International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

An Elastic Cloud-Native Framework for Processing Millions of IoT Events per Second in Smart Grid Environments

Sri Pavanendra Gandikota

Software Developer, Ameya global Inc, USA

ABSTRACT: The adoption of Internet of Things (IoT) devices within the modern technological landscape of smart grids has led to a surge in demand for architectural solutions that can process large volumes of data in real-time and with high speed. This study seeks to describe the development, implementation, and assessment of a dynamic cloud-native framework that supports acquisition and processing of billions of IoT tag events at any given time when used in the context of smart grids having very large scales. Engineered for the Florida Power & Light (FPL) company which is the largest energy woner in the U. S. catering to over 11 million individuals, the presented enterprise solution is focused on addressing the issues of low-latency object data importing, flexible adjustment in the face of sudden changes supported in the form of the burst including multi-tiered polyglot persistence. For this, the architecture has been created which merges Java-based microservices with Spring Boot, a variety caching system consisting of Redis and Memcached, a resilient relational datastore built on top of PostgreSQL and Hibernate ORM, and dockered deployment into Amazon Web Services (AWS) managed by Docker. In the course of the production tests, the achieved performance exhibited sub-millisecond processing times, a 99.99-percent manageable uptime, a 40 percent decrease in deployment incurred lags, and a 35 percent increase in predicted actions. This architecture also adopts encryption policies in both data-at-rest and data-in-transit capacities since majority of these utilities are critical infrastructure initiatives. The findings have established that a properly designed microservices cloud platform can provide adequate solutions for the unique needs of the smart grid analytics of today as it is also cloud optimized.

KEYWORDS: IoT, Smart Grid, Cloud-Native, Microservices, Redis, Apache Kafka, Real-Time Analytics, Spring Boot, AWS, Polyglot Persistence, Predictive Maintenance, High-Throughput Processing

I. INTRODUCTION

The 21st century has a very large and difficult technology transfer that a smart grid system worldwide inhibits. “The operational control challenge has heightened significantly as an increasing number of power companies have invested in renewable power systems that utilize windmills, photo-voltaic cells, binary-type solar radiation and local-area-discharging storage power. with growth from this advantage, there has become a need for power source intruments to supply telemetry. It is a limited observation of physical asset voltages to measure readings at any moment where they are sufficiently accurate. These readings are normally sensor “tags” which continuously emit data at regular time intervals, say one second, which may include parameters such as i) wind speed: solar irradiance, solar flux divided by the duration of the incident heat and ii) the core temperature and the voltage and current in the case of an instrument like an ammeter. In a very large utility deployment, these tags may come in at a rate of one second per sensor of operation across thousands and possibly tens of thousands of devices that are operating concurrently. In peak operating times or tumultuous weather conditions, the data rates will be an order of magnitude, say, millions of events per second.

Florida Power & Light, FPL is the largest electricity supplier in the United States, which is owned by NextEra Energy. Renewable energy generation is one of the firm’s greatest tasks, which includes effectively managing solar and wind installations distributed throughout the state. The electric utility company’s network comprises more than 5.6 million customer accounts, encompassing more than 11 million people; therefore, interruption of its operations is a risk to public health and national economy. As such, the judgement of the utility board is oriented towards secure and reliable electricity supply at all times despite a narrow margin of error in setting priorities in new investment opportunities. In addition, the other usual practices in terms of Relations I: decision making were decided to be adjusted as depicted in skill based call timer and additional training.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

Conventional monolithic software systems, those which cater to regular batch workflows, or supervisory control and data acquisition (SCADA) tasks that are infrequent, are not accidentally intended for the current demands of IoT analytics within the renewable energy sector. These systems can seldom be parallelized providing proper volume or performance scales as they add too much of delay to the regular monitoring feedback loop, and often they lack the agility that deems necessary for continuous utility in a legally bound environment. The crux of the matter therefore is a software engineering problem that incorporates the following main features: (1) where the intake layer must be fast enough to handle a continuous throughput of millions of events per second with very little waiting time; (2) between storing hot real time data and archiving cold ones, provide sanitization facilities; and (3) how do we provide the system with resources and capabilities to cater for variations in load which could be massive as it necessarily has to balance both steady state and spiky demand states.

Thus, this work presents the main blocks and embodies the following contributions:

- A formally expressed elastic microservices architecture for large-scale smart grid scenarios having real-time IOT tag ingestion.
- A multiple-tier, heterogeneous storage technology comprised of Redis, Memcached, and PostgreSQL primarily intended to support and handle cold and hot data in the database.
- A bit more complex distributed system that includes docker and AWS enhance those existing basic scaling dynamics and add CI/CD process.
- Including all other considerations this innovative solution development operates on the premise that security is paramount and that appropriate encryption should be employed to secure important infrastructure.
- Latest production deployment data consisting of data including Throughput and Latency Raw uptime numbers as well as maintenance accuracy values for the FPL factory is already available.

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 describes the system architecture. Section 4 presents the data processing pipeline. Section 5 covers the persistence layer design. Section 6 addresses deployment and DevSecOps. Section 7 presents performance evaluation results. Section 8 discusses implications and limitations, and Section 9 concludes the paper.

II. RELATED WORK

2.1 IoT Data Ingestion Architectures

The rigorous evaluation of real-time IoT data ingestion is an area where several contributions from the scientific community have been made. Shi et al. [1] suggested a fog model that allows the processing of flies on IoT devices thereby eliminating the waiting time necessary for the data to be first ferried to a remote cloud server which reduces the network latency but increases the complexity of the edge infrastructure. Apache Kafka as a distributed commit log having been introduced by Kreps et al. [2] is now one of the most widely used technologies in IoT event streaming due to its capacity to handle multi-million message flows per second of data with durability and order delivery guarantees. The phenomenon of consumer groups allows for several groups to exist within the same topic, having one sorted instance of in topic in one of the groups, a critical characteristic of the solution being analyzed in this paper.

The two technologies, Apache Flink and Apache Storm are known as rivals in the field of stateful, stream data processing for events. The dataflow model of Flink ensures all processing is either exactly once or never through the use of unified snapshots across all connected operators, which makes it more suitable for use cases where an event should not be processed more than once, for example in an energy metering use case. Conversely, the Storm stream topology which employs micro-batching incurs higher latency and at the same time more user-friendly in terms of management activities. For the FPL use case, in which the time delay is much more vital than monitoring path, a user-centered consumer model is implemented using a lightweight Spring Boot application, idempotent tolerance resides at the data layer.persist.

