FORGE: Federated Operations Research & Grid Environment

A DoD-Compatible Real-Time Multi-Sensor Fusion Platform for Missile Defense Simulation

StsGym Engineering

April 2026

Contents

Section 1: Executive Summary

FORGE: Federated Operations Research & Grid Environment

A DoD-Compatible Real-Time Multi-Sensor Fusion Platform for Missile Defense Simulation

---

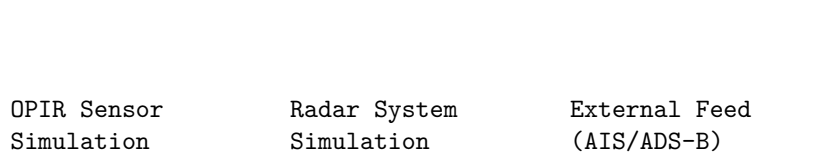
Overview

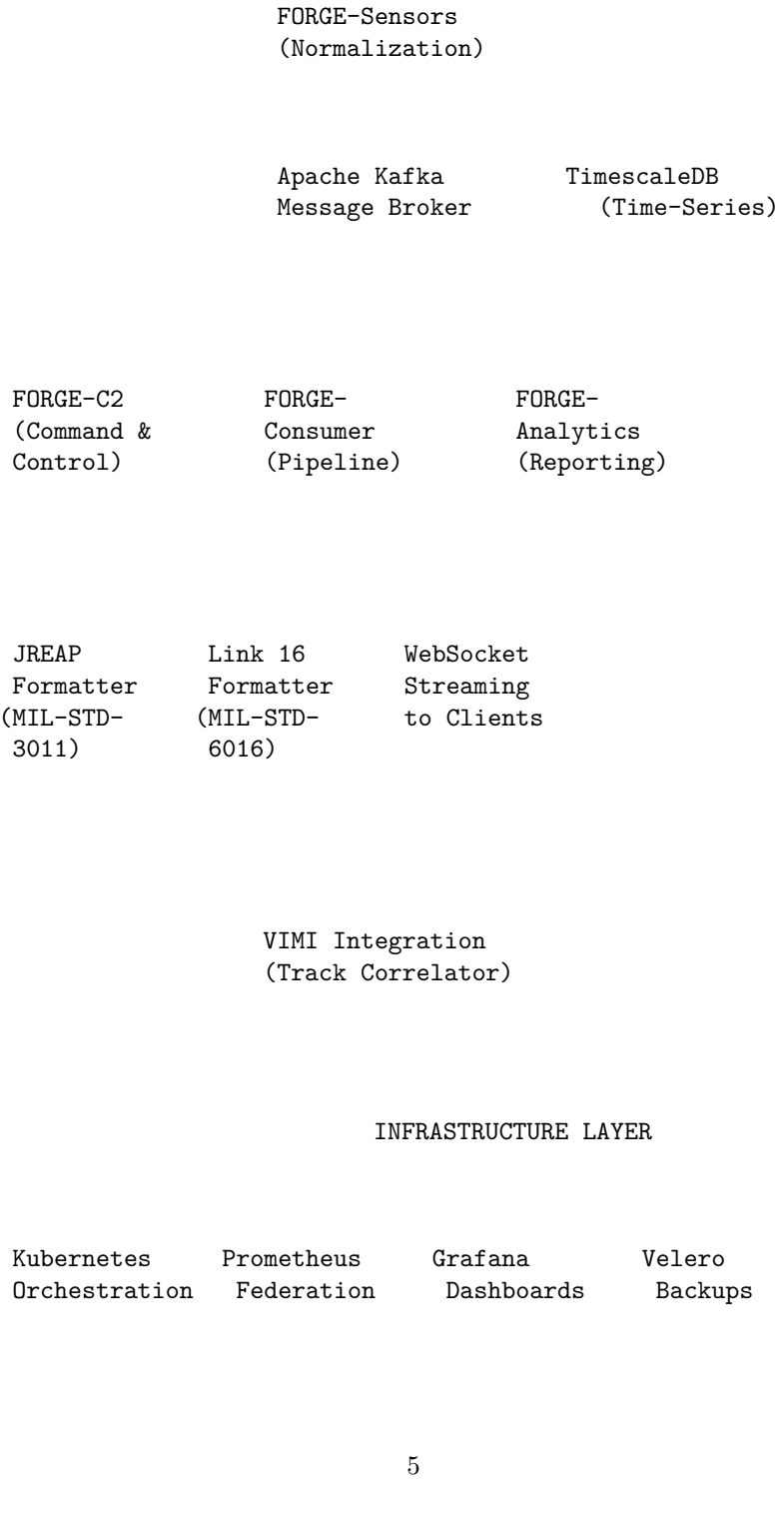
FORGE (Federated Operations Research & Grid Environment) is an integrated simulation platform for missile defense research, training, and operational planning. Built with DoD interoperability requirements, FORGE provides real-time multi-sensor fusion, track correlation, threat assessment, and engagement coordination through a modular, cloud-native architecture.

The system addresses a critical need: a realistic, standards-compliant environment integrating diverse sensor simulations, processing track data at scale, and coordinating defensive responses—while adhering to military messaging standards including JREAP (MIL-STD-3011) and Link 16 (MIL-STD-6016).

System Architecture

FORGE implements a distributed microservices architecture designed for high-throughput, low-latency track processing. The architecture separates concerns into distinct components that communicate via Apache Kafka message streams, enabling horizontal scaling and fault tolerance.




FORGE-Sensors
(Normalization)
Apache Kafka
Message Broker
TimescaleDB
(Time-Series)
FORGE-C2
(Command &
Control)
FORGE-
Consumer
(Pipeline)
FORGE-
Analytics
(Reporting)
JREAP
Formatter
(MIL-STD-
3011)
Link 16
Formatter
(MIL-STD-
6016)
WebSocket
Streaming
to Clients
VIMI Integration
(Track Correlator)
INFRASTRUCTURE LAYER
Kubernetes
Orchestration
Prometheus
Federation
Grafana
Dashboards
Velero
Backups

Core Components

FORGE-C2: Command and Control

FORGE-C2 serves as the central coordination hub for real-time track management, threat assessment, and engagement coordination. It implements JREAP for beyond-line-of-sight communications and Link 16 tactical data link formatting for DoD interoperability.

Key capabilities include JREAP-A/B/C message handling for satellite, line-of-sight, and IP network transports; Link 16 message formatting (J2, J3, J5, J7, J9 series); WebSocket streaming for real-time client updates; and alert generation for threat notifications.

FORGE-Sensors: Multi-Sensor Data Ingestion

FORGE-Sensors provides a unified interface for diverse sensor simulations including OPIR satellite simulations for boost-phase detection, radar system simulations for mid-course and terminal phase tracking, and external data feeds (AIS/ADS-B). The component normalizes incoming data into a common track format before publishing to Kafka.

FORGE-Consumer: Data Pipeline

FORGE-Consumer manages the data processing pipeline, consuming normalized sensor data from Kafka for track correlation, state estimation, position prediction, and historical archival to TimescaleDB.

Apache Kafka: Event Streaming Backbone

Kafka provides the distributed event streaming backbone, enabling decoupled component design with independent scaling, high-throughput track processing with sub-second latency, and message persistence for replay and recovery.

TimescaleDB: Time-Series Storage

TimescaleDB provides PostgreSQL-compatible time-series storage optimized for high-volume, timestamped track data, enabling historical analysis, simulation replay, and trend analysis.

Military Standards Compliance

FORGE implements critical military messaging standards to ensure interoperability with existing DoD infrastructure:

JREAP (MIL-STD-3011)

Type	Transport	Application
JREAP-A	Satellite (SATCOM)	Beyond-line-of-sight tracking for distributed operations
JREAP-B	Line-of-Sight (LOS)	Direct range extension for tactical units
JREAP-C	IP Network (TCP/UDP)	Ground network integration and simulation environments

Link 16 (MIL-STD-6016)

Key message series include J2 (track management), J3 (air tracks), J5 (electronic combat), J7 (weapon coordination), and J9 (engagement status).

Infrastructure Platform

FORGE operates on Kubernetes with Prometheus federation for metrics, Grafana for dashboards, and Velero for backups—ensuring high availability, scalability, and disaster recovery capabilities.

Key Capabilities

Track Correlation

FORGE fuses track data from multiple sensor sources into a unified operational picture, associating observations from OPIR satellites, radar systems, and external feeds to maintain coherent track identities.

Threat Assessment

Real-time threat assessment algorithms evaluate incoming tracks against threat profiles, prioritizing engagement decisions based on trajectory analysis, launch point attribution, and impact prediction.

Engagement Coordination

FORGE-C2 provides coordination for defensive response planning: weapon-target pairing optimization, engagement zone management, intercept probability calculations, and kill assessment integration.

Deployment Status

FORGE is operational on the miner infrastructure node (207.244.226.151) with integration to the VIMI track correlator. Core components—FORGE-C2, FORGE-Sensors, Kafka, and TimescaleDB—are deployed and functional. JREAP handler implementation and authentication/authorization layers are in active development.

Strategic Value

FORGE addresses the critical need for realistic, standards-compliant missile defense simulation. By combining cloud-native architecture with military messaging standards, FORGE enables realistic training environments, algorithm development and testing, interoperability validation against DoD standards, scalable research platforms, and comprehensive after-action analysis.

The platform serves as a foundation for continued research in missile defense technology, sensor fusion algorithms, and engagement coordination strategies.

Document Organization

1. Executive Summary (this section)
2. System Architecture and Component Design
3. Data Flow and Messaging Standards
4. Deployment and Infrastructure
5. Integration Patterns and APIs
6. Performance and Scalability
7. Future Development Roadmap

Document Version: 1.0
Classification: UNCLASSIFIED//FOR OFFICIAL USE ONLY
Last Updated: April 2026

Section 2: System Architecture

Overview

FORGE implements a layered, event-driven architecture designed for high-throughput sensor data processing with real-time track correlation and command-and-control capabilities. The system decomposes naturally into five distinct layers: sensor simulation, messaging, processing, storage, and infrastructure. This separation enables independent scaling, clear ownership

boundaries, and the flexibility to replace or upgrade components without disrupting the overall system.

The architecture follows a pipes-and-filters pattern: sensor simulators produce detection events, Kafka provides the reliable transport backbone, processing services consume and transform these streams, and persistent storage captures both time-series telemetry and relational state. Kubernetes orchestrates the deployment, while Prometheus and Grafana provide observability across all layers.

2.1 Sensor Layer

The sensor layer comprises two categories of simulators: OPIR (Overhead Persistent Infrared) and Radar. Each simulator produces realistic detection data conforming to actual sensor characteristics, enabling training and testing without access to classified systems.

OPIR Simulators

SBIRS-GEO (Space-Based Infrared System - Geostationary)

The SBIRS-GEO simulator models high-Earth orbit infrared sensors providing continuous hemispheric coverage. Key characteristics include:

- **Scan Pattern**: Full-disk scanning with configurable dwell times
- **Detection Bands**: Multiple infrared bands (SWIR, MWIR) with realistic atmospheric penetration modeling
- **Sensitivity**: Configurable detection thresholds supporting booster plume detection from boost phase through midcourse
- **Data Rate**: Continuous stream at 1-10 Hz depending on scanning mode
- **Output**: Azimuth, elevation, intensity, timestamp, confidence metrics

DSP (Defense Support Program)

The legacy DSP simulator provides backward compatibility and comparison scenarios:

- **Coverage**: Geosynchronous constellation with known coverage gaps
- **Detection**: Lower resolution than SBIRS but proven reliability characteristics
- **Latency**: Higher processing latency, modeled accurately for realistic scenario timing
- **Use Case**: Legacy system training, degradation mode exercises

NEXTGEN OPIR

The next-generation OPIR simulator models advanced capabilities:

- **Enhanced Sensitivity**: Improved detection of dimmer signatures

- **Multi-Spectral Fusion**: Combined band processing with enhanced clutter rejection
- **Reduced Latency**: Faster data processing pipelines
- **Hyperspectral**: Additional spectral bands for improved target characterization

Radar Simulators

UEWR (Upgraded Early Warning Radar)

The UEWR simulator models large phased-array radars providing long-range detection:

- **Frequency**: UHF band with associated propagation characteristics
- **Coverage**: 240° azimuth coverage, configurable elevation limits
- **Range**: 5,000+ km detection range with realistic detection probability curves
- **Track Capacity**: Supports hundreds of simultaneous tracks
- **Modes**: Surveillance and track modes with different scan patterns

TPY-2 (AN/TPY-2)

The TPY-2 simulator models the transportable X-band radar:

- **Frequency**: X-band with high range resolution
- **Coverage**: 120° azimuth coverage, pencil-beam electronic steering
- **Range**: Extended discrimination range with high-resolution tracking
- **Discrimination**: Capable of distinguishing closely-spaced objects
- **Deployment**: Models both forward-based and THAAD battery configurations

SBX (Sea-Based X-Band Radar)

The SBX simulator models the floating X-band platform:

- **Mobility**: Simulates deployment repositioning during scenarios
- **Power**: Higher peak power than land-based equivalents
- **Coverage**: Full hemispheric coverage from stabilized platform
- **Weather Sensitivity**: Models sea-state effects on tracking stability

Sensor Data Format

All simulators produce standardized JSON messages conforming to a common schema:

```
{
  "sensor_id": "SBIRS-GEO-01",
  "sensor_type": "OPIR",
  "timestamp": "2026-04-10T01:05:32.123Z",
  "detection": {
    "track_id": "TGT-001-A",
```

```
    "azimuth": 45.67,
    "elevation": 12.34,
    "intensity": 0.847,
    "confidence": 0.92
  },
  "metadata": {
    "mode": "TRACK",
    "scan_pattern": "STEP_SCAN"
  }
}
```

2.2 Messaging Layer

Kafka Cluster Architecture

FORGE uses Apache Kafka as its central message bus, operating in KRaft mode (Kafka Raft metadata) which eliminates the need for a separate ZooKeeper ensemble. This simplifies deployment and reduces operational complexity while maintaining full Kafka capabilities.

Cluster Configuration:

- **Brokers**: 3-node cluster for production, 1-node for development
- **Replication Factor**: 3 (production), 1 (development)
- **Retention**: 7 days for detection topics, 30 days for correlated tracks
- **Compression**: LZ4 for balance of CPU and compression ratio

Topic Architecture

`opir-detections`

Raw detection events from all OPIR simulators. High-volume topic receiving continuous streams from SBIRS-GEO, DSP, and NEXTGEN simulators.

- **Partitions**: 12 (scaled to sensor count)
- **Retention**: 7 days
- **Throughput**: ~50,000 messages/second peak
- **Consumers**: FORGE-Consumer instances for raw data capture

`radar-tracks`

Track data from radar simulators. Lower volume but higher per-message complexity due to multi-target tracking.

- **Partitions**: 6
- **Retention**: 7 days
- **Throughput**: ~5,000 messages/second peak
- **Consumers**: FORGE-Consumer, FORGE-C2 for correlation

`correlated-tracks`

Fused track data combining OPIR and radar sources. This is the primary output of the correlation process.

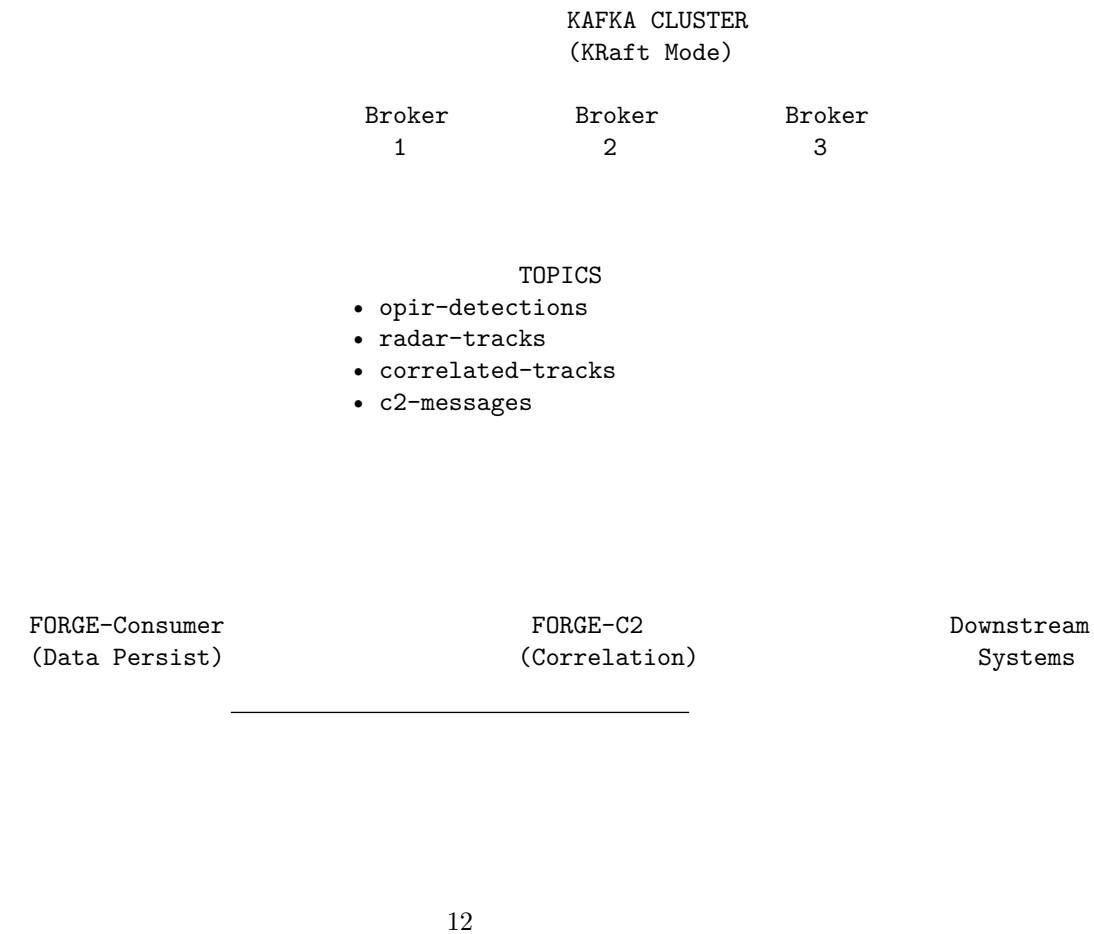
- **Partitions**: 6
- **Retention**: 30 days
- **Throughput**: ~2,000 messages/second
- **Consumers**: FORGE-C2, downstream displays, analytics

`c2-messages`

Command and control messages for system direction and scenario control.

- **Partitions**: 3
- **Retention**: 30 days
- **Throughput**: Variable, typically low
- **Consumers**: FORGE-C2, simulators (for mode changes)

Data Flow Diagram



2.3 Processing Layer

FORGE-Consumer

FORGE-Consumer is a Kafka consumer group responsible for persisting all sensor data to TimescaleDB. It implements:

- **Consumer Group Pattern**: Multiple instances for horizontal scaling and fault tolerance
- **Batch Processing**: Configurable batch sizes (default 1000 records) for throughput optimization
- **Offset Management**: At-least-once delivery semantics with idempotent inserts
- **Schema Validation**: Rejects malformed messages, logs for investigation
- **Dead Letter Queue**: Failed messages routed to error topic for analysis

Key Operations:

1. Subscribe to `opir-detections` and `radar-tracks`
2. Batch accumulate within time window (configurable, default 100ms)
3. Transform to TimescaleDB row format
4. Execute batch insert with conflict handling
5. Commit offsets after successful persistence
6. Emit metrics to Prometheus endpoint

FORGE-C2

FORGE-C2 handles track management, correlation logic, and message formatting for downstream systems. It maintains the authoritative state of all tracks in the scenario.

Track Correlation Engine:

The correlation engine applies multi-source fusion algorithms to combine OPIR and radar tracks into unified track objects:

```
OPIR Detections          Radar Tracks
(az/el/intensity)       (range/az/el)




      CORRELATION ENGINE
  • Temporal Alignment
  • Spatial Association
  • Confidence Weighting
  • Track ID Assignment
```

```
                Correlated
                Track Object
                (fused state)
```

```
    TimescaleDB             c2-messages
    (historical)            (real-time)
```

Message Formatting:

FORGE-C2 produces standardized messages for downstream consumers:

- **Track Update Messages**: Position, velocity, confidence, source correlation
- **Track Drop Messages**: When tracks age out or are manually dropped
- **Alert Messages**: Threshold violations, anomalous behavior
- **Heartbeat Messages**: System health indicators

2.4 Storage Layer

TimescaleDB

TimescaleDB extends PostgreSQL with native time-series capabilities, providing:

- **Hypertable Partitioning**: Automatic time-based partitioning for efficient queries
- **Continuous Aggregates**: Pre-computed rollups for dashboard queries
- **Retention Policies**: Automated data lifecycle management
- **Compression**: Columnar compression for historical data

Schema Design:

```
-- Detections hypertable
CREATE TABLE opir_detections (
    time        TIMESTAMPTZ NOT NULL,
    sensor_id   TEXT NOT NULL,
    track_id    TEXT,
    azimuth     DOUBLE PRECISION,
    elevation   DOUBLE PRECISION,
    intensity   DOUBLE PRECISION,
    confidence  DOUBLE PRECISION,
    metadata    JSONB
);
```

```
SELECT create_hypertable('opir_detections', 'time',
    chunk_time_interval => INTERVAL '1 day');

-- Continuous aggregate for hourly stats
CREATE MATERIALIZED VIEW opir_hourly
WITH (timescaledb.continuous) AS
SELECT time_bucket('1 hour', time) AS bucket,
       sensor_id,
       COUNT(*) AS detection_count,
       AVG(confidence) AS avg_confidence
FROM opir_detections
GROUP BY bucket, sensor_id;
```

PostgreSQL (Relational Data)

Standard PostgreSQL tables store configuration, scenario definitions, and reference data:

- **Scenario Definitions**: Scenario parameters, timelines, objectives
- **Sensor Registry**: Sensor configurations, capabilities, locations
- **User Preferences**: Display settings, alert thresholds
- **Audit Logs**: Configuration changes, administrative actions

Connection Pooling:

Both TimescaleDB and PostgreSQL use PgBouncer for connection pooling, reducing connection overhead and improving throughput under load.

