THUNDRA™

Avoiding Pitfalls of Debugging Microservices

Contents

Abstract

Software application development is not all sunshine and roses. Most of the time, you have to deal with annoying bugs – developers are increasingly confronting problems, not only to locate where the problem is, but to find out the nature of the problem in order to quickly resolve it. In today's remote world, understanding if a bug is caused by your code or another team's code, or if some third party is slowing down your application, is almost impossible without having end-to-end observability in your application.

Traditional remote debugging solutions are not effective for today's distributed microservice architectures because they are slow, non-collaborative, and insecure. Additionally, the rich ecosystem of monitoring and observability tools is focused on resolving production issues, but they are not optimized for finding and detecting pre-production bugs. Most software application development teams apply their own solutions, such as logging for troubleshooting bugs in pre-production; however, the effectiveness of these methods is minimal.

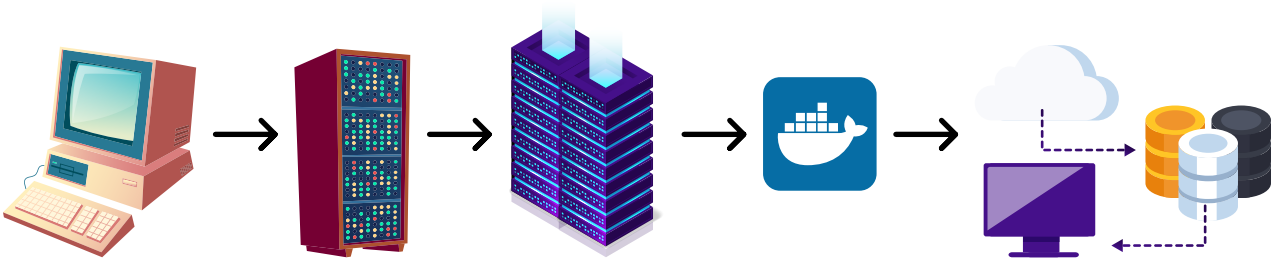
So what is the best way to debug remote applications in the new world's remote order without sacrificing the comfort of local development? Efficiency can only be obtained with two basic competencies attached to the mechanics of debugging: One, the breakpoints which you set to your code should not stop the application's execution, and two, the traces collected from different nodes of a microservice application in one transaction should be correlated.

Debugging Issues in Pre-Production

When it comes to debugging production issues, we have many opportunities because we are easily able to understand what's going on in production applications because the market is rich with many solutions, such as APMs or log management and log integration tools. But when it comes to debugging pre-production issues, it's hard to find suitable solutions in the market. For example, when something fails in the CI or a test fails when you want to push a PR, you are mostly alone with the CI logs and other logs coming from different streams.

You don't have the comfort and the toolset of troubleshooting pre-production issues like you do with production issues. Resolving production issues is not a comfortable and enjoyable task, indeed, but pre-production debugging is much more underserved.

TECHNOLOGY HAS EVOLVED



During the last couple of years, the tech industry has evolved quickly and drastically. We first swiftly shifted from good-old PC's to server farms, and then (thankfully) the cloud entered the scene. We started putting our workloads in containers, and then we started building complex distributed microservices that rely on many external resources. This technology is still evolving even now with the introduction of Kubernetes, serverless, and more.

However, with this new technology, it's now not as easy to debug your applications. Previously, when you wanted to understand what was going on with code running on your local PC, all you had to do was break the processes, stop the execution, and you were able to see exactly what was happening. But now, you don't even know where to have SSH connect to your applications because they run on "some cloud machine" – and you don't even know where it's located.

Most of the time, it's just not possible to know where your code is running. It's remote, it's encapsulated, and it relies on other microservices.

