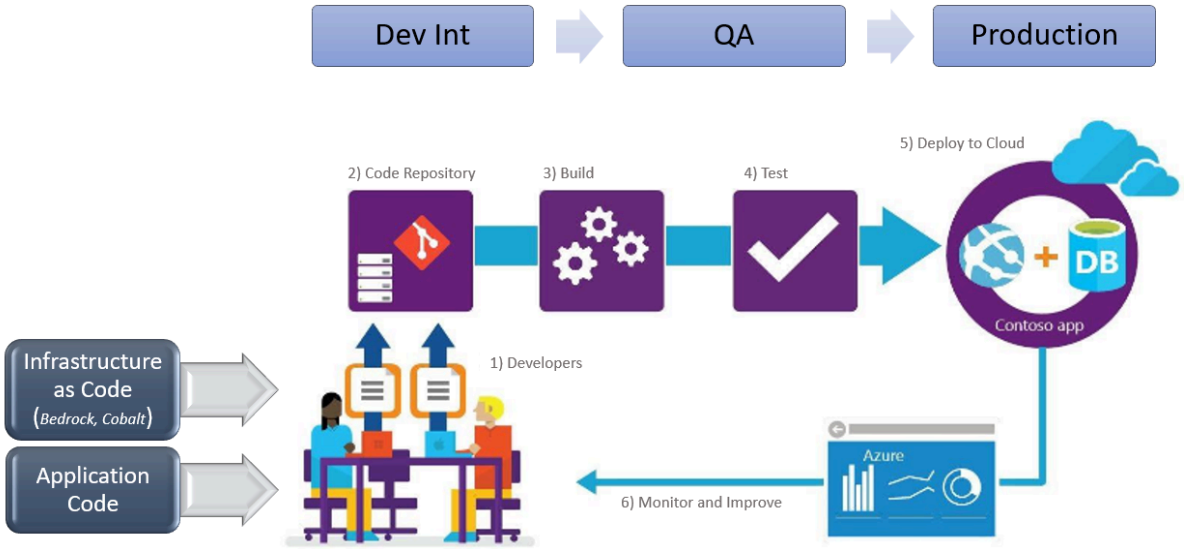
Continuous Integration



We encourage engineering teams to make an upfront investment during Sprint 0 of a project to establish an automated and repeatable pipeline which continuously integrates code and releases system executable(s) to target cloud environments. Each integration should be verified by an automated build process that asserts a suite of validation tests pass and surface any errors across the developer team.

We encourage teams to implement the CI/CD pipelines before any service code is written for customers, which usually happens in Sprint 0(N). This way, the engineering team can develop and test their work in isolation without impacting other developers and promote a consistent devops workflow throughout the engagement.

These principles map directly agile software development lifecycle practices.

Goals

Continuous integration automation is an integral part of the software development lifecycle intended to reduce build integration errors and maximize velocity across a dev crew.

A robust build automation pipeline will:

- Accelerate team velocity

- Prevent integration problems
- Avoid last minute chaos during release dates
- Provide a quick feedback cycle for system-wide impact of local changes
- Separate build and deployment stages
- Measure and report metrics around build failures / success(s)
- Increase visibility across the team enabling tighter communication
- Reduce human errors, which is probably the most important part of automating the builds

Build Definition Managed in Git

Code / Manifest Artifacts Required to Build Your Project Should be Maintained Within Your Projects Git Repository

- CI provider-specific build pipeline definition(s) should reside within your project(s) git repository(s).

Build Automation

An automated build should encompass the following principles:

Build Task

- A single step within your build pipeline that compiles your code project into a single build artifact.

Unit Testing

- Your build definition includes validation steps to execute a suite of automated unit tests to ensure that application components meets its design and behaves as intended.

Code Style Checks

- Code across an engineering team must be formatted to agreed coding standards. Such standards keep code consistent, and most importantly easy for the team and customer(s) to read and refactor. Code styling consistency encourages collective ownership for project scrum teams and our partners.

- There are several open source code style validation tools available to choose from (code style checks, StyleCop). The Code Review recipes section of the playbook has suggestions for linters and preferred styles for a number of languages.
- Your code and documentation should avoid the use of non-inclusive language wherever possible. Follow the Inclusive Linting section to ensure your project promotes an inclusive work environment for both the team and for customers.
- We recommend incorporating security analysis tools within the build stage of your pipeline such as: code credential scanner, security risk detection, static analysis, etc. For Azure DevOps, you can add a security scan task to your pipeline by installing the Microsoft Security Code Analysis Extension. GitHub Actions supports a similar extension with the RIPS security scan solution.
- Code standards are maintained within a single configuration file. There should be a step in your build pipeline that asserts code in the latest commit conforms to the known style definition.

