# Chapter 10: Operating and Monitoring the Release

Once our software changes are deployed and running in Production, we have just begun. Consider our model for DevOps that was introduced at the beginning of this book in the figure below. We must have a strategy for operations. Then we must execute that strategy consistently and measure to ensure what we expect is happening. Our DevOps cycle around the outside calls out learning as a feedback into planning for future changes. Through operating the software with real customers using it we can:

1. Verify that our customers can accomplish their goals
2. Learn what is the best change to make next

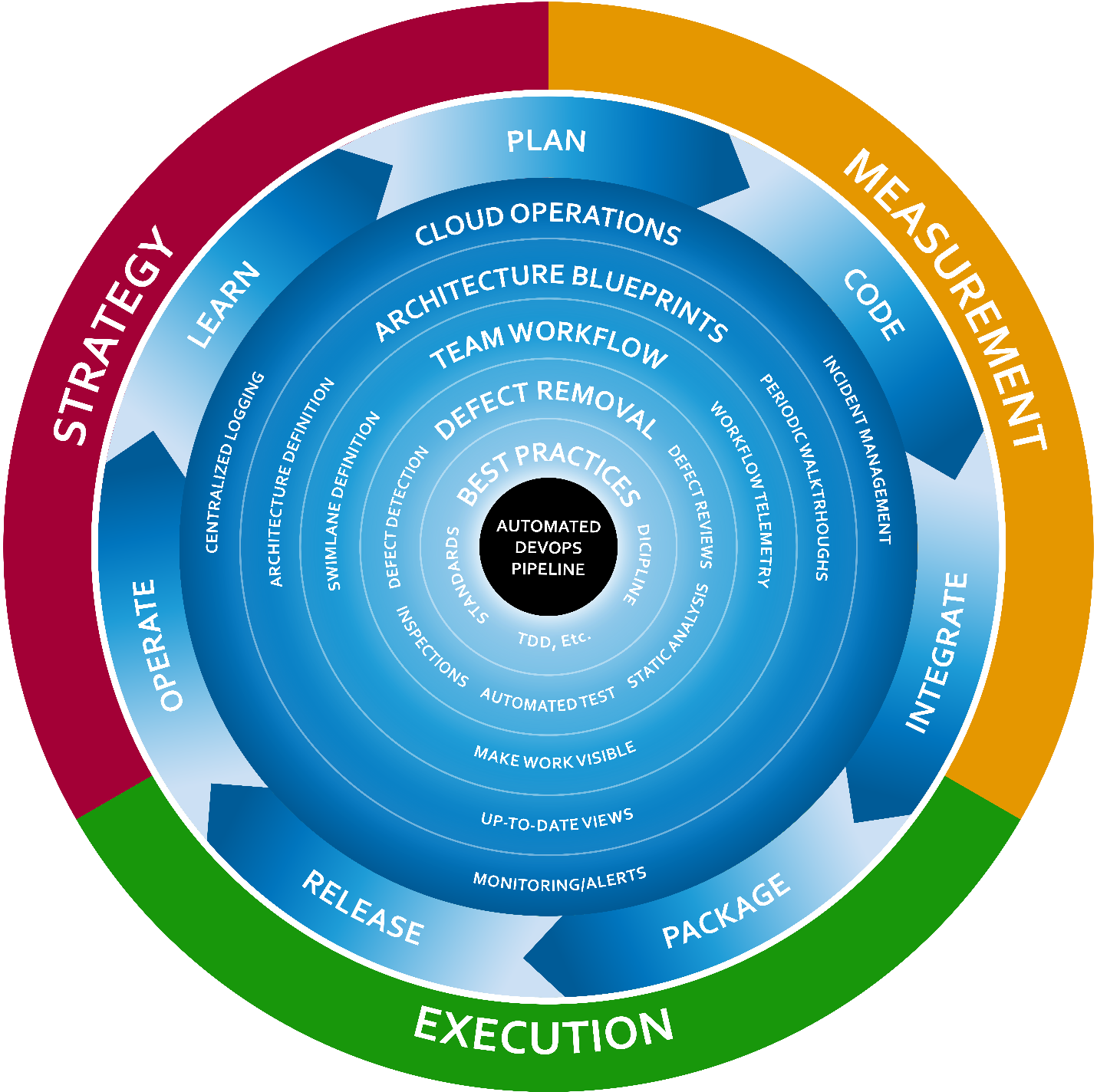


Figure: Onion DevOps Architecture provides a model for a complete DevOps environment.