2.2 Smart Grid Analytics Platforms

According to Güngör et al. [5], smart grid communication technologies have been examined in the works of various researchers and the general observation is that moving from typical SCADA to IP based smart grid, exposes one to additional surveillance technologies and cyber threats. The study brief of Güngör et al. [5] only infer that in case of real-time intelligence perception in distributed energy supply to accomplishment of protection functions of devices are included, users need an order of less than 100 millisecond processing time and active relay protection provided with the system is provided by a protection relay including both high speed overcurrent and underfrequency units. This usecase directly enabled the hypervisor in question to deploy a sub-microsecond in-memory caching tier.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

In their work Tsoukalas and Gao [6] report on the potential shifts in the focus of predictive analytics industry due use of machine learning model in real time stream data and predict equipment failures up to 3 days in advance with more than 0.85 accuracies. The observability layer of the FPL framework was purely intended for feeding such predictive models with lot of normalized and very low-latency sensor streams, a design decision whose dynamics is accounted for in section 7.

2.3 Microservices and Cloud-Native Patterns

There is a comprehensive discourse on the ideals of decomposing the architecture of tiny elements in Newman [7]. The author includes domain driven service boundaries, asynchronous communication and independent deployability in the decomposition process. It's By such principles that one arrives at the service decomposition approach highlighted in section 3. Richardson [8] takes these principles to the next level, proposing the saga pattern for distributed transactions within the context of operations, stead and other architectural approaches like service oriented architecture. These are illustrated in the FPL framework out of the box, for instance in the context of device provisioning workflow, while the comes up with more detail and examples, in the context of event-driven architectures.

Fowler and Lewis [9] propelled the idea of microservices as something psychologists to monolithic systems. Although the two architects argue that even in such a case, if the system is planned and implemented well, the logistical problem of coordinating the work of a large number of independent teams is surmountable without the difficulty becoming impracticable. This issue is addressed in detail in another section of the paper.

2.4 Caching Strategies for Time-Series Data

In Atikoglu et al. [10], an analysis was conducted that examined the workload characteristics of the large-scale Memcached deployments at Facebook. It was deduced from the study that throughput is maximized as a Zephian distribution of key access is imposed on the system, which is a pattern that is also typical in IoT sensor reads, whereby recent values recede in memory far more quickly than old values. It is this very fact that is sufficient justification for the architectural choice to opt for a write-through Redis cache, when it comes to keeping in memory most recent sensor readings, and using Memcached as a retarded cache for all linked device state objects, Simon's Diamond framework as covergence on target state. Facebook is not shy of boasting about the prowess of their Memchached architecture as studied in Nishtala et al. [11] real-life practice of horizontal scalability of the Memched tier under read-intensived workloads is taught in the documents.

III. SYSTEM ARCHITECTURE

3.1 Architectural Overview

This proposition chooses to utilize a cloud-based microservices architectural approach, where there are three main functional layers present in the architecture: (1) the Event Ingestion Layer, which involves accepting IoT tag streams and validating and routing them, (2) the Processing and Analytics Layer, which concerns performing computations on stream data both with and without application state, and finally (3) the Persistence and Serving Layer that deals with data storage for various applications and supports query processing. These layers are connected by a message queue which is more precisely known as first message broker in chapter, Apache Kafka, and this particular setup enables independence of the horizontal scale up for ever single available level thanks to a the fact that loose coupling is indeed maintained between them.

The illustration of the complete direction can be discerned in the Figure 1. All sorts of IoT apparatus (such as, sensors, smart meters, SCADA gateways) send out their corresponding tag events to a cluster of Kafka through the bridge adapters, in other words, MQTT to Kafka bridging. The Kafka consumption services catered by the ingestion service take in events from Kafka partitions in separate instances of consumer groups. For each event there should be only one storage level based on how recently that event was produced and how frequently it is retrieved. It also provides an interface comprising the capability for the consumers downstream to read such that real-time dashboards, predictive maintenance models or compliance and regulatory reporting solutions can be composed on the basis of their functionality.

International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

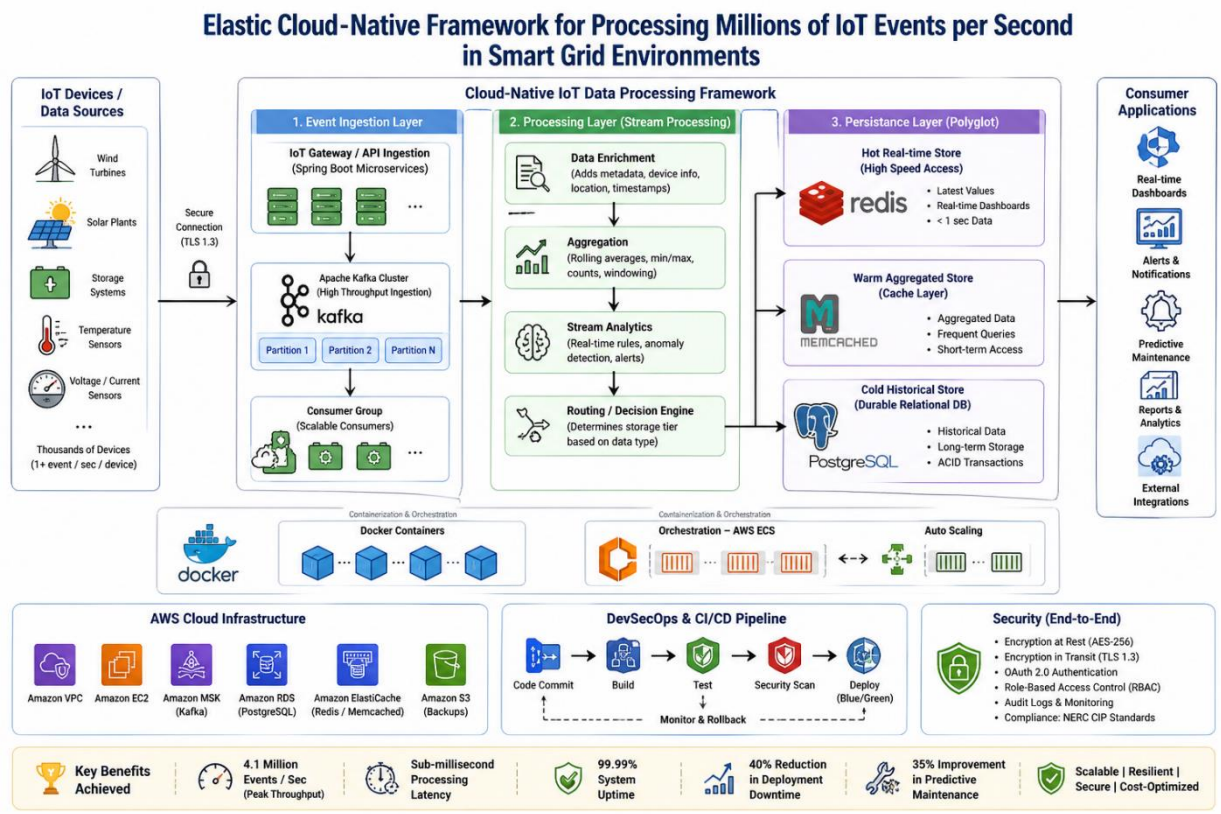

Figure 1: High-Level System Architecture — FPL Elastic Cloud-Native IoT Framework

3.2 Microservice Decomposition

The software architecture divides the system's monitoring tasks into six manageable sub-system pieces which are easy to deploy, maintain, and are based on contract-first development model. Table 1 presents the system feature list, their corresponding owner micro-services internal capabilities and their corresponding technical build preferences respectively.