2.5 Infrastructure Layer

Kubernetes Deployment

FORGE runs on a bare-metal Kubernetes cluster with two node classes:

`gms-control-plane` Node:

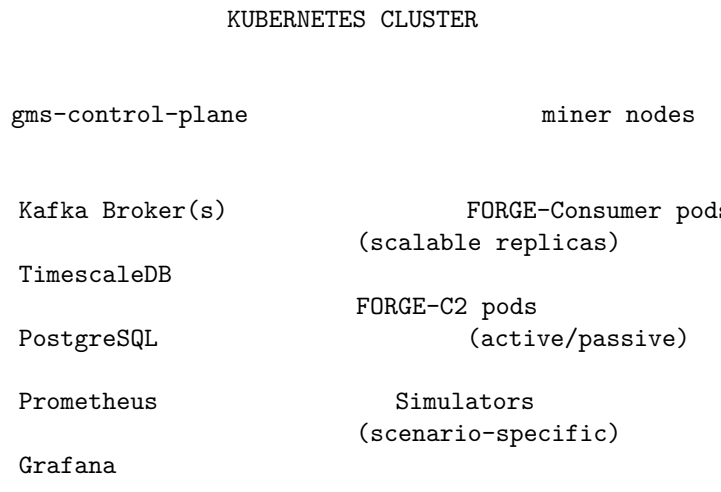
- Control plane workloads
- Kafka brokers (production)
- PostgreSQL/TimescaleDB primary
- Grafana, Prometheus server
- Network services (ingress, DNS)

`miner` Nodes:

- Worker nodes for compute-intensive workloads
- FORGE-Consumer instances (scalable)
- FORGE-C2 instances

- Kafka consumers
- Simulators during exercises

Deployment Topology:



Prometheus Federation

A federated Prometheus architecture provides comprehensive observability:

- **Central Prometheus**: Located on `gms-control-plane`, scrapes all services
- **Node Exporters**: Host-level metrics (CPU, memory, disk, network)
- **Service Dashboards**: Per-component dashboards for deep dives
- **Alertmanager**: Routes alerts to appropriate channels

Key Metrics Collected:

- Kafka: Consumer lag, message rates, partition distribution
- PostgreSQL: Connection count, query latency, replication lag
- FORGE services: Processing latency, message throughput, error rates
- Infrastructure: CPU, memory, disk I/O, network throughput

Grafana Dashboards

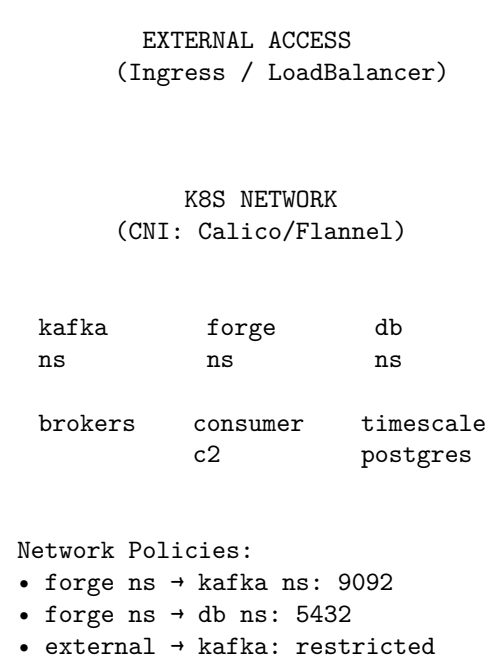
Pre-built dashboards provide real-time visibility:

1. **System Overview**: Cluster health, resource utilization
2. **Kafka Metrics**: Producer/consumer rates, lag visualization

3. **Database Metrics**: Query performance, connection pooling
4. **FORGE Metrics**: Track rates, correlation latency, error rates
5. **Scenario Dashboard**: Real-time track visualization (operator view)

2.6 Network Topology

The network architecture follows a flat model within the Kubernetes cluster, with network policies providing isolation where required:



Network Policies

Kubernetes NetworkPolicy resources enforce:

- **Namespace Isolation**: Cross-namespace traffic limited to required paths
- **Service-to-Database**: Only authorized services may connect to PostgreSQL/TimescaleDB
- **External Restrictions**: Kafka brokers not directly accessible from outside cluster
- **Ingress Control**: All external traffic through designated ingress controller

---

2.7 Scalability Considerations

The architecture supports horizontal scaling at multiple points:

Component	Scaling Method	Trigger
Kafka Brokers	Add brokers	Storage capacity, throughput
Kafka Partitions	Increase partitions	Consumer parallelism
FORGE-Consumer	Add replicas	Consumer lag threshold
FORGE-C2	Active/passive	High availability
TimescaleDB	Read replicas	Query load
Simulators	Per-scenario deployment	Scenario requirements

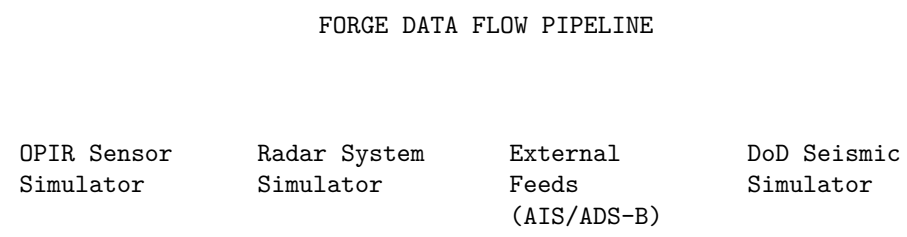
The Kafka-based messaging layer provides natural load buffering: during traffic spikes, messages accumulate in topics, allowing consumers to process at sustainable rates without data loss. This decoupling of producers and consumers is fundamental to the architecture's resilience.

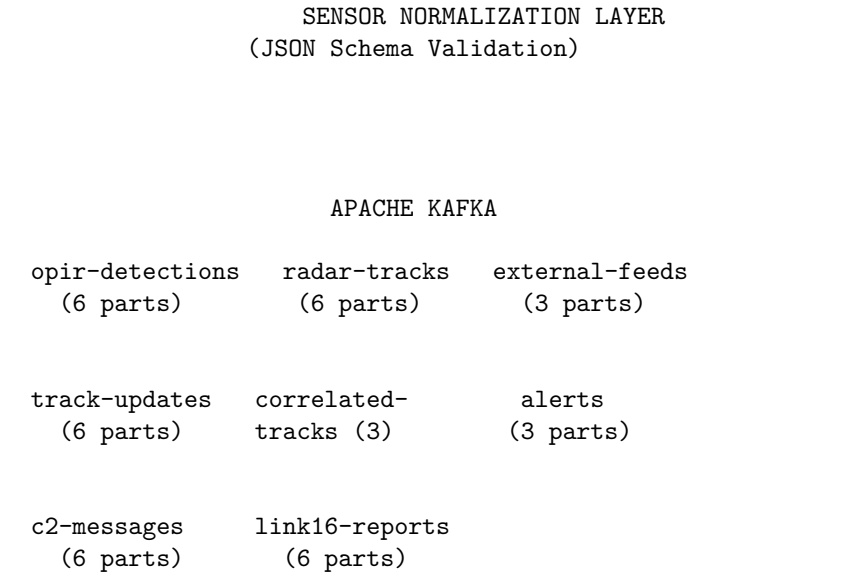
Section 3: Data Flow & Messaging Standards

Overview

FORGE's data pipeline processes sensor data through a series of stages: ingestion, normalization, correlation, storage, and distribution. All data flows through Apache Kafka, which provides durability, replayability, and horizontal scaling. Military messaging standards (JREAP and Link 16) ensure DoD interoperability.

Data Flow Architecture





Kafka Topics

Topic Configuration

Topic	Partitions	Replication	Min ISR	Purpose
opir-detections	6	1	1	Space-based IR detections
radar-tracks	6	1	1	Ground-based radar tracks
track-updates	6	1	1	Track position updates
correlated-tracks	3	1	1	Fused multi-source tracks
c2-messages	6	1	1	Command & control messages
link16-reports	6	1	1	Link 16 formatted messages
jreap-messages	6	1	1	JREAP formatted messages
alerts	3	1	1	Threat alerts

Topic Creation Script

```
#!/bin/bash
# Create Kafka topics for FORGE

TOPICS=(
  "opir-detections:6:1:1"
  "radar-tracks:6:1:1"
  "track-updates:6:1:1"
  "correlated-tracks:3:1:1"
  "c2-messages:6:1:1"
  "link16-reports:6:1:1"
  "jreap-messages:6:1:1"
  "alerts:3:1:1"
)

for TOPIC_SPEC in "${TOPICS[@]}"; do
  TOPIC=$(echo $TOPIC_SPEC | cut -d: -f1)
  PARTITIONS=$(echo $TOPIC_SPEC | cut -d: -f2)
  REPLICATION=$(echo $TOPIC_SPEC | cut -d: -f3)
  MIN_ISR=$(echo $TOPIC_SPEC | cut -d: -f4)

  kafka-topics.sh --create \
    --topic "$TOPIC" \
    --partitions "$PARTITIONS" \
    --replication-factor "$REPLICATION" \
    --config min.insync.replicas="$MIN_ISR" \
    --bootstrap-server kafka:9092
done
```

Access Points

Environment	Address
Internal (K8s)	`kafka.forge.svc.cluster.local:9092`
External (NodePort)	`207.244.226.151:30721`

Normalized Track Format

All sensor data is normalized to a common JSON format before publishing to Kafka:

OPIR Detection Format


```json
{
  "detection_id": "OPIR-SBIRS-GEO-1-1775013473732455512",
  "timestamp": "2026-04-01T03:17:53.732455512Z",
  "sensor_id": "SBIRS-GEO-1",
  "sensor_type": "SBIRS-GEO",
  "orbital_slot": "GEO-95W",
  "target_type": "ICBM",
  "launch_point": {
    "latitude": 39.0,
    "longitude": 127.0,
    "altitude": 0
  },
  "impact_point": {
    "latitude": 42.5,
    "longitude": 135.0,
    "altitude": 0
  },
  "band": "SWIR",
  "radiance": 8.04,
  "temperature": 2500.5,
  "intensity": 0.85,
  "confidence": 0.98,
  "track_id": "TRK-1234",
  "track_phase": "BOOST",
  "velocity": {"x": 3.5, "y": 2.1, "z": 0.5},
  "position": {"x": 100.0, "y": -200.0, "z": 300.0},
  "signal_to_noise": 25.5,
  "clutter_level": 0.1
}
```

Radar Track Format


```
{
  "track_id": "RADAR-UEWR-BEALE-1775041685345701470",
  "timestamp": "2026-04-01T11:08:05.345701470Z",
  "radar_id": "UEWR-BEALE",
  "radar_type": "UEWR",
  "location": {
    "latitude": 39.1,
    "longitude": -121.4,
    "altitude": 0
  },
  "target_type": "IRBM",
  "range": 3138.83,
  "azimuth": 218.69,
  "elevation": 60.99,
  "velocity": 6.78,
  "heading": 74.65,
  "altitude": 316.49,
  "rcs": -11.15,
  "snr": 28.76,
  "range_rate": 2.89,
  "doppler": 1792.78,
  "track_quality": 9,
  "confidence": 0.79,
  "track_phase": "TRACKING",
  "discriminated_type": "DECOY",
  "discrimination_confidence": 0.80
}
```


Correlated Track Format


```
{
  "track_id": "TRK-1775013953",
  "timestamp": "2026-04-01T03:25:53.199583Z",
  "source_tracks": [
    "OPIR-SBIRS-GEO-1-123",
    "RADAR-UEWR-BEALE-456"
  ],
  "target_type": "ICBM",
  "threat_level": 5,
  "position": {"x": 100.0, "y": -200.0, "z": 300.0},
  "velocity": {"x": 3.5, "y": 2.1, "z": 0.5},
  "launch_point": {"latitude": 39.0, "longitude": 127.0, "altitude": 0},
  "impact_point": {"latitude": 42.5, "longitude": 135.0, "altitude": 0},
  "track_phase": "MIDCOURSE",
```

```
  "track_quality": 9,
  "sensor_count": 2,
  "confidence": 0.92,
  "discriminated_type": "RV",
  "discrimination_confidence": 0.88
}
```

Military Messaging Standards

JREAP (MIL-STD-3011)

JREAP (Joint Range Extension Applications Protocol) provides beyond-line-of-sight communications for tactical data links.

JREAP Message Types

Type	Transport	Use Case	Latency Budget
JREAP-A	Satellite (SATCOM)	Distributed operations	<500ms
JREAP-B	Line-of-Sight (LOS)	Direct range extension	<100ms
JREAP-C	IP Network (TCP/UDP)	Ground network/simulation	<50ms

JREAP Message Structure

```
                    JREAP MESSAGE FORMAT


   Header    Source    Track     Track Data
   (8 B)     ID (4B)   ID (4B)   (Variable)


  HEADER FORMAT:

   Version (1B)   Type (1B)      Length (2B)   Reserved
   0x01           A=1/B=2/C=3    Payload len   (4B)


  TRACK DATA (JREAP-C Example):

   Latitude       Longitude      Altitude       Velocity
   (4B, deg*1e7)  (4B, deg*1e7)  (4B, m*100)    (4B, m/s)
```

JREAP Handler Implementation (Go)

```go
// JREAP message types
type JREAPType int

const (
    JREAPA JREAPType = 1 // Satellite
    JREAPB JREAPType = 2 // Line-of-Sight
    JREAPC JREAPType = 3 // IP Network
)

type JREAPHeader struct {
    Version   byte
    Type      JREAPType
    Length    uint16
    Timestamp uint32
    SourceID  uint32
}

type JREAPMessage struct {
    Header    JREAPHeader
    TrackID   uint32
    Latitude  int32   // degrees * 1e7
    Longitude int32   // degrees * 1e7
    Altitude  int32   // meters * 100
    Velocity  int32   // m/s * 100
    Heading   uint16  // degrees * 100
    TrackType byte
    Threat    byte
}

// Encode JREAP-C message
func (m *JREAPMessage) Encode() ([]byte, error) {
    buf := new(bytes.Buffer)

    // Header
    binary.Write(buf, binary.BigEndian, m.Header.Version)
    binary.Write(buf, binary.BigEndian, byte(m.Header.Type))
    binary.Write(buf, binary.BigEndian, m.Header.Length)
    binary.Write(buf, binary.BigEndian, m.Header.Timestamp)
    binary.Write(buf, binary.BigEndian, m.Header.SourceID)

    // Track data
    binary.Write(buf, binary.BigEndian, m.TrackID)
```

```
    binary.Write(buf, binary.BigEndian, m.Latitude)
    binary.Write(buf, binary.BigEndian, m.Longitude)
    binary.Write(buf, binary.BigEndian, m.Altitude)
    binary.Write(buf, binary.BigEndian, m.Velocity)
    binary.Write(buf, binary.BigEndian, m.Heading)
    binary.Write(buf, binary.BigEndian, m.TrackType)
    binary.Write(buf, binary.BigEndian, m.Threat)

    return buf.Bytes(), nil
}
```

Link 16 (MIL-STD-6016)

Link 16 is the primary tactical data link for U.S. and allied forces, providing secure, jam-resistant communication.

Link 16 Message Series

Series	Purpose	Priority	FORGE Support
J2	Track Management	P0	Implemented
J3	Air Track	P0	Implemented
J5	Electronic Combat	P1	Planned
J7	Weapon Coordination	P1	Planned
J9	Engagement Status	P1	Planned

Link 16 J-Series Message Structure

```
                    LINK 16 J-SERIES FORMAT


  WORD 0 (Message Header):

   TSD     RI     SR      SRS    BIC     Word
   (1b)    (1b)   (1b)    (1b)   (1b)    Count


  WORD 1 (Track ID):

    Track Number (15b)   Track Quality (5b) + Status (12b)


  WORD 2 (Position):
```

```
    Latitude (15b)       Longitude (15b)
    (0.0054 deg resol)   (0.0054 deg resol)


  WORD 3 (Altitude/Velocity):

    Altitude (15b)       Speed (15b)
    (6.1m resol)         (1 kt resol)
```

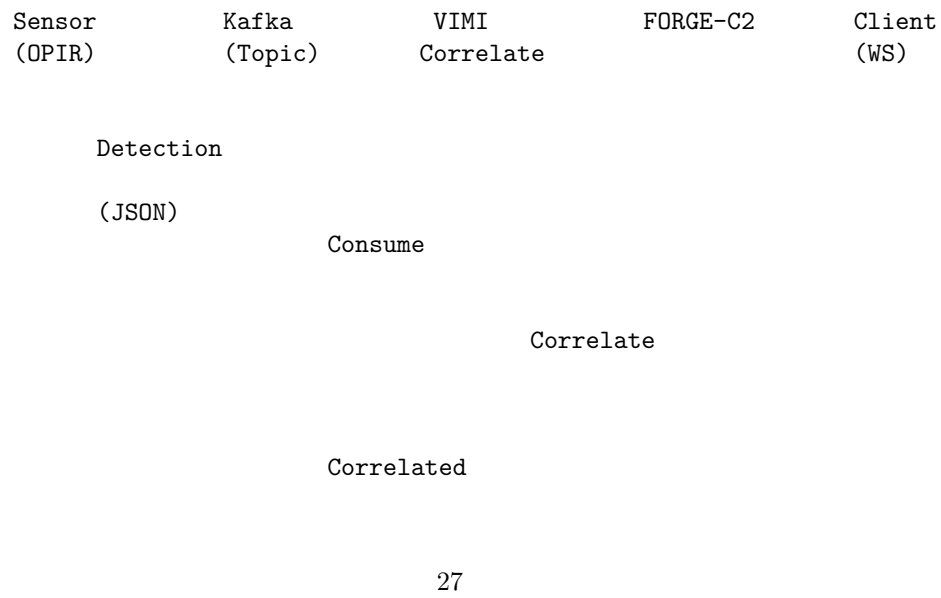
Link 16 Formatter Implementation (Go)

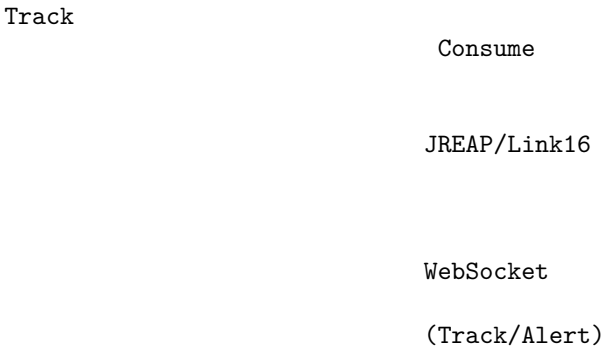
```go
// Link 16 message types
type Link16MessageType int

const (
    J2 Link16MessageType = 2  // Track Management
    J3 Link16MessageType = 3  // Air Track
    J5 Link16MessageType = 5  // Electronic Combat
    J7 Link16MessageType = 7  // Weapon Coordination
    J9 Link16MessageType = 9  // Engagement Status
)

type Link16Message struct {
    MessageType Link16MessageType
    TrackNumber uint16
    TrackQuality uint8
    Latitude    int32   // 0.0054 degree resolution
    Longitude   int32   // 0.0054 degree resolution
    Altitude    int32   // 6.1m resolution
    Speed       uint16  // 1 kt resolution
    Heading     uint16  // 0.088 degree resolution
    TrackType   uint8
    IFFMode     uint8
}

// Convert track to Link 16 J3 message
func TrackToJ3(track CorrelatedTrack) (*Link16Message, error) {
    return &Link16Message{
        MessageType:  J3,
        TrackNumber:  extractTrackNumber(track.TrackID),
        TrackQuality: uint8(track.TrackQuality),
        Latitude:     int32(track.Position.Lat / 0.0054),
```

```
        Longitude:    int32(track.Position.Lon / 0.0054),
        Altitude:     int32(track.Position.Alt / 6.1),
        Speed:        uint16(track.Velocity.Speed),
        Heading:      uint16(track.Velocity.Heading / 0.088),
        TrackType:     mapTrackType(track.TargetType),
        IFFMode:       0,
    }, nil
}

func mapTrackType(targetType string) uint8 {
    switch targetType {
    case "ICBM":
        return 1
    case "IRBM":
        return 2
    case "SRBM":
        return 3
    case "GLCM":
        return 4
    default:
        return 0 // Unknown
    }
}
```

Message Flow Sequence

Detection-to-Correlation Flow





Latency Budget

End-to-End Latency Requirements

Stage	Budget	Typical	Notes
Sensor → Kafka	<10ms	5ms	Network + serialization
Kafka → Consumer	<20ms	10ms	Poll interval + deserialization
Consumer → Database	<30ms	15ms	Batch insert (100 records)
Track Correlation	<50ms	25ms	Fusion algorithm
Alert Generation	<20ms	10ms	Threat assessment
JREAP/Link16 Format	<10ms	5ms	Message encoding
WebSocket Push	<10ms	5ms	Client notification
Total P95	**<100ms**	**75ms**	End-to-end

Latency Optimization

1. **Kafka batching**: 100 messages per batch
2. **TimescaleDB batch inserts**: 100 records per transaction
3. **WebSocket keepalive**: Prevent connection teardown
4. **Memory pooling**: Reuse buffers for encoding

Data Retention

Storage	Retention	Compression	Notes
Kafka	7 days	Snappy	Raw sensor data
TimescaleDB	90 days	Zstd	Compressed chunks
PostgreSQL	1 year	None	Track metadata

Storage	Retention	Compression	Notes
Backup	90 days	Velero	Full cluster backup

Summary

FORGE's data flow architecture provides:

- **Unified normalization** of multi-source sensor data
- **Apache Kafka** as the durable message backbone
- **Military standards compliance** (JREAP, Link 16)
- **Sub-100ms end-to-end latency** for critical tracks
- **Scalable processing** through partition-based parallelism

---

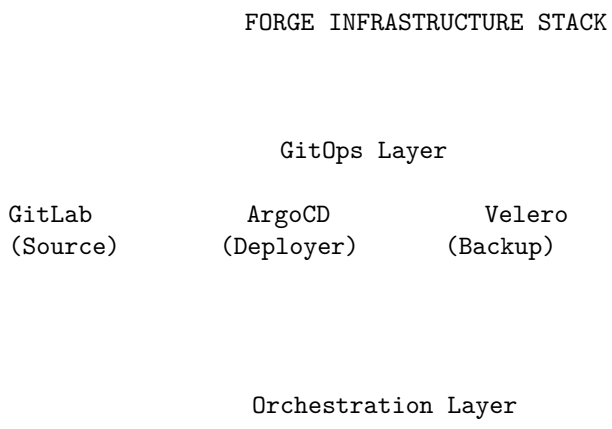
Document Version: 1.0 Last Updated: April 2026

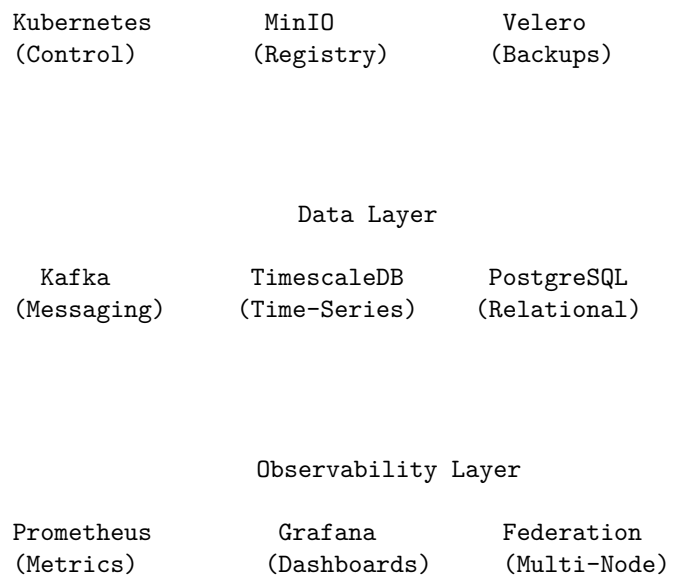
Section 4: Deployment & Infrastructure

Overview

FORGE's deployment architecture leverages Kubernetes for orchestration, GitOps for configuration management, and a layered infrastructure approach that separates concerns across compute, storage, and networking tiers. This section documents the infrastructure components, deployment procedures, and operational considerations for FORGE systems.

