Automating Infrastructure with Cisco

7.1.1

Introduction to Automating Infrastructure

In this module you will learn about automation. Automation is using code to configure, deploy, and manage applications together with the compute, storage, and network infrastructures, and the services on which they run.

The tools in this area include Ansible, Puppet, and Chef, to name a few. For automation with Cisco infrastructure, the platforms can integrate with those common tools, or provide direct API access to the programmable infrastructure. Whether in a campus/branch configuration, in your own data center, or as a service provider, there are Cisco tools that do more than network configuration management.

When you understand what automation is and what it can do for you, you'll be ready to visit the Cisco DevNet Automation Exchange to explore solutions that can work for you.

7.1.2

Cisco Automation Solutions

There are several use cases for automation for the network. Depending on the operational model you want to follow, you have choices in how to programmatically control your network configurations and infrastructure. Let’s look at Cisco automation through the DevNet Automation Exchange to understand the levels of complexity and choices available.

Walk: Read-only automation

Using automation tools, you can gather information about your network configuration. This scenario offers answers to the most basic and common question you can ask, which is "What changed?"

By gathering read-only data, you minimize risk of causing a change that could break your network environment. Using GET requests is also a great way to start, by writing code solutions to data collection tasks. Plus, you can use a read scenario to audit configurations and do the next natural step, which is to put the configuration back into compliance. In the Automation Exchange, this shift is categorizes as a walk-run-fly progression.

Run: Activate policies and provide self-service across multiple domains

With these "Run stage" automation scenarios, you can safely enable users to provision their own network updates. You can also automate on-boarding workflows, manage day-to-day network configurations, and run through Day 0, Day 1, and daily (Day n) scenarios.

Fly: Deploy applications, network configurations, and more through CI/CD

For more complex automation and programmable examples, you want to go to the Fly stage of the DevNet Automation Exchange. Here you can get ahead of needs by monitoring and proactively managing your users and devices while also gaining insights with telemetry data.

There are many use cases for infrastructure automation, and you are welcome to add to the collection in the DevNet Automation Exchange.

7.1.3

Why Do We Need Automation?

Enterprises compete and control costs by operating quickly and being able to scale their operations. Speed and agility enable the business to explore, experiment with, and exploit opportunities ahead of their competition. Scaling operations lets the business capture market share efficiently and match capacity to demand.

Developers need to accelerate every phase of software building: coding and iterating, testing, and staging. DevOps practices require developers to deploy and manage apps in production, so developers should also automate those activities.

Below are some of the risks that can be incurred in manually-deployed and -managed environments.

Disadvantages of manual operations

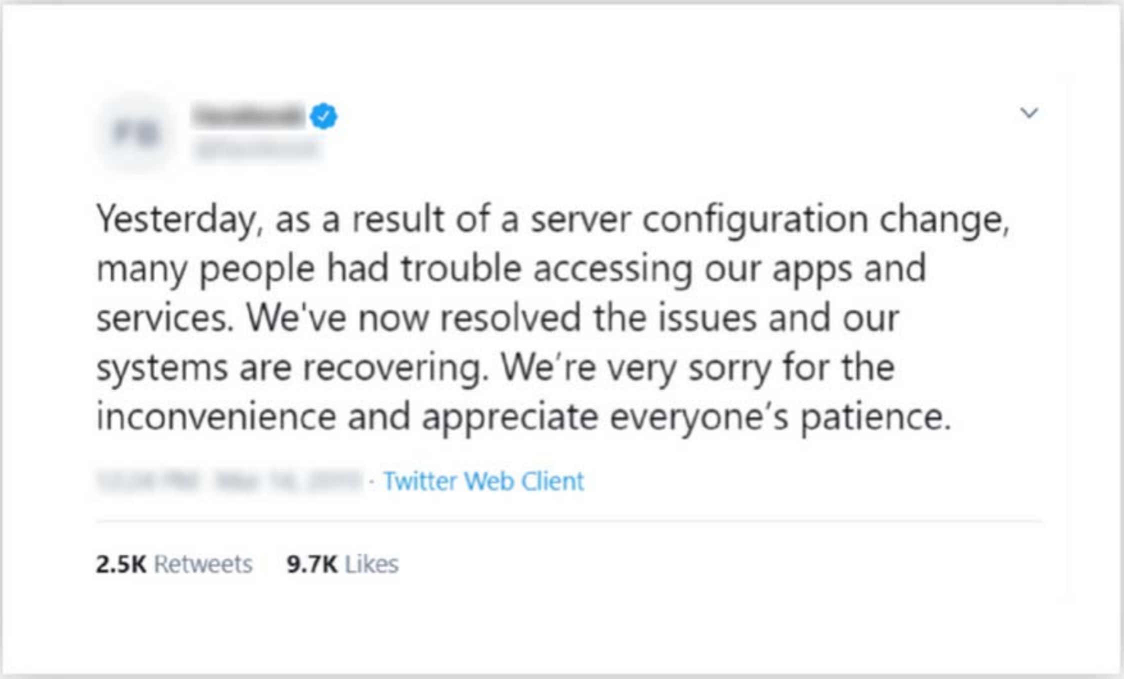
Building up a simple, monolithic web application server can take a practiced IT operator 30 minutes or more, especially when preparing for production environments. When this process is multiplied by dozens or hundreds of enterprise applications, multiple physical locations, data centers and/or clouds; manual processes will, at some point, cause a break or even a network failure. This adds costs and slows down the business.