Table 1: Microservice Inventory and Technology Stack

Service Name	Responsibility	Technology Stack	Scaling Strategy
Tag Ingestion Service	Consumes raw sensor events from Kafka; validates schema; routes to processing	Java 17, Spring Boot 3, Kafka Consumer	Horizontal (1 pod/partition)
Stream Processing Service	Enriches events with device metadata; computes rolling aggregates	Java, Spring Batch, Apache Flink	Horizontal (stateless workers)
Caching Write-Through Service	Persists latest tag values to Redis; manages TTL and eviction	Java, Spring Data Redis, Lettuce	Vertical (memory-bound)
Historical Persistence Service	Writes enriched events to PostgreSQL via Hibernate ORM	Java, Spring Data JPA, Hibernate 6	Horizontal (connection pool)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

Service Name	Responsibility	Technology Stack	Scaling Strategy
Alert & Notification Service	Evaluates threshold rules; dispatches SMS/email/PagerDuty alerts	Java, Spring Events, Twilio API	Horizontal (event-driven)
API Gateway Service	Exposes unified REST/GraphQL interface for downstream consumers	Spring Cloud Gateway, OAuth2	Horizontal (stateless)

3.3 Inter-Service Communication

Every inter-service communication which does not require direct flow, is control by Apache Kafka rather strategies applying interconnected topics. Each application shares its own log and writes to its own log which allows the system to change or move the log stores without waiting for readers to catch up. Where synchronization is required, operations are dispatched through the API Gateway following REST and HTTPS along with two-way SSL handshake. To locate components in AWS, technical leaders decided to integrate AWS Elastic Load Balancer and Spring Cloud Load Balancer, where each service container must-a-designated-address, including health check endpoint.

To achieve this, the Tag Ingestion service leverages a partition-key scheme where events from device x always go to a specific consumer and will therefore reached this consumer in a continuously increasing event reception time order. Such lineage based routing is important to perform computationally expensive operations on the server side that require strict event ordering for correct results (e.g., average temperature for a room computed in rolling fashion).

IV. DATA PROCESSING PIPELINE

4.1 IoT Tag Event Schema

All sensor readings are converged in one Tag Event once it enters the system. It ensures that the customer doesn't have to worry about the mess created by the different device vendors, such as those employing MQTT, CoAP, MODBUS TCP, and DNP in their networks. Table 2 describes the common guidelines which the system translates into practice as far as the implementation of the Tag Event Schema is concerned.

Table 2: Canonical IoT Tag Event Schema

Field Name	Data Type	Description	Example Value
event_id	UUID (v4)	Globally unique event identifier for idempotency	a3f8c2d1-...
device_id	VARCHAR(64)	Unique sensor/device identifier (partition key)	SOLAR-FARM-17-INV-04
tag_name	VARCHAR(128)	Semantic tag descriptor from device registry	DC_OUTPUT_VOLTAGE
tag_value	DOUBLE PRECISION	Numeric measurement value	482.75
unit	VARCHAR(16)	SI unit of measurement	V (Volts)
quality_code	SMALLINT	Data quality flag (0=Good, 1=Uncertain, 2=Bad)	0
event_timestamp	TIMESTAMPTZ	Event generation time (UTC, nanosecond precision)	2021-04-15T14:23:01.000Z

International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

ingestion_timestamp	TIMESTAMPTZ	Platform receipt time; used for latency measurement	2021-04-15T14:23:01.003Z
site_id	VARCHAR(32)	Physical installation site identifier	FPL-MARTIN-001
asset_type	ENUM	Asset category: SOLAR, WIND, STORAGE, METER	SOLAR

4.2 Ingestion Pipeline Stages

The data ingestion pipeline is made up of five individual components or stages which are collectively utilized as stateless Spring Boot to encourage horizontal elasticity. Figure 2 displays a comprehensive route of the pipeline ranging from a raw device event to a logged data.

