Carnegie Mellon University
Software Engineering Institute

INFRASTRUCTURE AS CODE–FINAL REPORT

John Klein, PhD and Douglas Reynolds

December 2018

1 Introduction

This report concludes work on Research Project 6-18_518 *Feasibility of Infrastructure as Code*, summarizing the problem addressed by the research, the research solution approach, and results.

1.1 Background

1.1.1 What Is Infrastructure as Code?

Infrastructure as code (IaC) is a set of practices that use "code (rather than manual commands) for setting up (virtual) machines and networks, installing packages, and configuring the environment for the application of interest" [3]. The infrastructure managed by this code includes both physical equipment ("bare metal") and virtual machines, containers, and software-defined networks. This code should be developed and managed using the same processes as any other software; for example, it should be designed, tested, and stored in a version-controlled repository.

Although information technology (IT) system operators have long employed automation through ad hoc scripting of tasks, IaC technology and practices emerged with the introduction of cloud computing, and particularly infrastructure-as-a-service (IaaS) technology. In an IaaS-based environment, all computation, storage, and network resources are virtualized and must be allocated and configured using application programming interfaces (APIs). While cloud service providers furnish management consoles that layer an interactive application on top of the APIs, it is not practical to use a management console to create a system with more than a couple of nodes. For example, creating a new virtual machine (VM) using the Amazon Web Services management console requires stepping through at least five web forms and filling in 25 or more fields. VMs are created and torn down many times every day during development and test, so performing these tasks manually is not feasible.

IaC technology emerged to address this issue of high-frequency creation and destruction of environments that contain many VMs, even thousands in the case of Internet-scale services. IaC tools generally comprise a scripting language to specify the desired system configuration and an orchestration engine that executes the scripts and invokes the IaaS API. The scripting languages are often based on existing programming languages; for example, the Chef tool[1] bases its scripting language on Ruby, and the Ansible tool[2] uses YAML for scripting. The execution engines are extensible to support many different IaaS APIs to allocate and configure virtual resources. Some tools, such as Chef and Puppet,[3] put a small client (or agent) application on each VM to facilitate installation and

[1] https://www.chef.io
[2] https://www.ansible.com
[3] https://www.puppet.com

configuration of software, while other tools such as Ansible do not place any custom software on the VMs. (Ansible connects to the newly created VM using the standard Secure Shell (SSH) protocol, and it does require that Python is included in the VM base image, which is a typical configuration for Linux nodes.)

1.1.2 What Does Infrastructure as Code Enable?

IaC practices and technology enable several capabilities that are being used today in the software lifecycle practices of leading organizations. Note that these capabilities build on each other, with automation at the core.

The most obvious capability that IaC enables is automated configuration and deployment of systems and software into an environment. Automating this process increases efficiency and provides repeatability.

As we noted above, the infrastructure code used for IaC should be stored in a version-controlled repository. This enables robust versioning of a deployed infrastructure: Any version of the infrastructure can be created using the IaC code corresponding to the desired version. Together, automation and versioning provide the capability to efficiently and reliably recreate a particular configuration. This can be used to roll back a change made during development, integration, or even production and to support trouble-ticket recreation and debugging.

IaC code can be shared across development, integration, and production environments. This improves *environment parity* and can eliminate scenarios where software works in one developer's environment but not for another developer, or scenarios where software works in development but not in the integration or production environment.

Finally, IaC can enable an IT operations practice called *immutable infrastructure*. In a traditional operations approach, infrastructure and application software is installed on system nodes. Over time, each node is individually patched, software is updated, and network and other configuration parameters are changed as needed. *Configuration drift* may ensue, for example, as the patch level varies across nodes. In some cases, nodes can be recreated only from a backup, with no ability to reconstruct the configuration from scratch. In an immutable infrastructure, patches, updates, and configuration changes are never applied to the deployed nodes. Instead, a new version of the IaC code is created with the modifications that reflect the needed changes to the deployed infrastructure and applications. Environment parity allows the new version to be tested in development and integration environments prior to production, and environment versioning allows the new changes to be rolled back if there is an unexpected issue after deploying to production.

