


Monitor Azure Functions

 09/15/2017 •  13 minutes to read • Contributors      [all](#)

In this article

[Overview](#)

[Enable Application Insights integration](#)

[View telemetry data](#)

[Query telemetry data](#)

[Configure categories and log levels](#)

[Configure the aggregator](#)

[Configure sampling](#)

[Write logs in C# functions](#)

[Write logs in JavaScript functions](#)

[Custom telemetry in C# functions](#)

[Custom telemetry in JavaScript functions](#)

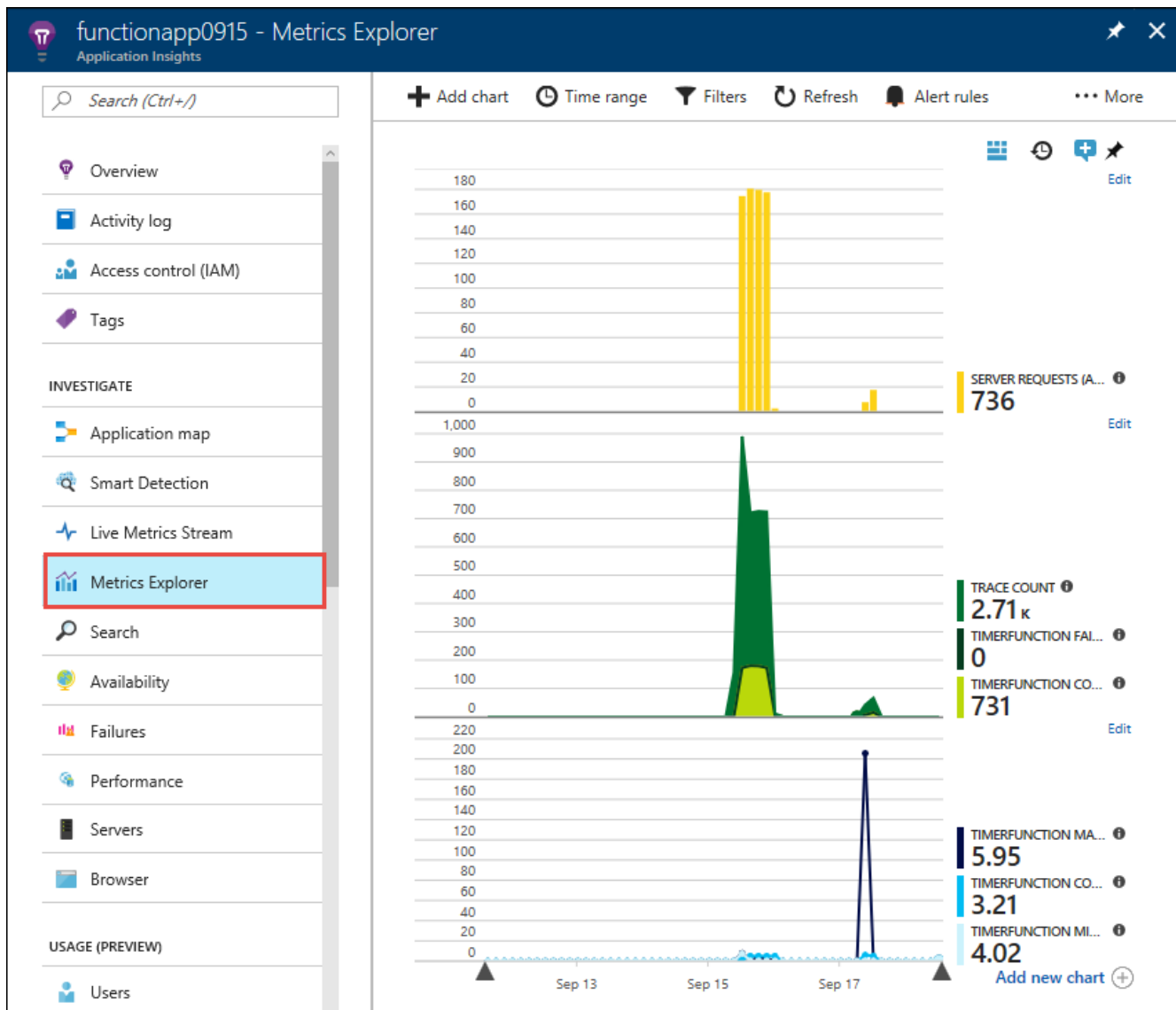
[Known issues](#)

[Monitoring without Application Insights](#)

[Next steps](#)

Overview

[Azure Functions](#) offers built-in integration with [Azure Application Insights](#) for monitoring functions. This article shows how to configure Functions to send telemetry data to Application Insights.



Functions also has built-in monitoring that doesn't use Application Insights. We recommend Application Insights because it offers more data and better ways to analyze the data. For information about the built-in monitoring, see the [last section of this article](#).

Enable Application Insights integration

For a function app to send data to Application Insights, it needs to know the instrumentation key of an Application Insights instance. There are two ways to make that connection in the [Azure portal](#):

- [Create a connected Application Insights instance when you create the function app.](#)
- [Connect an Application Insights instance to an existing function app.](#)

New function app

Enable Application Insights on the Function App **Create** page:

1. Set the **Application Insights** switch **On**.

2. Select an **Application Insights Location**.

The screenshot shows the 'Function App' creation page in the Microsoft Azure portal. The page has a dark blue header with the 'Microsoft Azure' logo and navigation icons. A left sidebar contains icons for various Azure services. The main content area is titled 'Function App' with a 'Create' button. Below the title, there are several configuration fields:

- * App name:** A text box containing 'functionapp0920' with a green checkmark and '.azurewebsites.net' below it.
- * Subscription:** A dropdown menu showing 'Windows Azure MSDN - Visual Studio Ultin'.
- * Resource Group:** Radio buttons for 'Create new' (selected) and 'Use existing'. Below is a text box with 'functionapp0920' and a green checkmark.
- * Hosting Plan:** A dropdown menu showing 'Consumption Plan'.
- * Location:** A dropdown menu showing 'South Central US'.
- * Storage:** Radio buttons for 'Create New' (selected) and 'Select Existing'. Below is a text box with 'functionapp0920' and a green checkmark.
- Application Insights:** A toggle switch set to 'On' (highlighted with a red box) and an 'Off' button.
- * Application Insights Location:** A dropdown menu showing 'East US' (highlighted with a red box).

At the bottom, there is a 'Pin to dashboard' checkbox and a 'Create' button. A link for 'Automation options' is also present.

