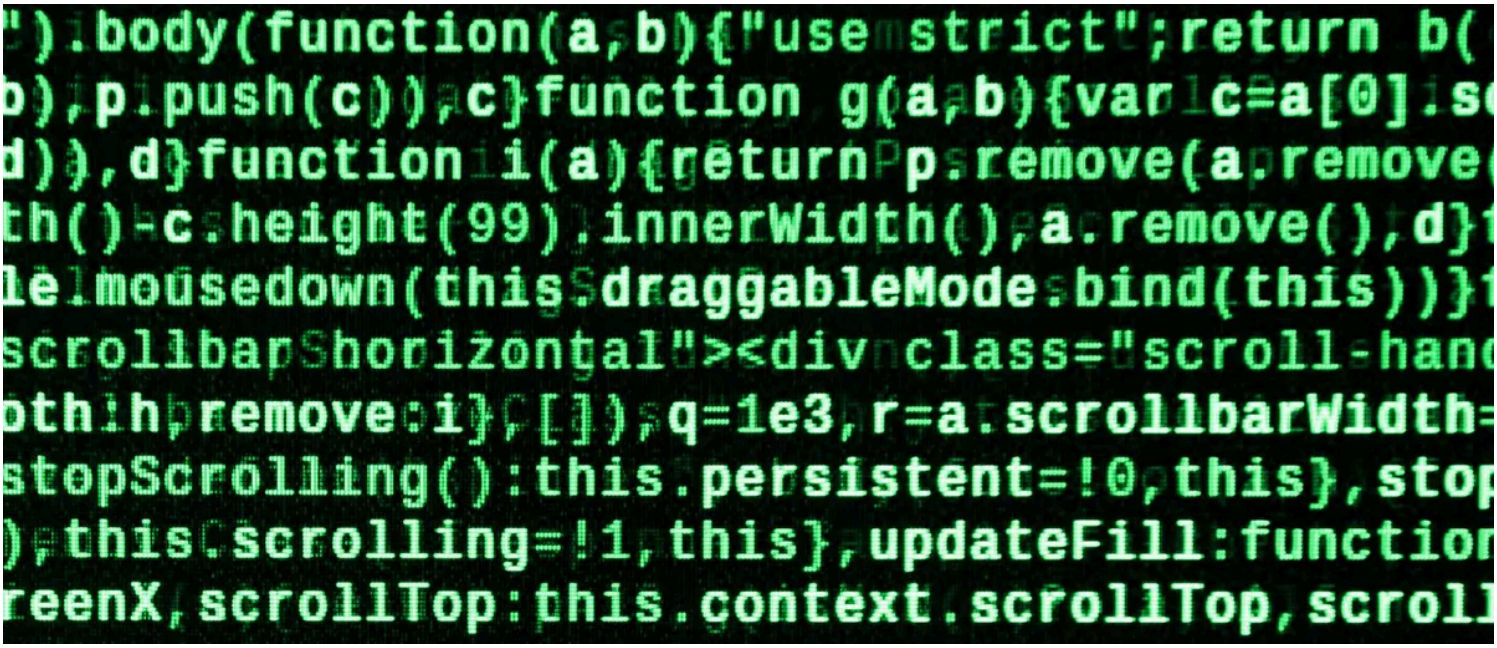
What is Everything-as-Code? Examining the Explosion of "as Code" Buzzwords

October 27th 2019 7,583 reads 5



@mpron
Mitch Pronschinske

11 yr veteran of the software development and DevOps content space. Opinions my own.

About @mpron

If you're confused when you read about "[some software term] as code" or "everything as code," all you really need to know is that we're talking about *automation*: The thing we use to do tedious tasks for us, or orchestrate tasks when they become too large and complex for manual methods.

Update 2022: For some deeper research after this article, I highly recommend a new blog by Patrick Debois (the person who coined the

word "DevOps") about trends analyzed from 50+ different concepts "as code".