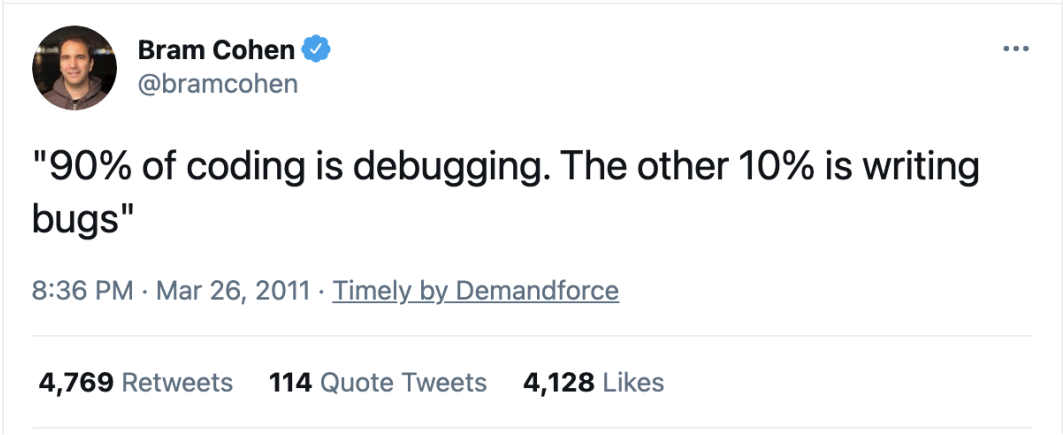
APPS BECAME MORE COMPLEX

As applications move to the cloud, the complexity of architectures is increasing. Even if you design the most simple system architecture, your application still relies on services provided by other cloud vendors (databases, queues, etc.), services provided by other vendors (Twilio, Stream, etc.), and services provided by other teams in the same organization.

Needless to say, it's crucial to determine the root cause of an issue ASAP when your customer-facing service is corrupted – and that means you have to understand your application behavior in all cases. In other words, developers need to ensure that they use external services in a resilient way.
When an issue occurs in your application, the initial step of troubleshooting is to determine which component is causing the problem. Is it your code or some other service?

Developers need to not only know how to find the problem, but to uncover what the problem is in order to quickly enact a solution. Bear in mind that interactions among all the components might be the cause of reliability problems – not just bugs in the application.

IT'S STILL VALID



10 years ago, Bram Cohen, the creator of BitTorrent, said that 90% of coding is debugging and the other 10% is writing bugs. This is still valid today.

Let's say, for example, you're building a project as a microservice written in Node runtime. But on the other hand, another developer in your organization is building a separate microservice written in Java runtime, which is hosted on a different cloud than your microservice and uses a different orchestration solution. When your application needs to connect to your colleague's application to do a specific job, you need to understand both applications at the same time (which is almost impossible). Why do you need to understand both applications? Because when something crashes, or a response takes more time than expected, you have to understand the root cause of the issue to debug and troubleshoot the problem – and if the issue is on the other developer's end, you need to know how to find and fix the error so your application isn't affected.

In addition to that, debugging code is clearly about fixing issues, but the typical cycle to resolve an issue in cloud applications is a lot different than how we used to do it. It consists of several steps, such as creating a new build, deploying it to the cloud, and verifying the accuracy of the correction. This can be a very time-consuming and daunting process.

Development Environments

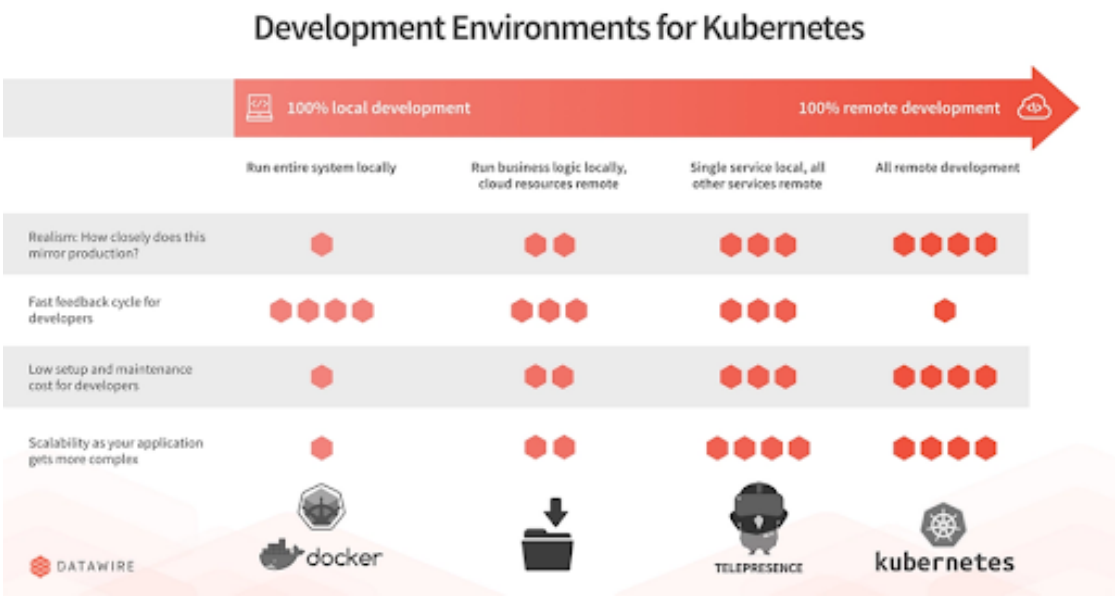
ALTERNATIVES TO DEVELOPING MICROSERVICES

Theoretically, there are three basic alternative ways that you can develop microservices in the cloud:

1. You can either spin up the full system in a local environment.

2. You can spin up the full system in the cloud.

3. You can employ multiple methods of spinning up the system and create a hybrid alternative.



Let's dive a little bit deeper into these methods. The most ancient (and perhaps the most problematic) way to develop microservices is by spinning up everything locally. You can develop your applications using local replicas of cloud resources: There are very useful tools in the market to replicate any cloud resources in your local environment, but this comes with some problems, as you may already know.

When you are working with local replicas, you may not be sure if the local replica reflects the latest version of the cloud service that you are currently using. So, the problem is that you may lose your focus on developing your business logic in order to maintain and update all the local replicas of external apps you have mocked up.

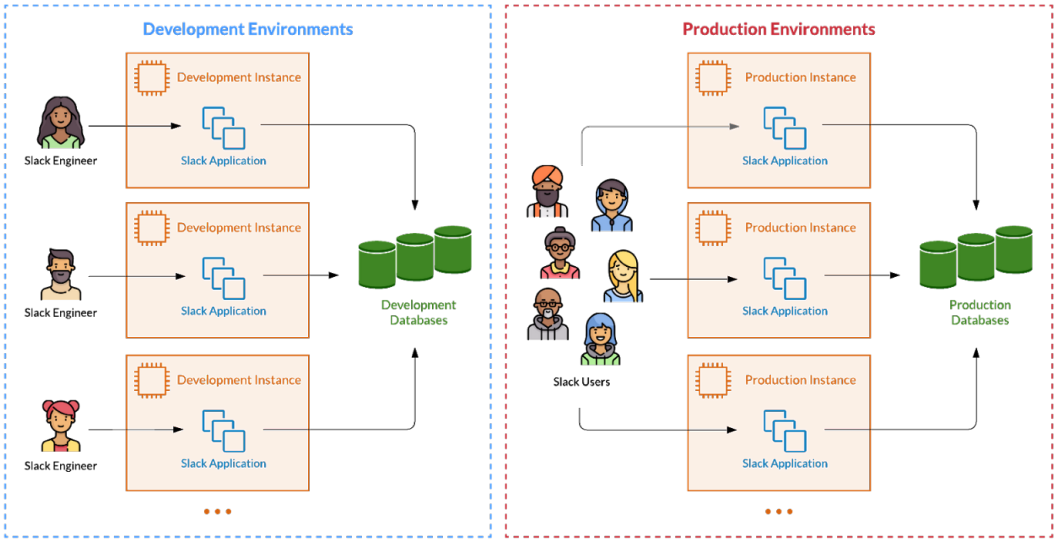
The other alternative is that you can spin up everything on the cloud. Using this method, your development environment would be very realistic, and you can focus more deeply on your code and business logic.

And the third alternative is that you can spin up everything on the cloud, but additionally, you can tunnel into cloud services via ports while you are developing your applications in your local environment.

If your applications are going to be served from the cloud, then the most reasonable alternative to building your development environment and creating microservice cloud applications may be spinning up everything in the cloud and shifting development into the cloud as much as possible. This method ensures that you are developing your applications against real cloud resources.

A REAL-WORLD EXAMPLE OF REMOTE DEVELOPMENT

Below you can see a great example of how remote development can be applied in real-world applications.



In this article, the Slack engineering team explains how they actually set up development environments for their engineers. They have different cloud environments for each and every developer, and every developer can use the cloud version of the Slack application.

They are essentially using a replica of the identical Slack application, which is very close to their production version. They are working on it, starting their branches, making changes, testing their changes, debugging their code on these remote apps, and taking advantage of the fully remote development without actually interfering with each other's environments.

What Is the Downside of Local Debugging?

THE COMFORT OF LOCAL DEVELOPMENT

Many developers don't want to leave the comfort of their local environment while developing code because it feels like home. But despite that, many organizations are moving to the cloud and modernizing their applications at a faster pace than before.

Distributed applications, microservices, containers, and serverless technologies are taking the place of the old, bulky monolithic structures. Developers are playing a major role in this shift, and they are being asked to do more. Microservices help to isolate different modules and their functions. This isolation provides many benefits, but it also makes it harder to track bugs and errors.