IaC practices and technology may also enable capabilities to improve future software lifecycle practices. For example, IaC is starting to provide portability across IaaS cloud service providers. In the future, IaC artifacts may support analysis or provide evidence for assurance processes such as the Risk Management Framework. Finally, IaC could support cyber defense approaches, such as moving target defense, by transforming the IaC artifacts to deploy an equivalent system with a different attack surface.

1.1.3 Relationship Between IaC, Agile, and DevOps

IaC is one of many modern software development practices; it is loosely related to Agile software development and more closely related to DevOps.

Agile software development is a very broad set of practices organized around the tenets of the Manifest for Agile Software Development [1]. Most relevant here are the agile values of *working software* and *responding to change*. By automating the creation of execution environments, IaC practices promote these agile values.

DevOps is the integration of *Dev*elopment and *Op*erations processes to "reduce the time between committing a change to a system and the change being placed into production, while ensuring high quality" [3]. Continuous integration and continuous delivery are core DevOps processes that depend on IaC. Furthermore, IaC promotes the goals of DevOps. Automation of deployment reduces cycle time and improves the quality of the deployment delivery mechanism. Environment parity contributes to improving overall quality of the software that is deployed, by ensuring that tests in the development and integration environments reflect the conditions of the production environment.

However, as we discussed above, IaC enables capabilities that are unrelated to either Agile practices or DevOps. For this reason, we do not consider IaC to be a subset of either of these. Figure 1 shows IaC intersecting both Agile practices and DevOps, but also extending beyond the scope of both of them.

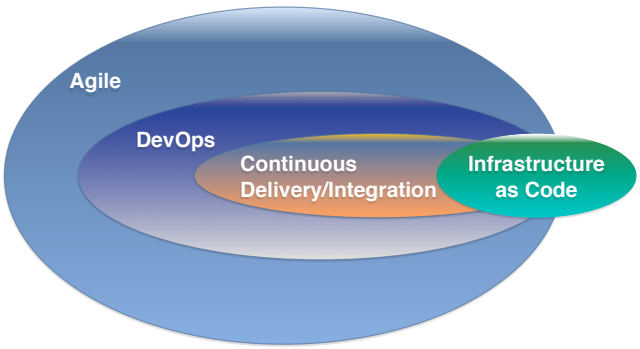

Figure 1: Relationship of IaC to Agile and DevOps

1.2 IaC and DoD Software Sustainment

The SEI's working definition of software sustainment is "the processes, procedures, people, material, and information required to support, maintain, and operate the software aspects of a system" [6]. While the Department of Defense (DoD) generally acquires custom software by contracting with a commercial organization to design and develop that software, the DoD often sustains that software using organic resources. DoD organic software sustainment is performed by diverse organizations such as

- Navy Surface Weapons Centers, e.g., Philadelphia Division
- Army Software Engineering Directorates, e.g., AMRDEC SED and CERDEC SED

- Air Force Software Maintenance Groups, e.g., 309th SMXG and 76th SMXG

DoD acquirers generally ensure that the government has data rights to the software source code and the scripts or other artifacts needed to build and install the software. Beyond that minimum baseline, the government's data rights in other artifacts is highly variable, and sustainers may or may not have access to

- architecture documentation
- unit and integration test scripts and software test fixtures
- ad hoc scripts or other aids for software deployment or configuration
- IaC artifacts

In our discussions with DoD sustainers, a common approach to transition from the development contractor to DoD sustainer is for the contractor to provide a "golden image," along with instructions for basic system configuration. The golden image is a backup image of the system taken after the contractor delivered the system to the DoD.

Although a golden image allows a sustainer to install the software, it can be used *only* to install a single version of the infrastructure and application software. Furthermore, the image is opaque; it does not provide any visibility into the system's software. Neither can it be evolved directly as changes are made to the infrastructure and application.

IaC can provide sustainers with a "safety net" that allows them to efficiently gain knowledge about a system by performing controlled experiments: making changes to the software and observing the results of those changes. The automation and versioning provided by IaC technology allow sustainers to roll back unsuccessful experiments and to capture and roll forward successful changes in an incremental, agile fashion.

In addition to supporting experimentation, IaC artifacts specify the as-implemented deployment structure of the system, which can be checked for conformance with the as-designed architecture documentation. Deviations can be added to the backlog for remediation in a future release, or at the very least, noted to aid in future operational support of the software system. This knowledge is important in the DoD context, as sustainers may change several times during the system's lifecycle.