Existing function app

Get an instrumentation key and save it in a function app:

1. Create the Application Insights instance. Set application type to **General**.

Application Insights
Monitor web app performance and usage

Name *i*
functionapp0920 ✓

* Application Type *i*
General ▼

* Subscription
Windows Azure MSDN - Visual Studio Ultin ▼

* Resource Group *i*
☒ Create new ☐ Use existing
functionapp0920 ✓

* Location
East US ▼

☐ Pin to dashboard

Create Automation options

2. Copy the instrumentation key from the **Essentials** page of the Application Insights instance. Hover over the end of the displayed key value to get a **Click to copy** button.

functionapp0920
Application Insights - Last 6 minutes (1 minute granularity) - General

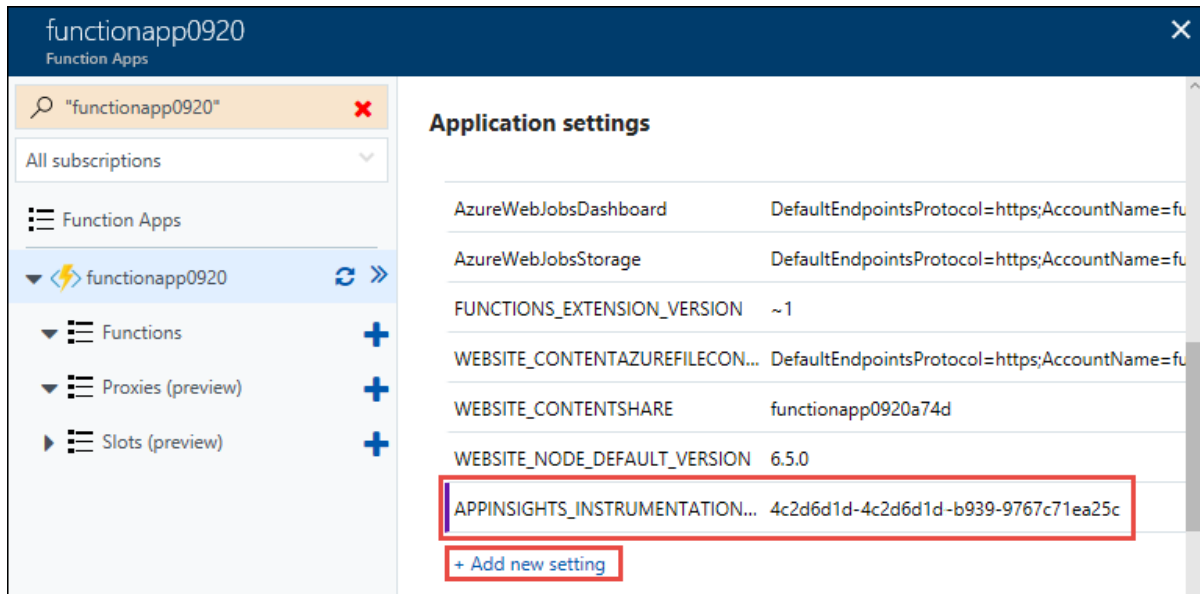
Search (Ctrl+/,) Metrics Explorer Analytics Time range Refresh More

Essentials ^

Resource group (change) functionapp0920	Type General
Location East US	Instrumentation Key aeb4ae60-3a15-4cee-b939-9767c71ea256
Subscription name (change) Windows Azure MSDN - Visual Studio Ulti...	
Subscription ID aeb4ae60-b7cb-aeb4ae60-fa43b6607f30	

Click to copy

3. In the function app's **Application settings** page, [add an app setting](#) named APPINSIGHTS_INSTRUMENTATIONKEY and paste the instrumentation key.



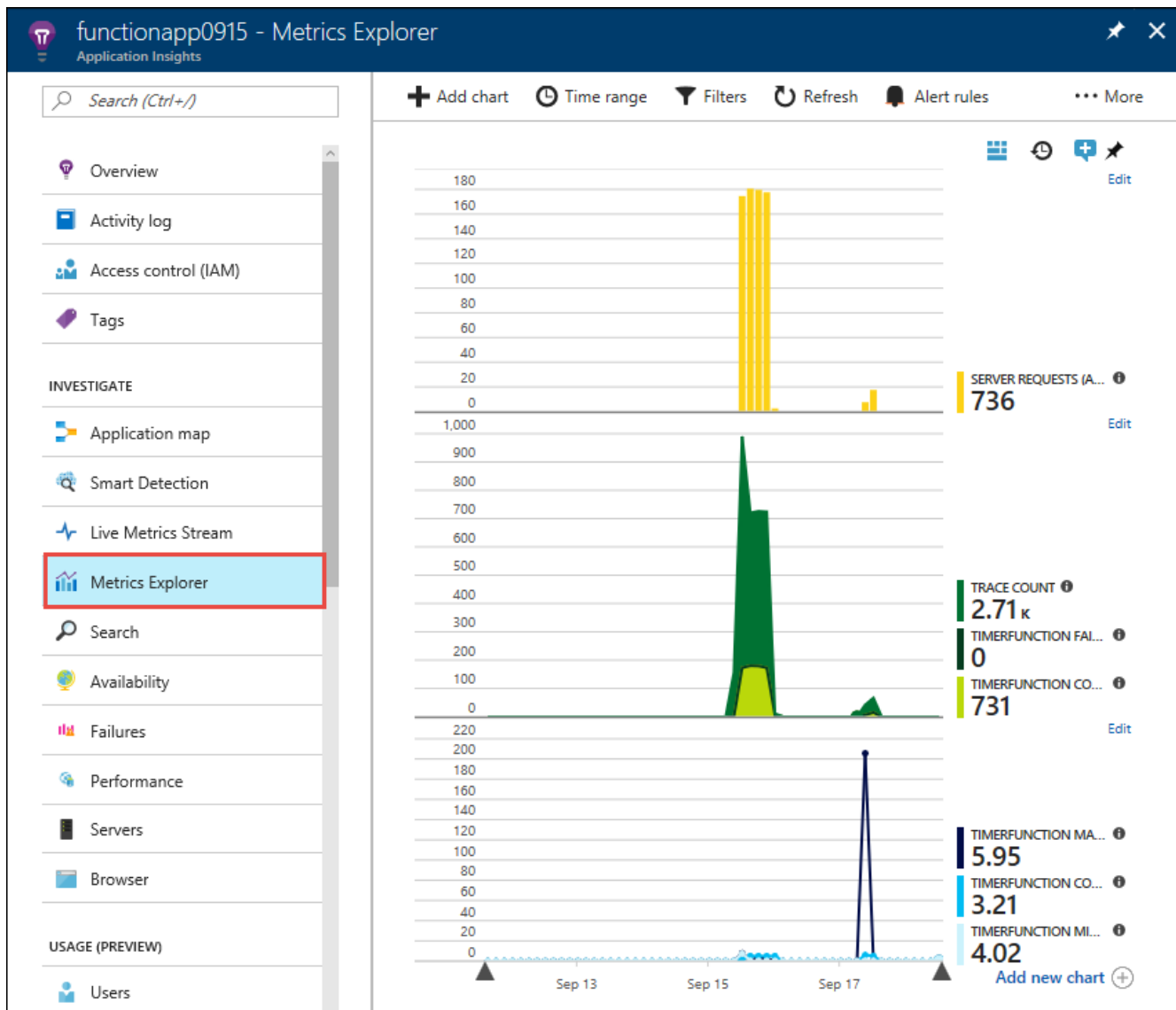
4. Click **Save**.