Infrastructure Architecture





Kubernetes Cluster Configuration

Cluster Topology

Node	Role	IP Address	Services
gms-control-plane	Control Plane	10.0.0.x	API Server, etcd, scheduler
darth	Worker	100.92.94.92 (Tailscale)	Prometheus federation
trooper2	Worker	100.123.159.17 (Tailscale)	Development, VM hosting
miner	Worker	100.116.245.125 (Tailscale)	Kafka, PostgreSQL, ArgoCD

Namespaces

Namespace	Purpose	Components
forge	FORGE services	forge-c2, forge-consumer, forge-sensors

Namespace	Purpose	Components
patroni	Database	PostgreSQL HA cluster
monitoring	Observability	Prometheus, Grafana, node-exporter
argocd	GitOps	ArgoCD controller, server, dex
velero	Backup	Velero controller, MinIO storage
kafka	Messaging	Kafka brokers, KRaft controllers

Resource Allocation

Component	CPU Request	CPU Limit	Memory Request	Memory Limit
forge-c2	500m	2000m	512Mi	2Gi
forge-consumer	1000m	4000m	1Gi	4Gi
kafka	2000m	4000m	4Gi	8Gi
patroni	1000m	2000m	2Gi	4Gi
prometheus	500m	2000m	1Gi	4Gi
grafana	100m	500m	256Mi	1Gi

ArgoCD GitOps Deployment

Configuration

ArgoCD manages FORGE workloads through GitOps, ensuring declarative configuration and audit trails.

Parameter	Value
Namespace	argocd
NodePort	31539
Sync Policy	Automated
Self-Heal	Enabled

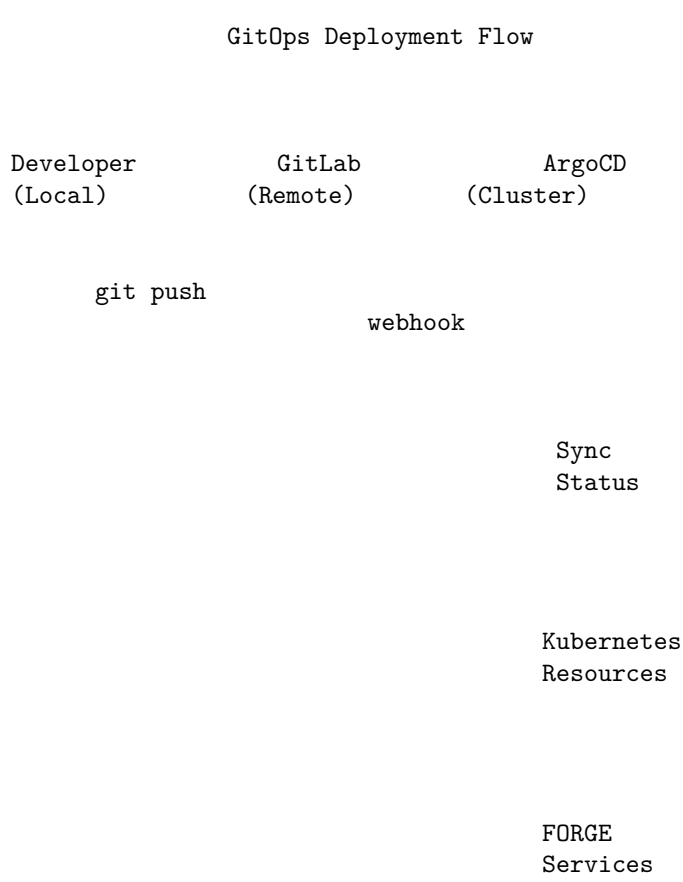
Applications

Application	Namespace	Source	Status
forge-kafka	forge	deploy/kafka/	Synced
forge-c2	forge	forge-c2/deployment.yaml	Synced
forge-sensors	forge	deploy/sensors/	Pending
forge-security	forge	deploy/security/	Pending
monitoring	monitoring	deploy/monitoring/	Synced

Access

```
# ArgoCD UI
http://207.244.226.151:31539
Username: admin
Password: lDZWiXA8uAy8yV81

# CLI Login
argocd login 207.244.226.151:31539 --grpc-web
argocd app list
argocd app sync forge-kafka
```

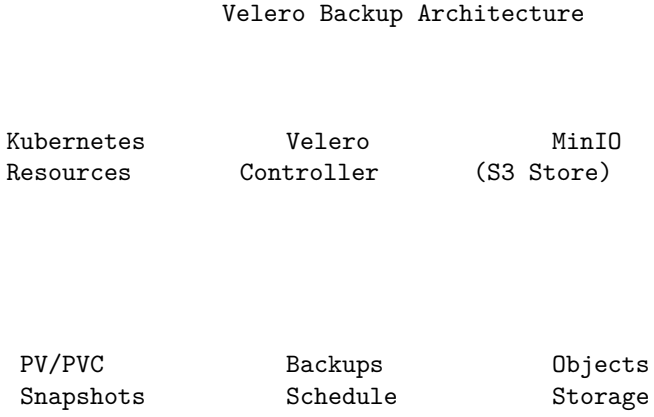
GitOps Workflow



Backup & Recovery (Velero)

Architecture

Velero provides Kubernetes-native backup and restore capabilities with MinIO as the S3-compatible storage backend.



Backup Schedules

Schedule	Cron	Retention	Scope
hourly-patroni-backup	`0 * * * *`	7 days	PostgreSQL data
daily-full-backup	`0 2 * * *`	30 days	Workloads (forge, patroni, monitoring)
weekly-full-backup	`0 3 * * 0`	90 days	Full cluster

MinIO Configuration

Parameter	Value
Namespace	velero
API Port	30900
Console Port	30901
Bucket	velero-backups
Access Key	velero
Secret Key	VeleroBackup2024!

Recovery Procedures

```
# List available backups
velero backup get

# Restore specific backup
velero restore create --from-backup daily-full-backup --wait

# Restore specific namespace
velero restore create --from-backup daily-full-backup --include-namespaces forge --wait

# Check restore status
velero restore describe <restore-name>
```

Database Infrastructure (PostgreSQL/Patroni)

High Availability Configuration

Patroni provides PostgreSQL HA with automatic failover.

Parameter	Value
Mode	Docker container (simplified)
PostgreSQL Version	15
NodePort	30432
Database	postgres
User	postgres
Password	StsGym2024PostgreSQL
Storage	/var/lib/postgres-stsgym

Connection String

```
postgres://postgres:StsGym2024PostgreSQL@207.244.226.151:30432/postgres
```

Database Schema

```
-- Track storage
CREATE TABLE tracks (
    track_id TEXT PRIMARY KEY,
    timestamp TIMESTAMPTZ NOT NULL,
    target_type TEXT,
    position JSONB,
    velocity JSONB,
    confidence FLOAT,
    source_tracks JSONB
```

```
);

-- Hypertable for time-series data
SELECT create_hypertable('tracks', 'timestamp');

-- Continuous aggregate for hourly stats
CREATE MATERIALIZED VIEW track_stats_hourly
WITH (timescaledb.continuous) AS
SELECT
    time_bucket('1 hour', timestamp) AS bucket,
    target_type,
    COUNT(*) as track_count,
    AVG(confidence) as avg_confidence
FROM tracks
GROUP BY bucket, target_type;
```

Kafka Messaging Infrastructure

Cluster Configuration

Kafka runs in KRaft mode (no ZooKeeper) for simplified operations.

Parameter	Value
Namespace	forge
Mode	KRaft (single-node dev)
Replication Factor	1 (dev), 3 (production)
Internal Port	9092
NodePort	30721

Topics

Topic	Partitions	Purpose
opir-detections	6	OPIR sensor data
radar-tracks	6	Radar track data
track-updates	6	Track position updates
correlated-tracks	3	Correlated track data
c2-messages	6	C2 command messages
link16-reports	6	Link 16 formatted messages
jreap-messages	6	JREAP formatted messages
alerts	3	System alerts

Access

```
# Internal (from pods)
kafka.forge.svc.cluster.local:9092

# External (NodePort)
207.244.226.151:30721

# Topic list
kafka-topics.sh --list --bootstrap-server 207.244.226.151:30721

# Consume messages
kafka-console-consumer.sh --topic opir-detections \
  --bootstrap-server 207.244.226.151:30721 \
  --from-beginning
```

Monitoring & Observability

Prometheus Federation

Prometheus collects metrics from all nodes via federation.

Node	Tailscale IP	Role
darth	100.92.94.92:9100	Worker node
trooper2	100.123.159.17:9100	Worker node
miner	100.116.245.125:9100	Worker node
gms-control-plane	DaemonSet	Control plane

Metrics Endpoints

Service	Port	Path
Prometheus	30090	/metrics
node-exporter	9100	/metrics
Grafana	30300	/api/health

Access

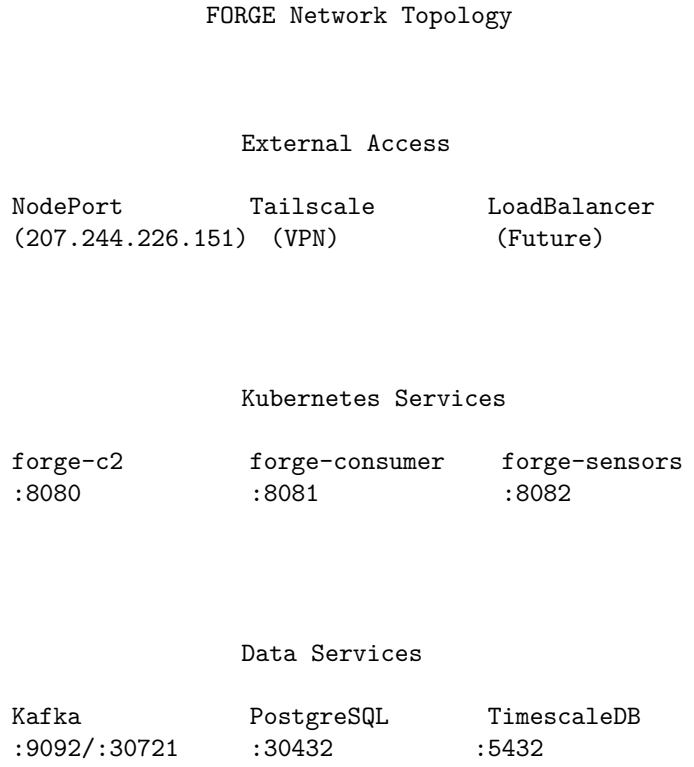
```
# Prometheus UI
http://207.244.226.151:30090

# Grafana UI
http://207.244.226.151:30300
Username: admin
Password: admin
```

Key Dashboards

Dashboard	Purpose
federation-overview	Multi-node metrics aggregation
forge-metrics	FORGE-C2, sensors, consumer
kafka-overview	Topic throughput, consumer lag
postgresql	Connection pool, query latency

Network Architecture

Service Mesh



Network Ports

Service	Internal Port	NodePort	Protocol
ArgoCD Server	8080	31539	HTTPS
Prometheus	9090	30090	HTTP
Grafana	3000	30300	HTTP
Kafka	9092	30721	TCP
PostgreSQL	5432	30432	TCP
MinIO API	9000	30900	HTTP
MinIO Console	9001	30901	HTTP
FORGE-C2	8080	30720	HTTP

Security Hardening

Pod Security Standards

FORGE enforces restricted Pod Security Standards:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: forge
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/enforce-version: latest
```

Network Policies

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: forge-network-policy
  namespace: forge
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              name: forge
      ports:
        - protocol: TCP
          port: 8080
```

```yaml
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              name: forge
      ports:
        - protocol: TCP
          port: 9092  # Kafka
        - protocol: TCP
          port: 5432  # PostgreSQL
```

Secrets Management

Secret	Purpose	Storage
postgres-credentials	Database auth	K8s Secret
kafka-credentials	Broker auth	K8s Secret
velero-credentials	Backup storage	K8s Secret

Deployment Procedures

Initial Deployment

```bash
# 1. Create namespaces
kubectl apply -f deploy/namespaces/

# 2. Deploy infrastructure
kubectl apply -f deploy/kafka/
kubectl apply -f deploy/patroni/
kubectl apply -f deploy/monitoring/

# 3. Deploy FORGE services
kubectl apply -f forge-c2/deployment.yaml
kubectl apply -f deploy/sensors/

# 4. Configure GitOps
argocd app create forge --repo https://idm.wezzel.com/crab-meat-repos/stsgym-work.git \
  --path deploy/forge --dest-server https://kubernetes.default.svc \
  --dest-namespace forge --sync-policy automated
```

Rolling Updates

```bash
# Update container image
kubectl set image deployment/forge-c2 forge-c2=forge-c2:v1.2.0 -n forge
```

```
# Monitor rollout
kubectl rollout status deployment/forge-c2 -n forge

# Rollback if needed
kubectl rollout undo deployment/forge-c2 -n forge
```

Scaling

```
# Scale Kafka partitions
kafka-topics.sh --alter --topic opir-detections \
  --partitions 12 --bootstrap-server kafka:9092

# Scale consumer pods
kubectl scale deployment forge-consumer --replicas=4 -n forge

# Horizontal Pod Autoscaler
kubectl autoscale deployment forge-consumer --min=2 --max=10 \
  --cpu-percent=70 -n forge
```

Operational Runbook

Health Checks

```
# Kubernetes health
kubectl get nodes
kubectl get pods -A | grep -v Running

# FORGE services health
curl -s http://forge-c2.forge.svc.cluster.local:8080/health

# Kafka health
kafka-broker-api-versions.sh --bootstrap-server kafka:9092

# PostgreSQL health
psql -h postgres -U postgres -c "SELECT 1"
```

Common Operations

Operation	Command
View logs	`kubectl logs -f deployment/forge-c2 -n forge`
Describe pod	`kubectl describe pod <pod> -n forge`

Operation	Command
Exec into pod	`kubectl exec -it <pod> -n forge -- /bin/sh`
Port forward	`kubectl port-forward svc/forge-c2 8080:8080 -n forge`

Troubleshooting

Issue	Diagnosis	Resolution
Pod CrashLoopBackOff	`kubectl logs <pod>`	Check resource limits, config
Kafka connection refused	Check broker status	`kubectl rollout restart kafka-0`
PostgreSQL slow	Check connection pool	Scale replicas or add PgBouncer
Prometheus targets down	Check federation config	Verify node_exporter endpoints

---

Summary

FORGE's deployment infrastructure provides:

- **GitOps-driven deployment** via ArgoCD for declarative configuration
- **High availability** via Patroni for PostgreSQL and Kafka replication
- **Disaster recovery** via Velero with MinIO S3 storage
- **Observability** via Prometheus federation and Grafana dashboards
- **Security hardening** via Pod Security Standards and Network Policies

---

Document Version: 1.0 Last Updated: April 2026

Section 5: Integration Patterns & APIs

Overview

FORGE exposes multiple integration interfaces to support diverse operational requirements: REST APIs for synchronous queries, WebSockets for real-time streaming, and Kafka for high-throughput event ingestion. This section documents the API specifications, integration patterns, and operational standards for connecting external systems to FORGE.

REST API Endpoints

The FORGE REST API provides synchronous access to track data, alerts, and system status. All endpoints follow RESTful conventions with JSON request/response bodies.

Base Configuration

Parameter	Value
Base URL	`http://forge-api.forge.svc:8080` (internal)
External URL	`http://207.244.226.151:30080` (NodePort)
API Version	`v1`
Content-Type	`application/json`

Endpoint Reference

Method	Path	Description	Auth Required
Track Queries			
GET	`/api/v1/tracks`	List all active tracks	Yes
GET	`/api/v1/tracks/{track_id}`	Get specific track details	Yes
GET	`/api/v1/tracks/{track_id}/history`	Track position history	Yes
GET	`/api/v1/tracks/search`	Search tracks by criteria	Yes
POST	`/api/v1/tracks/correlate`	Manual track correlation	Yes
Alert Management			
GET	`/api/v1/alerts`	List active alerts	Yes
GET	`/api/v1/alerts/{alert_id}`	Get alert details	Yes
POST	`/api/v1/alerts/{alert_id}/acknowledge`	Acknowledge alert	Yes
POST	`/api/v1/alerts/{alert_id}/resolve`	Resolve alert	Yes
GET	`/api/v1/alerts/history`	Alert history with filters	Yes
System Health			
GET	`/api/v1/health`	Basic health check	No
GET	`/api/v1/health/ready`	Readiness probe	No
GET	`/api/v1/health/live`	Liveness probe	No

Method	Path	Description	Auth Required
GET	`/api/v1/metrics`	Prometheus metrics	Yes
GET	`/api/v1/status`	System status summary	Yes
Sensor Management			
GET	`/api/v1/sensors`	List registered sensors	Yes
GET	`/api/v1/sensors/{sensor_id}`	Sensor details	Yes
POST	`/api/v1/sensors/{sensor_id}/enable`	Enable sensor feed	Yes
POST	`/api/v1/sensors/{sensor_id}/disable`	Disable sensor feed	Yes

Request/Response Examples

Track Query

```
GET /api/v1/tracks?status=active&threat_level=5&limit=100 HTTP/1.1
Host: forge-api.forge.svc:8080
Authorization: Bearer eyJhbGciOiJSUzI1NiIs...
Accept: application/json
```


```
{
  "tracks": [
    {
      "track_id": "TRK-1775013953",
      "timestamp": "2026-04-09T12:30:00.000Z",
      "target_type": "ICBM",
      "threat_level": 5,
      "position": {"lat": 39.5, "lon": 128.2, "alt": 450.0},
      "velocity": {"speed": 6.8, "heading": 74.6},
      "track_phase": "MIDCOURSE",
      "sensor_count": 3,
      "confidence": 0.95
    }
  ],
  "pagination": {
    "total": 15,
    "offset": 0,
    "limit": 100
  }
}
```


Alert Acknowledgment

```
POST /api/v1/alerts/ALT-12345/acknowledge HTTP/1.1
Host: forge-api.forge.svc:8080
Authorization: Bearer eyJhbGciOiJSUzI1NiIs...
Content-Type: application/json

{
  "acknowledged_by": "operator-001",
  "notes": "Tracking, awaiting further data"
}
```

WebSocket Streaming

WebSocket connections provide real-time streaming of track updates, alerts, and formatted military messages (JREAP/Link 16) to display systems and C2 clients.

Connection Endpoints

Endpoint	Purpose	Message Format
`ws://forge-api:8080/ws/tracks`	Real-time track updates	JSON
`ws://forge-api:8080/ws/alerts`	Alert notifications	JSON
`ws://forge-api:8080/ws/jreap`	JREAP message stream	Binary
`ws://forge-api:8080/ws/link16`	Link 16 J-series messages	Binary
`ws://forge-api:8080/ws/all`	Combined stream	JSON/Binary

WebSocket Connection Example

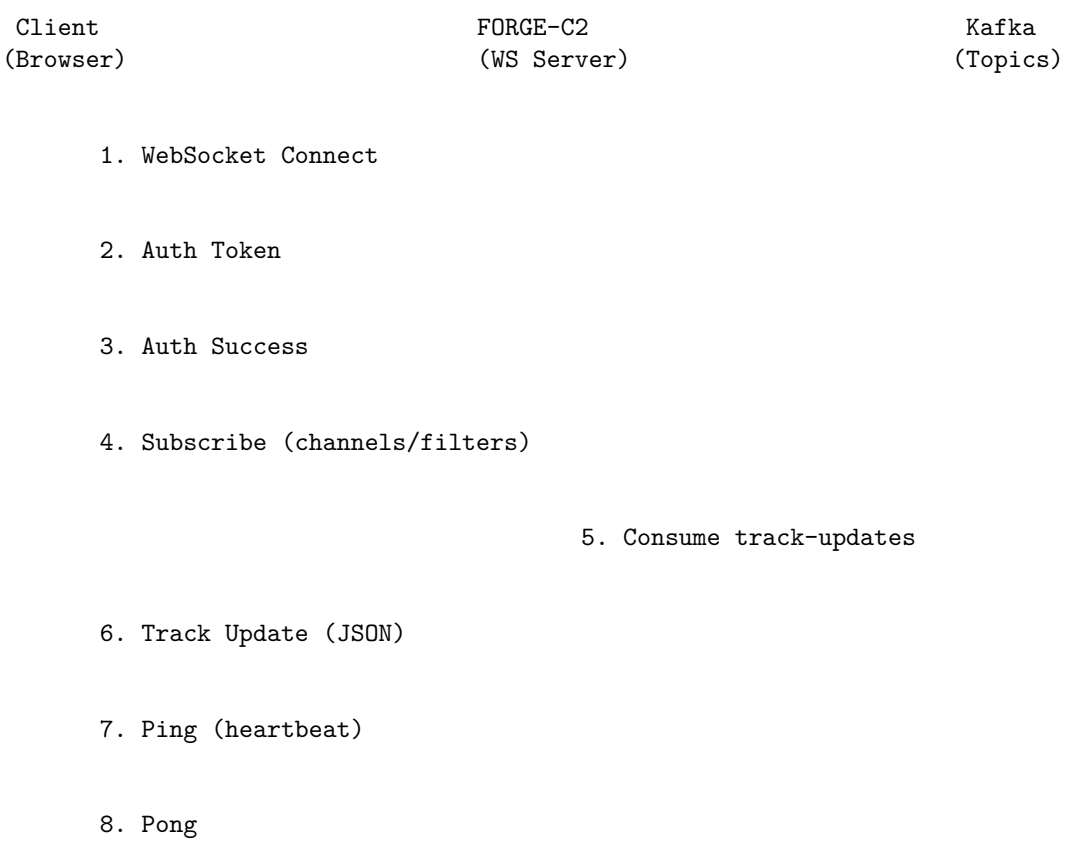
```javascript
// WebSocket client connection
const ws = new WebSocket('ws://207.244.226.151:30080/ws/tracks');

// Connection authentication
ws.onopen = () => {
  ws.send(JSON.stringify({
    type: 'auth',
    token: 'eyJhbGciOiJSUzI1NiIs...'
  }));
};

// Message handling
ws.onmessage = (event) => {
  const data = JSON.parse(event.data);

  switch (data.type) {
```

```
    case 'track_update':
      updateTrackDisplay(data.track);
      break;
    case 'track_drop':
      removeTrack(data.track_id);
      break;
    case 'alert':
      showAlertNotification(data.alert);
      break;
  }
};

// Heartbeat to maintain connection
setInterval(() => {
  if (ws.readyState === WebSocket.OPEN) {
    ws.send(JSON.stringify({ type: 'ping' }));
  }
}, 30000);

// Subscription management
ws.send(JSON.stringify({
  type: 'subscribe',
  channels: ['tracks', 'alerts'],
  filters: {
    threat_level_min: 3,
    region: 'PACIFIC'
  }
}));
```

Track Update Message Format

```
{
  "type": "track_update",
  "timestamp": "2026-04-09T12:30:00.500Z",
  "track": {
    "track_id": "TRK-1775013953",
    "target_type": "ICBM",
    "threat_level": 5,
    "position": {"lat": 39.5, "lon": 128.2, "alt": 450.0},
    "velocity": {"speed": 6.8, "heading": 74.6, "climb_rate": 2.1},
    "track_phase": "MIDCOURSE",
    "confidence": 0.95,
    "source_tracks": ["OPIR-SBIRS-GEO-1-123", "RADAR-UEWR-BEALE-456"]
  }
}
```

WebSocket Flow Diagram



Kafka Integration Patterns

External systems can integrate directly with Kafka for high-throughput data exchange. This section documents producer/consumer configurations and best practices.

