Webinar:
Digital Twins: The Next Generation in Stream-Processing and Real-Time Analytics

Dr. William Bain, Founder and CEO
(wbain@scaleoutsoftware.com)
December 11, 2018
Agenda

• About ScaleOut Software
• Our core technologies: software for in-memory data grids and computing
• Challenges for stream-processing
• A solution: the digital twin model
• Running digital twins on an IMDG
  • Advantages
  • Comparison to traditional approaches
• IoT example with C# code using ScaleOut Digital Twin Builder™
• IoT example with Java code incorporating data-parallel feedback
About ScaleOut Software

- Develops and markets **In-Memory Data Grids**, software middleware for:
  - **Scaling application performance** and
  - **Providing operational intelligence** using
  - **In-memory data storage and computing**
- Dr. William Bain, Founder & CEO
  - Career focused on parallel computing
  - Bell Labs, Intel, Microsoft
- Eleven years in the market:
  - 450+ customers, 10,000+ servers
- Sample customers:
ScaleOut Software’s Products

• **ScaleOut StateServer® & ScaleOut GeoServer®**
  - In-Memory Data Grid (IMDG) for Windows and Linux
  - Application scaling with strong consistency & high availability
  - APIs in Java, C#, C/C++
  - Deployable on-premises and in public clouds (Azure, AWS)
  - Global data replication and remote data access
  - Released in 2005; now in 5th major version

• **ScaleOut StreamServer™ & ScaleOut Digital Twin Builder™**
  - Stateful stream-processing with digital twins
  - Simplified development for digital twins in Java, C#
  - Support for ReactiveX APIs, Kafka, and Azure IoT
  - Integrated IMDG and in-memory compute engine
  - Real-time, data-parallel analytics
Core Technology: IMDG + IMC

• **In-Memory Data Grid (IMDG):** cluster-hosted software which provides fast, distributed in-memory storage for live data:
  • Uses object-oriented, key/value storage model
  • Location-transparent access to data by multiple clients
  • Create/read/update/delete APIs for Java/C#/C++
  • Parallel query by object properties

• **In-Memory Computing:** integrated software-based compute engine for streaming & data-parallel ops
  • Runs o-o methods on live data with low latency
  • Avoids network bottlenecks by computing in the IMDG.

• **Both:** Transparent scalability and high availability:
  • Automatic load-balancing across commodity servers
  • Automatic data replication, failure detection, and recovery

© ScaleOut Software, Inc.
How an IMDG Can Integrate Computation

- Each grid host runs a worker process which executes application-defined methods on stored objects.
  - The set of worker processes is called an **invocation grid (IG)**.
  - IG usually runs language-specific runtimes (JVM, .NET).
  - IMDG can ship user code to the IG workers.
- Key advantages for IGs:
  - Follows object-oriented model.
  - Avoids network bottlenecks by moving computing to the data.
  - Leverages IMDG’s cores & servers.
IMDG Runs Handlers for Stream-Processing

Event handlers run independently for each incoming event:

- IMDG directs event to a specific object (e.g., using ReactiveX) for low latency.
- IMDG executes multiple event handlers in parallel for high throughput.
IMDG Also Runs Data-Parallel Computations

Method execution implements a parallel operation on a stored object collection:

- Client runs a single method on all objects in a collection.
- Execution runs in parallel across the grid.
- Results are merged and returned to the client.
- Runs with lower latency than batch jobs.
A Basic Data-Parallel Execution Model

A fundamental model from parallel supercomputing:

- Run one method (“eval”) in parallel across many data objects.
- Optionally **merge** the results.
  - Binary combining is a special case, but...
  - It runs in logN time to enable scalable speedup

objects stored in the IMDG:
Example: MapReduce Computation

- Implements “group-by” computations on live data.
- Example: “Determine average RPM for all wind turbines by region (NE, NW, SE, SW).”
- Runs in two data-parallel phases (map, reduce):
  - Map phase extracts, repartitions, and optionally combines source data.
  - Reduce phase analyzes each data partition in parallel.
  - Returns results for each partition.
Goals for Stream-Processing

• **Goals:**
  • Process incoming data streams from many (1000s) of sources.
  • Analyze events for patterns of interest.
  • Provide timely (real-time) feedback and alerts.
  • Provide data-parallel analytics for aggregate statistics and feedback.

