discrete events (RFID scans, job completions) → Kinesis → Timestream  
  
Why it matters: SiteWise is optimised for time-series data, not discrete events. Job tracking requires a different storage model (Timestream), creating a dual-storage architecture.  
  
Complexity impact: Applications must query two different databases. Data correlation across systems requires custom logic. Increases overall system complexity.

### Layer 2: AWS IoT SiteWise (Core Platform)

**SiteWise Asset Models and Hierarchies**

What it does: Define templates for equipment types (e.g., "CNC Machine Model X") with properties (spindle speed, temperature), metrics (utilisation %), and alarms. Organise assets into hierarchies (Site → Building → Floor → Machine).  
  
Why it matters: Provides structure for industrial data. Calculations like OEE (Overall Equipment Effectiveness) are built-in formulas, not custom code. Hierarchy enables roll-up reporting (see total factory OEE, drill down to specific machines).  
  
Technical details: Asset models defined via API or console. Supports inheritance (base model → variants). Formulas use SiteWise Expression Language (proprietary, similar to Excel formulas). Assets can have static attributes (serial number, install date) and dynamic measurements.  
  
Manufacturing-specific benefit: Unlike generic databases, SiteWise "understands" concepts like availability, performance, and quality (the three components of OEE).

**SiteWise Compute Expressions**

What it does: Automatically calculates derived metrics from raw sensor data. Example: "If machine\_state = RUNNING, increment running\_time; calculate utilisation = running\_time / total\_time."  
  
Why it matters: No Lambda functions needed for standard manufacturing calculations. SiteWise performs these computations in real-time as data arrives.  
  
Technical details: Transform metrics (unit conversions, averaging), Metric metrics (aggregations like sum/avg/min/max over windows), Measurement metrics (raw sensor values). Computed at ingestion time. Results stored alongside raw data.  
  
Limitation: Expression language less flexible than Python/Java. Complex business logic still requires Lambda.

**SiteWise Time-Series Storage**

What it does: Stores sensor readings in a managed time-series database optimised for industrial IoT query patterns.  
  
Why it matters: Automatic tiering: hot tier (13 months) for fast queries, cold tier (14+ months) for archival. No manual lifecycle management.  
  
Technical details: Storage charged per GB stored + per million values. Automatically indexes by timestamp and asset hierarchy for efficient queries. Supports batch and real-time queries.  
  
Cost structure: $0.75 per million values ingested + $0.35/GB/month hot tier + $0.03/GB/month cold tier. This pricing model can be expensive for high-frequency sensors (1Hz = 86,400 values/day/sensor).

**⚠️ Multi-Tenancy Challenge: Tag-Based Isolation**

What it does: SiteWise does NOT have native multi-tenancy. The workaround is to tag every asset with a "tenant\_id" and filter queries in the application layer.  
  
Why this is problematic: Unlike Snowflake's Row-Level Security (database-enforced), tag-based filtering relies on application code correctly filtering queries. A bug or misconfiguration could expose one tenant's data to another.  
  
Technical implementation: Lambda functions must add WHERE tenant\_id = :authorized\_tenant to EVERY query. Custom React application must enforce filtering in UI. Requires extensive security testing.  
  
Risk assessment: Medium-high risk for SaaS platforms. Acceptable for single-tenant deployments or <10 trusted tenants. Requires rigorous code review and penetration testing for multi-tenant SaaS.

### Layer 3: Event Storage (Amazon Timestream)

**Why Dual Storage? SiteWise Limitations for Events**

SiteWise is designed for continuous sensor readings (machine running at 1500 RPM, temperature is 65°C). It is NOT well-suited for discrete events like:  
 • RFID scan at 10:23 AM: "Job #1234 scanned at Station 5"  
 • Barcode read: "Product X456 completed assembly"  
  
These events have irregular timing, rich metadata (job details, operator ID), and different query patterns (find all scans for Job #1234 in the past week). Timestream is purpose-built for this use case.  
  
Consequence: Option C requires BOTH SiteWise (time-series) AND Timestream (events), increasing complexity.

**Amazon Timestream for Event Data**

What it does: Stores job tracking events, RFID scans, barcode reads, and other discrete event data.  
  
Why it matters: Native SQL interface. Efficient for event-based queries ("find all movements of Job #1234"). Separate from SiteWise to avoid forcing event data into time-series model.  
  
Technical details: Same as Option D (see Section 7). Memory store for recent data, magnetic store for historical. SQL query engine. Native multi-tenancy via partition keys (better than SiteWise's tags).

### Layer 4: Custom Dashboard Layer

**Why Custom Dashboards? SiteWise Monitor Limitations**

AWS provides SiteWise Monitor—a built-in dashboarding tool. However, it has critical limitations for SaaS:

• Cannot fully white-label: AWS branding and URLs remain visible

• Limited customisation: Dashboard layout and styling options are constrained

• No multi-tenancy support: Not designed for SaaS platform with many companies

• Basic visualisations: Chart types limited compared to QuickSight or Grafana

Consequence: Option C requires building a custom React application that calls SiteWise and Timestream APIs directly. This adds 6-8 weeks to the timeline and ongoing maintenance burden.

