CA Wily Introscope®

.NET Agent Guide

Version 8.2

Date: 10-2009

For help with Introscope or any other product from CA Wily Technology, contact Wily Technical Support at 1-888-GET-WILY ext. 1 or support@wilytech.com.

If you are the registered support contact for your company, you can access the support Web site directly at www.ca.com/wily/support.

We value your feedback

Please take this short online survey to help us improve the information we provide you. Link to the survey at: http://tinyurl.com/6j6ugb

If you have other comments or suggestions about Wily documentation, please send us an e-mail at wily-techpubs@ca.com.

6000 Shoreline Court, Suite 300
South San Francisco, CA 94080

US Toll Free 888 GET WILY ext. 1
US +1 630 505 6966
Fax +1 650 534 9340
Europe +44 (0)870 351 6752
Asia-Pacific +81 3 6868 2300
Japan Toll Free 0120 974 580
Latin America +55 11 5503 6167
www.ca.com/apm

Table of Contents

SECTION I

Installation Requirements and Options

The chapters in this section have information to help you understand, plan, and perform the process of setting up an Introscope .NET Agent and instrumenting your applications.

CHAPTER 1

The .NET Agent Overview

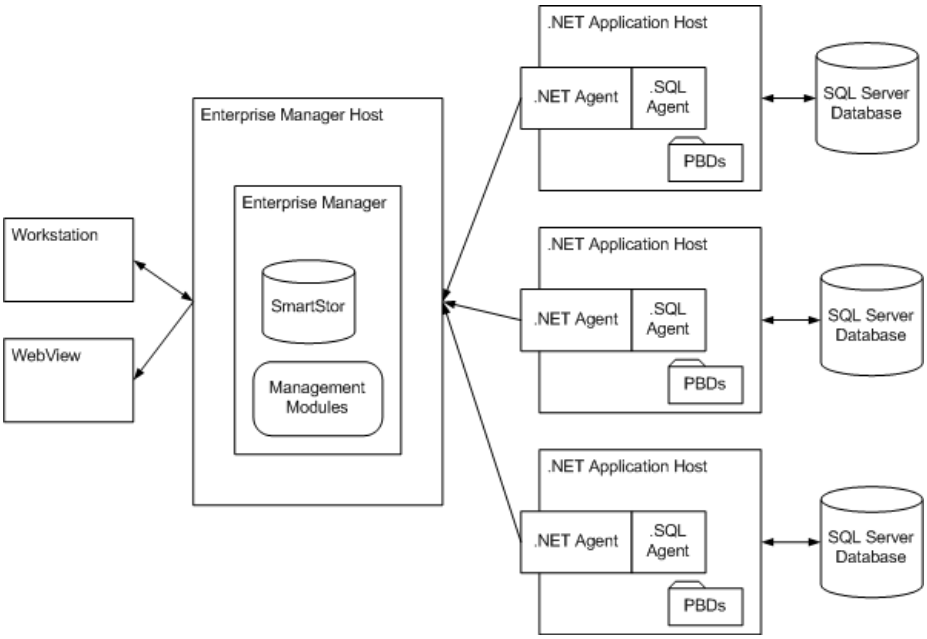
This chapter is an introduction to the Introscope .NET Agent.

.NET Agent overview

The .NET Agent is an application management solution for Managed Enterprise .NET applications. The .NET Agent monitors mission critical .NET applications running in Microsoft's Common Language Runtime (CLR) environment, providing visibility to the component level.

In an Introscope deployment, an agent collects application and environment metrics and relays them to the Enterprise Manager. An application that reports metrics to an Introscope agent is referred to as *instrumented.* After you install and configure the .NET Agent on a system, the applications that run there are automatically instrumented at start up.

This illustration shows a simple Introscope deployment. Larger more complex deployments can have more agents and multiple Enterprise Managers.



By default, only IIS ASP.NET applications active on a system are instrumented. It is possible to instrument any .NET application, including IIS ASP.NET applications as well as stand alone .NET executables. It is also possible to only instrument a subset of, or specific, applications. For more information on instrumenting specific applications, see Configuring instrumentation on page 34.

The instrumentation process is performed using *ProbeBuilding* technology, in which *tracers,* defined in ProbeBuilder Directive (PBD) files, identify the metrics an agent gathers from applications and the CLR at run-time.

The Enterprise Manager stores the metrics reported by multiple agents. End users access metric data using the Introscope Workstation client or the WebView application. The Introscope Workstation client or the WebView application allow users to monitor applications, determine the likely source of performance and availability issues, and diagnose problems.

The .NET Agent operating environment

In your Introscope deployment, you install the .NET Agent on each system that runs an application you wish to monitor.

The lifecycle of the .NET Agent and AutoProbe

The lifecycle of an ASP.NET application is managed by the Internet Information Server (IIS). The lifecycle of a .NET Agent is also controlled by IIS.

The startup process for the .NET Agent and AutoProbe components starts when the managed application code (`.aspx`, `.asmx`, etc.) is first exercised.

Until a user request for an application is received by IIS, the .NET Agent is not active, and does not appear in the Introscope client views in the Workstation or WebView. Similarly, if an instrumented application stops experiencing user activity, IIS stops the application process and the .NET Agent. As a result, the node for the .NET Agent in the Introscope Investigator will be grayed out.

The idle time for the .NET Agent can be configured in the IIS Manager. This allows you to set the amount of time an application is idle before being "timed out" and appearing grayed out in the Introscope Investigator.

To configure application idle time:

1. Open the IIS Manager.
2. Right click the application you wish to configure and select **Properties**.
3. Navigate to the **Home Directories** tab.
4. Click the **Configuration** button at the bottom of the tab.
5. Navigate to the **Options** tab.
6. Make sure the check box for **Enable Session State** is selected and set the idle time in minutes.
7. Click **OK**.

How the .NET Agent works with your applications

These are the key steps that occur when a managed .NET application, monitored by Introscope, starts up.

» **Note** Stand alone (non-ASP.NET) applications are bootstrapped by Windows, skipping step one and two below.

Step 1 IIS receives a user request for an application.

Step 2 IIS starts the .NET worker process.

Step 3 The Managed .NET application starts up.

Step 4 The CLR starts the AutoProbe Connector.

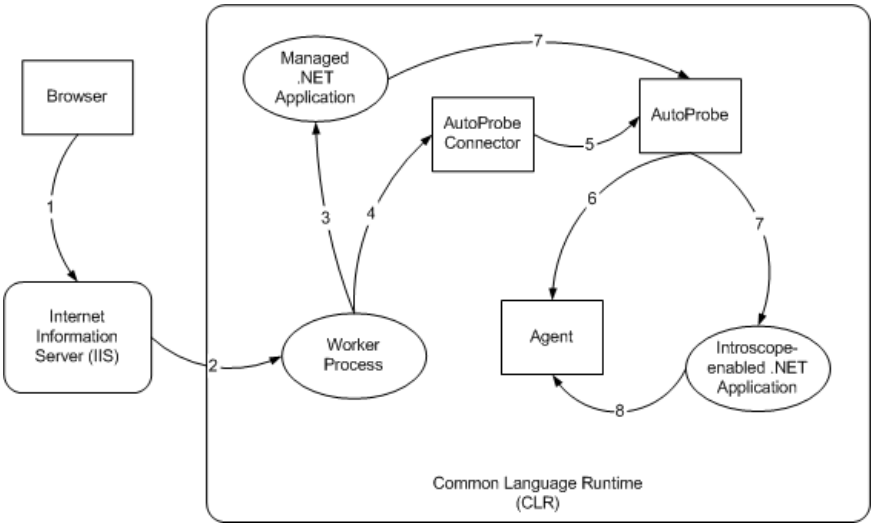
Step 5 The AutoProbe Connector looks up AutoProbe in the Windows registry and invokes AutoProbe.

Step 6 AutoProbe loads the .NET Agent from the Global Assembly Cache.

Step 7 AutoProbe instruments the managed .NET application.

Step 8 The instrumented application starts reporting metrics to the .NET Agent.

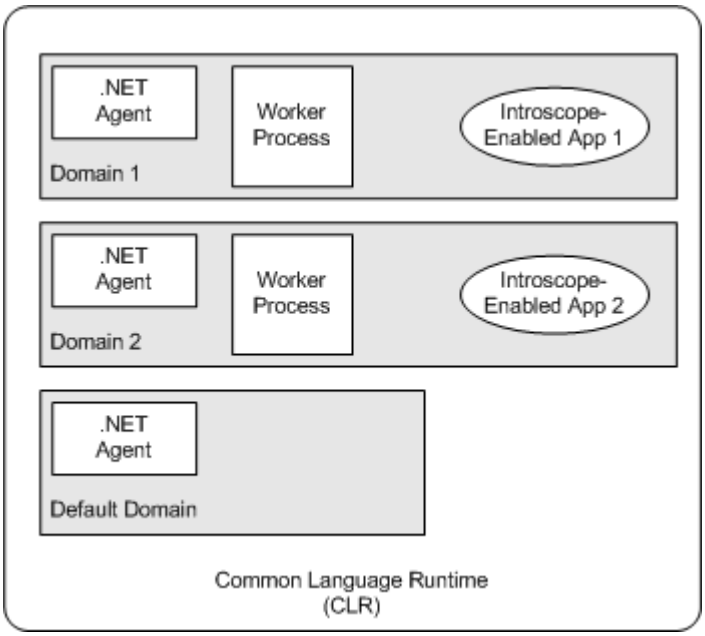
The numbered arrows in this illustration correspond to these steps.



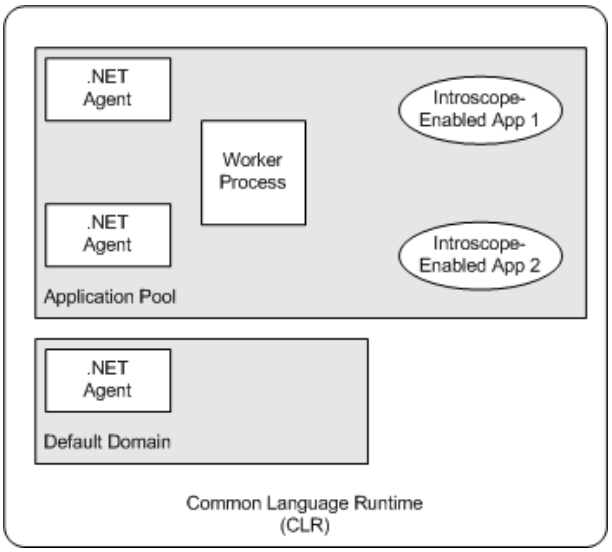
.NET Agent instantiation

You install a .NET Agent on each system that hosts the managed .NET applications you wish to monitor. At the time of the .NET Agent startup, one agent instance is created for the CLR's default domain. In addition, one .NET Agent instance is created for each application running in the CLR as illustrated in the illustration below.

This illustration shows a managed ASP.NET application that has one IIS worker process:

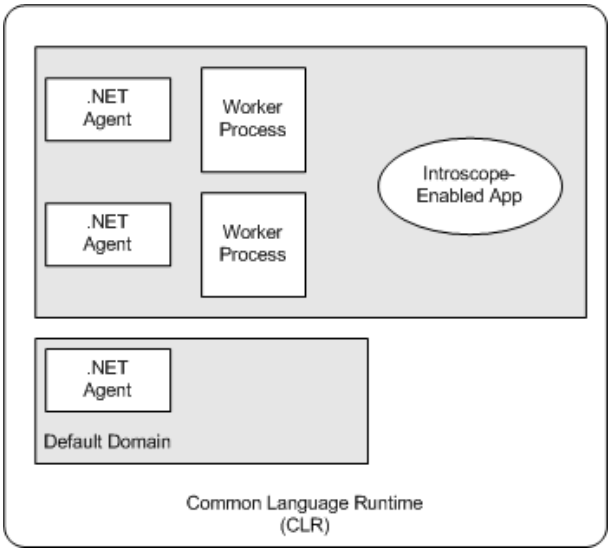


If multiple .NET applications are grouped together in an IIS application pool that share a single worker process, there is one .NET Agent for the default domain, and still one .NET Agent for each of the applications in the application pool, as the following illustration shows:



For scalability reasons, some companies may assign multiple worker processes to a single application, in which case one .NET Agent instance is created for the default domain, and one for each of the worker processes associated with the application, as shown in the following illustration.

» **Note** This is the most commonly used configuration.



If there are multiple worker processes, and hence multiple .NET Agents associated with a single managed application, you can configure those agents as a Virtual Agent, which enables metrics from multiple physical .NET Agents to be aggregated. For more information, see *Virtual Agents* on page 73.

.NET Agent instance in the default domain

As described in .NET Agent instantiation on page 13, a .NET Agent is always created for the CLR's default domain. By default, the .NET Agent for the default domain does not connect to the Enterprise Manager, or appear as a node in the Investigator tree in Workstation or WebView. However, you can configure the .NET Agent for the default domain to connect to the Enterprise Manager if necessary. For more information, see the agent property in Default domain configuration on page 121.

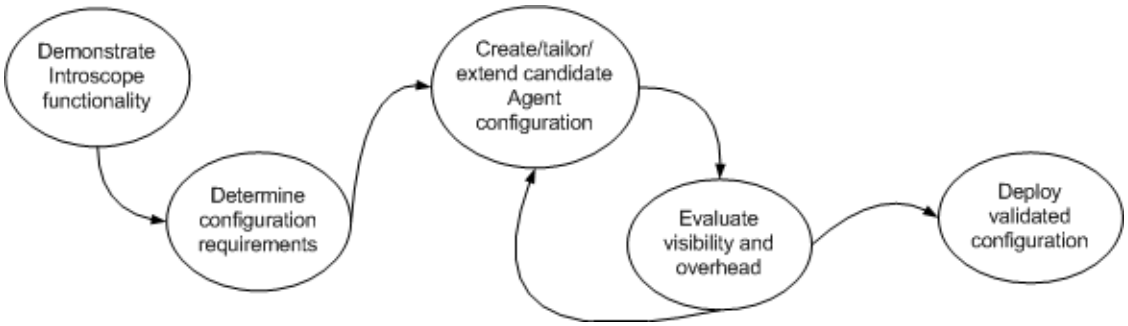
CHAPTER 2

.NET Agent Implementation

This chapter is an introduction to the .NET Agent implementation process.

The .NET Agent implementation overview

Developing the right .NET Agent configuration for your application and the environments in which it runs is an iterative process. The figure below illustrates the key processes in the .NET Agent implementation lifecycle; the sections that follow describe the steps in the implementation process.



Demonstrate Introscope functionality

The first step in developing an Introscope implementation involves "test driving" the default Introscope .NET Agent configuration. A default .NET Agent configuration demonstrates data collection functionality and is key to understanding and evaluating the out-of-the box features of the .NET Agent and Introscope as a whole. When you install Introscope, a default .NET Agent configuration is included.

The .NET Agent provides a variety of data collection options out-of-the box and can be customized to collect more environment-specific data. However, the more metrics a .NET Agent collects, the more system resources it consumes.

When evaluating the environment, the primary goal is to understand the depth and breadth of Introscope's data collection and application management features. As you refine your .NET Agent configuration, you will streamline data collection to balance the depth of data collection against overhead constraints and configure .NET Agent features that help manage and limit resource consumption.

Determine configuration requirements

Before introducing Introscope into your environment, whether pre-production or live, you should determine your data collection requirements. This information will help you tailor the data collection behaviors of the .NET Agent, and evaluate the impact on overhead through alternative configurations of the .NET Agent.

Since Introscope is employed across an application lifecycle—in development, test, and production—your monitoring goals, environmental constraints, and service level requirements will change over time. You will need to configure .NET Agents differently in each phase or environment.

.NET Agent configuration is a trade-off between visibility vs. overhead. The goal is to obtain optimal visibility at a reasonable cost.

In pre-production environments, such as development and QA, you typically configure a higher level of data collection to provide deeper visibility into the performance characteristics of the application.

In production or production-like environments, you reduce the level of metric reporting to control .NET Agent overhead, and when appropriate, implement optional configurations, such as Virtual Agents or agent failover.

If you intend to collect data from multiple environments, you will need to develop an appropriate .NET Agent configurations for each.

Define .NET Agent configuration

Having defined configuration requirements based on the characteristics of your application and its operating environment, you create a candidate .NET Agent configuration. Most agent behaviors are configured in the Introscope .NET Agent profile. Some features may also require some configuration in your application server, or other configuration steps.

Depending on the complexity of your configuration and the target environment, you may choose to build up the .NET Agent configuration in stages, so that you can evaluate the impact of each add-on components in isolation.

Evaluate .NET Agent performance overhead

When evaluating a .NET Agent configuration, verify that the metrics collected provide sufficient visibility into application performance and availability, and that the volume of metrics do not impose an unacceptable load on the operating environment. The .NET Agent should not report more metrics than are necessary to identify and localize performance and availability problems.

To effectively understand and evaluate .NET Agent overhead, you must understand the performance characteristics of the application prior to Introscope-enabling it.

For example, you can load test your application before and after implementing out-of-the-box monitoring to verify impact. Similarly, a conservative approach is to extend data collection in a controlled fashion and evaluate the impact of each add-on individually.

Validate and deploy .NET Agent configuration

After you have verified that a candidate .NET Agent configuration provides the visibility required for the target environment without imposing unacceptable overhead, you deploy the validated configuration to that environment.

In practice, the process of deploying a validated configuration includes installing the validated configuration artifacts—specifically the `IntroscopeAgent.profile` and modified or custom PBD files—to the target environment.

Implementing the .NET Agent

The following sections describe the implementation steps for data collection, and the .NET Agent directory structure and configuration artifacts.

Basic implementation

The .NET Agent implementation process is as follows:

Step 1 Install the .NET Agent.

Step 2 Configure user permissions for the .NET Agent installation directory and its subdirectories.

Step 3 Configure the .NET Agent properties that specify which applications should be monitored and the Enterprise Manager to which the .NET Agent reports.

.NET Agent configuration options

The following is an overview of configurable .NET Agent behaviors.

Communications with Enterprise Manager

You must explicitly configure the location of the Enterprise Manager to which the .NET Agent reports. If you do not, by default the .NET Agent will try to connect with the Enterprise Manager on localhost port 5001. If the .NET Agent will connect to an Enterprise Manager cluster, you must configure it to connect to a Collector Enterprise Manager, rather than to the Manager of Managers (MOM). To enable an .NET Agent to failover to a secondary Enterprise Manager, you must define connection properties and connection order as well. For more information, see Connecting to the Enterprise Manager on page 33.

.NET Agent naming

By default, .NET Agents are automatically assigned a name, based on the application's context path or domain name. Wily recommends using this default autonaming capability. If your requirements dictate, you can explicitly assign .NET Agent names. For more information, see .NET Agent name options on page 35.

Virtual Agents

If you have multiple .NET Agents that monitor a single application that is allocated to multiple worker process, as described in .NET Agent instantiation on page 13, configuring those .NET Agent as a Virtual Agent allows you to aggregate metrics at the application level. For more information, see Virtual Agents on page 73.

Logging

By default, the .NET Agent writes information log messages to console windows and log files. You can configure the .NET Agent for more detailed logging. For more information, see Configuring .NET Agent logging options on page 69.

Introscope domains and permissions

Unless you assign an .NET Agent to a custom Introscope Domain, it is part of the SuperDomain by default. For information about Domains and their use in configuring user permissions, see the *Introscope Configuration and Administration Guide*.

ProbeBuilder Directives (PBDs)

The PBDs used during the ProbeBuilding process determine the metrics that the .NET Agent reports. You configure a list of the desired PBDs or PBLs in the .NET Agent profile. The default configuration specified in the profile results in the "full" level of ProbeBuilding for the CLR, your applications, including Web Services, and the SQL Agent. The "full" PBDs are appropriate in development or QA environments.