• **Many applications:**
  • Internet of Things (IoT)
  • Medical monitoring
  • Logistics
  • Financial trading systems
  • Ecommerce recommendations

• **Challenge:** How can we track the dynamic state of data sources to enhance real-time analysis?
Example: Ecommerce Recommendations

1000s of online shoppers:
• Each shopper generates a clickstream of products searched.

• Stream-processing system must:
  • Correlate clicks for each shopper.
  • Maintain a history of clicks during a shopping session.
  • Analyze clicks to create new recommendations within 100 msec.

• To be effective, analysis should:
  • Take into account the shopper’s preferences and demographics.
  • Use aggregate feedback on collaborative shopping behavior.
Real-Time Recommendations

• Requires *stateful* stream-processing to analyze each click and respond in <100ms:
  • Can accept input with each event on shopper’s preferences and track these preferences.
  • Can analyze aggregate behavior and provide feedback on best-selling products.
Real-Time, Aggregate Metrics

- Dynamically aggregates statistics for all shoppers:
  - Track real-time shopping behavior.
  - Chart key purchasing trends.
  - Enable merchandizer to create promotions dynamically.

- Combined statistics can be shared with all shoppers:
  - Allows shoppers to obtain collaborative feedback.
  - Examples include most viewed and best selling products.
Challenges for Stream-Processing

• Basic stream-processing architecture is a pipeline (or acyclic graph):

  ![Stream Processing Pipeline Diagram]

  - **Data Sources**
  - **Events**

• **Challenges** unmet by traditional architectures:
  - How efficiently correlate events from each data source?
  - How combine events with relevant state information to create the necessary context for analysis?
  - How embed application-specific analysis algorithms in the pipeline?
  - How generate feedback/alerts with low latency?
  - How perform data-parallel analytics to determine aggregate trends?
Adding Context to Stream-Processing

- Stateful stream-processing platforms add “unmanaged” data storage to the pipeline:
  - Pipeline stages perform transformations in a sequence of stages from data sources to sinks.
  - Data storage (distributed cache, database) is accessed from the pipeline by application code in an unspecified manner.
  - Examples: Apama (CEP), Apache Flink, Storm

- Problems:
  - Data stores for managing state information are not integrated into the pipeline.
  - This adds complexity and creates a network bottleneck.
  - Does not address need for data-parallel analytics.

How can we efficiently combine stream-processing with state (context) to enable real-time analytics, simplify design, and maximize performance?
A Solution: the “Digital Twin” Model

• Term coined by Dr. Michael Grieves (U. Michigan) in 2002 for use in product life cycle management

• Popularized in Gartner’s “Top 10 Strategic Technology Trends for 2017: Digital Twins” for use with IoT

• **Definition:** a digital representation of a physical entity; an encapsulated software object that comprises (per Gartner):
  - A model (e.g., composition, structure, metadata for an IoT sensor)
  - Data (e.g., sensor data, entity description)
  - Unique identity (e.g., sensor identifier)
  - Monitoring (e.g., alerts)

• **Significance:** focuses on modeling data sources
  - A basis for correlating and analyzing streaming data
  - A context for deep introspection and interaction
Many Uses of the Term “Digital Twin”

Although created by Michael Grieves for product life cycle management (PLM)...

- The term “digital twin” has several interpretations, for example:
  - **Digital twin** as used in PLM and product-line engineering (from Marc Lind, SVP Aras Corp.)
    - A virtual version of a physical entity
    - Adds context to interpret the time-series data streaming back from the field
  - **Azure digital twin**: spatial graph of spaces, devices, and people for modeling relationships in context
  - Azure IoT **device twin**: JSON document that stores per-device state information (metadata, conditions)
  - **AWS device shadow**: cloud-based repository for per-device state information with pub/sub messaging

- **ScaleOut’s use of digital twin**:  
  - Object-oriented model of a data source (or higher-level entity) for use in real-time streaming analytics  
  - **Benefit**: enables real-time streaming analytics which is:
    - Fast and scalable
    - Easy to use
Examples of Digital Twins in IoT

**Live System – Physical Objects**

- (Autonomous) Vehicles
  - Vehicle subsystems for safety monitoring & predictive maintenance
- Manufacturing floors and equipment
  - Networks of machine tooling for real-time interactive view and predictive maintenance
- Wind turbines and wind farms
  - Collections of wind turbine components for remote operations and predictive maintenance

**Digital Twins**

- Telemetry streams
- Immediate feedback

© ScaleOut Software, Inc.
• A digital twin **model** represents a type of data source (e.g., a wind turbine).