Aside from the benefits that sustainers gain from IaC adoption, the DoD is calling for broad adoption of DevOps. In 2018, the Defense Innovation Board (DIB) issued the "10 Commandments of Software," with the fourth being "Adopt a DevOps culture for software systems" [5]. Another of the DIB's commandments is "Automate testing of software to enable critical updates to be deployed in days to weeks, not months or years," which depends on deployment automation such as provided by IaC. The result is that DoD sustainers are both pulled toward IaC by its benefits *and* pushed toward it by these top-down initiatives.

Successful IaC adoption by DoD software sustainers requires a broad set of skills and knowledge. First, DevOps has primarily a process focus, with technology support as an important but secondary concern [3], so there are the challenges of developing new processes. Second, the scope of DevOps converges development and operations, requiring knowledge of how infrastructure software works to support application software. This includes a level of proficiency in operating systems, middleware, and networking. Third, IaC technology is relatively new and is evolving rapidly. There are several

competing core products (e.g., Chef, Puppet, SaltStack, or Ansible), and each of those has its own ecosystem of supporting products. Much of the scripting (the actual code in IaC) uses programming languages such as Ruby, which are not widely used in other parts of the DoD portfolio and so are unfamiliar to many sustainers. Finally, this is a significant change to both process and technology, and that brings its own set of challenges. Each of these issues leads to potential barriers to successful IaC adoption by DoD software sustainers, and those barriers led us to the research problem and goals that we discuss in the following section.

1.3 Research Problem and Goals

This project addresses the problem of accelerating IaC adoption among DoD software sustainment organizations. A full solution would include technology to automatically recover the deployment architecture structure of a software system and automatically create IaC artifacts, with the technology user needing no knowledge about the source system and no knowledge of IaC technology.

However, this project is a Line-enabled Exploratory New Start (LENS) with limitations in funding and schedule. In this one-year project, we sought to explore the feasibility and limits of automated deployment recovery and IaC artifact creation. Our goal was to answer this question:

> *What are the challenges and limitations to automatically generate the IaC scripts needed to instantiate a deployment that is identical to an original system deployment, including measures of the amount of manual intervention needed to perform the tasks, and the amount and type of specialized knowledge needed about the system and IaC technology?*

Our approach included developing prototype tools to inventory the system computing nodes, applying heuristic rules to make sense of the inventory, populating a model of the deployment architecture, and automatically generating IaC scripts from that model. The prototype tools we developed were released as open source software, available at https://github.com/cmu-sei/DRAT.

1.4 Report Structure

The rest of this report is structured as follows: Section 2 presents the details of our overall solution approach and the design of the prototype tools. Our results, conclusions, and future work are discussed in Section 3.

2 Approach and Prototype Design

Our research goal was related to feasibility, limitations, and challenges, so we built a set of prototype tools and tested the tools on a multi-node distributed system.

2.1 Approach

We labeled the original system as the *Source* and the system deployed using the automatically generated IaC artifacts as the *Target*. This is shown in Figure 2.

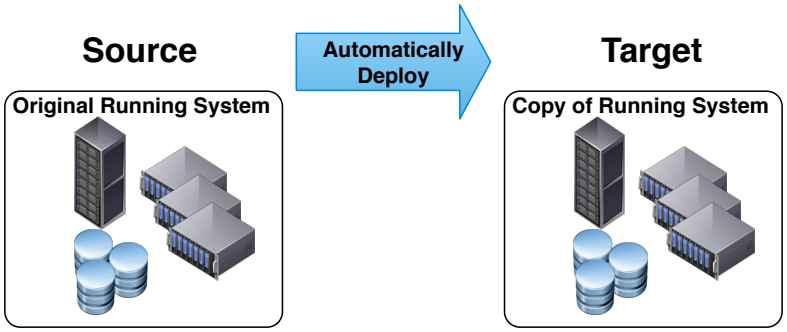

Figure 2: Project Terminology

The source system consists of a number of compute nodes with attached disk storage, accessible over a network. The compute nodes may be physical computers ("bare metal") or VMs.

