THE APAMA PLATFORM

TABLE OF CONTENTS

"As enterprises strive to cut costs and improve their responsiveness to customers, suppliers and the world at large, the concept of event-driven design is becoming more widely used. Event-driven business processes and event-driven application systems help enterprises react quickly and precisely to rapidly changing conditions."

— **Roy Schulte** | Gartner, Inc.

Introduction to event processing

In today's digital business age, enterprises are rapidly moving applications to an event-driven model to achieve the speed, agility and responsiveness they need to remain competitive. Digital businesses—and their supporting technologies—must become more adept at responding to changing circumstances and they must do so quickly.

At the same time, organizations are dealing with ever-increasing volumes of data from an ever-increasing number of data-generating sources. This is so they can respond to opportunities and threats by capturing the correct context and taking the most appropriate action. Orthogonally to the volume of data is the speed at which the data needs to be analyzed and the viable time window to act.

Not only must they analyze "big data in motion" in "real time," they need technology that delivers "performance at scale" by supporting a wide number of concurrent applications while consuming large volumes of changing data. And this must be done in real time so the business can respond to events immediately.

Another important dimension to performance is in the speed of evolution. Deployed applications are commonly outdated in some way immediately after deployment; new threats and opportunities arrive second by second, and the ability to react to this simply and quickly provides an enterprise with a massive advantage.

Event-driven architectures, such as Apama from Software AG, enable digital businesses to detect changes in circumstances, discern the impact of those changes and respond quickly. These architectures enable businesses to exploit opportunities and forestall threats that are often nestled within events. These key attributes are summarized as "monitor, analyze and act"—and they are precisely what you can do with Apama, the market-leading platform for event-driven systems. The Apama system was specifically developed to address the requirements of event-based systems, providing:

- A sophisticated event processing infrastructure that easily scales to deliver rapid analysis on large volumes of streaming events
- A unique architecture that supports simultaneous operation of thousands of individual event processing monitors
- A richly featured event processing language that can express both data and temporal attributes for sophisticated time-sensitive logic
- Graphical tools that enable business analysts—who possess the knowledge of the business environment—to be fully engaged in the event processing development process
- A development environment that enables IT professionals to quickly develop and deploy (and subsequently evolve) event-driven applications
- An end-user "dashboard" environment that captures and conveys what is happening to users in a visually intuitive manner
- Capture and replay of event streams for root-cause analysis of conditions, pre-flight testing of new application scenarios, and "what-if" simulation
- A unified framework for event pattern matching with analytics that span different event types and streams
- An integration framework that complements existing application infrastructures, including service-oriented architectures

Note: Scale-out is limited in community edition (see the community edition FAQ for limits)

Overview of the Apama platform

The capabilities of the Apama platform have evolved over time to incorporate additional functionality that combine to create a comprehensive event processing platform. Rather than a pure event "engine," the Apama architecture also incorporates:

- An integrated development environment that embraces business users and IT
- Real-time graphical dashboards for monitoring application execution
- An application for event replay
- An integration framework for integrating Apama into event messaging environments, databases and application environments

Components of Apama Event Processing Architecture

Apama covers three tiers:

1. External systems that connect to Apama, whether providing real-time feeds, connecting to static data sources, receiving data or visualizing it
2. A runtime engine called the Apama Correlator, which consumes all data sources, executes the desired applications, looks for event patterns and delivers insights and actions in real time
3. Unified productivity tooling so both business and IT developers can quickly and easily build, evolve and test Apama applications

The heart of Apama: The Apama Correlator

The core execution engine for the Apama platform is the Apama Correlator. It acts as a "container" for Apama applications that can be injected and removed dynamically at run-time without disrupting other running applications. When injected into the Correlator they are divided into:

- Event patterns that define the events and patterns of events that an application is interested in. These are passed to internal units within the Correlator for high-performance execution. Uniquely event patterns can be modified programmatically and need not be fixed-stream networks
- Application logic that is executed when event patterns are matched, such like a traditional "call-back" or "listener;" the application code can perform any operation, including generating output events

There are a number of unique and patented aspects to Apama, the two most important can be described as the "inversion of the problem" and the Apama HyperTree.

Inversion of the problem

Traditional approaches to event processing are derived from the fundamental architecture of receiving the data, storing the data and querying the data. This requires a number of additional stages, such as updating an index to improve the speed of the query. In a canonical form, we could consider approach as below:

The limitations of this approach are numerous. In summary:

- There is significant performance overhead of maintaining an accurate index for all data values at even modest volumes
- Loading the data, updating the index and relying on a trigger all incur additional latency before your query and application logic are addressed
- The performance bottleneck of the index requires limited use of the index, commonly using only a single field within the event and often updated in a batch fashion to limit the compotation overhead, which in turn, dramatically extends the reaction latency

To overcome these problems, Apama inverts the architecture. At the highest level of abstraction, there are incoming data, a query to execute over that data and an index to be used to speed up the query. Rather than apply the index to the high volumes of changing data like in traditional systems, Apama instead applies the index to the query. That is, the query is deconstructed and its elements used to update an index. Now, when data is received, it is immediately compared to the index to execute the query; the data need not be stored, no indexes are updated, and there is no need for a secondary trigger. This creates a simpler and more efficient architecture that is suited to event processing of big data in motion.

The Apama HyperTree

This approach requires a different type of indexing technology: the Apama HyperTree. The Apama HyperTree contains both data structures and algorithms that are designed for high-performance multi-dimensional event filtering. It autonomously self-optimizes the event patterns it is asked to detect while delivering massive scalability with zero effective latency. Unlike traditional systems, the HyperTree enables many fields within an event to be indexed, resulting in high-speed matches of fields, including those that are rarely used. It's massively scalable and has a logarithmic performance curve, where performance degrades minimally even with massive numbers of concurrent queries. See the benchmarks section on page 16 for more details.

With Apama's real-time event processing model, the detection of events is not dependent on index recalculation as would be required to perform a similar operation with a database. Event patterns, in the context of temporal or spatial constraints (Apama supports native location references that can be used to define spatial relationships), can be detected instantly without additional processing.