In the cycle around the outside of the onion, the Release stage is just past the half-way point of the cycle. After releasing software to our customers, we must operate it well, learn how it’s performing, and then funnel that learning back into future plans. This chapter will cover the fundamentals for operating our .Net software in Azure.

## Principles

In discussions around quality and testing, a desirable trait of a software system has become known as **testability**. Software architects and engineers can have discussions about the design of a system and the components therein and evaluate the testability of the design. When operating a software system in Azure, or any environment for that matter, a desirable trait for the system is **observability**. Here are the principles.

* Know what your software is doing at all times.  
  It is not enough to know that a server is up, or that a web site is online. If any function of the system no longer functions as needed, then a customer is down. We should think in terms of our customers and their goals. If customers cannot do work, then the customer is down, even if the technical parts of our system are up. With that lens, we can ask ourselves “what do we need to know so that we are confident that our customers are up?”
* Listen to what your system is saying  
  Through the various types of telemetry you can have your software system emit, it speaks. Listen for what the system is asking for. Eric Hexter, a visionary in DevOps Diagnostics, related, when he presented to the Azure DevOps User Group[[1]](#footnote-1), that through telemetry, the system can ask for non-functional features or for maintenance. Through examining the logs and metrics, we can discover work that needs to be done on the system. This work might not show up in regular product backlogs.

Many teams discuss the level of logging that should be a part of the software. In addition, many already have some type of alerting in place. If trouble tickets or problem reports come in from customers, that is one sign that the **observability** sophistication of the system is lacking. Consider the work in this area an insurance policy. It does take investment of effort, time, and some money on products in order to achieve success in this area. This insurance policy has a premium that must be paid for the return on risk avoidance. If this is neglected and the insurance premium not paid, your organization will pay the full losses of a business disruption. This is another instance of the “Shift Left” way of thinking where we can design observability into the system that will yield better service to our customers.

Sam Guckenheimer has discussed observability and the importance of it on an Azure DevOps Podcast interview[[2]](#footnote-2)

## Architecture for observability

While this chapter cannot cover all techniques or forms that observability can take, we will focus on the basics that are universally application to what this author believes if >80% of .Net applications in the business world. Let’s start with the types of telemetry that should be emitted from your software.

* Metrics/performance counters  
  Many of these are built into the Azure platform, but if you are using queues, you will want to capture queue length, for example. Another useful metric is number of users by type currently using your system in the past hour/day. These types of trends can be used to trigger alarms. For example, if your normal usage drops off unexpectedly, you might be having a technical issue that needs to be investigated.
* Log messages/log files  
  While the most common type of telemetry, this is not consistently done. Every operation or transaction a system executes should be logged. Further, log files and log messages from various components should be aggregated and centralized into a repository that can be queried as a whole in order to provide a complete picture of what the system is doing.
* Heartbeats  
  Is it alive? Heartbeats can come from the outside or be built directly into application components themselves. These are signals and synthetic transactions that are built-in health checks. For example, if a critical integration is between the application and a payment processor, it is useful to know that the connection with the payment processor is functional. Integrations are notorious for breaking, and heartbeats that test these on a constant basis can alert us before a customer, frustrated, notifies us that a software feature is not working.

The following figure illustrates the architectural model for observability in Azure. While many products exist in the marketplace that can collect, aggregate, and search telemetry, we will focus on the capabilities built into Azure.



Figure: Each application should send telemetry and diagnostics information to a single Application Insights service.

Application Insights has the capability to gather a wealth of information from every running component of your application, including the DevOps pipeline itself, which is part of your system. One of the ways we improve the observability of our software is to collect all available information in the same place. Application Insights provides that. If you have never used Application Insights before, referred to also as AppInsights, Microsoft has a great overview in its documentation[[3]](#footnote-3).

As you have seen in our DevOps pipeline, we have a stable Production environment, one or more UAT environments, and a whole host of TDD environments that are constantly being created and destroyed. You will want the data in AppInsights to be durable across changes to environments. Consider the following environment architecture.