The “as code” buzzwords started with “infrastructure as code” and the DevOps movement, when IT operations/sysadmins and developers started working together to automate IT environment modifications with reusable code and then version control that code much like developers had been handling their application code changes for many years before.

Developers had been automating their software delivery processes for a while, but IT infrastructure operators/sysadmins have been catching up only in the past decade. Sysadmins still used automation, but it was usually through a series of manually triggered scripts—some of them less organized than others. Developers are helping Ops drive this evolution so that devs will have fewer bottlenecks and can take care of most of the software production process themselves, without much manual gatekeeping.

So where did everything as code come from? To understand that, let’s actually look at every stage of software production and see where there’s already “*-as-code” automation. Let’s use this article as an opportunity to think about areas of the software development lifecycle (SDLC) that are already well-automated and abstracted so that we can see if there are any areas for novel automation and learn where “everything as code” resides. This might help you sort through the “*-as-code” buzzword noise and distinguish which tools are actually novel and helpful versus those that are just trying to sound new and innovative.

Planning/Requirements/Design

This is the phase before coding begins. Requirements are created by business and technical stakeholders, plans for the software are drawn up, and architecture and UI designs are diagrammed and wireframed.

Architecture as code

Being able to create software architecture models as code has been possible for a while. Tools like Structurizr allow you to build diagrams this way and check them into version control. But for decades developers have tried, with varied results, to do more with these application models.

The concept of model-driven development was heavily discussed 10-20 years ago, but it never really took off in mainstream development. Its basic premise is that you can make models of software components using modeling languages like UML or EMF to generate entire starter applications based on the requirements you code.

While model-driven development is largely dead in the mainstream development circles because it usually involves big design up front, a

few domains that have to use big design up front still use it, such as NASA.

***Note about an emerging usage of the term "architecture as code"*:** This term is also starting to be used by Amazon Web Services, and others, as a term to describe higher-level infrastructure as code patterns. This is obviously different from the application-level architecture I've referenced in this section. AWS's architecture as code instead refers to an infrastructure architecture—such as container orchestrator configuration, load balancers, security groups, and IAM roles—all defined via reusable templates for particular service types (e.g. an API service behind a load balancer, a service that pulls work from a queue, or a scheduled job).

These templates automate the provisioning of components such as container orchestrator configurations, load balancers, security groups, and IAM roles with your pre-specified configurations. This version of architecture as code exists in the form of Terraform modules, Kubernetes custom resource definitions, and various constructs from the major cloud vendors.

Projects as code + Scaffolding

The code generation concepts from model-driven development, however, are as old as programming itself. The compilers that turn Ruby, Python, or Java into C code are all generators. Even novice frontend programmers today understand the importance of code generation, which saves you from writing blocks and blocks of the dreaded stuff we consider boilerplate code.

When you run a Ruby on Rails scaffolding process, you're running code that was written and modified to build some other code—a template or skeleton application—so you don't have to start from scratch. Incredibly popular frontend frameworks like Bootstrap and Foundation are responsive UI template frameworks that are noticeable all over the web because it's so much easier to create a website with CSS components and mobile-compatible dynamics already built for you.

There is a lot of room for novel automation in this category, and in the last few years "projects as code"—repo and delivery pipeline setup for applications—has filled in some gaps that tools coming out of the DevOps movement hadn't addressed yet. Tools like Yeoman for JavaScript projects and JHipster for Java projects have provided bare-bones, flexible frameworks for setting up projects using Node.js and Spring Boot+Angular respectively. Cloud vendors, such as Microsoft Azure, are creating tools that set up delivery pipelines. Another company, realMethods, has built a tool that takes database schema or modeled requirements and uses them to generate tech stack specific skeleton code, along with build and config files, CI/CD YAML, Docker files and images, Kubernetes cluster config, and Terraform provisioning files.