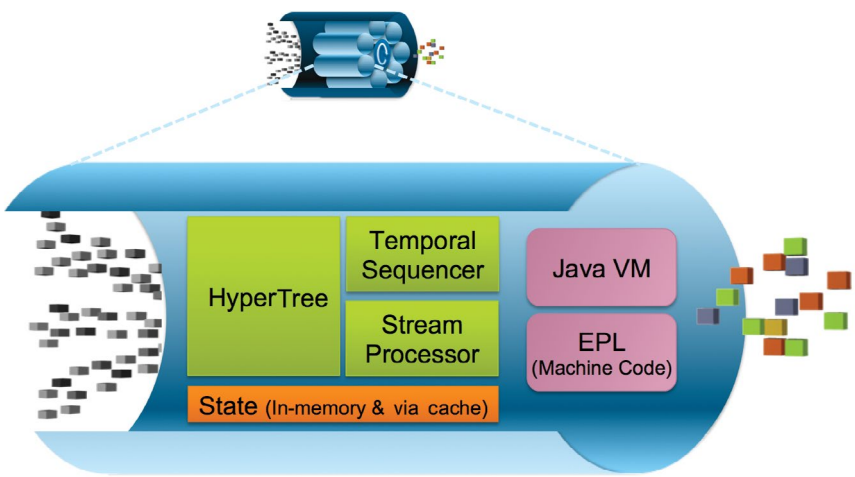
Inside the Apama Correlator

Apama applications are defined in any one of three ways or any combination:

- **Apama Event Programming Language (EPL)**: a concise and easy-to-use description language that has been designed to around the requirements of defining and acting upon event patterns. Its richly expressive semantics provide a depth of event processing functionality that is unmatched in the market
- **Java®**: Standard Java
- **Apama queries**: graphically constructed applications built using the Software AG Designer environment

Correlators do not execute application in the sequential manner of traditional imperative programs. Apama applications are best viewed as modular compositions that segment the event monitoring, analysis and action stages into logically related but independent segments. This segmentation is particularly valuable in addressing the requirements inherent in event processing applications where there may be thousands of instances operating simultaneously.

To do this, Apama includes the notion of multiple parallel executing "contexts." Each context can be considered as a lightweight execution container. These can each process any number of Apama applications and through the use of the Correlator's internal and highly optimized thread scheduler will map many numbers of the contexts onto a smaller number of hardware or O/S threads. This allows Apama to scale simply, elastically and efficiently over all the available CPUs and cores within a single host.



Single Context Shown in More Detail

The key elements that execute an Apama application are:

- The **Apama HyperTree**, is where events are initially presented to the Correlator and are immediately inspected. At this point the HyperTree knows whether there is any value in processing this event any further (i.e., whether any application is interested in it) and then what to do next
- The **temporal sequencer** builds upon the event matching capabilities of the HyperTree to provide multiple temporal correlations. For example, if an application seeks event A, to be followed by event B within 500 milliseconds, the Correlator only concerns itself with looking for event A and does not waste effort looking for event B yet. Only when event A is detected does the Correlator begins to look for B, and only for a maximum of 500 milliseconds as specified in the monitor. If no event B is detected in that time, the application terminates
- The **stream processor** stores and organizes windows of events and orchestrates the execution of real-time analytics over these event windows. Streams provides a powerful, efficient and easy to use set of capabilities for defining real-time analytics and for looking for relational style patterns across multiple streams of data

Once an event pattern (simple or complex) is identified the application that requested the listener is notified and this is achieved by invoking the application whether it was written in Apama's EPL or Java. Any action/application logic written in Java is executed within an in-built Java Virtual Machine (JVM®) within the Correlator. Any action/application logic written in EPL is executed as native machine code*. This gives applications written in the easy-to-use and fit-for-purpose event programming language a significant performance advantage; our benchmarks show that applications such as an option pricing benchmark execute faster in EPL than in either Java or C++.

Note: there are limits to scalability in the community edition. Full scaling is achievable using the commercial edition.

Writing applications in Apama

Apama incorporates the use of a unique EPL that is designed specifically for event-driven applications. EPL offers a native capability to use event attributes, their time of occurrence, and any relationships among the events as fundamental programming constructs. EPL offers clear advantages over the structured, relational algebra of SQL, which focuses upon transactional data requirements. And, if implemented as has been done within Apama, EPL offers equally clear advantages over lower-level programming environments that have are not focused on the unique aspects of event processing.

Apama's EPL

EPL is Apama's event-based language, designed to be compact, powerful and easy to use. This section describes at a high level, with illustrative examples, some of the capabilities available through EPL, though it is intended to be only a brief overview. More detailed discussion, with supporting examples, is available via Apama's documentation.

In addition to EPL, it is also possible to extend the built-in capabilities through the use of plug-ins (such as C++ libraries) that are callable from EPL or its Java equivalent. A plug-in is an externally linked software module that registers with the Correlator via its extension API. Plug-ins can be particularly valuable in the use of existing program libraries of real-time functions. Such functions can be made available as objects that are invoked via EPL actions.

* On Linux® AMD/Intel® 64-bit platforms as of April 2013

Defining event types

Before EPL can look for patterns in streaming events, it must understand the available events. An "event type definition" informs the Correlator about the composition of the underlying event. This definition allows the Correlator to understand the event messages it receives and create the best indexing structures. An example of an event type for a stock exchange tick feed is shown below.

```
event StockTick {
        string symbol;
        float price;
        float volume;
}
```

Each attribute of the event has an event type parameter that instructs the Correlator as to how to handle the attribute and what operations can be executed on it. As noted earlier, a Correlator can handle multiple data types that include string, integer, float, Boolean, decimal, sequence, dictionary, location and event, with events consisting of multiple attributes. In this example, event processing operations specific to a stock symbol, the price, the volume—either independently or in some combination of the three—can be defined.