But despite these advances, debugging techniques haven't changed for a long time. We still develop in local environments and utilize either print debugging or log debugging. In today's cloud-native world and the wide adoption of cloud technologies and development environments, debugging techniques need to change to keep up with the times.

WHAT IS THE DOWNSIDE?

As a rough assumption, when you have simple code, you spend at least 40% of your time debugging, refactoring, and modifying your code. If you're developing complex architectures with several components or upgrading an existing application to add more features on top of it, then it's very easy to spend up to 90% of your time debugging, refactoring, and modifying your code.

The process of observing the code you are debugging is very tedious if you are developing a cloud application locally. Ironically, you have to start by writing more code to understand your code, because you don't know where it's running or how to connect to it.

You have to add more and more log lines or SDK calls, and then you have to go through tests.
Then, you have to go through an approval process and deployment (for example, through CI/CD). Finally, at the end of this long process, the data you receive is likely to be something completely different than what you actually need.

You end up "rinsing and repeating" over and over again until you find something meaningful. In a best-case scenario, each iteration can take up to 15 minutes; in a worst-case scenario, it can take an hour or more. As a result, when you have to do a lot of work, you basically end up wasting a lot of time just to iterate on your own software to understand its behavior.

The Challenges of Traditional Remote Debugging

WITH REMOTE DEVELOPMENT, LOCAL DEBUGGING MAKES (ALMOST) NO SENSE...

With Remote Development, Local Debugging Makes (Almost) No Sense...
Even if your development environment is local, your test/staging environment is probably in the cloud if your application lives in the cloud.

If you want to do local debugging, you basically can't see anything of value because all of the application is running on some other machine. That means it's not actually possible to debug this application by just pausing, changing the code, and using local debugging.

In this case, "traditional remote debugging" might make sense for you – almost all IDEs, and many other tools, offer remote debugging solutions. The current remote debugging solutions let you debug an application that isn't on your machine but is instead running on another remote machine.

...BUT NEITHER DOES REMOTE DEBUGGING

The idea of using traditional remote debugging solutions sounds like a pain reliever, but there are some underlying issues with using a remote debugger in a production
or even a pre-production environment.

Using traditional remote debuggers is a good way of understanding the behavior of your remote applications running on the cloud, but it's not the best way of debugging today's modern architectures. It solves some debugging problems, but at the same time, it creates other major problems.

WHAT IS REMOTE DEBUGGING?

Simply put, remote debugging allows you to debug someone else's program while it's running on another machine. This program could even be your own, but the point is that you are debugging a program or application that isn't running on your local machine. You are making a port connection to some other machine and then stopping, pausing, and playing that application.

This means that the application should keep a port open for the debugger application to interfere with its flow. However, this also means that if there is an open port, it is open to everybody – and this may create a security hole in your system. With this open port, some other application other than your debugger can use this port and create any kind of damage that you can imagine to your application.

Additionally, remote debugging lets you debug one service at a time. Let's say you have several microservices working together and one of your colleagues is connected to one of the microservices. If you want to connect and debug that application, you simply cannot because your colleague might have stopped it for debugging purposes.

With all that in mind, that means the “traditional” remote debugging solutions are not suitable for debugging remote microservices applications. As another example, let's say you have a microservice architecture and one of your colleagues deployed the service. You are sharing the applications with them. When you have shared applications in development environments, you may be pausing your colleagues’ applications while you are trying to remotely debug them.

Additionally, should you have a very distributed architecture, using a traditional remote debugging solution can cause you to miss important details since you can only debug one server at a time. This limitation almost ensures that you won’t be able to see the full picture of the transactions flowing through your microservices.

Wrapping up: The traditional remote debugging solutions make you a one-man show because you are just debugging one service at a time and you are by yourself. You are not able to collaborate with your colleagues because the breakpoints are intrusive. And you are most likely breaking security compliances by creating potential security holes in applications.

Poor Alternatives of Remote Debugging

Since we know that local debugging isn't very effective, and that traditional remote debugging solutions aren't very efficient, that leaves us with our conundrum. What might be the most feasible solutions to remotely debug our applications?

Every software development team has its own different methods of remote debugging. Let's discuss two of the most common methodologies.