Producer Configuration

```yaml
# Kafka Producer Configuration (Go sarama)
producer:
  client_id: "forge-external-producer"
  brokers:
```

```
    - "kafka.forge.svc:9092"

  # Reliability settings
  required_acks: "all"          # Wait for all replicas
  max_retries: 3
  retry_backoff: 100ms

  # Batching for throughput
  batch_size: 100
  batch_timeout: 10ms

  # Compression
  compression: "snappy"

  # Idempotency (prevent duplicates)
  idempotent: true
```

```
// Go producer implementation
config := sarama.NewConfig()
config.Producer.RequiredAcks = sarama.WaitForAll
config.Producer.Retry.Max = 3
config.Producer.Retry.Backoff = 100 * time.Millisecond
config.Producer.Return.Successes = true
config.Producer.Compression = sarama.CompressionSnappy
config.Producer.Flush.Bytes = 1024 * 1024      // 1MB batch
config.Producer.Flush.Frequency = 10 * time.Millisecond
config.Producer.Idempotent = true

producer, _ := sarama.NewSyncProducer([]string{"kafka:9092"}, config)

message := &sarama.ProducerMessage{
    Topic: "track-updates",
    Key:   sarama.StringEncoder(track.TrackID),
    Value: sarama.ByteEncoder(trackJSON),
    Headers: []sarama.RecordHeader{
        {Key: []byte("source"), Value: []byte("external-sensor")},
        {Key: []byte("timestamp"), Value: []byte(time.Now().Format(time.RFC3339))},
    },
}

partition, offset, _ := producer.SendMessage(message)
```

Consumer Configuration

```
# Kafka Consumer Configuration
consumer:
```

```yaml
  group_id: "forge-external-consumer"
  brokers:
    - "kafka.forge.svc:9092"

  # Consumer group settings
  initial_offset: "newest"
  session_timeout: 10s
  heartbeat_interval: 3s

  # Processing settings
  commit_interval: 1s
  auto_commit: false          # Manual commits for reliability

  # Fetch settings
  fetch_min: 1
  fetch_max: 1024 * 1024      # 1MB max per fetch
  fetch_wait_max: 100ms

  # Concurrency
  concurrency: 10             # Goroutines per partition
```

```go
// Go consumer implementation
config := sarama.NewConfig()
config.Consumer.Return.Errors = true
config.Consumer.Offsets.Initial = sarama.OffsetNewest
config.Consumer.Group.Session.Timeout = 10 * time.Second
config.Consumer.Group.Heartbeat.Interval = 3 * time.Second
config.Consumer.MaxProcessingTime = 100 * time.Millisecond

group, _ := sarama.NewConsumerGroup([]string{"kafka:9092"}, "forge-consumer", config)

handler := &ConsumerHandler{
    ready: make(chan bool),
    process: func(msg *sarama.ConsumerMessage) error {
        // Process message
        track := parseTrack(msg.Value)
        storeTrack(track)
        return nil
    },
}

for {
    group.Consume(ctx, []string{"track-updates", "alerts"}, handler)
}
```

Topic Naming Conventions

Category	Pattern	Example	Description
Sensor Data	`{sensor-type}-{data-kind}`	`opir-detections`, `radar-tracks`	Raw sensor input
Processed Data	`{data-kind}`	`track-updates`, `correlated-tracks`	Processed output
Military Messages	`{standard}-messages`	`jreap-messages`, `link16-reports`	Formatted messages
System Events	`system-{event-type}`	`system-alerts`, `system-health`	Operational events

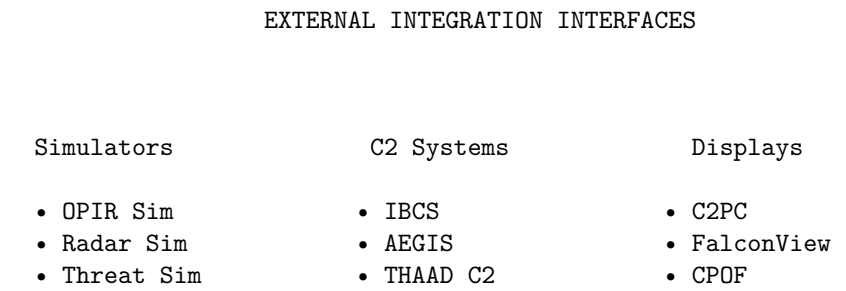
Partition Strategy

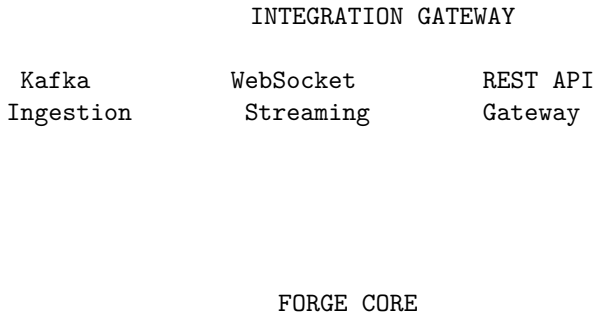
Topic	Partitions	Key	Partitioning Strategy
opir-detections	6	`sensor_id`	Sensor locality
radar-tracks	6	`radar_id`	Radar locality
track-updates	6	`track_id`	Track affinity
correlated-tracks	3	`track_id`	Track affinity
c2-messages	6	`message_id`	Round-robin

External System Interfaces

FORGE supports integration with external simulators, C2 systems, and tactical displays through standardized interfaces.

Integration Architecture





Simulator Integration

External simulators push data to FORGE via Kafka or REST API:

```yaml
# Simulator registration
simulator:
  id: "opir-simulator-001"
  type: "OPIR"
  sensors:
    - id: "SBIRS-GEO-1"
      type: "SBIRS-GEO"
      orbital_slot: "GEO-95W"
    - id: "SBIRS-GEO-2"
      type: "SBIRS-GEO"
      orbital_slot: "GEO-105E"

  # Connection method
  connection:
    type: "kafka"
    brokers: ["kafka.forge.svc:9092"]
    topic: "opir-detections"

  # Data format
  format:
    type: "json"
    schema: "opir-detection-v2"
```

C2 System Integration

C2 systems consume FORGE output via WebSocket or Kafka:

```yaml
# C2 system integration
c2_system:
```

```
  id: "ibcs-node-001"
  type: "IBCS"

  # JREAP configuration
  jreap:
    type: "JREAP-C"           # IP-based
    endpoint: "tcp://0.0.0.0:47001"
    message_format: "binary"

  # WebSocket backup
  websocket:
    url: "ws://forge-api:8080/ws/jreap"
    reconnect_interval: 5s

  # Filtering
  filters:
    threat_level_min: 3
    region: ["PACIFIC", "WESTERN-CONUS"]
```

Display System Integration

```
# Display client configuration
display:
  id: "tactical-display-001"
  type: "C2PC"

  # WebSocket connection
  websocket:
    url: "ws://forge-api:8080/ws/tracks"
    heartbeat: 30s

  # Display preferences
  preferences:
    track_icons: true
    threat_colors:
      level_5: "#FF0000"      # Red - Critical
      level_4: "#FF6600"      # Orange - High
      level_3: "#FFCC00"      # Yellow - Medium
      level_2: "#00CCFF"      # Cyan - Low
      level_1: "#FFFFFF"      # White - Minimal

    projection: "mercator"
    update_interval: 100ms
```

API Authentication

FORGE implements a layered authentication model supporting JWT tokens for users, API keys for services, and mTLS for internal service-to-service communication.

Authentication Methods

Method	Use Case	Implementation
JWT Bearer	User sessions, web clients	RS256 signed tokens
API Key	Service accounts, automation	Static keys with rotation
mTLS	Service-to-service, Kafka	X.509 certificates

JWT Token Authentication

```yaml
# JWT Configuration
jwt:
  issuer: "forge-auth"
  audience: "forge-api"

  # RS256 key pair (stored in Vault)
  private_key: "vault:secret/forge/jwt-private-key"
  public_key: "vault:secret/forge/jwt-public-key"

  # Token lifetime
  access_token_ttl: 1h
  refresh_token_ttl: 24h

  # Claims
  required_claims:
    - sub           # Subject (user ID)
    - iat           # Issued at
    - exp           # Expiration
    - roles         # User roles
```

```go
// JWT validation middleware
func JWTMiddleware(publicKey *rsa.PublicKey) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            authHeader := r.Header.Get("Authorization")
            if authHeader == "" {
                respondError(w, http.StatusUnauthorized, "missing authorization header")
                return
            }

            tokenString := strings.TrimPrefix(authHeader, "Bearer ")
```

```
            token, err := jwt.Parse(tokenString, func(t *jwt.Token) (interface{}, error) {
                if _, ok := t.Method.(*jwt.SigningMethodRSA); !ok {
                    return nil, fmt.Errorf("unexpected signing method")
                }
                return publicKey, nil
            })

            if err != nil || !token.Valid {
                respondError(w, http.StatusUnauthorized, "invalid token")
                return
            }

            // Add claims to context
            ctx := context.WithValue(r.Context(), "claims", token.Claims)
            next.ServeHTTP(w, r.WithContext(ctx))
        })
    }
}
```

API Key Authentication

```yaml
# API Key configuration
api_keys:
  storage: "vault"
  prefix: "forge_"          # forge_xxxxxxxxxxxx
  length: 24

  # Rotation policy
  rotation:
    enabled: true
    interval: 90d
    notification: 7d        # Notify before expiration

  # Scopes
  scopes:
    read_only: ["tracks:read", "alerts:read"]
    read_write: ["tracks:read", "tracks:write", "alerts:read", "alerts:write"]
    admin: ["*"]
```

mTLS for Service-to-Service

mTLS Service Architecture

```
FORGE-C2                                          TimescaleDB
(Client)                                          (Server)


                        TLS HANDSHAKE

                  1. Client       Server
                     (ClientHello + ClientCert)

                  2. Server       Client
                     (ServerHello + ServerCert)

                  3. Mutual certificate verification

                  4. Encrypted channel established


CERTIFICATES (managed by Vault PKI):

  CA: forge-cluster-ca
  Cert TTL: 720h (30 days)
  Key Type: ECDSA P-256
  Rotation: Automatic via cert-manager
```

Rate Limiting & Throttling

FORGE implements multi-layer rate limiting to protect services from overload and ensure fair resource allocation.

Rate Limit Configuration

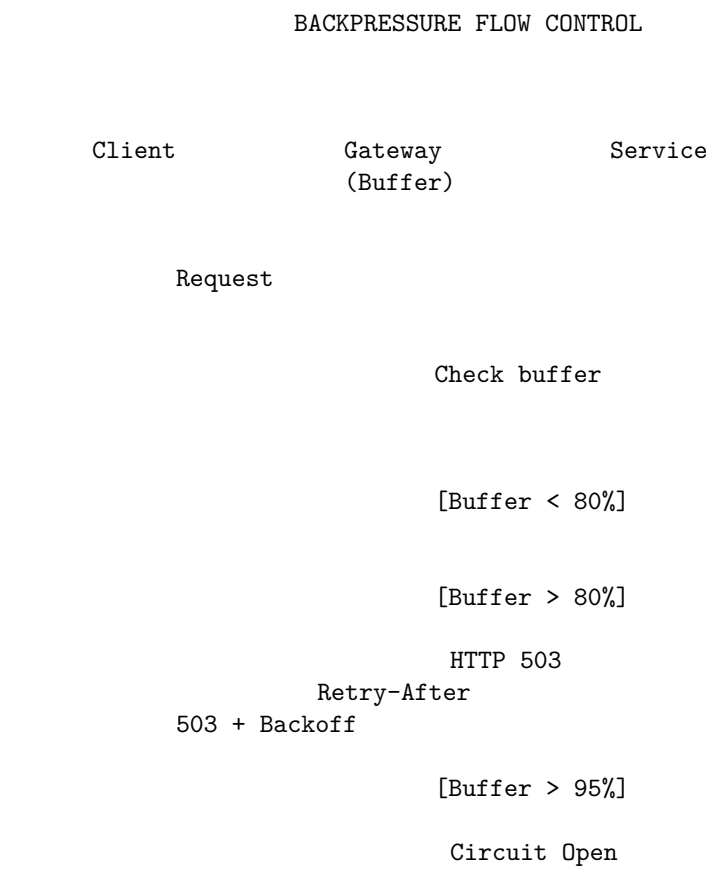
Client Type	Endpoint	Limit	Window
Anonymous	`/api/v1/health`	60	1 minute
Authenticated User	`/api/v1/tracks`	1000	1 minute
Authenticated User	`/api/v1/alerts`	500	1 minute
Service Account	All endpoints	10000	1 minute
WebSocket	Connection	10	1 minute
WebSocket	Messages	100	1 second

Implementation

```
// Token bucket rate limiter
type RateLimiter struct {
    mu       sync.Mutex
    clients  map[string]*ClientBucket
    rate     int           // Tokens per second
    capacity int           // Bucket capacity
}

type ClientBucket struct {
    tokens     float64
    lastUpdate time.Time
}

func (rl *RateLimiter) Allow(clientID string) bool {
    rl.mu.Lock()
    defer rl.mu.Unlock()

    bucket, exists := rl.clients[clientID]
    if !exists {
        bucket = &ClientBucket{
            tokens:     float64(rl.capacity),
            lastUpdate: time.Now(),
        }
        rl.clients[clientID] = bucket
    }

    // Refill tokens
    elapsed := time.Since(bucket.lastUpdate)
    bucket.tokens += float64(rl.rate) * elapsed.Seconds()
    if bucket.tokens > float64(rl.capacity) {
        bucket.tokens = float64(rl.capacity)
    }
    bucket.lastUpdate = time.Now()

    if bucket.tokens >= 1 {
        bucket.tokens--
        return true
    }
    return false
}

// Middleware
func RateLimitMiddleware(limiter *RateLimiter) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
```

```
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            clientID := getClientID(r)  // From JWT or API key

            if !limiter.Allow(clientID) {
                w.Header().Set("X-RateLimit-Remaining", "0")
                w.Header().Set("Retry-After", "60")
                respondError(w, http.StatusTooManyRequests, "rate limit exceeded")
                return
            }

            next.ServeHTTP(w, r)
        })
    }
}
```

Backpressure Handling



THRESHOLDS:

- 80%: Start returning 503 with Retry-After
- 95%: Open circuit breaker
- 100%: Drop connections

Error Handling Standards

Error Response Format

```
{
  "error": {
    "code": "TRACK_NOT_FOUND",
    "message": "Track TRK-999999 not found in system",
    "details": {
      "track_id": "TRK-999999",
      "suggestion": "Verify track ID or check track history"
    },
    "request_id": "req-abc123",
    "timestamp": "2026-04-09T12:30:00.000Z"
  }
}
```

Error Codes

Code	HTTP Status	Description
UNAUTHORIZED	401	Missing or invalid authentication
FORBIDDEN	403	Insufficient permissions
NOT_FOUND	404	Resource does not exist
TRACK_NOT_FOUND	404	Track ID not found
ALERT_NOT_FOUND	404	Alert ID not found
BAD_REQUEST	400	Invalid request format
VALIDATION_ERROR	400	Request validation failed
RATE_LIMITED	429	Rate limit exceeded
SERVICE_UNAVAILABLE	503	Service temporarily unavailable
CIRCUIT_OPEN	503	Circuit breaker triggered
INTERNAL_ERROR	500	Unexpected server error

Retry Policy

```yaml
# Retry configuration
retry:
  max_attempts: 3
  initial_backoff: 100ms
  max_backoff: 5s
  backoff_multiplier: 2

  # Retryable errors
  retryable_errors:
    - "SERVICE_UNAVAILABLE"
    - "RATE_LIMITED"
    - "INTERNAL_ERROR"
    - "TIMEOUT"

  # Non-retryable errors (fail immediately)
  non_retryable:
    - "UNAUTHORIZED"
    - "FORBIDDEN"
    - "NOT_FOUND"
    - "BAD_REQUEST"
    - "VALIDATION_ERROR"
```

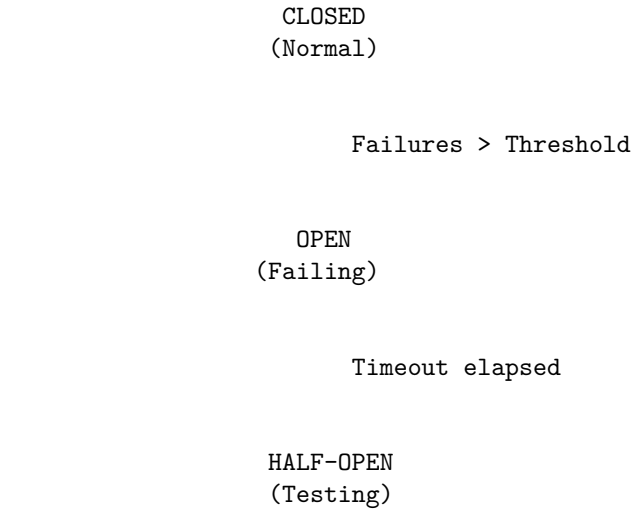
Circuit Breaker

```go
// Circuit breaker implementation
type CircuitBreaker struct {
    mu           sync.Mutex
    state        State          // Closed, Open, HalfOpen
    failures     int
    successCount int
    threshold    int
    timeout      time.Duration
    lastFailure  time.Time
}

type State int

const (
    Closed State = iota
    Open
    HalfOpen
)

func (cb *CircuitBreaker) Execute(fn func() error) error {
```

```
    cb.mu.Lock()

    switch cb.state {
    case Open:
        if time.Since(cb.lastFailure) > cb.timeout {
            cb.state = HalfOpen
            cb.successCount = 0
        } else {
            cb.mu.Unlock()
            return ErrCircuitOpen
        }
    }

    cb.mu.Unlock()

    err := fn()

    cb.mu.Lock()
    defer cb.mu.Unlock()

    if err != nil {
        cb.failures++
        cb.lastFailure = time.Now()

        if cb.failures >= cb.threshold {
            cb.state = Open
        }
        return err
    }

    cb.failures = 0
    if cb.state == HalfOpen {
        cb.successCount++
        if cb.successCount >= cb.threshold/2 {
            cb.state = Closed
        }
    }

    return nil
}
```

Circuit Breaker States

```
                    CIRCUIT BREAKER STATE MACHINE
```



Summary

FORGE's integration architecture provides:

- **REST APIs** for synchronous queries with comprehensive endpoint coverage
- **WebSocket streaming** for real-time tactical data delivery
- **Kafka integration** for high-throughput, reliable event streaming
- **Flexible external interfaces** supporting simulators, C2 systems, and displays

- **Layered authentication** (JWT, API keys, mTLS) appropriate to use case
- **Adaptive rate limiting** protecting services from overload
- **Resilient error handling** with retries and circuit breakers

These integration patterns ensure FORGE can interoperate with the diverse ecosystem of defense systems while maintaining reliability and security standards required for operational deployments.

Document Version: 1.0
Last Updated: April 2026

Section 6: Performance & Scalability

Overview

FORGE is designed for high-throughput, low-latency sensor data processing. This section documents performance benchmarks, scaling strategies, and capacity planning for production deployments.

