

BEST PRACTICES:  
APPLICATION PERFORMANCE MANAGEMENT

JAVA AND .NET



# Java and .NET best practices

Whether or not you've read the earlier papers in this series, it should be clear application performance management (APM) is increasingly important as end users rely on ever-more complex applications to enable critical business transactions. In many cases, applications do more than simply enable the business, they are the business; business performance is immediately and directly related to application performance.

Whether you're well on the way to adopting Service-Oriented Architecture (SOA), experimenting with web services or "front-ending" legacy applications with modern user interfaces, it's a safe bet your application infrastructure is much more complex today than yesterday. And what are the most common reasons for undertaking these endeavors? Enhance business agility and innovation while reducing costs. In short, IT must better support the needs of the business.

What happens when IT's APM approach does not align with the performance goals of the business? The business loses revenue due to dissatisfied users, diminished brand equity, loss of productivity or competitive disadvantage. This quickly translates to a loss of credibility for IT operations. IT's reactive response to performance problems is inherently inefficient. Take, for example, allocating unnecessary infrastructure resources in a blind attempt to solve the problem with added capacity or pulling staff from strategic projects to solve complex and poorly understood problems. The greater the inefficiency, the greater the impact on the business. The resulting higher cost of IT services not only affects the business, but also IT. As IT services become more competitive, the risk of outsourcing increases.

## BUSINESS APPLICATION PERFORMANCE MANAGEMENT

When considering APM, keep in mind the goals of identification, prioritization and resolution of performance and availability problems. This discipline—directly related to ITIL's Incident and Problem Management—should be aligned or integrated with business goals by applying the business's definitions of performance and availability.

How do we measure application performance from the business viewpoint? From the end-user perspective, not by the performance of CPU, network, Java method or SQL query, nor even JSP/ASP/servlet. While these and other examples of "internal" or component-level metrics are important for root-cause analysis and troubleshooting, as well as capacity management (and therefore critical features of an APM solution), they, by themselves, are not business goals. Similarly, many business goals are too abstract to be directly meaningful to APM. Take, for instance, the goals of 95 percent customer retention or a 48-hour window for underwriting new insurance policies. The intersection of business and IT performance metrics is end-user transaction response time, because it is both measurable by IT and meaningful to the business.

## MODERN APM CHALLENGES

The failure of traditional APM approaches is essentially a story of how collections of infrastructure tools, crafted into performance management solutions, have not kept pace with the increasing complexity of application environments.

In simpler times, it is easy to envision how monitoring the health of the physical system—network, server, disk storage, etc.—reasonably could provide reliable insight into end-user experience. Think of environments such as telnet and 3270 applications, the more recent two-tier client/server environments or even today's simple static web sites. As these systems became more complex, performance management solutions continued to add monitoring capabilities for databases, firewalls, load balancers, web and application servers, SANs, etc. IT also monitored the

activities of the operating system and application environments. Faced with a large number of consoles and the resulting “silo” approach to performance management, IT intended to provide an “end-to-end” view of system performance by viewing these metrics on a single dashboard. Logical grouping of these component monitors offered a representation of the collection of devices that support specific applications. Given today’s complexity, this repackaging of traditional monitoring tools provides very narrow views, suitable for domain experts to diagnose problems with equipment or defend their part of the delivery path. They cannot, however, provide holistic and reliable insight into the subtleties of application performance as perceived by the end user, for these two reasons:

- Component performance may have little impact on actual end-user experience (EUE); certain thresholds may be inconsequential for some transactions, devastating for others.
- Similarly, and with a more significant negative impact, degraded EUE may be nearly invisible to component-level monitoring.

## PERFORMANCE MANAGEMENT INSTRUMENTATION FOR JAVA AND .NET

Specifically for the managed application environments of J2EE and .NET, performance management has evolved to offer built-in capabilities to help uncover performance bottlenecks. For Java, the Java Management Extension (JMX) is an architecture that enables the management of Java resources (application servers as well as the Java Virtual Machine (JVM)) without significant development investment. The JVM includes built-in JMX instrumentation, and applications can become JMX-manageable if they embed a managed object server. Similarly, the Microsoft .NET Framework makes management Common Language Runtime (CLR) information available using the Windows Management Instrumentation (WMI) interface, providing access to .NET-specific counters and measurements.

JVM and CLR instrumentation provide access to information about the managed environment, including:

- memory consumption
- garbage collections
- thread contention and execution statistics
- JVM/CLR statistics.

These interfaces also provide different degrees of information about external (unmanaged) resources, including the operating system. WMI is significantly more capable in this area.