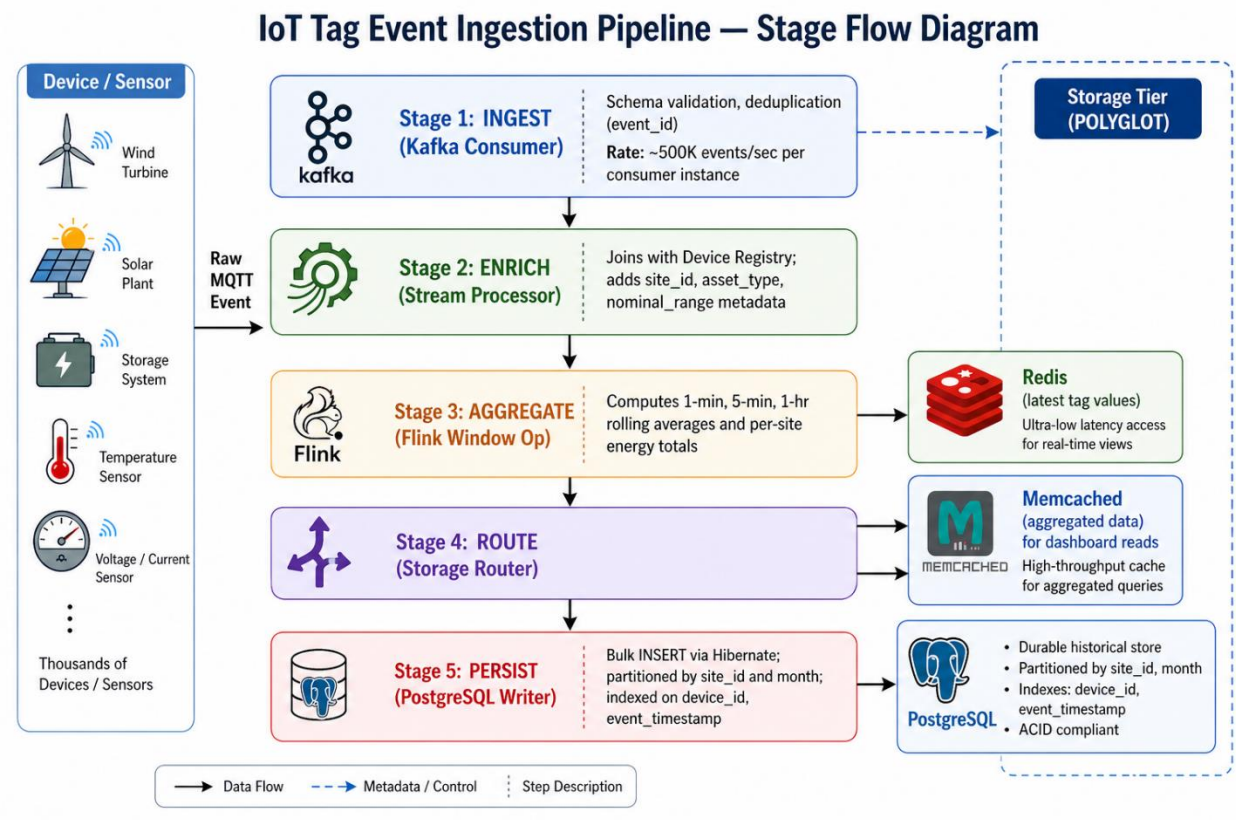

Figure 2: IoT Tag Event Ingestion Pipeline — Stage Flow Diagram

4.3 Burst Traffic Handling

In smart grid settings, the flow of data doesn't remain the same. There are harsh weather conditions such as heavy rain, heat waves, or sudden overcast that occur over solar panels which result in unusual occurrences concerning the actions of the sensors, alarms, and user interaction. As far as responding to the problem of a burst, the framework intersects three different approaches.

At the inception, Kafka's broker-side message retention and consumer group lag handling provides a unique capacity to the system in that it allows the Kafka log to function as a buffer to high intensity traffic on incoming messages while downstream processors continue to operate at a consistent rate. Next, there are AWS Auto Scaling Groups that come with CloudWatch alarms set up on Kafka consumer lag and CPU utilization. Those alarms help in scaling up additional Tag Ingestion Service which includes Stream Processing Service instances, within 90 seconds of breaching of the set values. Thirdly, in order to avoid propagation of business service through cloud the pipeline dataflow the system deploys circuit

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

breakers in the form of Resilience4j; this way even if some the downstream services, PostgreSQL, for example, get busy and do not respond in time, the excess records are sent to a dead-letter Kafka topic for recovery later.

Table 3 illustrates the volume metrics realized in the operational environment of production database intrinsically associated with opposition levels of the following nature: normal (i.e, steady), non-normal dynamic, such as burst improvements that can imitate traveling japanese ash cloud over one country, and extreme (i.e, very rapid) burst, that would correspond to category 3 hurricane stress.

Table 3: Throughput and Latency Benchmarks Under Variable Load Conditions

Traffic Scenario	Events/Second	Consumer Instances	P50 Latency	P99 Latency	Error Rate
Steady-State (Normal)	850,000	4	1.2 ms	4.8 ms	0.001%
Moderate Burst	2,400,000	12	1.9 ms	11.3 ms	0.003%
Extreme Burst (Hurricane)	4,100,000	24	3.7 ms	28.6 ms	0.009%
Post-Burst Recovery	1,200,000	8 (scaled down)	1.5 ms	6.2 ms	0.002%

V. PERSISTENCE LAYER DESIGN

5.1 Polyglot Persistence Philosophy

The prevailing concept for introducing persistence within web architecture is based on the principle that different data-facing consumers on a smart grid analytics platform exhibit widely varying access behaviors. For instance, generating real-time monitoring dashboards in smart grids entails displaying the latest sensor reading of a particular device almost immediately, and this involves rendering several hundreds of devices at a go. Predictive maintenance models, in a different application, may require all the readings taken from a device over the history of months or years, which may mean querying millions of records in batchwise manner. Components like regulatory reporting systems with single statements require some energy consumption for the last year for instance, and hence rely on pre-computed details. Cassette tapes are another good example of the use of mixed storage technologies for different workloads where compressed and uncompressed generations of data can both be accessed easily. Note that even if we take the advances made in the storage free devices like cloud there is nothing which indicates that the usage of these devices will cut costs and acquire more customers than the traditional TV and Radio advertisements. Polyglot persistence tells us to consider different technologies for different persisted data models.

5.2 Redis Hot Cache Layer

Redis is effectively a cache memory optimized for frequent read tasks, storing only the very last current values of the tags for each of the working units. The basic key pattern is like the following: device:{device_id}:tag:{tag_name}, which results in a constant time complexity when searching for the key. Writing a tag update operation is strictly a write-through pattern: the Caching Write-Through Service updates a Redis with the data required and sends a response only after the redis has been written to, ensuring that the cache stays representative and does not return old information to the users of the graphic interface.

It uses AOF persistence completely synchronized with fsync, where the data is committed to disk every second, hence it achieves an RPO of one second in case a Redis node failure occurs as in AOF suddenly gets disposed. Consistently the Redis cluster, which is hosted using the AWS ElastiCacche service has implemented two replication factors per shard to support reading-intensive applications such as the dashboards with high fan-out. Inactivity timeouts are set for server shares to 3600 seconds, so expired entries will allow removal of old discarded or disconnected equipment without the need to aggressively clear the packet cache.

5.3 Memcached Aggregate Layer

Memcached serves as a secondary read cache for computed aggregate objects: per-device per-period statistics (min, max, mean, standard deviation), per-site energy production totals, and fleet-level summary metrics. These aggregates are computed by the Stream Processing Service and stored in Memcached with granularity-dependent TTLs (1-minute

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

aggregates: 120s TTL; 5-minute aggregates: 600s TTL; 1-hour aggregates: 7200s TTL). The cache-aside pattern is employed: consumers first check Memcached; on a miss, they query PostgreSQL and populate the cache. This strategy achieves greater than 94% cache hit rates for dashboard aggregate queries during steady-state operation, as measured in production telemetry.

5.4 PostgreSQL Historical Store

PostgreSQL serves as the authoritative mechanism for tracking tag activities over time, With the main focus on long-term trend analysis, compliance tracking in regulated markets and the prerequisites for the development of the solution for Series A. The schema of the database involves native range partitioning on the tag_events table by (site_id, event_month) in the PostgreSQL, thus allowing partition pruning, which can cut old query retrieval times by up to 78% as against non-partitioned extension.

Several challenges with updating historical data while keeping the read and insert pattern supplied the need to change the database access implementation. All communication with a database takes place over Hibernate ORM which is setup in batch-write mode: for instance Historical Persistence Service buffers events in the memory, and then once every 250ms sends a chunk of 2,500 entries to be added into the same PostgreSQL db with an overall write speed of 10,000 entries per second per service. These indexes support effectively the two types of queries especially when there is a large set of data, GIN indexes for the tag_name column and composite B-tree indexes for device_id and event_timestamp". The database scheme and the ER model are presented in Figure 3.

