Why Understandability is Essential to Every Software Developer

Jason Bloomberg

President and Principal Analyst, Intellyx
October 2020

Intellyx™

Let's consider an all-too-common scenario: some newly deployed software isn't behaving as it should. The operators take one glance at the problem and conclude that it's a bug.

Now it's the developers' problem. They take a look at what they think is the relevant source code, but they can't see the issue. Now what?

They take a look at the logs or application performance management (APM) output to narrow down the location of the bug. Then they write multiple lines of debug code to output various variable values to the log file.

Then the developers recompile the code, and go through their organization's process to test the code. But for some reason, the bug isn't turning up in test. Now what?

Time to promote the code to production – only to do that, it must go through all the reviews and approvals the organization requires, which might take days or weeks. Meanwhile, the bug continues to impact end-users in production.

Clearly, this scenario is sub-optimal. Developers need better information about how their code is behaving in production – ideally on a line-by-line basis. And they need this information without having to recompile and redeploy the code.

What they need is understandability – information about running code sufficient to understand its behavior on a line-by-line basis, regardless of the complexity of the production environment.

From Monolith to the Cloud

The more complicated the production environment, the more important understandability becomes – but understandability is essential for all generations of software architectures and coding practices.

Perhaps your developers are responsible for maintaining and updating some monolithic application. Without understandability, those coders are working in the dark – and the risks inherent in introducing new bugs into production are substantial. After all, a single bug can cause an entire legacy application to crash.

In addition, the processes for maintaining legacy apps generally follow the slower waterfall approach, leading to potentially unacceptable delays when developers must manually track down bugs that impact the production application.

Traditional on-premises n-tier architectures – say, applications running on older .NET or J2EE platforms – can also take advantage of better understandability.

These applications consist of multiple software objects running across multiple runtime environments. As a result, tracking down a bug in a particular line of code becomes that much more difficult as compared to a monolithic application.

Not only to objects depend on other objects, but the n-tier architecture will run multiple instances of code simultaneously. Tracking down bugs has just become far trickier.

What about applications that run in the cloud? Now we have all the challenges that on-premises n-tier applications present, plus a plethora of new problems.

In the cloud, the software runs on virtual machines (VMs), which the team must provision and manage. VMs however, can be ephemeral: they may come and go on a moment's notice. Well-built software, of course, must be sufficiently resilient to maintain the end-user experience in spite of this unpredictability.

How, then, are developers supposed to track the behavior of individual lines of code in such a dynamic environment? Understandability becomes critically important in such situations.

Cloud-Native: Where Understandability Shines

VMs, of course, are oh-so-2015. Today all the buzz is around containers, microservices, Kubernetes, and cloud-native computing in general.

While VMs might come and go on the order of hours or days, we can measure containers' ephemerality by seconds or even fractions of a second.

Furthermore, this dynamic nature of software follows two dimensions: first, cloud-native software must scale faster and more broadly than VMs alone. Second, as the software teams themselves scale up, the cadence of software deployment also accelerates.

In other words, the software in production may be changing as it's scaling up (or down) – and the changes will typically take place too quickly for humans to manage. Instead, automation drives the behavior of cloud-native software at scale.

This inherently fluid, largely automated vision for enterprise software dramatically ups the game for understandability. A single line of code may not only affect multiple microservices running across the Kubernetes landscape, but each microservice may depend upon other microservices or other software components – again dynamically.

The connection between individual lines of code and the behavior of all software running in production is thus tenuous at best. The only way to keep the entire effort from running off the rails is to rigorously follow cloud-native principles while leveraging tools that support those principles.

Among those principles: the emerging practice of GitOps – a new operational model for running such dynamic, ephemeral software – is rapidly becoming the established approach for driving configuration-driven deployments onto immutable infrastructure.

Understandability is an essential part of this GitOps model for cloud-native computing. An understandability tool like Rookout brings information about the running software in production to developers, providing visibility into how individual lines of code impact the behavior of running software, without impacting that software.