"Projects as code" fits in with a growing school of called "hello, production," which is like an extreme version or reemphasis of the minimum viable product (MVP) concept: You don't want to deploy an application that's just a readme file in a version control repository, but you also don't need to build any complete application features. You just need to automate the setup for the schema and pipelines that can deploy the simplest possible skeleton application (e.g. a web app exposing a single endpoint in production that responds with a 200 status code) end-to-end as quickly as possible.

Application environment as code

Virtual machines (VMs), containers, and other various abstractions of a software production environment have been a blessing to developers for many years as they continue to get better and easier to manage. Why try and debug 100 different things that could go wrong while manually setting up an operational environment when you can automatically spin up template environments that allow you to start coding with almost no issues?

Technologies like Vagrant and later Docker (and containers in general) made development environment setup a whole lot easier. Both containers and container orchestrators (such as Kubernetes or Nomad) became popular tools for deploying environments in agile, isolated, correctly configured pieces.

You can think of these tools as providing system dependencies as code. A Dockerfile is a codification and templatization of the dependencies—such as the version of your programming language, or the version of your database—and Docker is the tool that runs and enforces those dependencies.

Version Control

Version control is an overarching phase that is constantly in use from the first completed lines of code all the way to patches on the final product years later. Even most beginning developers quickly start to understand that modern development is impossible without version control. It's change tracking, development gatekeeping, race condition management, and a host of other things *as code*.

These days, version control systems like GitHub, GitLab, and BitBucket have become the central nervous system for most software production pipelines, providing integrated continuous integration (CI), documentation generation, deployment, and more.

One of the biggest reasons for having different elements of software production expressed as code, is so that that code can be checked into version control and managed intelligently. Many organizations find version control so useful, that they're often stretching the boundaries of what should and shouldn't be in version control.

Development

This is the phase that begins after the first code commit to version control and keeps going as long as the software continues to be

maintained. Note that these plan-develop-deploy-gather feedback cycles can be quick (and often, they should be), so this phase may be as big as a whole project's creation or as small as a single line of code changed.

Test automation

Testing is a critical part of most development workflows and the need for testing frameworks became apparent early on in the history of programming. While not all categories of testing need to be automated—you don't want to write a test for every conceivable UI user flow—there are definitely tests that nearly every development team will automate (the Test Pyramid gives some guidance on this).

Unit testing is almost always codified and run by the development team as one big automated checklist. Integration and UI tests are typically run alongside this bundle of unit tests after a code commit. Some tests will happen before the build, some will happen after. The amount of automatic checks, error catching, and code gatekeeping that test frameworks give is indispensable in modern software development.

Each programming language ecosystem comes with its own list of testing frameworks, some using the language itself to define the test actions as code, and some using their own particular syntax.

- Popular test frameworks for **JavaScript** include Jest and Mocha
- Popular test frameworks for **Java** include JUnit and Spock
- Popular test frameworks for **Ruby** include RSpec and Minitest
- Popular frameworks for **browser test automation** include Selenium and Cypress

Build automation & dependency management

Build automation has been around for a very long time. It's a simple idea that developers have quickly understood—instead of trying to do a number of complex manual steps off of a checklist to correctly build a working version of a project, you should create a tool that runs down your checklist in the form of code and automates that build process without accidental omissions.

This is especially useful when you have complex build environments that might require:

- Code generation
- Compilation
- Packaging
- Metadata customization
- And more

In a Java environment, this might mean creating DAOs from your configuration, or generating class-mapping code such as JAXB.

Build automation also helps bridge the gap between a build that might pass locally but might still be missing dependencies or fail for another reason when it goes to the production environment. These build automation tools provide another form of dependency management by gathering all the right libraries and other pieces for the build.

A developer runs one command, clicks a button, or in some cases they have set up an automation chain of events where they don't have to do anything but simply commit code and the build manager will automatically run.

Most major programming languages have their own community-preferred options for build automation.