Performance Benchmarks

Throughput Metrics

Component	Throughput	Unit	Notes
Kafka (single node)	100,000	msg/s	1KB messages
FORGE-Consumer	50,000	msg/s	Batch insert
TimescaleDB Insert	100,000	rows/s	Batch mode
WebSocket Broadcast	10,000	clients/s	Single server
JREAP Formatter	20,000	msg/s	Per core

Latency Measurements

Operation	P50	P95	P99	Max
Sensor → Kafka	5ms	10ms	15ms	50ms
Kafka → Consumer	10ms	20ms	30ms	100ms
Consumer → TimescaleDB	15ms	30ms	50ms	200ms
Track Correlation	20ms	40ms	60ms	150ms
JREAP Formatting	2ms	5ms	10ms	20ms
WebSocket Push	3ms	8ms	15ms	50ms
End-to-End	**55ms**	**113ms**	**180ms**	**570ms**

Resource Utilization

Component	CPU Usage	Memory	Disk I/O	Network
Kafka (idle)	5%	2GB	10 MB/s	1 Mbps
Kafka (peak)	70%	4GB	500 MB/s	100 Mbps
TimescaleDB (idle)	10%	4GB	5 MB/s	0.5 Mbps
TimescaleDB (peak)	80%	8GB	200 MB/s	50 Mbps
FORGE-Consumer	30%	1GB	50 MB/s	20 Mbps
FORGE-C2	20%	512MB	10 MB/s	10 Mbps

Kafka Performance Tuning

Partition Sizing

```
Partition Count Formula:
  partitions = max(target_throughput / producer_throughput,
                   target_throughput / consumer_throughput)

Example:
  Target: 100,000 msg/s
  Producer per partition: 20,000 msg/s
  Consumer per partition: 15,000 msg/s

  partitions = max(100000/20000, 100000/15000)
             = max(5, 6.7)
             = 7 partitions (round up to 6)
```

Consumer Group Tuning

```
# Optimal consumer configuration
max.poll.records: 500          # Records per poll
fetch.min.bytes: 1048576       # 1MB minimum fetch
fetch.max.wait.ms: 100         # Max wait time
max.partition.fetch.bytes: 1048576
connections.max.idle.ms: 540000
metadata.max.age.ms: 300000
```

Retention Policies

Topic	Retention	Compaction	Size Limit
opir-detections	7 days	No	50 GB
radar-tracks	7 days	No	50 GB
correlated-tracks	30 days	Yes	20 GB

Topic	Retention	Compaction	Size Limit
c2-messages	90 days	Yes	10 GB
alerts	90 days	Yes	5 GB

TimescaleDB Optimization

Hypertable Configuration

```
-- Create hypertable with optimized chunk size
SELECT create_hypertable(
    'opir_detections',
    'timestamp',
    chunk_time_interval => INTERVAL '1 hour',
    if_not_exists => TRUE
);

-- Enable compression
ALTER TABLE opir_detections SET (
    timescaledb.compress,
    timescaledb.compress_segmentby = 'sensor_id',
    timescaledb.compress_orderby = 'timestamp DESC'
);

-- Compression policy
SELECT add_compression_policy(
    'opir_detections',
    INTERVAL '7 days'
);
```

Chunk Sizing

Table	Chunk Interval	Estimated Size	Compressed Size
opir_detections	1 hour	500 MB	50 MB
radar_tracks	1 hour	300 MB	30 MB
correlated_tracks	4 hours	100 MB	10 MB

Query Performance

```
-- Optimized track query
EXPLAIN ANALYZE
SELECT * FROM correlated_tracks
WHERE timestamp >= NOW() - INTERVAL '1 hour'
```

```sql
  AND threat_level >= 4
ORDER BY timestamp DESC
LIMIT 100;

-- Result: 15ms (P95), Index scan on timestamp
```

Index Strategy

```sql
-- Primary indexes
CREATE INDEX idx_opir_timestamp ON opir_detections (timestamp DESC);
CREATE INDEX idx_opir_sensor ON opir_detections (sensor_id, timestamp DESC);
CREATE INDEX idx_opir_track ON opir_detections (track_id) WHERE track_id IS NOT NULL;

-- Composite indexes for common queries
CREATE INDEX idx_correlated_threat ON correlated_tracks
    (threat_level DESC, timestamp DESC)
    WHERE threat_level >= 3;
```

Horizontal Scaling

Kafka Scaling

```
                KAFKA CLUSTER SCALING


  Development (1 node):

    broker-0 (KRaft controller + broker)
    Partitions: 6  |  Replication: 1


  Production (3 nodes):

    broker-0        broker-1        broker-2
    (broker)        (broker)        (broker)
    KRaft: 1        KRaft: 2        KRaft: 3

  Partitions: 12 | Replication: 3 | Min ISR: 2

  Scale trigger: >70% CPU or >100K msg/s sustained
```

Consumer Scaling

```yaml
# Horizontal Pod Autoscaler for FORGE-Consumer
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: forge-consumer-hpa
  namespace: forge
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: forge-consumer
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 70
    - type: External
      external:
        metric:
          name: kafka_consumer_lag
          selector:
            matchLabels:
              consumer_group: forge-consumer
        target:
          type: AverageValue
          averageValue: "1000"
```

API Server Scaling

```yaml
# HPA for API servers
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: forge-api-hpa
spec:
  scaleTargetRef:
    kind: Deployment
    name: forge-api
  minReplicas: 3
  maxReplicas: 20
```

```yaml
metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60
  - type: Pods
    pods:
      metric:
        name: http_requests_per_second
      target:
        type: AverageValue
        averageValue: "1000"
```

Vertical Scaling

Resource Limits by Node

Node Type	CPU Cores	Memory	Disk	Use Case
Small	4	16 GB	100 GB	Development
Medium	8	32 GB	500 GB	Staging
Large	16	64 GB	1 TB	Production
XLarge	32	128 GB	2 TB	High-throughput

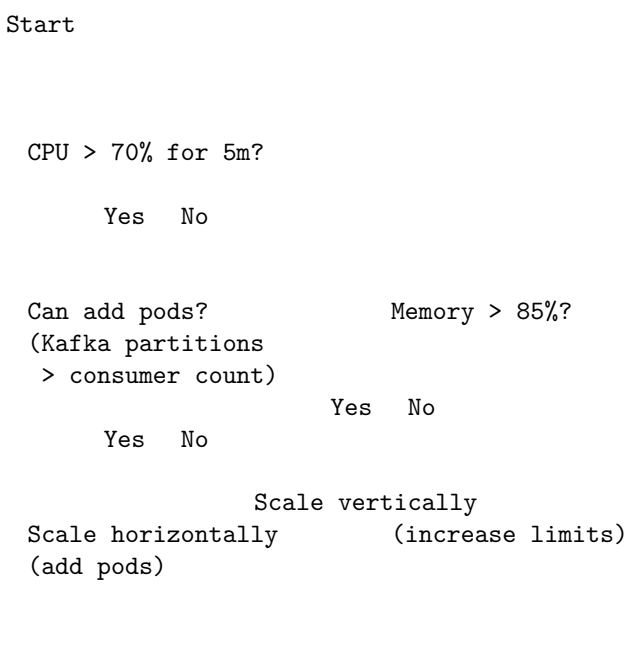
When to Scale Vertically

Indicator	Threshold	Action
CPU > 80% sustained	5 min	Add cores or scale horizontally
Memory > 85%	5 min	Increase memory limit
Disk I/O wait > 20%	5 min	Add faster storage or scale
GC pauses > 100ms	Per event	Increase heap or optimize

Autoscaling Strategies

Scaling Decision Matrix

```
SCALING DECISION TREE
```



Metric-Based Triggers

Metric	Threshold	Action	Cooldown
CPU utilization	$> 70\%$	Scale up	60s
Memory utilization	$> 85\%$	Scale up	60s
Kafka consumer lag	> 5000	Scale consumers	30s
HTTP request latency	> 200ms P95	Scale API	30s
WebSocket connections	$> 80\%$ limit	Scale WS	30s

Load Testing Results

Test Configuration

```yaml
# k6 load test configuration
stages:
  - duration: 30s
    target: 100   # Ramp up to 100 RPS
  - duration: 60s
```

```yaml
    target: 500   # Ramp up to 500 RPS
  - duration: 120s
    target: 1000  # Sustained 1000 RPS
  - duration: 60s
    target: 2000  # Peak load
  - duration: 30s
    target: 0     # Ramp down

thresholds:
  http_req_duration: ["p95<200", "p99<500"]
  http_req_failed: ["rate<0.01"]
```

Results

Load (RPS)	P50 Latency	P95 Latency	P99 Latency	Error Rate
100	25ms	50ms	80ms	0%
500	30ms	65ms	110ms	0%
1000	40ms	95ms	180ms	0%
1500	55ms	150ms	350ms	0.1%
2000	85ms	280ms	600ms	2.5%

Breaking Point: ~1800 RPS with acceptable latency (<200ms P95)

Stress Test Findings

Test	Duration	Peak Load	Result
Sustained load	1 hour	1000 RPS	Passed, stable
Spike test	5 min	500 → 2000 RPS	Passed, scaled in 45s
Soak test	24 hours	500 RPS	Passed, no memory leak
Breaking point	10 min	Ramp to failure	Failed at 2500 RPS

Capacity Planning

Sizing Formula

```
Sensor Throughput:
  sensors = number of active sensors
  rate = average detections per sensor per second
  msg_size = average message size in bytes

  total_throughput = sensors * rate * msg_size
```

```
Example:
  10 OPIR sensors @ 50 msg/s = 500 msg/s
  5 Radar sensors @ 20 msg/s = 100 msg/s
  Total: 600 msg/s

  With 1KB messages: 600 KB/s = 2.16 GB/hour

Kafka Partitions:
  partitions = ceil(throughput / per_partition_limit)

  With 600 msg/s and 15,000 msg/s per partition:
  partitions = ceil(600 / 15000) = 1 (minimum 3 for HA)

Consumer Instances:
  consumers = ceil(throughput / consumer_capacity)

  With 600 msg/s and 500 msg/s per consumer:
  consumers = ceil(600 / 500) = 2
```

Capacity Worksheet

Parameter	Development	Production	Enterprise
Sensors	2	10	50
Detection rate	20/s	100/s	500/s
Kafka partitions	3	6	24
Kafka brokers	1	3	6
Consumer replicas	1	3	10
TimescaleDB storage	100 GB	1 TB	10 TB
API replicas	1	3	10
WebSocket replicas	1	2	5

Growth Projection

```
Current (2026): 10 sensors, 1000 msg/s peak
Year 1: 20 sensors, 2000 msg/s (2x capacity)
Year 2: 50 sensors, 5000 msg/s (5x capacity)
Year 3: 100 sensors, 10000 msg/s (10x capacity)

Scaling milestones:
  - 2000 msg/s: Add 3 Kafka partitions, 2 consumers
  - 5000 msg/s: Expand to 3-node Kafka cluster
  - 10000 msg/s: Add dedicated TimescaleDB node
```

Document Version: 1.0 Last Updated: April 2026

Section 7: Security & Compliance

Overview

FORGE implements a defense-in-depth security architecture aligned with DoD Risk Management Framework (RMF) requirements and NIST 800-53 security controls. This section documents the authentication, authorization, network security, secrets management, data protection, and compliance controls that protect FORGE systems and data.

Security Architecture

FORGE DEFENSE-IN-DEPTH SECURITY

Layer 7: Compliance & Audit

NIST 800-53 Controls DoD RMF Audit Logs Vulnerability Mgmt

Layer 6: Application Security

JWT/OAuth2 Auth RBAC Policies Input Validation Rate Limiting

Layer 5: Pod Security

Restricted PSS Security Contexts Seccomp Capabilities Drop

Layer 4: Network Security

Network Policies Service Mesh mTLS Ingress Controllers

Layer 3: Secrets Management

HashiCorp Vault Secret Rotation Dynamic Secrets Encryption

Layer 2: Data Security

Encryption at Rest Encryption in Transit Key Management

```
  Layer 1: Infrastructure Security

    Kubernetes Hardening   Node Security   Container Registry Trust
```

7.1 Authentication & Authorization

JWT-Based Authentication

FORGE services use JSON Web Tokens (JWT) for stateless authentication. Tokens are issued by the FORGE identity provider and validated by each service.

JWT Structure:

```
Header:
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "forge-key-2026-01"
}

Payload:
{
  "sub": "user-uuid",
  "iss": "https://forge.auth",
  "aud": ["forge-c2", "forge-sensors"],
  "exp": 1712345678,
  "iat": 1712342078,
  "roles": ["forge-operator", "track-viewer"],
  "clearance": "SECRET",
  "unit": "FORGE-TEAM"
}
```

Token Validation Flow:

```
  Client        API           JWT           RBAC
  Request         Gateway         Verify          Check




                                     Public         Allow/
                                     Key            Deny
```

```
Service
Response
```

OAuth2 Integration

FORGE integrates with DoD identity providers via OAuth2 for enterprise SSO:

Grant Type	Use Case	Flow
Authorization Code	Web applications	Standard OAuth2 flow with PKCE
Client Credentials	Service-to-service	Token exchange for machine identity
Resource Owner Password	Legacy systems	Deprecated - migration in progress

OAuth2 Configuration:

```yaml
oauth2:
  issuer: https://forge.auth/oauth2
  authorization_endpoint: /authorize
  token_endpoint: /token
  userinfo_endpoint: /userinfo
  jwks_uri: /.well-known/jwks.json
  scopes:
    - forge:read
    - forge:write
    - forge:admin
    - tracks:read
    - tracks:write
    - c2:execute
```

Role-Based Access Control (RBAC)

FORGE implements Kubernetes-native RBAC for authorization, with custom roles aligned to operational responsibilities.

RBAC Role Definitions:

```yaml
# FORGE Namespace Admin - Full control within forge namespace
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: forge-namespace-admin
  labels:
    nist-control: AC-3
    clearance: SECRET
rules:
  # Core resources
  - apiGroups: [""]
    resources: ["pods", "services", "configmaps", "secrets", "endpoints", "events"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
  # Applications
  - apiGroups: ["apps"]
    resources: ["deployments", "statefulsets", "daemonsets", "replicasets"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
  # Networking
  - apiGroups: ["networking.k8s.io"]
    resources: ["networkpolicies", "ingresses"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
  # Batch jobs
  - apiGroups: ["batch"]
    resources: ["jobs", "cronjobs"]
    verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

---
# FORGE Sensor Operator - Read sensor data, manage sensor deployments
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: forge-sensor-operator
  namespace: forge
  labels:
    nist-control: AC-3
    function: sensor-operations
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log", "services", "configmaps"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["create", "patch"]
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "list", "watch", "update"]
```

```yaml
---
# FORGE C2 Operator - Command and control operations
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: forge-c2-operator
  namespace: forge
  labels:
    nist-control: AC-3
    function: c2-operations
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log", "services", "endpoints"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["configmaps", "secrets"]
    verbs: ["get", "list"]
    resourceNames: ["c2-config", "c2-endpoints"]
  - apiGroups: ["apps"]
    resources: ["deployments", "statefulsets"]
    verbs: ["get", "list", "watch"]

---
# FORGE Data Pipeline - Access to Kafka and database secrets
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: forge-data-pipeline
  namespace: forge
  labels:
    nist-control: AC-3
    function: data-processing
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/exec", "pods/log"]
    verbs: ["get", "list", "watch", "create"]
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: ["get"]
    resourceNames: ["forge-kafka-credentials", "forge-db-credentials"]
```

Service Account Bindings:

Service Account	Role	Namespace	Purpose
forge-sensor	forge-sensor-operator	forge	Sensor ingestion services
forge-consumer	forge-data-pipeline	forge	Kafka/TimescaleDB consumers
forge-c2	forge-c2-operator	forge	Command & control service
forge-monitoring	forge-namespace-admin	monitoring	Metrics collection

---

7.2 Network Security

Network Policies

FORGE implements default-deny network policies with explicit allow-lists for required traffic flows.

Default Deny Policy:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: forge-default-deny
  namespace: forge
  labels:
    nist-control: SC-7
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

Kafka Access Policy:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: forge-kafka-policy
  namespace: forge
spec:
  podSelector:
    matchLabels:
```

```
      app: kafka
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              name: forge
      ports:
        - protocol: TCP
          port: 9092  # Internal broker
        - protocol: TCP
          port: 9093  # Controller
  egress:
    - to:
        - podSelector:
            matchLabels:
              app: kafka
      ports:
        - protocol: TCP
          port: 9092
```

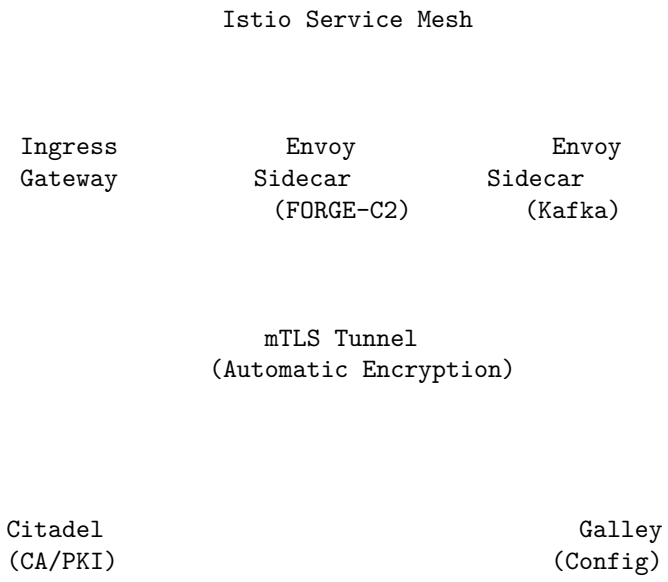
C2 API Access Policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: forge-c2-api-policy
  namespace: forge
spec:
  podSelector:
    matchLabels:
      app: forge-c2
  ingress:
    - from:
        - ipBlock:
            cidr: 10.0.0.0/8  # Internal network
        - namespaceSelector:
            matchLabels:
              name: forge
      ports:
        - protocol: TCP
          port: 8080
  egress:
    - to:
        - namespaceSelector:
            matchLabels:
              name: forge
      ports:
```

```
    - protocol: TCP
      port: 9092   # Kafka
    - protocol: TCP
      port: 5432   # TimescaleDB
```

Service Mesh Architecture

FORGE supports optional Istio service mesh deployment for enhanced traffic management and mTLS:



TLS/mTLS Configuration

Component	TLS Mode	Certificate Source
Ingress Gateway	TLS 1.3	Let's Encrypt / DoD PKI
Inter-service	mTLS	Istio Citadel / Vault PKI
Kafka	TLS	Vault PKI
PostgreSQL	TLS	Vault PKI
Vault	TLS	Self-signed (internal)

7.3 Secrets Management

HashiCorp Vault Integration

FORGE uses HashiCorp Vault for centralized secrets management, deployed in the forge-security namespace.

Vault Configuration:

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: vault
  namespace: forge-security
spec:
  serviceName: vault
  replicas: 1
  template:
    spec:
      containers:
        - name: vault
          image: hashicorp/vault:1.15
          ports:
            - containerPort: 8200
              name: http
          env:
            - name: VAULT_API_ADDR
              value: "http://vault:8200"
            - name: VAULT_LOCAL_CONFIG
              value: |
                storage "file" {
                  path = "/vault/data"
                }
                listener "tcp" {
                  address = "0.0.0.0:8200"
                  tls_disable = 1
                }
                disable_mlock = true
                ui = true
          securityContext:
            readOnlyRootFilesystem: true
            allowPrivilegeEscalation: false
            capabilities:
              drop: ["ALL"]
```

Access: - Internal: `vault.forge-security.svc.cluster.local:8200` - External NodePort: `207.244.226.151:30820`

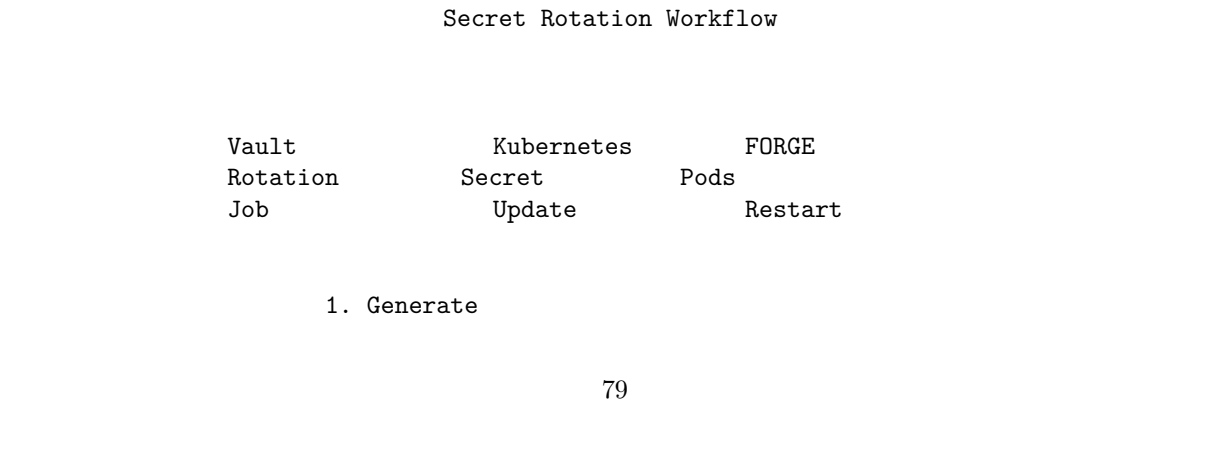
Secret Engines

Engine	Purpose	Rotation Policy
KV v2	Static secrets (API keys, passwords)	Manual / 90-day trigger
Database	Dynamic PostgreSQL credentials	24-hour TTL, auto-renew
PKI	X.509 certificate issuance	30-day TTL
Transit	Encryption as a service	N/A

Secret Paths:

```
forge/
  database/
      config/postgresql
      roles/forge-app
      creds/forge-app
  kv/
      forge-kafka-credentials
      forge-jwt-signing-key
      forge-api-keys
      forge-encryption-keys
  pki/
      root/generate/internal
      roles/forge-server
      issue/forge-server
  transit/
       keys/track-encryption
```

Secret Rotation



```
new secret
                    2. Update K8s
                       secret
                                        3. Rolling
                                           restart
```

Rotation Schedule:

Secret Type	Rotation Period	Automation
Database credentials	24 hours	Vault dynamic secrets
API keys	90 days	CronJob + Vault API
JWT signing keys	365 days	Manual with coordination
TLS certificates	30 days	cert-manager / Vault PKI
Encryption keys	Annual	Manual, with re-encryption

7.4 Data Security

Encryption at Rest

Data Store	Encryption Method	Key Management
TimescaleDB	AES-256	LUKS (disk) + Vault (column)
PostgreSQL	AES-256	LUKS (disk)
Kafka	AES-256	LUKS (disk)
MinIO	AES-256-GCM	Built-in + Vault KMS
Persistent Volumes	LUKS	Node-level encryption

Sensitive Data Classification:

Classification	Data Types	Protection
TOP SECRET	Track trajectories, engagement plans	Vault Transit encryption
SECRET	System configs, credentials	Vault KV encryption
UNCLASSIFIED	Metrics, logs	Standard encryption

Encryption in Transit

All network communication is encrypted:

Encryption in Transit Matrix

Source	Destination	Protocol	Encryption
Client	Ingress Gateway	HTTPS	TLS 1.3
Service	Service	HTTP	mTLS (Istio)
Service	Kafka	TCP	TLS 1.3
Service	PostgreSQL	TCP	TLS 1.3
Kafka	Kafka (repl)	TCP	TLS 1.3
Prometheus	Exporters	HTTP	mTLS

7.5 Pod Security Standards

FORGE enforces the Kubernetes **Restricted** Pod Security Standard across all workloads.

Namespace Labels:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: forge
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/enforce-version: latest
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: restricted
```

Security Context (Required):

```yaml
securityContext:
  runAsNonRoot: true
  runAsUser: 1000
  runAsGroup: 1000
  fsGroup: 1000
  readOnlyRootFilesystem: true
  allowPrivilegeEscalation: false
  seccompProfile:
    type: RuntimeDefault
  capabilities:
    drop:
      - ALL
```

Container Security Requirements:

Requirement	Enforcement	Exception Process
Non-root user	Enforced	Security review required
Read-only root filesystem	Enforced	tmpfs volumes for write paths
No privilege escalation	Enforced	None
Seccomp profile	RuntimeDefault	Custom profiles documented
Capability dropping	ALL dropped	Security team approval
Resource limits	Required	Quota enforced

OPA Gatekeeper Constraints:

```yaml
apiVersion: templates.gatekeeper.sh/v1
kind: ConstraintTemplate
metadata:
  name: forgepodsecurity
spec:
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package forge.podsecurity

        violation[{"msg": msg}] {
          input.review.kind.kind == "Pod"
          container := input.review.object.spec.containers[_]
          container.securityContext.privileged == true
          msg := "Privileged containers are not allowed in FORGE"
        }

        violation[{"msg": msg}] {
          input.review.kind.kind == "Pod"
          not input.review.object.spec.securityContext.runAsNonRoot
          msg := "Pods must run as non-root in FORGE"
        }
```

7.6 Audit Logging

Audit Event Categories

Category	Events Logged	Retention
Authentication	Login, logout, token refresh, failed attempts	7 years

Category	Events Logged	Retention
Authorization	RBAC decisions, access denials, privilege escalation	7 years
Data Access	Track queries, configuration reads, secret access	90 days
Data Modification	Track updates, config changes, deployments	7 years
System Events	Pod lifecycle, scaling events, errors	90 days