Figure: Each environment benefits from an AppInsights instance, which can then be aggregated to Log Analytics or other analytics sink.

By providing each environment type with an AppInsights instance, we can tune queries for each environment type to the audience. For example, our TDD environment will have environments come and go as new builds are produced. By capturing performance metrics while full system acceptance tests execute, we might be able to detect when runtimes of transactions change by more than a certain percent. This could be indicative of a performance slowdown from version to version. In addition, our Production AppInsights repository should have alerts configured on it.

The intent of AppInsights is for there to be one AppInsights service per application. While you can add custom tags to telemetry to facilitate filtering out environments in a single AppInsights instance, the service was not designed with that in mind. The service was designed to collect telemetry from a single application. That is, one versioned, complete unit of software. If you have broken your software into multiple git repositories with multiple DevOps pipelines, you can still use a single AppInsights service for the software system running in Production. To make this decision, you must ask yourself what kind of queries you will want to execute against the data and how you’d like the data to be segments. For our example application. We have multiple components that will send Telemetry to a single AppInsights instance for each environment. In total, we have three AppInsights services, one for each environment. Now that you can see the relationship between application components and AppInsights, let’s begin enabling observability in our software.

## Jumpstarting observability

While we are destroying and recreating TDD environments – and UAT environments from time to time – we want AppInsights to be durable. Because of this, we place the AppInsights services in a separate resource group from the environments that might be destroyed.



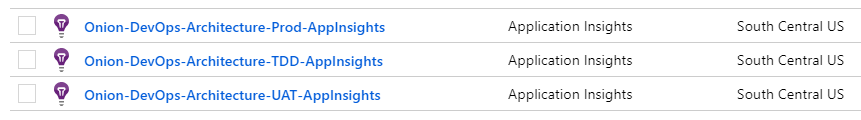


Figure: Each environment type has a dedicated AppInsights instance in the same region as the environment.

These services live in their own resource group “Onion-DevOps-Architecture-diagnostic”. You can name this resource group whatever you like. This resource group will live a long time, in contrast to your pre-production environments. Once we have our AppInsights in place, it is time to prepare the application for the sending of Telemetry. First, you will add the AppInsights Nuget packages to your projects in Visual Studio. ASP.NET projects have some quick start tooling to help with this, but you can add these packages to any projects you like. In our Visual Studio solution, we have kept the dependencies of the Core project very minimal – essentially this library contains plain old C# objects (POCOs). Because of these, we will choose the UI and Core.AppStartup projects to receive the AppInsights dependency.

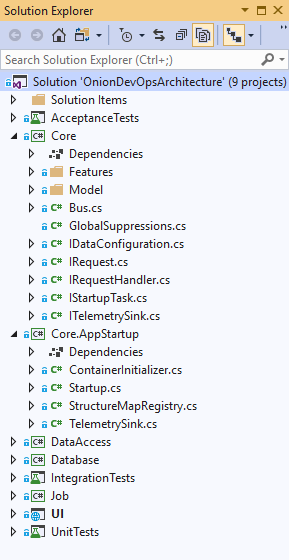


Figure: The Core project is just POCOs without dependencies, so AppInsights will be added to Core.AppStartup and UI.

The Nuget package to select is Microsoft.ApplicationInsights.AspNetCore. This package is appropriate for code running in Azure AppServices, including WebJobs and Azure Functions. There are other packages for software running on Windows VMs, but if your application is .Net Core, you just need this one.

Once you have the AppInsights Nuget package available to you, you will want to find an architecturally suitable place in your application to observe the transactions happening so that the information can be emitted to AppInsights. In this application, we already have an implementation of a bus patter whereby user intent is packaged as a C# message (object). Command and query objects are sent “down” the bus for execution. We can add some interception code in our TelemetrySink class here.

using Microsoft.ApplicationInsights;

using Microsoft.ApplicationInsights.DataContracts;

namespace ClearMeasure.OnionDevOpsArchitecture.Core.AppStartup

{

public class TelemetrySink : ITelemetrySink

{

public void RecordCall<TResponse>(IRequest<TResponse> request, TResponse response)