Figure 3 depicts the elements of our solution and the information flows between elements.

Crawl and Inspect visits each node in the source system and inventories the package installation history and all files on the node. Ideally, this component should not require software to be installed onto the source system and should not modify the source system in any meaningful way.[4]

The Analyzer processes the inventory and applies heuristic rules to identify the source of each file. We discuss the heuristic rules in more detail in the next subsection. The prototype tool uses the metaphor of *marking* a file from the source system with the file's origin. The marking provides a measure of progress—that is, how many files have their origin identified—and guides the Generator's actions.

The output of the Analyzer is a populated Deployment Model of the system. A Deployment Model is an architecture view that relates files to nodes and directories [4] where the files reside. In our prototype tool, the model is represented in a relational database, as described below in Section 2.3.3.

The Generator uses the contents of the Deployment Model to generate artifacts used by an off-the-shelf IaC tool (e.g., Chef or Ansible) to provision and configure the target system.

2.2 Overview of Heuristics

Our approach uses heuristic rules to make sense of the file and package inventory for each node in the source system. A heuristic "is any approach to problem solving, learning, or discovery that employs a practical method, not guaranteed to be optimal, perfect, logical, or rational, but instead sufficient for reaching an immediate goal."[5] The heuristics that we used can be characterized as rules of thumb or educated guesses.

[4]The caveat on modification of the source system admits that there will be some incidental changes; for example, our remote login to the source system should be logged on the source system.
[5]https://en.wikipedia.org/wiki/Heuristic