The parallel here to component-level monitoring is clear; solutions based solely on these interfaces remain focused on the statistical impact of low-level metrics, often without context or correlation to real performance concerns.

The JVM Tools Interface (JVM TI) is a programming interface used by monitoring tools, offering application-specific profiling capabilities such as insight into the execution of methods and classes. (JVM TI replaces the JVM Profiler Interface, JVMPPI, beginning with version 1.5.) For .NET, the equivalent is called the .NET Profiler API. These interfaces can provide information about the execution of the application itself, enabling function call tracing, timing, error reporting, memory analysis and more.

Essentially, we have two levels of Java and .NET monitoring:

- the Java or .NET application server, where we can monitor the managed runtime environment
- the Java or .NET application, where we can monitor the execution of an application.

## JAVA AND .NET APPLICATION PERFORMANCE MANAGEMENT

Let’s consider the second level, application monitoring. These interfaces (JVMTI and .NET Profiler API) trace transactions in great detail and permit a management application to “follow” the sequence of events: As one method executes, the interface calls another method, which calls another (perhaps a database query or remote process). Method timing, CPU usage and parent/child relationships can all be monitored.

While these capabilities appear, and in many cases are, compellingly enlightening, there are some important implementation and capability considerations:

- Monitoring overhead can be quite large given the level of instrumentation available and, therefore, limit use in production environments.
  - To avoid the heavy penalty of state transitions, the monitoring application may use byte code instrumentation, allowing the VM to run uninterrupted (at full speed) since the agent code is standard byte codes.
- Monitoring can remain focused on statistical measurements, limiting insight into more complex or intermittent performance bottlenecks.
  - Transaction-level insight is usually required to isolate and trace problems that often get “lost” in statistical sampling or averaging.
  - Dynamic tracing of an individual instance of a transaction—via approaches such as manual configuration, transaction “tagging” or threshold monitoring—is important to keep overhead low.
- Metrics can remain isolated, lacking transaction context and dependency insight, especially in complex environments.
  - To satisfy even basic enterprise-level analysis, IT staffs need the ability to follow transactions across multiple tiers in order to maintain visibility into method dependencies.

- Configuration can require intimate knowledge of the application architecture and structure.
  - To avoid the often impossible challenge of determining what level of detail to monitor beforehand, dynamic or conditional instrumentation permits a practitioner without knowledge of the application code to drill down into interesting or problematic portions of the code and effectively manage and analyze performance.
- Measurements may lack correlation with end-user experience, and therefore offer little insight into business impact.
  - To ensure analysis and tuning efforts are aligned with end-user performance—and therefore business performance—method-level bottlenecks should be understood in context of their impact on end-user experience.

## AN EFFECTIVE APM METHODOLOGY

Assuming these capabilities, let's examine a methodology for Java/.NET performance management. There are four facets to this methodology:

- a proactive “consumption-oriented” profiling or opportunistic approach to tuning performance. It is best applied in either a load test environment prior to production, or in production for periodic tuning checks.
- a reactive troubleshooting approach is appropriate for responding to EUE alerts, whether from end users or (more efficiently) from EUE monitoring solutions.
- an approach to identify memory allocation problems.
- continual service improvement.

These methods can be applied effectively in both load test and production environments.

### 1. Profiling/Finding optimization opportunities

Filter the analysis to focus on the pertinent time period (i.e., the time the transaction or script was executing).

- Identify the methods with the highest percentage of CPU time not in child calls.
  - If the method has a high average response time, this is a method whose code should be examined for optimization.
  - If the method has a high number of invocations (each with a small CPU time component), examine the parent method for optimization opportunities.
  - If one invocation of the method results in an abnormally high response time, treat this as an anomaly that requires troubleshooting.
- Identify SQL calls with the highest percentage of CPU time not in child calls.
  - If the SQL query has a high average response time, this query should be examined for possible optimization.
  - If the SQL query has a high number of invocations (each with a small CPU time component), examine the parent method for optimization opportunities.
  - If one invocation of the query results in an abnormally high response time, treat this as an anomaly that requires troubleshooting.
- Identify methods with the highest percentage of elapsed time not in child calls and with low percentage of CPU time not in child calls.
  - If the method has a high average response time, it is waiting on something external (i.e., I/O, COM).
  - If the method has a high number of invocations (each with a small CPU time component), examine the parent method for optimization opportunities.
  - If one invocation of the method results in an abnormally high response time, treat this as an anomaly that requires troubleshooting.
- Identify poorly performing transactions that impact the largest number of end users. (These don't have to trigger a performance alarm, but may instead require ongoing service improvement/tuning.)
  - Determine optimization needs for those transactions/end users by following a troubleshooting methodology.