{

var telemetryClient = new TelemetryClient();

EventTelemetry telemetry = new EventTelemetry();

telemetry.Name = request.GetType().Name;

telemetryClient.TrackEvent(telemetry);

telemetryClient.TrackTrace(request.GetType().Name +

":- " + request.ToString(), SeverityLevel.Information);

}

}

}

If you have not yet explored the source code of this book’s example application, the following code is from the ExpenseReportController ASP.NET MVC controller action.

public IActionResult Index()

{

var command = new ListExpenseReportsCommand();

ExpenseReport[] reports = \_bus.Send(command);

var orderedReports = reports.OrderBy(report => report.Number);

return View(orderedReports.ToArray());

}

//..//

public class ListExpenseReportsCommand : IRequest<ExpenseReport[]>

{

}

The code in the user interface takes the request from the user and sends that down the bus. The request is to list the expense reports. This class implements the IRequest<T> interface. Our TelemetrySink has access to every request that flows through the application, and in a few lines of code, and records a trace for that request and a custom event. In your application, you might extract even more information about what the application is doing, the users performing the action, and other pertinent elements for querying later. In addition to this code, take care to ensure that the appsettings.json file does not send telemetry from a workstation to an environment’s AppInsight’s instance. In fact, take care not to accidentally commit to git any Instrumentation Key from the AppInsights resource. The appsettings.json file might look like the following.

{

"Logging": {

"LogLevel": {

"Default": "Debug"

}

},

"AllowedHosts": "\*",

"ApplicationInsights": {

"InstrumentationKey": "bogus value"

}

}

By putting a fake value in ApplicationInsights.InstrumentationKey, you will ensure that no telemetry can be sent from local developer workstations. Instead, you can try out and validate your telemetry by running the application in Debug (F5) mode.

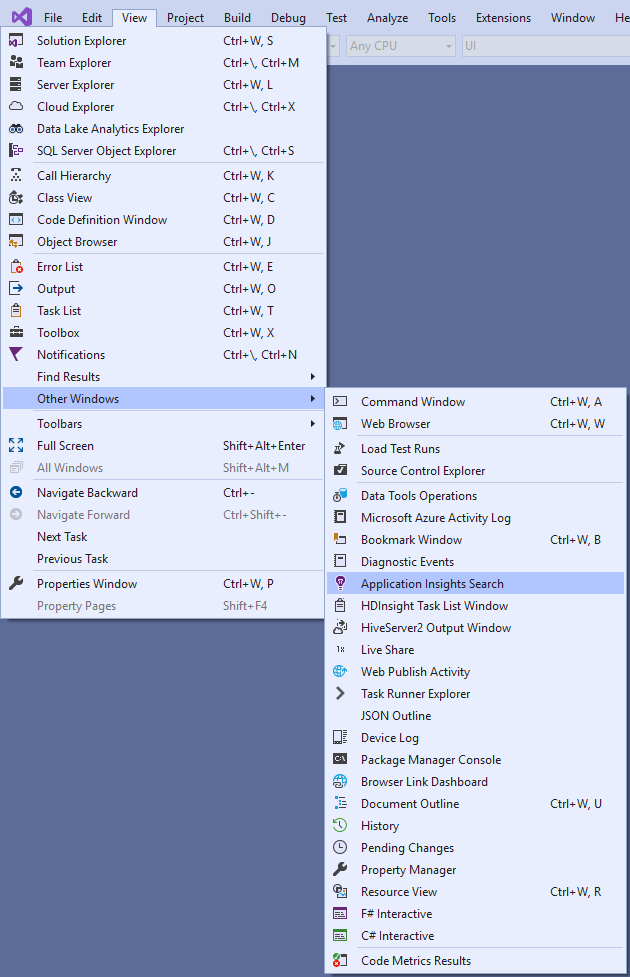


Figure: The AppInsights window can be used to see telemetry on a local workstation without connecting to Azure.

The Application Insights Search window can be a bit hard to find, but it allows you to develop your diagnostics capability locally without having to connect to Azure. You’ll want to make sure you are exporting the telemetry that you think you are. When you run the application in Debug mode, you can examine not only the telemetry that you add but also the phenomenal amount of data that is automatically captured for you.