Build Script Target

- A single command should have the capability of building the system. This is also true for builds running on a CI server or on a developers local machine.

No IDE Dependencies

- It's essential to have a build that's runnable through standalone scripts and not dependent on a particular IDE. Build pipeline targets can be triggered locally on their desktops through their IDE of choice. The build process should maintain enough flexibility to run within a CI server as well. As an example, dockerizing your build process offers this level of flexibility as VSCode and IntelliJ supports docker plugin extensions.

DevOps Security Checks

- Introduce security to your project at early stages. Follow the DevSecOps section to introduce security practices, automation, tools and frameworks as part of the CI.

Build Environment Dependencies

Automated Local Environment Setup

- We encourage maintaining a consistent developer experience for all team members. There should be a central automated manifest / process that streamlines the installation and setup of any software dependencies. This way developers can replicate the same build environment locally as the one running on a CI server.
- Build automation scripts often require specific software packages and version pre-installed within the runtime environment of the OS. This presents some challenges as build processes typically version lock these dependencies.
- All developers on the team should be able to emulate the build environment from their local desktop regardless of their OS.
- For projects using VS Code, leveraging Dev Containers can really help standardize the local developer experience across the team.
- Well established software packaging tools like Docker, Maven, npm, etc should be considered when designing your build automation tool chain.

Document Local Setup

- The setup process for setting up a local build environment should be well documented and easy for developers to follow.

Infrastructure as Code

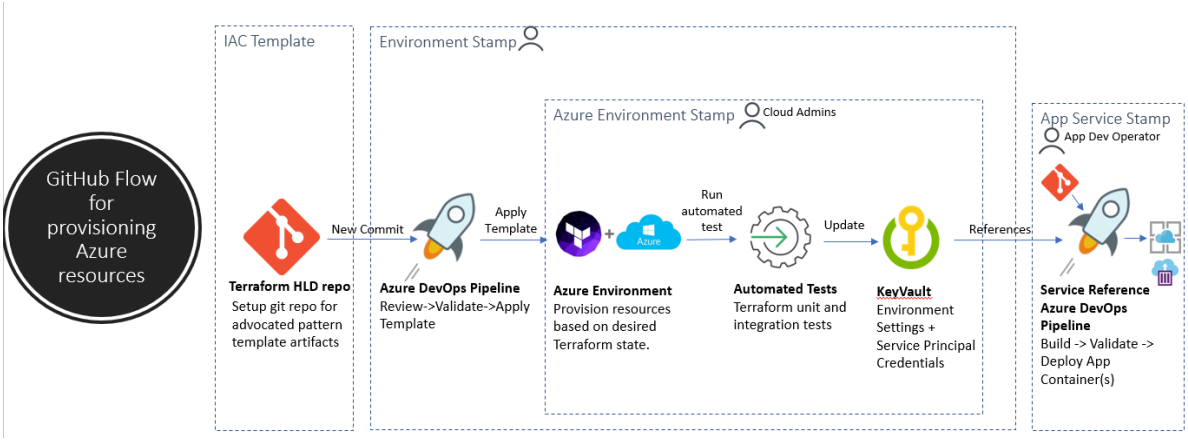
Manage as much of the following as possible, as code:

- Configuration Files
- Configuration Management(ie environment variable automation via terraform)
- Secret Management(ie creating Azure secrets via terraform)
- Cloud Resource Provisioning
- Role Assignments
- Load Test Scenarios
- Availability Alerting / Monitoring Rules and Conditions

Decoupling infrastructure from the application codebase simplifies engineering teams move to cloud native applications.

Terraform resource providers like Azure DevOps is making it easier for developers to manage build pipeline variables, service connections and CI/CD pipeline definitions.

Sample DevOps Workflow using Terraform and Cobalt



Why

- Repeatable and auditable changes to infrastructure make it easier to roll back to known good configurations and to rapidly expand to new stages and regions without having to hand-wire cloud resources
- Battle tested and templated IAC reference projects like Cobalt and Bedrock enable more engineering teams deploy secure and scalable solutions at a much more rapid pace
- Simplify “lift and shift” scenarios by abstracting the complexities of cloud-native computing away from application developer teams.

IAC DevOPS: Operations by Pull Request

- The Infrastructure deployment process built around a repo that holds the current expected state of the system / Azure environment.
- Operational changes are made to the running system by making commits on this repo.
- Git also provides a simple model for auditing deployments and rolling back to a previous state.

Infrastructure Advocated Patterns

- You define infrastructure as code in Terraform / ARM / Ansible templates
- Templates are repeatable cloud resource stacks with a focus on configuration sets aligned with app scaling and throughput needs.