Manual processes such as waiting for infrastructure availability, manual app configuration and deployment, and production system maintenance, are slow and very hard to scale. They can prevent your team from delivering new capabilities to colleagues and customers. Manual processes are always subject to human error, and documentation meant for humans is often incomplete and ambiguous, hard to test, and quickly outdated. This makes it difficult to encode and leverage hard-won knowledge about known-good configurations and best practices across large organizations and their disparate infrastructures.

Financial costs

Outages and breaches are most often caused when systems are misconfigured. This is frequently due to human error while making manual changes. An often-quoted Gartner statistic (from 2014) places the average cost of an IT outage at upwards of $5,600 USD per minute, or over $300,000 USD per hour. The cost of a security breach can be even greater; in the worst cases, it represents an existential threat to human life, property, business reputation, and/or organizational survival.

Financial Costs of Server Outages



Yesterday, as a result of a server configuration change, many people had trouble accessing our apps and services. We've now resolved the issues and our systems are recovering. We're very sorry for the inconvenience and appreciate everyone's patience.

· Twitter Web Client

2.5K Retweets **9.7K** Likes

Dependency risks

Today's software ecosystem is decentralized. Developers no longer need to build and manage monolithic, full-stack solutions. Instead, they specialize by building individual components according to their needs and interests. Developers can mix and match the other components, infrastructure, and services needed to enable complete solutions and operate them efficiently at scale.

This modern software ecosystem aggregates the work of hundreds of thousands of independent contributors, all of whom share the benefits of participating in this vast collaboration. Participants are free to update their own work as needs and opportunities dictate, letting them bring new features to market quickly, fix bugs, and improve security.

Responsible developers attempt to anticipate and minimize the impact of updates and new releases on users by hewing closely to standards, deliberately engineering backwards compatibility, committing to provide long-term support for key product versions (e.g., the "LTS" versions of the Ubuntu Linux distribution), and other best practices.

This ecosystem introduces new requirements and new risks:

- Components need to be able to work alongside many other components in many different situations (this is known as being flexibly configurable) showing no more preference for specific companion

components or architectures than absolutely necessary (this is known as being unopinionated).
- Component developers may abandon support for obsolete features and rarely-encountered integrations. This disrupts processes that depend on those features. It is also difficult or impossible to test a release exhaustively, accounting for every configuration.
- Dependency-ridden application setups tend to get locked into fragile and increasingly insecure deployment stacks. They effectively become monoliths that cannot easily be managed, improved, scaled, or migrated to new, perhaps more cost-effective infrastructures. Updates and patches may be postponed because changes are risky to apply and difficult to roll back.

7.1.4

Why Do We Need Full-Stack Automation?

Why do we need full-stack automation?