USING LOGS

The first most commonly known and implemented solution is using logs. Let's say that your remote applications are generating log streams that flow between several resources. In an average case, if you have four services, that means you have four different log streams. If you also have log streams for your non-compute resources, then you'll have at least five or six (or maybe more) log streams in total. In order to debug such a system, you need to check all of these log streams and correlate them to be able to understand the state of a transaction.

There are many open-source alternatives to doing this with tracing, but the harsh part of these options is that you need to implement those solutions. For example, when you need to implement a handler that is suitable to work with open telemetry or open tracing, you also need to implement the correlation yourself.

Additionally, cost is a concern for anybody who keeps logs. Most of the time when you examine the cost of applications, the biggest cost item is not compute resources, but is instead the logs and the database. Costs can really skyrocket when your application starts to get more and more traffic.

Lastly, when you look at logs to debug the application and don't understand anything, you will generally need to write some new logs to make it more understandable. This means that you need to deploy the application again and again, which creates very long development cycles – all to find a bug.

With all of these problems, you can see that using logs is a very time-consuming, very ineffective methodology to debug remote microservices applications.

USING APMS OUT OF THEIR PURPOSES

The other methodology to debug remote applications in pre-production is using APMs out of their purposes.

So here's the scenario: You're developing an application and you don't understand what's going on in your remote development environment. Then you remember you have an APM that you use for post-production issues. You can use your APM solution to see what's going on with your application while you're developing it. While this can work in most cases, it does have some issues.

WHY DO APMS STAY SHORT FOR DEBUGGING?

First of all, you cannot replace the mechanics of debugging with APM solutions. Let's assume that you are examining an event that happened one week ago, trying to debug the application so you can understand what went wrong by checking your APM solution. It's not possible to pause an event that ran in the past. You also cannot change the code, go step by step over the code, and etc.

APM's are not optimized to solve pre-production issues, which means they aren't very informative about distinguishing the requests coming from your customers or requests coming from your infrastructure. Also, the cost is another component to consider carefully. If you just want to debug a single application that is generating a high amount of data, then it's very likely that your costs can increase drastically with your APM.

What Is Non-intrusive Debugging?

NON-INTRUSIVE DEBUGGING WITH BREAKPOINTS

Non-intrusive debugging with non-breaking breakpoints means that your application is running in the cloud and you are able to debug it just like you would for a local application. You can set breakpoints, but these breakpoints don't necessarily stop the execution of the application.

This is very useful for production environments, but it can also be useful in pre-production environments as well. Multiple developers can work on the same remote application without interfering with each other while debugging.

The non-intrusive breakpoints are supposed to take snapshots of variables during execution and let the execution flow. The time spent taking a snapshot should be less than 20 - 50 microseconds, which for most applications is considered a negligible amount of overhead.

After you receive the data, you should be able to send it asynchronously to any type of collector to make your research more comprehensive.

Is Non-intrusive Debugging Sufficient to Debug Distributed Systems?

CONNECTING BREAKPOINTS ASYNCHRONOUSLY

A remote debugger is not completely sufficient if it only uses non-intrusive breakpoints and takes snapshots of the current state of the application. There is still a missing component when you remotely debug distributed microservice architectures.

Let's assume that you have a distributed application. Service A takes a request from a customer, processes the request, and writes the result to a queue. Then service B consumes this result and writes it to a database.

If you want to debug this flow, you normally put a breakpoint somewhere in the code of service A and also service B. You now have two breakpoints, but in different applications. How are you going to integrate these events of the breakpoints of the same transaction?

Distributed tracing is the solution that can help connect the dots. By implementing distributed tracing in your system, you can navigate or step into a breakpoint, or just go to the next breakpoint in the same transaction – as if you were actually debugging a local application.

You can build your own distributed tracing using open-source platforms that have very nice standards of distributed tracing, such as open telemetry and open tracing. But this can be a costly solution if you're only using it for remote debugging purposes. Additionally, you may need to own, maintain, and solve problems on your own. As a result, this can actually take your attention away from the problem that you are trying to solve for your customers.

