5 Common Pitfalls of Azure Pipelines

Azure Pipelines gives you the power to create automated build, test, and deploy workflows for continuous integration and delivery: a foundational element of your DevOps infrastructure. But don't dive in blindly—there are plenty of ways to get blocked if you don't know what to look for. Luckily, Pipelines has many solutions built right in. Here are the most common pitfalls we've seen teams fall into and methods for nimbly navigating around them.

INSECURE VARIABLES

A common practice in building release pipelines is applying application configuration on deployment. This can mean setting environment variables and transforming configuration files. These variables often include sensitive information such as connection strings, API keys, and account credentials: Data that could do damage if compromised.

Azure Pipelines offers a few ways to secure these variables. First, you can simply lock the variables in our pipeline configuration. This will store the value encrypted, once entered. If you need greater security, you can use Azure App Configuration, or Azure KeyVault, both of which allow separate security profiles than the Pipeline variables.

LACK OF VISIBILITY

Azure Pipelines are well-integrated into many issue tracking and source control tools. You can look at a pull request on GitHub, BitBucket, or Azure Repos (of course), and see if the build passed or failed, with very little set-up and configuration. But this only tells you the status when you actually visit the website—what if something goes wrong when you're not looking? Will the right people be notified of the right set of problems at the right time?

The default way to send notifications is via email—but that's probably not what provides the most visibility for your team. Azure Pipelines has fully configurable notifications, so if your team members prefer SMS, Slack, or Microsoft Teams, they can get notified where they want to. Just about anything is possible here.

NOT ENOUGH AGENTS

You check in your code, push up a branch, and request a review for our PR. There's one problem—the build isn't done! Checking the status, it turns out that the build is in that dreadful "queued" state: Other builds need to finish before yours can begin. The larger your team and organization, the more builds can stack up.

While it's possible to create completely "on-demand" build agents, they can be tricky to set up. Instead, your organization needs to determine how many parallel build agents you need to configure to make sure that the team doesn't wait too long. It can be a balancing act, but the pricing of parallel build agents ($40/mo/agent) is still quite a bit lower than the calculated cost of waiting on builds to complete.

SLOW BUILDS

When practicing Continuous Integration and Delivery, you need fast feedback loops. A common bottleneck that crops up for those new to CI/CD is slow CI build times. Anecdotally, a build taking longer than about 10 minutes introduces frustration and inefficiency: If I have to wait longer than the time it takes to get a cup of coffee, then I'm blocked on the next task or user story.

Fixing slow CI builds requires a bit of research as to *why* the build is slow. There are a few common sources: 1) Package restore 2) Integration tests 3) Code scanning (security/style/code coverage). There's not often an easy fix, but you can start by looking at ways of providing secondary/chained builds. Continuous integration is the first step to continuous delivery: You should examine each step in the build and ask, "Should a failure here prevent deployment? If not, can we move these steps to a secondary chained/triggered build?"

NO CACHING

While it can be straightforward to move slow steps to a secondary build, package restoration can be painfully slow. In some builds, we've seen package restoration alone take 10 minutes. You have to have dependencies to build and deploy, so it's not a step we can just move out.

Depending on your agents, you can speed up your build by caching resolved dependencies. For hosted agents, you likely won't run into caching problems unless the hosted agent is a virtual machine that gets wiped every time. For cloud build agents, you'll need to configure your build pipeline to cache your dependencies (NPM, NuGet, etc.). The next time the build runs, it will cut the dependency resolution time by a significant margin—sometimes over 95%.