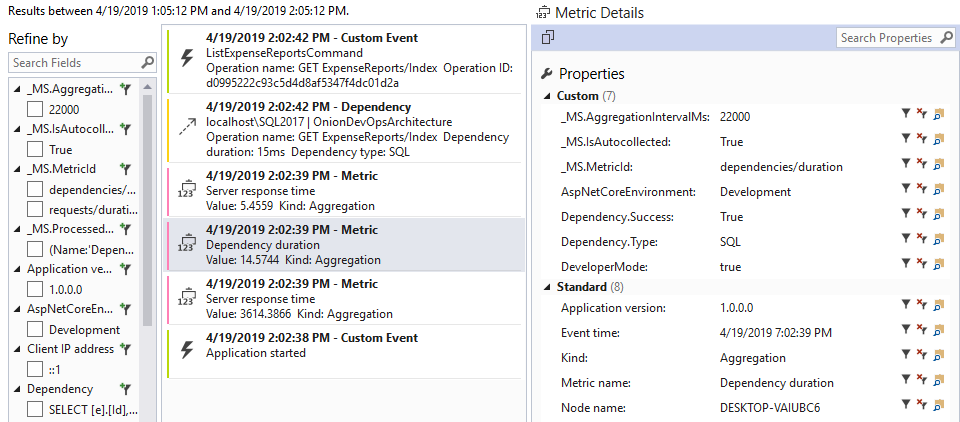


Figure: AppInsights captures common metrics and dependencies for you.

You added the Custom Event at the top of the above figure, but you did not add anything in order to have the SQL statement that was run from the application to the SQL Server database captured. This view allows you to search across any number of attributes to learn more about your application. To dig into more data, you can run your full suite of automated full system acceptance tests from the command line while capturing the telemetry in a Debug session.

dotnet vstest .\ClearMeasure.OnionDevOpsArchitecture.AcceptanceTests.dll

If you run the above command from the folder of the acceptance tests assemblies, you can amass quite a bit of telemetry to search. You will want to take a break for a cup of tea or coffee as you wait for your browser window to stop flashing across your screen as all the Selenium tests exercise your application as fast as it will go.

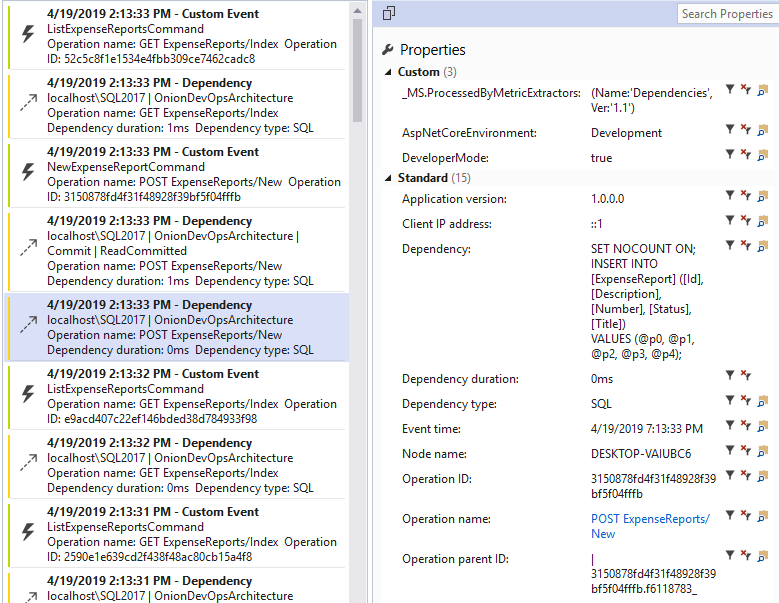


Figure: After running the acceptance tests, you can see all the telemetry captured in Application Insights.

It’s interesting to note that Application Insights does not automatically capture any data, parameters, or arguments. Therefore, you will have to add code to explicitly do that. When you do, take care that you are exporting any sensitive data field to a monitoring system that might have different data security controls that the production database. Then, once you are satisfied that you have a useful iteration of telemetry for your application, it’s time to link the application with the various environment specific AppInsights services in your Azure subscription.