In production environments where overhead is a concern, configure the less exhaustive "typical" level of ProbeBuilding to obtain fewer metrics and reduce overhead. You can further control the ProbeBuilding process by customizing `.pbd`s to skip classes or packages, or to instrument custom classes and methods that the default `.pbd`s do not specify. For more information, see ProbeBuilder Directives on page 47.

Data collection and reporting

Most optional data collection behaviors are controlled by agent properties:

- **Socket metrics**—By default, the .NET Agent reports input and output bandwidth rate metrics for individual sockets. For more information, see Turning off socket metrics in the .NET Agent profile on page 69.
- **URL Groups for Blame Reporting**—To control the way that metrics for front-ends are aggregated and presented in the Investigator in WebView and the Workstation, you must configure URL groups. The agent profile contains properties for specifying URL groups. For more information, see Using URL Groups on page 84.
- **Stall Event Reporting**—By default, the agent reports stalls as Events, and stores them in the Transaction Event Database. You can disable this behavior, or tailor the stall reporting behavior. For more information, see Disable the capture of stalls as events on page 98.
- **Transaction Tracing Behavior**—You can tailor the behavior of the automatic transaction tracing the Agent performs, and configure the collection of User IDs for Servlet and JSP invocations. For more information, see Transaction Tracer Options on page 93.
- **PerfMon metrics**—By default, the .NET Agent is configured to obtain PerfMon metrics from the Windows system. You can tailor the configuration to obtain more, fewer, or custom PerfMon metrics. You can also filter the PerfMon metrics by Agent. For more information, see Performance monitoring (PerfMon) metric collection on page 36
- **SQL Agent**—The Introscope SQL Agent is installed automatically with the .NET Agent installation. This agent extension provides visibility into the performance of individual SQL statements running under SQL Server. By default, AutoProbe

will instruments your database access components. To disable the SQL Agent, remove the SQL Agent `.pbd` from the agent property that specifies `.pbd`s to use in the ProbeBuilding process. For more information, see Configure the Introscope SQL Agent on page 99.

CHAPTER 3

Installing the .NET Agent

This chapter contains instructions for installing a .NET Agent. For information about installing the Introscope Enterprise Manager, Introscope Workstation, and the Introscope WebView application, see the *Introscope Configuration and Administration Guide*.

Before you start

This section lists key information, decisions, and resources you should identify or obtain before you start installing your .NET Agent.

Agent implementation planning

If you are new to Introscope or the .NET Agent, review the information in the .NET Agent Implementation on page 17.

Software requirements

The .NET Agent supports the following versions of .NET:

- .NET 1.1 Framework, v1.1.4322
- .NET 2.0 Framework, v2.0.50727
- .NET 3.0 Framework, v3.0.4506
- .NET 3.5 Framework, v3.5.21022

The .NET Agent supports the following versions of Windows and IIS:

- Windows XP Professional 2002 – SP2 / IIS Version 5.1
- Windows 2000 5.000.2195 – SP4 / IIS Version 5.00.2195.6620
- Windows Server 2003 Enterprise Edition – SP1 / IIS Version 6.0
- Windows Server 2003 Enterprise Edition – SP2 / IIS Version 6.0
- Windows 2008 Server Enterprise SP1 32-bit / IIS Version 7.0.6000.16386
- Windows 2008 Server Enterprise SP1 64-bit / IIS Version 7.0

The .NET Agent supports the following versions of SQL Server:

- SQL Server 2000 Version 8.00.2039 SP4
- SQL Server 2005 Version 9.00.1399.06

» **Important** To install the .NET Agent on any of the above platforms, sufficient disk space is required. CA Wily recommends disk space equal to three times the size of the installer `.exe` file.

.NET Agent support of x64

The .NET Agent supports 64-bit Windows DLL files under x64 (also known as AMD64 or Intel64). As x64 is backward compatible with x86, in Windows Server 2003 Enterprise Edition – SP2, x86 binaries run as expected.

If you intend to monitor 32-bit applications running under 64-bit environment, you must use the 32-bit .NET Agent DLL files. If you intend to monitor 64-bit applications, you must install the new 64-bit DLL files.

» **Note** Managed code must be compiled under .NET Framework 2.0 or later. .NET Framework 1.1.4 does not support the 64-bit platform.

Previous agents

If a previous version of the .NET Agent has been installed on a machine in your environment, CA Wily recommends the previous version of the agent be uninstalled before you install a new version of the .NET Agent.

» **Note** If you wish to use an older version of the .NET Agent rather than the current version, you **must** uninstall the current version of the agent first before installing the older version.

Enterprise Manager connection information

The .NET Agent runs on the same system as the applications you wish to monitor, and connects to the Introscope Enterprise Manager, which runs on a separate system. During the .NET Agent configuration process, you specify the location of the Enterprise Manager host system. In preparation, determine the hostname of the Enterprise Manager.

» **Note** When viewing metrics in the Introscope Investigator, the .NET Agent reports the host name only in lower case, regardless of the host name case. For example:

- If the host/machine name is HOST_1X2Y3Z (all upper case), the .NET Agent converts it to lower case and reports it as host_1x2y3z in the Investigator.
- If the host/machine name is Host_1X2y3Z (mixed case), the .NET Agent converts it to lower case and reports it as host_1x2y3z in the Investigator.

If your .NET Agent reports to a clustered Enterprise Manager, you must configure it to connect to a *Collector* Enterprise Manager, rather than the Manager of Managers (MOM) Enterprise Manager. If you have a cluster of Enterprise Managers, determine the hostname of the collector.

If you have multiple Enterprise Managers, clustered or not, you can configure your .NET Agent to failover to an alternate Enterprise Manager if it disconnects from its primary Enterprise Manager. If you plan to configure .NET Agent failover, determine the hostname of each Enterprise Manager that is a failover target for the .NET Agent.

Verify IIS application operations

Before installing the .NET Agent, make sure that your application runs and behaves as expected, and verify your .NET application can run as a worker process.

The .NET Agent monitors .NET applications that IIS runs in a worker process. To verify that IIS is using a worker process, load a page from the application and then look in the task manager for `aspnet_wp.exe` (for IIS 5.0, 5.1 or 6.0 running in 5.0 compatibility mode) or `w3wp.exe` (for IIS 6.0). If you do not see the worker process, enable IIS to handle managed components using this command:

```
C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\aspnet_regiis.exe -i
```

» **Note** In Windows 2000 environments, the default Windows directory is C:\WINNT. The above command when used in a Windows 2000 environment would be:

```
C:\WINNT\Microsoft.NET\Framework\v1.1.4322\aspnet_regiis.exe -i
```

Reducing startup time

The .NET Agent version 8.0 includes performance enhancements that reduce the startup time of .NET applications by 25-50%. To take advantage of these enhancements, the following configurations are required:

- Do not turn on Introscope LeakHunter.
- Do not use the `webservices.pbd`.
- Do not have any active `IdentifyAllClassesAs` directives in any custom PBD files.

Installing the .NET Agent

The .NET Agent installer can be run interactively (in GUI mode) or in silent mode. GUI mode is described in the following section. For information on silent mode, see *Installing the .NET Agent in silent mode*, below.

Installing the .NET Agent in GUI mode

The .NET Agent can install an x64 or x86 agent, but not both. Refer to the table below to see the instances when the .NET Agent will install an x64 agent or an x86 agent.

If your OS is...	And this Framework is present...	The installer will:
x86	x86 only	install an x86 agent. No other choice is available.
x64	x86 only	install an x86 agent. No other choice is available.
x64	x86 and x64	offer a choice to install either an x86 agent or x64 agent.
x64	x64 only	install an x64 agent. No other choice is available.

To start the .NET Agent installer in GUI mode:

1. Double-click the `introscope8.0windowsAgent_dotNET.exe` file.

   The Introduction page displays.

   » **Note** If you do not run one of the versions of .NET listed in Software requirements on page 26, the installer will not start.

2. Click **Next**.

   If your system is both x86 and x64 compatible, the installer will ask you to choose the mode here. Refer to the table above for more information. If you can only run one type of Framework, you go directly to the Select Installation Folder page.

   The Select Installation Folder page is displayed, and prompts you to accept the default installation directory, or select another directory. The installation directory is referred to through this guide as the *<Agent_Home>* directory.

3. Enter the desired directory path, or accept the default, and click **Install**.

   The .NET Agent is installed and the Installation Complete page is displayed. For information about the directory structure created by the installation process, see .NET Agent installation directories and files on page 31.

Once the .NET Agent has been installed, you must set the permissions for the `<Agent_Home>` directory. See User permissions for .NET Agent directory on page 37 for more information.

Installing the .NET Agent in silent mode

In silent mode, you invoke the .NET Agent installer from a command line and specify a response file that supplies installation selections to the installer. This eases the process of installing multiple .NET Agents.

You can use an automatically generated response file or edit a sample response file provided with your installation.

Using an automatically generated response file

When you run the .NET Agent installer in GUI mode, it creates a response file with the installation options you selected in the *<Agent_Home>*\install directory. You can use this response file for subsequent silent mode installations. The file name indicates the date and time that the installer created the response file:

```
autogenerated.responsefile.<year>.<month>.<day>.<hour>.<minutes>.<seconds>
```

For example, an installation done April 30, 2005 at 7:10:05 a.m. would have a response file with this name:

```
autogenerated.responsefile.2005.4.30.7.10.05
```

» **Note** The silent installer responsefile has a new property, `bitMode`. The default of this property is 32. The installer will honor an entry of 64 if it detects both an x64 OS and an x64 framework present on the system.

Using a manually configured sample response file

The `<Agent_Home>\wily\install` directory contains a sample response file you can edit for use in silent mode installation. The file name is:

```
SampleResponseFile.AgentForDotNet.txt
```

The response file can have any name, and can be located in any directory, as long as you specify the name and location in the command line when invoking silent mode.

Keep in mind these issues when entering values for the properties in the response file:

- Backslashes need to be "escaped"; for example: `C:\\myagent`
- Any directory supplied needs to be preceded by a slash and followed by a backslash. These backslashes must also be "escaped". For example:

  ```
  C:\\Wily1\\IntroscopeDir\\
  ```

» **Note** The silent installer responsefile has a new property, `bitMode`. The default of this property is 32. The installer will honor an entry of 64 if it detects both an x64 OS and an x64 framework present on the system.

Launching the silent mode installer

To run the installer in silent mode, specify the path to the installer and the absolute path to the response file:

```
<path to installer> -f <absolute path to responsefile>
```

For example:

```
D:\IntroscopeDotNet> introscope8.0windowsAgent_dotNET.exe -f /tmp/
  myResponseFile.txt
```

» **Note** If the response file specified does not exist or the path is invalid, the installer starts up in GUI mode.

Silent mode installer for your version of .NET

If you do not run one of the versions of .NET listed in Software requirements on page 26, add a line to your silent response file that specifies your version of .NET:

```
dotNet11Version=VersionNum
dotNet20Version=VersionNum
```

Where *VersionNum* is the version of your .NET Framework.

.NET Agent installation directories and files

An agent's operating and data collection behaviors are controlled by configuration properties stored in the agent profile, and the PBDs it references.

Once the .NET Agent is installed, the following directory structure is created in your *<Agent_Home>* directory:

```
install
UninstallerData
wily
  bin
  ext
  hotdeploy
  logs
```

Contents of the install directory

The `install` directory contains a log of the installation process, and files for use in performing silent mode installation of the .NET Agent and uninstalling the .NET Agent.

Contents of the UninstallerData directory

The `UninstallerData` directory contains an executable file and associated resources for uninstalling the .NET Agent.

This directory also contains the `wilyregtool.exe` executable utility. This custom utility is used to clean up Introscope .NET Agent related registry entries (registering and unregistering `.dll`s in the GAC). Use this tool instead of `gacutil`.

» **Note** There are two versions of the `wilyregtool`. 32-bit users get a version that is compiled with .NET 1.1. 64-bit users get a version that is compiled with .NET 2.0.

Contents of the wily directory

The `wily` directory contains:

- `IntroscopeAgent.profile`—The .NET Agent profile contains properties that control the behavior of the .NET Agent and AutoProbe. Defaults are supplied for many properties. Typically, each .NET Agent running in a CLR uses the same `IntroscopeAgent.profile` and a shared set of configurations, such as `.pbd`s and `.pbl`s.
- ProbeBuilder Directives (PBDs)—Contains the standard ProbeBuilder Directives (PBDs) provided with the .NET Agent, which vary depending on the application server your .NET Agent installation supports. AutoProbe looks for the PBDs you configure for use during the ProbeBuilding process.

  The following are the default PBD files contained in this directory:

  - `dotnet.pbd`
  - `errors.pbd`–used with CA Wily ErrorDetector
  - `nativeskip.pbd`
  - `sqlagent.pbd`
  - `toggles-full.pbd`
  - `toggles-typical.pbd`
  - `webservices.pbd`

  Each .NET Agent running in a CLR uses the same set of PBD`s` and PBL`s`. Each .NET Agent also outputs a distinct log file so differences in performance can be distinguished.
- ProbeBuilder Lists (PBLs)—Contains the `default-full.pbl` and `default-typical.pbl` files.
- `logging.config.xml`—Configures logging options.
- `Sample.exe.config`—A sample application configuration file, for optional configuration settings.

Contents of the wily\bin directory

This directory contains:

- `wily.Agent.dll`—Contains the core .NET Agent libraries.
- `wily.AutoProbe.dll`—Contains the AutoProbe bridge.
- `wily.AutoProbeConnector.dll`—Contains CLR profiler information.
- `wily.Agent.pdb`—Used to help debug the .NET Agent.
- `wily.AutoProbe.pdb`—Used to help debug AutoProbe.
- `wily.AutoProbeConnector.pdb`—Used to help debug AutoProbe Connector.

Contents of the wily\ext directory

This directory contains:

- `wily.LeakHunter.ext.dll`—This component appears when you install CA Wily LeakHunter.
- `wily.ProbeBuilder.ext.dll`—Contains the engine that instruments your applications.
- `wily.SQLAgent.ext.dll`—This component is the .NET SQL Agent, which enables reporting of SQL Server metrics.
- `wily.LeakHunter.ext.pdb`—Used to help debug LeakHunter.
- `wily.ProbeBuilder.ext.pdb`—Used to help debug ProbeBuilder.
- `wily.SQLAgent.ext.pdb`—Used to help debug the SQL Agent.

Contents of the wily\hotdeploy directory

ProbeBuilder Directives placed in this directory will be automatically deployed to the .NET Agent. When you create custom PBDs, save them to this directory. When PBDs are placed in this directory, you do not have to edit the `IntroscopeAgent.profile` to pick up new or changed PBDs.

For more information about creating custom PBDs, see ProbeBuilder Directives on page 47 and Creating custom tracers on page 53.

» **Note** Any ProbeBuilder Lists (PBLs) placed in this directory will be ignored by the .NET Agent.

The `hotdeploy` directory allows Introscope administrators to deploy new directives more quickly and easily, without editing the `IntroscopeAgent.profile`, and potentially without restarting applications. This ability heightens the need for caution. If your custom PBDs contain invalid syntax, or are configured to collect too many metrics, the impact will be felt more quickly. Invalid PBDs will cause AutoProbe to shut off and PBDs that collect too many metrics can affect application performance.

To address this, CA Wily recommends:

- testing and validating all directives in QA and performance environments before pushing them out to production environments.
- ensuring that your server environment's change control process is updated to reflect the new option for deploying PBDs.

Additionally, you can decide not to use the hotdeploy directory.

To unconfigure the hotdeploy directory:

1. Move any of the custom PBDs stored in the `hotdeploy` directory to the main `<Agent_Home>\wily` directory.
2. Open the `IntroscopeAgent.profile`.
3. Remove hotdeploy from the `introscope.autoprobe.directivesFile` property.
4. Add the PBDs you want use to the `introscope.autoprobe.directivesFile`, for example:

   ```
   introscope.autoprobe.directivesFile=default-
     typical.pbl,custom1.pbd,custom2.pbd,custom3.pbd
   ```

5. Save the `IntroscopeAgent.profile` and restart the agent.

Contents of the wily\logs directory

.NET Agent log files are stored here.

Installing agent extensions

Periodically, the .NET Agent installer does not install extensions (such as DLLs) correctly if the version number of the extension differs from the version number of the .NET Agent. To avoid this situation, you can:

- configure individual applications to accept different version numbers.

OR

- configure globally all applications to accept different version numbers.

» **Important** Choose only **one** option; do not perform both options.

To configure individual applications to avoid extension installation errors:

1. If the extension you are installing is a regular `.exe`, create a file called `<Extension_Name>.exe.config` and place this file in the same directory as the original `.exe`.
2. If the extension is an IIS application, create a file called `w3wp.exe.config` and place it in the same directory as `w3wp.exe`. This is for the default domain.
3. If the extension is an IIS application, also add the following to `web.config` for each individual application:

```
assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <dependentAssembly>
    <assemblyIdentity name="..." .../>
    <bindingRedirect oldVersion="0.0.0.0 - 65535.65535.65535.65535"
      newVersion="<AGENT.VERSION.NUMBER>"/>
  </dependentAssembly>
</assemblyBinding>
```

Enter an `assemblyIdentity name` and replace the agent version number with your .NET Agent version number. For example, if you installed the **7.2** version of the .NET Agent, you would add the following:

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <dependentAssembly>
    <assemblyIdentity name="wily.Agent"
      publicKeyToken="2B41FDFB6CD662A5"/>
    <bindingRedirect oldVersion="0.0.0.0 - 65535.65535.65535.65535"
      newVersion="7.2.0.1" />
  </dependentAssembly>
</assemblyBinding>
```

» **Note** If the files above already exist, add the `<assemblyBinding>` node under `<runtime>`.

To configure all applications globally:

- Add the following to `machine.config`:

```
assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <dependentAssembly>
    <assemblyIdentity name="..." .../>
    <bindingRedirect oldVersion="0.0.0.0 - 65535.65535.65535.65535"
      newVersion="<AGENT.VERSION.NUMBER>"/>
  </dependentAssembly>
</assemblyBinding>
```

Enter an `assemblyIdentity name` and replace the agent version number with your .NET Agent version number. For example, if you installed the **7.2** version of the .NET Agent, you would add the following:

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <dependentAssembly>
```

```
      <assemblyIdentity name="wily.Agent"
        publicKeyToken="2B41FDFB6CD662A5"/>
      <bindingRedirect oldVersion="0.0.0.0 - 65535.65535.65535.65535"
        newVersion="7.2.0.1" />
    </dependentAssembly>
</assemblyBinding>
```

» **Note** Adding the code snippet to `machine.config` globally affects all applications.

User permissions for .NET Agent directory

When you install the .NET Agent, the installation directory (typically `<Agent_Home>\install`) is created with permissions in accordance with the system policy.

For the .NET Agent and AutoProbe to run, the user authorization token executing the worker process must have permissions for the .NET Agent home directory, the `bin`, `ext`, and `log` subdirectories.

The wilypermissions utility

When you install the .NET Agent, the installer automatically checks for x86 Framework 2.0 or later. If the installer detects this Framework, it installs the `wilypermissions.exe` file, a utility provided by CA Wily to allow you to grant the user performing the installation permissions to the `<Agent_Home>` directory, and access to performance monitoring (PerfMon) counters.

If no user is specified, the utility will detect the default IIS user for the application platform and grant permissions to that user.

To use the wilypermissions utility:

1. Navigate to `<Agent_Home>\wily`.
2. Run `wilypermissions.exe` from command line as:

   `wilypermissions.exe path_to_wily_dir [process name].`

   » Note The process name must include the file extension. If the process name is not specified, the IIS worker process name will be used by default.