View telemetry data

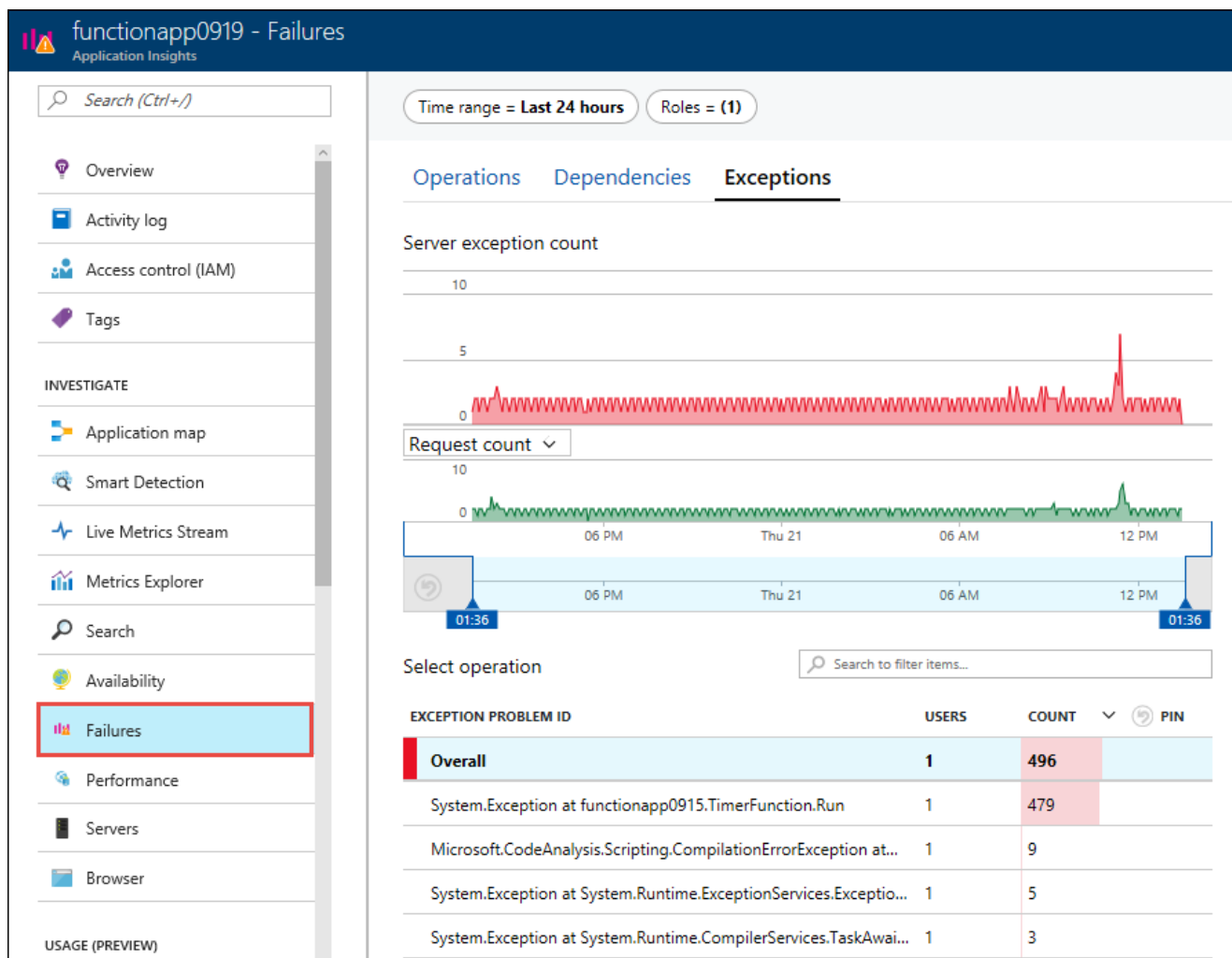
To navigate to Application Insights from a function app in the portal, select the **Application Insights** link on the function app's **Overview** page.

For information about how to use Application Insights, see the [Application Insights documentation](#). This section shows some examples of how to view data in Application Insights. If you are already familiar with Application Insights, you can go directly to [the sections about configuring and customizing the telemetry data](#).

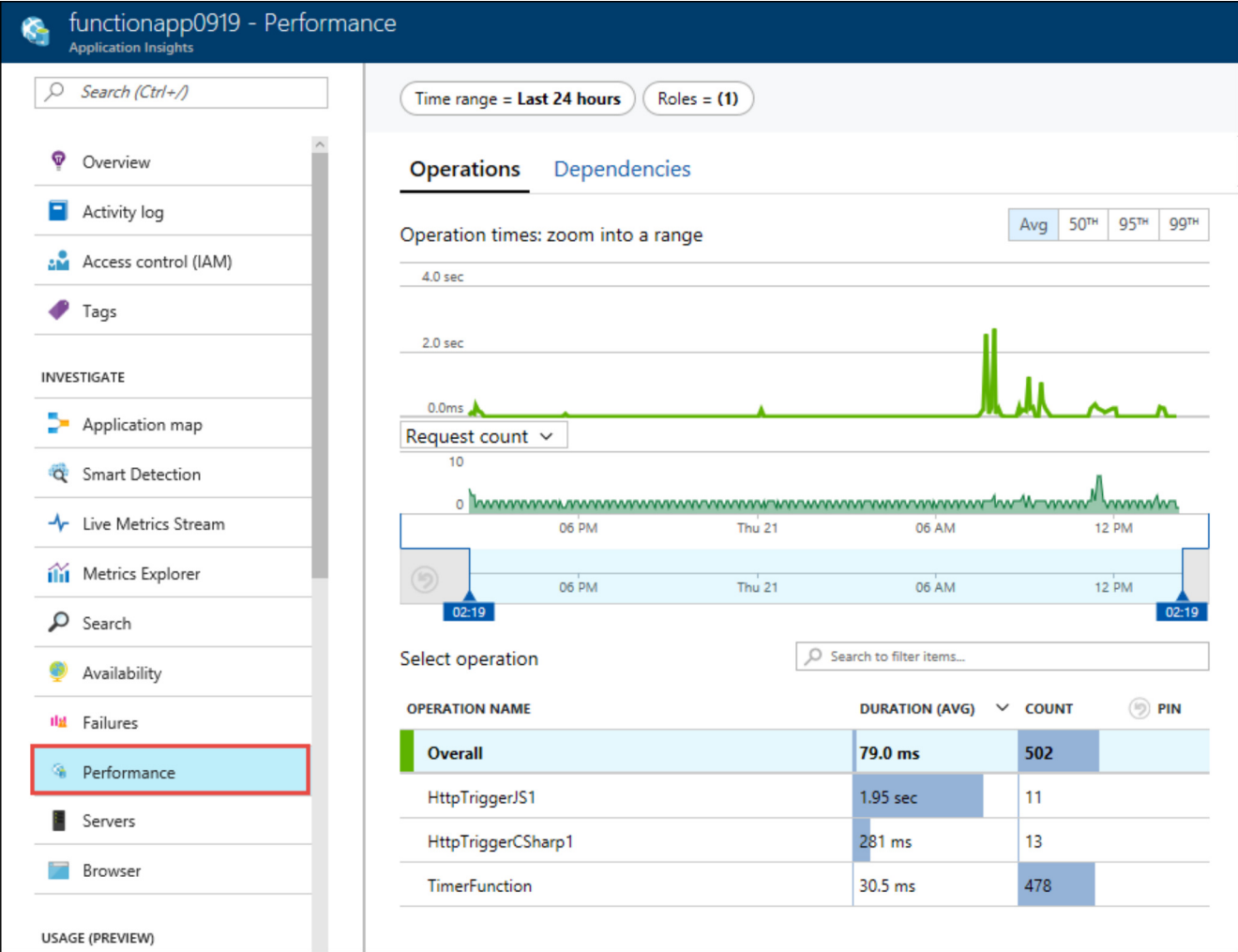
In [Metrics Explorer](#), you can create charts and alerts based on metrics such as number of function invocations, execution time, and success rate.