Apama's support for different event types represents a key advantage when compared to systems that achieve performance by optimizing their architectures around a single event class. For example, another example of an event class is shown below. This defines a news event, which may be extracted from a real-time news feed, providing headlines about particular stock symbols.

```
event NewsItem {
        string subject;
        string headline;
}
```

Support for different event types—and their accompanying diverse formats—allows Apama to deliver an expansive set of solutions. For example, cross-asset trading in finance requires the support of different types of financial instruments in a single applciation. Military applications that focus on information "fusion" capture data streams from different sources and must correlate across those different streams to identify patterns. Likewise, in cold chain automation solutions (the shipment of food through a supply chain) may require the system to respond to temperature, moisture and other pertinent data. As event processing becomes more broadly deployed, the use cases for identifying patterns that span multiple event streams will continue to grow.

Monitors

The basic EPL structure is called a "monitor" that consists of two parts:

- A **Listener** sifts through the streams of all events passed to the Correlator seeking events that match against an event expression. Listeners provide a mechanism in Apama's EPL (or Java) to express and activate an event template. As events are injected into the Correlator, each attribute is examined and compared against all event templates for all active Listeners. Listeners are expressed declaratively, which is more appropriate for complex pattern matching
- An **Action** (commonly expressed as an "action block") provides a set of imperative operations to take should the associated listener fire. If the relevant event template specified by the listener is matched, the action goes on to invoke a different action. Note that the invocation of Listeners themselves is performed by action blocks (the "onload" action) as a bootstrap operation within the Correlator

The EPL example below illustrates these concepts in the form of a simple monitor called PriceRise. The monitor is composed of three action blocks. Within the first two action blocks, listeners are registered.

```
monitor PriceRise {

        StockTick firstTick;
        StockTick finalTick;

        action onload() {
                on StockTick (symbol="IBM", price > 210.54):firstTick {
                        furtherRise();
                }
        }

        action furtherRise() {
                on StockTick (symbol="IBM", price > firstTick.price * 1.05):finalTick {
                        hitLimit();
                }
        }

        action hitLimit() {
                log "IBM has hit " + finalTick.price.toString();
                send PlaceSellOrder ("IBM", 100.0) to "Market";
        }

}
```

By default, when a monitor starts running, the "onload" action is run, creating a listener that seeks the first event defined within the event definition. In the PriceRise monitor, the "onload" action creates a listener for an event template that is looking for stock ticks about IBM® that have a price above 210.54. This monitor will effectively go to sleep until this event pattern is detected.

If it is detected, the values will be in variable "firstTick" and the action "furtherRise" will be called. In action "furtherRise" another listener is created. This listener awaits the next part of the event pattern, which involves detecting if the IBM stock price increases by more than 5 percent from its new value within 60 seconds of the firstTick event being found. Note that we are able to use the "firstTick" object to obtain the price value of the event that caused the preceding listener to fire. If the new price rise occurs, we want the event values in variable "finalTick" and to call action "hitLimit." If we do not see the expected increase within 60 seconds then we do not take any further actions and we stop looking at IBM prices. However, if the expected price rise occurs, we log the result and emit an event externally to the Correlator that is designed to place a stock order for 1000 units of IBM. This event will be picked up by an interface adapter and translated into the right message to be sent to the appropriate exchange order book.

Composite events

One can build monitors with complex temporal sequences of monitoring using these techniques. Additionally, there are capabilities to allow complex temporal and logical sequences of events to be described using a composition algebra. These composite event expressions are generally much more concise. Composite events utilize the capabilities of the temporal sequencer, as described earlier.

The example of a monitor below, called NewsCorrelation, gives an example of a composite event expression. The literal translation of the expression is "look for any news article about IBM, followed by a drop in the value of IBM to below $200 within 5 minutes." This kind of prospective impact from a news event would be of interest to a trader or a market risk analyst.

```
monitor NewsCorrelation {
        action onload() {
                on NewsItem (subject = "IBM") ->
                    StockTick(symbol="IBM", price < 200.0)
                    within (300.0) {
                            alertUser();
                    }
        }
        action alertUser() {
                log "IBM price may be impacted by latest news"
        }
}
```

Within the "onload" action a listener looks for a news item about any topic for IBM. The "->" operator means "followed by" and so once we find a NewsItem event about IBM we now look for a stock tick for IBM with a price below a threshold. As above, the "within" temporal constraint means that if this isn't detected within 300 seconds (five minutes) then the overall composite event will not fire. If we do detect the final stage within five minutes of the preceding event, then we log the fact that we've got a news-to-price correlation for IBM.

Monitor factories and multiple threads

Monitors can be created that spawn new "sub-monitors" from a template when sent an event instruction to do so. Such a monitor is known as a "factory," an extract of which is shown below. Such factories are useful for encoding a strategy as a monitor and then being able to create a tailored instance of that strategy at runtime by simply sending in an event. This use of sub-monitors is "lighter weight" than sending in an entirely new monitor.

```
action onload {
        on all NewStrategy():ns {
                spawn strategy(ns) to context ("strategyWorker");
        }
}

action strategy(NewStrategy ns) {
        monitor.subscribe(ns.symbol);
        StockTick st;
        on all StockTick (symbol=ns.symbol, price=ns.price):st {
                placeOrder(st);

        }
}
```

The concept that underlies this simple example is that many different traders might want to monitor different limits on different stocks. Control events of type "NewStrategy" are sent into this Monitor to specify what stock is to be monitored and what is the limit price of interest before an order is placed in the market automatically. On sending the "NewStrategy" event, a completely new sub-monitor is spawned and it enters action "strategy." The new sub-monitor is an identical copy of the parent monitor. The sub-monitor now monitors for the specified symbol at the specified price. However, if another "NewStrategy" event is injected then a completely new sub-monitor will be created, which will run in parallel with the first.

The key word phrase "to context" after the spawn is an optional addition to the spawn statement that specifics the new sub-monitor to be run in another context. A context is a separate event processing thread managed by the Correlator system thread scheduler. This provides easy to implement parallel processing to take advantage of modern multiple core machines. As multiple monitors and sub-monitors communicate through passing events, not shared memory, there is no memory or object locking, and therefore an EPL developer does not have to worry about the complexities of multi-thread development that affects languages like C++ and Java.