Application Insights does not automatically capture any user data or SQL parameters. If you need parameters captured, you will add that directly, taking care of sensitive data fields.

For the execution of our deployment, it’s important to know when new release candidates were promoted from one environment to another. For this, you will add a Release Annotation task to your deployment steps.

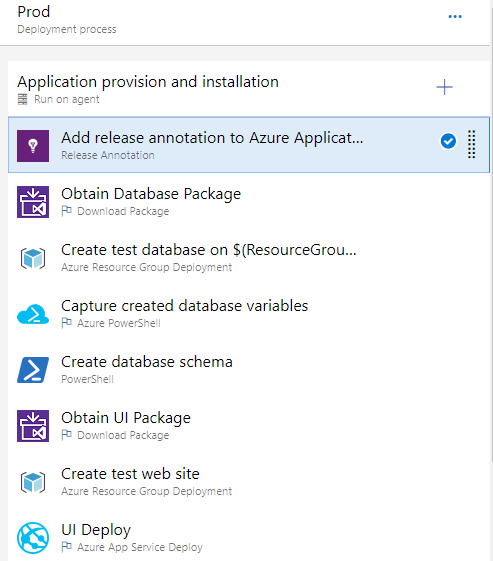


Figure: The Release Annotation task marks in AppInsights the deployment.

You will want to place this as the first task so that even if the deployment fails, Application Insights receives a marker that a deployment was started. In this way, AppInsights records that a version started deploying even if the deployment fails in the middle. At some later date when reviewing a period of exceptions and bad failures, you won’t have to strain your memory to remember that a bad deployment was associated with a period of elevated exceptions. Within the “UI Deploy” step, you will also add the appsettings.json file to the JSON variable substitution text area so that your variables will be evaluated for JSON substitution.

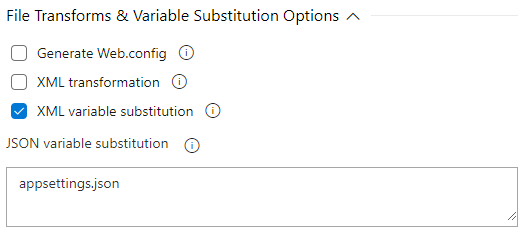


Figure: The AppInsights Instrumentation Key resides in the appsettings.json file.

Once this is configured, and after adding a properly named variable with the InstrumentationKey per environment, your release configuration is ready to deploy.



Figure: Each of the environments needs its own InstrumentationKey.

When your next release runs, you’ll be able to see release markers and telemetry from each of the environments. Beyond the Production observability you gain, it can also be useful to find non-functional defects like scalability issues. As your acceptance tests run, your release candidate receives its first full-system exercise.

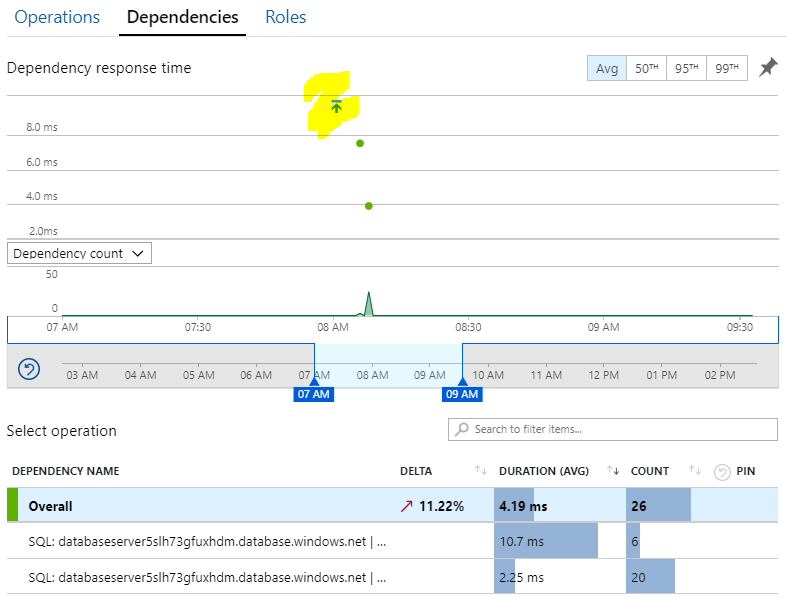


Figure: Our TDD environment’s AppInsights performance view now shows our release marker that is correlated with the run of the acceptance tests.