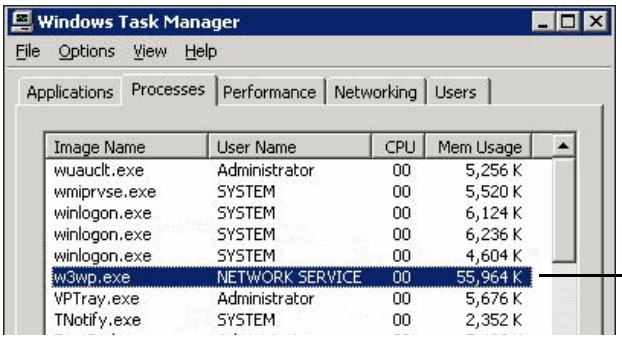
The utility executes, granting permission to the `<Agent_Home>` directory to the user performing the installation, and giving access to performance monitoring (PerfMon) counters.

If the installer does not detect an x86 Framework 2.0 or later, the utility file is not installed and the permission for the `<Agent_Home>` directory is set to `Everyone`—full control is given to `Everyone`. To change the permissions for this directory, see Determine the user running the application on page 37, and Verify minimum user permissions on page 38 for more information.

Determine the user running the application

Typically applications to be monitored run in an IIS worker process. To determine the user name for the process, open the Windows Task Manager, select the Processes tab, and locate the entry for `w3wp.exe` (on IIS 6) or `aspnet_wp.exe` (on IIS 5.)

Typically the user for the IIS worker process is "NETWORK SERVICE", as shown below:



» **Note** The application must be running to appear in the Windows Task Manager. If you do not see the application in the Windows Task Manager, close the Task Manager, access the application, then open the Task Manager again.

Verify minimum user permissions

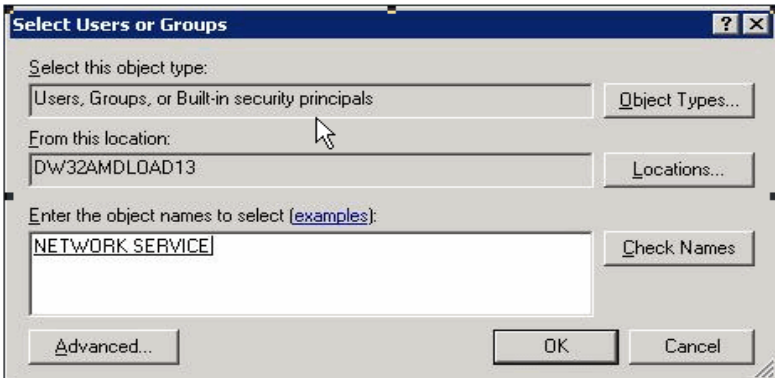
To verify minimum user permissions:

1. In Windows Explorer, right-click on the root .NET Agent folder and select **Properties**.

   The Properties window for the folder is displayed.

2. Click on the **Security tab** from the Properties window.
3. Verify that the **Administrator** and **Create Owner** user names have **Special Permissions**.
4. Click the **Add** button.

   The Select Users or Groups dialogue box is displayed.



5. Enter part of the name or the complete name of the user in the **Enter the object names to select** field.

6 Click the **Check Names** button to validate the user. If the user is not validated and underlined, check for spelling errors or spaces in the user name. If multiple users are listed, select the correct user. Click **OK** after the user name is validated and underlined.

   The Permission Entry dialogue box is displayed.

7 Verify that the user has the following permissions:

   - Full Control
   - Modify
   - Read & Execute
   - List Folder Contents
   - Read
   - Write

8 Click the **Advanced** button.

   The Advanced Security Settings for Wily dialogue box is displayed.

9 Select the **Replace permission entries on all child objects with entries shown here that apply to child Objects** check box to propagate permissions to the child directories, and click **OK**.

Connecting to the Enterprise Manager

The default communications settings in the .NET Agent profile enables a .NET Agent to connect to a local Enterprise Manager.

In a recommended Introscope implementation, an agent and the Enterprise Manager to which it reports do not reside on the same system—so you must identify the remote Enterprise Manager in the `IntroscopeAgent.profile`.

The properties in the `IntroscopeAgent.profile` specify connection behavior for the DEFAULT Enterprise Manager communication channel:

- `introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT`
  Defines the host name or IP address of the target Enterprise Manager.
- `introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT`
  Defines the Enterprise Manager's listen port, which should be the same port specified in the Enterprise Manager's `introscope.enterprisemanager.port.DEFAULT` property.
- `introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT`
  Defines the socket factory used for connections to the Enterprise Manager.

To specify an alternative Enterprise Manager channel, define its connection properties by adding the following properties to the agent profile:

- `introscope.agent.enterprisemanager.transport.tcp.host.`*NAME*
- `introscope.agent.enterprisemanager.transport.tcp.port.`*NAME*
- `Introscope.agent.enterprisemanager.transport.tcp.socketfactory.`*NAME*

Replace *NAME* with an identifier for the new Enterprise Manager channel, and specify the Enterprise Manager's connection details.

For more information about connection properties, see .NET Agent to Enterprise Manager connection on page 112.

Configuring instrumentation

As your environment becomes more complex, you may want to prevent specific processes or applications from being instrumented by Introscope. By default, all IIS 5 and IIS 6 ASP.NET applications are instrumented, as well as all application pools in IIS. Stand alone applications (non-IIS applications) are not instrumented by default. You must configure the .NET Agent to instrument non-IIS applications.

When you disable instrumentation for an application, the CLR profiler for that application is still active. For applications that are no longer instrumented:

- AutoProbe is turned off. This shuts down all automatic instrumentation.
- The .NET Agent associated with the application never connects to the Enterprise Manager and no metrics are reported. The overhead of a socket connection is also prevented.

For applications that have instrumentation enabled, the .NET Agent reports metrics as usual and AutoProbe is active.

Instrumenting specific processes and applications

If you monitor only applications running in IIS, you do not have to configure the following property. If the applications to be instrumented are only running in IIS, see Instrumenting application pools on page 41 to configure instrumentation of application pools.

To enable or disable instrumentation of processes or applications:

1. Stop IIS.
2. Open the `IntroscopeAgent.profile` and locate the *Restricted Instrumentation* section.
3. Add application names or fully qualified paths to the following property:

```
introscope.agent.dotnet.monitorApplications=
```

By default, `w3wp.exe` and `aspnet_wp.exe` are already listed in this property. Add other applications in a comma delineated list. For example:

```
introscope.agent.dotnet.monitorApplications=w3wp.exe,aspnet_wp.exe,
  RandomApp.exe,testapp.exe,readloop.exe,C:\windows\winmerge.exe,
  S:\sw\prvciew.exe
```

» **Important** The property list is case sensative. Relative paths and wildcards are not supported.

4 Restart IIS.

Instrumenting application pools

By default, all application pools are instrumented. However, if you want to limit which application pools are instrumented, there by limiting which IIS applications are instrumented, you must specify which application pools to instrument.

To instrument specific application pools:

1 Stop IIS.

2 Open the `IntroscopeAgent.profile` and locate the *Restricted Instrumentation* section.

3 Uncomment the following property:

```
introscope.agent.dotnet.monitorAppPools=
```

4 Add the application pools to be instrumented in a comma deliniated list. For example:

```
introscope.agent.dotnet.monitorAppPools="NULL","DefaultAppPool",
  "AppPool1","AppPool2"
```

» **Note** IIS 5 can run without application pools. Use the "Null" value if you are instrumenting applications running in IIS 5.

5 Restart IIS.

.NET Agent name options

There are two methods for naming .NET Agents: automatic and manual.

A .NET Agent obtains its name automatically, by default. An agent monitoring an ASP.NET application obtains its name based on the name of the virtual directory (or context path) of the application. An agent monitoring non-web-based applications determines its name from the application domain name.

No configuration is required to enable automatic agent naming.

You can explicitly assign a name to an agent, if you prefer not to use autonaming. The following sections have instructions for manual agent naming.

Create profiles for each application

If you wish to explicitly assign a name to an agent monitoring ASP.NET applications, you must create a separate `IntroscopeAgent.profile` for each application. This process is more labor-intensive, but provides greater control over the agent name. Using a separate profile for each agent also allows you to control the agent's process name.

Define an agent name

Specify the agent naming properties in `IntroscopeAgent.profile`:

```
introscope.agent.agentAutoNamingEnabled=false
introscope.agent.customProcessName=<CustomProcessName>
introscope.agent.agentName=<CustomAgentName>
```

Performance monitoring (PerfMon) metric collection

By default, the .NET Agent is configured to obtain system performance metrics (performance monitoring or PerfMon) from a Windows system. The property that enables performance monitoring is `introscope.agent.perfmon.enable`, which is set to `true` in the `IntroscopeAgent.profile` file.

IIS permissions

To view PerfMon metrics in the Introscope Investigator, the user running IIS must first have the correct permissions set.

» Important This configuration was performed for you automatically if you used the wilypermissions utility. See The wilypermissions utility on page 37 for more information.

To set IIS user permissions on Windows 2003 and XP:

1. Navigate to **Start > Settings > Control Panel > Administrative Tools > Local Security Policy > Local Policies > User Rights Assignment**.
2. Right click **Profile Single Process**, select **Properties**, and add the user running IIS to the list of permissioned users.
3. Right click **Profile System Performance**, select **Properties**, and add the user running IIS to the list of permissioned users.
4. Close the window.

To set IIS user permissions on Windows 2000:

- To enable IIS users on Windows 2000, see the following Microsoft knowledge base article:

  http://support.microsoft.com/kb/555129

Tailoring PerfMon metric collection

A .NET Agent property, `perfmon.metric.filterPattern`, specifies the PerfMon keys the .NET Agent reads. The default setting is:

```
introscope.agent.perfmon.metric.filterPattern=|Processor|*|*,|.NET Data
  Provider*|*|*,|.NET CLR*|{osprocessname}|*,|.NET CLR
  Data|*|*,|Process|{osprocessname}|*,|ASP.NET|*
```

The filter follows the format `|Category|Instance|Counter` or `|Category|Counter` (if there is no instance) where:

- `Category` identifies a performance monitor category, such as Memory, Processor, or Process.
- `Instance` identifies a specific instance of the specified category. Some categories, such as Memory, do not have instances.
- `Counter` identifies metrics for the `Category|Instance` to be collected. For example, the .NET CLR Memory performance counter category has performance counters such as # Bytes in all heaps, Gen 0 heap size, # GC handles, and % time in GC.

You can tailor the PerfMon data the .NET Agent collects by modifying the value of the `introscope.agent.perfmon.metric.filterPattern` property. You can expand or decrease the data reported, and include custom PerfMon counters if you have defined them for your applications.

» **Note** At times Windows 2003 users may not be able to see .NET CLR PerfMon metrics, like .NET CLR Exceptions or .NET CLR Interop. To avoid this, add the user running the monitored .NET application to the Administrators group.

Some PerfMon metrics have been reserved for future implementation by Microsoft. These metrics are tagged "NotDisplayed" when seen in PerfMon. When these metrics are viewed in the Introscope Investigator, the place holder tag is displayed.

Filtering PerfMon metrics by .NET Agent and process

By default, the .NET Agent will report PerfMon data for all agent instances and processes.

You can limit the PerfMon data by CustomerProcessName and AgentName, using the `perfmon.agentExpression` property setting in the `IntroscopeAgent.profile`.

The filtering format is: `ProcessName|AgentName`

For example:

```
perfmon.agentExpression=*|RGSTR
```

restricts the PerfMon statistics to agents whose AgentName is `RGSTR`.

To reduce the overhead of PerfMon reporting, you can limit the metric volume and the frequency of reporting with the following properties:

- `perfmon.metric.limit`—Sets an upper limit to the number of PerfMon metrics reported.
- `perfmon.metric.pollIntervalInMinutes`—Frequency with which the .NET Agent obtains metric values.
- `perfmon.metric.browseIntervalInMinutes`—Frequency with which the .NET Agent checks for new categories of counters.

In the Windows 2000 environments, if a user tries to access metrics they do not have permissions to view, the PerfMon metrics may not display correctly in the Introscope Investigator. To circumvent this problem, either add the ASPNET user to the MSSQLSERVER2005 group to correctly view PerfMon metrics or limit the data to be displayed using the above properties.

.NET Agent profile location

By default the .NET Agent profile, `IntroscopeAgent.profile`, is installed in the .NET Agent home directory, typically `<Agent_Home>/wily`. If you move the profile, you must update the environment variable that specifies the location of the .NET Agent profile.

To update the location of the .NET Agent profile location:

1. Navigate to the Windows System Properties page: **Start > Settings > Control Panel > System**.
2. Navigate to the **Advanced** tab and click the **Environment Variables** button.
3. Under the **System Variables** portion of the Environment Variables dialogue box, click **New**.
4. In the **New System Variable** dialogue box:
   - Enter the Variable Name: `com.wily.introscope.agentProfile`
   - Enter the Variable Value as the full path to the .NET Agent profile, including the .NET Agent profile name. For example:
     `c:\<Agent_Home>\IntroscopeAgent.profile`

Click **OK**.

5 Restart the Internet Information Services (IIS).

» **Note** If the IIS server was running before the .NET Agent installation, the IIS server must be restarted through the IIS Administration Service.

6 Start the Enterprise Manager and the Workstation if they are not already running.

Uninstalling the .NET Agent

The .NET Agent is active whenever an instrumented application is running. To delete or modify any agent DLL files, such as during an unistall, all instrumented applications must be stopped.

To uninstall the Introscope .NET Agent:

1 Restart the IIS service.

2 Double-click `Uninstall Introscope Agent for .NET.exe` in the `<Agent_Home>\UninstallerData\dotnet` directory.

The uninstall dialogue box is displayed.

3 Click **Next**.

A confirmation dialogue box is displayed.

4 Click **Continue** to uninstall the .NET Agent.

The .NET Agent DLLs are unregistered, related environment variables are removed, and installed files that have not been modified since installation are removed.

5 Reboot your system to complete the uninstall process.

» **Important** Rebooting your system is the only way to inform the IIS process that the system environment variables have changed. You must reboot your system to instruct IIS to turn off the .NET Agent.

CHAPTER 4

ProbeBuilder Directives

This chapter describes how to create and modify ProbeBuilder Directive files.

ProbeBuilder Directives overview

ProbeBuilder Directive (PBD) files tell the Introscope ProbeBuilder how to add probes, such as timers and counters, in order to instrument an application. PBD files govern what metrics your agents report to the Introscope Enterprise Manager.

» **Note** All metrics are calculated using the time set by your system clock. If the system clock is reset during a transaction, the elapsed time reported for that transaction may be misleading.

Introscope includes a set of default PBD files. You can also create custom Introscope PBD files to track any classes or methods to obtain specific information about your applications. See *Creating custom tracers* on page 53 and Creating advanced custom tracers on page 58.

There are two kinds of files used to specify ProbeBuilder Directives:

- **ProbeBuilder Directive (PBD) files**

  A ProbeBuilder Directive (PBD) file contains directives used by ProbeBuilder to instrument your applications. This determines which metrics the agents report to the Enterprise Manager.

- **ProbeBuilder List (PBL) files**

  A ProbeBuilder List (PBL) file contains a list of multiple PBD filenames. Different PBL files can refer to the same PBD files.

Components traced by default PBDs

The default Introscope PBD files implement tracing of the following .NET components:

- ADO.NET
- ASP.NET
- .NET Remoting
- .NET Directory Services
- .NET Messaging
- Network Sockets
- Enterprise Services
- Web Services
- SMTP Mail

Default ProbeBuilder Directive (PBD) files

The .NET Agent has the following default PBD files:

PBD File Name	Description
dotnet.pbd	This file provides directives to support the .NET Framework Class Libraries.
errors.pbd	This file configures ErrorDetector by specifying what code-level events constitute serious errors. By default, only front- and back-end errors are considered serious. That is, only errors that will be manifest as a user-facing error page or that indicate a problem with a backend system (ADO.NET, Messaging, etc.).
nativeskip.pbd	This file lists native Namespace skips. Skipping native Namespaces helps to avoids extra CPU overhead by not instrumenting certain classes. You can select Namespaces to instrument by commenting out lines. You can also add native skips to other Namespaces, classes, or assemblies
sqlagent.pbd	This is the SQL Agent configuration file. You use this file to instrument your ADO.NET vendor's library (.dll). In most cases you will not need to edit this file.
toggles-full.pbd	This file provides on/off switches in the form of "TurnOn" directives for the tracing provided in other directives files. Most tracer groups are turned on. For more information about turning tracers on or off, see Default tracer groups and toggles files on page 50 and Turning tracer groups on or off on page 51.
toggles-typical.pbd	This file provides on/off switches in the form of "TurnOn" directives for the tracing provided in other directives files. Only a small section of tracer groups are turned on. For more information about turning tracers on or off, see Default tracer groups and toggles files on page 50 and Turning tracer groups on or off on page 51.
webservices.pbd	This file provides directives to support .NET web services.

Default ProbeBuilder List (PBL) files

There are two sets of PBL files available with each agent:

PBL File Name	Description
default-full.pbl (default)	References PBD files in which most tracer groups are turned on. Introscope uses this set by default to demonstrate full Introscope functionality.
default-typical.pbl	A subset of tracer groups in the referenced PBD files are turned on. The typical set includes common settings, and is the set you can customize for a particular environment.

Tracer groups are found in PBD files, and referred to in PBL files. They cause the reporting of information about a set of classes. In PBD files, tracer group information is referred to by the term "flag". For example, `TraceOneMethodIfFlagged` or `SetFlag` are defining tracer group information.

Default tracer groups and toggles files

A tracer group consists of a set of tracers that is applied to a set of classes. For example, there are tracer groups which report the response times and rates for all system messaging classes.

You can refine the gathering of metrics on your systems by turning on or off certain tracer groups. This affects overhead usage, either increasing or decreasing it, depending on how you configure the tracer groups.

Tracer groups are modified in the `toggles-full.pbd` and the `toggles-typical.pbd` files, which are referred to by the `default-full.pbl` and `default-typical.pbl` files. This table lists the default tracer groups and their default configurations:

Tracer group	Definition	Default Full Setting	Default Typical Setting
ASPNETTracing	ASP Tracing Configuration	On	On
SocketTracing	Network Configuration	On	Off
WebServicesProducerTracing	Web Services Configuration	On	On
WebServicesClientTracing	Web Services Configuration	On	On
ServicedComponentTracing	Enterprise Services Tracing	On	On
ContextUtilTracing	Transaction Utilities Tracing	On	Off
RemotingClientProxyTracing	Runtime Remoting Tracing	On	On

Tracer group	Definition	Default Full Setting	Default Typical Setting
RemotingWebServiceTracing	Runtime Remoting Tracing	On	Off
DirectoryServicesTracing	Directory Services Tracing	On	On
MessagingTracing	System Messaging Tracing	On	On
MessagingTransactionTracing	System Messaging Tracing	On	On
WebMailTracing	System Web Mail Tracing	On	Off
SQLAgentConnections	SQL Agent connection configuration	On	On
SQLAgentCommands	SQL Agent command configuration	On	On
SQLAgentDataReaders	SQL Agent datareader configuration	On	On
SQLAgentTransactions	SQL Agent transaction configuration	On	On

Generally, the default `toggles` PBD files should not be edited. However, you can refine the gathering of metrics by turning on or off certain tracer groups. Tracer groups can be modified in the `toggles` files by:

- Turning on/off tracer groups to save on system overhead
- Adding classes to a tracer group

Tracer groups report information only when turned on (uncommented) and are activated with the keyword `TurnOn`.

Turning tracer groups on or off

You can refine the gathering of metrics on your systems by turning on or off certain tracer groups.

To turn tracer groups on or off:

1. Locate and open the `toggles-full.pbd` or `toggles-typical.pbd` file (depending on which file type is in use as defined by: `default-full.pbl` or `default-typical.pbl`). These files are found within the *<Agent_Home>*`\wily` directory.
2. Locate the tracer group you wish to turn on or off.
3. To turn a tracer group on or off, comment or uncomment the line by adding or removing the pound sign (#) from the beginning of the line. The directives in the following example illustrate a turned on and a turned off tracer group:

```
TurnOn: SocketTracing
```

This tracer group is turned on. The line is uncommented.

```
#TurnOn: SocketTracing
```

This tracer group is turned off. The line is commented.