**Custom React Application**

What it does: Custom web application built in React that queries SiteWise and Timestream via AWS SDK, renders charts (using libraries like Chart.js or Recharts), and provides white-labelled UI.  
  
Why it matters: Full control over branding, layout, and user experience. Can integrate data from both SiteWise and Timestream into unified dashboards.  
  
Technical details: React 18+ with TypeScript. AWS Amplify for authentication integration. API Gateway + Lambda for backend. Queries SiteWise via BatchGetAssetPropertyValue API and Timestream via SQL.  
  
Development effort: 6-8 weeks for initial dashboard suite. Ongoing: new chart types, dashboard customisation requests, browser compatibility, performance optimisation.

## 6.4 Data Flow and Processing

### Time-Series Data Path (Machine/Energy/Air)

1. Sensor publishes via OPC-UA/Modbus to SiteWise Gateway (on-premises)

2. Gateway forwards to AWS IoT SiteWise (latency: <1 second)

3. SiteWise ingests data, applies asset model, calculates metrics (latency: <2 seconds)

4. Custom React app queries SiteWise API for dashboard updates (latency: <5 seconds total)

5. User sees real-time updates in browser

### Event Data Path (RFID/Barcode Scans)

1. Scanner HTTP POST to API Gateway (latency: <100ms)

2. Lambda validates and writes to Kinesis Data Streams (latency: <200ms)

3. Lambda consumer writes to Timestream (latency: <1 second)

4. Custom React app queries Timestream SQL for job tracking dashboard (latency: <3 seconds total)

5. User sees job location updates

## 6.5 Strengths and Limitations

### Strengths

**Purpose-Built for Manufacturing:** SiteWise provides native industrial concepts: asset models, OEE calculations, equipment hierarchies. Speaks the language of manufacturing rather than generic database terms.

**Managed Asset Modeling:** No custom code needed for standard manufacturing calculations. Built-in formulas for utilisation, availability, performance. Reduces development compared to building from scratch.

**Real-Time Latency:** Achieves <1 second latency for time-series data. Meets real-time monitoring requirements without complex stream processing like Flink.

**Industrial Protocol Support:** SiteWise Gateway handles OPC-UA, Modbus TCP without custom integration code. Easier connectivity to legacy equipment.

**Automatic Metric Calculations:** Compute expressions run at ingestion time. No separate stream processing layer needed for standard transformations.

**Predictive Maintenance Integration:** Amazon Lookout for Equipment integrates directly with SiteWise. Pre-built anomaly detection for manufacturing equipment (though less flexible than custom SageMaker models).

### Limitations and Challenges

**NOT Designed for Multi-Tenant SaaS:** CRITICAL: SiteWise has no native Row-Level Security. Tag-based filtering is application-layer workaround that increases data leakage risk. Acceptable for <10 trusted tenants; risky for SaaS with many companies.

**Most Expensive Option:** Cost: £6,334/month (£211/tenant) vs £2,170/month for Option B (cheapest). 3x more expensive. Ingestion pricing ($0.75 per million values) adds up quickly with high-frequency sensors.

**Dual Storage Complexity:** Requires BOTH SiteWise (time-series) AND Timestream (events). Applications must query two databases and correlate data. Increases architectural complexity and operational overhead.

**Custom Dashboard Development Required:** SiteWise Monitor cannot be adequately white-labelled. Must build custom React application (6-8 weeks + ongoing maintenance). Dashboards not included "out of the box" like QuickSight/Grafana.

**Longest Implementation Timeline:** 28 weeks due to custom dashboard development, dual storage integration, and SiteWise asset model configuration. 40% longer than Option B (20 weeks).

**SiteWise Learning Curve:** Proprietary concepts (asset models, transforms, metrics) require training. Expression language different from SQL or Python. Smaller talent pool compared to general databases.

**Limited ML Flexibility:** Lookout for Equipment is pre-built (less flexible than SageMaker). Cannot use custom algorithms. Still requires SageMaker integration for advanced use cases (adds complexity back).

**On-Premises Gateway Management:** SiteWise Gateway requires deploying software at each manufacturing site. Software updates, troubleshooting, and security patching across distributed locations.

### When to Choose Option C

Select this architecture ONLY if:

* Team has existing deep AWS IoT SiteWise expertise (rare)
* Manufacturing focus with <10 tenants (multi-tenancy risk acceptable)
* Already using SiteWise for other projects (leverage existing skills)
* OPC-UA/Modbus connectivity is critical and must be managed (not delegated to device vendors)
* Budget supports £200+/tenant (most expensive option)
* Willing to invest 28 weeks + custom dashboard development
* Can accept dual storage complexity (SiteWise + Timestream)
* AWS-native preference outweighs Snowflake simplicity (but see Option D for better AWS-native choice)

**⚠️ ASSESSMENT:** Option C is viable after cost correction but NOT recommended for multi-tenant SaaS. The multi-tenancy risk (tag-based isolation), dual storage complexity, and custom dashboard burden outweigh the benefits of managed asset modelling. Choose Option B for simplicity or Option D for AWS-native without SiteWise limitations.