• Each digital twin **instance** represents a specific physical data source (e.g., wind turbine 73).

• Digital twin typically comprises:
  • An event collection
  • State information about the data source
  • Logic for managing events & commands, updating & analyzing state, generating alerts

• **Object oriented model:**
  • Holds source’s dynamic state information.
  • Encapsulates domain-specific logic (e.g., ML, rules engine, etc.).
  • Runs code where the data lives (avoids data motion) for fast response times.
  • Enables data-parallel analysis.
Using an IMDG to Host Digital Twins

The IMDG:

• Can host thousands of digital twins as objects.
• Can post incoming events to their respective digital twin objects.
• Can run the twin’s event handler method with low latency:
  • Event handler uses and updates in-memory state.
  • Event handler can manage an event collection and use time windows for its analysis.
  • Event handler can use/update off-line state.
  • Event handler optionally generates alerts and feedback to its digital twin.
• Also can run data-parallel methods to analyze all digital twins in real-time.
  • Collects and reports periodic aggregate statistics.
  • Results can be used for both alerting and feedback.
Why Use an IMDG to Host Digital Twins?

• **Object-oriented data storage:**
  - Offers a natural model for hosting digital twins.
  - Cleanly separates domain logic from data-parallel orchestration.
  - Provides rich context for correlating and processing streaming data.
  - Allows easy addition of specialized analysis algorithms (rules, ML, etc.)
  - Integrates streaming and data-parallel processing.

• **High performance:**
  - Avoids data motion and associated network bottlenecks.
  - Fast and scales to handle large workloads.

• **Integrated high availability:**
  - Uses data replication designed for live systems.
  - Can ensure that computation is high av.
Comparison to Traditional Architecture

An IMDG:
- Avoids the need to correlate events from each data source in the stream processing pipeline:
  - Reduces application complexity.
  - Eliminates network bottlenecks.
- Refactors processing steps to perform them in one location:
  - Allows application encapsulation.
  - Avoids data motion between pipeline stages.
- Provides a basis for transparent scaling:
  - Leverages the grid’s load-balancing of digital twin objects across the IMDG.
- Enables data-parallel analytics.
Important to Avoid Network Bottlenecks

- Hosting digital twins in an IMDG avoids network bottlenecks associated with accessing a database or networked cache in a stream-processing pipeline.
  - External data storage requires network access to obtain an event’s context.
  - Network bottleneck prevents scalable throughput.
Moves Streaming Analytics into Real Time

- Lambda architecture separates stream-processing ("speed layer") from data-parallel analytics ("batch layer").
- Performance limitations keep streaming analytics in the batch layer.
- This prevents real-time responses with deep introspection.
- ScaleOut’s digital twin model running on ScaleOut StreamServer’s IMDG+IMC enables:
  - **Deep introspection** in the speed layer
  - **Real-time feedback** from event analytics
  - Data-parallel analytics to detect aggregate trends in real time
Many Applications for Digital Twins

A digital twin correlates incoming events with context using domain-specific algorithms to generate alerts:

<table>
<thead>
<tr>
<th>Application</th>
<th>Context</th>
<th>Events</th>
<th>Logic</th>
<th>Alerts</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoT devices</td>
<td>Device status &amp; history</td>
<td>Device telemetry</td>
<td>Analyze to predict maintenance.</td>
<td>Maintenance requests</td>
</tr>
<tr>
<td>Medical monitoring</td>
<td>Patient history &amp; medications</td>
<td>Heart-rate, blood-pressure, etc.</td>
<td>Evaluate measurements over time windows with rules engine.</td>
<td>Alerts to patient &amp; physician</td>
</tr>
<tr>
<td>Cable TV</td>
<td>Viewer preferences &amp; history, set-top box status</td>
<td>Channel change events, telemetry</td>
<td>Cleanse &amp; map channel events for reco. engine; predict box failure.</td>
<td>Viewer recommendations, repair alerts</td>
</tr>
<tr>
<td>Ecommerce</td>
<td>Shopper preferences &amp; buying history</td>
<td>Clickstream events from web site</td>
<td>Use ML to make product recommendations.</td>
<td>Product list for web site</td>
</tr>
<tr>
<td>Fraud detection</td>
<td>Customer status &amp; history</td>
<td>Transactions</td>
<td>Analyze patterns to identify probable fraud.</td>
<td>Alerts to customer &amp; bank</td>
</tr>
</tbody>
</table>
Example: Tracking a Fleet of Vehicles

- **Goal**: Track telemetry from a fleet of cars or trucks.
  - Events indicate speed, position, and other parameters.
  - Digital twin object stores information about vehicle, driver, and destination.
  - Event handler alerts on exceptional conditions (speeding, lost vehicle).