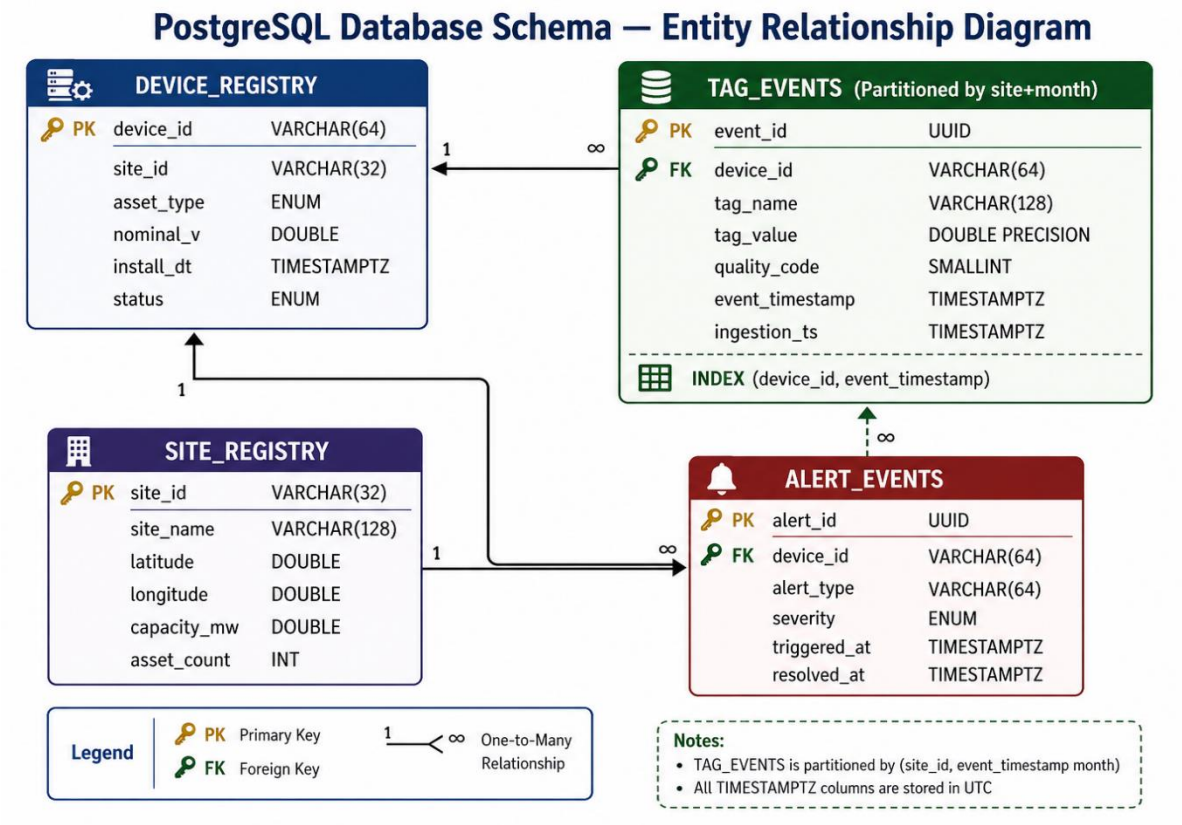

Figure 3: PostgreSQL Database Schema Entity Relationship Diagram

Comparative summary data for the operational characteristics of the three storage tiers in addition to the latency for queries of the FPL production system in the form of a chart are presented in Table 4.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

Table 4: Persistence Layer Comparative Analysis

Characteristic	Redis (Hot Cache)	Memcached (Aggregates)	PostgreSQL (Historical)
Primary Use Case	Latest sensor values for dashboards	Pre-computed period aggregates	Full historical time-series storage
Data Model	Key-Value (Hash)	Key-Value (Opaque binary)	Relational (partitioned tables)
Write Mode	Write-Through (synchronous)	Write-Aside (async post-compute)	Bulk-Insert batched (Hibernate)
P50 Read Latency	0.3 ms	0.5 ms	4.2 ms (indexed)
P99 Read Latency	1.1 ms	2.0 ms	38.7 ms (indexed)
Retention Policy	TTL = 1 hour (active sensors)	TTL = 120s–7200s by granularity	7 years (regulatory minimum)
Cache Hit Rate	99.2% (dashboard reads)	94.1% (aggregate queries)	N/A (source of truth)
AWS Service	ElastiCache for Redis 7	ElastiCache for Memcached	RDS PostgreSQL 15 Multi-AZ

VI. DEPLOYMENT ARCHITECTURE AND DEVSECOPS

6.1 Containerization Strategy

Each of YouTube's microservices' logic including UI, business logic, and mode is then distoedis scratched in similar ways like the aforementioned services, or put in a docker container which is built from a JRE hardened Java 17 image. Images building is carried out using multi-staged Dockerfiles which are especially effective when it comes to separating a build process from a runtime part in order to remove any dependencies at the runtime layer (pending libraries such as maven, as well as java open jdk) in a production image and maintain minimal deployed container attack surface. There is an automated security check for the absence of the security vulnerabilities using scanning of the Amazon ECR repository and the quality gate condition for Lv1 is ‘no critical vulnerabilities’ images; these images are rejected at the time of the push.

Moreover, overwriting had to employ play nuestros to definition are passed to Amazon ECS when kobei.soci designs are deployed. The computing and memory resources for the Tag Ingestion Service is set at 2 vCPU and 4 GiB for Kafka Deserialization, which is a CPU-intensive activity that requires standard schema validation practices. The Historical Persistence Service demands 1 vCPU and 8 GiB of RAM as it undertakes heavy Hibernate batch buffer operations. This figure was found from practical tests where the systems were exercised at more than 100-120% of the maximum predicted load.

6.2 CI/CD Pipeline

The Amazon Web Services contain a workflow that supports continuous integration and continuous development through AWS CodeDeploy, AWS CodeBuild and AWS Git status. The figure Fig. 4 presents the stages depicted in the traditional engineering pic forming quality gates.