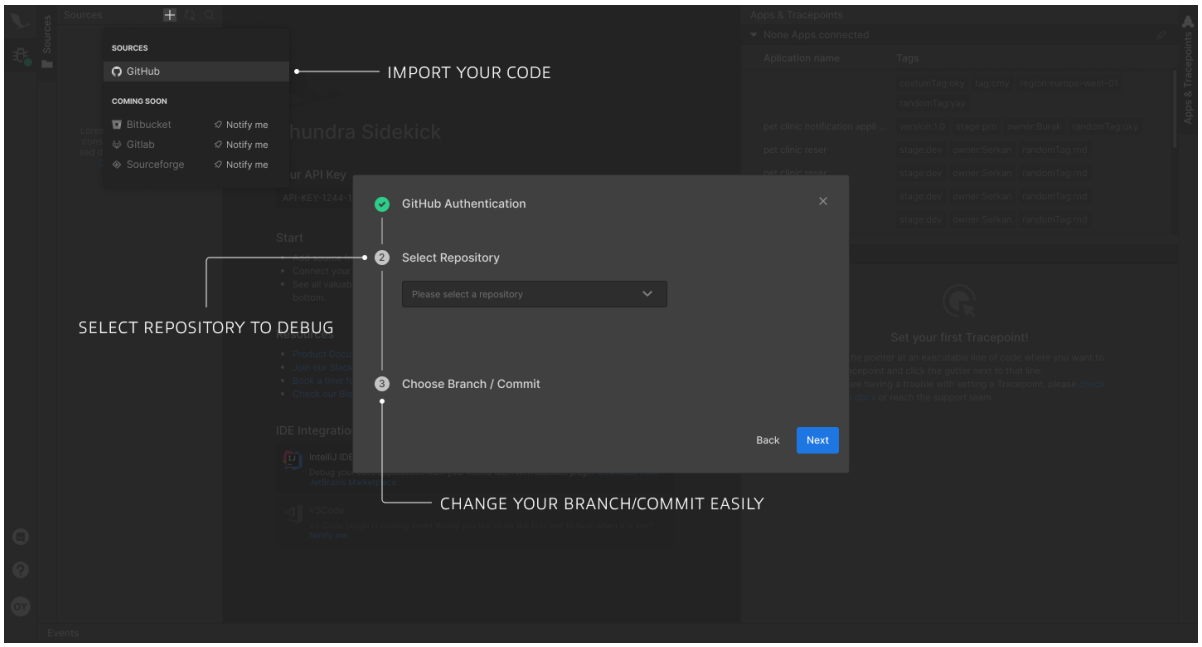
Remote Debugging with Thundra Sidekick

Thundra Sidekick was developed by Thundra's engineering team to respond to its own requirements. This APM application is developed, tested, and served in the cloud. Developers at Thundra had tried all the solutions mentioned above: logs, other APMs, and traditional remote debugging solutions. None of them were efficient enough to help developers cope with their development speed.

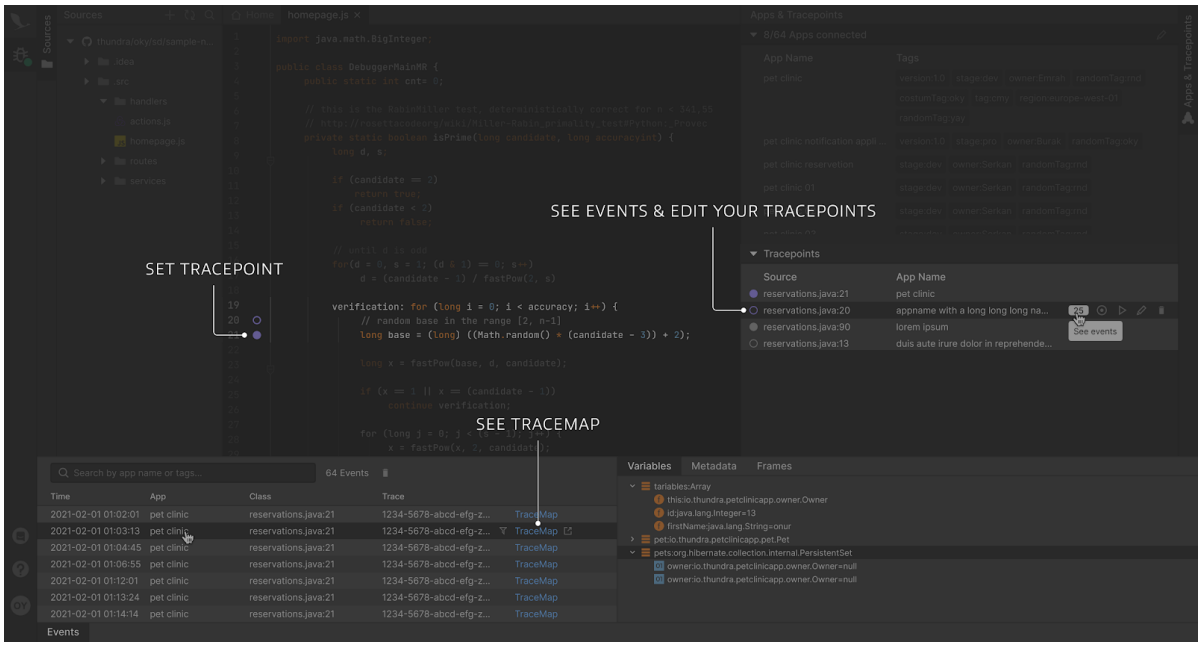
Eventually, we came up with a solution that combines non-intrusive breakpoints with distributed tracing. We call these non-intrusive breakpoints "tracepoints," because Thundra Sidekick genuinely connects the snapshots taken from different points in a particular transaction.