» **Note** Any uncommented (turned on) directive for a tracer group causes the tracer group to be used.

4 Save the `toggles-full.pbd` or `toggles-typical.pbd` file.

Adding classes to an existing tracer group

You can turn on tracing for a particular class by adding the class to an existing tracer group. To identify a class as being part of a tracer group, use one of the identify keywords.

For example, to add the class `System.EnterpriseServices.ServicedComponent` to the tracer group `ServicedComponentTracing`:

```
IdentifyClassAs:
  System.EnterpriseServices.ServicedComponent ServicedComponent
  Tracing
```

For a list of identify keywords, see Supplementary information on directives and tracers on page 63.

Applying ProbeBuilder Directives

When you are ready to implement a ProbeBuilder Directive file, add it to the PBD directory. AutoProbe looks for PBD files in the directory that contains the `IntroscopeAgent.profile` file (by default, this is the `<Agent_Home>\wily` directory), and resolves filenames relative to this directory. If you have moved the location of your `wily` directory, be sure to map the file path to the correct directory.

To apply PBDs using the hotdeploy directory:

- Copy custom or modified files (PBDs and PBLs) into the `<Agent_Home>\wily\hotdeploy` directory.

  For more information on the hotdeploy directory, see Contents of the wily\hotdeploy directory on page 27.

To apply PBDs without the hotdeploy directory:

1 Copy custom or modified files (PBDs and PBLs) into the `<Agent_Home>\wily` directory.

2 Update the `introscope.autoprobe.directivesFile` property, in the `IntroscopeAgent.profile` file to include the names of any new files, separated by commas.

In the following example, a directives file named `petstore.pbd` has been added:

```
introscope.autoprobe.directivesFile=default-full.pbl,petstore.pbd
```

» **Note** Do not remove any ProbeBuilder List files already in the property, just add your custom PBD to the end of the list.

3 Save the `IntroscopeAgent.profile`.

4 Restart the application (for ASP.NET, restart IIS)

Custom locations and permissions

In addition to using the `hotdeply` directory or the `wily` directory as explained above, you can place PBDs in a custom location of your choosing that is not in either of these directories.

If you place PBD files in a custom location, you must specify the location of the PBD files in the `IntrosocopeAgent.profile`. For example, if you placed the `leakhunter.pbd` in a custom location on the `C:` drive, you would update the `introscope.autoprobe.directivesFile` property in the following way:

```
introscope.autoprobe.directivesFile=default-
  full.pbl,C:\\sw\\leakhunter.pbd
```

When placing PBDs in a custom location, the user starting the IIS process should have appropriate permissions to that custom location (`C:\\sw` in the above example). If the user starting the IIS process does not have permissions to this location, an error message is reported in the default domain logs, and the PBDs in the custom location do not take effect.

» **Important** CA Wily highly recommends you place PBDs in the `hotdeploy` directory.

Creating custom tracers

You can further refine your metric collection by creating custom PBD files. Creating custom directives, by creating tracers to track application specific measurements, require the use of specific syntax and keywords.

To write custom tracers, you must define:

- The directive type (indicating generically how many class(es) or method(s) to trace)
- The specific class(es) or method(s) to trace

- The type of information to trace in the class(es) or method(s) (for example, a time, a rate, or a count)
- The fully-qualified metric name (including the resource path) under which to present this information

Once a custom PBD is created, Introscope treats it as if it was an out-of-the-box PBD. You can set alerts on the metrics created, save them to SmartStor, or use them in the creation of custom dashboards in the Introscope Workstation.

» **Note** Be sure to choose methods to trace carefully, as more methods traced means more overhead.

Common custom tracer example

A `BlamePointTracer` is the most commonly used tracer. This tracer generates five separate metrics for associated methods or classes:

- Average Response Time (ms)
- Concurrent Invocations
- Errors Per Interval
- Responses Per Interval
- Stall Count

The following is an example of a `BlamePointTracer`. A `BlamePointTracer` has been set for a method called “search” in class “petshop.catalog.Catalog”. “MSPetShop|Catalog|search” is the name of the node in the Introscope Investigator that the BlamePoint metrics will be displayed under:

```
TraceOneMethodOfClass: petshop.catalog.Catalog search BlamePointTracer
  "MSPetShop|Catalog|search"
```

Tracer syntax

In addition to simple keywords that associate tracers into groups or enable/disable groups, PBD files contain tracer definitions. For Introscope to recognize and process your tracers, you must use a specific syntax when constructing custom tracers. A tracer is composed of a directive and information about the method or class to trace, in the following format:

```
<directive>: [arguments]
```

where `[arguments]` is a list, and is directive-specific. Arguments used in trace directives include `<Tracer-Group>`, `<class>`, `<method>`, `<Tracer-name>`, and `<metric-name>`.

» **Note** Depending on the directive used, only a subset of these parameters are required.

Tracer arguments	Definition
`<directive>`	There are six main directives available for custom tracing: ■ **TraceOneMethodOfClass**—traces a specified method in the specified class. ■ **TraceAllMethodsOfClass**—traces all methods in the specified class. ■ **TraceOneMethodIfInherits**—traces one method in all direct subclasses or direct interface implementations of the specified class or interface. ■ **TraceAllMethodsIfInherits**—traces all methods in all direct subclasses or direct interface implementations of the specified class or interface. **Note:** Only concrete, implemented methods can be traced and report metric data while running. An abstract method specified in a custom tracer results in no metric data being reported. ■ **TraceOneMethodIfFlagged**—traces one method if the specified class is included in a tracer group that has been enabled with the `TurnOn` keyword. ■ **TraceAllMethodsIfFlagged**—traces all methods if the specified class is included in a tracer group that has been enabled with the `TurnOn` keyword.
`<Tracer-Group>`	The group to which the tracer is associated.
`<class>`	A fully qualified class or interface name to trace. Fully qualified classes include the full assembly name of the class as well as the name, for example: `[MyAssembly]com.mycompany.myassembly.MyClass` **Note:** The assembly name must be enclosed in [] brackets.

Tracer arguments	Definition
`<method>`	The method name (e.g. `MyMethod`) OR the full method signature with return type and parameters (e.g. `myMethod;[mscorlib]System.Void([mscorlib]System.Int32)`. For more information on method signatures, see Signature differentiation on page 58.)
`<Tracer-name>`	Specifies the tracer type to be used. For example, `BlamePointTracer`. See the *Tracer name* table below for descriptions of tracer names.
`<metric-name>`	Controls how the collected data is displayed in the Introscope Workstation. The following examples describe three ways to specify the name and location of a metric at different levels of the metrics tree. ■ **metric-name**—the metric appears immediately inside the agent node. ■ **resource:metric-name**—the metric appears inside one resource (folder) below the agent node. ■ **resource|sub-resource|sub-sub-resource:metric-name**—the metric appears more than one resource (folder) level deep below the agent node. Use pipe characters (|) to separate the resources.

This table describes tracer names and what they trace:

Tracer name	What it traces
`BlamePointTracer`	Provides a standard set of metrics including average response time, per interval counts, concurrency, stalls, and errors for a blamed component.
`ConcurrentInvocationCounter`	Reports the number of times a method has started but not yet finished. The result is reported under the metric name specified in the tracer, `<metric-name>`, in the Investigator tree. An example use of this tracer would be counting the number of simultaneous database queries.

Tracer name	What it traces
`DumpStackTraceTracer`	Dumps a stack trace to the instrumented application's standard error for methods to which it is applied. The exception stack trace thrown by the Dump Stack Tracer is not a true exception—it is a mechanism for printing the method stack trace. This feature is useful for determining callpaths to a method. » **WARNING** This feature imposes heavy system overhead. It is strongly recommended that this tracer only be used in a diagnostic context where a sharp increase in overhead is acceptable.
`MethodTimer`	Average method execution time in milliseconds and reports it under the metric name specified in the tracer, `<metric-name>`, in the metrics tree.
`PerIntervalCounter`	Number of invocations per interval. This interval will change based on the view period of the consumer of the data (for example, the View pane in the Investigator). It is reported under the metric name specified in the tracer, `<metric-name>`, in the Investigator tree.

Custom method tracer examples

The following are examples of method tracers. In the following example, quotes ("") are used around the metric names because there are spaces in the metric names.

Average tracer example

This tracer tracks the average execution time of the given method in milliseconds.

```
TraceOneMethodOfClass: petshop.catalog.Catalog search BlamedMethodTimer
  "MSPetShop|Catalog|search:Average Method Invocation Time (ms)"
```

Rate tracer example

This tracer counts the number of times the method is called per second, and reports this rate under the specified metric name.

```
TraceOneMethodOfClass: petshop.catalog.Catalog search
  BlamedMethodRateTracer "MSPetShop|Catalog|search:Method Invocations Per
  Second"
```

Per interval counter tracer example

This method tracer counts the number of times the method is called per interval, and reports the per-interval count under the specified metric name.

```
TraceOneMethodOfClass: petshop.catalog.Catalog search PerIntervalCounter
  "MSPetShop|Catalog|search:Method Invocations Per Interval"
```

The interval is determined by the monitoring logic in the Enterprise Manager, such as the Graph frequency.

The preview pane in the Introscope Investigator defaults to 15-second intervals.

Counter tracer example

This tracer counts the total number of times the method is called.

```
TraceOneMethodOfClass: petshop.cart.ShoppingCart placeOrder
  BlamedMethodTraceIncrementor "MSPetShop|ShoppingCart|placeOrder:Total
  Order Count"
```

Combined counter tracer example

These tracers combine incrementor and decrementor tracers to keep a running count.

```
TraceOneMethodOfClass: petshop.account.LoginComponent login
  MethodTraceIncrementor "MSPetShop|Account:Logged In Users"
TraceOneMethodOfClass: petshop.account.LogoutComponent logout
  MethodTraceDecrementor "MSPetShop |Account:Logged In Users"
```

Creating advanced custom tracers

The following sections detail creating advanced customs tracers, such as single-metric tracers, skips, and combined custom tracers.

Advanced single-metric tracers

Directives and tracers track methods, classes, and sets of classes. A single-metric tracer reports a specific metric for a specific method, which is the smallest unit that Introscope can track. Single-metric tracers can be created in several ways: through the method signature, by substituting keywords, or by manipulating the metric name parameters.

Signature differentiation

Tracers can be applied to a method based on the method signature.

To trace a single instance of a method with a specific signature, append the signature to the method name (including return type) specified using the internal method descriptor format.

For example, `myMethod;[mscorlib]System.Void([mscorlib]System.Int32)` traces the instance of the method with an `int` argument and a `void` return type.

Metric name keyword-based substitution

Keyword-based substitution allows runtime substitution of values into the metric name.

The parameters in the metric name in the tracer are substituted at runtime for the actual values into the metric name. This feature can be used with any directive.

Parameter	Runtime substitution
`{method}`	Name of the method being traced
`{classname}`	Runtime class name of the class being traced
`{namespacename}`	Runtime namespace name of the class being traced
`{namespaceandclassname}`	Runtime name space and class name of the class being traced
`{assemblyname}`	Name of the assembly being traced.
`{fullclassname}`	Reports the full class name including the assembly name.

» **Note** If Introscope processes a class which does not have a `namespace`, it will replace `{namespacename}` with the string "`<Unnamed Namespace>`".

Keyword-based substitution: Example 1

If the metric name for a tracer in the PBD file is:

```
"{namespacename}|{classname}|{method}:Response Time (ms)"
```

and a tracer is applied to the method `myMethod` with a runtime class of `myClass` that is in `myNamespace`, the resulting metric name would be:

```
"myNamespace|myClass|myMethod:Response Time (ms)"
```

Keyword-based substitution: Example 2

If a tracer with a metric name in the PBD file of

```
"{namespaceandclassname}|{method}:Response Time (ms)"
```

was applied to the same method in example 1, the resulting metric name would be:

```
"myNamespace.myClass|myMethod:Response Time(ms)"
```

» **Note** In this example, a . (period) is used between the `namespace` and `class` instead of the | (pipe symbol) in the first example.

Metric-name-based parameters

You can create a single-method tracer that creates a metric name based on parameters passed to a method using the `TraceOneMethodWithParametersofClass` keyword, using this format:

```
TraceOneMethodWithParametersOfClass: <class> <method> <Tracer-name>
  <metric-name>
```

Parameters can be used in the metric name. This is accomplished by substituting the value of parameters for placeholder strings in the metric name. Use the following string as a place holder:"{#}", where # is the index of the parameter to substitute. The indices start counting at zero. Any number of parameter substitutions can be used in any order. All parameters are converted to strings before substitution into the metric name. Object parameters other than strings should be used with caution because they are converted using the `ToString()` method.

» **WARNING** If you are unclear about what string the parameter will be converted to, do not use it in the metric name.

Metric-name-based example

A Web site uses a class named `order`, with a method named `process`. The method has parameters for different kinds of orders, either `book` or `music`.

You can create a tracer like this:

```
TraceOneMethodWithParametersOfClass: order process;
  [mscorlib]System.Void([mscorlib]System.Int32) MethodTimer
  "Order|{0}Order:Average Response Time (ms)"
```

This tracer produces metrics like these:

```
Order
  BookOrder
    Average Response Time (ms)
  MusicOrder
    Average Response Time (ms)
```

You can also use the `TraceOneMethodWithParametersIfInherits` keyword. For more information on both keywords, see Supplementary information on directives and tracers on page 63.

Skip directives

Certain packages, classes, or methods can be skipped by AutoProbe or ProbeBuilder by using skip directives. Skip directives cause ProbeBuilder to skip over a namespace, class, or assembly.

When the .NET Agent is installed, a `nativeskip.pbd` file is installed in the *<Agent_Home>*`\wily` directory. This PBD file contains pre-defined classes to be skipped (not instrumented), improving agent startup time. You can modify which classes, namespaces, or assemblies are natively skipped by modifying the `nativeskip.pbd` file.

For a complete list of skip directives used with the .NET Agent, see the *Directive & Tracer Type Definitions* guide. See Supplementary information on directives and tracers on page 63 for more information about this guide.

Combining custom tracers

You can use multiple tracers that affect the same metric, in effect combining them. This is most commonly used with incrementors and decrementors.

This example creates a metric named `Logged-in Users`. With a class `user` and methods `login` and `logout`, create the following tracers:

```
TraceOneMethodOfClass user login MethodTraceIncrementor "Logged-in Users"
TraceOneMethodOfClass user logout MethodTraceDecrementor "Logged-in Users"
```

This increments the metric `Logged-in Users` when someone logs in, and decrements `Logged-in Users` when someone logs out.

Notes about specific tracers

The following identifier and tracer have actions specific to a .NET environment:

- **IdentifyAnnotatedClassAs**: <attribute-class-name> <Tracer-group>

  Associates all classes annotated with the specified attribute class to the specified tracer group.

  Some classes can be annotated with an attribute class to provide extra functionality to the class. In the example below, an attribute class called `System.EnterpriseServices.Transaction` is attached to the class `ServicedComponent`:

  ```
  [Transaction]
  Public class ServicedComponent {
  }
  ```

  In a PBD you would write the following:

```
IdentifyAnnotatedClassAs: System.EnterpriseServices.Transaction
  MyTracerGroup
```

This identifies all classes that have the `[Transaction]` annotation, including `ServicedComponent`.

» **Note** The Introscope .NET Agent does not trace inherited attributes when using this identifier, but does trace attributes applied to the base class.

- **TraceAnnotatedMethodsIfFlagged**: <Tracer-group> <attribute-class-name> <Tracer-name> <metric-name>

  Traces all methods which are annotated by the specified class for classes associated with the specified tracer group.

Explicit interface implementation

The .NET Agent uses explicit interface implementation in PBD files. If you are tracing a method of a class that has the identical name as another method used in another class, you must explicitly name the method and the interface to which it belongs. For example, if `InterfaceA` and `InterfaceB` both have a method named `MethodX`, when calling `MethodX` for `InterfaceA`, you must name both the interface and the method: `InterfaceA.MethodX`.

The following is an example of tracing a method of a class with an explicit interface implementation:

```
SetFlag: customInterfaceTracing
TurnOn: customInterfaceTracing
IdentifyInheritedAs: EdgeCaseInterface customInterfaceTracing
TraceOneMethodIfFlagged: customInterfaceTracing EdgeCaseInterface.method2
  BlamePointTracer "Interface|{namespaceandclassname}|{method}"
```

Instrumenting and inheritance

Introscope does not automatically instrument classes in the deeper levels of a class hierarchy. For example, assume a class hierarchy in which `ClassB` extends `ClassA`, and `ClassC` extends `ClassB`, like so:

```
Interface\ClassA
  ClassB
    ClassC
```

When you instrument `ClassA`, `ClassB` is also instrumented because it explicitly extends `ClassA`. However, Introscope does *not* instrument `ClassC` because `ClassC` does not explicitly extend `ClassA`. To instrument `ClassC` you must explicitly identify `ClassC`.

Supplementary information on directives and tracers

For a complete list of the tracers and directives used with the Introscope.NET Agent, see the *Directive & Tracer Type Definitions* guide, available on the Wily Technology Community site, here: *https://community.wilytech.com/kbclick.jspa?categoryID=414&externalID=1927*

To access the Wily Technology Community site, you first need to register for an account using your corporate email address, here: *https://community.wilytech.com/account!default.jspa*

Once you have completed the account information, CA Wily will contact you within 3-5 business days to confirm your registration. If you do not register with a corporate email address, your request for access will be denied.

SECTION II

Operation and Management

The chapters in this section have information to help you operate and manage your .NET Agents.

CHAPTER 5

Monitoring and Logging

This chapter details .NET Agent configuration options that relate to agent operations and monitoring.

Configuring .NET Agent connection metrics

By default, Introscope generates metrics on the connect/disconnect status of an agent connected to an Enterprise Manager. You can monitor .NET Agent connection metrics to determine the current state of the connection between a .NET Agent and the Enterprise Manager.

The .NET Agent connection metrics appear in the Introscope Investigator, under the custom metric host:

```
Custom Metric Host (Virtual) \ Custom Metric Process(Virtual) \ Custom
  Metric Agent (Virtual) (*SuperDomain*) \ Agents \ <HostName> \ <Agent
  Process Name> \ <Agent Name> \ ConnectionStatus
```

The `ConnectionStatus` metric can have these values:

- 0—No data is available
- 1—.NET Agent is connected
- 2—.NET Agent is slow to report
- 3—.NET Agent is disconnected

A .NET Agent disconnect also generates a "What's Interesting" event. As with other events, users can query for .NET Agent disconnects using the historical query interface. .NET Agent disconnect events form a portion of the data used in assessing application health in the **Overview** tab for agents and applications in the Introscope Workstation Console.

Once an agent disconnects, Introscope continues to generate "not connected" state metrics until the agent is timed out, as specified by this property:

```
introscope.enterprisemanager.agentconnection.metrics.agentTimeoutInMinutes
```

After the .NET Agent has timed out, no additional connection metrics are generated. For information about Enterprise Manager properties, see the *Introscope Configuration and Administration Guide*.

Turning off socket metrics in the .NET Agent profile

Metrics that trace per-socket bandwidth have a potential for high overhead, and can be turned off if you notice that collecting network metrics is consuming a lot of processor or I/O time.

To turn off socket metrics:

1. Open the `IntroscopeAgent.profile`.
2. Under the Agent Socket Rate Metrics heading, enter a value of `false` for the following property. The default value is `true`.

   ```
   introscope.agent.sockets.reportRateMetrics=false
   ```
3. Save the `IntroscopeAgent.profile`.

Configuring .NET Agent logging options

The following section describes how to run the .NET Agent in verbose mode and set logfile details for the agent. The .NET Agent for Introscope uses Log4net functionality for these functions. If you want to use other Log4net functionality, see Log4net documentation at http://logging.apache.org/log4net/release/features.html.