In other words, Rookout provides the understandability that developers need to create working code within the context of GitOps. Developers can debug code before merging it into the next release, thus 'shifting debugging to the left' as compared to an approach that requires releasing debug code into production.

In fact, when following GitOps – that is, following cloud-native best practice – developers should never release ad hoc debug code into production. Git maintains all the code plus all the configurations necessary to support all releases. If a problem crops up in production, make the appropriate changes in Git and redeploy. Understandability is the key.

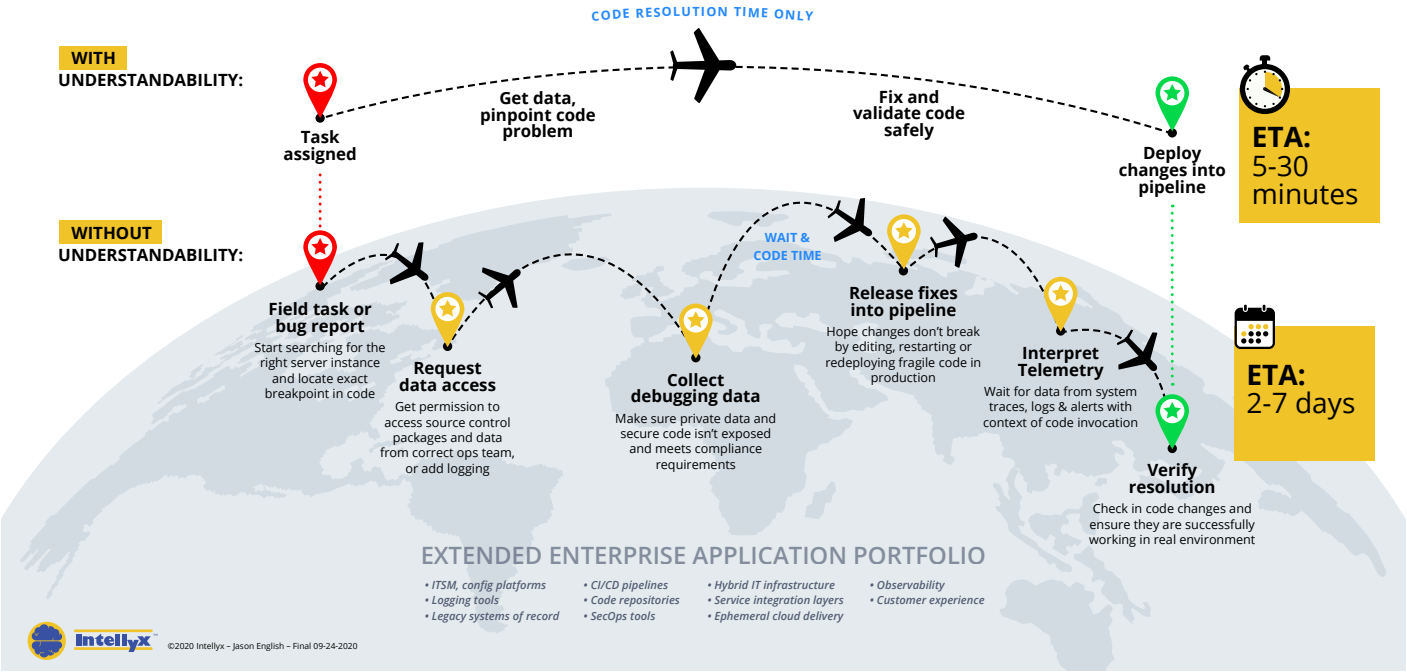
Rethinking Debugging Across the Modern Software Lifecycle

When organizations followed the waterfall methodology, each phase of the software lifecycle was clearly delineated and discrete. Development, test, deployment and operations depended upon different people with different priorities and different tools.