Infrastructure automation can deliver many benefits. These are summarized as speed, repeatability, and the ability to work at scale, with reduced risk.

Automation is a key component of functional software-defined infrastructure and distributed and dynamic applications. Below are additional benefits of full-stack automation.

Self-service

Automated self-service frameworks enable users to requisition infrastructure on demand, including:

- Standard infrastructure components such as database instances and VPN endpoints
- Development and testing platforms
- Hardened web servers and other application instances, along with the isolated networks and secured internet access that make them useful, safe, and resistant to errors
- Analytics platforms such as Apache Hadoop, Elastic Stack, InfluxData, and Splunk

Scale on demand

Apps and platforms need to be able to scale up and down in response to traffic and workload requirements and to use heterogeneous capacity. An example is burst-scaling from private to public cloud, and appropriate traffic shaping. Cloud platforms may provide the ability to automatically scale (autoscale) VMs, containers, or workloads on a serverless framework.

Observability

An observable system enables users to infer the internal state of a complex system from its outputs. Observability (sometimes abbreviated as *o11y*) can be achieved through platform and application monitoring. Observability can also be achieved through proactive production testing for failure modes and performance issues. But, in a dynamic operation that includes autoscaling and other application behaviors, complexity increases, and entities become ephemeral. A recent report by observability framework provider DataDog, states that the average lifetime of a container under orchestration is only

12 hours; microservices and functions may only live for seconds. Making ephemeral entities observable and testing in production are only possible with automation.

Automated problem mitigation

Some software makers and observability experts recommend what is known as Chaos Engineering. This philosophy is based on the assertion that failure is normal: as applications scale, some parts are always failing. Because of this, apps and platforms should be engineered to:

- Minimize the effects of issues: Recognize problems quickly and route traffic to alternative capacity, ensuring that end users are not severely impacted, and that on-call operations personnel are not unnecessarily paged.
- Self-heal: Allocate resources according to policy and automatically redeploy failed components as needed to return the application to a healthy state in current conditions.
- Monitor events: Remember everything that led to the incident, so that fixes can be scheduled, and post-mortems can be performed.

Some advocates of Chaos Engineering even advocate using automation tools to cause controlled (or random) failures in production systems. This continually challenges Dev and Ops to anticipate issues and build in more resilience and self-healing ability. Open source projects like Chaos Monkey and "Failure-as-a-Service" platforms like Gremlin are purpose-built for breaking things, both at random and in much more controlled and empirical ways. An emerging discipline is called "failure injection testing."

7.1.5

Software-Defined Infrastructure: A Case for Automation

Software-defined infrastructure, also known as cloud computing, lets developers and operators use software to requisition, configure, deploy, and manage bare-metal and virtualized compute, storage, and network resources.

- Cloud computing also enables more abstract platforms and services, such as Database-as-a-Service (DaaS), Platform-as-a-Service (PaaS), serverless computing, container orchestration, and more.
- Private clouds let businesses use expensive on-premises hardware much more efficiently.
- Public and hosted private clouds let businesses rent capacity at need, letting them move and grow (or shrink) faster, simplifying planning and avoiding fixed capital investment.

Benefits of cloud paradigms

- **Self-service (platforms on demand) -** Cloud resources can be available within hours or minutes of needing them. Thus speeding all phases of development, and enabling rapid scaling of production capacity. Applications can be scaled in the public cloud region, service set, or provider that is most cost-effective.
- **Close specification, consistency, repeatability -** Developers can capture and standardize unique configurations, maintaining configurational consistency of platforms through development, testing,

staging, and production. Deploying a known-good application and configuration prevents bugs that can be introduced during manual platform configuration changes.
- **Platform abstraction -** Container technologies abstract apps and platforms away from one another, by encapsulating application dependencies and letting your containerized app run on a generically-specified host environment.

Challenges of cloud paradigms

Developers must pay close attention to platform design, architecture, and security. Cloud environments make new demands on applications. Public or private cloud frameworks have varying UIs, APIs, and quirks. This means that users cannot always treat cloud resources as the commodities they really should be, especially when trying to manage clouds manually.