Event streams

Apama EPL allows code authors to express event-driven programs using natural event-processing constructs. An EPL program consists of a set of interacting monitors that receive, process and emit events. Monitor instances are self-contained, communicating with other monitor instances via events. An Apama application can, thus, be viewed as a dynamic network of interacting monitor instances communicating via events. Why dynamic? Because the application creates and destroys monitor instances in response to the external events received; similarly, the monitor instances dynamically subscribe and unsubscribe to particular event patterns or complex event expressions as needed. Thus, at any given instant, the application has only the monitor instances it needs and is only listening for the events of interest at that time. This novel approach makes Apama a highly efficient and responsive tool for complex event processing.

Complex event processing systems come in different flavors, one of which is event stream processing. The event stream processing approach is similar to the Apama approach but tends to involve networks that are much less dynamic. These networks are constructed from streams and processing nodes, where a processing node is typically a query, defined using declarative, relational language elements.

Event stream processing is useful in cases where one or more flows of raw events are to be converted into a set of "refined" flows of added-value events. For these operations, the use of event stream processing language elements allows these operations to be expressed more clearly and concisely than when using procedural language constructs. For this reason, Apama EPL includes event stream processing elements.

Thus, Apama stream queries are not static; they are closely integrated with the rest of the EPL language. Application developers can write code to add and remove stream queries as required, and the streams language elements allow the values controlling the stream query behavior to be varied dynamically.

```
from s in all Temperature (sensorId = "T0001")
        within 60.0
        select Out (Last(s.value), mean(s.value), stddev(s.value)):o {
                // define the upper & lower bands as mean +/- 2x StdDev
                if ( o.value  > (o.mean + o.stdv *2.0) ) or
                  ( o.value  < (o.mean – o.stdv *2.0) ) then {
                    log "Unusual Temperature for T0001";
                }
}
```

The above example creates receives a stream of "Temperature" events where the sensorId is "T0001"; it maintains a 60 second window of these events and then builds an augmented event that includes the last reading, the moving average (mean) and the moving standard deviation calculations—these last two are over the data within the 60-second window.

When we get an update to the window we then calculate some simpler upper and lower bands using the formula Band := Mean +/- 2 x StdDev. These define confidence bands around the temperature readings such that if we received a temperature reading that lies outside these bands it would be 2 x Standard Deviations out of the norm, and we would define that here as unusual. When we see an unusual reading then we log an alert, but could of course undertake any action.

Plug-ins

As already introduced, it is possible to link third-party libraries to the Correlator as plug-ins. By either coding or auto-generating a plug-in wrapper around the interface of the library, the library's operations can be exposed to EPL. Plug-ins can be developed in Java, C++ or C. The fragment of EPL below shows how a plug-in library, which has been appropriately built , can be imported and its operations invoked from EPL.

```
import "apama_math" as math
...
float a, b;
...
a := math.cos(b)
```

Writing Apama applications in Java®

Though EPL was designed and optimized for the purpose of real-time monitoring and analysis, some organizations have standardized on the Java programming language. Apama has made the features of the Correlator available in Java, and Java applications are executed through the Correlator's embedded JVM. Java also has rich libraries of functionality that an EPL developer might wish to use. A simple example would be sending an email and using the javamail library.

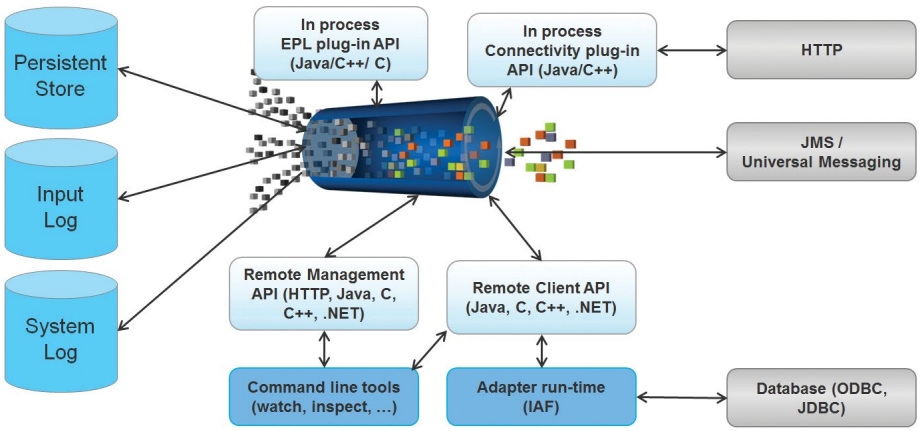
EPL code and Java can co-exist within the same Correlator engine, and communicate via the sending of events. This allows some functionality to be developed in each language were appropriate. However due to the performance benefits of EPL and EPL being designed for development of event-based applications, we recommend that EPL is leveraged by developers for high-throughput, low-latency event processing over Java.

Connecting to Apama

Apama provides a suite of off-the-shelf connectivity and integration strategies plus a number of client software development kits. These allow developers to write custom adapters or software applications that interface existing enterprise applications, event sources and user interface clients to the Correlator. Integration with event data sources is a critical component to the delivery of CEP applications and is both a key component within the Apama architecture and a key focus of Apama's engineering efforts. Apama has a dedicated engineering team that is focused exclusively upon the development and upkeep of connectivity adapters.

The correlator can directly connect to a Java® Message Service (JMS) bus or to Software AG's Universal Messaging broker. For the closest coupling, connectivity plug-ins provide a simple abstraction with high performance for in-process connectivity to external data sources. The Integration Adapter Framework (IAF) provides a separately monitorable process that can work with a range of adapters communicating with third-party messaging systems, extract and decode self-describing or schema-formatted messages, and transform them into Apama events.