- For **JavaScript**, it's Node.js + npm + Webpack
- For **Java**, it's Gradle and Maven
- For **Ruby**, it's Rake
- For **C#**, it's MSBuild
- For **C**, it's Make

Continuous Integration / Continuous Delivery (CI/CD)

The term CI/CD is commonly used today to describe the overarching tools and practices that are pulling the practices of build, test, and deployment automation together to happen multiple times a day as code merges to the product's production branch/trunk become more frequent.

Continuous integration is the practice of using a CI-specific server to frequently merge code branches into the mainline version of working application source code, or to flag code if a merge will cause errors. Continuous delivery is the concept of being able to have a chain of automation that makes any new code commit result in a new, deployable version of the application with the changes included. This deployable artifact automatically goes through test automation, CI, build automation, and any other deployment preparations, such as building new containers for the application.

Most of today's CI tools have expanded their feature sets to do more than just CI. The development or release engineering teams can codify many different build steps and the build output as code. That CI/CD configuration then gets version controlled and monitored with the rest of the application code. For the most popular CI platform, Jenkins, it uses a Jenkinsfile as the codification. YAML is a frequent, but sometimes maligned configuration language for many of these CI platforms.

The four most popular CI engines (they are all language-agnostic) are Jenkins, TravisCI, CircleCI, and TeamCity.

Operations

Operations is the phase of software production commonly managed by people with different skill sets than developers (though not always in smaller organizations). These include sysadmins, operations engineers, DevOps engineers, and other titles.

This is the stage where the physical and virtual hardware that the applications will run on are managed and provisioned. These days, many operations professionals only deal with abstractions on top of pools of those physical machines.

Configuration Management

This category has a very broad sounding name, but in most software engineering circles it's used as a descriptor for tools that codify individual machine definitions. These tools install and manage operating system configurations on existing servers.

The four major configuration management tools are Chef, Puppet, Ansible, and Salt. They were some of the first tools to codify IT infrastructure management—much in the same way that developers had codified their testing, build, and CI practices and version-controlled those files—making them the first drivers of the "infrastructure as code" and DevOps movements.

Their big advantage is the ability to create immutable infrastructure: which means that instead of continually updating existing machines' configurations, operators will just decommission the machines, then spin up brand new machines and graft updated templates from the configuration management tool onto the blank machine, giving it a fresh start every time a change is made. This may sound extreme, but in the complex world of IT operations, it's better to remove the possibility of strange bugs emerging as compounding updates allow entropy to creep in.

Infrastructure as Code

The full promise of infrastructure as code came with provisioning tools, which—in addition to providing templates for the configuration of infrastructure components—could also boot up the infrastructure itself. This also became much easier as more organizations started using cloud infrastructure that could be spun up with the push of a button.

Provisioning systems include CloudFormation for AWS and ARM for Azure, and Terraform, which is one of the most popular

Provisioning systems include CloudFormation for AWS only, ARM for Azure only, and Terraform, which is one of the more popular open-source options because it can provision across any major cloud provider. Because of these systems' ability to manage the complexity of infrastructure at scale, they are becoming one of the primary interfaces that operators use to interact with their infrastructure.

***Note about "everything as code" usage in developer parlance*:** While this article takes a more literal approach to the emerging term, *everything as code*, the term more commonly references the growing interest around flexible state engines in the category of infrastructure as code, especially Terraform, which can be used to automate more than just infrastructure. Using plugin-like providers, it can automate company tools like 1Password, Google Calendar, or automate actions in CI/CD pipelines using the GitHub provider. This makes it possible to centralize a lot more custom automation in one place.

Container / Cluster orchestrators

Container orchestrators take the benefits of containers and make it easy to scale up their deployment and management. VMs can also benefit from this approach.

At their core, they are cluster schedulers—intelligent systems that optimize the usage of cloud/virtual infrastructure as if your applications were Tetris pieces. Essentially, a scheduler takes a configuration file with some rules and then goes and deploys any number of applications, services, or components (e.g. load balancers, etc) on containers or VMs in the most optimal way for a given infrastructure according to your parameters.