Running the .NET Agent in verbose mode

Running the .NET Agent in verbose mode records many details to the log file, which is helpful in debugging.

To run the .NET Agent in verbose mode:

1. Stop the .NET Agent.
2. Open the `logging.config.xml` file.
3. Change the `level value` attribute to **`VERBOSE`**. The default is `INFO`.

   ```
   <root>
   <level value="VERBOSE" />
   <appender-ref ref="logfile" />
   <appender-ref ref="console" />
   </root>
   ```
4. Save the `logging.config.xml` file and restart the .NET Agent.

Changing the .NET Agent log file location

Log files are written to the *<Agent_Home>*\wily\logs directory by default, usually: C:\Program Files\CA Wily\Introscope*X.Y*\wily\logs, where *X.Y* is your version of Introscope. To facilitate ease of use, you may want to change the location of the .NET Agent logfile.

To change the location of the .NET Agent logfile:

1. Stop the .NET Agent.
2. Open the logging.config.xml file.
3. Change the **file value** attribute to the desired location of the log file, for example:

   ```
   <file value="c:\introscope_logs\IntroscopeAgent.log" />
   ```
4. Save the logging.config.xml file.
5. Restart the .NET Agent.

If you have configured a name for your agent, as described in *.NET Agent name options* on page 35, the named agent logs are written to the logs directory, either the default location or the new location you have just designated.

.NET Agent log files and automatic agent naming

By default, the .NET Agent obtains its name automatically. When the .NET Agent name is found automatically, the log files associated with that agent are named automatically using that same information.

The following examples show how the .NET Agent log files are named. The examples use an agent name of MyDomain//MyStuff (where MyDomain is the domain, and MyStuff is the instance). When a .NET Agent log file is created (by default, named AutoProbe.log), if the agent name is not yet available, a timestamp will be included in the filename:

```
AutoProbe20060928-175024.log
```

Once the .NET Agent name becomes available, the logfile will be renamed:

```
AutoProbeMyDomain_MyStuff.log
```

To disable automatic naming and specify individual names for .NET Agents, see *.NET Agent name options* on page 35.

When using .NET Agent log file automatic naming, if the original name of the agent logfile does not end in .log, a period is appended, and log is added. All characters that are not letters or digits will be replaced by underscores. Also, if advanced Log4Net functionality is used, the automatic naming capability might not work.

» **Note** If you have log files with the time stamp name as opposed to the actual name of the log, the process may have timed out before the .NET Agent name could be obtained.

Default domain logs

The default domain does not connect to an Enterprise Manager to report metrics, and does not run any applications itself. However, the .NET Agent that resides on the default domain still generates log files, as it handles all byte code instrumentation for all application domains hosted by the default domain. One of these files, `AutoProbe.DefaultDomain.log`, contains information about the byte code instrumentation that occurs in the default domain. As all byte code instrumentation occurs in the default domain, these log files contain critical information regarding the instrumentation.

The default domain also generates the `IntroscopeAgent.DefaultDomain.log` file for the .NET Agent.

Managing ProbeBuilder logs

Introscope ProbeBuilder logs the PBDs it used and the probes it added during the instrumentation process.

AutoProbe log name and location

By default, the AutoProbe log file is named `AutoProbe.log`. To change the name, change the `introscope.autoprobe.logfile` property in the .NET Agent profile.

The `AutoProbe.log` file is generated relative to the location of the `IntroscopeAgent.profile` file.

» **Note** When loading the .NET Agent profile from a resource on a classpath, AutoProbe will be unable to write to the AutoProbe log file, because the `IntroscopeAgent.profile` file is located within a resource.

CHAPTER 6

Virtual Agents

This chapter has information about configuring Virtual Agents.

Understanding Virtual Agents

You can configure multiple physical .NET Agents into a single *Virtual Agent.* A Virtual Agent enables an aggregated, logical view of the metrics reported by multiple .NET Agents.

A Virtual Agent is useful if you manage clustered applications with Introscope—a Virtual Agent is comprised of the agents that monitor different instances of the same clustered application. The Virtual Agent appears in the Introscope Workstation as a single agent. This allows metrics from multiple instances of a clustered application to be presented at a logical, application level, as opposed to separately for each application instance.

You can view performance and availability data for a specific application instance, by scoping your views and interactions in terms of a single Agent.

Virtual Agent requirements

If you have multiple stand-alone Enterprise Managers, a Virtual Agent can contain only agents that report to the same Enterprise Manager.

In an Enterprise Manager cluster, described in the *Introscope Configuration and Administration Guide*, agents that report to Enterprise Managers within a single cluster can belong to the same Virtual Agent, regardless of the Collector Enterprise Manager to which they report.

An agent can be assigned to multiple Virtual Agents.

Virtual Agents cannot include other Virtual Agents.

If you define multiple Virtual Agents, they must be uniquely named.

Configuring Virtual Agents

Configure Virtual Agents in the `agentclusters.xml` file, in the *<Introscope_Home>*`\config` directory of the Enterprise Manager to which the agents report. If you run multiple Enterprise Managers that are clustered, define Virtual Agents in the `agentclusters.xml` file in the `config` directory of the cluster's Manager of Managers (MOM).

The sample `agentclusters.xml` below defines a Virtual Agent named `BuyNowAppCluster`, in the Introscope SuperDomain. The Virtual Agent includes all agents, on any host, whose agent name starts with `BuyNow`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<agent-clusters xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="agentclusters0.1.xsd" version="0.1" >
  <agent-cluster name="BuyNowAppCluster" domain="SuperDomain" >
    <agent-specifier>.*\|.*\|BuyNow.*</agent-specifier>
    <metric-specifier>Frontends\|.*</metric-specifier>
  </agent-cluster>
</agent-clusters>
```

The root element, `<agent-clusters>`, is required. The `<agent-cluster>` element defines a Virtual Agent, and has two required attributes:

- `name`—If you define multiple Virtual Agents, each must have a unique name.
- `domain`—Assigns the Virtual Agent to an Introscope domain.

If you define multiple Virtual Agents, you define an `<agent-cluster>` element for each. The `<agent-cluster>` element requires two child elements:

- `<agent-specifier>`—Contains a regular expression that specifies the Agents in the Virtual Agent, using the standard fully qualified agent name:

  `<host> | <process> | <agentName>`

- `<metric-specifier>`—Contains a prefix that specifies the metrics to collect from the agents in the Virtual Agent, in terms of resource type, or subsets of the instances of a resource type. Recommended prefixes are:
  - Frontends
  - Backends
  - PerfMon
  - GC Heap
  - LeakHunter
  - ASP.NET
  - Thread Pool

» **Note** While the above are the recommended prefixes, any resource can be used as a metric specifier.

The `<agent-cluster>` element can contain multiple `<metric-specifier>` stanzas. Note that a higher volume of matching metrics imposes high overhead on the Enterprise Manager, and can ultimately have an effect on Enterprise Manager capacity.

» Important Regular expressions and wildcard metric specifiers such as ".*" and "(.*)" are allowed, but should be used with caution, if at all. Use of wildcards can result in a high volume of metrics and a performance impact.

A sample `agentclusters.xml` is available in your *<Introscope_Home>*`\config` directory.

CHAPTER 7

.NET Agent Failover

This chapter has information about .NET Agent failover.

Understand .NET Agent failover

An agent that cannot connect to its primary Enterprise Manager, or loses its connection with the Enterprise Manager, can failover to an alternative Enterprise Manager. To enable failover, you specify a list of alternative Enterprise Managers in the agent profile.

When a .NET Agent configured for failover cannot connect to its default Enterprise Manager, it tries to connect to the next Enterprise Manager on its list of failover hosts. If the agent does not connect with a failover host, it cycles through the Enterprise Managers on the list until it succeeds in connecting. If the .NET Agent goes through the list without connecting to an Enterprise Manager, it waits ten seconds before cycling through the list again.

In a basic Introscope configuration, you define the host and port settings for one Enterprise Manager. To enable agent failover, you define connection properties for one or more backup Enterprise Managers, and a list that specifies the failover order.

Define backup Enterprise Managers

For each backup Enterprise Manager, create an alternate communication channel, as described in *Connecting to the Enterprise Manager* on page 33. Assign each additional Enterprise Manager communication channel a unique name—do not use the name DEFAULT or the name of an existing channel when creating a new channel.

Given a primary Enterprise Manager and two backups, your agent profile will include property definitions similar to the following:

Default Enterprise Manager location and names

```
introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT=localhost
introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT=5001
introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT=
  com.wily.isengard.postofficehub.link.net.DefaultSocketFactory
```

Backup Enterprise Managers

```
introscope.agent.enterprisemanager.transport.tcp.host.BackupEM1=voyager
introscope.agent.enterprisemanager.transport.tcp.port.BackupEM1=5002
introscope.agent.enterprisemanager.transport.tcp.socketfactory.BackupEM1=
  com.wily.isengard.postofficehub.link.net.DefaultSocketFactory
introscope.agent.enterprisemanager.transport.tcp.host.BackupEM2=space9
introscope.agent.enterprisemanager.transport.tcp.port.BackupEM2=5003
introscope.agent.enterprisemanager.transport.tcp.socketfactory.BackupEM2=
  com.wily.isengard.postofficehub.link.net.DefaultSocketFactory
```

Define failover connection order

After specifying the connection properties for your backup Enterprise Managers, define the order in which the .NET Agent will attempt to failover when it loses connection to its primary Enterprise Manager

Use the property `introscope.agent.enterprisemanager.connectionorder`, described on *page 112*, to list the primary communication channels and backups for the Enterprise Manager. Put the primary Enterprise Manager first in the list. For the Enterprise Manager communication channels specified on the previous page, the connection order could be specified as follows:

```
introscope.agent.enterprisemanager.connectionorder=DEFAULT,BackupEM1,
  BackupEM2
```

Configure failback to primary Enterprise Manager

In the default .NET Agent failover scenario, if the .NET Agent loses the connection to its primary Enterprise Manager (the first one defined in the Enterprise Connection Order list), the agent tries to connect to the next Enterprise Manager defined in the agent profile. You can specify the interval by which the .NET Agent tries to reconnect to the primary Enterprise Manager.

To configure the .NET Agent failback interval to the primary Enterprise Manager:

1. In the agent profile, `IntroscopeAgent.profile`, locate the *Enterprise Manager Failback Retry Interval* section.
2. Specify the interval in seconds the .NET Agent should use to retry connecting to the Enterprise Manager. The default is set to 120 seconds (two minutes):

   ```
   introscope.agent.enterprisemanager.failbackRetryIntervalInSeconds=120
   ```
3. Save your changes.
4. Restart the application.

Failover and domain\user configuration

To use the agent failover feature and also have users, domains, and authentication settings defined, you must synchronize this information across the specified failover Enterprise Managers. For more information on domains and user permissions, see the *Introscope Configuration and Administration Guide*.

SECTION III

Tailoring and Extending Data Collection

The chapters in this section have information about tailoring and extending .NET Agent data collection.

CHAPTER 8

Configuring Boundary Blame

This chapter describes default .NET Agent blame reporting behaviors, and related configuration options.

Understanding Boundary Blame

Introscope's Blame Technology enables you to view metrics at the application tiers in managed .NET applications: the front and backends of your application. This capability, referred to as Boundary Blame, allows users to triage problems to the application frontend or backend.

This section describes configuration options for Introscope's Boundary Blame feature.

Introscope takes advantage of the SQL statement monitoring functionality of the SQL Agent to detect back-ends automatically. If the SQL Agent is unavailable, Introscope will automatically detect socket calls as backends, such as client/server databases, or LDAP servers accessed through a socket.

For information about how Boundary Blame is presented in the Introscope Investigator, see the *Introscope Workstation User Guide*.

Using URL Groups

URL Groups are named groupings of transactions that you define in terms of a URL path prefix. Introscope aggregates metrics for each URL Group and presents those metrics under the `Frontends|Apps|<ApplicationName>|URLs` node of the Introscope Investigator.

A path prefix is the portion of the URL that follows the hostname. For example, in this URL:

```
http://burger1.com/TestWar/burgerServlet?ViewItem&category=
   11776&item=5550662630&rd=1
```

the path prefix is:

```
/TestWar
```

You can define a URL Group for any useful category of requests that can be derived from a URL's path prefix. For example, depending on the form of your application URLs, you could define URL Groups for each customer your application supports, for each major application, or for sub-applications. This allows you to monitor performance in the context of committed service levels, or for mission-critical portions of your application.

By default, all URLs are assigned to a URL Group called `Default`. This prevents the overhead that would be imposed if invalid URLs, for instance those that result in a 404 error, do not create unique, one-time metrics. Configuring meaningful URL Groups allows users to monitor performance at the sub-application level.

URL Group properties

Define URL Groups in the `IntroscopeAgent.profile` file, using these properties

- `introscope.agent.urlgroup.`**`keys`**
- `introscope.agent.urlgroup.`**`group.default.pathprefix`**
- `introscope.agent.urlgroup.`**`group.default.format`**

Sample URL Groups

The listing below is an excerpt from an agent profile that illustrates how URL Groups are defined. The sections following the listing provide instructions for configuring the required properties.

```
introscope.agent.urlgroup.keys=alpha,beta,gamma
introscope.agent.urlgroup.group.alpha.pathprefix=/testWar
  introscope.agent.urlgroup.group.alpha.format=foo {host} bar {protocol}
  baz {port} quux {query_param:foo} red {path_substring:2:5} yellow
  {path_delimited:/:0:1} green{path_delimited:/:1:4} blue
  {path_substring:0:0}
introscope.agent.urlgroup.group.beta.pathprefix=/nofilterWar
  introscope.agent.urlgroup.group.beta.format=nofilter foo {host} bar
  {protocol} baz {port} quux{query_param:foo} red {path_substring:2:5}
  yellow{path_delimited:/:0:1} green {path_delimited:/:1:4}
  blue{path_substring:0:0}
introscope.agent.urlgroup.group.gamma.pathprefix=/examplesWebApp
  introscope.agent.urlgroup.group.gamma.format=Examples Web App
```

Defining URL Groups

The following sections provide information on the properties that configure URL Groups.

Define keys for URL Groups

Use the `introscope.agent.urlgroup.keys` property to define a list of the IDs, or keys, of all of your URL Groups. The key for a URL Group is referenced in other property definitions that declare an attribute of the URL group. This example defines the keys for three URL Groups:

```
introscope.agent.urlgroup.keys=alpha,beta,gamma
```

The order in which you specify keys for URL Groups is important. For more information, see *URL grouping* on page 133.

Define membership of each URL Group

Use the `introscope.agent.urlgroup.group.`*`GroupKey`*`.pathprefix` to define which requests fall within the URL Group, by specifying the pattern against which the path prefix of a URL is matched.

- **Example 1**

  This property definition assigns all requests in which the path portion of the URL starts with `/TestWar` to the URL Group whose key is `alpha`:

  ```
  introscope.agent.urlgroup.group.alpha.pathprefix=/TestWar
  ```

  Requests that match the specified `pathprefix` include:

  ```
  http://burger1.com/TestWar/
    burgerServlet?ViewItem&category=11776&item=5550662630&rd=1
  http://burger1.com/TestWar/
    burgerServlet?Command=Order&item=5550662630
  ```

- **Example 2**

  A company that provides call center services could monitor response time for functional areas by setting up a URL Group for each application function. If customers access services with this URL:

  ```
  http://genesystems/us/application_function/
  ```

  where *`application_function`* corresponds to applications such as `OrderEntry`, `AcctService`, and `Support`, the `pathprefix` property for each URL group would specify the appropriate *`application_function`*

» **Note** You can use the asterisk symbol (*) as a wildcard in `pathprefix` properties.

Define name for a URL Group

Use the `introscope.agent.urlgroup.group.`*`GroupKey`*`.format` to determine the names under which response time metrics for a URL Group, whose key is *`GroupKey`*, will appear in the Workstation.

Typically, the `format` property is used to assign a text string as the name for a URL. This example causes metrics for the URL Group with key `alpha` to appear in the Workstation under the name *`Alpha Group`*:

```
introscope.agent.urlgroup.group.alpha.format=Alpha Group
```

Advanced naming techniques for URL Groups (optional)

If desired, you can derive a URL Group name from request elements, such as the server port, the protocol, or from a substring of the request URL. This is useful if your application modules are easily differentiated by inspection of the request. The following sections describe advanced forms of the `format` property.

Using a host as a URL Group name

To organize metrics for a URL Group under names that reflect the hostname of the HTTP server associated with requests, define the `format` parameter as follows:

```
introscope.agent.urlgroup.group.alpha.format={host}
```

When `format={host}`, statistics for the following requests appear under the metric names `us.mybank.com` and `uk.mybank.com` respectively:

```
https://us.mybank.com/mifi/loanApp...
https://uk.mybank.com/mifi/loanApp...
```

Using a protocol as a URL Group name

To organize statistics for a URL Group under names that reflect the protocol associated with requests, define the `format` parameter as follows:

```
introscope.agent.urlgroup.group.alpha.format={protocol}
```

When `format={protocol}`, statistics will be grouped in the Investigator under metric names that correspond to the protocol portion of request URLs. For example, statistics for these requests would appear under the metric name `https`:

```
https://us.mybank.com/cgi-bin/mifi/scripts......
https://uk.mybank.com/cgi-bin/mifi/scripts......
```

Using a port as a URL Group name

To organize statistics for a URL Group under names that reflect the port associated with requests, define the `format` parameter as follows:

```
introscope.agent.urlgroup.group.alpha.format={port}
```

When `format={port}`, statistics will be grouped under names that correspond to the port portion of request URLs. For example, statistics for these requests would appear under the name `9001`.

```
https://us.mybank.com:9001/cgi-bin/mifi/scripts......
https://uk.mybank.com:9001/cgi-bin/mifi/scripts......
```

Using a parameter as a URL Group name

To organize statistics for a URL Group in the Investigator under metric names that reflect the value of a parameter associated with requests, define the format parameter as follows:

```
introscope.agent.urlgroup.group.alpha.format={query_param:param}
```

When `format={query_param:param}`, statistics will be grouped in the Investigator under metric names that correspond to value of the parameter specified. Requests without parameters will be listed under `<empty>`. For example, using this parameter definition:

```
introscope.agent.urlgroup.group.alpha.format={query_param:category}
```

statistics for these requests would appear under the metric name "734".