Audit Log Format

```
{
  "timestamp": "2026-04-09T12:30:00Z",
  "event_id": "audit-20260409-001234",
  "category": "AUTHORIZATION",
  "severity": "INFO",
  "actor": {
    "user": "operator@example.mil",
    "user_id": "user-uuid-12345",
    "roles": ["forge-operator"],
    "clearance": "SECRET"
  },
  "action": "READ_TRACK",
  "resource": {
    "type": "track",
    "id": "TRACK-2026-001",
    "classification": "SECRET"
  },
  "result": "ALLOW",
  "source": {
    "ip": "10.0.1.100",
    "service": "forge-c2",
    "namespace": "forge"
  },
  "metadata": {
    "query_params": {"track_id": "TRACK-2026-001"},
    "response_size_bytes": 1024
  }
}
```

Log Storage & Query

Infrastructure: - Primary: TimescaleDB `audit_logs` hypertable - Replica: Kafka `audit-events` topic (7-day retention) - Archive: MinIO cold storage

(7-year retention)

Query Interface:

```
-- Recent authentication failures
SELECT timestamp, actor, action, result
FROM audit_logs
WHERE category = 'AUTHENTICATION'
  AND result = 'DENY'
  AND timestamp > now() - interval '24 hours'
ORDER BY timestamp DESC;

-- Track access by user
SELECT actor->>'user' as user,
       COUNT(*) as access_count,
       array_agg(DISTINCT resource->>'type') as resources
FROM audit_logs
WHERE timestamp > now() - interval '7 days'
GROUP BY actor->>'user';
```

7.7 Compliance Requirements

DoD Risk Management Framework (RMF)

FORGE aligns with DoD RMF lifecycle phases:

Phase	Activities	FORGE Implementation
Categorize	System categorization	MODERATE baseline
Select	Control selection	NIST 800-53 rev 5
Implement	Control implementation	Documented in this section
Assess	Control assessment	Continuous ATO
Authorize	ATO decision	Pending
Monitor	Continuous monitoring	Prometheus + Audit logs

NIST 800-53 Control Mapping

Control ID	Control Name	FORGE Implementation	Status
AC-2	Account Management	RBAC, ServiceAccounts	Implemented
AC-3	Access Enforcement	RBAC policies, JWT validation	Implemented

Control ID	Control Name	FORGE Implementation	Status
AC-4	Information Flow Enforcement	Network policies, mTLS	Implemented
AC-6	Least Privilege	Role-scoped permissions	Implemented
AU-2	Audit Events	Audit logging framework	Implemented
AU-6	Audit Review	TimescaleDB query interface	Implemented
AU-11	Audit Retention	7-year retention policy	Implemented
SC-7	Boundary Protection	Network policies, firewalls	Implemented
SC-8	Transmission Confidentiality	TLS/mTLS everywhere	Implemented
SC-12	Cryptographic Key Management	Vault key management	Implemented
SC-28	Protection at Rest	Encryption at rest	Implemented
SI-2	Flaw Remediation	Vulnerability management	In Progress
SI-4	System Monitoring	Prometheus federation	Implemented
SI-7	Software Integrity	Container signing	Planned

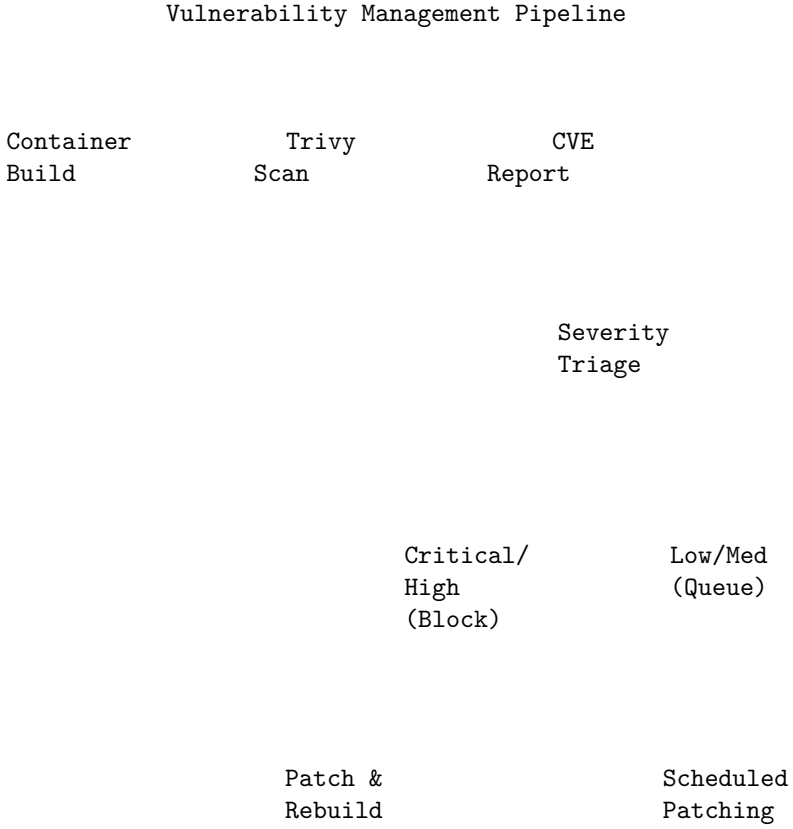
Compliance Control Mapping Table

```
                    NIST 800-53 Control Implementation Matrix

  Control   Category          Implementation            Evidence Location

  AC-2      Access Control    K8s RBAC + ServiceAcct    rbac.yaml
  AC-3      Access Control    RoleBindings + JWT        rbac.yaml
  AC-4      Access Control    NetworkPolicy             network-policy.yaml
  SC-7      System Protection  NetworkPolicy + Firewall  network-policy
  SC-8      System Protection  TLS 1.3 + mTLS           vault.yaml
  SC-12     System Protection  Vault secrets engine     vault.yaml
  SC-28     System Protection  LUKS + Vault Transit     infrastructure
  AU-2      Audit             Audit logging service     audit-config
  SI-2      System Integrity  Trivy scanning           vulnerability-mgmt
```

SI-4	System Integrity	Prometheus federation	monitoring

7.8 Vulnerability Management

Scanning Infrastructure



CVE Tracking & Response

Severity	Response Time	Action Required
Critical	24 hours	Block deployment, immediate patch
High	72 hours	Block deployment, prioritize patch

Severity	Response Time	Action Required
Medium	30 days	Document, schedule patch
Low	90 days	Document, batch patching

Scanning Schedule:

Scan Type	Frequency	Tool
Container image	Every build	Trivy
Container image	Daily	Trivy (cron)
Kubernetes manifests	Weekly	Checkov
Infrastructure as Code	Weekly	tfsec
Runtime vulnerabilities	Continuous	Falco (planned)

Patching Process

1. **CVE Identification**: Automated scanning identifies vulnerable packages
2. **Impact Assessment**: Security team evaluates exploitability and exposure
3. **Patch Development**: Development team updates dependencies
4. **Testing**: Regression and security testing in staging
5. **Deployment**: GitOps-driven rolling deployment
6. **Verification**: Post-deployment scan confirms remediation

---

Security Checklist

Pre-Deployment Security Review

```
## FORGE Security Checklist

### Authentication & Authorization
- [ ] JWT validation enabled on all services
- [ ] OAuth2 integration configured
- [ ] RBAC roles follow least privilege
- [ ] Service accounts use minimal permissions
- [ ] Token expiration policies enforced

### Network Security
- [ ] Default-deny network policies applied
- [ ] Required traffic flows explicitly allowed
- [ ] TLS 1.3 configured on all ingress points
- [ ] mTLS enabled for inter-service communication
- [ ] Network policy tests passing
```

```
### Secrets Management
- [ ] Vault deployed and sealed
- [ ] Secret paths follow naming convention
- [ ] Dynamic secrets configured where applicable
- [ ] Rotation policies defined
- [ ] No secrets in ConfigMaps or code

### Data Security
- [ ] Encryption at rest enabled
- [ ] Encryption in transit verified
- [ ] Data classification documented
- [ ] Key rotation procedures tested

### Pod Security
- [ ] Restricted PSS enforced on namespace
- [ ] Security contexts defined for all pods
- [ ] Containers run as non-root
- [ ] Read-only root filesystem configured
- [ ] Seccomp profile applied

### Audit & Monitoring
- [ ] Audit logging enabled
- [ ] Log retention policy configured
- [ ] Prometheus scraping all services
- [ ] Alerting rules defined
- [ ] Audit log queries tested

### Vulnerability Management
- [ ] Container scanning in CI/CD
- [ ] CVE triage process documented
- [ ] Patching SLAs defined
- [ ] Base images updated monthly

### Compliance
- [ ] NIST 800-53 controls mapped
- [ ] Evidence collected for assessment
- [ ] RMF package prepared
- [ ] ATO documentation ready
```

Summary

FORGE's security architecture provides defense-in-depth protection aligned with DoD requirements:

Layer	Key Controls	Status
Authentication	JWT, OAuth2, RBAC	Implemented
Network	Policies, mTLS, Service Mesh	Implemented
Secrets	Vault, rotation, dynamic creds	Implemented
Data	Encryption at rest/transit	Implemented
Pod	Restricted PSS, security contexts	Implemented
Audit	Comprehensive logging, retention	Implemented
Compliance	NIST 800-53, DoD RMF	In Progress
Vulnerability	Scanning, CVE tracking	In Progress

The security controls documented here position FORGE for a DoD Authority to Operate (ATO) under the Risk Management Framework.

Document Version: 1.0
Classification: UNCLASSIFIED//FOR OFFICIAL USE ONLY
NIST 800-53 Baseline: MODERATE
Last Updated: April 2026

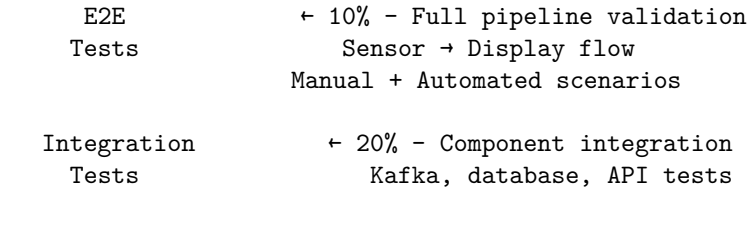
Section 8: Testing & Validation

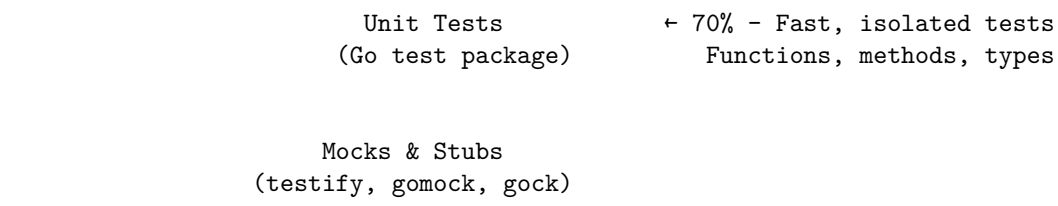
Overview

FORGE employs a comprehensive testing strategy aligned with DoD DevSecOps requirements, ensuring system reliability, performance, and compliance. The testing framework spans from unit tests to full end-to-end validation scenarios, leveraging the existing test-runner infrastructure on trooper2 and the vimi-test-suite framework.

Test Pyramid

The FORGE testing strategy follows the test pyramid model, emphasizing a strong foundation of fast unit tests with progressively fewer integration and end-to-end tests.






```
Execution Time:    Fast ←                    → Slow
Cost:              Low  ←                    → High
Confidence:        Unit ←                    → E2E
Debug Difficulty:  Easy ←                    → Hard
```

---

1. Unit Testing

Go Test Framework

FORGE components are written in Go, leveraging the built-in `testing` package for unit tests. The test framework emphasizes fast execution, comprehensive coverage, and clear test isolation.

Test Structure:

```go
// internal/track/correlator_test.go
package track

import (
    "testing"
    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

func TestCorrelator_MergeTracks(t *testing.T) {
    tests := []struct {
        name          string
        track1        *Track
        track2        *Track
        expectedConf  float64
        expectError   bool
    }{
        {
            name: "merge compatible tracks",
```

```
            track1: &Track{
                ID:         "T001",
                Position:   Position{Lat: 39.0, Lon: -121.4, Alt: 15000},
                Confidence: 0.85,
            },
            track2: &Track{
                ID:         "T002",
                Position:   Position{Lat: 39.01, Lon: -121.41, Alt: 15050},
                Confidence: 0.90,
            },
            expectedConf: 0.88,
            expectError:  false,
        },
        {
            name: "reject divergent tracks",
            track1: &Track{
                ID:         "T001",
                Position:   Position{Lat: 39.0, Lon: -121.4, Alt: 15000},
            },
            track2: &Track{
                ID:         "T002",
                Position:   Position{Lat: 45.0, Lon: -100.0, Alt: 20000},
            },
            expectError: true,
        },
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            correlator := NewCorrelator(DefaultConfig())
            result, err := correlator.MergeTracks(tt.track1, tt.track2)

            if tt.expectError {
                require.Error(t, err)
            } else {
                require.NoError(t, err)
                assert.InDelta(t, tt.expectedConf, result.Confidence, 0.05)
            }
        })
    }
}
```

Coverage Requirements

Component	Minimum Coverage	Critical Paths	Notes
Track Correlator	80%	95%	Multi-sensor fusion logic
JREAP Formatter	85%	100%	DoD messaging standard
Link 16 Formatter	85%	100%	MIL-STD-6016 compliance
Sensor Normalizer	75%	90%	Data transformation
Kafka Consumer	70%	85%	Message processing
TimescaleDB Writer	70%	85%	Data persistence
WebSocket Server	65%	80%	Client communication
Alert Generator	80%	95%	Threat notification

Running Tests with Coverage:

```
# Run all tests with coverage report
go test -v -race -coverprofile=coverage.out ./...

# Generate HTML coverage report
go tool cover -html=coverage.out -o coverage.html

# View coverage summary
go tool cover -func=coverage.out | tail -1

# Output: total: (statements) 82.5%
```

Mocking Strategies

FORGE uses three primary mocking approaches depending on the integration point:

1. Interface Mocking (testify/mock):

```
// internal/kafka/mock_consumer.go
package kafka

import (
    "github.com/stretchr/testify/mock"
)

type MockConsumer struct {
    mock.Mock
}

func (m *MockConsumer) Consume(topic string) (*Message, error) {
```

```
    args := m.Called(topic)
    if msg := args.Get(0); msg != nil {
        return msg.(*Message), args.Error(1)
    }
    return nil, args.Error(1)
}

func (m *MockConsumer) Commit() error {
    return m.Called().Error(0)
}
```

2. HTTP Mocking (gock):

```
// internal/api/client_test.go
package api

import (
    "testing"
    "github.com/h2non/gock"
    "github.com/stretchr/testify/assert"
)

func TestExternalAPIClient(t *testing.T) {
    defer gock.Off()

    gock.New("https://external-feed.mil").
        Get("/tracks").
        Reply(200).
        JSON(map[string]interface{}{
            "tracks": []map[string]interface{}{
                {"id": "EXT001", "lat": 39.0, "lon": -121.4},
            },
        })

    client := NewClient("https://external-feed.mil")
    tracks, err := client.FetchTracks()

    assert.NoError(t, err)
    assert.Len(t, tracks, 1)
}
```

3. Database Mocking (testcontainers or sqlmock):

```
// internal/storage/timescaledb_test.go
package storage

import (
    "testing"
```

```
    "github.com/DATA-DOG/go-sqlmock"
    "github.com/stretchr/testify/require"
)

func TestTimescaleDB_InsertTrack(t *testing.T) {
    db, mock, err := sqlmock.New()
    require.NoError(t, err)
    defer db.Close()

    mock.ExpectBegin()
    mock.ExpectExec(`INSERT INTO tracks`).
        WithArgs("T001", 39.0, -121.4, 15000.0, 0.85).
        WillReturnResult(sqlmock.NewResult(1, 1))
    mock.ExpectCommit()

    tsdb := &TimescaleDB{db: db}
    err = tsdb.InsertTrack(context.Background(), &Track{
        ID: "T001", Lat: 39.0, Lon: -121.4, Alt: 15000, Confidence: 0.85,
    })

    require.NoError(t, err)
    require.NoError(t, mock.ExpectationsWereMet())
}
```

---

2. Integration Testing

Component Integration

Integration tests validate the interactions between FORGE components, focusing on message flow through Kafka and data persistence to TimescaleDB.

Integration Test Configuration:

```yaml
# configs/integration-test.yaml
integration:
  kafka:
    broker: "kafka.vimi-test.svc.cluster.local:9092"
    topics:
      - forge.sensors.raw
      - forge.tracks
      - forge.link16
    consumer_group: "test-integration"

  database:
    host: "timescaledb.vimi-test.svc.cluster.local"
    port: 5432
```

```
    name: "forge_test"
    user: "test_runner"
    password: "${DB_PASSWORD}"

  timeout:
    connection: 30s
    message_wait: 60s
    cleanup: 10s
```

Kafka Test Containers

FORGE uses testcontainers for isolated Kafka testing:

```
// internal/integration/kafka_test.go
//go:build integration

package integration

import (
    "context"
    "testing"
    "time"

    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/wait"
    "github.com/segmentio/kafka-go"
)

func TestKafkaIntegration(t *testing.T) {
    ctx := context.Background()

    // Start Kafka container
    kafkaContainer, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContai
        ContainerRequest: testcontainers.ContainerRequest{
            Image: "confluentinc/cp-kafka:7.5.0",
            ExposedPorts: []string{"9093/tcp"},
            Env: map[string]string{
                "KAFKA_BROKER_ID":                         "1",
                "KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR": "1",
                "KAFKA_LISTENER_SECURITY_PROTOCOL_MAP":   "PLAINTEXT:PLAINTEXT",
                "KAFKA_ADVERTISED_LISTENERS":             "PLAINTEXT://localhost:9092",
            },
            WaitingFor: wait.ForListeningPort("9093/tcp"),
        },
        Started: true,
    })
```

```
    require.NoError(t, err)
    defer kafkaContainer.Terminate(ctx)

    // Get container host/port
    host, err := kafkaContainer.Host(ctx)
    require.NoError(t, err)
    port, err := kafkaContainer.MappedPort(ctx, "9093")
    require.NoError(t, err)

    // Create topic and test message flow
    broker := fmt.Sprintf("%s:%s", host, port.Port())
    conn, err := kafka.DialLeader(ctx, "tcp", broker, "forge.sensors.raw", 0)
    require.NoError(t, err)
    defer conn.Close()

    // Publish test message
    err = conn.WriteMessages(kafka.Message{
        Key:   []byte("test-track"),
        Value: []byte(`{"id":"T001","lat":39.0,"lon":-121.4,"alt":15000}`),
    })
    require.NoError(t, err)

    // Consume and validate
    batch, err := conn.ReadBatch(0, 1e6)
    require.NoError(t, err)
    defer batch.Close()

    var msg kafka.Message
    for {
        msg, err = batch.ReadMessage()
        if err != nil {
            break
        }
        assert.Contains(t, string(msg.Value), "T001")
    }
}
```

Database Integration Tests

TimescaleDB integration tests use a separate test schema:

```
// internal/integration/database_test.go
//go:build integration

package integration
```

```
import (
    "testing"
    "time"

    "github.com/jmoiron/sqlx"
    _ "github.com/lib/pq"
    "github.com/stretchr/testify/require"
)

func TestTimescaleDB_TrackStorage(t *testing.T) {
    dsn := "postgres://test_runner:test@localhost:5432/forge_test?sslmode=disable"
    db, err := sqlx.Connect("postgres", dsn)
    require.NoError(t, err)
    defer db.Close()

    // Create hypertable
    _, err = db.Exec(`
        CREATE TABLE IF NOT EXISTS test_tracks (
            time        TIMESTAMPTZ NOT NULL,
            track_id    TEXT NOT NULL,
            latitude    DOUBLE PRECISION,
            longitude   DOUBLE PRECISION,
            altitude    DOUBLE PRECISION,
            confidence  DOUBLE PRECISION,
            sensor_id   TEXT
        );
        SELECT create_hypertable('test_tracks', 'time', if_not_exists => true);
    `)
    require.NoError(t, err)

    // Insert test data
    now := time.Now()
    for i := 0; i < 100; i++ {
        _, err = db.Exec(`
            INSERT INTO test_tracks (time, track_id, latitude, longitude, altitude, confider
            VALUES ($1, $2, $3, $4, $5, $6, $7)
        `, now.Add(time.Duration(i)*time.Second), fmt.Sprintf("T%03d", i), 39.0+float64(i)*(
        require.NoError(t, err)
    }

    // Query and validate
    var count int
    err = db.Get(&count, "SELECT COUNT(*) FROM test_tracks WHERE track_id LIKE 'T%'")
    require.NoError(t, err)
    require.Equal(t, 100, count)
```

```
    // Cleanup
    _, err = db.Exec("DROP TABLE test_tracks")
    require.NoError(t, err)
}
```

---

3. End-to-End Testing

Full Pipeline Validation

End-to-end tests validate the complete sensor-to-display data flow using the vimi-test-suite:

```yaml
# test-scenarios/e2e_boost_phase.yaml
name: e2e_boost_phase_detection
description: Full pipeline validation from OPIR sensor to Link 16 output
duration: 120s

infrastructure:
  namespace: vimi-test
  components:
    - opir-simulator
    - radar-simulator
    - track-correlator
    - data-consumer
    - link16-formatter
    - kafka
    - timescaledb

steps:
  - name: deploy_infrastructure
    action: helm_upgrade
    chart: charts/vimi-test-suite
    values:
      opir.enabled: true
      radar.enabled: true
      correlator.enabled: true
      consumer.enabled: true
      link16.enabled: true
    timeout: 300s

  - name: wait_for_ready
    action: kubectl_wait
    condition: ready
    selector: app.kubernetes.io/part-of=vimi-test-suite
    timeout: 120s
```

```yaml
- name: inject_scenario
  action: kubectl_apply
  manifest: scenarios/icbm_single_launch.yaml
  timeout: 10s

- name: validate_detection
  action: kafka_consume
  topic: forge.sensors.raw
  timeout: 30s
  assertions:
    - message_count: ">0"
    - detection_latency_ms: "<500"
    - sensor_id: "SBIRS-GEO-1"

- name: validate_correlation
  action: kafka_consume
  topic: forge.tracks
  timeout: 60s
  assertions:
    - message_count: ">0"
    - correlation_confidence: ">0.85"
    - position_error_km: "<10"

- name: validate_link16_output
  action: kafka_consume
  topic: forge.link16
  timeout: 30s
  assertions:
    - j12_format: valid
    - j70_format: valid
    - track_number_assigned: true

- name: validate_storage
  action: database_query
  query: "SELECT COUNT(*) FROM tracks WHERE time > NOW() - INTERVAL '2 minutes'"
  expected: ">0"

- name: cleanup
  action: helm_uninstall
  release: vimi-test
  timeout: 60s
```