Among those tools: debugging tools for developers and monitoring tools for operators, including APM tools. The fact that developers had to put debug code into production on occasion thus broke the mold that waterfall tried to enforce, introducing delays, friction, and possible security vulnerabilities.

Software Understandability: The Non-Stop Developer Workflow

Each new bug report or feature modification request results in several unnecessary 'hops and stops' atop a high-traffic application portfolio for resolution. Developers must obtain access permissions, search the enterprise application portfolio for data to find code, wait for results, and write more instrumentation code to get more data. **Software understandability** practices complement observability and IT operations management tools with direct code-level breakpoint insertion, to eliminate unproductive work and unnecessary wait time, thereby reducing unproductive hours and days from issue resolution and feature delivery.



Development feature additions and issue resolution processes with and without software understandability, demonstrating in-flight time reduction and productivity gains.
Infographic from 'Software Understandability: The Non-Stop Developer Workflow' by Jason English, Intellyx, 2020.

Today, whether the engineering effort follows Agile or the more modern CI/CD approaches that align with DevOps, the firm gates that divided the phases of the waterfall lifecycle break down, as iterative and eventually continuous approaches replace the more restrictive methodologies of the past.

For developers, this shift means a greater responsibility for how their code behaves in production. DevOps practitioners like to say that developers 'carry their own pagers,' a nostalgic phrase that means that each developer must be responsible for their own code in the production environment.

This responsibility requires information. Clearly, developers can't carry those proverbial pagers if they don't know what's going on with their code when it runs. They require immediate feedback from deployed code, as well as information about how their code runs in the context of all the other components of the running software.

Rookout helps organizations achieve understandability by enabling developers to retrieve data from their running applications without affecting the performance of the application or requiring redeployment – and without breaking anything. Developers can retrieve data whenever they require in order to achieve full understanding of an application without having to know ahead of time what data they will need to resolve an issue.

Understandability isn't simply a nice to have. It's a must have. And we must not forget that we require understandability without impacting the running software. Only tools like Rookout can deliver this essential capability.

The Intellyx Take

Understandability is important for more than writing greenfield software applications. Understandability is also an important principle for professionals who are involved in various modernization and migration initiatives as well.

In some cases, the modernization professional is rewriting old applications wholesale, essentially replacing old with new. For such engineers, understandability provides vital insight for writing such code.

In other situations, modernization requires updating old code, mixing old and new code to meet new requirements with existing applications. In such situations, understandability is vitally important, as bugs may crop up in the older code, the newer code, or as a result of the new combination of code. How else would a developer figure out which is which?

Modernization might also involve a transition to greater modularization, where the development team largely leaves old software in place (perhaps API-enabling it), while adding new functionality in separate bespoke modules.

In such a situation, new code must interface with old – and those interfaces themselves may be under development. Understandability now plays a triple role: old, new, and the interfaces between them.

Finally, cloud migration may be the modernization scenario of the day. While it's tempting to simply take old software and 'lift and shift' it wholesale into VMs in the cloud, in reality developers must make numerous changes for such migrated code to take advantage of the special characteristics of the cloud environment. Indeed, understandability is essential in such situations as well.

The bottom line: regardless of whether your software is old or new, whether you're dealing with monolithic or cloud-native applications, or whether you're still following waterfall vs. adopting DevOps, understandability is a must-have. Without a vendor like Rookout, you'd still be stuck hand-coding debug code.

About the Author

Jason Bloomberg is a leading IT industry analyst, author, keynote speaker, and globally recognized expert on multiple disruptive trends in enterprise technology and digital transformation. He is founder and president of Digital Transformation analyst firm Intellyx. He is ranked among the top nine low-code analysts on the Influencer50 Low-Code50 Study for 2019, #5 on Onalytica's list of top Digital Transformation influencers for 2018, and #15 on Jax's list of top DevOps influencers for 2017.

Mr. Bloomberg is the author or coauthor of five books, including Low-Code for Dummies, published in October 2019.