Our example application has only a handful of acceptance tests, so the data captured is only from those; however, when you have multiple feature branches executing releases in parallel, and you have multiple TDD environments created an operating at once, you’ll see all that here. The searches will be able to correlate versions.

Stamping and embedding version numbers in every .Net assembly in your application has downstream benefits and consequences. The telemetry in Application Insights records version numbers. Make sure the version number is available.

If you click on the release marker within the Azure Portal, you can see detailed information about the release that deployed the application that is responsible for the recorded telemetry.

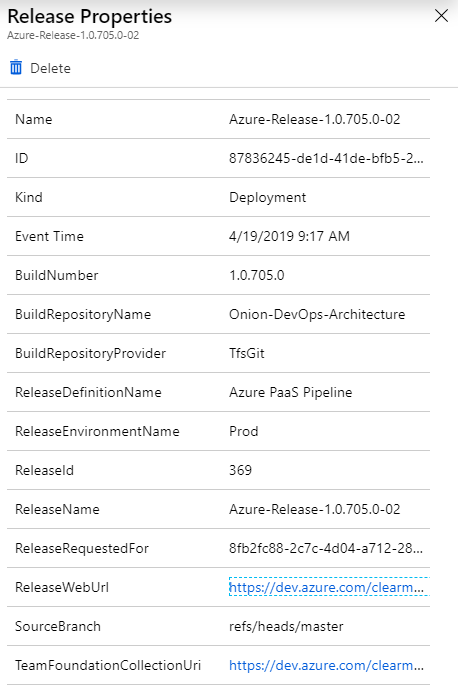


Figure: The release marker in AppInsights is recorded with a wealth of information about the release that deployed the application.

The release marker includes hyperlinks back to the release and the Azure DevOps project for the application that is emitting the telemetry. The BuildNumber, which originated in the continuous integration build is here with every bit of telemetry recorded in Application Insights. You have now connected code in Visual Studio, stored in Git with live captured usage data in Azure environments.

## Wrap up

You did it! You’ve been through the entire DevOps process on .Net for Azure. The term “DevOps Process” is discussed in some circles as if it is a simple thing. You can see from this text that there are numerous concepts, decisions, and steps that all go together in order to make the “DevOps Process” function. When put together properly, the team moves fast, can make changes when needed, and is able to operate the software well. You may wish to do some of the steps differently than the guidance in this book. If you already have a fully operational DevOps environment with at least all the capabilities illustrated in this book, then you are on your way to mastery. Implement your ideas. If you do not yet have a fully functioning DevOps environment with at least these capabilities, this author’s recommendation is to implement DevOps “by the book”. Then, once you can operate in very short cycles of code changes to stable deployments in a production environment, make your changes. Use the concept of ShuHaRi[[4]](#footnote-4) as you progress down your DevOps journey. Refer back frequently to the online public project that accompanies this book at https://dev.azure.com/clearmeasurelabs/Onion-DevOps-Architecture.

# Bibliography

Fowler, M. (n.d.). *ShuHaRi*. Retrieved from MartinFowler.com: https://www.martinfowler.com/bliki/ShuHaRi.html

Guckenheimer, S. (2018, 9 24). Sam Guckenheimer on Testing, Data Collection, and the State of DevOps Report. (J. Palermo, Interviewer) Retrieved from http://azuredevopspodcast.clear-measure.com/sam-guckenheimer-on-testing-data-collection-and-the-state-of-devops-report-episode-003

Hexter, E. (n.d.). *DevOps Diagnostics w/ Eric Hexter (Azure DevOps User Group).* Retrieved from https://www.youtube.com/watch?v=6O-17phQMJo

Microsoft. (n.d.). *What is Application Insights?* Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

1. (Hexter) [↑](#footnote-ref-1)
2. (Guckenheimer, 2018) [↑](#footnote-ref-2)
3. (Microsoft, n.d.) [↑](#footnote-ref-3)
4. (Fowler, n.d.) [↑](#footnote-ref-4)