```
http://ubuy.com/ws/shopping?ViewItem&category=734&item=3772&tc=photo
http://ubuy.com/ws/shopping?ViewItem&category=734&item=8574&tc=photo
```

Using a substring of the request path as a URL Group name

To organize statistics for a URL Group under names that reflect a substring of the path portion of request URLs, define the `format` parameter as follows:

```
introscope.agent.urlgroup.group.alpha.format={path_substring:m:n}
```

where *m* is the index of the first character, and *n* is one greater than the index of the last character. For example, using this setting:

```
introscope.agent.urlgroup.group.alpha.format={path_substring:0:3}
```

statistics for this request would appear under the metric node "/ht"

```
http://research.com/htmldocu/WebL-12.html
```

Using a delimited portion of the request path as a URL Group name

To organize statistics for a URL Group under names that reflect a character-delimited portion request URL path, define the `format` parameter as follows:

```
introscope.agent.urlgroup.group.alpha.format=
  {path_delimited:delim_char:m:n}
```

where *delim_char* is the character that delimits the segments in the path, *m* is the index of the first segment to select, and *n* is one greater than the index of the last segment to select. For example, using this setting:

```
introscope.agent.urlgroup.group.alpha.format={path_delimited:/:2:4}
```

statistics for the requests of this form:

```
http://www.buyitall.com/userid,sessionid/pageid
```

would appear under the metric name */pageid*.

Note that:

- a delimited character counts as a segment
- the segment count starts at 0

This table shows the above example's segments as delimited by the slash character:

Segment Index	0	1	2	3
Segment String	/	*userid,sessionid*	/	*pageid*

You can specify multiple delimiters, as desired. For example, using this setting:

```
introscope.agent.urlgroup.group.alpha.format={path_delimited:/,:3:4}
```

Statistics for requests of the form shown above would appear under the metric name *sessionid*.

This table shows the above example's segments as delimited by the slash and the comma character:

Segment Index	0	1	2	3	4	5
Segment String	/	*userid*	,	*sessionid*	/	*pageid*

Using multiple naming methods for URL Groups

You can combine multiple naming methods in a single `format` string, as shown below:

```
introscope.agent.urlgroup.group.alpha.format=red {host} orange {protocol}
  yellow {port} green {query_param:foo} blue {path_substring:2:5} indigo
  {path_delimited:/:0:1} violet {path_delimited:/:1:4} ultraviolet
  {path_substring:0:0} friend computer
```

Using Blame tracers to mark Blame points

Introscope Blame technology tracks the performance of .NET applications to enable you to view metrics at the front and backends of your applications. This capability, referred to as Boundary Blame, allows users to triage problems in the application frontend or backend.

The following sections describe how you can use tracers to explicitly mark the frontends and backends in your application.

Blame tracers

Introscope provides tracers for capturing front and backend metrics: FrontendTracer and BackendTracer. These tracers explicitly mark a frontend and backend, respectively.

You can use FrontendTracer and BackendTracer to instrument your own code, such as code that accesses a backend, to cause Introscope to capture and present metrics for custom components.

If no FrontendTracer is configured, the first component in the blame stack will be the default frontend. If no BackendTracer is configured, Introscope will infer a backend—any component that opens a client socket will be a default backend if none is explicitly marked.

It is useful to use a BackendTracer to assign a desired name to an item that Introscope detects as a backend or to mark custom sockets that Introscope does not instrument.

FrontendTracer and BackendTracer are instances of a BlamePointTracer, which provides metrics such as average response time, per interval counts, concurrency, and stalls. A BlamePointTracer can be applied to "middle" components for a more granular Blame stack. A BlamePointTracer, however, does not populate Errors Per Interval metrics in the Introscope Investigator.

Blame tracers in standard PBDs

Two of the standard PBDs provided with Introscope and the .NET Agent, dotnet.pbd and sqlagent.pbd, implement Boundary Blame tracing:

- `PageInfoTracer` in the `dotnet.pbd` is an instance of a FrontendTracer.
- `SQLBackendTracer` in the `sqlagent.pbd` is an instance of a BackendTracer.

Disabling Boundary Blame

By default, Boundary Blame is enabled. To disable Boundary Blame in favor of the component-level Blame implemented in Introscope versions prior to the 7.0 release:

1. Open the `IntroscopeAgent.profile` file.
2. Set the property `introscope.agent.blame.type` to `standard` to:

   ```
   introscope.agent.blame.type=standard
   ```
3. Save the `IntroscopeAgent.profile` file.

CHAPTER 9

Transaction Tracer Options

This chapter has information about default transaction tracing behaviors and related configuration options.

Controlling transaction trace sampling

By default, Introscope agents trace each normalized unique URL in an application once per hour, to provide a sampling of transaction behavior. This enables historical analysis of potentially problematic transaction types without explicitly running transaction traces.

Transaction trace sampling is enabled by default. You can disable the behavior by uncommenting the following property in the agent profile, `IntroscopeAgent.profile`:

- `introscope.agent.transactiontracer.sampling.enabled`—Uncomment and set to `false` to disable transaction trace sampling.

You can configure the how many transactions are sampled per interval and how long that interval is by uncommenting the following properties in the agent profile.

» **WARNING** These configurations are usually performed in the Enterprise Manager. Configuring the following properties in the agent profile overrides any configuration made in the Enterprise Manager.

- `introscope.agent.transactiontracer.sampling.perinterval.count`—Uncomment and set how many transactions are sampled per interval. The default is 1.
- `introscope.agent.transactiontracer.sampling.interval.seconds`—Uncomment and set how long the sample interval is in seconds. The default is 120 seconds.

Transaction Trace component clamp

Introscope now sets a clamp (set by default to 5,000 components) to limit the size of traces. When this limit is reached, warnings appear in the log, and the trace stops.

This allows you to clamp an infinitely expanding transaction—for example when a component executes hundreds of object interactions and backend SQL calls. Without the clamp, Transaction Tracer views this as one transaction, continuing infinitely. Without a clamp in place, the CLR runs out of memory before the trace can be completed.

The new property for clamping infinitely expanding transactions is in the `IntroscopeAgent.profile` file:

- `introscope.agent.transactiontrace.componentCountClamp=5000`

For traces producing clamped components—those exceeding the CountClamp—traces are marked with an asterisk and have a tool tip assiociated with them, providing more information about the clamped metrics. For more information about viewing these traces, see the *Introscope Workstation User Guide*.

» Important If the clamp is set too low, you may encounter Performance Monitoring (PerfMon) or Leakhunter exceptions when your applications start. If you encounter this, your managed .NET applications must to be restarted

Transaction Tracer options

You can configure the Introscope Transaction Tracer to trace only the transactions that meet criteria you specify. You can filter by user ID data or by HTTP request and session properties.

» Important You can filter by user ID *or* by HTTP attributes, but not both. Do not configure both types of filters—bad metrics can result from this configuration.

To control which transactions are traced:

Step 1 Enable the .NET Agent to report filter parameters, as described in *Enable collection of filter parameters* on page 95.

Step 2 Configure filtering either by user ID or HTTP request data:

- *Filter transaction traces by user ID* on page 96
- *Filter transaction traces by HTTP request data* on page 97.

Enable collection of filter parameters

By default, the .NET Agent reports only the URL for transactions it traces. Reporting of individual HTTP properties is restricted to minimize the impact of transaction tracing on system overhead. To enable filtering you must first enable collection of HTTP properties, by adding the following line to the agent profile:

```
introscope.agent.asp.disableHttpProperties=false
```

This enables collection of:

- Application Name
- Context Path
- URL
- Normalized URL
- Session ID
- Server Name
- Context Path

You can enable collection of additional properties by adding this line to the agent profile:

```
introscope.agent.asp.optionalProperties=true
```

This enables collection of:

- Scheme
- URL Referrer
- Method

Filter transaction traces by user ID

To configure the .NET Agent to filter transaction traces by user IDs in frontend components, you must determine how your application specifies user IDs. The application architect who developed the application can provide this information.

The Introscope Transaction Tracer can identify user IDs that are accessed through one of these methods:

- HTTP Context identity
- URL user information
- HTTP request header
- an attribute of the HTTP session

» **WARNING** Perform only the configuration process that applies to your application's method of specifying user IDs.

Filter by context identity

If the user ID is accessed through the HTTP Context identity, uncomment this property in the agent profile:

```
introscope.agent.transactiontracer.userid.method=HttpContext.User.
  Identity.Name
```

Filter by URL user

If the user ID is accessed through the URL user information, uncomment the property in the agent profile:

```
introscope.agent.transactiontracer.userid.method=HttpContext.Request.
  Uri.UserInfo
```

Filter by request header

If the user ID is determined from the HTTP request header, uncomment the following pair of properties in the agent profile, and define a key string for the second property:

```
introscope.agent.transactiontracer.userid.method=HttpRequest.Headers.Get
```

```
introscope.agent.transactiontracer.userid.key=<application defined key
  string>
```

Filter by session attribute

If the user ID is an attribute in the Http Session, uncomment the following pair of properties in the Agent profile, and define a key string for the second property:

```
introscope.agent.transactiontracer.userid.method=HttpContext.Session.
  Contents
introscope.agent.transactiontracer.userid.key=<application defined key
  string>
```

Filter transaction traces by HTTP request data

This section has instructions for filtering transaction traces by HTTP request properties, including:

- request headers
- request parameters
- session attributes

You can filter using multiple properties, for instance by request parameter and by session attribute.

» **WARNING** Do not configure filtering by HTTP property if you have configured filtering by user ID.

To filter by HTTP request data:

1. Open the `IntroscopeAgent.profile` file.
2. Locate the Transaction Tracer properties under the *Transaction Tracer Configuration* heading.
3. To collect specific HTTP request headers, uncomment the following property and specify the HTTP request header(s) to track, in a comma-separated list:

   ```
   introscope.agent.transactiontracer.parameter.httprequest.
     headers=User-Agent
   ```

4. To collect HTTP request parameters data, uncomment this property and specify the HTTP request parameter(s) to track, in a comma-separated list:

   ```
   introscope.agent.transactiontracer.parameter.httprequest.
     parameters=parameter1,parameter2
   ```

5. To collect HTTP session attributes data, uncomment this property and specify the HTTP session attribute(s) to track, in a comma-separated list, for example:

   ```
   introscope.agent.transactiontracer.parameter.httpsession.
     attributes=attribute1,attribute2
   ```

6. Save your changes to the `IntroscopeAgent.profile` file.

7 Restart the application.

Disable the capture of stalls as events

By default, Introscope captures transaction stalls as events in the Transaction Event database, and generates stall metrics from the detected events. Stall metrics are generated for the first and last method in the transaction. Users can view stall events and associated metrics in the Workstation Historical Event Viewer.

» **Note** Generated stall metrics are always available, but stall events are only visible if Introscope Error Detector is installed. Stalls are stored as ordinary errors, and will be visible in the Errors Tab View, or in the historical query viewer by querying for "type:errorsnapshot".

You can disable the capture of stalls as events, change the stall threshold, or change the frequency with which the .NET Agent checks for stalls using these properties:

- `introscope.agent.stalls.enable`—controls whether the Agent checks for stalls and creates events for detected stalls.
- `introscope.agent.stalls.thresholdseconds`—specifies the minimum threshold response time at which time a transaction is considered stalled.
- `introscope.agent.stalls.resolutionseconds`—specifies the frequency that the agent checks for stalls.

» **Important** Support for stall tracers in PBDs will be deprecated if you disable the capture of stalls as events.

CHAPTER 10

Configure the Introscope SQL Agent

This chapter has instructions for configuring Introscope SQL Agent.

The SQL Agent overview

The Introscope SQL Agent reports detailed database performance data to the Enterprise Manager. The SQL Agent provides visibility into the performance of individual SQL statements in your application by tracking the interaction between your managed application and your database.

In the same way that the .NET Agent monitors Java applications, the SQL Agent monitors SQL statements. The SQL Agent is non-intrusive, monitoring the application or database with very low overhead.

To provide meaningful performance measurements down to the individual SQL statement level, the SQL Agent summarizes performance data by stripping out transaction-specific data and converting the original SQL statements into Introscope-specific normalized statements. Since normalized statements do not include sensitive information, such as credit card numbers, this process also protects the security of your data.

For example, the SQL Agent converts this SQL query:

```
SELECT * FROM BOOKS WHERE AUTHOR = 'Atwood'
```

to this normalized statement:

```
SELECT * FROM BOOKS WHERE AUTHOR = ?
```

Similarly, SQL Agent converts this SQL update statement:

```
INSERT INTO BOOKS (AUTHOR, TITLE) VALUES ('Atwood', 'The Robber Bride')
```

to this normalized statement:

```
INSERT INTO BOOKS (AUTHOR, TITLE) VALUES (?, ?)
```

» **Note** Only text within quotation marks ('xyz') is normalized.

Metrics for normalized statements are aggregated and can be viewed in the JDBC node of the Workstation Investigator.

The SQL Agent files

When you install an Introscope agent, the agent installer automatically installs the SQL Agent. The following files are installed:

- `wily/ext/wily.SQLAgent.ext.dll`
- `wily/sqlagent.pbd`

» **Note** By default, agent extensions like the `wily.SQLAgent.ext.dll` file are installed in the `wily/ext` directory. You can change the location of the agent extension directory with the `introscope.agent.extensions.directory` property in the agent profile. If you change the location of the `/ext` directory, be sure to move the contents of the `/ext` directory as well.

SQL statement normalization

Some applications may generate an extremely large number of unique SQL statements. If technologies like Hibernate are in use, the likelihood of long unique SQL statements increases. Long SQL statements can contribute to a metric explosion in the agent, leading to poor performance as well as other system problems.

» **Note** For more information about Hibernate, see http://www.hibernate.org/.

How poorly written SQL statements create metric explosions

If your SQL Agent is showing a large and increasing number of unique SQL metrics even though your application uses a small set of SQL statements, the problem could be in how the SQL statement was written.

In general, the number of SQL Agent metrics should approximate the number of unique SQL statements. A common reason this becomes a problem is because of how comments are used in SQL statements. For example, in this statement,

```
"/* John Doe, user ID=?, txn=? */ select * from table..."
```

the SQL Agent creates the following metric:

```
"/* John Doe, user ID=?, txn=? */ select * from table..."
```

Note that the comment is part of the metric name. While the comment is useful for the database administrator to see who is executing what query, the SQL Agent does not parse the comment in the SQL statement. Therefore, for each unique user ID, the SQL Agent creates a unique metric, potentially causing a metric explosion. The database that executes the SQL statements does not see these metrics as unique because it ignores the comments.

This problem can be avoided is by putting the SQL comment in single quotes, as shown:

```
"/*' John Doe, user ID=?, txn=? '*/ select * from table..."
```

The SQL Agent then creates the following metric where the comment no longer causes a unique metric name:

```
"/* ? */ select * from table..."
```

Example 1

When looking at this path under an agent node in the Investigator Backends|{backendName}|SQL|{sqlType}|sql you notice that temporary tables are being accessed like this:

```
SELECT * FROM TMP_123981398210381920912 WHERE ROW_ID = ?
```

All the additional digits on the `TMP_ table` name are unique and steadily growing causing a metric explosion.

Example 2

You have been alerted to a potential metric explosion and your investigation brings you to a review of this SQL statement:

```
#1 INSERT INTO COMMENTS (COMMENT_ID, CARD_ID, CMMT_TYPE_ID,
  CMMT_STATUS_ID,CMMT_CATEGORY_ID, LOCATION_ID, CMMT_LIST_ID,
  COMMENTS_DSC, USER_ID,LAST_UPDATE_TS) VALUES (?, ?, ?, ?, ?, ?, ?, "CHANGE
  CITY FROM CARROLTON,TO CAROLTON, _ ", ?, CURRENT)
```

In studying the code, you notice that `"CHANGE CITY FROM CARROLTON, TOCAROLTON, _ "` recurs as a dizzying array of cities.

Example 3

You have been alerted to a potential metric explosion and your investigation brings you to a review of this SQL statement:

```
CHANGE COUNTRY FROM US TO CA _ CHANGE EMAIL ADDRESS FROM TO BRIGGIN @ COM _ "
```

In studying the code, you notice CHANGE COUNTRY results in an endless list of countries. In addition, the placement of the quotes for countries results in people's e-mail addresses getting inserted into SQL statements. Here’s the source of metric explosion as well as other negative consequences.

SQL statement normalization options

To address long SQL statements, the SQL Agent includes the following normalizers for use:

Default SQL statement normalizer

The standard SQL statement normalizer is on by default in the SQL Agent. It normalizes text within single quotation marks ('xyz'). For example, the SQL Agent converts this SQL query:

```
SELECT * FROM BOOKS WHERE AUTHOR = 'Atwood'
```

to this normalized statement:

```
SELECT * FROM BOOKS WHERE AUTHOR = ?
```

Metrics for normalized statements are aggregated and can be viewed in the Workstation Investigator.

Custom SQL statement normalizer

The SQL Agent allows users to add extensions for performing custom normalization. To do so, you create a DLL file containing a normalization scheme that is implemented by the SQL Agent.

To apply a SQL statement normalizer extension:

1. Create an extension DLL file.

   » **Note** The entry point class for the SQL normalizer extension file has to implement com.wily.introscope.agent.trace.ISqlNormalizer interface.

   Making a DLL extension file involves creating a manifest file that contains specific keys for the SQL normalizer extension, which are detailed in step 2 below. However, for your extension to work, other general keys are required. These keys are the type you would use to construct any extension file. The extension file you create relates to database SQL statement text normalization, for example metrics under the `Backends|{backendName}|SQL|{sqlType}|{actualSQLStatement}` node. The `{actualSQLStatement}` is normalized by the SQL normalizer.

2. Place the following keys in the manifest of the created extension:

   - `com-wily-Extension-Plugins-List:testNormalizer1`

     » **Note** The value of this key can be anything. In this instance, `testNormalizer1` is used as an example. Whatever you specify as the value of this key, use it in the following keys as well.

   - `com-wily-Extension-Plugin-testNormalizer1-Type: sqlnormalizer`

- com-wily-Extension-Plugin-testNormalizer1-Version: 1
- com-wily-Extension-Plugin-testNormalizer1-Name: normalizer1

  Should contain the unique name of your normalizer, for example normalizer1.
- com-wily-Extension-Plugin-testNormalizer1-Entry-Point-Class: <Thefully-qualified classname of your implementation of ISQLNormalizer>

3. Place the extension file you created in the <Agent_Home>/wily/ext directory.
4. In the IntroscopeAgent.profile, locate and set the following property:

   introscope.agent.sqlagent.normalizer.extension

   Set the property to the com-wily-Extension-Plugin-{plugin}-Name from your created extension's manifest file. The value of this property is case-insensitive. For example:

   introscope.agent.sqlagent.normalizer.extension=normalizer1

   » **Important** This is a hot property. Changes to the extension name will result in re-registration of the extension.
5. In the IntroscopeAgent.profile, you can optionally add the following property to set the error throttle count:

   introscope.agent.sqlagent.normalizer.extension.errorCount

   For more information about errors and exceptions, see *Exceptions*, below.

   » **Note** If the errors thrown by the custom normalizer extension exceeds the error throttle count, the extension is disabled.
6. Save the IntroscopeAgent.profile.
7. Restart your application.

Exceptions

If the extension you created throws an exception for one query, the default SQL statement normalizer uses the default normalization scheme for that query. When this happens, an ERROR message is logged, saying an exception was thrown by the extension, and a DEBUG message is logged with stack trace information. However, after five such exceptions are thrown, the default SQL statement normalizer disables the your created extension and stops attempting to use the created extension for future queries until the normalizer is changed.

Null or empty strings

If the extension you created returns a null string or empty string for a query, the StatementNormalizer uses the default normalization scheme for that query and logs an INFO message saying the extension returned a null value. However, after five such null or empty strings have been returned, the StatementNormalizer stops logging messages, but will attempt to continue to use the extension.

Regular expression SQL statement normalizer

The SQL Agent ships with an extension that normalizes SQL statements based on configurable regular expressions (regex). This file, `wily.RegexSqlNormalizer.ext.dll`, is located in the `<Agent_Home>/wily/ext` directory. The normalizer extension uses `Systems.Test.RegularExpressions` namespace classes.

For examples on how to use the regular expression SQL statement normalizer, see *Regular expression SQL statement normalizer examples* on page 107.

To apply the regular expressions extension:

1. Open the `IntroscopeAgent.profile`.
2. Locate and set the following properties:
   - `introscope.agent.sqlagent.normalizer.extension=RegexSqlNormalizer`

     Specifies the name of the SQL normalizer extension that will be used to override the preconfigured normalization scheme. When enabling the regular expressions extension, set this property to RegexSqlNormalizer.
   - `introscope.agent.sqlagent.normalizer.regex.keys=key1`

     This property specifies the regex group keys, which are evaluated in the order they are listed. This property is required to enable the regular expressions extension. There is no default value.
   - `introscope.agent.sqlagent.normalizer.regex.key1.pattern=A`

     This property specifies the regex pattern that is used to match against the SQL statements. All valid regular expressions allowed by the System.Test.RegularExpressions namespace classes can be used here. This property is required to enable the regular expressions extension. There is no default value.
   - `introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=B`

     This property specifies the replacement string format. All valid regex allowed by the System.Test.RegularExpressions namespace classes can be used here. This property is required to enable the regular expressions extension. There is no default value.
   - `introscope.agent.sqlagent.normalizer.regex.matchFallThrough=false`

     If this property is set to true, SQL strings are evaluated against all the regex key groups. The implementation is chained. Hence, if the SQL strings match multiple key groups, the normalized SQL output from group1 is fed as input to group2, and so on.

     If the property is set to false, as soon as a key group matches the SQL string, the normalized SQL output from that group is returned. The MatchFallThrough property does not enable or disable the extension.

For example, if you had a SQL string like: Select * from A where B, you would set the following properties:

```
introscope.agent.sqlagent.normalizer.regex.keys=key1,key2
introscope.agent.sqlagent.normalizer.regex.key1.pattern=A
introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=X
introscope.agent.sqlagent.normalizer.regex.key2.pattern=B
introscope.agent.sqlagent.normalizer.regex.key2.replaceFormat=Y
```

If `introscope.agent.sqlagent.normalizer.regex.matchFallThrough=`**`false`**, then the SQL is normalized against key1 regex. Output from that regex will be `Select * from X where B`. This SQL will be returned.

If `introscope.agent.sqlagent.normalizer.regex.matchFallThrough=`**`true`**, then the SQL is normalized against key1 regex first. The output from that regex is `Select * from X where B`. This output is then fed to `key2 regex`. The output from `key2 regex` is `Select * from X where Y`. This will be the SQL returned.

» **Note** This property is not required to enable the regular expressions extension.

- `introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false`

  This property specifies whether the pattern match is case sensitive. The default value is false. This property is not required to enable the regular expressions extension.

- `introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false`

  If this property is set to false, it will replace the first occurrence of the matching pattern in the SQL with the replacement string. If this property is set to true, it will replace all occurrences of the matching pattern in the SQL with the replacement string.

  For example, if you have a SQL statement like Select * from A where A like Z, you would set the properties as follows:

  ```
  introscope.agent.sqlagent.normalizer.regex.key1.pattern=A
  introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=X
  ```

  If `introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=`**`false`**, it will result in a normalized SQL statement: `Select * from X where A like Z`.

  If `introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=`**`true`**, it will result in a normalized SQL statement: `Select * from X where` **`X`** `like Z`.

  The default value is false. This property is not required to enable the regular expressions extension.

» **Note** If none of the regular expression patterns match the input SQL, the RegexNormalizer will return a null string. The statement normalizer will then use the default normalization scheme.

3 Save the `IntroscopeAgent.profile`.

» **Important** All properties listed above are hot, meaning changes to these properties take effect once you have saved the IntroscopeAgent.profile. Changes to these properties do not require IIS restart.

Regular expression SQL statement normalizer examples

The three examples below can help you understand how to implement the regular expression SQL statement normalizer.

Example 1

Here is a SQL query before regular expression SQL statement normalization:

```
INSERT INTO COMMENTS (COMMENT_ID, CARD_ID, CMMT_TYPE_ID,CMMT_STATUS_ID,
  CMMT_CATEGORY_ID, LOCATION_ID, CMMT_LIST_ID,COMMENTS_DSC, USER_ID,
  LAST_UPDATE_TS) VALUES(?, ?, ?, ?, ?, ?,?, ''CHANGE CITY FROM CARROLTON,
  TO CAROLTON, _ ", ?, CURRENT)
```

Here is the desired normalized SQL statement:

```
INSERT INTO COMMENTS (COMMENT_ID, ...) VALUES (?, ?, ?, ?, ?, ?,?, CHANGE
  CITY FROM ( )
```

Here is the configuration needed to the `IntroscopeAgent.profile` file to result in the normalized SQL statement shown above:

```
introscope.agent.sqlagent.normalizer.extension=RegexSqlNormalizer
introscope.agent.sqlagent.normalizer.regex.matchFallThrough=true
introscope.agent.sqlagent.normalizer.regex.keys=key1,key2
introscope.agent.sqlagent.normalizer.regex.key1.pattern=(INSERT INTO
COMMENTS \\(COMMENT_ID,)(.*)(VALUES.*)''(CHANGE CITY FROM \\().*(\\))
introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false
introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=$1 ...)
  $3$4 $5
introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false
introscope.agent.sqlagent.normalizer.regex.key2.pattern='[a-zA-Z1-9]+'
introscope.agent.sqlagent.normalizer.regex.key2.replaceAll=true
introscope.agent.sqlagent.normalizer.regex.key2.replaceFormat=?
introscope.agent.sqlagent.normalizer.regex.key2.caseSensitive=false
```

Example 2

Here is a SQL query before regular expression SQL statement normalization:

```
SELECT * FROM TMP_123981398210381920912 WHERE ROW_ID =
```

Here is the desired normalized SQL statement:

```
SELECT * FROM TMP_ WHERE ROW_ID =
```

Here is the configuration needed to the `IntroscopeAgent.profile` file to resultin the normalized SQL statement shown above:

```
introscope.agent.sqlagent.normalizer.extension=RegexSqlNormalizer
introscope.agent.sqlagent.normalizer.regex.matchFallThrough=true
introscope.agent.sqlagent.normalizer.regex.keys=key1
introscope.agent.sqlagent.normalizer.regex.key1.pattern=(TMP_)[1-9]*
introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false
introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=$1
introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false
```

Example 3

If you want to normalize a SQL statement like: `Select .... ResID1, CustID1` where `ResID1=.. OR ResID2=.. n times OR CustID1=.. OR n times`, you could set the properties like this:

```
introscope.agent.sqlagent.normalizer.regex.matchFallThrough=true
introscope.agent.sqlagent.normalizer.regex.keys=default,def
introscope.agent.sqlagent.normalizer.regex.default.pattern=(ResID)[1-9]
introscope.agent.sqlagent.normalizer.regex.default.replaceAll=true
introscope.agent.sqlagent.normalizer.regex.default.replaceFormat=$1
introscope.agent.sqlagent.normalizer.regex.default.caseSensitive=true
introscope.agent.sqlagent.normalizer.regex.def.pattern=(CustID)[1-9]
introscope.agent.sqlagent.normalizer.regex.def.replaceAll=true
introscope.agent.sqlagent.normalizer.regex.def.replaceFormat=$1
introscope.agent.sqlagent.normalizer.regex.def.caseSensitive=true
```

Command-line SQL statement normalizer

If the regular expression SQL normalizer is not in use, and you have SQL statements that enclose values in the where clause with double quotes (" "), use the following command-line command to normalize your SQL statements:

```
-DSQLAgentNormalizeDoubleQuoteString=true
```

» Important You can use the regular expressions SQL normalizer instead of this command to normalize SQL statements in double quotes. See *Regular expression SQL statement normalizer* on page 105 for more information.

Turning off statement metrics

Some applications may generate an extremely large number of unique SQL statements, causing a metric explosion in the SQL Agent. You can turn off SQL statement metrics in the SQL Agent.

» **Note** You will not lose backend or top-level JDBC metrics if you turn off statement metrics.

To turn off statement metrics:

1. Open the `sqlagent.pbd` file.
2. Remove **{sql}** from the trace directives you wish to turn off.
3. Save the `sqlagent.pbd` file.

Turning off Blame metrics

In a standard deployment of the SQL Agent, Blame metric data is collected by default. However, to reduce data overhead and reduce the number of metrics generated, you can turn Blame metric data off for the SQL Agent.

Note: If Blame metric generation is turned off, the SQL Agent data will not appear in Transaction Tracer viewer.

To turn off Blame metric data generation:

1. Open the `IntroscopeAgent.profile`.
2. Locate the property, `introscope.agent.sqlagent.useblame`.
3. Change the value to `false`:

   ```
   introscope.agent.sqlagent.useblame=false
   ```
4. Save your changes to the `IntroscopeAgent.profile`.
5. Restart the managed application.

SQL metrics

The SQL Agent metrics appear under the **Backends** node in the Introscope Workstation Investigator. SQL statement metrics can be found under the **Backends|<backendName>|SQL** node.

» **Note** `Average Response Time (ms)` will only display queries that return a data reader, i.e. queries executed via the `ExecuteReader()` method. This metric represents the average time spent in the data reader's `Close()` method.

Metric types specific to SQL data include:

- Connection Count—The number of live connection objects in memory.

  A connection is opened when a driver's `Open()` method is invoked, and closed when the connection invocation is closed via the `Close()` method. The SQL Agent maintains weak references to Connections in a Set. When the Connection objects are garbage collected, the counts reflect the changes.

- Average Result Processing Time (ms)—The average processing time of a query.

  This metric represents the average time spent processing a ResultSet from the end of the `executeQuery()` call to the invocation of the ResultSet's `close()` method.

» **Note** Instrumented XADataSources may not report commit or rollback metrics. Other instrumented DataSources may not report commit or rollback metrics unless those metrics contain data.

APPENDIX A

.NET Agent Properties

This appendix documents the properties in *<Agent_Home>*wily\IntroscopeAgent.profile.

.NET Agent to Enterprise Manager connection

introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT

Usage	The host name of machine running the Enterprise Manager.
Options	
Default	`localhost`
Example	`introscope.agent.enterprisemanager.transport.tcp.host.DEFAULT=localhost`
Notes	

introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT

Usage	The port on the Enterprise Manager machine that listens for the agent.
Options	
Default	`5001`
Example	`introscope.agent.enterprisemanager.transport.tcp.port.DEFAULT=5001`
Notes	

introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT

Usage	Change this property to use a different client socket factory.
Options	
Default	`com.wily.isengard.postofficehub.link.net.DefaultSocketFactory`
Example	`introscope.agent.enterprisemanager.transport.tcp.socketfactory.DEFAULT=com.wily.isengard.postofficehub.link.net.DefaultSocketFactory`
Notes	

.NET Agent failover

If the .NET Agent loses connection with its primary Enterprise Manager, these properties specify which Enterprise Manager the agent will failover to, and how often it will try to reconnect to its primary Enterprise Manager. For more information on agent failover, see *.NET Agent Failover* on page 77.

introscope.agent.enterprisemanager.connectionorder

Usage	The connection order of backup Enterprise Managers the agent uses if it is disconnected from its default Enterprise Manager.
Options	Names of other Enterprise Managers the agent can connect to.
Default	default

introscope.agent.enterprisemanager.connectionorder

Example	`introscope.agent.enterprisemanager.connectionorder=DEFAULT`
Notes	Use a comma separated list.

introscope.agent.enterprisemanager.failbackRetryIntervalInSeconds

Usage	Number of seconds between attempts by the agent to reconnect to its primary Enterprise Manager.
Options	
Default	Commented out; 120
Example	`#introscope.agent.enterprisemanager.failbackRetryIntervalInSeconds=120`
Notes	

.NET Agent metric clamp

This property allows you to configure the .NET Agent to approximately clamp the number of metrics sent to the Enterprise Manager. If the number of metrics pass this metric clamp value then no new metrics will be created.

introscope.agent.metricClamp

Usage	Configures the agent to approximately clamp the number of metrics sent to the Enterprise Manager.
Options	
Default	5000
Example	`introscope.agent.metricClamp=5000`
Notes	■ If the property is not set then no metric clamping will occur. Old metrics will still report values. ■ Changes to this property take effect immediately and do not require the managed application to be restarted.

.NET Agent naming

The following properties are for .NET Agent naming. For more information on .NET Agent naming, see *.NET Agent name options* on page 35.

introscope.agent.agentAutoNamingEnabled

Usage	Specifies whether agent autonaming will be used to obtain the .NET Agent name for supported application servers.
Options	True or False
Default	False

introscope.agent.agentAutoNamingEnabled

Example `#introscope.agent.agentAutoNamingEnabled=false`

Notes
- You must restart the managed application before changes to this property take effect.
- Requires the Startup Class to be specified for WebLogic; requires Custom Service to be specified for WebSphere.
- Set to true, and not commented out in agent profiles shipped with supported application servers

introscope.agent.agentAutoNamingMaximumConnectionDelayInSeconds

Usage Specifies the amount of time in seconds the agent waits for naming information before connecting to the Enterprise Manager.

Options

Default 120

Example
```
introscope.agent.agentAutoNamingMaximumConnectionDelayIn
  Seconds=120
```

Notes

introscope.agent.agentAutoRenamingIntervalInMinutes

Usage Specifies the time interval in minutes during which the agent will check to see if it has been renamed.

Options

Default 10

Example `introscope.agent.agentAutoRenamingIntervalInMinutes=10`

Notes

introscope.agent.disableLogFileAutoNaming

Usage Disables automatic naming of an agent's log files—the default behavior when an agent is configured for autonaming.

Options True or False

Default False

Example `introscope.agent.disableLogFileAutoNaming=false`

Notes

introscope.agent.agentName

Usage Name of Agent.

Options For any installation, if the value of this property is invalid or if this property is deleted from the profile, the agent name will be `Unknown Agent`.

Default

introscope.agent.agentName

Example	`#introscope.agent.agentName=AgentName`
Notes	■ Uncomment this property to provide a default Agent name if other Agent naming methods fail. ■ In the agent profile provided with application server-specific agent installers, the default reflects the application server, for instance `WebLogic Agent`. ■ In the agent profile provided with the default agent installer, the property value is `AgentName`, and the line is commented out.

introscope.agent.clonedAgent

Usage	
Options	True or False
Default	false
Example	`introscope.agent.clonedAgent=false`
Notes	Set to true when running identical copies of an application on the same machine.

Agent metric aging

Agent metric aging periodically removes dead metrics from the agent memory cache. A dead metric is a metric that has no new data reported in a given amount of time. This helps the agent improve performance and avoid potential metric explosions.

» **Note** A metric explosion happens when an agent is inadvertently set up to report more metrics than the system can handle. In this case, Introscope is bombarded with such a large number of metrics that performance gets very slow or the system cannot function at all.

Metrics that are in a group are removed only if all metrics in the group are considered candidates for removal. Currently, only `BlamePointTracer` group and `MetricRecordingAdministrator` metrics are removed as a unit; other metrics are removed individually.

The `MetricRecordingAdministrator` metric has APIs that can create a metric group. These APIs are:

- `getAgent().IAgent_getMetricRecordingAdministrator.addMetricGroup`
  String component, collection metrics. The component name is the metric resource name of the metric group. The metrics must be under the same metric node in order to qualify as a group. The metrics are a collection of `com.wily.introscope.spec.metric.AgentMetric` data structures. You can only add `AgentMetric` data structures to this Collection.
- `getAgent().IAgent_getMetricRecordingAdministrator.getMetricGroup`

String component. Based on the component name which is the metric resource name, you can get the Collection of metrics.

- `getAgent().IAgent_getMetricRecordingAdministrator.removeMetricGroup`
  String component. The metric group is removed based on the component which is the metric resource name.
- `getAgent().IAgent_getDataAccumulatorFactory.isRemoved`
  Checks if the metric is removed. You use this API if you keep an instance of an accumulator in your extension. If the accumulator is removed because of metric aging then you will be holding onto a dead reference.

» **Important** » Important If you create an extension that uses a `MetricRecordingAdministrator` API (for example, for use with CA Wily product), be sure to delete your own instance of an accumulator. When a metric ages out because it has not been invoked, and then after a time data does become available for that metric, if you are using an old accumulator instance, the accumulator will not create new metric data points for that metric. To avoid this situation, do not delete your own instance of an accumulator and use instead the `getDataAccumulatorFactory` API.

Configuring agent metric aging

Agent metric aging is on by default. You can choose to turn off this capability using the property *introscope.agent.metricAging.turnOn* on page 117. If you remove this property from the IntroscopeAgent.profile, agent metric aging is turned off by default.

Agent metric aging runs on a heartbeat in the agent. The heartbeat is configured using the property *introscope.agent.metricAging.heartbeatInterval* on page 117. Be sure to keep the frequency of the heartbeat low. A higher heartbeat will impact the performance of the agent and Introscope.

During each heartbeat, a certain set of metrics are checked. This is configurable using the property *introscope.agent.metricAging.dataChunk* on page 117. It is also important to keep this value low, as a higher value will impact performance. The default value is 500 metrics to be checked per heartbeat. Each of the 500 metrics is checked to see if it is a candidate for removal. For example, if you set this property to check chunks of 500 metrics per heartbeat, and you have a total of 10,000 metrics in the agent memory, then it will take longer with lower impact on performance to check all 10,000 metrics. However, if you set this property to a higher number, you would check all 10,000 metrics faster, but with possibly high overhead.

A metric is a candidate for removal if the metric has not received new data after certain period of time. You can configure this period of time using the property *introscope.agent.metricAging.numberTimeslices* on page 117. This property is set to 3000 by default. If a metric meets the condition for removal, then a check is performed to see if all the metrics in its group are candidates for metric removal. If this requirement has also been met then the metric is removed.

» **Note** For metrics that do not have a metric group then the rule does not apply.

Based on the rules outlined above, it may take a significant amount of time for metrics to be removed.

Use the following properties to configure agent metric aging. In all of the properties, if any unrecognised values are used, the default value will be used instead.

introscope.agent.metricAging.turnOn

Usage	Turns on or off agent metric aging.
Options	`True or False`
Default	`True`
Example	`introscope.agent.metricAging.turnOn=true`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.metricAging.heartbeatInterval

Usage	The time interval when metrics are checked for removal, in seconds.
Options	
Default	`1800`
Example	`introscope.agent.metricAging.heartbeatInterval=1800`
Notes	You must restart the managed application before changes to this property take effect.

introscope.agent.metricAging.dataChunk

Usage	During each interval, the number of metrics that are checked.
Options	
Default	`500`
Example	`introscope.agent.metricAging.dataChunk=500`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.metricAging.numberTimeslices

Usage	The number of intervals to check without any new data before making it a candidate for removal.
Options	
Default	`3000`

introscope.agent.metricAging.numberTimeslices

Example	`introscope.agent.metricAging.numberTimeslices=3000`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.metricAging.metricExclude.ignore.0

Usage	To exclude metrics from being removed. Add the metric name or metric filter to the list.
Options	comma seperated list; use the * wildcard
Default	
Example	`introscope.agent.metricAging.metricExclude.ignore.0=Threads*`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

Agent thread priority

The following property controls the priority of agent threads.

introscope.agent.thread.all.priority

Usage	Specifies the priority of agent threads.
Options	Values vary from 0 (low) to 4 (high).
Default	`2`
Example	`introscope.agent.thread.all.priority=2`
Notes	

AutoProbe

These properties are for configuring AutoProbe. For more information, see *ProbeBuilder Directives* on page 47.

introscope.autoprobe.directivesFile

Usage	Specifies directives files for AutoProbe. For more information, see *ProbeBuilder Directives* on page 47.
Options	
Default	Varies by installer.
Example	
Notes	This is a required parameter. If the AutoProbe properties are not set, or the values are invalid, Introscope will not function.

introscope.autoprobe.enable

Usage	When this property is set to false, it disables AutoProbe. The .NET Agent does not connect to the Enterprise Manager, does not appear in the Investigator, and does not report metrics.
Options	True or False
Default	`true`
Example	`introscope.autoprobe.enable=true`
Notes	

introscope.autoprobe.logfile

Usage	Name and location of AutoProbe log file.
Options	
Default	`logs\AutoProbe.log`
Example	`introscope.autoprobe.logfile=logs\AutoProbe.log`
Notes	The location of the AutoProbe log file can be changed.

ChangeDetector configuration

The following properties configure the Local Product Short interaction with ChangeDetector.

introscope.changeDetector.disable

Usage	This boolean property gives you the ability to enable Introscope ChangeDetector by settings the property value to true.
Options	True or False
Default	false
Example	`introscope.changeDetector.disable=false`
Notes	■ Commented out by default. ■ You must restart the managed application before changes to this property take effect.

introscope.changeDetector.rootDir

Usage	The root directory is the folder where ChangeDetector creates its local cache files.
Options	
Default	
Example	`introscope.changeDetector.rootDir=c:\\sw\\AppServer\\wily\\ \change_detector`
Notes	■ Commented out by default. ■ Use a backslash to escape the backslash character, as in the example.

introscope.changeDetector.isengardStartupWaitTimeInSec

Usage	Time to wait after the agent starts before trying to connect to the Enterprise Manager.
Options	
Default	15
Example	`introscope.changeDetector.isengardStartupWaitTimeInSec=15`
Notes	■ Commented out by default. ■ Second increments.

introscope.changeDetector.waitTimeBetweenReconnectInSec

Usage	Specify the number of seconds to wait before retrying connection to the Enterprise Manager.
Options	
Default	10
Example	`introscope.changeDetector.waitTimeBetweenReconnectInSec=10`
Notes	■ Commented out by default. ■ Second increments.

introscope.changeDetector.disableEPA

Usage	When ChangeDetector Local Product Short is enabled, an Local Product Short plug-in can be used as a datasource for change data in XML format.
Options	True or False
Default	true
Example	`introscope.changeDetector.disableEPA=true`
Notes	Commented out by default.

introscope.changeDetector.agentID

Usage	A string used by ChangeDetector to identify the Local Product Short.
Options	
Default	
Example	`introscope.changeDetector.agentID=SampleApplicationName`
Notes	Commented out by default.

introscope.changeDetector.profile

Usage	The absolute or relative path to the ChangeDetector datasources configuration file.
Options	
Default	
Example	`introscope.changeDetector.profile=CDConfig\\changedetector-config.xml`
Notes	■ Commented out by default. ■ Use a backslash to escape the backslash character, as in the example.

introscope.changeDetector.profileDir

Usage	The absolute or relative path to the directory where the datasources configuration file lives.
Options	
Default	
Example	`introscope.changeDetector.profileDir=CDConfig`
Notes	■ Commented out by default. ■ Use a backslash to escape the backslash character, as in the example. ■ You can use this value in the `introscope.changeDetector.profile` property, as shown in the example for that property.

Default domain configuration

The following property controls the default domain connection to the Enterprise Manager.

introscope.agent.dotnet.enableDefaultDomain

Usage	This property determines if the agent connected to the default domain connects to the Enterprise Manager.
Options	True or False
Default	`False`
Example	`introscope.agent.dotnet.enableDefaultDomain=false`
Notes	■ You must add this property to the `IntroscopeAgent.profile` to enable it. ■ When this property is set to true, the agent monitoring the default domain also gets reported in the Investigator.

Error Detector

These properties are for configuring the .NET Agent's interactions with Introscope ErrorDetector.

introscope.agent.errorsnapshots.enable

Usage Enable the agent to captures transaction details about serious errors.

Options True or False

Default `True`

Example

Notes
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- Requires Introscope Error Detector.

introscope.agent.errorsnapshots.throttle

Usage The maximum number of error snapshots that the agent can send in a 15-second period.

Options

Default `10`

Example
```
introscope.agent.errorsnapshots.throttle=10
```

Notes
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- Requires Introscope Error Detector.

introscope.agent.errorsnapshots.ignore.<index>

Usage This indexed property allows you to specify error messages to ignore. Error snapshots will not be generated or sent for errors with messages matching these filters. You may specify as many as you like (using .0, .1, .2 ...). You may use wildcards (*).

Options

Default Example definitions are provided, and commented out, as shown below. The following are examples only.

Example
```
#introscope.agent.errorsnapshots.ignore.0=*com.company.HarmlessException*
#introscope.agent.errorsnapshots.ignore.1=*HTTP Error Code: 404*
```

Notes
- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- Requires Introscope Error Detector.

Extensions

The following property is for .NET Agent extensions. For more information on Agent extensions, see *ProbeBuilder Directives* on page 47.

introscope.agent.extensions.directory

Usage	Specifies the location of all extensions to be loaded by the .NET Agent. Non-absolute names are resolved relative to the location of the `IntroscopeAgent.properties` file.
Options	
Default	`ext`
Example	`introscope.agent.extensions.directory=ext`
Notes	

LeakHunter configuration

These properties are for configuring the .NET Agent's interactions with Introscope LeakHunter.

introscope.agent.leakhunter.enable

Usage	Controls whether the feature is enabled if the LeakHunter Add-on is present. Set the value to true to enable LeakHunter.
Options	True or False
Default	`false`
Example	`introscope.agent.leakhunter.enable=false`
Notes	You must restart the managed application before changes to this property take effect.

introscope.agent.leakhunter.logfile.location

Usage	Controls the location for the LeakHunter log file. Filenames are relative to the application working directory. Leave the value blank if you do not want LeakHunter to record data to a log file.
Options	
Default	`logs/LeakHunter.log`
Example	`introscope.agent.leakhunter.logfile.location=logs/LeakHunter.log`
Notes	You must restart the managed application before changes to this property take effect.

introscope.agent.leakhunter.logfile.append

Usage	Controls whether LeakHunter will append or overwrite the log file. Set the value to true to append to the log file.
Options	True or False
Default	`false`

introscope.agent.leakhunter.logfile.append

Example	`introscope.agent.leakhunter.logfile.append=false`
Notes	You must restart the managed application before changes to this property take effect.

introscope.agent.leakhunter.leakSensitivity

Usage	Controls the sensitivity of the leak detection algorithm. A higher sensitivity setting will result in more potential leaks reported and a lower sensitivity will result in fewer potential leaks reported.
Options	The value should be an integer from 1-10.
Default	`5`
Example	`introscope.agent.leakhunter.leakSensitivity=5`
Notes	You must restart the managed application before changes to this property take effect.

introscope.agent.leakhunter.timeoutInMinutes

Usage	Controls the length of time LeakHunter spends looking for new potential leaks. After the timeout, LeakHunter will stop looking for new potential leaks and just continue tracking the previously identified potential leaks.
Options	
Default	`120`
Example	`introscope.agent.leakhunter.timeoutInMinutes=120`
Notes	■ Set the value to zero if you want LeakHunter to always look for new potential leaks. ■ You must restart the managed application before changes to this property take effect.

introscope.agent.leakhunter.collectAllocationStackTraces

Usage	Controls whether LeakHunter generates allocation stack traces for potential leaks. Turning this on gives you more precise data about the potential leak's allocation, but requires additional memory and CPU overhead. For this reason, the default setting is false.
Options	True or False
Default	`false`
Example	`introscope.agent.leakhunter.collectAllocationStackTraces=false`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

#introscope.agent.leakhunter.ignore.0

Usage	Ignore properties let the user specify particular collections that should be ignored by LeakHunter. For Generic collections, use a syntax that includes the generic type qualification, for example: `system.Collections.Generic.List`1`
Options	

#introscope.agent.leakhunter.ignore.0

Default	`Commented out.`
Example	`#introscope.agent.leakhunter.ignore.0=`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

Logging

These properties are for configuring the logging properties. For more information, see *Monitoring and Logging* on page 67.

introscope.agent.log.config.path

Usage	This property points to the Log4Net configuration file.
Options	
Default	`logging.config.xml`
Example	`introscope.agent.log.config.path=logging.config.xml`
Notes	

Performance monitoring configuration

The following properties define the performance monitoring configurations. For more information about performance monitoring, see *Performance monitoring (PerfMon) metric collection* on page 36.

introscope.agent.perfmon.enable

Usage	When set to true, enables performance monitoring metric collection.
Options	True of False
Default	`True`
Example	`introscope.agent.perfmon.enable=true`
Notes	

introscope.agent.perfmon.metric.filterPattern

Usage	The performance monitor counter expression.
Options	See notes below.
Default	`|Processor|*|*,|.NET Data Provider*|*|*,|.NET CLR*|{osprocessname}|*,|.NET CLR Data|*|*,|Process|{osprocessname}|*,|ASP.NET|*`
Example	`introscope.agent.perfmon.metric.filterPattern=|Processor|*|*,|.NET Data Provider*|*|*,|.NET CLR*|{osprocessname}|*,|.NET CLR Data|*|*,|Process|{osprocessname}|*,|ASP.NET|*`
Notes	To add new counters, such as process specific thread metrics, add the new expression to the end of the list, separated by a comma. For example: `...,|Thread|{osprocessname}*|*`

introscope.agent.perfmon.metric.limit

Usage	Specifies the metric ceiling.
Options	
Default	`1000`
Example	`introscope.agent.perfmon.metric.limit=1000`
Notes	

introscope.agent.perfmon.metric.pollIntervalInSeconds

Usage	Specifies the category polling interval in seconds.
Options	
Default	`15`
Example	`introscope.agent.perfmon.metric.pollIntervalInSeconds=15`
Notes	All metric values are polled every 15 seconds.

introscope.agent.perfmon.category.browseIntervalInSeconds

Usage	Intervals in which new PerfMon categories are discovered.
Options	
Default	`600 seconds (10 minutes)`
Example	`introscope.agent.perfmon.category.browseIntervalInSeconds=600`
Notes	

introscope.agent.perfmon.agentExpression

Usage	Enables or disables performance monitoring reporting from specific agent processes. Use this when more than one agent instance uses the same agent profile. `AgentName|ProcessName`
Options	
Default	*|*
Example	`introscope.agent.perfmon.agentExpression=*|*`
Notes	By default the Agent will report performance monitoring data for all processes.

Process name

The following properties define the process name. For more information, please see *ProbeBuilder Directives* on page 47.

introscope.agent.customProcessName

Usage	Specifies the process name as it appears in the Enterprise Manager and Workstation. Uncomment this property and define the process name to have the custom name appear in the Enterprise Manager and Workstation.
Options	

introscope.agent.customProcessName

Default	`CustomProcessName`
Example	`introscope.agent.customProcessName=CustomProcessName`
Notes	This property is commented out by default.

introscope.agent.defaultProcessName

Usage	The default process name will be used if no custom process name has been defined and the Agent is unable to determine the name of the main application class.
Options	
Default	`UnknownProcess`
Example	`introscope.agent.defaultProcessName=UnknownProcess`
Notes	

Restricting instrumentation configuration

The following property allows you to enable or disable instrumentation for targeted sets of processes or executables. For more information on restricting instrumentation, see *Configuring instrumentation* on page 34.

introscope.agent.dotnet.monitorApplications

Usage	Specifies processes and applications to instrument.
Options	See notes.
Default	`w3wp.exe,aspnet_wp.exe`
Example	`introscope.agent.dotnet.monitorApplications=w3wp.exe,aspnet_wp.exe,dllhost.exe`
Notes	Both partial names and fully qualified paths are supported.

introscope.agent.dotnet.monitorAppPools

Usage	Specifies application pools to be instrumented.
Options	See notes.
Default	`w3wp.exe, aspnet_wp.exe`
Example	`introscope.agent.dotnet.monitorAppPools=`
Notes	Leave commented to instrument all application pools, or uncomment and list only the application pools you want instrumented.

Socket metrics

The following properties are for socket metrics. For more information on socket metrics, see *ProbeBuilder Directives* on page 47.

introscope.agent.sockets.reportRateMetrics

Usage	Enables reporting of individual Socket's Input/Output Bandwidth rate metrics.
Options	True or False
Default	True
Example	`introscope.agent.sockets.reportRateMetrics=true`
Notes	

SQL Agent

The following properties are for the SQL Agent. For more information on the SQL Agent, see *Configure the Introscope SQL Agent* on page 99.

introscope.agent.sqlagent.useblame

Usage	Specifies whether SQL Agent will generate blame metric data.
Options	True or False
Default	True
Example	`introscope.agent.sqlagent.useblame=true`
Notes	

introscope.agent.sqlagent.sql.maxlength

Usage	Limits how much of a SQL statement appears in the Investigator tree for SQL Agent metrics, in bytes.
Options	
Default	990
Example	`introscope.agent.sqlagent.sql.maxlength=990`
Notes	Does not appear in `IntroscopeAgent.profile`. To change the value, add the property to the agent profile.

The following property is used to set the custom SQL Agent normalizer extension:

introscope.agent.sqlagent.normalizer.extension

Usage	Limits how much of a SQL statement appears in the Investigator tree for SQL Agent metrics, in bytes.
Options	The name of the SQL normalizer extension that will be used to override the preconfigured normalization scheme.
Default	RegexSqlNormalizer

introscope.agent.sqlagent.normalizer.extension

Example	`introscope.agent.sqlagent.normalizer.extension=RegexSqlNormalizer`
Notes	If you use the default setting, you also must configure the regular expressions SQL statement normalizer properties below.

The following properties are used to set the regular expressions SQL statement normalizer:

introscope.agent.sqlagent.normalizer.regex.matchFallThrough

Usage	This property if set to true will make sql strings to be evaluated against all the regex key groups.
Options	True or False
Default	false
Example	`introscope.agent.sqlagent.normalizer.regex.matchFallThrough=false`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.keys

Usage	This property specifies the regex group keys.
Options	
Default	key1
Example	`introscope.agent.sqlagent.normalizer.regex.keys=key1`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.pattern

Usage	This property specifies the regex pattern that will be used to match against the SQL.
Options	All valid regex allowed by `System.Test.RegularExpressions` namespace classes package can be used here.
Default	.*call(.*\)\.FOO(.*\)
Example	`introscope.agent.sqlagent.normalizer.regex.key1.pattern=.*call(.*\)\.FOO(.*\)`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.replaceAll

Usage	This property if set to 'false' will replace the first occurrence of the matching pattern in the sql with the replacement string. If set to 'true' it will replace all occurrences of the matching pattern in the sql with replacement string.
Options	True or False
Default	false

introscope.agent.sqlagent.normalizer.regex.key1.replaceAll

Example	`introscope.agent.sqlagent.normalizer.regex.key1.replaceAll=false`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat

Usage	This property specifies the replacement string format.
Options	All valid regex allowed by `System.Test.RegularExpressions` namespace classes can be used here.
Default	$1
Example	`introscope.agent.sqlagent.normalizer.regex.key1.replaceFormat=$1`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive

Usage	This property specifies whether the pattern match is sensitive to case.
Options	true or false
Default	false
Example	`introscope.agent.sqlagent.normalizer.regex.key1.caseSensitive=false`
Notes	Changes to this property take effect immediately and do not require the managed application to be restarted.

Stall metrics

The following sections define the properties related to the stall metrics. For more information on the stall metrics, see *Disable the capture of stalls as events* on page 98.

introscope.agent.stalls.thresholdseconds

Usage	Specifies the minimum threshold in seconds for stall event duration.
Options	
Default	30
Example	`introscope.agent.stalls.thresholdseconds=30`
Notes	This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.

introscope.agent.stalls.resolutionseconds

Usage	Specifies the frequency that the agent checks for stalls.
Options	

introscope.agent.stalls.resolutionseconds

Default	10
Example	`introscope.agent.stalls.resolutionseconds=10`
Notes	This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.

Transaction tracing

The following properties are for Transaction Tracing. For more information on Transaction Tracing, see *Transaction Tracer Options* on page 93.

introscope.agent.transactiontracer.parameter.httprequest.headers

Usage	Specifies HTTP request header data to capture. Use a comma separated list.
Options	
Default	Commented out; `User-Agent`
Example	`#introscope.agent.transactiontracer.parameter.httprequest.headers=User-Agent`
Notes	The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the desired header names.

introscope.agent.transactiontracer.parameter.httprequest.parameters

Usage	Specifies HTTP request parameter data to capture. Use a comma separated list.
Options	
Default	Commented out; generic parameters.
Example	`#introscope.agent.transactiontracer.parameter.httprequest.parameters=parameter1,parameter2`
Notes	The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the desired parameter names.

introscope.agent.transactiontracer.parameter.httpsession.attributes

Usage	Specifies HTTP session attribute data to capture. Use a comma separated list.
Options	
Default	Commented out; generic parameters.

introscope.agent.transactiontracer.parameter.httpsession.attributes

Example
```
#introscope.agent.transactiontracer.parameter.httpsession.
  attributes=attribute1,attribute2
```

Notes The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the desired parameter names.

introscope.agent.transactiontracer.userid.key

Usage User-defined key string.

Options

Default Commented out; generic parameters.

Example
```
#introscope.agent.transactiontracer.parameter.httpsession.
  attributes=attribute1,attribute2
```

Notes The `IntroscopeAgent.profile` contains a commented out statement that sets the value of this property to a null value. The user may optionally uncomment the statement and supply the correct value if, in your environment, user IDs are accessed using `HttpServletRequest.getHeader` or `HttpServletRequest.getValue`.

For more information, see *introscope.agent.transactiontracer.userid.method*, below.

introscope.agent.transactiontracer.userid.method

Usage Specifies the HttpRequest.getRemoteUser method to specify User IDs.

Options Allowable values are:

- `HttpContext.User.Identity.Name`
- `HttpContext.Request.Uri.UserInfo`
- `HttpRequest.Headers.Get`
- `HttpContext.Session.Contents`

Default Commented out; see options above.

Example The `IntroscopeAgent.profile` includes a commented out property definition for each of the allowable values.

Notes Uncomment the appropriate statement, based on whether user ID is accessed by getRemoteUser, getHeader, or getValue.

introscope.agent.transactiontrace.componentCountClamp

Usage Limiting the number of traces.

Options

Default 5000

Example
```
introscope.agent.transactiontrace.componentCountClamp=5000
```

Notes

- This property is dynamic. You can change the configuration of this property during run time and the change will be picked up automatically.
- When the set limit is reached, warnings appear in the log, and the trace stops.

URL grouping

These properties are for configuring URL Groups for frontend metrics. For more information, see *Using URL Groups* on page 84.

introscope.agent.urlgroup.keys

Usage	Configuration settings for Frontend naming.
Options	
Default	`Default`
Example	`introscope.agent.urlgroup.keys=default`
Notes	If a URL address belongs to two URL Groups, the order in which you list the keys for the URL Groups in this property is important. The URL Group defined by the narrower pattern should precede the URL Group specified by the broader pattern. For example, if the URL Group with key alpha contains a single address, and the URL Group with key beta includes all addresses on the network segment that contains the address in the first URL Group, alpha should precede beta in the keys parameter.

introscope.agent.urlgroup.group.default.pathprefix

Usage	Configuration settings for Frontend naming.
Options	
Default	`*`
Example	`introscope.agent.urlgroup.group.default.pathprefix=*`
Notes	

introscope.agent.urlgroup.group.default.format

Usage	Configuration settings for frontend naming.
Options	
Default	`Default`
Example	`introscope.agent.urlgroup.group.default.format=default`
Notes	

APPENDIX B

Additional Configuration of Application Parameters

The .NET framework permits configuration of application specific parameters using an optional XML format file with a `.config` extension

For ASP.NET applications, `Web.config` is the main file for application settings and configuration. This file is stored in the application root directory.

For other .NET executables, the configuration file is named the same as the application appended with an additional `.config` extension. The file is stored in the same directory as the application executable. For example, for `testapp.exe`, the optional configuration file would be `testapp.exe.config` .

It is possible to add Introscope specific configuration to the `.config` file. For example, parameters can be set that enable individual applications to reference their own instance of the `IntroscopeAgent.profile` file (to permit different applications to have different agent configurations), as well as to enable cross-process transaction correlation for web services.

To add specific properties to the .config file:

1. Open the application configuration file.
2. Add a `sectionGroup` and a `section` to your application configuration file. Name them as follows:

```
<configuration>
  <configSections>
    <sectionGroup>
      <sectionGroup name="com.wily.introscope.agent">
        <section name="env.parameters"
         type="System.Configuration.NameValueSectionHandler" />
      </sectionGroup>
```

3 Add new properties to the `env.parameters` section. For example:

```
<com.wily.introscope.agent>
  <env.parameters>
    <add key="com.wily.introscope.agentProfile"
      value="e:\\junkyard\\dotnettest\\Agent.profile" />
  </env.parameters>
</com.wily.introscope.agent>
```

For an example, see `sample.exe.config`.

» **Note** Configuration files are not required for .NET Agent installation.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X