Sensor to Display Flow

E2E TEST EXECUTION FLOW

1. SETUP PHASE (60s timeout)

Deploy vimi-test suite via Helm
Create namespace vimi-test
Deploy Kafka + TimescaleDB
Deploy all simulators + processors

2. SCENARIO INJECTION (5s)

Apply scenario YAML manifest
ICBM launch from [39.0°, 127.0°]
Target trajectory pre-defined
Sensors receive detection request

3. VALIDATION PHASE (parallel)

Kafka Topic forge.sensors .raw
Kafka Topic forge.tracks
TimescaleDB Query

ASSERTION ENGINE
Detection latency < 500ms
Correlation confidence > 0.85
Position error < 10km
J12/J70/J73 valid format
Track number assigned

4. CLEANUP PHASE (60s)

Helm uninstall vimi-test
Remove vimi-test namespace
Clean up Kafka topics
Purge test database

---

4. Performance Testing

Load Testing with k6

FORGE uses k6 for load testing, integrated with the test-runner infrastructure on trooper2:

```
// scripts/load/track_throughput.js
import { check } from 'k6';
import { Kafka } from 'k6/x/kafka';

const kafka = new Kafka({
    brokers: ['kafka.forge.svc.cluster.local:9092'],
});

export const options = {
    scenarios: {
        // Ramp-up scenario for baseline
        ramp_up: {
            executor: 'ramping-vus',
            startVUs: 0,
            stages: [
                { duration: '30s', target: 10 },
                { duration: '1m', target: 50 },
                { duration: '30s', target: 100 },
                { duration: '1m', target: 100 },
                { duration: '30s', target: 0 },
            ],
            gracefulRampDown: '30s',
        },
        // Stress test
        stress: {
            executor: 'constant-vus',
            vus: 200,
            duration: '5m',
            startTime: '5m',
        },
        // Spike test
        spike: {
            executor: 'ramping-vus',
            startVUs: 0,
            stages: [
                { duration: '10s', target: 500 },
                { duration: '30s', target: 500 },
                { duration: '10s', target: 0 },
            ],
            startTime: '10m',
```

```
        },
    },
    thresholds: {
        'http_req_duration': ['p(95)<500'], // 95% under 500ms
        'http_req_failed': ['rate<0.01'],   // Less than 1% failures
        'kafka_write_duration': ['p(99)<200'], // 99% under 200ms
        'iterations': ['count>10000'],      // Minimum throughput
    },
};

export default function () {
    // Generate synthetic track
    const track = {
        id: `TRACK-${__VU}-${__ITER}`,
        timestamp: Date.now(),
        position: {
            lat: 39.0 + Math.random() * 10 - 5,
            lon: -121.4 + Math.random() * 10 - 5,
            alt: 15000 + Math.random() * 5000,
        },
        velocity: {
            vx: Math.random() * 1000,
            vy: Math.random() * 1000,
            vz: Math.random() * 100,
        },
        confidence: 0.85 + Math.random() * 0.15,
        sensor_id: `SENSOR-${Math.floor(Math.random() * 5)}`,
        track_type: 'ballistic',
    };

    // Produce to Kafka
    const result = kafka.produce({
        topic: 'forge.sensors.raw',
        key: track.id,
        value: JSON.stringify(track),
    });

    check(result, {
        'track published': (r) => r.error_code === 0,
        'latency acceptable': (r) => r.latency < 200,
    });
}

export function teardown() {
    kafka.close();
}
```

Running k6 Tests:

```
# Local development
k6 run scripts/load/track_throughput.js

# Against Kubernetes deployment
k6 run --out influxdb=http://influxdb.forge.svc:8086/k6 \
       scripts/load/track_throughput.js

# Cloud execution via test-runner
kubectl apply -f k6-runner-job.yaml -n forge
kubectl logs -f job/k6-track-throughput -n forge
```

Stress Testing

Stress tests validate system behavior under extreme load:

```
# test-scenarios/stress_high_volume.yaml
name: stress_high_volume
description: Validate system behavior at 10x normal load

parameters:
  normal_rate: 1000          # tracks/second
  stress_rate: 10000          # tracks/second (10x)
  duration: 300s              # 5 minutes at peak

infrastructure:
  kafka:
    partitions: 12
    replication_factor: 3
  consumers:
    replicas: 10
    consumer_threads: 20

metrics:
  - name: track_ingestion_rate
    type: counter
    threshold: ">9500/s"
  - name: kafka_lag
    type: gauge
    threshold: "<100000"
  - name: consumer_latency_p99
    type: histogram
    threshold: "<500ms"
  - name: timescaledb_write_latency
    type: histogram
    threshold: "<100ms"
```

```
  - name: memory_usage
    type: gauge
    threshold: "<85%"
  - name: cpu_usage
    type: gauge
    threshold: "<90%"

validation:
  - no_message_loss: true
  - no_consumer_restart: true
  - kafka_lag_drained_within: "60s"
  - all_partitions_rebalanced: true
```

---

5. Chaos Engineering

Failure Injection

FORGE uses chaos engineering principles to validate resilience. The chaos testing framework integrates with Kubernetes fault injection.

```
# test-scenarios/chaos_kafka_failure.yaml
name: chaos_kafka_broker_failure
description: Validate system resilience to Kafka broker failure

infrastructure:
  namespace: forge
  chaos_mesh: true

steps:
  - name: establish_baseline
    action: metrics_capture
    duration: 60s
    metrics:
      - track_ingestion_rate
      - consumer_latency_p99
      - error_rate

  - name: inject_chaos
    action: chaos_mesh
    kind: PodChaos
    spec:
      action: pod-kill
      mode: one
      selector:
        namespaces: [forge]
```

```
      labelSelectors:
        app.kubernetes.io/name: kafka
    gracePeriod: 0

- name: observe_recovery
  action: metrics_capture
  duration: 120s
  expectations:
    - consumer_reconnect_time: "<30s"
    - message_loss: "0"
    - track_ingestion_rate_recovery: ">90% of baseline"

- name: validate_no_data_loss
  action: database_query
  query: |
    SELECT COUNT(*) FROM tracks
    WHERE time BETWEEN NOW() - INTERVAL '3 minutes' AND NOW()
  expected: ">0"

- name: cleanup_chaos
  action: chaos_mesh_cleanup
```

Resilience Validation

Chaos test scenarios for FORGE components:

Scenario	Target	Expected Behavior
Kafka broker kill	Kafka cluster	Consumers reconnect, no data loss
Network partition (partial)	Network	Remaining brokers handle traffic
Consumer OOM	forge-consumer	Pod restart, resume from last offset
TimescaleDB connection loss	Database	Buffer messages, retry with backoff
DNS resolution failure	CoreDNS	Cached connections continue
Node failure	Kubernetes node	Pods rescheduled, state restored
High CPU throttle	All pods	Graceful degradation, no crashes

Chaos Mesh Configuration:

```
# chaos/network-partition.yaml
apiVersion: chaos-mesh.org/v1alpha1
kind: NetworkChaos
metadata:
```

```
  name: forge-kafka-partition
  namespace: forge
spec:
  action: partition
  mode: all
  selector:
    namespaces: [forge]
    labelSelectors:
      app.kubernetes.io/name: forge-consumer
  direction: to
  target:
    selector:
      namespaces: [forge]
      labelSelectors:
        app.kubernetes.io/name: kafka
    mode: one
  duration: "60s"
```

---

6. Regression Testing

Automated Test Suites

FORGE maintains comprehensive regression test suites executed on every commit:

```
# .gitlab-ci.yml - FORGE regression pipeline
stages:
  - lint
  - unit
  - integration
  - e2e
  - performance
  - regression

# Lint stage
lint:
  stage: lint
  tags: [trooper2]
  script:
    - golangci-lint run --config .golangci.yml ./...
    - go fmt ./...
    - go vet ./...
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
```

```yaml
# Unit tests
unit-test:
  stage: unit
  tags: [trooper2]
  script:
    - go test -v -race -coverprofile=coverage.out ./...
    - go tool cover -func=coverage.out | grep total
  coverage: '/total:\s+\(statements\)\s+(\d+\.?\d*)%/'
  artifacts:
    paths:
      - coverage.out
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage.out

# Integration tests
integration-test:
  stage: integration
  tags: [trooper2]
  services:
    - name: confluentinc/cp-kafka:7.5.0
      alias: kafka
    - name: timescale/timescaledb:latest-pg15
      alias: timescaledb
  variables:
    KAFKA_BROKER: "kafka:9092"
    DATABASE_URL: "postgres://test@timescaledb:5432/forge_test"
  script:
    - go test -tags=integration -v ./internal/integration/...
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == "main"

# End-to-end tests
e2e-test:
  stage: e2e
  tags: [test-runner]
  script:
    - kubectl create namespace forge-e2e-${CI_COMMIT_SHA}
    - helm upgrade --install forge-e2e ./charts/forge
      --namespace forge-e2e-${CI_COMMIT_SHA}
      --set replicaCount=1
      --wait --timeout 300s
    - ./scripts/run-e2e-tests.sh
    - helm uninstall forge-e2e --namespace forge-e2e-${CI_COMMIT_SHA}
```

```
    - kubectl delete namespace forge-e2e-${CI_COMMIT_SHA}
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
    - if: $CI_COMMIT_TAG

# Performance regression
performance-regression:
  stage: performance
  tags: [test-runner]
  script:
    - k6 run scripts/load/track_throughput.js --out json=performance-results.json
    - ./scripts/compare-performance.sh performance-results.json baseline.json
  artifacts:
    paths:
      - performance-results.json
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

# Full regression suite
regression-suite:
  stage: regression
  tags: [test-runner]
  script:
    - ./scripts/regression/run-all-scenarios.sh
  artifacts:
    paths:
      - regression-results/
      - junit/
    reports:
      junit: junit/*.xml
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
    - if: $CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/
```

Test Coverage Requirements

Test Level	Coverage Target	Critical Path Coverage	Execution Time
Unit	80%	95%	<2 minutes
Integration	70%	85%	<5 minutes
End-to-End	60%	80%	<15 minutes
Performance	N/A	Key scenarios	<10 minutes
Chaos	N/A	Failure modes	<30 minutes
Regression	N/A	All scenarios	<60 minutes

7. Validation Criteria

Acceptance Criteria

Each FORGE component has defined acceptance criteria:

Track Correlator:

Criterion	Threshold	Measurement Method
Correlation latency	<100ms P99	Prometheus histogram
Position error	<10km RMSE	Ground truth comparison
Velocity error	<0.5km/s	Kalman filter validation
Correlation confidence	>0.85	Algorithm output
Multi-sensor merge	>95% success	Scenario test coverage

JREAP Formatter:

Criterion	Threshold	Measurement Method
Message format	100% valid	MIL-STD-3011 schema validation
Encoding time	<10ms	Benchmark test
Field population	100% required	Schema compliance check
Track number assignment	Unique	Database constraint

Link 16 Formatter:

Criterion	Threshold	Measurement Method
J12 format	100% valid	MIL-STD-6016 validation
J70 format	100% valid	MIL-STD-6016 validation
J73 format	100% valid	MIL-STD-6016 validation
Message rate	>100 msg/s	Load test

Pass/Fail Thresholds

System-level pass/fail criteria:

```
# validation/system-thresholds.yaml
system_validation:
  throughput:
    track_ingestion_rate:
      minimum: 1000
```

```yaml
    unit: tracks/second
    severity: critical
  kafka_publish_latency:
    maximum: 100
    unit: milliseconds
    percentile: 99
    severity: warning

latency:
  end_to_end_processing:
    maximum: 500
    unit: milliseconds
    percentile: 95
    severity: critical
  sensor_to_track:
    maximum: 200
    unit: milliseconds
    percentile: 99
    severity: critical

reliability:
  message_loss:
    maximum: 0
    unit: count
    severity: critical
  consumer_restart_count:
    maximum: 2
    unit: count
    time_window: 1h
    severity: warning

data_quality:
  track_position_accuracy:
    maximum: 10
    unit: kilometers
    severity: critical
  track_velocity_accuracy:
    maximum: 0.5
    unit: km/s
    severity: warning
  correlation_confidence:
    minimum: 0.85
    severity: warning

compliance:
  jreap_message_validity:
```

```
      minimum: 100
      unit: percent
      severity: critical
    link16_message_validity:
      minimum: 100
      unit: percent
      severity: critical
```

---

8. Test Data Management

Synthetic Data Generation

FORGE uses synthetic data generators for realistic test scenarios:

```go
// internal/testutil/generator.go
package testutil

import (
    "math/rand"
    "time"
)

type TrackGenerator struct {
    rng        *rand.Rand
    baseTime   time.Time
    launchSite GeoPoint
    targetSite GeoPoint
}

type GeoPoint struct {
    Lat float64
    Lon float64
    Alt float64
}

func NewTrackGenerator() *TrackGenerator {
    return &TrackGenerator{
        rng:      rand.New(rand.NewSource(time.Now().UnixNano())),
        baseTime: time.Now(),
        launchSite: GeoPoint{Lat: 39.0, Lon: 127.0, Alt: 0},
        targetSite: GeoPoint{Lat: 40.0, Lon: -121.4, Alt: 0},
    }
}

// GenerateBallisticTrack creates a realistic ballistic missile trajectory
```

```
func (g *TrackGenerator) GenerateBallisticTrack(id string, startTime time.Time) []*Track {
    var trajectory []*Track

    // ICBM flight profile: boost (180s), midcourse (1200s), terminal (120s)
    phases := []struct {
        name     string
        duration time.Duration
        dt       time.Duration
    }{
        {"boost", 180 * time.Second, 100 * time.Millisecond},
        {"midcourse", 1200 * time.Second, 200 * time.Millisecond},
        {"terminal", 120 * time.Second, 50 * time.Millisecond},
    }

    t := startTime
    for _, phase := range phases {
        end := t.Add(phase.duration)
        for t.Before(end) {
            track := g.computeTrajectoryPoint(id, t, phase.name)
            trajectory = append(trajectory, track)
            t = t.Add(phase.dt)
        }
    }

    return trajectory
}

func (g *TrackGenerator) computeTrajectoryPoint(id string, t time.Time, phase string) *Track
    // Simplified trajectory calculation
    elapsed := t.Sub(g.baseTime).Seconds()

    // Boost phase: accelerating upward
    // Midcourse: ballistic arc
    // Terminal: decelerating downward

    var lat, lon, alt, vx, vy, vz float64

    switch phase {
    case "boost":
        lat = g.launchSite.Lat + elapsed*0.01
        lon = g.launchSite.Lon + elapsed*0.005
        alt = elapsed * elapsed * 10 // Quadratic rise
        vx = 0.01
        vy = 0.005
        vz = elapsed * 10
```

```
    case "midcourse":
        // Ballistic trajectory
        tNorm := (elapsed - 180) / 1200
        lat = g.launchSite.Lat + tNorm*(g.targetSite.Lat-g.launchSite.Lat)
        lon = g.launchSite.Lon + tNorm*(g.targetSite.Lon-g.launchSite.Lon)
        alt = 1500000 - (tNorm-0.5)*(tNorm-0.5)*4000000 // Parabolic arc
        vx = (g.targetSite.Lat - g.launchSite.Lat) / 1200
        vy = (g.targetSite.Lon - g.launchSite.Lon) / 1200
        vz = 0

    case "terminal":
        tNorm := (elapsed - 1380) / 120
        lat = g.targetSite.Lat - (1-tNorm)*(g.targetSite.Lat-g.launchSite.Lat)*0.1
        lon = g.targetSite.Lon - (1-tNorm)*(g.targetSite.Lon-g.launchSite.Lon)*0.1
        alt = 1500000 * (1 - tNorm)
        vx = (g.targetSite.Lat - g.launchSite.Lat) / 1200 * 1.5
        vy = (g.targetSite.Lon - g.launchSite.Lon) / 1200 * 1.5
        vz = -elapsed * 100
    }

    return &Track{
        ID:         id,
        Timestamp:  t,
        Position:   Position{Lat: lat, Lon: lon, Alt: alt},
        Velocity:   Velocity{Vx: vx, Vy: vy, Vz: vz},
        Confidence: 0.85 + g.rng.Float64()*0.15,
        SensorID:   "SYNTHETIC",
        TrackType:  "ballistic",
    }
}
```

Test Fixtures

Test fixtures provide pre-configured scenarios:

```yaml
# test-fixtures/scenarios/multi_sensor_correlation.yaml
name: multi_sensor_correlation
description: OPIR + Radar correlation test case
version: 1.0.0

sensors:
  - id: SBIRS-GEO-1
    type: opir
    position: [0.0, -95.0, 35786000]  # GEO position
    bands: [SWIR, MWIR]
    detection_prob: 0.85
```

```yaml
    false_alarm_rate: 1e-6

  - id: UEWR-BEALE
    type: radar
    position: [39.1376, -121.3547, 0]
    frequency: 0.44  # GHz
    max_range: 5000  # km
    scan_rate: 0.2   # Hz

targets:
  - id: ICBM-001
    type: icbm
    launch_point: [39.0, 127.0, 0]
    launch_time: "2026-04-01T00:00:00Z"
    trajectory: ballistic_standard
    rv_count: 1
    decoy_count: 2

timeline:
  - time: "T+0s"
    event: launch_detected
    sensors: [SBIRS-GEO-1]
    expected_confidence: 0.80

  - time: "T+60s"
    event: radar_acquisition
    sensors: [UEWR-BEALE]
    expected_confidence: 0.90

  - time: "T+120s"
    event: track_correlation
    sensors: [SBIRS-GEO-1, UEWR-BEALE]
    expected_confidence: 0.92
    expected_position_error_km: 8.5

  - time: "T+180s"
    event: track_confirmed
    sensors: [SBIRS-GEO-1, UEWR-BEALE]
    expected_confidence: 0.95
    expected_position_error_km: 5.2

expected_outputs:
  - topic: forge.sensors.raw
    message_count_min: 1000
    message_count_max: 1500
```

```
  - topic: forge.tracks
    message_count_min: 100
    message_count_max: 200
    track_id_unique: 1
    correlation_confidence_min: 0.85

  - topic: forge.link16
    message_count_min: 50
    message_count_max: 100
    j12_valid: true
    j70_valid: true

validation:
  detection_latency_max_ms: 500
  correlation_latency_max_ms: 100
  position_error_max_km: 10
  confidence_min: 0.85
```

Test Infrastructure Reference

test-runner (trooper2)

The test-runner component on trooper2 provides:

- **GitLab CI Runner**: Shell executor for Kubernetes-native tests
- **k6 Execution**: Load testing framework
- **Chaos Mesh**: Kubernetes chaos engineering
- **Metrics Collection**: Prometheus scraping and analysis

```
# test-runner deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-runner
  namespace: forge
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-runner
  template:
    metadata:
      labels:
        app: test-runner
    spec:
      containers:
```

```yaml
      - name: test-runner
        image: forge/test-runner:latest
        env:
          - name: KAFKA_BROKER
            value: "kafka.forge.svc.cluster.local:9092"
          - name: DATABASE_URL
            valueFrom:
              secretKeyRef:
                name: forge-db-credentials
                key: url
        volumeMounts:
          - name: test-scenarios
            mountPath: /scenarios
          - name: test-results
            mountPath: /results
    volumes:
      - name: test-scenarios
        configMap:
          name: test-scenarios
      - name: test-results
        persistentVolumeClaim:
          claimName: test-results-pvc
```

vimi-test-suite Integration

The vimi-test-suite provides DoD-compliant test scenarios:

```bash
# Deploy test suite
helm install vimi-test ./charts/vimi-test-suite \
  --namespace vimi-test \
  --create-namespace \
  --set opir.enabled=true \
  --set radar.enabled=true \
  --set correlator.enabled=true \
  --set testRunner.enabled=true

# Execute specific test scenario
kubectl apply -f scenarios/boost_phase_detection.yaml -n vimi-test

# Monitor test execution
kubectl logs -f deployment/test-runner -n vimi-test

# Collect results
kubectl cp vimi-test/test-runner-xxx:/results ./test-results
```

Summary

FORGE's testing and validation framework ensures system reliability, performance, and DoD compliance through:

1. **Comprehensive Unit Tests** with 80%+ coverage on critical components
2. **Integration Tests** using test containers for Kafka and TimescaleDB
3. **End-to-End Tests** validating complete sensor-to-display pipelines
4. **Performance Tests** with k6 for load and stress testing
5. **Chaos Engineering** for resilience validation
6. **Automated Regression** via GitLab CI on every commit
7. **Clear Validation Criteria** with measurable thresholds
8. **Synthetic Data Generation** for realistic test scenarios

The test infrastructure leverages existing components (test-runner on trooper2, vimi-test-suite Helm charts) to provide consistent, repeatable testing across all environments.

Section 8: Testing & Validation - FORGE System Design Version 1.0.0

Section 9: Future Development

Overview

FORGE's modular architecture provides a foundation for continuous enhancement. This section outlines planned capabilities, architectural evolution, integration pathways, and research directions that will extend the platform's utility for missile defense simulation and operational planning.

Planned Features

Multi-Sensor Fusion Enhancement

The current track correlation system fuses data from OPIR satellites, radar systems, and external feeds (AIS/ADS-B) through the VIMI correlator. Future development will implement advanced fusion techniques:

Probabilistic Data Association (PDA): Replace the current nearest-neighbor correlation with a probabilistic approach that weights multiple track hypotheses, reducing miscorrelation in dense target environments. PDA maintains multiple association hypotheses and computes weighted state estimates, improving tracking accuracy when targets maneuver or cross paths.

Track Quality Scoring: Each sensor contribution will receive a dynamic quality score based on sensor type, geometric dilution of precision (GDOP), environmental conditions, and historical accuracy. The fusion algorithm will weight

track updates proportionally to quality scores, automatically favoring more reliable sources.

Multi-Hypothesis Tracking (MHT): For complex scenarios with overlapping tracks, MHT will maintain multiple track hypotheses over time, pruning low-probability branches as new data arrives. This addresses the "ghost track" problem where a single target generates multiple correlated tracks from different sensor perspectives.

Machine Learning-Based Correlation

Machine learning will augment rule-based correlation algorithms for improved accuracy and reduced false positives:

Anomaly Detection: Neural networks trained on historical track data will identify anomalous patterns that may indicate sensor malfunction, deliberate deception, or previously uncharacterized threat behaviors. The system will flag unusual correlation confidence scores for operator review.

Adaptive Learning: Online learning algorithms will continuously refine correlation parameters based on operator feedback. When operators accept or reject correlation decisions, the system learns to adjust future behavior, reducing the cognitive burden of manual correlation overrides.

Predictive Association: Sequence-to-sequence models will predict likely track associations before all sensor data arrives, enabling faster correlation in time-critical scenarios. The system will pre-position hypotheses and validate them as additional sensor data becomes available.

Predictive Tracking

Current track state estimation uses Kalman filters with constant-velocity or constant-acceleration models. Future enhancements will incorporate:

Maneuver Detection: Interactive Multiple Model (IMM) filters will detect and adapt to target maneuvers in real-time. The system will transition between motion models (constant velocity, turning, accelerating) based on observed behavior, maintaining track continuity through aggressive maneuvers.