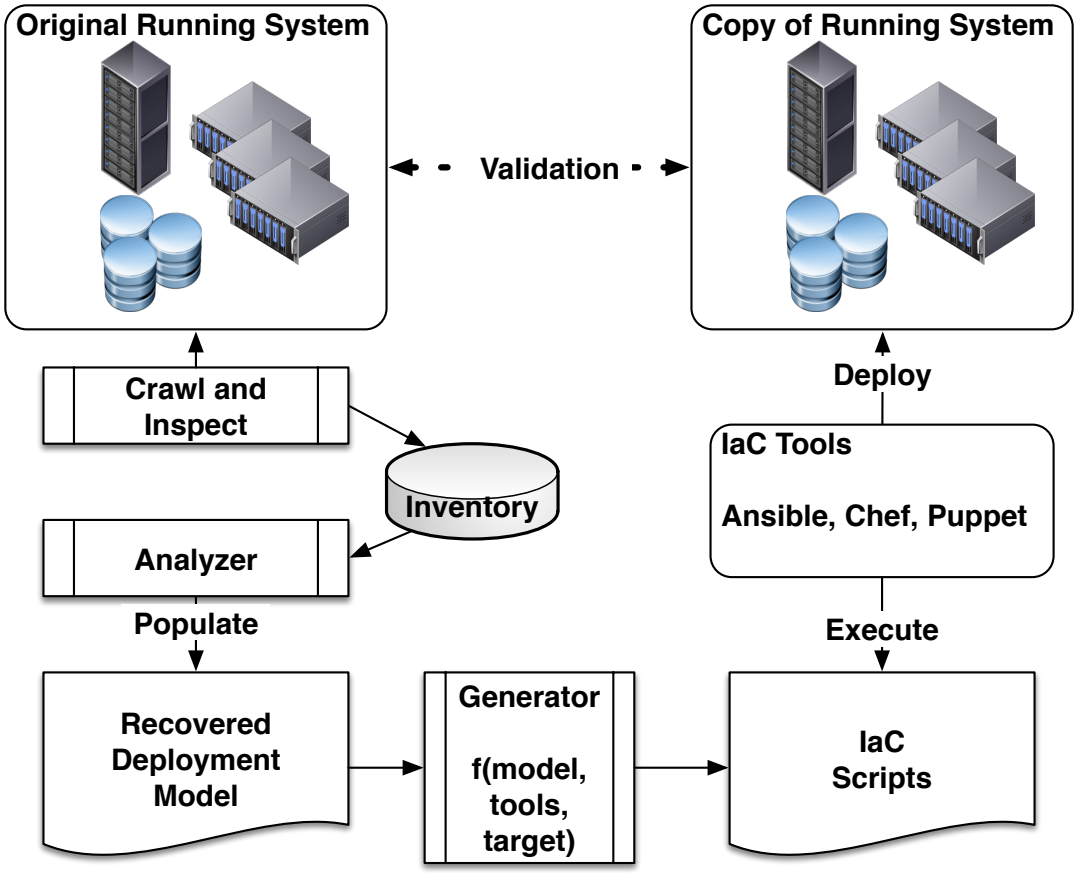

Figure 3: Solution Approach

The heuristics represent the approach that an expert would use to reason about the contents of the source system and to create the target system. Our approach is limited to source systems that run on a Linux operating system distribution, as we rely on the Linux approach that treats "everything as a file." This provides the following capabilities:

- Unlike Windows, there is no Registry in Linux. All configurations and settings are stored in files (usually text files) that can be read, parsed, and compared, rather than being hidden behind the Registry API.
- Installation of an application or service is accomplished by copying files.
- When a file (such as a configuration file) has been modified after installation, it can be reliably and efficiently detected. We compare the file hash (md5 or SHA1) to the hash included in the installation manifest.
- The Linux file system metadata includes information about whether a file is executable.

While implementation details of the heuristics may vary between Linux distributions, the approach does not rely on any distribution-specific features. This subsection presents the general strategy used by the heuristics. The implementation of the heuristics in the prototype tool is discussed below in Section 2.3.2.

We discuss each heuristic in roughly the order that it is applied during the analysis.

The first heuristic is a principle that guides the rest of the analysis:

Heuristic 0: In creating the target system, we care only about executable files on the source system, and about any files that an executable file depends on.

We use the Linux convention that executable files include scripts and library files. Executable files can depend on configuration files and on files that we label *content*. Examples of content files include the files served up by a web server and the files containing the contents of a database managed by a database server.

This heuristic excludes *ephemeral* files that may exist on the source system, such as logs and temporary files. There is no need to duplicate these on the target system.

The next heuristic is as follows:

Heuristic 1: Use the package installation history from the package manager.

There are a number of package managers used by Linux distributions, such as rpm, yum, and apt. In all cases, the package manager maintains a list of packages that have been installed on the system (and packages that have been removed). Each installed package includes a *manifest* that lists all files copied, both executable files and dependent files such as configuration files.

The package manager manifest will allow us to identify:

- files that were copied onto the source system as part of the operating system distribution. These include the Linux kernel, shell, and core services.
- files that were copied onto the source system later, as part of an installed application or service package.

This heuristic identifies the source of a majority of the files on most server systems. Service configuration files may be edited after installation to customize the service. For example, a web server configuration specifies the directory that holds the content served up by the web server, or an application server configuration specifies the directory that holds the code executed in the server. This leads us to our next heuristic:

Heuristic 2: Many Linux configuration files for common services have a regular structures. Within a configuration file, we can identify entries that specify directories, and those directories contain content files.

Linux configuration files specify values for configuration items as follows:

```
<optional-whitespace><configuration item name><whitespace><value>
```

This pattern can be easily parsed using regular expression matching, and it allowed us to easily develop rules that parsed the configuration files for specific services, using knowledge about the names of the configuration items for each service and knowledge about whether the directory value specified is a fully qualified or relative path.

The executable files that remain at this point were not copied onto the system by the package manager, and the heuristics that follow become more ad hoc. The next heuristic looks at how an executable can be started.

Heuristic 3: Executables are started either by the service manager or by an interactive user.

We first focus on the service manager side of this heuristic. Every Linux distribution includes a service manager that manages the execution of daemon or background processes. These include *systemd* in more recent distributions and *upstart* and *initd* in older distributions. For most server systems, the service manager is the principal mechanism for starting and automatically restarting these processes—the service manager runs after the operating system boots and starts and monitors the services. If a service crashes, the service manager detects that event and may restart the service.

The source system's Linux distribution and version determines which service manager is being used. Each service manager looks in a specific directory for the scripts that perform the start, stop, and other management operations for a service. This directory will contain the scripts to manage services installed by the package manager, along with scripts for services installed using ad hoc mechanisms.