All of the connectivity options are bi-directional, providing both source and sink support. In addition to receipt of disparate events and normalization for processing within the correlator, Apama can generate events that are transformed by adapters into the requisite proprietary representations of the event (e.g., message on an ESB or an order to a trading exchange) as required by third-party messaging systems. Therefore, Apama adapters are the key mechanism by which Apama can trigger actions, converting internal events into a target format that is emitted to a target external service.



Apama Connections to Other Systems

Connectivity plug-ins

Connectivity plug-ins allow adapters to be run in the same process as the correlator, providing a tightly coupled mechanism to send or receive events from external ystems. A simple configuration file specifies the plug-ins used and provides powerful configuration data for plug-ins to use.

A number of connectivity plug-ins can be used together to form a chain of plug-ins to separate transports from mapping or decoding functionality. Connectivity plug-ins can be written in Java or C++, and a mixture of plug-in types used within the same chain. Plug-ins handle events in a native, protocol-neutral key-value "map" format (using the standard Java "Map" interface).

Sample plug-ins provided include:

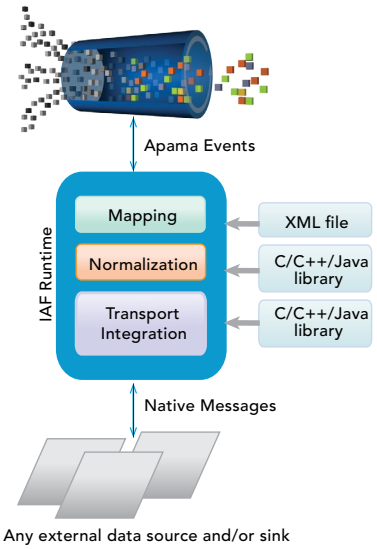
- JSON™ codec to convert Apama events to/ from JSON form
- A simple HTTP client
- A simple HTTP server
- Classifiers for identifying the type of events via configuration
- Mapper to provide rule-based transformation of event fields

Apama's Integration Framework

Adapters can also be hosted in Apama's Integration Adapter Framework (IAF). The IAF is a middleware-independent and protocol-neutral adapter framework that is designed to allow easy and straightforward creation of software adapters to interface Apama with middleware buses and other message sources. All events, regardless of source are converted to the internal format, thus enabling Apama to natively support correlations that span disparate external data formats.

As illustrated, there are three primary components to the integration framework:

- **Transport layer**: Communicates with the event stream's message source/sink. Its functionality is defined through one or more Apama or user-provided transport plug-ins that are written in C, C++ or Java
- **Codec layer**: Translates messages from any custom representation into a normalized form and vice versa. Provided by Apama or the customer, they can be written in C, C++ or Java
- **Semantic mapper layer**: Using XML, transforms the normalized messages produced by the codec layer into Apama events and vice versa using a set of mapping rules



The IAF supports highly configurable and maintainable interfaces and semantic data transformations. An adapter generated with the IAF can be re-configured and upgraded at will and, in many cases, without having to restart it. Its dynamic plug-in loading mechanism allows a user to customize it to communicate with proprietary middleware buses and decode message formats.

Client library Software Development Kits (SDKs)

The client library allows developers to embed an Apama library inside their own or third-party applications and send events to and from a correlator, or even monitor and control correlators. These custom applications can be written in C, C++, Java and Microsoft® .NET.

The interfaces, in order of increasing abstraction, are:

- Event client API: Base layer upon which the other API layers are built, allowing sending and receiving of events and monitoring and control of correlators (only this layer is available in the C++ SDK)
- JavaBeans® API: A Java only API that is more powerful than the message-level API and provides extensive higher-level functionality
- EventService API: Consider events as encapsulated objects at the API level
- ScenarioService API: Used to provide an interface to queries that have been built with Designer, or data exposed by applications written in EPL

The SDKs provide client Application Programming Interfaces (APIs) that allow one to directly connect to Apama and transfer events in and out. The SDKs provide none of the abstractions and functionality of connectivity plug-ins or the IAF and, hence, their use is only recommended when a developer needs to write a highly customized and very high-performance software client, such as a user interface.

Software AG Designer

In support of designing and writing Apama applications and services, Apama provides a comprehensive toolset called Software AG Designer. It is an Eclipse™-based Integrated Development Environment (IDE) for development, debugging, testing, profiling, back testing and deployment. Designer's use of Eclipse delivers many natural benefits to Apama developers, including the abilities to:

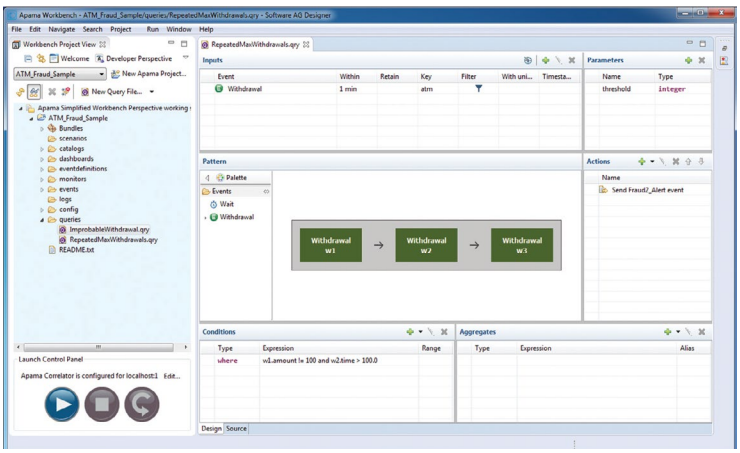
- Mix/match development tasks
- Take advantage of integration with third-party tools, such as source code repositories
- Create project definitions that manage source code components
- Incorporate specific features as plug-ins

Designer supports a number of perspectives. An Eclipse perspective is a named organization of views, menus and toolbars that can be saved and switched to a unique tab of the organized for a particular task. Software AG Designer perspectives include:

- Developer/workbench/runtime for the development of applications
- Debugger for debugging EPL applications
- Runtime for monitoring the execution of running services, such as Correlators and adapters), and applications, such as monitors
- Profiler for analyzing the execution statistics of running monitors