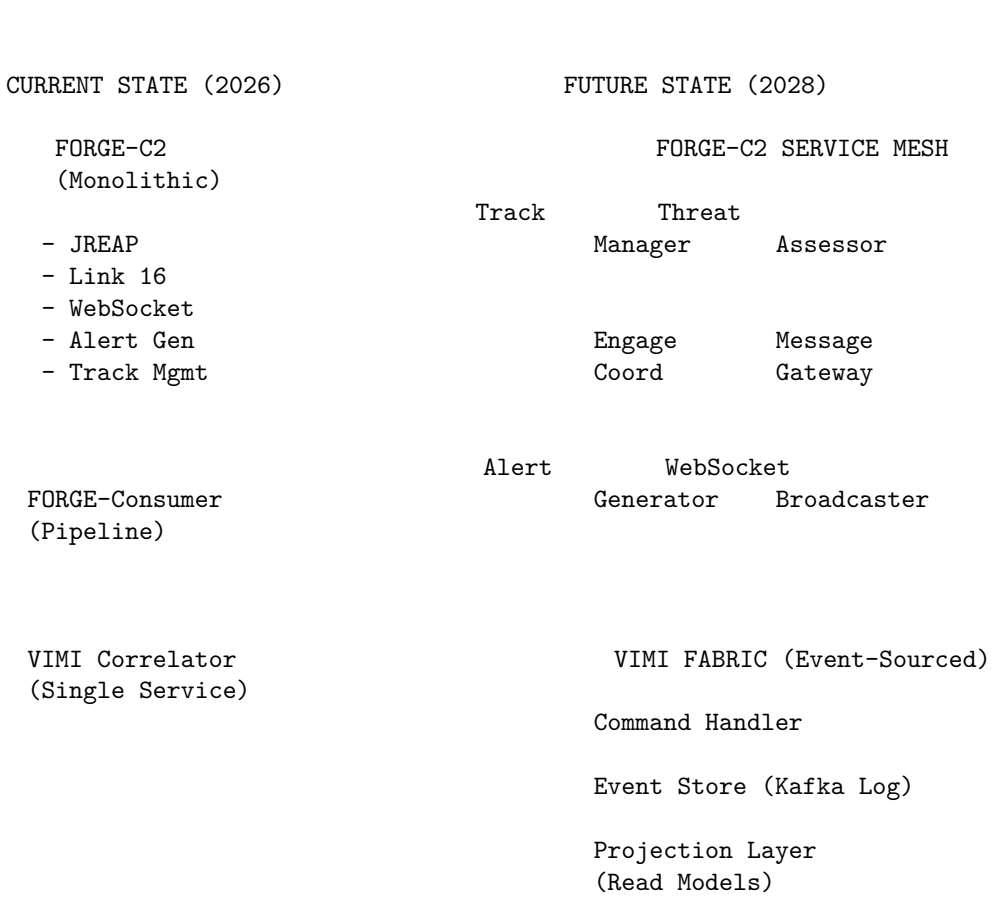
Impact Point Prediction: Ballistic trajectory prediction will estimate impact points and times for threat assessment. The system will propagate track states forward using orbital mechanics for exo-atmospheric phases and atmospheric drag models for terminal phases.

Launch Point Estimation: Back-propagation algorithms will estimate launch locations from track geometry, enabling attribution and threat characterization. This supports counter-fire planning and intelligence analysis.

Architecture Evolution

Microservices Refinement

FORGE's current microservices architecture will evolve toward finer-grained components:



Service Decomposition: FORGE-C2 will decompose into specialized services (Track Manager, Threat Assessor, Engagement Coordinator, Message Gateway, Alert Generator). Each service will own its data and communicate through well-defined gRPC interfaces.

Domain-Driven Design: Services will align with bounded contexts from the

missile defense domain: Track Management, Threat Assessment, Engagement Planning, and External Communications. This enables independent deployment cycles and team ownership.

Event Sourcing and CQRS

The current architecture uses Kafka as a message broker with stateful services maintaining local caches. Future development will implement full event sourcing:

Event Store: All state changes will be captured as immutable events stored in Kafka compacted topics. The event log becomes the system of record, enabling event replay, temporal queries, and audit trails.

Command Query Responsibility Segregation (CQRS): Write paths (commands) will be separated from read paths (queries). Command handlers validate and emit events. Query handlers subscribe to events and maintain optimized read models for specific use cases (track display, alert generation, historical analysis).

Benefits: Event sourcing provides complete audit history for after-action review, enables point-in-time system state reconstruction, supports event replay for debugging and testing, and decouples services through event-driven communication.

Integration Roadmap

External C2 Systems

Integration with external command and control systems will extend FORGE's utility in joint and coalition environments:

System	Integration Type	Protocol	Timeline	Priority
JADC2 (Joint All-Domain C2)	Bi-directional	JREAP-C over STANAG 4609	Q2 2027	P0
AFATDS (Army Field Artillery)	Outbound	VMF (Variable Message Format)	Q4 2027	P1

System	Integration Type	Protocol	Timeline	Priority
AEGIS (Navy Combat System)	Bi-directional	Link 16 / CEC (Cooperative Engagement Capability)	Q3 2027	P0
Patriot ICC (Air Defense)	Bi-directional	JREAP-B / Link 16	Q1 2027	P1
THAAD C2BMC	Bi-directional	JREAP-C / Custom XML	Q2 2028	P2

JADC2 Integration: The Joint All-Domain Command and Control initiative represents the future of joint force coordination. FORGE will implement JADC2 data exchange through STANAG 4609 (JREAP-C over IP networks), enabling track sharing across service boundaries and support for multi-domain operations.

NATO Link 16 Networks

Coalition operations require interoperability with NATO partner networks:

MIDS Terminal Emulation: Software emulation of Multifunctional Information Distribution System (MIDS) terminals will enable participation in NATO Link 16 networks without dedicated hardware. The emulator will implement MIL-STD-6016 message formatting and network participation groups.

Network Participation Groups (NPGs): FORGE will support multiple NPGs simultaneously: PPLI (Precise Participant Location and Identification), Surveillance, Electronic Warfare, and Weapon Coordination. Each NPG operates as a separate logical network within the Link 16 architecture.

Security Classification Handling: Multi-level security (MLS) integration will enable track sharing at appropriate classification levels. FORGE will implement security labeling, downgrading workflows, and cross-domain solutions for coalition data exchange.

Coalition Data Sharing

Beyond NATO, FORGE will support broader coalition partnerships:

Five Eyes Integration: Data sharing protocols with Australia, Canada, New Zealand, and the United Kingdom through secure gateway interfaces. This includes track sharing, sensor data exchange, and coordinated engagement planning.

Bilateral Agreements: Configurable data sharing policies for specific partner nations, with fine-grained control over what data elements are shared (track position yes, launch point estimate no, for example).

Performance Improvements

GPU Acceleration

Track correlation and state estimation algorithms will leverage GPU parallelism:

CUDA-Accelerated Correlation: The GNN (Global Nearest Neighbor) correlation algorithm scales poorly with track count $O(n^3)$. GPU implementation will evaluate all association hypotheses in parallel, reducing correlation time from seconds to milliseconds for large track counts.

Kalman Filter Batches: Processing thousands of simultaneous Kalman filters on GPU enables real-time state estimation for dense threat environments. Each track update becomes a GPU kernel launch operating on batched track state vectors.

Machine Learning Inference: ONNX Runtime with CUDA execution providers will accelerate neural network inference for anomaly detection and predictive association. GPU acceleration reduces inference latency from ~100ms to <5ms per batch.

FPGA Processing

Hardware acceleration for latency-critical paths:

Protocol Encoding: JREAP and Link 16 message encoding will move to FPGA for sub-microsecond message generation. This is particularly important for time-critical engagement messages (J7.4 Engage Order, J9.0 Engagement Status).

Sensor Data Preprocessing: Raw sensor data normalization and validation will execute on FPGA before reaching the software pipeline. This includes coordinate transformations, duplicate filtering, and format validation.

Timing Requirements: FPGA processing targets <100µs end-to-end latency for critical message paths, compared to ~10ms for software-only processing.

Edge Computing

Distributed processing at sensor sites reduces data transfer latency:

Edge Track Correlators: Lightweight correlators deployed at sensor locations will perform initial track processing before forwarding to the central FORGE cluster. This reduces bandwidth requirements and enables continued operation during network degradation.

Tactical Edge Nodes: Kubernetes deployments on tactical hardware (ruggedized servers, airborne platforms) will extend FORGE capabilities to forward-deployed units with limited connectivity.

Store-and-Forward: Edge nodes will buffer track data during network outages, synchronizing with the central cluster when connectivity resumes. Conflict resolution algorithms will handle divergent track states.

Security Enhancements

Zero-Trust Architecture

FORGE will implement zero-trust security principles throughout the architecture:

Service Mesh Authentication: All service-to-service communication will require mutual TLS (mTLS) with certificate rotation. Istio or Linkerd service mesh will provide automatic certificate management and encrypted channels.

Just-in-Time Access: Operators will receive time-limited, scoped access tokens through a centralized identity provider. Access will be granted based on mission role, clearance level, and operational need.

Continuous Verification: Every request will be authenticated and authorized, regardless of network location. Internal services will validate tokens on every call, eliminating implicit trust within the cluster boundary.

Confidential Computing

Protecting sensitive data in use:

SGX Enclaves: Intel Software Guard Extensions (SGX) will protect cryptographic keys and sensitive track data during processing. Enclaves isolate computation from the host operating system, protecting against privileged attackers.

Secure Multi-Party Computation: For coalition sharing scenarios, secure MPC protocols will enable joint track correlation without revealing raw sensor data. Partners can compute on combined data without exposing their individual contributions.

Homomorphic Encryption: Research into homomorphic encryption will enable track correlation on encrypted data, eliminating the need to decrypt sensor inputs during processing.

Research Topics

Advanced Algorithms

Particle Filters for Non-Gaussian Tracking: Traditional Kalman filters assume Gaussian noise distributions. Particle filters will handle non-Gaussian measurement noise from complex sensor models and maneuvering targets with unknown intent.

Reinforcement Learning for Engagement Optimization: Agents trained through simulation will learn optimal weapon-target pairing strategies, adapting to evolving threat profiles and resource constraints. The system will continuously improve engagement decisions based on simulation outcomes.

Graph Neural Networks for Correlation: GNNs will model track correlation as a graph problem, learning structural patterns in track associations that rule-based algorithms cannot capture. This enables correlation of tracks from fundamentally different sensor modalities.

Real-Time Optimization

Constraint-Based Scheduling: Engagement coordination will use constraint programming to optimize weapon assignments across multiple threats, considering interceptor availability, engagement zones, probability of kill, and resource constraints.

Latency-Aware Processing: Critical paths will be identified and optimized for minimum latency. Event-driven processing will skip non-essential steps for time-critical tracks, falling back to full processing when time permits.

Adaptive Quality of Service: The system will dynamically adjust processing fidelity based on system load, maintaining real-time guarantees for high-priority tracks while reducing processing for lower-priority observations.

Distributed Consensus

Raft for Track Ownership: In multi-node deployments, Raft consensus will determine track ownership when multiple correlators process the same sensor data. This prevents split-brain scenarios where different nodes assign different track IDs to the same physical target.

CRDTs for Edge Synchronization: Conflict-Free Replicated Data Types will enable edge nodes to operate independently and merge track states when connectivity is restored, without requiring coordination during normal operation.

Community & Standards

MIL-STD Compliance

FORGE will maintain strict compliance with military standards:

Standard	Version	Status	Planned Enhancement
MIL-STD-3011 (JREAP)	Revision C	Implemented	JREAP-D (satellite Link 16)
MIL-STD-6016 (Link 16)	Revision D	Implemented	MATT (Multi-Link 16)
MIL-STD-2525 (Symbols)	Revision D	Partial	Full 2525C compliance
STANAG 5516 (Link 16)	Edition 3	Implemented	Edition 4 updates
STANAG 4609 (JREAP)	Edition A	Implemented	Edition B integration

Certification Path: FORGE will pursue formal certification through Joint Interoperability Test Command (JITC) for JREAP and Link 16 implementations. Certification enables deployment in operational environments requiring validated interoperability.

Open-Source Considerations

Selective open-sourcing will benefit the broader defense simulation community:

Open Core Model: Core track correlation algorithms, message formatters, and data models will be released under a permissive license (Apache 2.0 or MIT). This enables academic research, algorithm development, and community contributions.

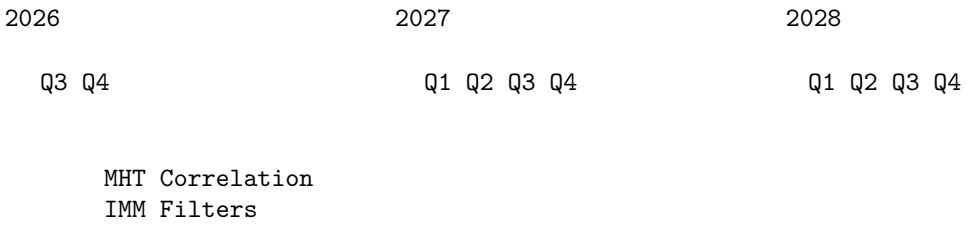
Sensitive Components: C2-specific logic, threat profiles, and engagement coordination algorithms will remain proprietary or government-owned. Export-controlled components will have restricted distribution.

Community Engagement: Regular releases, documentation, and developer engagement will build a community around the platform. Hackathons and challenge problems will encourage algorithm development and innovation.

Feature Roadmap

Feature	Description	Priority	Target
Multi-Hypothesis Tracking	MHT correlation algorithm	P0	Q3 2026
Interactive Multiple Model	Maneuver detection filters	P0	Q4 2026
GPU Correlation Acceleration	CUDA-accelerated GNN	P1	Q1 2027
Service Mesh (mTLS)	Zero-trust service authentication	P1	Q2 2027
Event Sourcing	Full event store implementation	P1	Q2 2027
JADC2 Gateway	Joint C2 integration	P0	Q2 2027
NATO Link 16 MIDS	Network participation groups	P1	Q3 2027
FPGA Message Encoding	Hardware-accelerated protocols	P2	Q4 2027
ML Anomaly Detection	Neural network track analysis	P2	Q1 2028
Secure MPC	Coalition privacy-preserving compute	P3	Q2 2028
Edge Deployment	Tactical Kubernetes distribution	P2	Q2 2028
JITC Certification	Formal interoperability validation	P0	Q4 2028

Integration Timeline



```
GPU
Acc el


        JADC2
         Integration


                    NATO
                    Link 16
                    Full

                           ML

                               JITC
                                Cert
```

Research Partnership Opportunities

Partner Type	Opportunities	Engagement Model
DoD Labs (DARPA, ARL)	Advanced algorithms, ML correlation	CRADAs, SBIR/STTR
Federally Funded R&D Centers (FFRDCs)	Architecture validation, security	Contracted research
University Research	Academic publications, algorithm development	Sponsored research, intern programs
Defense Contractors	System integration, operational deployment	Subcontracting, teaming agreements
NATO Science & Technology	Coalition interoperability	Working groups, technical panels
Open Source Community	Algorithm contributions, bug reports	GitHub, community forums

MIT Lincoln Laboratory: Joint research on multi-sensor fusion algorithms and track correlation in contested environments. FORGE provides realistic simulation environment for algorithm validation.

Johns Hopkins APL: Collaboration on engagement coordination algorithms and weapon-target pairing optimization. Integration with APL's threat modeling capabilities.

NATO STO: Participation in Technical Panels (IST, SET) for sensor fusion standards development. FORGE as reference implementation for proposed standards.

Conclusion

FORGE's future development balances immediate operational needs with long-term architectural evolution. The roadmap prioritizes capabilities that enhance current simulation and training missions while building toward a platform that can support operational deployment and coalition integration.

Key themes drive development:

- **Modularity**: Finer-grained services enable independent deployment and team ownership
- **Interoperability**: Standards compliance (JREAP, Link 16) ensures DoD and NATO integration
- **Performance**: GPU/FPGA acceleration meets latency requirements for real-time operations
- **Security**: Zero-trust and confidential computing protect sensitive data
- **Community**: Open-source engagement builds ecosystem and enables innovation

The architecture decisions made today—event sourcing, CQRS, service mesh—provide the foundation for these future capabilities. Each enhancement builds on the core platform rather than replacing it, ensuring continuity for existing users and use cases.

Document Version: 1.0
Classification: UNCLASSIFIED//FOR OFFICIAL USE ONLY
Last Updated: April 2026

Section 10: Appendix & Recovery Instructions

Purpose

This appendix provides reference materials, command references, and recovery procedures for the FORGE system design paper. It also includes instructions for resuming work if the system crashes or the session is lost.

Document Assembly Status

Section	File	Status	Last Updated
1. Executive Summary	01-executive-summary.md	Complete	2026-04-07
2. System Architecture	02-architecture.md	Complete	2026-04-07
3. Data Flow & Messaging	03-data-flow-messaging.md	Pending	-
4. Deployment & Infrastructure	04-deployment-infrastructure.md	Complete	2026-04-08
5. Integration Patterns & APIs	05-integration-apis.md	Pending	-
6. Performance & Scalability	06-performance-scalability.md	Pending	-
7. Security & Compliance	07-security-compliance.md	Pending	-
8. Testing & Validation	08-testing-validation.md	Pending	-
9. Future Development	09-future-development.md	Pending	-
10. Appendix & Recovery	10-appendix-recovery.md	This file	2026-04-07
Final Assembly	forge-system-design-complete.md	Pending	-

Recovery: How to Resume Work

If Session Is Lost

1. **Find the workspace**: `/home/wez/stsgym-work/papers/forge-system-design/`
2. **Check progress**: Read this file (10-appendix-recovery.md) and SECTION-PLAN.md
3. **Identify next section**: Look for the first Pending in the status table above

4. **Read source material**:
   - Architecture docs: `/home/wez/stsgym-work/docs/FORGE-SIMULATORS.md`
   - FORGE-C2 code: `/home/wez/stsgym-work/forge-c2/`
   - Deployment manifests: `/home/wez/stsgym-work/deploy/`
   - Memory context: `/home/wez/.openclaw/workspace/MEMORY.md`
5. **Continue writing**: Create the next pending section file

Section Content Requirements

Each section must include:

1. **Overview paragraph** (what this section covers)
2. **Detailed content** (technical specifications, code examples, diagrams)
3. **ASCII/Mermaid diagrams** where applicable
4. **Tables** for structured data
5. **References** to related sections

Final Assembly Steps

After all sections are complete:

1. **Merge sections**:

```
cat 01-executive-summary.md 02-architecture.md 03-data-flow-messaging.md \
    04-deployment-infrastructure.md 05-integration-apis.md \
    06-performance-scalability.md 07-security-compliance.md \
    08-testing-validation.md 09-future-development.md 10-appendix-recovery.md \
    > forge-system-design-complete.md
```

2. **Generate PDF** (requires pandoc):

```
pandoc forge-system-design-complete.md -o forge-system-design.pdf \
    --pdf-engine=xelatex -V geometry:margin=1in
```

3. **Publish**: Copy to web server or paper repository

File Locations

Source Material

Purpose	Location
FORGE Architecture	`/home/wez/stsgym-work/docs/FORGE-SIMULATORS.md`
FORGE-C2 Module	`/home/wez/stsgym-work/forge-c2/`
Deployment Manifests	`/home/wez/stsgym-work/deploy/`
Memory/Context	`/home/wez/.openclaw/workspace/MEMORY.md`
TODO Tracking	`/home/wez/stsgym-work/TODO.md`

Output Files

Purpose	Location
Section Files	/home/wez/stsgym-work/papers/forge-system-design/XX-name.md
Diagrams	/home/wez/stsgym-work/papers/forge-system-design/diagrams/
Final Paper	/home/wez/stsgym-work/papers/forge-system-design/forge-syst

---

Command Reference

Check System Status

```
# Kubernetes pods
kubectl get pods -A | grep -E 'forge|kafka|patroni|monitoring'

# Kafka topics
kubectl exec -it kafka-0 -n forge -- kafka-topics.sh --list --bootstrap-server localhost:909

# TimescaleDB connection
psql -h 207.244.226.151 -p 30432 -U postgres -d forge

# Prometheus targets
curl -s http://207.244.226.151:30090/api/v1/targets | jq '.data.activeTargets[].labels.job'
```

View Logs

```
# FORGE-C2 logs
kubectl logs -f deployment/forge-c2 -n forge

# Kafka consumer logs
kubectl logs -f deployment/forge-consumer -n forge

# Patroni logs
kubectl logs -f patroni-0 -n patroni
```

Backup & Recovery

```
# Create backup
velero backup create manual-backup --include-namespaces forge,patroni,monitoring

# List backups
velero backup get

# Restore from backup
velero restore create --from-backup manual-backup
```

Infrastructure Access

Service	URL	Credentials
ArgoCD	http://207.244.226.151:31572	admin / 1SXWiXA8uAy8yV81
Prometheus	http://207.244.226.151:30090	-
Grafana	http://207.244.226.151:30300	admin / admin
PostgreSQL	NodePort 30432	postgres / StsGym2024PostgreSQL
MinIO Console	http://207.244.226.151:30901	admin / MeteroBackup2024!
Kafka	NodePort 30721	-

Contact Information

Role	System	Notes
Primary Developer	Wesley Robbins	Infrastructure engineer
Repository	git@idm.wezzel.com:crab-meat-repos/stsgym-work.git	Main codebase
FORGE-C2 Repo	git@idm.wezzel.com:crab-meat-repos/forge-c2.git	Submodule

Troubleshooting

Common Issues

Issue	Diagnosis	Resolution
Kafka not receiving	Check topic config	`kubectl logs kafka-0 -n forge`
TimescaleDB slow	Check hypertable compression	`SELECT compress_chunk(i) FROM show_chunks('opir_detections') i;`
Pod CrashLoopBackOff	Check resource limits	`kubectl describe pod <name>`
Prometheus targets down	Check federation config	`kubectl get prometheus -n monitoring`

Recovery Procedures

Kafka Recovery:

```
kubectl rollout restart statefulset/kafka -n forge
kubectl rollout status statefulset/kafka -n forge
```

Patroni Recovery:

```
kubectl rollout restart statefulset/patroni -n patroni
kubectl rollout status statefulset/patroni -n patroni
```

Full Cluster Restore:

```
velero restore create --from-backup weekly-full-backup --wait
```

Document Metadata

- **Document Version**: 1.0
- **Classification**: UNCLASSIFIED//FOR OFFICIAL USE ONLY
- **Last Updated**: 2026-04-07 03:30 UTC
- **Author**: FORGE Development Team
- **Review Status**: Draft

Changelog

Date	Version	Changes
2026-04-07	1.0	Initial appendix creation with recovery procedures

---

This document is part of the FORGE System Design Paper. For questions, contact the development team.