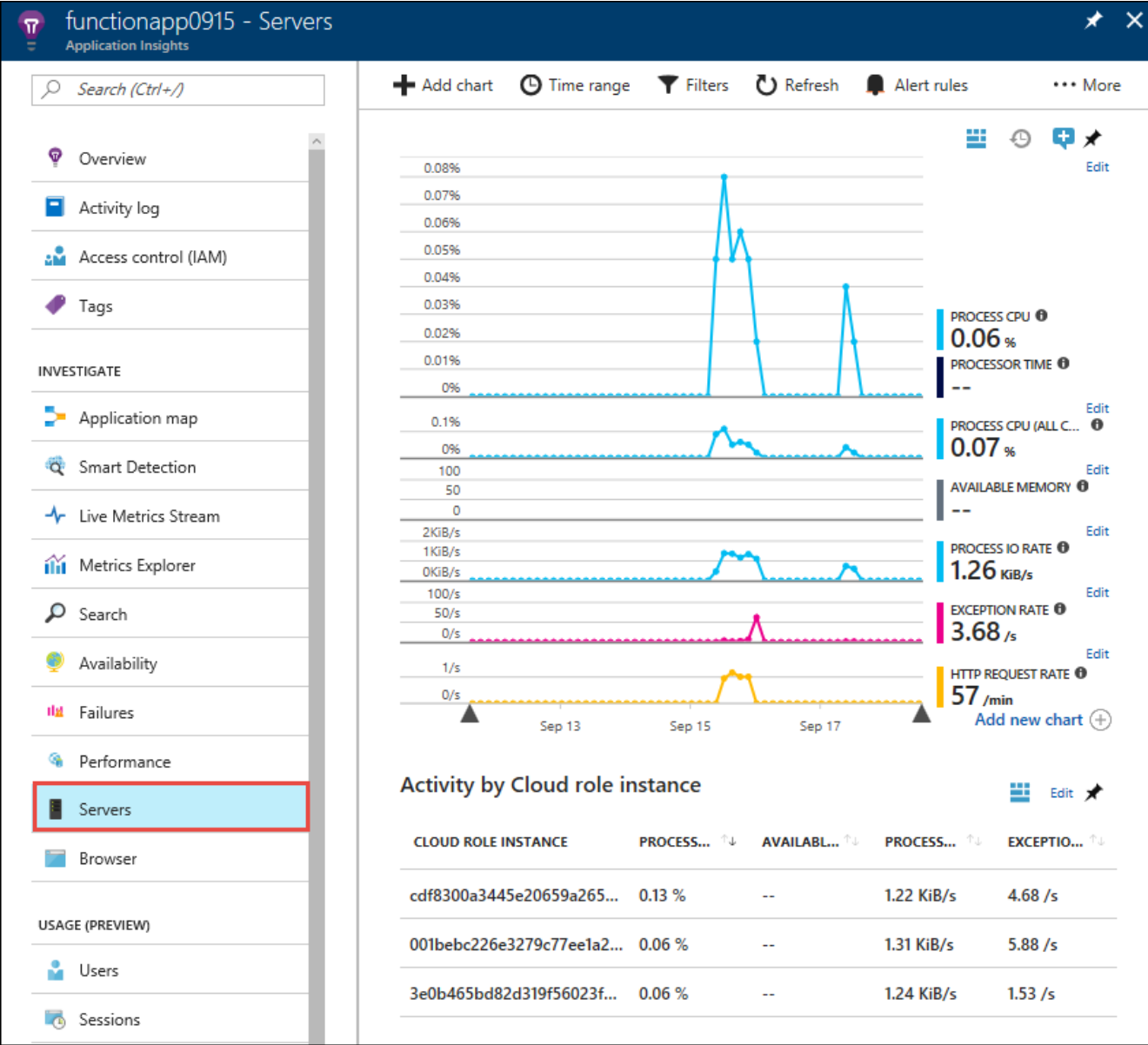
On the [Failures](#) tab, you can create charts and alerts based on function failures and server exceptions. The **Operation Name** is the function name. Failures in dependencies are not shown unless you implement [custom telemetry](#) for dependencies.

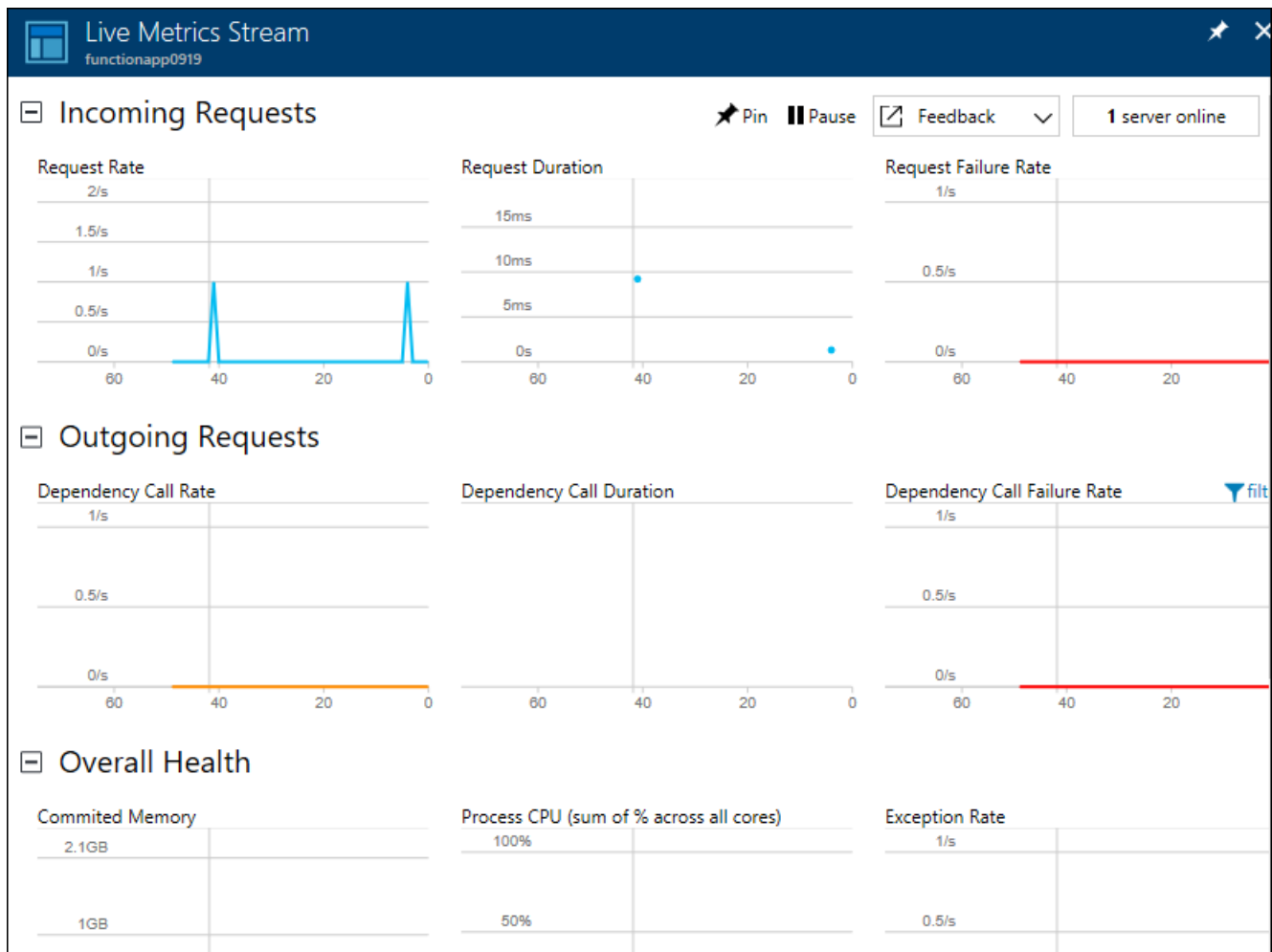


On the [Performance](#) tab, you can analyze performance issues.