The developer and workbench perspective of Designer provides the common proj-ect orientation for construction of Apama-based applications and services. A project includes many aspects of an application, including:

- Monitors
- Apama queries
- Dashboards
- Adapters (connection to external data sources—inbound and outbound)



Apama Query Graphical Editor

Software AG Designer provides all the standard state-of-the-art capabilities expected in modern IDEs for:

Editing

- Syntax highlighting of errors in statements
- Semantic checking to facilitate the automatic completion of statements
- Code outlines that provide tree-based structures for EPL code content
- Automatic formatting that provides automatic bracketing and indentation of code for enhanced interpretation
- Quick reference help for keywords and event-type definitions

Debugging
Designer includes a state-of-the-art source code debugger to debug EPL applications. It includes these features:

- Visual debugging based on the well-known Eclipse interface
- Set code-level breakpoints
- Single stepping and continue from breakpoint
- Call stack display and variable inspection
- Does not require "debug" compilation mode--therefore, you can debug a production deployment
- Command line interface (optionally supported)
- Remote (networked) connectivity

Profiling
Designer also includes a profiler. This is a tool for inspecting running Apama monitors. The profiler helps to guide developers towards hotspots in application code. This provides a huge benefit in getting the most out of Apama and the EPL language. It:

- Includes a number of profile views into running monitors (summary, execution statistics, invocations, etc.)
- Shows frequency of calls to actions
- Includes average, max/min and cumulative CPU times (real and percentages)

Extensive tooling for simple and rapid application development and deployment
Apama Studio provides a number of other features, some of which are detailed in other sections of this document:

- Build and deploy outside of Apama Studio
- Back testing through the Apama DataPlayer and Database Connector (ADBC)
- Direct integration with source code control systems, such as Apache Subversion™ and ClearCase®
- Java monitor development

Apama queries

Apama queries let business analysts and developers create scalable applications that process events originating from very large populations of real-world entities. Scaling, both vertically (same machine) and horizontally (across multiple machines) is inherent in Apama query applications.

Apama queries can be used alongside EPL monitors in the same correlator process, interacting by sending events between them. Incoming events that queries process are partitioned by, for example, customer account numbers, car license plate numbers, devices or some other entity. In a query application, the correlator processes the events in each partition independently of other partitions.

Apama is often deployed in situations where requirements can change quickly and it's imperative to ensure that applications are developed quickly and evolved over time to address changing circumstances. Another aspect is often to ensure that the tools be made are accessible to business users, who can take greater portion of control of the applications over time and who are inevitably the source of the "business logic" that drives many event processing applications. Apama queries are designed to be easy to develop for both the business analyst and the application developer. Graphical tools to specify the application design and full round-trip engineering allow both the business analyst and the developer to work on the same queries. At the developer level, an Apama query is defined using the Apama EPL.

Queries comprise:

- Parameters—an optional list of values that can be specified at runtime
- Inputs—a list of event types which the query will consume; they are organized into windows of a time duration or number of events, and partitioned by the key fields of the event
- Pattern—an event pattern, ranging from a simple query looking for a single event to complex patterns involving multiple event types
- Conditions—optional restrictions on the event pattern, including time constraints, conditions on the contents of the events, or for detecting absence of events
- Aggregates—optional aggregation such as averages, counts or user-defined aggregate operators
- Actions—action to execute when the query fires—either EPL or sending an event to be output or processed further

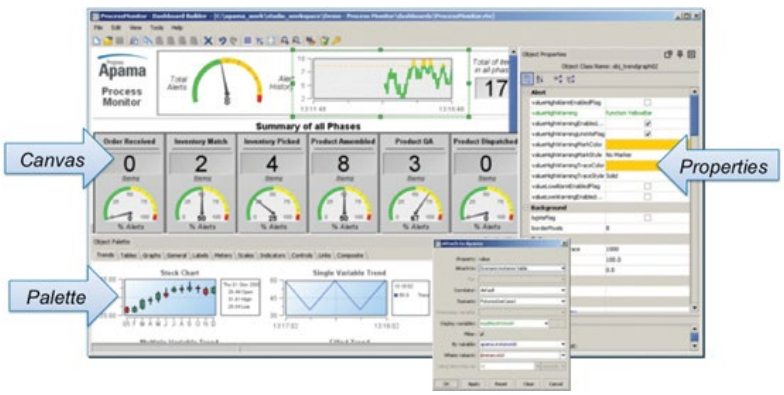
Note: the scaling features of Apama Queries are only available in the commercial edition.

Apama dashboards

Apama dashboards are a key component in the overall architecture as customers employ dashboards to initiate and parameterize, monitor and terminate their applications. Apama dashboards are highly customizable with complete flexibility in terms of layout and a wide range of objects with which to render the execution of applications.

Apama Dashboard Builder

Apama provides a rich, customizable interface that requires no programming to create dashboards that are tailored to customer requirements. The Dashboard Builder creation environment offers an intuitive graphical interface with which to rapidly prototype and deploy dashboards. Once created, dashboards can be re-used, applying the same graphical display to different event processing applications.



Dashboard Builder Elements

Dashboard creation is performed through a drag-and-drop interface. As shown in the figure above, a palette is available (lower left) from which dashboard designers can select objects that are placed on a canvas (upper left). Once on the canvass, the object's attributes are available in the right panel for customization of the design elements. Among the objects available to designers are trend charts, tables, graphs, meters, scales and various other graphical elements.

Rather than a monolithic display, dashboards are composed of discrete panels, each of which can be associated with a specific Apama Correlator. Thus, the dashboard can serve as a portal to a distributed implementation of Apama in which it provides a single common view into data provided by a correlator that is logically distributed. Dashboards can be dynamically assigned to correlators while the Correlators are running, thus enabling updates and enhancements to be incorporated with no system downtime.

Dashboards can incorporate navigable drill-downs that allow the user to click on a summary view and see the underlying data that has been aggregated into the higher level view. Apama also supports overlay charts that can be rendered as a composite object in a single view to provide greater context and clarity than might be available from a single graphical depiction.