- Periodic data-parallel analytics determines aggregate fleet performance:
  - Computes overall fuel efficiency, driver performance, vehicle availability, etc.
  - Can provide feedback to drivers to optimize operations.
OOP Techniques Simplify Digital Twins

• Digital twin objects can use **inheritance** to create specialized behaviors:

  - Baseline Controller
    - Event Collection
    - `HandleEvent()`, `Start()`, `Stop()`
    - Status: "started"

  - **Sub-Class**
    - `ReadTemperature()`, `ReadFlow()`
    - Temperature: "180°F"
    - Flow rate: "5.2 GPM"

  - **Base Class**

  - Pump Room Controller
    - `Start()`, `Stop()`, `GetStatus()`

• Instances of objects can be organized in a **hierarchy**:

  - **Circuit Breaker**
    - `ReadCurrent()`, `ReadTemperature()`
    - Current: "3.5 A"
    - Temperature: "50°C"

  - **Hot Water Valve**
    - `ReadTemperature()`, `ReadFlow()`
    - Temperature: "160°F"
    - Flow rate: "5.2 GPM"

  - **Base Controller**

© ScaleOut Software, Inc.
Using Digital Twins in a Hierarchy

Tracks complex systems as hierarchy of digital twin objects:

- Leaf nodes receive telemetry from physical endpoints.
- Higher level nodes represent subsystems:
  - Receive telemetry from lower-level nodes.
  - Supply telemetry to higher-level nodes as alerts.
  - Allow successive refinement of real-time telemetry into higher-level abstractions.

Example: Hierarchy of Digital Twins for a Wind Turbine
Digital Twins Simplify Migration to Edge

• Migration of stream-processing intelligence to the edge is an ongoing trend driven by continuous advances in technology.

• Constructing software components as o-o digital twins simplifies migration:
  • Makes software decomposition independent of execution location.
  • Avoids rewriting code for execution at the edge; can leverage containers.
Digital Twins Enable Data Parallel Analysis

• Uses IMDG’s in-memory compute engine to create aggregate statistics \textit{in real time}.

• Results can be reported to analysts and updated every few seconds.

• Results can be used as feedback to event analysis in digital twin objects and/or reported to users.
ScaleOut Digital Twin Builder™ Toolkit

- API libraries for building digital twin models in Java and C#
- Deployment libraries for hosting in ScaleOut StreamServer

- Connectors to Kafka, Azure IoT, and REST
Sample Application (C#)

• Goal: Illustrate use of digital twin to analyze temperature telemetry from a wind turbine.

• Digital twin tracks:
  • *Parameters*: model, pre-maintenance period based on model, max. allowed temperature, max. allowed over-temp duration (normal and pre-maintenance)
  • *Dynamic state*: time to next maintenance, over-temp condition and its duration

• Message processing:
  • Determines onset of and recovery from over-temp condition
  • Alerts at maximum allowed duration
  • Logs incidents for time-windowing analysis
Sample State Object (C#)

```csharp
[JsonObject]
public class WindTurbine : DigitalTwinBase
{
    // physical characteristics:
    public const string DigitalTwinModelType = "windturbine";
    public WindTurbineModel TurbineModel { get; set; } = WindTurbineModel.Model7331;
    public DateTime NextMaintDate { get; set; } = new DateTime().AddMonths(36);
    public const int MaxAllowedTemp = 100; // in Celsius
    public TimeSpan MaxTimeOverTempAllowed = TimeSpan.FromMinutes(10);
    public TimeSpan MaxTimeOverTempAllowedPreMaint = TimeSpan.FromMinutes(2);

    // dynamic state variables:
    public bool TrackingOverTemp { get; set; }
    public DateTime OverTempStartTime { get; set; }
    public int NumberMsgsWithOverTemp { get; set; }

    // list of incidents and alerts:
    public List<Incident> IncidentList { get; } = new List<Incident>();
}
```
public override ProcessingResult ProcessMessages(ProcessingContext context, WindTurbine dt, IEnumerable<DeviceTelemetry> newMessages)
{
    var result = ProcessingResult.NoUpdate;