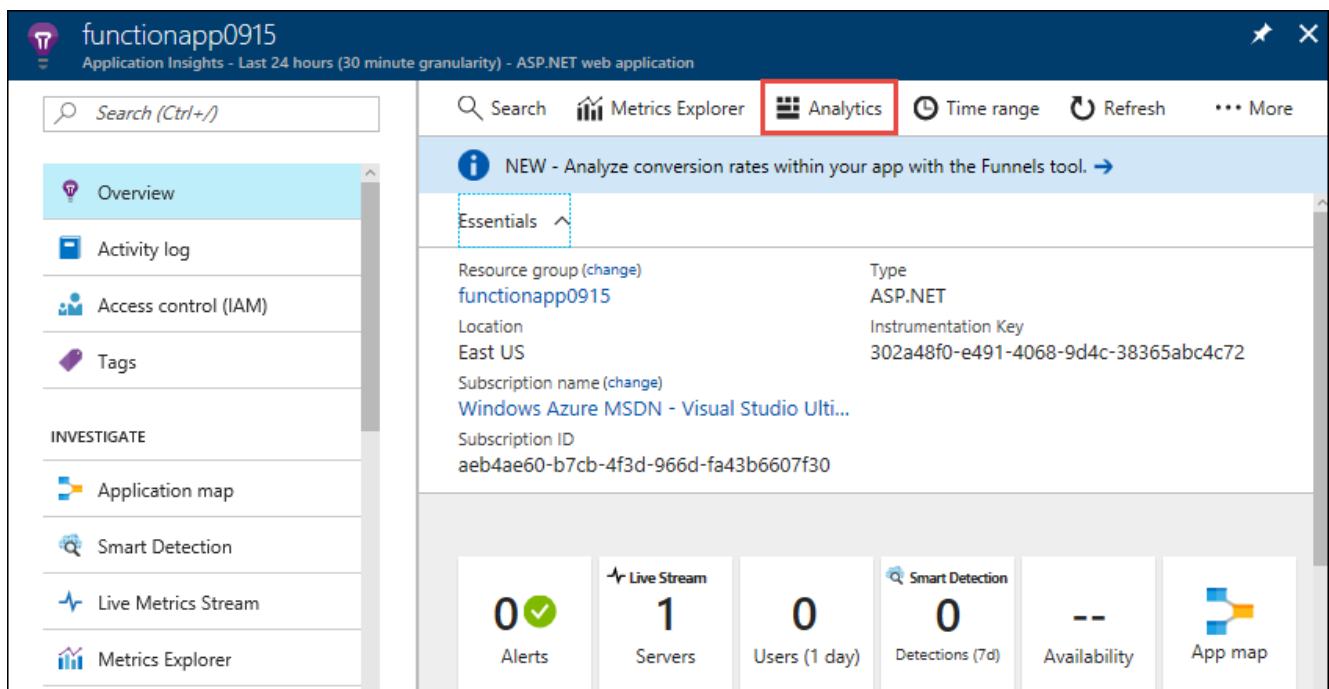
The **Servers** tab shows resource utilization and throughput per server. This data can be useful for debugging scenarios where functions are bogging down your underlying resources. Servers are referred to as *Cloud role instances*.





Query telemetry data

[Application Insights Analytics](#) gives you access to all of the telemetry data in the form of tables in a database. Analytics provides a query language for extracting and manipulating the data.



The screenshot shows the Azure Application Insights web interface. On the left, the 'SCHEMA' pane is active, showing a tree view of data sources under 'functionapp0915'. The 'traces' table is selected. The main pane displays the 'traces' table with columns: timestamp [UTC], message, and severityLevel. The table shows several log entries, including messages about host configuration, function status, and custom extensions.

timestamp [UTC]	message	severityLevel
2017-09-18T20:44:25.738	Reading host configuration file 'D:\home\site\wwwroo...	1
2017-09-18T20:44:25.738	Host configuration file read: { "logger": { "categor...	1
2017-09-18T20:44:26.114	Function 'TimerFunction' is disabled	1
2017-09-18T20:44:26.194	Loaded custom extension: EventGridExtensionConfig fr...	1
2017-09-18T20:44:26.194	Loaded custom extension: SendGridConfiguration from...	1
2017-09-18T20:44:26.194	Loaded custom extension: BotFrameworkConfiguration f...	1
2017-09-18T20:44:27.097	Generating 3 job function(s)	1
2017-09-18T20:44:27.146	Starting Host (HostId=functionapp0915, Version=1.0.1...	1
2017-09-18T20:44:27.334	Found the following functions: functionapp0915.HttpT...	1
2017-09-18T20:44:27.394	Host lock lease acquired by instance ID '78e1fb3ed5a...	1

Here's a query example. This one shows the distribution of requests per worker over the last 30 minutes.

Copy

```
requests
| where timestamp > ago(30m)
| summarize count() by cloud_RoleInstance, bin(timestamp, 1m)
| render timechart
```

The tables that are available are shown in the **Schema** tab of the left pane. You can find data generated by function invocations in the following tables:

- **traces** - Logs created by the runtime and by function code.
- **requests** - One for each function invocation.
- **exceptions** - Any exceptions thrown by the runtime.
- **customMetrics** - Count of successful and failing invocations, success rate, duration.
- **customEvents** - Events tracked by the runtime, for example: HTTP requests that trigger a function.
- **performanceCounters** - Info about the performance of the servers that the functions are running on.

The other tables are for availability tests and client/browser telemetry. You can implement custom telemetry to add data to them.

Within each table, some of the Functions-specific data is in a `customDimensions` field. For example, the following query retrieves all traces that have log level `Error`.

Copy

```
traces
| where customDimensions.LogLevel == "Error"
```

The runtime provides `customDimensions.LogLevel` and `customDimensions.Category`. You can provide additional fields in logs you write in your function code. See [Structured logging](#) later in this article.

Configure categories and log levels

You can use Application Insights without any custom configuration, but the default configuration can result in high volumes of data. If you're using a Visual Studio Azure subscription, you might hit your data cap for App Insights. The remainder of this article shows how to configure and customize the data that your functions send to Application Insights.

Categories

The Azure Functions logger includes a *category* for every log. The category indicates which part of the runtime code or your function code wrote the log.

The Functions runtime creates logs that have a category beginning with "Host". For example, the "function started," "function executed," and "function completed" logs have category "Host.Executor".

If you write logs in your function code, their category is "Function".

Log levels

The Azure functions logger also includes a *log level* with every log. `LogLevel` is an enumeration, and the integer code indicates relative importance:

LogLevel	Code
Trace	0
Debug	1

LogLevel	Code
Information	2
Warning	3
Error	4
Critical	5
None	6

Log level is explained in the next section.