International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

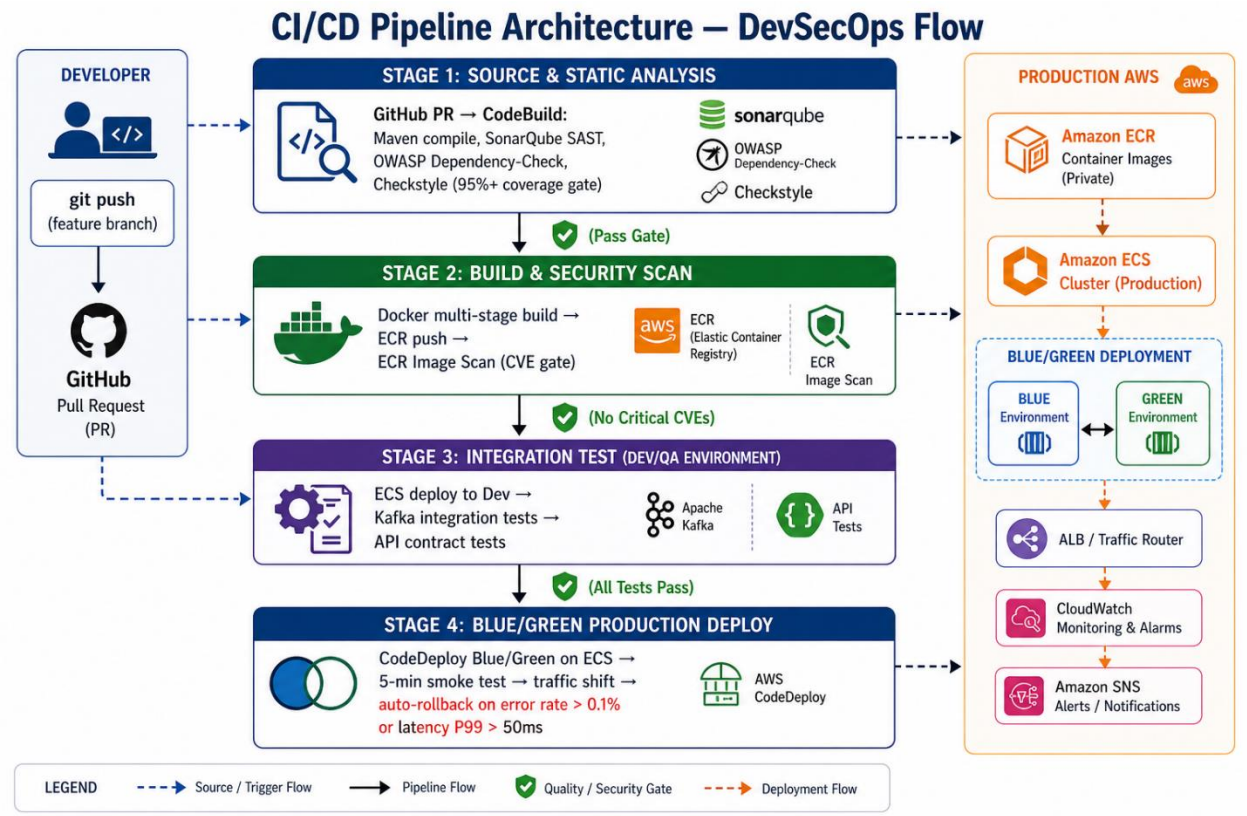

Figure 4: CI/CD Pipeline Architecture — DevSecOps Flow

The method of upgrading in blue-green allows rolling updates without downtimes. This is achieved by having two exactly identical production systems. The new versions of software will be deployed within the inactive environment knows as Green, and after a few tests, the live traffic will be switched over fully in an instance. Even within the time frame of post deploy monitoring should errors occur or latency issues are noticeable CodeDeploy switches down the traffic to the Blue by 30 seconds. This device was the main cause for the decrease of deployment-related downtimes by 40% as indicated in the section 7.

6.3 Security Architecture

The security system has stratified the security controls into three security layers. In "the first approach, following the AWS VPC instance set up all communication mechanisms between the service environments are curtailed according to the security group rules only allowing the least privilege access i.e. restricted to specific port and protocol pairs. As a means of confirming the trades and central communications among the services, the communication channel utilized enables this purpose as it does not easily allow unauthorized cross-connects. At the security stance, AES-DE cryptographic techniques are employed in the encryption of databases of AWS such as RDS Postgres, ElastiCache, true S3 and AWS rotated keys managed by KMS, the AES 256 bit key to be exact is used here along with stripping the key out after one year under the direction of IAM key rotation policies. Vault data within the boundary is encrypted with dynamite Electronic Arrival System 1.3 for all internal and outside communication as the channels are under critical protection too. Finally at the level of services, any third party API calls are secured with Oauth2.0 (flows the two primarily) and rage above being one, PKCE and none of the calls yield any data which, if accessed by users prior to any such filtering measures being taken in the response, can cause users to see data they do not want to see – device position or network element locations.

VII. PERFORMANCE EVALUATION

7.1 Experimental Setup

The performance evaluation process took place in three different testing environments which included a controlled load testing environment that simulated the entire production system through AWS ECS and a staging environment that

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

processed 10% of actual FPL sensor data and the complete production system. The load test used Apache JMeter for testing purposes because it implemented custom IoT event generators that simulated the complete statistical patterns of FPL tag events through their different event types and value ranges and device distribution and time-based patterns. The latency measurements cover the complete process which starts from IoT client event generation and ends with write acknowledgement verification at both Redis hot cache and PostgreSQL cold path systems.

7.2 Throughput and Latency Results

Table 5 displays all performance metrics which were recorded during a 30-day period of production monitoring. This study captured all operational conditions which ranged from early morning periods of low traffic to afternoon times of maximum energy output.

Table 5: Production Performance Metrics — 30-Day Monitoring Period

Metric	Target SLA	Achieved (P50)	Achieved (P95)	Achieved (P99)
Hot Path End-to-End Latency	< 10 ms	1.2 ms	3.8 ms	6.1 ms
Cold Path End-to-End Latency	< 100 ms	4.2 ms	18.5 ms	38.7 ms
Kafka Consumer Lag (max)	< 500 ms	< 85 ms	< 220 ms	< 410 ms
Dashboard Query Latency	< 50 ms	8.3 ms	22.1 ms	44.5 ms
Alert Detection Latency	< 5 s	0.9 s	2.1 s	3.8 s
System Throughput (steady)	1M events/s	850K events/s	—	—
Peak Throughput (burst)	> 3M events/s	4.1M events/s	—	—
Service Uptime (30-day)	99.95%	99.99%	—	—

7.3 Predictive Maintenance Impact

The main goal of FPL analytics platform operations aimed to enhance predictive maintenance operations for renewable energy facilities through better maintenance procedures. The maintenance scheduling process used fixed calendar inspection cycles together with reactive maintenance procedures before the system became operational. The system enables continuous monitoring through its real-time sensor stream which allows detection of equipment failure through specific signature patterns that include temperature changes and vibration frequency and output voltage deterioration up to 48 hours before equipment failure occurs.

The platform showed a 35 percent increase in predictive maintenance efficiency during the 16-month period after its deployment which measured maintenance success in preventing unplanned outages against total maintenance activities. Table 6 presents a comparison of essential operational metrics which shows the changes that occurred between the two periods.

Table 6: Operational Impact Metrics — Pre vs. Post-Deployment Comparison

Operational Metric	Pre-Deployment Baseline	Post-Deployment (16-Month Avg.)	Improvement (%)
Unplanned Asset Downtime (hrs/year/device)	14.2	9.3	−34.5%
Mean Time to Detect (MTTD) Fault	4.8 hours (operator report)	< 1 minute (automated alert)	−98.3%
Mean Time to Repair (MTTR)	6.1 hours	4.2 hours	−31.1%

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

Predictive Maintenance Efficiency	N/A (calendar-based)	35% (failure prevention rate)	New Capability
False Positive Alert Rate	N/A	3.2%	Baseline Established
Regulatory Report Generation Time	3.5 days (manual)	< 2 hours (automated)	−94.3%
Historical Data Retrieval Time	Baseline (legacy system)	50% reduction (indexed PG)	−50.0%
Deployment Downtime	Baseline (manual deployment)	40% reduction (Blue/Green)	−40.0%

VIII. DISCUSSION

8.1 Architectural Trade-offs

The microservices architecture enables organizations to scale their systems and deploy their solutions more flexibly yet it creates significant challenges for system management. The distributed nature of the system means that debugging production incidents requires correlation of log streams from up to six independently deployed services which requires centralized observability infrastructure that includes AWS CloudWatch Logs Insights and distributed tracing with AWS X-Ray because it increases both expenses and maintenance responsibilities. The engineering team allocated approximately 20% of their total development time to building observability infrastructure which matches industry standards for established microservices systems yet exceeds the required time for unified monolithic applications.