    // determine if we are in the pre-maintenance period for this wind turbine model:
    var preMaintTimePeriod = _preMaintPeriod[dt.TurbineModel];
    bool isInPreMaintPeriod = ((dt.NextMaintDate - DateTime.UtcNow) < preMaintTimePeriod) ? true : false;

    // process incoming messages to look for over-temp condition:
    foreach (var msg in newMessages) {
        // if message reports a high temp indication, track it:
        if (msg.Temp > WindTurbine.MaxAllowedTemp)
            <track over-temp condition>
        else if (dt.TrackingOverTemp)
            <resolve over-temp condition>
    }
    return result;
}
// track over-temp condition:
{dt.NumberMsgsWithOverTemp++;

if (!dt.TrackingOverTemp) {
    dt.TrackingOverTemp = true; dt.OverTempStartTime = DateTime.UtcNow;
    <add a notification to the incident list> }

TimeSpan duration = DateTime.UtcNow - dt.OverTempStartTime;

// if we have exceeded the max allowed duration for an over-temp, send an alert:
if (duration > dt.MaxTimeOverTempAllowed ||
    (isInPreMaintPeriod && duration > dt.MaxTimeOverTempAllowedPreMaint)) {
    var alert = new Alert(); <fill out the alert message>
    context.SendToDataSource(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(alert)));
    <add a notification to the incident list> }

// resolve the condition and reset our state:
{dt.TrackingOverTemp = false; dt.NumberMsgsWithOverTemp = 0;
  <add a notification to the incident list> }
Deploying the Model

• Deploy the WindTurbine model to ScaleOut StreamServer:

```csharp
ExecutionEnvironmentBuilder builder = new ExecutionEnvironmentBuilder()
    .AddDependency(@"WindTurbine.dll")
    .AddDigitalTwin<
        WindTurbine, WindTurbineMessageProcessor,
        DeviceTelemetry>(
        WindTurbine.DigitalTwinModelType);
```

• Connect to a data source (Azure IoT Hub):

```csharp
EventListenerManager.StartAzureIoTHubConnector(
    eventHubName : _eventHubName,
    eventHubConnectionString: _eventHubConnectionString,
    eventHubEventsEndpoint  : _eventHubEventsEndpoint,
    storageConnectionString : _storageConnectionString,
    consumerGroupName : "");
```
Example: Heart-Rate Watch Monitoring

**Goal**: Track heart-rate for a large population of runners.

- Heart-rate events flow from smart watches to their respective digital twin objects for analysis.
- The analysis uses wearer’s history, activity, and aggregate statistics to determine feedback and alerts.
Digital Twin Object (Java)

- Holds event collection and user’s context (age, medical history, current status, etc.):
Events & Alerts

• Event holds periodic telemetry sent from watch to IMDG:

```java
class HeartRateEvent {
    private int _userId;
    private int _heartRate;
    private long _timestamp;
    private WorkoutType _workoutType;
    private WorkoutProgress _workoutProgress;
    private Event _event;
    ...
}
```

• Alert holds data to be sent back to wearer and/or to medical personnel:

```java
class HeartRateAlert {
    private int _userId;
    private String _alertType;
    private String _params;
    ...
}
```
Event Analysis

• Handles an event for an active user doing a running workout:

```java
private static void processMessage(HeartRateEvent hre, User u) {
    long start = twoWeeksAgo();
    long sessionTimeout = threeHours();
    SessionWindowCollection<HeartRate> swc = new SessionWindowCollection<>
                                            (u.getRunningHeartRateTelemetry(),
                                             heartRate -> heartRate.getTimestamp(), start, sessionTimeout);
    swc.add(new HeartRate(hre.getHeartRate(), hre.getTimestamp()));

    int total = 0; int windowCount = 0;
    for(TimeWindow<HeartRate> window : swc) {
        int avg = 0;
        for(HeartRate hr : window) {avg += hr.getHeartRate();}
        total += (avg/window.size());
        windowCount++;
    }
    u.setAverageHr(total/windowCount);
    u.analyzeAndCheckForAlert(hre);
}
```

Create time windows
Add event
Analyze event history
Analyze user’s context
Enable detailed heart-rate monitoring for a high intensity exercise program:

• Example of data to be tracked:
  • **Exercise specifics**: type of exercise, exercise-specific parameters (distance, strides, altitude change, etc.)
  • **Participant background/history**: age, height, weight history, heart-related medical conditions and medications, injuries, previous medical events
  • **Exercise tracking**: session history, average # sessions per week, average and peak heart rates, frequency of exercise types
  • **Aggregate statistics**: average/max/min exercise tracking statistics for all participants

• Example of logic to be performed:
  • **Notify participant** if session history across time windows indicates need to change mix.
  • **Notify participant** if heart rate trends deviate significantly from aggregate statistics.
  • **Alert participant/medical personnel** if heart rate analysis across time windows indicates an imminent threat to health.
  • **Report** aggregate statistics to analysts and/or users.
Data Parallel Analysis Across all Digital Twins

• Uses IMDG’s in-memory compute engine to create aggregate statistics in real time.

• Results can be reported to analysts and updated every few seconds.

• Results can be used as feedback to event analysis in digital twin objects and/or reported to users.
Computing Aggregate Data

- Performs a data-parallel computation using the IMDG’s Eval and Merge methods:

```java
public class AggregateStatsInvokable implements Invokable<User, Integer, AggregateStats> {
    @Override
    public AggregateStats eval(User u, Integer numUsers) {
        AggregateStats userStats = new AggregateStats(numUsers);
        userStats.merge(u);
        return userStats;
    }

    @Override
    public AggregateStats merge(AggregateStats mergedStats, AggregateStats u) {
        mergedStats.merge(u);
        return mergedStats;
    }
}
```

© ScaleOut Software, Inc.
Computing Aggregate Data (2)

• Computes running average of heart-rate by categories:

```java
public void merge(AggregateStats user) {
    numEvents += user.getNumEvents();
    totalHeartRate18to34 += user.getTotalHeartRate18to34();
    totalHeartRate35to50 += user.getTotalHeartRate35to50();
    totalHeartRateOver50 += user.getTotalHeartRateOver50();
    count18to34 += user.getCount18to34();
    count35to50 += user.getCount35to50();
    countOver50 += user.getCountOver50();

    totalHeartRateBmiUnderWeight += user.getTotalHeartRateBmiUnderWeight();
    totalHeartRateBmiNormalWeight += user.getTotalHeartRateBmiNormalWeight();
    totalHeartRateBmiOverweight += user.getTotalHeartRateBmiOverweight();
    countUnderweight += user.getCountUnderweight();
    countNormalWeight += user.getCountNormalWeight();
    countOverWeight += user.getCountOverWeight();
}
```
Running the Data-Parallel Computation

- Uses a single method to run a data-parallel computation and return results.
- Publishes merged results to an IMDG object for access by user objects and/or analysts.

```java
public void run() {
    NamedCache usersCache = CacheFactory.getCache("userCache");
    NamedCache statsCache = CacheFactory.getCache("statsCache");
    AggregateStats stats;

    InvokeResult<AggregateStats> result =
        usersCache.invoke(AggregateStatsInvokable.class, null, _numUsers,
                          TimeSpan.fromMilliseconds(10000));

    stats = result.getResult();
    statsCache.put("globalStats", stats);
}
```

Invoke data-parallel op

Store result in IMDG
Digital Twins: The Next Generation in Stream-Processing and Real-Time Analytics

• **Challenge**: Current techniques for stateful stream-processing:
  • Lack a coherent software architecture for managing context.
  • Can suffer from performance issues due to network bottlenecks.

• **The digital twin model**:
  • Offers a flexible, powerful, scalable architecture for stateful stream-processing:
    • Associates events with context about their physical sources for deeper introspection.
    • Enables flexible, object-oriented encapsulation of analysis algorithms.
  • Provides a basis for aggregate analysis and feedback.

• **Stateful stream-processing using digital twin models in ScaleOut StreamServer**:
  • Automatically correlates incoming events and processes them in parallel.
  • Enables integrated stream-processing and real-time analytics.
Thank you!

For more information:
- ScaleOut Software: [www.scaleoutsoftware.com](http://www.scaleoutsoftware.com)
- Java Digital Twin Builder libraries: [github.com(scaleoutsoftware/JavaDigitalTwinCore](https://github.com/scaleoutsoftware/JavaDigitalTwinCore)
- REST Digital Twin message service: [hub.docker.com/r/scaleout/dtbuilder_webmessenger/](https://hub.docker.com/r/scaleout/dtbuilder_webmessenger/)

[www.scaleoutsoftware.com](http://www.scaleoutsoftware.com)