Configure logging in host.json

The *host.json* file configures how much logging a function app sends to Application Insights. For each category, you indicate the minimum log level to send. Here's an example:

JSON Copy

```
{
  "logger": {
    "categoryFilter": {
      "defaultLevel": "Information",
      "categoryLevels": {
        "Host.Results": "Error",
        "Function": "Error",
        "Host.Aggregator": "Information"
      }
    }
  }
}
```

This example sets up the following rules:

1. For logs with category "Host.Results" or "Function", send only level and above to Application Insights. Logs for level and below are ignored.
2. For logs with category Host. Aggregator, send only level and above to Application Insights. Logs for level and below are ignored.
3. For all other logs, send only level and above to Application Insights.

The category value in *host.json* controls logging for all categories that begin with the same value. For example, "Host" in *host.json* controls logging for "Host.General", "Host.Executor", "Host.Results", and so forth.

If *host.json* includes multiple categories that start with the same string, the longer ones are matched first. For example, suppose you want everything from the runtime except "Host.Aggregator" to log at **Error** level, while "Host.Aggregator" logs at **Information** level:

JSON Copy

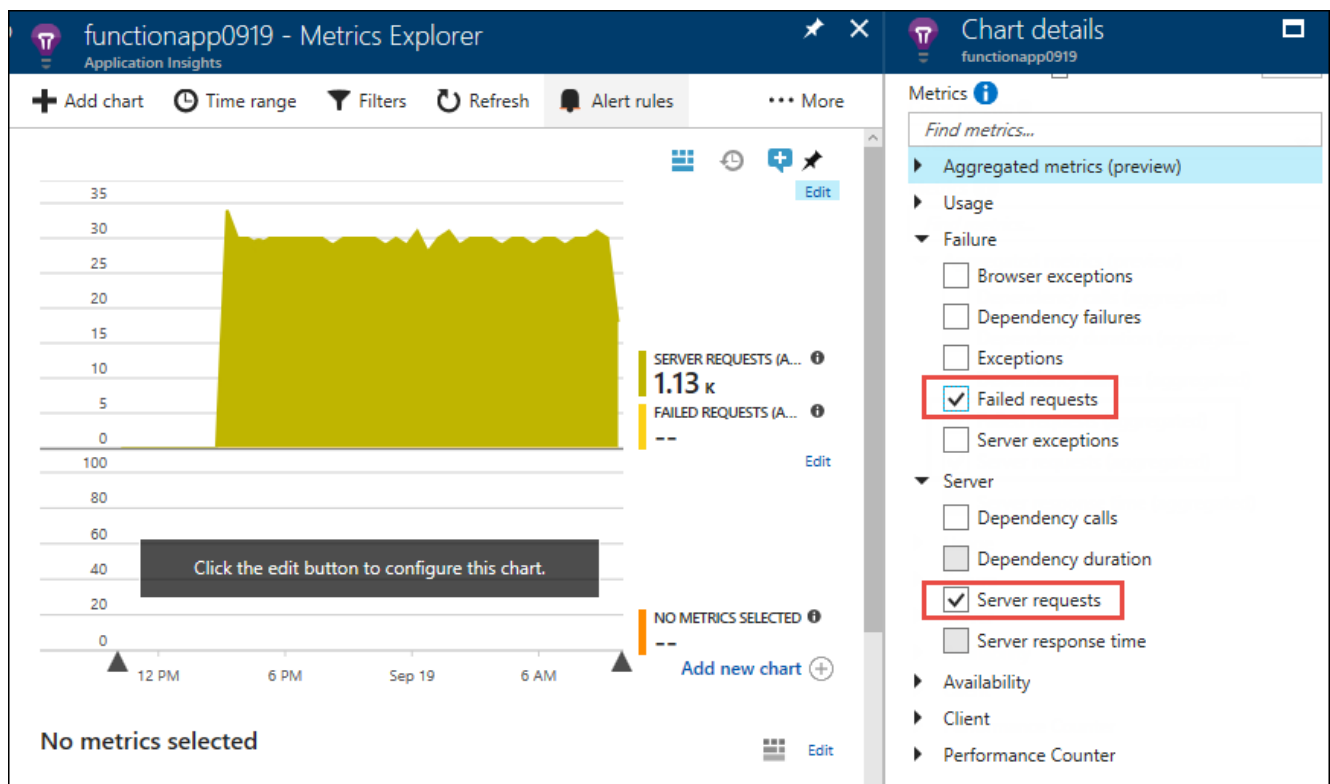
```
{
  "logger": {
    "categoryFilter": {
      "defaultLevel": "Information",
      "categoryLevels": {
        "Host": "Error",
        "Function": "Error",
        "Host.Aggregator": "Information"
      }
    }
  }
}
```

To suppress all logs for a category, you can use log level **None**. No logs are written with that category and there is no log level above it.

The following sections describe the main categories of logs that the runtime creates.

Category Host.Results

These logs show as "requests" in Application Insights. They indicate success or failure of a function.

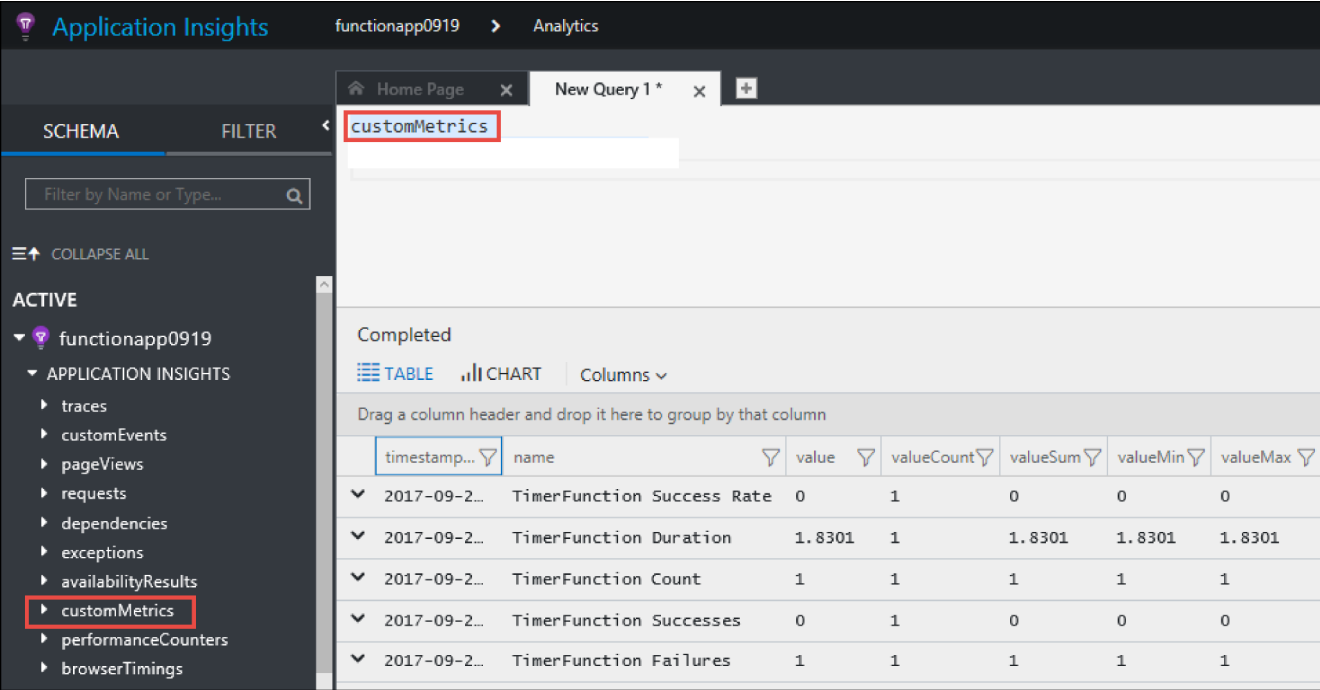


All of these logs are written at `Information` level, so if you filter at `Warning` or above, you won't see any of this data.