Thundra Sidekick is a forever-free remote application debugger. It's built to serve the developer community and resolve remote debugging pain. Sidekick is set up very easily and no code configuration is required. You can debug your Java and Python applications remotely from Thundra Sidekick's cloud debugger, or from your IntelliJ IDEA IDE with the Thundra Sidekick IntelliJ IDEA Plugin.

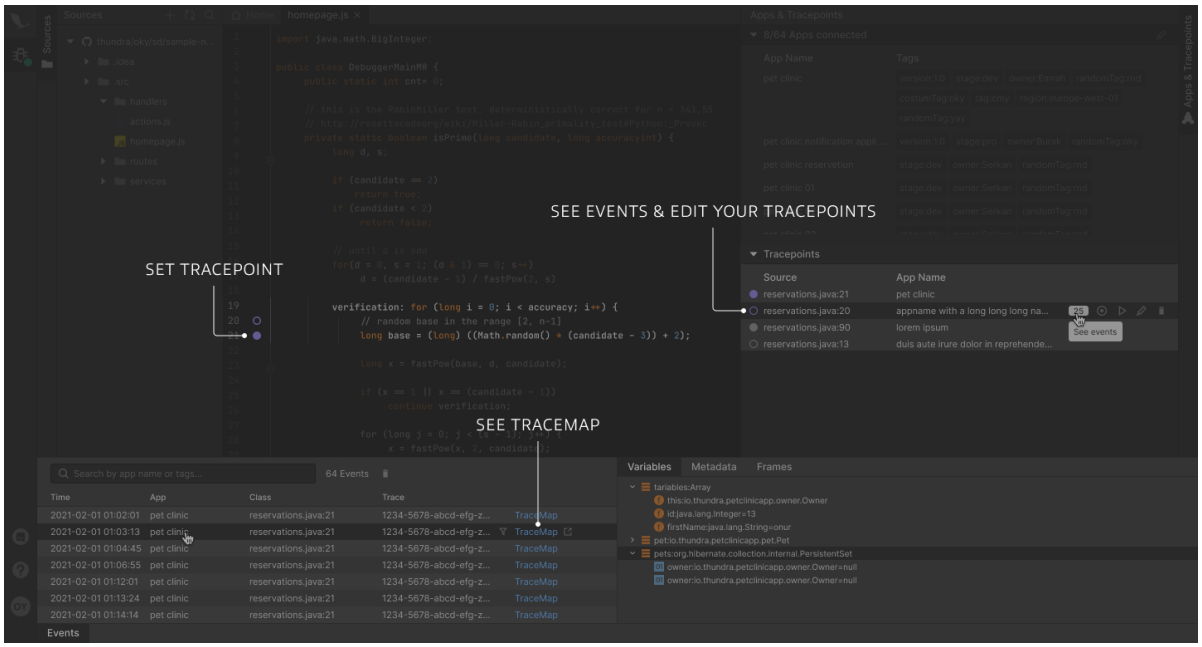
To start using Thundra's cloud debugger, you should import your code from your code repository first. You can import your projects from GitHub, GitLab, Bitbucket, and more.