A rule implementing this heuristic looks into the service manager's directory and inventories all scripts. Scripts related to services installed by the package manager are ignored. For each of the remaining scripts, the rule parses the script to find the reference to the main executable for the service. For example, in a script for the *systemd* service manager, we look for a line in the script matching this pattern:

```
<optional-whitespace>EXECSTART=<command-to-start-executable>
```

Again, this pattern is easily found and parsed using regular expression matching, and the path of the executable file can be extracted from the command.

The interactive user side of the heuristic leads to our next heuristic, which we did not explore in our prototype tool.

Heuristic 4: Interactive users will often set their PATH environment variable to include the executables that they run.

Environment variables for interactive users are set in the Linux shell customization files, often referred to as *dotfiles* because the filename begins with a "." character to hide the file in a typical directory listing. Examples are .bashrc and .bash_profile files. These files can be parsed to locate changes to the PATH environment variable, and any directories added to the PATH should be carried over to the target system.

As we noted above, the heuristics become increasingly service-specific. The Secure Shell service (*sshd*) is an example of a service that depends on content files identified through convention rather than explicit configuration. This produces our next heuristic:

Heuristic 5: For services like *sshd*, use the service's conventions to identify content files.

Continuing the *sshd* example, when an interactive user initiates an SSH connection to a system, the *sshd* service looks for the user's private key in the file:

```
~/.ssh/authorized_keys
```

For each user, these files must be carried over to the target system to allow that user to connect to the target system.

2.3 Prototype Tool Design

The prototype tool is called DRAT, an acronym for Deployment Recovery and Automation Technology. The tool is a collection of Python programs and has been released as open source, as discussed in the Section 1.3. The Python programs, along with supporting software, are packaged in a set of Docker containers to allow portability and reproducibility.

Figure 4 shows a context diagram for the tool. The source system and target system are shown on the right, and the DRAT tool running as a set of containers on another machine is shown on the left.

We assume that the source system is running Linux. Within the scope of DRAT, the Linux distribution and version determine which package manager and service manager are used. The prototype implementation was tested using only a source system running CentOS 7.x, with the *rpm* package manager and the *systemd* service manager. We think that support for CentOS 6.x would require only simple configuration changes to point to a different package repository, but this was not tested.

DRAT requires access to an instance of the source system through an SSH connection as a user with privileges to read all directories and files. Write access to the `/tmp` directory is also needed.

The operation of the prototype tool is orchestrated by the *Main* component, which sequences and coordinates the other components in the tool. The first activity is Crawl and Inspect.

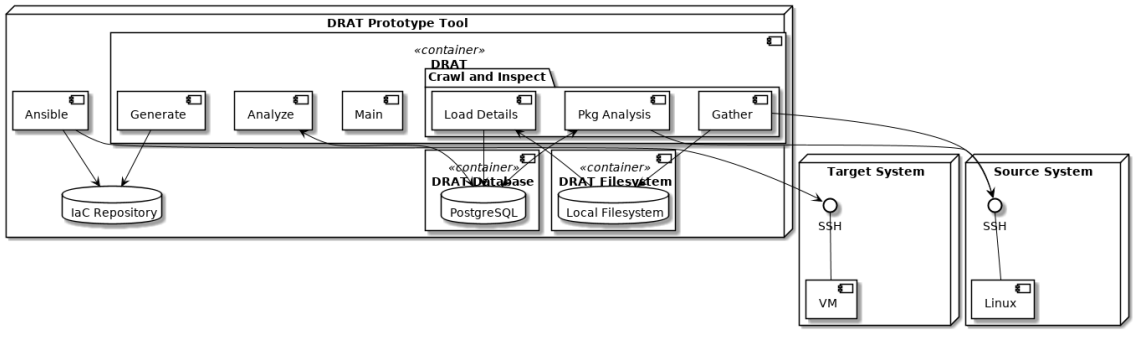

Figure 4: DRAT Prototype Tool Context Diagram

2.3.1 Crawl and Inspect

The Crawl and Inspect capability is realized by a group of components in Figure 4: *Gather*, *Load Details*, and *Package Analysis*.

The *Gather* component is driven by a configuration file that enumerates all nodes in the source system and provides the credentials needed to open the SSH connection to each node. The prototype implementation used public/private key pairs to authenticate the SSH connection; however, other authentication mechanisms could be easily substituted. The configuration file format supports environments where a bastion host (also referred to as a *jump box* or *relay*) is used to mediate access to the source system nodes.