Orchestrators include Kubernetes, which has a host of additional features, and Nomad, which has a more focused scheduling use case.

Security & Compliance

Security shouldn't be treated as a phase after development, but too often, in many organizations it is. The general consensus from leaders in the IT security space is that security and compliance should include actions—some manual, some automated—that are *baked in* to your software production lifecycle.

Security as code

Security as code is a bit broad to be taken literally (like many of the "as code" categories here), but it's helpful to think of it as inserting automated security actions into each area of software production. These automatable actions could include a number of things:

Planning & Design

- Generating initial threat models
- Building templates for automating security test generation

Development & Testing

- Security-focused static analysis (SAST) - SonarQube, FindBugs, Checkmarx
- Dynamic security test automation, such as DAST and penetration testing (pen testing) - FindBugs, Checkmarx, OWASP ZAP, ThreadFix
- Third-party component risk analysis and alerting - Snyk, BlackDuck, WhiteSource
- Credential/secrets management - HashiCorp Vault, AWS Secrets Manager

Deployment & In Production

- Configuration checks for applications and infrastructure code - Burp, Hardening.io
- Monitoring and alerting - Snort, OSSEC
- Production chaos testing and pen testing - Gauntlt, Mittn, Simian Army

Using all of the previously mentioned *as code* practices will also improve security as a byproduct since all of these automations create authorized, repeatable, and auditable actions that take a lot of human error and entropy out of the equation—leaving less chance to create security holes.

Policy as code / Compliance as code

Policy as code is similar to the gates in the CI/CD pipelines that would stop a code merge if certain tests failed or the build broke. In fact, most of the policy as code frameworks integrate right into your version control system. Some policy as code involves managing security and security-related compliance at scale, so in that sense it's partially security as code. But policy as code can also be used for various other engineering governance rules as well.

Maybe you'd like to limit the types or amount of infrastructure a certain user can provision. Or you'd like to require engineers to use tags for the resources they're creating. There are countless possibilities.

There are open-source policy as code frameworks like Open Policy Agent and Chef InSpec. Some companies are building policy as code into their infrastructure management products, like HashiCorp's Sentinel framework, allowing deeper, more specialized policy creation.

“As code” should mean added efficiency, safety, or insight

I have no doubt that we’ll see more technologies designated as “[thing] as code.” The important thing to remember is that it’s all just automation and abstraction. The real understanding of a new tool will come from understanding the exact areas of software production that it automates, and whether or not that’s anything new.

Automating things according to a piece of configuration code can be beneficial in a number of ways. When something is “codified” you get:

- **Linting, static analysis, and alerting** to enforce code consistency in a large organization
- **Testing** to make sure the automation works solidly
- **A common language** to allow anyone with knowledge of the automation language to collaborate on how it’s built
- **Separation of concerns** to make it easy to give pre-packaged, best-practice modules of automation to other groups who shouldn’t need to be experts to use the automation
- **A model** to give you an overarching conceptual map of the section of pipeline that you’re automating.

Hopefully this article serves as a starting point to think about where you might want to introduce and configure some new automation in your own organization.

If any of my descriptions or delineations of these “as code” sections seem incorrect or off to you, let me know in the comments or message me on LinkedIn (fastest way to notify me) and I can update this.

PLATFORM

LOADING

. . . comments &*more!*

ABOUT

Careers
Contact
Cookies
Emails
Help
Privacy
Terms

READ

Archive
Leaderboard
Noonification
Signup
Tech Brief
Tech Tags
Top Stories

WRITE

Distribution
Editor Tips
Guidelines
New Story
Perks
Prompts
Why Write

PARTNER

Billboard
Brand Publishing
Case Studies
Contests
Niche Marketing
Newsletter
Writing Contests

THE HACKERNOON NEWSLETTER

Quality Weekly Reads About Technology Infiltrating Everything

name@company.com Subscribe*free*

Yes, I agree to receive emails about tech eating the world.