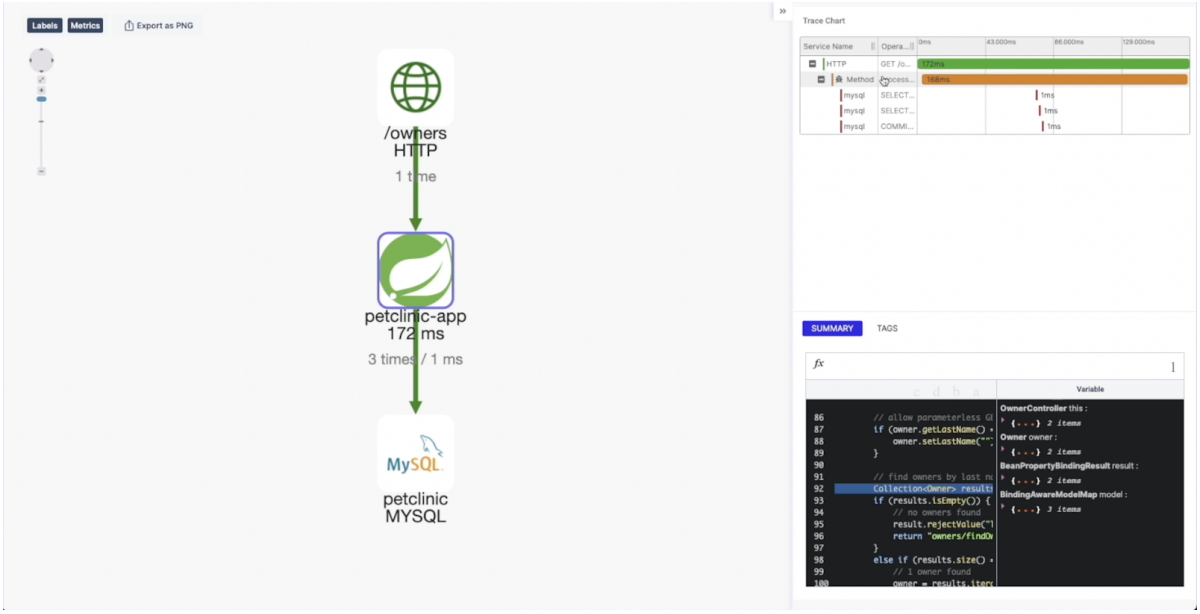
With Thundra Sidekick, it's very easy to set tracepoints. Simply select your application and the version of it you want to examine.



After setting the tracepoint in your application's code, Sidekick is ready to take snapshots of the current stage. When the code execution hits the tracepoint, Thundra Sidekick automatically captures a tracepoint event, which will appear in the table of the Thundra Tracepoint tool at the bottom.



Then, Thundra Sidekick will connect several tracepoints in the same transaction. When you click “See in Thundra APM” on a snapshot event, you will be redirected to the Thundra trace map page for that specific event.



You can easily make small changes and apply hotfixes after taking some tracepoints and looking at their trace maps where you feel where the problem might be. You do not need to redeploy the application to the cloud to see the effect of your hotfix – Thundra Sidekick seamlessly reloads your application without having to redeploy. This feature will be ready and available in a short time. Please contact the Thundra team to be first to know when it is released.

Getting started with Thundra is pretty easy: Just add the SDK and start debugging your remote applications. Thundra's lightweight instrumentation adds negligible minimal overhead and stays silent when not working.

Thundra Sidekick is able to work on any infrastructure on any cloud platform. You can remotely debug your applications while in pre-production environments or production environments, as well on AWS Cloud, Google Cloud, Azure Cloud, and more.

Summary

Remote development is the new way to build modern architectures. But when you want to debug such systems, classic debugging techniques don't work as well as they used to. Using logs can be very time-consuming, costly, and not very effective. Traditional remote debugging solutions are not secure because you need to keep a port open to the public, they're intrusive, and you have to pause the execution of the program you are debugging.

We implemented a tool with non-intrusive debugging as a solution to our own remote debugging requirements for our development and test environment in the cloud. We can easily extract data out of remote applications without actually pausing them or adding an overhead onto them with Thundra Sidekick. As another benefit, we connect the breakpoints of non-intrusive debugging using Thundra's distributed tracing engine.

Sidekick is made for developers by developers. It is meant to serve the developer community and that's why it's forever free. You can use Sidekick on its web application or in your IDE. It's super easy to setup and configure. Singup for Sidekick for free, and use it forever free.

About Thundra

Thundra is an enterprise SaaS company providing the industry's first Application Observability and Security Platform™ for serverless-centric, container, and virtual machine workloads. Application teams spanning software development, DevOps/SRE, IT operations, and IT security rely on Thundra to run fast safely, troubleshooting and debugging with improved MTTR while ensuring security and compliance policies are enforced. Thundra is committed to making the lives of enterprise IT professionals better by reducing the complexity, costs, and bottlenecks slowing teams down, leveraging Thundra's unique technology footprint to replace numerous existing enterprise tools while improving productivity and efficiency.

Want to see Thundra in action?

start.thundra.io

Any questions or inquiries?

Contact us at info@thundra.io

THUNDRA™