In addition to the mathematical calculations available through the Apama Correlator, Apama dashboards themselves support mathematical functions that can operate on dashboard data values. The functions, similar to those available in Microsoft® Excel®, greatly extend the display sophistication of the dashboards in a fashion that is highly complementary to the core processing power of the underlying engine.

Other development tools

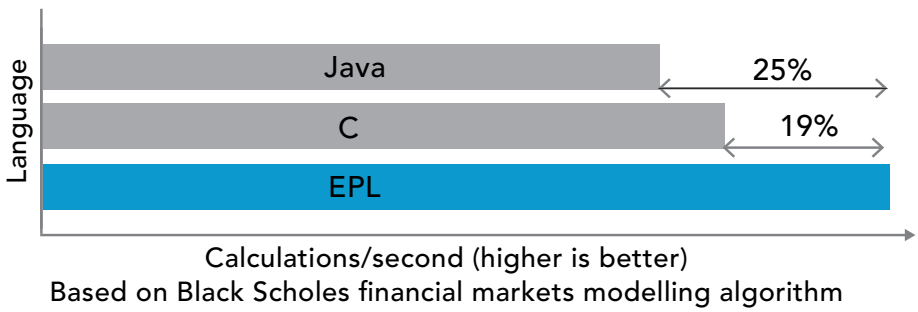
In addition to Software AG Designer, Apama includes:

- Pysys test framework and extensions for use with Apama—tools to aid in building reproducible test cases
- Ant macros to automate deployment of Apama applications
- Code coverage recording – records which lines of EPL have been executed; a report in HTML form can be generated, showing which lines have been completely, partially or not executed at all
- Memory profiler—allows statistics of memory use to be retrieved from a correlator, which can be visualized in a spreadsheet, such as Microsoft Excel

Benchmarks

When C/C++ or Java programs are compiled using standard methods for general distribution they typically target the lowest common denominator CPU architecture since the developer often does not know exactly which CPU will be used. However, recent microarchitectures for Intel® CPUs extend this lowest common denominator architecture with optimized instruction sets. An application compiled to run across all architectures will not be able to take advantage of these new instructions or correctly optimize for the latest CPUs.

The Apama Correlator compiles its EPL on every application start-up (as each piece of code is "injected"). This modern, sophisticated compiler technology has proven to outshine many other commonly used compilers. It has been demonstrated using numerous benchmarks, including a computationally intensive Black Scholes option pricing benchmark (results shown above), that this new method of running EPL will execute faster than the equivalent implementations in Java and C/C++ when compiled for general use across any CPU architecture.



Comparing Optimized EPL Performance with Java and C/C++

Complex fraud prevention

In this benchmark, there are these scenarios:

1. Unusual transaction detection: Detect large purchase amounts over a channel between business hours, or multiple subsequent small withdrawals after hours. The goal of this scenario is to quickly identify unusual transactions based on the time of ay and the channel, attempting to identify unusual patterns of behavior.

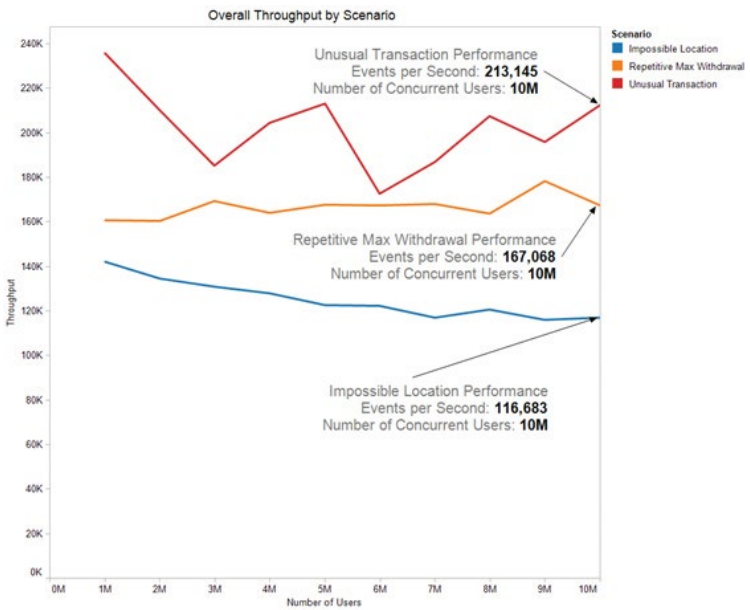
2. Repetitive maximum withdrawals: Detect multiple cards being used on the same ATM machine, where the withdrawal amounts are all between some range of values to indicate high withdrawal amounts. This scenario aims to detect a situation where someone has stolen a number of cards, and is using the same ATM repetitively for each card to withdraw high amounts (near the max) within a time window.

3. Impossible location: Detect when the same card has been used in two different locations within four hours. If the first location is greater than some distance from the second location, it should create a fraud notification. The business purpose for this is to determine if a card has been stolen or possibly cloned and is now being used at a location whose distance doesn't make sense based on the time frame.

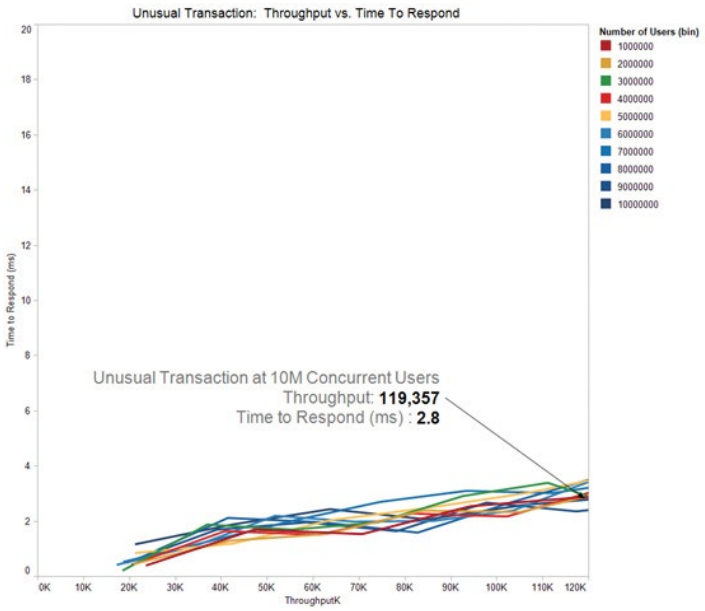
Hardware used in these tests:

• Benchmarks run on dual Intel® Xeon® X5570 with RHEL 5

Results:



The above chart show the performance of these three scenarios running from between 1 million and 10 million concurrent users. The critical aspect here is in the gradients of the curves of these results, the Impossible Location scenario clearly demonstrates the logarithmic performance curve of the HyperTree: user count (and thus queries) increases by a factor of 10 or **1,000** percent, but the performance drop is from 140,000 to 120,000 or **15** percent.

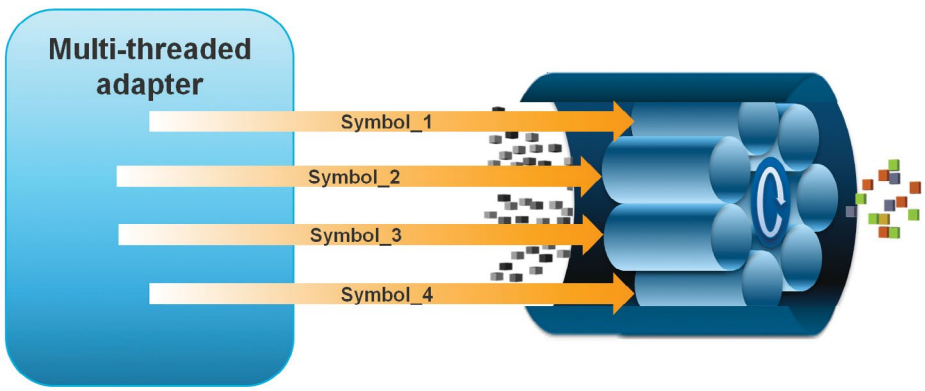


Time to respond varies little with number of concurrent monitored users and stable.

Scalability

Earlier sections discuss the efficient vertical scalability of Apama, enabling applications to easily maximize the available resources on a single host. Apama applications can also be composed in flexible topologies where applications can be partitioned in parallel or serial paths and event streams partitioned or combined.

For example, if the logical design of the application permits segmenting the event flow based on event attributes, Apama applications and adapters can partition and/or multiplex event streams across multiple channels as in the diagram below. This partitioning can be implemented by the adapters that send the events to the correlators or by using a Correlator do perform the partitioning and/or multiplexing.



Here the symbol names capture business context and events are routed to a particular channel based on their symbol. Applications can run on a context per symbol, each subscribed only to the events for that channel. For example, an algorithmic trading application may split the events by stock symbol. By using Apama's built-in lightweight scheduler, a very large number of contexts can be declared and will be run efficiently on parallel hardware, even if some receive high event rates and others only receive low-event rates, with low latency response across contexts.

Event delivery from adapters to application and back to adapters can be fully parallelized. See also the Intel benchmarks demonstrating high scaling of event delivery and processing across a large number of cores.

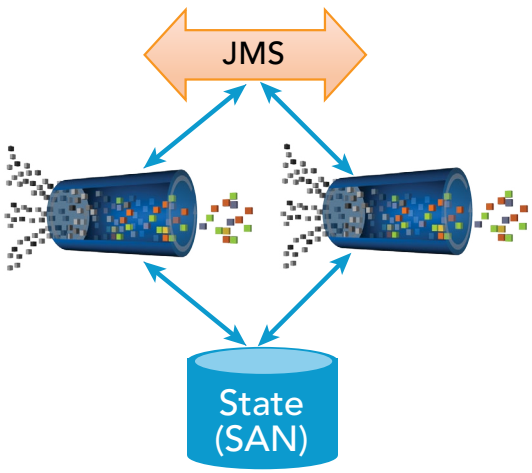
Note: there are limits to scalability in the community edition. The above scaling benchmarks are achievable using the commercial edition.

Resiliency

There are a number of built-in capabilities to support a wide range of resilient deployments of Apama, in addition to these there are a number of aspects that allow customized resilient architectures to be deployed.

The Apama recovery mechanism links a Correlator to a resilient JMS bus. EPL deployed into that Correlator merely need by tagged with a "persistent" keyword. When running normally the Correlator receive events form the bus and processes them; transparently it also periodically records incremental updates to its external state on disk in such a way that it still delivers high-performance capabilities. If a failure occurs a new Correlator can be started; it then loads the last snapshot from the disk and automatically asks the resilient JMS bus for those events that occurred between the last snapshot and now; when processed the Correlator transparently continues to process live events.

This architecture can be extended to cover a parallel host that is capable of running a second Correlator that has access to both the JMS bus and the snapshot state information. In this case a failure of either the Correlator or the hardware it is running on can be recovered by using the second host.



Take the next step

For more information on how Software AG can help, talk to your Software AG representative, or visit **www.softwareag.com**.

ABOUT SOFTWARE AG

The digital transformation is changing enterprise IT landscapes from inflexible application silos to modern software platform-driven IT architectures which deliver the openness, speed and agility needed to enable the digital real-time enterprise. Software AG offers the first end-to-end Digital Business Platform, based on open standards, with integration, process management, in-memory data, adaptive application development, real-time analytics and enterprise architecture management as core building blocks. The modular platform allows users to develop the next generation of application systems to build their digital future, today. With over 45 years of customer-centric innovation, Software AG is ranked as a leader in many innovative and digital technology categories. Learn more at www.SoftwareAG.com.

© 2016 Software AG. All rights reserved. Software AG and all Software AG products are either trademarks or registered trademarks of Software AG.
Other product and company names mentioned herein may be the trademarks of their respective owners.

SAG_The_Apama_Platform_20P_WP_Jun16

software AG