## 2. Troubleshooting/Responding to end-user complaints or performance alerts

For a response time alert or user complaint, identify the transaction based on alert correlation, timestamp, end-user ID or IP address, and/or transaction identifier (i.e., URL).

- Find the method associated with this transaction that has the highest percentage of CPU time not in child calls.
  - If the method has a high average response time, this is a method executing code that should be examined for optimization.
  - If the method has a high number of invocations (each with a small CPU time component), examine the parent method for optimization opportunities.
- Find the SQL query associated with this transaction that has the highest percentage of CPU time not in child calls.
  - If the SQL query has a high average response time, this query should be examined for possible optimization.
  - If the SQL query has a high number of invocations (each with a small CPU time component), examine the parent method for optimization opportunities.
- Find the method associated with this transaction that has the highest elapsed time not in child calls.
  - If the method has a high average response time, this is a method executing code that is waiting on something external (i.e., I/O, COM).
  - If the method has a high number of invocations (each with a small CPU time component), examine the parent method for optimization opportunities.

## 3. Finding memory leaks

- Find growing collections. These are collections of objects that are not being deallocated at the same rate as they are allocated and will eventually use all of the HEAP storage.
  - Determine which application method creates these objects and correct the problem.
- Find orphaned collections. These are collections of objects that have not been modified in a long time, perhaps days. They will create HEAP fragmentation, slowing down the system with heavier garbage collection activity and wasted HEAP space.
  - Determine which application method creates these objects and correct the problem.
- Find methods that cause excessive allocations and deallocations; while these will not consume all the HEAP storage, they may slow down the system.
  - Determine which application method creates these objects and correct the problem.

## 4. Continual service improvement

Continual service improvement (CSI) should not be a project implemented only when something has failed. Instead, IT efforts should proactively look for opportunities to improve performance and reduce resource consumption. This, in turn, improves the efficiency of the IT service. CSI offers the greatest benefit to application environments that are reasonably stable, where performance is mostly consistent under normal conditions.

- Trigger an alert when a monitored high-level component is slow to complete. For Java, these components include JSPs, servlets and EJBs. For .NET, the component would be ASPX. The alert should dynamically trigger a trace of the transaction's execution to identify the root cause of the high-level delay.
  - Communicate the root cause to the correct team.
  - Include frequent low-level culprits in ongoing monitoring so they can be tracked with more visibility.
  - In connection with measured end-user experience, identify practical thresholds so you can trigger an alarm prior to actual user impact.
  - Create trending reports to provide visibility into service quality trends.
  - Create before/after performance reports to illustrate the impact of optimization applied to application components.

## SUMMARY

The managed application environments of Java and .NET offer unique APM opportunities that, when complemented with robust end-user experience monitoring, can provide a level of alignment between business and IT performance management unattainable in most other environments. Organizations managing or planning application architectures to meet today's and tomorrow's needs should take advantage of these capabilities. The result is dramatic improvement in IT efficiency.

## ABOUT THE AUTHOR

---

### GARY KAISER, APM SUBJECT MATTER EXPERT, COMPUWARE CORPORATION

Gary has more than 25 years of technical expertise in network and application performance, with consulting, management, end-user and teaching experience. He has authored numerous technical papers and speaks frequently on performance topics, and is co-inventor of key ApplicationVantage capabilities, including Active Latency Detection and the Transaction Expert.

---

### FRANCIS CORDON, SUBJECT MATTER EXPERT, ITSM, COMPUWARE CORPORATION

Francis Cordon has worked in IT for 12 years, mostly in IT Service Management. At Compuware, Francis has had many roles, including ITSM Post-Sales Consultant and Technical Account Manager. He was an early adopter of our J2EE/.NET monitoring solution and has contributed to the growth and improvement of this technology. In his current role as ITSM Subject Matter Expert, Francis focuses on J2EE and .NET Application Performance Management in seamless integration with End-user Experience Monitoring.

To learn more about Vantage, visit:  
[www.compuware.com/vantage](http://www.compuware.com/vantage)

Founded in 1973, Compuware provides software, experts and best practices to ensure applications work well and deliver business value. Our unique approach, Business Service Delivery, helps CIOs **optimize end-to-end application performance** for leading businesses around the world, including 46 of the top 50 Fortune 500 companies. Learn more at [compuware.com](http://compuware.com).

**Compuware Corporation** Corporate Headquarters  
One Campus Martius  
Detroit, MI 48226-5099

All Compuware products and services listed within are trademarks or registered trademarks of Compuware Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries. All other company or product names are trademarks of their respective owners.

08.09

© 2009 Compuware Corporation