Access control is critical, because cloud users with the wrong permissions can do a lot of damage to their organization's assets. Cloud permissions can be also challenging to manage, particularly in manually operated scenarios.

When cloud resources can be self-served quickly via manual operations, consumption can be hard to manage, and costs are difficult to calculate. Private clouds require frequent auditing and procedures for retiring unused virtual infrastructure. Public cloud users can be surprised by unexpected costs when pay-by-use resources are abandoned, but not torn down.

Many of these challenges can be addressed through automation.

7.1.6

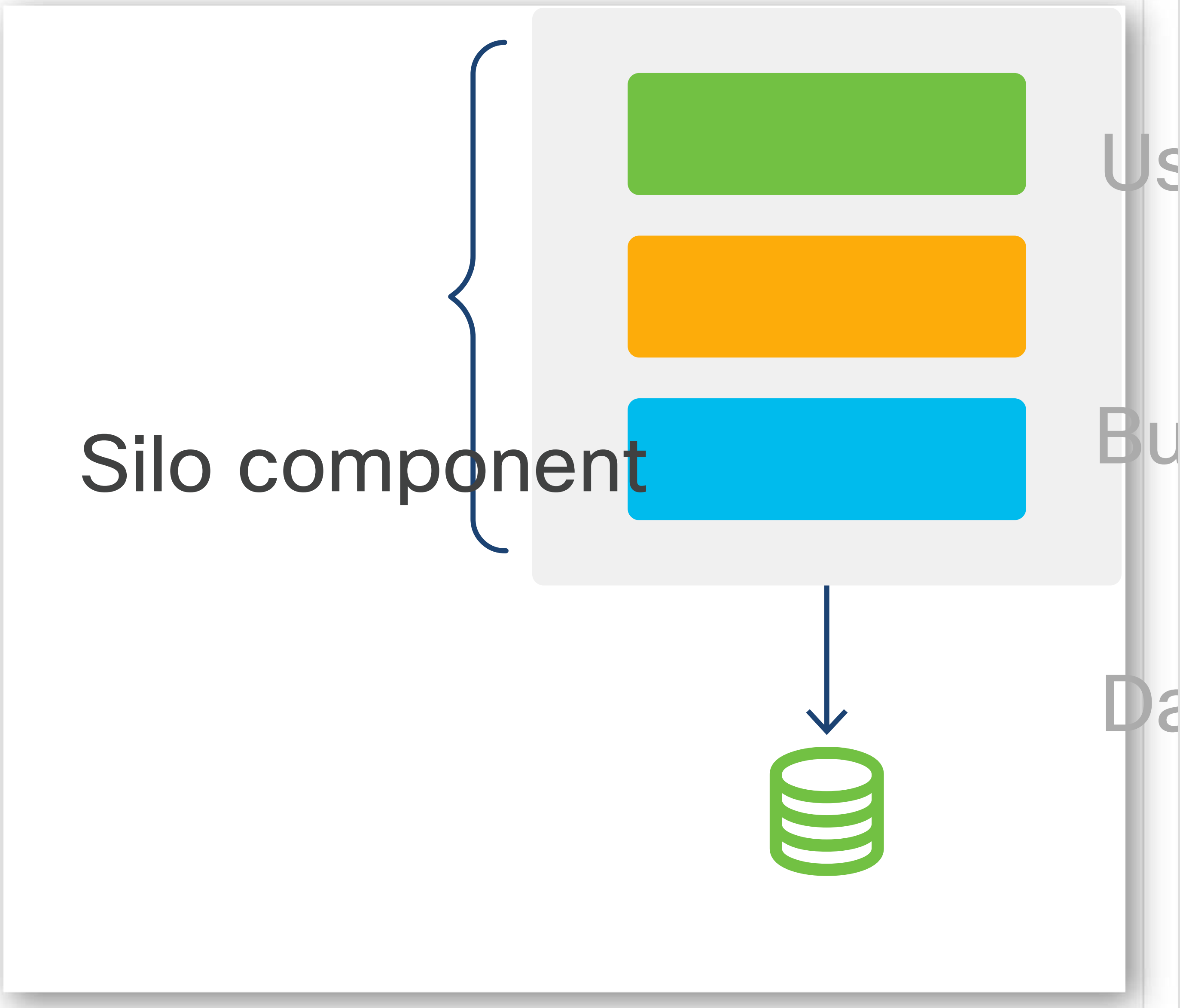
Distributed and Dynamic Applications: Another Case for Automation

Large-scale enterprise and public-facing applications may need to manage heavy and variable loads of traffic, computation, and storage.