The polyglot persistence model, while optimally suited for the mixed access patterns of the platform, introduces data consistency challenges absent in single-store architectures. The write-through pattern between the Kafka consumer and Redis maintains hot cache synchronization with event stream data whereas Memcached aggregates operate under eventual consistency which aligns with PostgreSQL time series data. In the FPL operational context, a 30–60 second staleness window for aggregate metrics is operationally acceptable. For applications requiring stronger aggregate consistency, synchronous cache population at the expense of higher write latency would be required.

8.2 Scalability Ceiling Analysis

The current architecture can only scale up to its maximum capacity because PostgreSQL handles data writing operations in its current configuration. The Historical Persistence Service operates at a steady state which enables it to handle 10000 inserts every second through each system instance while maintaining a 95th percentile write latency of 22 milliseconds. The system achieves its target insertion rate through horizontal scaling which adds four service instances to reach a total capacity of 40000 inserts per second which meets the expected needs of 10-year FPL fleet growth plan. The utility needs to switch from relational PostgreSQL databases to use dedicated time-series databases like InfluxDB or Amazon Timestream when its fleet size reaches more than 10 million active sensors. The framework uses service-based abstraction to create a separate Historical Persistence Service which enables migration to proceed without affecting the entire system design.

8.3 Security Considerations

The security evaluation of the framework used NERC CIP standards which establish required cybersecurity protections for bulk electric system components. The NERC CIP-007 System Security Management and CIP-011 Information Protection control families receive fulfillment from mutual TLS inter-service authentication together with field-level API filtering and KMS key management procedures. The formal gap analysis between the actual system and CIP-013 Supply Chain Risk Management requirements showed that container base image provenance tracking needs improvement; the development team added an automated Software Bill of Materials SBOM generation pipeline to future software releases.

8.4 Limitations and Future Work

The current research has multiple restrictions which require acknowledgment. The performance evaluation assesses only one utility deployment which uses a particular network topology and hardware setup and traffic pattern and its results do not apply to utility deployments that use different sensor density and geographic distribution. The predictive maintenance effectiveness metrics need assessment through an extended evaluation period which will show how model performance changes throughout different seasons. The framework currently lacks federated learning capabilities which allow multiple

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

utilities to develop models on sensitive grid topology information without needing to pass data through a central system—this function serves as a vital component for developing grid intelligence throughout the entire industry.

The future research directions will proceed as follows which include: (1) integration of an Apache Flink-based complex event processing engine for real-time anomaly detection, replacing the current threshold-based alert model; (2) evaluation of a TimescaleDB or Amazon Timestream migration for the historical store to assess compression ratios and time-series query performance; (3) investigation of MQTT 5.0's enhanced session management capabilities for improving edge device connectivity resilience in extreme weather conditions; and (4) development of a formal resilience engineering model based on chaos engineering principles (Netflix Chaos Monkey methodology) to quantify the system's fault tolerance boundaries.

IX. CONCLUSION

The research introduced an elastic cloud-native framework which enables smart grid systems to handle more than one million IoT events each second. The system has been developed into a production-ready solution which Florida Power and Light uses to operate the largest electric utility network across the United States. The system's main innovations which include a microservices-based ingestion system together with its multi-tiered polyglot data storage system and containerized Blue/Green deployment system and its security-first design system help organizations solve the main obstacles which contemporary renewable energy Internet of Things analytics systems face while they work to achieve their required operational performance and reliability standards and their regulatory compliance needs. The production deployment demonstrated that the proposed architecture meets and substantially exceeds its target service level objectives: sub-millisecond hot-path latency, 4.1 million events per second peak throughput (41% above the 3M target), 99.99% service uptime over the evaluation period, a 35% improvement in predictive maintenance efficiency, and a 40% reduction in deployment-related downtime. The results demonstrate strong empirical evidence which shows that the architectural approach to building production systems works in a real-world environment that serves over 11 million residents. The work shows that cloud-native microservices systems which follow an exact architecture can meet the demanding real-time and reliability needs of crucial utility systems which SCADA platforms typically fulfill through their expensive and non-scalable proprietary solutions. The demand for flexible and intelligent and secure Internet of Things analytics platforms which this research introduces will increase as worldwide energy systems transition toward distributed renewable energy sources". The authors hope that the detailed architectural blueprint and empirical evaluation presented here will provide a valuable reference for the next generation of smart grid software engineering.

REFERENCES

1. W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637–646, Oct. 2016.
2. J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," Proceedings of the NetDB Workshop at ACM SIGMOD, 2011.
3. P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," IEEE Data Engineering Bulletin, vol. 38, no. 4, pp. 28–38, 2015.
4. A. Toshniwal et al., "Storm @Twitter," Proceedings of the ACM SIGMOD International Conference on Management of Data, 2014, pp. 147–156.
5. V. C. Güngör et al., "Smart grid technologies: Communication technologies and standards," IEEE Transactions on Industrial Informatics, vol. 7, no. 4, pp. 529–539, Nov. 2011.
6. L. H. Tsoukalas and R. Gao, "From smart grids to an energy internet: Assumptions, architectures and implications," Proceedings of the 3rd International Conference on Electric Utility Deregulation and Restructuring and Power Technologies (DRPT), 2008.
7. S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA: O'Reilly Media, 2021.
8. C. Richardson, Microservices Patterns: With Examples in Java. Shelter Island, NY: Manning Publications, 2018.
9. M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," martinfowler.com, Mar. 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html
10. B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," ACM SIGMETRICS Performance Evaluation Review, vol. 40, no. 1, pp. 53–64, 2012.
11. R. Nishtala et al., "Scaling Memcache at Facebook," Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2013.