IAC Principles

Automate the Azure Environment

- All cloud resources are provisioned through a set of infrastructure as code templates. This also includes secrets, service configuration settings, role assignments and monitoring conditions.
- Azure Portal should provide a read-only view on environment resources. Any change applied to the environment should be made through the IAC CI tool-chain only.
- Provisioning cloud environments should be a repeatable process that's driven off the infrastructure code artifacts checked into our git repository.

IAC CI Workflow

- When the IAC template files change through a git-based workflow, A CI build pipeline builds, validates and reconciles the target infrastructure environment's current state with the expected state. The infrastructure execution plan candidate for these fixed environments are reviewed by a cloud administrator as a gate check prior to the deployment stage of the pipeline applying the execution plan.

Developer Read-Only Access to Cloud Resources

- Developer accounts in the Azure portal should have read-only access to IAC environment resources in Azure.

Secret Automation

- IAC templates are deployed via a CI/CD system that has secrets automation integrated. Avoid applying changes to secrets and/or certificates directly in the Azure Portal.

Infrastructure Integration Test Automation

- End-to-end integration tests are run as part of your IAC CI process to inspect and validate that an azure environment is ready for use.

Infrastructure Documentation

- The deployment and cloud resource template topology should be documented and well understood within the README of the IAC git repo.
- Local environment and CI workflow setup steps should be documented.

Configuration Validation

Applications use configuration to allow different runtime behaviors and it's quite common to use files to store these settings. As developers, we might introduce errors while editing these files which would cause issues for the application to start and/or run correctly. By applying validation techniques on both syntax and semantics of our configuration, we can detect errors before the application is deployed and execute, improving the developer (user) experience.

Application Configuration Files Examples

- JSON, with support for complex data types and data structures
- YAML, a super set of JSON with support for complex data types and structures
- TOML, a super set of JSON and a formally specified configuration file format

Why Validate Application Configuration as a Separate Step?

- **Easier Debugging & Time saving -** With a configuration validation step in our pipeline, we can avoid running the application just to find it fails. It saves time on having to deploy & run, wait and then realize something is wrong in configuration. In addition, it also saves time on going through the logs to figure out what failed and why.
- **Better user/developer experience -** A simple reminder to the user that something in the configuration isn't in the right format can make all the difference between the joy of a successful deployment process and the intense frustration of having to guess what went wrong. For example, when there is a Boolean value expected, it can either be a string value like "True" or "False" or an integer value such as "0" or "1" . With configuration validation we make sure the meaning is correct for our application.
- **Avoid data corruption and security breaches -** Since the data arrives from an untrusted source, such as a user or an external webservice, it's particularly important to validate the input . Otherwise, it will run at the risk of performing errors, corrupting data, or, worse, be vulnerable to a whole array of injection attacks.

What is Json Schema?

JSON-Schema is the standard of JSON documents that describes the structure and the requirements of your JSON data. Although it is called JSON-Schema, it also common to use this method for YAMLs, as it is a super set of JSON. The schema is very simple; point out which fields might exist, which are required or optional, what data format they use. Other validation rules can be added on top of that basic premise, along with human-readable information. The metadata lives in schemas which are .json files as well. In addition, schema has the widest adoption among all

standards for JSON validation as it covers a big part of validation scenarios. It uses easy-to-parse JSON documents for schemas and is easily extensible.

How to Implement Schema Validation?

Implementing schema validation is divided in two - the generation of the schemas and the validation of yaml/json files with those schemas.

Generation

There are two options to generate a schema:

- From code - we can leverage the existing models and objects in the code and generate a customized schema.
- From data - we can take yaml/json samples which reflect the configuration in general and use the various online tools to generate a schema.

Validation

The schema has 30+ validators for different languages, including 10+ for JavaScript, so no need to code it yourself.

Integration Validation

An effective way to identify bugs in your build at a rapid pace is to invest early into a reliable suite of automated tests that validate the baseline functionality of the system:

End-to-End Integration Tests

- Include tests in your pipeline to validate the build candidate conforms to automated business functionality assertions. Any bugs or broken code should be reported in the test results including the failed test and relevant stack trace. All tests should be invoked through a single command.
- Keep the build fast. Consider automated test runtime when deciding to pull in dependencies like databases, external services and mock data loading into your test harness. Slow builds often become a bottleneck for dev teams when parallel builds on a CI server are not an option. Consider adding max timeout limits for lengthy validations to fail fast and maintain high velocity across the team.

Avoid Checking in Broken Builds

- Automated build checks, tests, lint runs, etc should be validated locally before committing your changes to the scm repo. Test Driven Development is a practice dev crews should consider to help identify bugs and failures as early as possible within the development lifecycle.

Reporting Build Failures

- If the build step happens to fail then the build pipeline run status should be reported as failed including relevant logs and stack traces.

Test Automation Data Dependencies

- Any mocked dataset(s) used for unit and end-to-end integration tests should be checked into the mainline repository. Minimize any external data dependencies with your build process.