- The applications need to provide a good experience to their user base.
- They need to be resilient, highly available, protect user data integrity and security, and comply with local regulations about where and how data is stored.
- They need to grow (and shrink) efficiently to meet business needs, exploit trends, and run cost-effectively.

Single-server, "monolithic" application architectures, while conceptually simple, do not serve these needs very well. One server is a single point of failure, limits performance and capacity, and has limited upgrade capability. Duplicating the single server can increase capacity for very simple apps, but does not work for applications requiring data consistency across all instances. And it will not protect user data if there is a failure on the server on which their data resides.

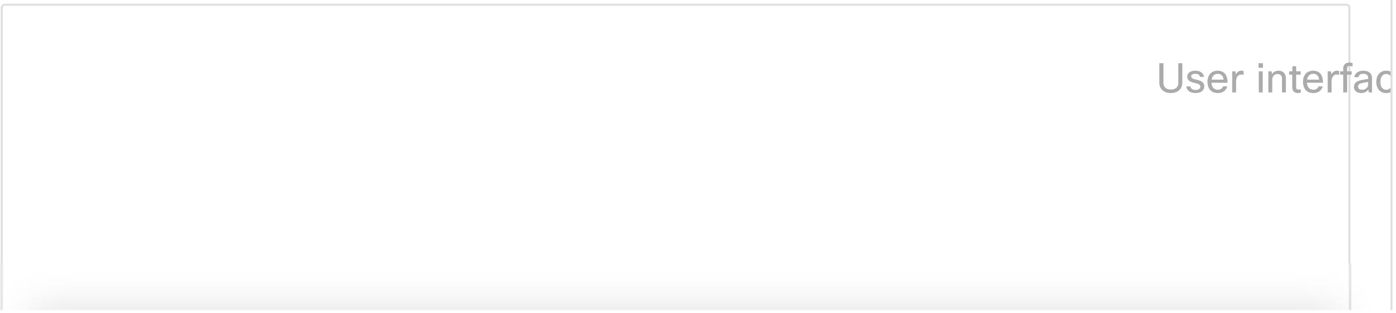
For these and other reasons, modern application architectures are increasingly distributed. They are built up out of small and relatively light components that are sometimes called microservices. These components may be isolated in containers, connected via discovery and messaging services (which abstract network connectivity) and backed by resilient, scalable databases (which maintain state).