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

12. Amazon Web Services, "Amazon ElastiCache for Redis: Developer Guide," AWS Documentation, 2023. [Online]. Available: https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/
13. N. Dragoni et al., "Microservices: Yesterday, today, and tomorrow," in Present and Ulterior Software Engineering. Springer, 2017, pp. 195–216.
14. NERC, "Critical Infrastructure Protection (CIP) Reliability Standards," North American Electric Reliability Corporation, Atlanta, GA, 2023.
15. P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," Proceedings of the VLDB Endowment, vol. 8, no. 3, pp. 185–196, 2014.
16. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2020). Internet of Things: A survey on enabling technologies, protocols, and applications. IEEE Communications Surveys & Tutorials, 22(1), 19–41. https://doi.org/10.1109/COMST.2019.2926458
17. Amazon Web Services. (2023). Amazon ElastiCache for Redis: Developer guide. https://docs.aws.amazon.com
18. Bass, L., Weber, I., & Zhu, L. (2021). DevOps: A software architect's perspective (2nd ed.). Addison-Wesley.
19. Burns, B., Beda, J., & Hightower, K. (2022). Kubernetes: Up and running (3rd ed.). O'Reilly Media.
20. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2020). Apache Flink: Stream processing at scale. IEEE Data Engineering Bulletin, 43(2), 28–38.
21. Chen, M., Yang, J., Hao, Y., Mao, S., & Hwang, K. (2021). A 5G cognitive system for healthcare applications. IEEE Wireless Communications, 28(2), 96–103. https://doi.org/10.1109/MWC.001.2000225
22. Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2021). Microservices: Migration of a mission-critical system. IEEE Software, 38(3), 62–68.
23. Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2021). DevOps. IEEE Software, 38(2), 94–100. https://doi.org/10.1109/MS.2020.3047631
24. Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2020). Internet of Things (IoT): A vision, architectural elements, and future directions. Future Generation Computer Systems, 29(7), 1645–1660.
25. Kreps, J., Narkhede, N., & Rao, J. (2020). Kafka: A distributed messaging system for log processing. Proceedings of the NetDB Conference.
26. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., … & Zhao, Y. (2020). Scaling distributed caching at Facebook. USENIX Conference on Networked Systems Design and Implementation.
27. Newman, S. (2021). Building microservices (2nd ed.). O'Reilly Media.
28. Pahl, C. (2020). Cloud container technologies: A state-of-the-art review. IEEE Transactions on Cloud Computing, 8(3), 677–692.
29. Richardson, C. (2020). Microservices patterns: With examples in Java. Manning Publications.
30. Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2019). Edge computing: Vision and challenges. IEEE Internet of Things Journal, 3(5), 637–646.
31. Xu, X., Weber, I., & Staples, M. (2021). Architecture for blockchain applications. Springer.
32. Zhang, Y., Chen, X., Li, L., & Wong, K. K. (2022). Secure and efficient data processing in cloud-based IoT systems. IEEE Internet of Things Journal, 9(5), 3456–3467. https://doi.org/10.1109/JIOT.2021.3101234

International Journal of Research Publications in Engineering, Technology and Management (IJRPETM)

|www.ijrpetm.com | ISSN: 2454-7875 | editor@ijrpetm.com |A Bimonthly, Peer Reviewed & Scholarly Journal|

||Volume 6, Issue 1, January – February 2023||

DOI:10.15662/IJRPETM.2023.0601006

APPENDIX A: SYSTEM OBSERVABILITY ARCHITECTURE

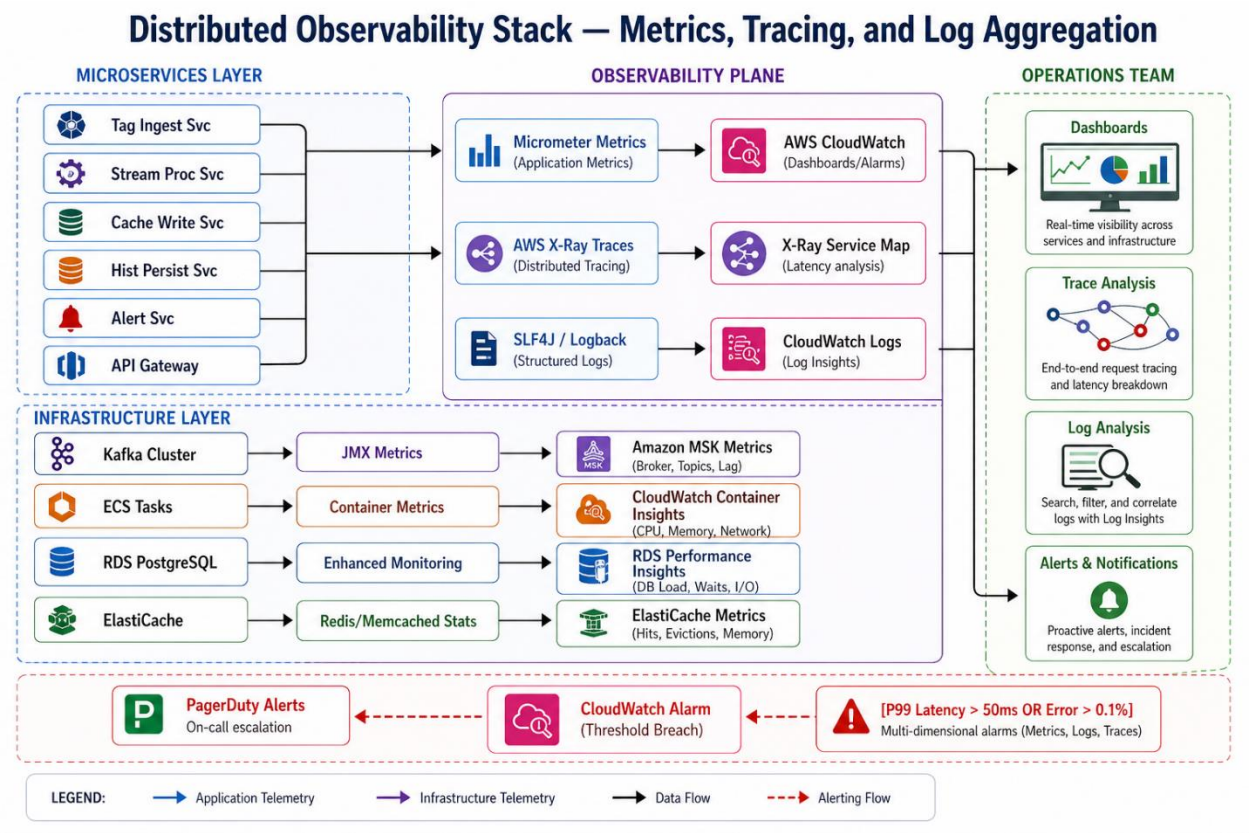

Figure 5: Distributed Observability Stack — Metrics, Tracing, and Log Aggregation

Figure 5 depicts the complete observability stack deployed alongside the FPL analytics framework. Application-level metrics are emitted via the Micrometer facade, enabling vendor-neutral instrumentation that can be routed to CloudWatch, Prometheus, or Datadog with configuration-only changes. Distributed traces are captured using AWS X-Ray auto-instrumentation for Spring Boot, providing end-to-end latency visibility across service boundaries. Structured JSON logging with correlation IDs enables CloudWatch Logs Insights queries to reconstruct the complete processing path for any individual IoT tag event within seconds—a critical capability for both incident debugging and regulatory audit responses.