Category Host.Aggregator

These logs provide counts and averages of function invocations over a [configurable](#) period of time. The default period is 30 seconds or 1,000 results, whichever comes first.

The logs show as "customMetrics" in Application Insights. Examples are number of runs, success rate, and duration.



All of these logs are written at `Information` level, so if you filter at `Warning` or above, you won't see any of this data.

Other categories

All logs for categories other than the ones already listed show as "traces" in Application Insights.

Application Insights

functionapp0915 > Analytics

Home Page x New Query 1 * x +

SCHEMA FILTER

Filter by Name or Type...

COLLAPSE ALL

ACTIVE

- functionapp0915
 - APPLICATION INSIGHTS
 - traces
 - customEvents
 - pageViews
 - requests
 - dependencies
 - exceptions
 - availabilityResults
 - customMetrics
 - performanceCounters
 - browserTimings
 - OTHER DATA SOURCES
 - Add new data source

Completed. Showing results from the last 24 hours.

TABLE CHART Columns

Drag a column header and drop it here to group by that column

timestamp [UTC]	message	severityLevel
2017-09-18T20:44:25.738	Reading host configuration file 'D:\home\site\wwwroo...	1
2017-09-18T20:44:25.738	Host configuration file read: { "logger": { "categor...	1
2017-09-18T20:44:26.114	Function 'TimerFunction' is disabled	1
2017-09-18T20:44:26.194	Loaded custom extension: EventGridExtensionConfig fr...	1
2017-09-18T20:44:26.194	Loaded custom extension: SendGridConfiguration from...	1
2017-09-18T20:44:26.194	Loaded custom extension: BotFrameworkConfiguration f...	1
2017-09-18T20:44:27.097	Generating 3 job function(s)	1
2017-09-18T20:44:27.146	Starting Host (HostId=functionapp0915, Version=1.0.1...	1
2017-09-18T20:44:27.334	Found the following functions: functionapp0915.HttpT...	1
2017-09-18T20:44:27.394	Host lock lease acquired by instance ID '78e1fb3ed5a...	1

Page 1 of 1 50 items per page

All logs with categories that begin with "Host" are written by the Functions runtime. The "Function started" and "Function completed" logs have category "Host.Executor". For successful runs, these logs are **Information** level; exceptions are logged at **Error** level. The runtime also creates **Warning** level logs, for example: queue messages sent to the poison queue.

Logs written by your function code have category "Function" and may be any log level.

Configure the aggregator


As noted in the previous section, the runtime aggregates data about function executions over a period of time. The default period is 30 seconds or 1,000 runs, whichever comes first. You can configure this setting in the *host.json* file. Here's an example:

JSON Copy

```
{
  "aggregator": {
    "batchSize": 1000,
    "flushTimeout": "00:00:30"
  }
}
```


Configure sampling

Application Insights has a [sampling](#) feature that can protect you from producing too much telemetry data at times of peak load. When the number of telemetry items exceeds a specified rate, Application Insights starts to randomly ignore some of the incoming items. You can configure sampling in *host.json*. Here's an example:

JSON	
<pre>{ "applicationInsights": { "sampling": { "isEnabled": true, "maxTelemetryItemsPerSecond" : 5 } } }</pre>	


Write logs in C# functions

You can write logs in your function code that appear as traces in Application Insights.

ILogger

Use an [ILogger](#) parameter in your functions instead of a `TraceWriter` parameter. Logs created by using `TraceWriter` do go to Application Insights, but `ILogger` lets you do [structured logging](#).

With an `ILogger` object you call `Log<level>` [extension methods on ILogger](#) to create logs. For example, the following code writes `Information` logs with category "Function".

C#	
<pre>public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, ILogger logger) { logger.LogInformation("Request for item with key={itemKey}.", id); }</pre>	

Structured logging

The order of placeholders, not their names, determines which parameters are used in the log message. For example, suppose you have the following code:

C#

 Copy

```
string partitionKey = "partitionKey";  
string rowKey = "rowKey";  
logger.LogInformation("partitionKey={partitionKey}, rowKey={rowKey}", partitionKey, rowKey);
```

If you keep the same message string and reverse the order of the parameters, the resulting message text would have the values in the wrong places.

Placeholders are handled this way so that you can do structured logging. Application Insights stores the parameter name-value pairs in addition to the message string. The result is that the message arguments become fields that you can query on.

For example, if your logger method call looks like the previous example, you could query the field `customDimensions.prop__rowKey`. The prefix is added to ensure that there are no collisions between fields the runtime adds and fields your function code adds.

You can also query on the original message string by referencing the field

```
customDimensions.prop__{OriginalFormat}.
```

Here's a sample JSON representation of `customDimensions` data:

JSON

 Copy

```
{  
  customDimensions: {  
    "prop__{OriginalFormat}": "C# Queue trigger function processed: {message}",  
    "Category": "Function",  
    "LogLevel": "Information",  
    "prop__message": "c9519cbf-b1e6-4b9b-bf24-cb7d10b1bb89"  
  }  
}
```

Logging custom metrics

In C# script functions, you can use the `LogMetric` extension method on `ILogger` to create custom metrics in Application Insights. Here's a sample method call:

C#

 Copy

```
logger.LogMetric("TestMetric", 1234);
```

This code is an alternative to calling `TrackMetric` using [the Application Insights API for .NET](#).

Write logs in JavaScript functions

In Node.js functions, use `context.log` to write logs. Structured logging is not enabled.

Copy

```
context.log('JavaScript HTTP trigger function processed a request.' + context.invocation
```

Logging custom metrics

In Node.js functions, you can use the `context.log.metric` method to create custom metrics in Application Insights. Here's a sample method call:

JavaScript Copy

```
context.log.metric("TestMetric", 1234);
```

This code is an alternative to calling `trackMetric` using [the Node.js SDK for Application Insights](#).

Custom telemetry in C# functions

You can use the [Microsoft.ApplicationInsights](#) NuGet package to send custom telemetry data to Application Insights.

Here's an example of C# code that uses the [custom telemetry API](#). The example is for a .NET class library, but the Application Insights code is the same for C# script.