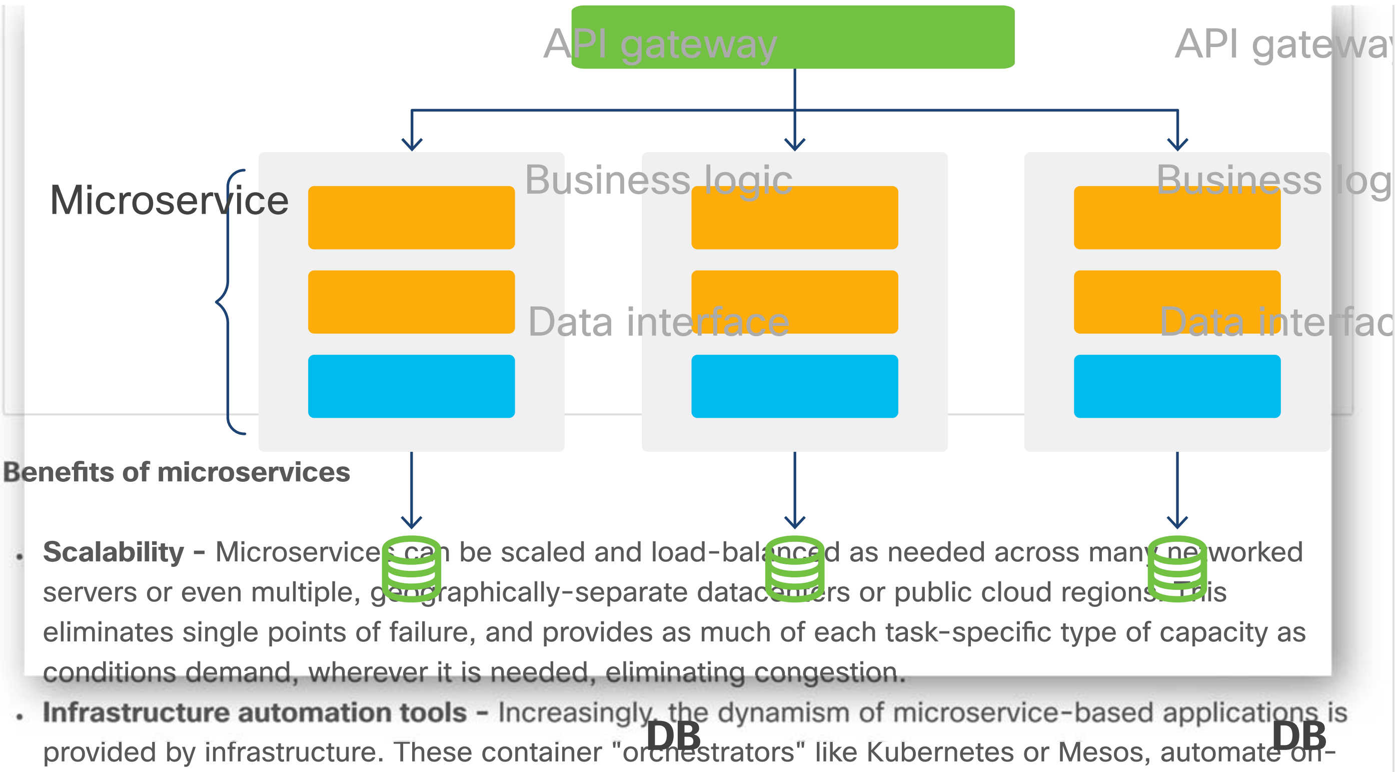


Monolithic applications can only be scaled by duplicating the whole application on additional machines.

Microservices-based applications can (in theory) be scaled out service-by-service, as needed to meet traffic or performance demands. This architecture helps obtain the greatest benefit from hyperconverged infrastructure, which enables fine-tuning of compute, storage, and network connectivity and services to match dynamic application load requirements.

Microservices-Based Applications





Benefits of microservices

- **Scalability -** Microservices can be scaled and load-balanced as needed across many networked servers or even multiple, geographically-separate datacenters or public cloud regions. This eliminates single points of failure, and provides as much of each task-specific type of capacity as conditions demand, wherever it is needed, eliminating congestion.
- **Infrastructure automation tools -** Increasingly, the dynamism of microservice-based applications is provided by infrastructure. These container "orchestrators" like Kubernetes or Mesos, automate on-demand scaling, self-healing, and more.

Challenges of microservices

- **Increased complexity -** Microservices mean that there are many moving parts to configure and deploy. There are more demanding operations, including scaling-on-demand, self-healing and other features.
- **Automation is a requirement -** Manual methods can not realistically cope with the complexity of deploying and managing dynamic applications and their orchestrator platforms, with their high-speed, autonomous operations and their transitory and ephemeral bits and pieces.

7.1.7

Automating Infrastructure Summary

These business and technical needs, trends, and dynamics, encourage developers and operators to use automation everywhere for the following tasks:

- Manage all phases of app building, configuration, deployment and lifecycle management. This includes coding, testing, staging, and production.
- Manage software-defined infrastructures on behalf of the applications you build.
- Alongside your applications, to preserve, update, and continually improve the automation code. This code helps you develop, test, stage, monitor, and operate our apps at production scales, and in various environments. You can increasingly treat all this code as one work-product.

7.0 7.2

Introduction to Infrastructure and Automati... DevOps and SRE