Code Coverage Checks

- We recommend integrating code coverage tools within your build stage. Most coverage tools fail builds when the test coverage falls below a minimum threshold(80% coverage). The coverage report should be published to your CI system to track a time series of variations.

Git Driven Workflow

→ should triggered on each commit
— Artifacts → deployed continiously on non-prod

Build on Commit

- Every commit to the baseline repository should trigger the CI pipeline to create a new build candidate.
- Build artifact(s) are built, packaged, validated and deployed continuously into a non-production environment per commit. Each commit against the repository results into a CI run which checks out the sources onto the integration machine, initiates a build, and notifies the committer of the result of the build.

Avoid Commenting Out Failing Tests

- Avoid commenting out tests in the mainline branch. By commenting out tests, we get an incorrect indication of the status of the build.

Branch Policy Enforcement

- Protected branch policies should be setup on the main branch to ensure that CI stage(s) have passed prior to starting a code review. Code review approvers will only start reviewing a pull request once the CI pipeline run passes for the latest pushed git commit.
- Broken builds should block pull request reviews.
- Prevent commits directly into main branch.

before requesting pull request, CI pipeline run passes for the latest pushed commit.

Branch Strategy

- Release branches should auto trigger the deployment of a build artifact to its target cloud environment. You can find additional guidance on the Azure DevOps documentation site under the Manage deployments section

release branch → should target perticular environment

Deliver Quickly and Daily

> "By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making."

In the spirit of transparency and embracing frequent communication across a dev crew, we encourage developers to commit code on a daily cadence. This approach provides visibility to feature progress and accelerates pair programming across the team. Here are some principles to consider:

Everyone Commits to the Git Repository Each Day

- End of day checked-in code should contain unit tests at the minimum.
- Run the build locally before checking in to avoid CI pipeline failure saturation. You should verify what caused the error, and try to solve it as soon as possible instead of committing your code. We encourage developers to follow a lean SDLC principles.
- Isolate work into small chunks which ties directly to business value and refactor incrementally.

Isolated Environments

One of the key goals of build validation is to isolate and identify failures in staging environment(s) and minimize any disruption to live production traffic. Our E2E automated tests should run in an

environment which mimics our production environment(as much as possible). This includes consistent software versions, OS, test data volume simulations, network traffic parity with production, etc.

Test in a Clone of Production

- The production environment should be duplicated into a staging environment(QA and/or Pre-Prod) at a minimum.

Pull Request Updates Trigger Staged Releases

- New commits related to a pull request should trigger a build / release into an integration environment. The production environment should be fully isolated from this process.

Promote Infrastructure Changes Across Fixed Environments

- Infrastructure as code changes should be tested in an integration environment and promoted to all staging environment(s) then migrated to production with zero downtime for system users.

Testing in Production

- There are various approaches with safely carrying out automated tests for production deployments. Some of these may include:
  - Feature flagging
  - A/B testing
  - Traffic shifting

Developer Access to the Latest Release Artifacts

Our devops workflow should enable developers to get, install and run the latest system executable. Release executable(s) should be auto generated as part of our CI/CD pipeline(s).

Developers can Access the Latest Executable

- The latest system executable is available for all developers on the team. There should be a well-known place where developers can reference the release artifact.

Release Artifacts are Published for Each Pull Request or Merges into the Main Branch

Integration Observability

Applied state changes to the mainline build should be made available and communicated across the team. Centralizing logs and status(s) from build and release pipeline failures are essential for developers investigating broken builds.

We recommend integrating Teams or Slack with CI/CD pipeline runs which helps keep the team continuously plugged into failures and build candidate status(s).

Integrate team with CI/CD

Continuous Integration Top Level Dashboard

- Modern CI providers have the capability to consolidate and report build status(s) within a given dashboard.
- Your CI dashboard should be able to correlate a build failure with a git commit.

Build Status Badge in the Project Readme

- There should be a build status badge included in the root README of the project.

Build Notifications

- Your CI process should be configured to send notifications to messaging platforms like Teams / Slack once the build completes. We recommend creating a separate channel to help consolidate and isolate these notifications.

Resources

- Martin Fowler's Continuous Integration Best Practices
- Bedrock Getting Started Quick Guide
- Cobalt Quick Start Guide
- Terraform Azure DevOps Provider
- Azure DevOps multi stage pipelines
- Azure Pipeline Key Concepts

- Azure Pipeline Environments
- Artifacts in Azure Pipelines
- Azure Pipeline permission and security roles
- Azure Environment approvals and checks
- Terraform Getting Started Guide with Azure
- Terraform Remote State Azure Setup
- Terratest - Unit and Integration Infrastructure Framework

Last update: December 10, 2024