C# Copy

```
using System;
using System.Net;
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.Extensibility;
using Microsoft.Azure.WebJobs;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.Extensions.Logging;
using System.Linq;

namespace functionapp0915
{
    public static class HttpTrigger2
    {
```

```

private static string key = TelemetryConfiguration.Active.InstrumentationKey =
    System.Environment.GetEnvironmentVariable(
        "APPINSIGHTS_INSTRUMENTATIONKEY", EnvironmentVariableTarget.Process);

private static TelemetryClient telemetry =
    new TelemetryClient() { InstrumentationKey = key };

[FunctionName("HttpTrigger2")]
public static async Task<HttpResponseMessage> Run(
    [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post", Route = null)]
    HttpRequestMessage req, ExecutionContext context, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");
    DateTime start = DateTime.UtcNow;

    // parse query parameter
    string name = req.GetQueryNameValuePairs()
        .FirstOrDefault(q => string.Compare(q.Key, "name", true) == 0)
        .Value;

    // Get request body
    dynamic data = await req.Content.ReadAsAsync<object>();

    // Set name to query string or body data
    name = name ?? data?.name;

    telemetry.Context.Operation.Id = context.InvocationId.ToString();
    telemetry.Context.Operation.Name = "cs-http";
    if (!String.IsNullOrEmpty(name))
    {
        telemetry.Context.User.Id = name;
    }
    telemetry.TrackEvent("Function called");
    telemetry.TrackMetric("Test Metric", DateTime.Now.Millisecond);
    telemetry.TrackDependency("Test Dependency",
        "swapi.co/api/planets/1/",
        start, DateTime.UtcNow - start, true);

    return name == null
        ? req.CreateResponse(HttpStatusCode.BadRequest,
            "Please pass a name on the query string or in the request body")
        : req.CreateResponse(HttpStatusCode.OK, "Hello " + name);
}
}

```

Don't call `TrackRequest` or `StartOperation<RequestTelemetry>`, because you'll see duplicate requests for a function invocation. The Functions runtime automatically tracks requests.

Set `telemetry.Context.Operation.Id` to the invocation ID each time your function is started. This makes it possible to correlate all telemetry items for a given function invocation.

C#

 Copy

```
telemetry.Context.Operation.Id = context.InvocationId.ToString();
```

Custom telemetry in JavaScript functions

The [Application Insights Node.js SDK](#) is currently in beta. Here's some sample code that sends custom telemetry to Application Insights:

JavaScript

 Copy

```
const appInsights = require("applicationinsights");
appInsights.setup();
const client = appInsights.defaultClient;

module.exports = function (context, req) {
    context.log('JavaScript HTTP trigger function processed a request.');
```



```
    client.trackEvent({name: "my custom event", tagOverrides:{"ai.operation.id": context
    client.trackException({exception: new Error("handled exceptions can be logged with
    client.trackMetric({name: "custom metric", value: 3, tagOverrides:{"ai.operation.id
    client.trackTrace({message: "trace message", tagOverrides:{"ai.operation.id": conte
    client.trackDependency({target:"http://dbname", name:"select customers proc", data:
    client.trackRequest({name:"GET /customers", url:"http://myserver/customers", durati
```



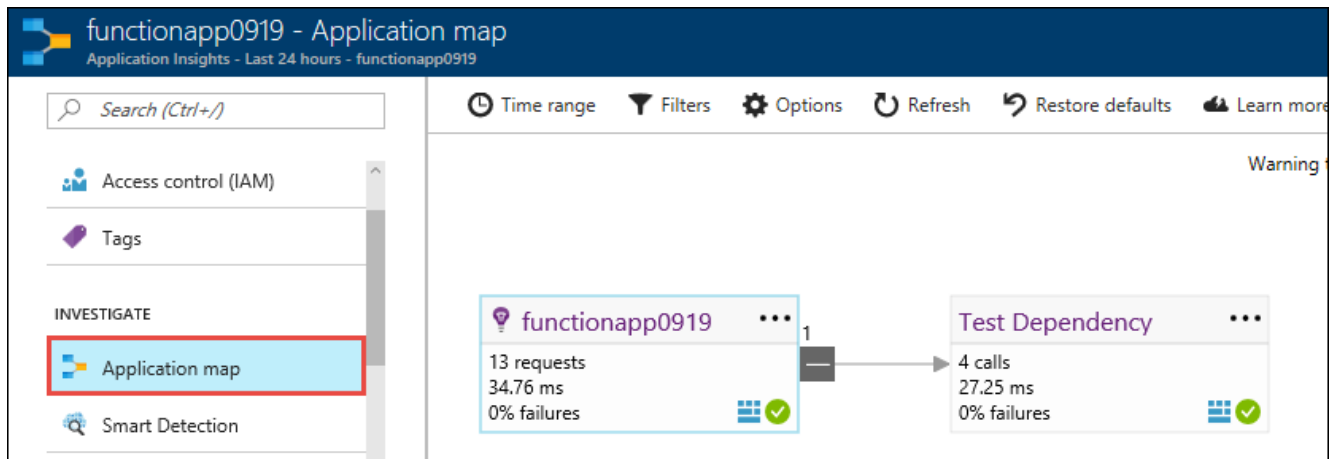
```
    if (req.query.name || (req.body && req.body.name)) {
        context.res = {
            // status: 200, /* Defaults to 200 */
            body: "Hello " + (req.query.name || req.body.name)
        };
    }
    else {
        context.res = {
            status: 400,
            body: "Please pass a name on the query string or in the request body"
        };
    }
    context.done();
};
```

The `tagOverrides` parameter sets `operation_Id` to the function's invocation ID. This setting enables you to correlate all of the automatically-generated and custom telemetry for a given function invocation.

Known issues

Dependencies

Dependencies don't show up automatically, but you can write custom code to show dependencies. The sample code in the [C# custom telemetry section](#) shows how. The sample code results in an *application map* in Application Insights that looks like this:



Report issues

To report an issue with Application Insights integration in Functions, or to make a suggestion or request, [create an issue in GitHub](#).

Monitoring without Application Insights

We recommend Application Insights for monitoring functions because it offers more data and better ways to analyze the data. But you can also find telemetry and logging data in the Azure portal pages for a Function app.

Select the **Monitor** tab for a function and you get a list of function executions. Select a function execution to review the duration, input data, errors, and associated log files.

ⓘ Important

When using the [Consumption hosting plan](#) for Azure Functions, the **Monitoring** tile in the Function App does not show any data. This is because the platform dynamically scales and manages compute instances for you. These metrics are not meaningful on a Consumption plan.

Real-time monitoring

Real-time monitoring is available by clicking **live event stream** on the function **Monitor** tab. The live event stream is displayed in a graph in a new tab in the browser

Note

There is a known issue that may cause your data to fail to be populated. You may need to close the browser tab containing the live event stream and then click **live event stream** again to allow it to properly populate your event stream data.

These statistics are real-time but the actual graphing of the execution data may have around 10 seconds of latency.

Monitor log files from a command line

You can stream log files to a command-line session on a local workstation using the Azure Command Line Interface (CLI) 1.0 or PowerShell.

Monitor function app log files with the Azure CLI 1.0

To get started, [install the Azure CLI 1.0](#) and [sign in to Azure](#).

Use the following commands to enable classic Service Management mode, choose your subscription, and stream log files:

 Copy

```
azure config mode asm
azure account list
azure account set <subscriptionNameOrId>
azure site log tail -v <function app name>
```

Monitor function app log files with PowerShell

To get started, [install and configure Azure PowerShell](#).

Use the following commands to add your Azure account, choose your subscription, and stream log files:

 Copy

```
PS C:\> Add-AzureAccount
PS C:\> Get-AzureSubscription
PS C:\> Get-AzureSubscription -SubscriptionName "MyFunctionAppSubscription" | Select-Az
PS C:\> Get-AzureWebSiteLog -Name MyFunctionApp -Tail
```

For more information, see [How to: Stream logs for web apps](#).