After establishing the SSH session, the *Gather* component traverses the file system and collects the file pathname, MIME type, and hash. It also collects the package manager installation history. These actions are performed by invoking the Python interpreter on the source system with command line arguments that cause the interpreter to read a script from SSH session. *Gather* sends a Python script to the source system that performs the actions listed above, returning the results to the *Gather* component.

The *Gather* component writes the results to a flat file in the *DRAT Filesystem* container on the DRAT node. This was done to isolate the *Gather* activities from downstream Crawl and Inspect components, which was very helpful during prototype tool development and integration but is not essential to the design.

The next component in the Crawl and Inspect pipeline is *Load Details*. This component reads the contents of the flat file created by *Gather* and writes it to a PostgreSQL database, which also runs in a Docker container, as depicted in Figure 4. A relational database, such as PostgreSQL, provided query capabilities that simplified implementation of the rules for the heuristics. Also, a durable data store allows heuristic rules to be developed and evolved without having to reconnect to the source system.

During the database load process, all files are initialized to mark their origin as `Unknown`.

The final component in the Crawl and Inspect pipeline is *Package Analysis*, which implements much of Heuristic 1, discussed above, specifically for the *RPM* package manager. Each package installed on the source system includes a manifest that lists files that were added during the installation. The *Package Analysis* component iterates through each manifest. If a file that was added during package installation has not been modified (as determined by inspecting the file's hash to the manifest), then the file is marked as `PackageInstalled`. If the file was modified after installation, then the file origin is marked `PackageModified`. In both cases, the record for the file is linked to the package in the DRAT database.

Up to this point in the tool's execution, only file metadata (e.g., pathname and hash) has been retrieved from the source system. At the end of the package analysis process, the Crawl and Inspect pipeline reconnects to the source system to collect the changes to the file contents for all files marked as `PackageModified`, and those changes are stored in the DRAT database.

2.3.2 Analyze

The *Analyze* component provides an extensible framework to allow efficient creation of new heuristic rules. This includes a base class that provides an API allowing the framework to discover and execute rules and including utility functions to manage the interface with the DRAT database and logging of rule execution and actions.

The framework organizes rule execution into a series of *passes* over the DRAT database. The concept of passes was introduced early in the tool design, when there was a possibility that rule execution would be interspersed with retrieving file contents from a source system node: A set of rules would execute during a pass, and based on the results of that execution, certain file contents would be retrieved to be used by rules in a subsequent pass. This approach to interleaving rule execution with source system node access was replaced by retrieving the contents of all changed package files during the initial crawl and inspect, and then performing a single retrieval of other changed files at the completion of all rule execution. However, the concept was left in the prototype implementation as a way of enforcing an order on rule execution in the future, if needed.

All rules are registered to a pass during initialization, and then every rule is invoked for every source system node. Source system nodes are processed independently. The first action a rule takes when invoked is to decide if it should run, for the specific source system node. If, for example, the rule operates on the configuration file for a particular package (see Heuristic 2), then if that package is not installed on the source system node, the rule should not run. This modularization keeps all decisions about when a rule should execute and what the rule should do when it executes within the source code for a rule.

Storing the metadata details about each file in the source system node in the PostgreSQL relational database allows for easy queries to find particular directories and files. Parsing of the information within, for example, a configuration file is generally performed using regular expression matching.

The component also includes an initial set of heuristic rules to demonstrate each of the heuristics discussed in Section 2.2.

2.3.3 Deployment Model

The *Deployment Model* is represented in a relational schema in the DRAT Database, shown in Figure 5. Each box in the diagram represents a relational table, and the rows in each box represent columns within that table.

The *systems* table holds information about each node in the source system, including information about how to connect to the node and information about the operating system running on the node.

The *file_detail* table holds the metadata for each file on the source system nodes.

The *rpm_info* table holds metadata about each installed package, and the *rpm_detail* table stores metadata about each file that is copied to the source system node as part of the package installation process.

The *file_storage* table holds the contents of files that were modified after package installation or files that have been marked as package content and must be propagated to the target system.

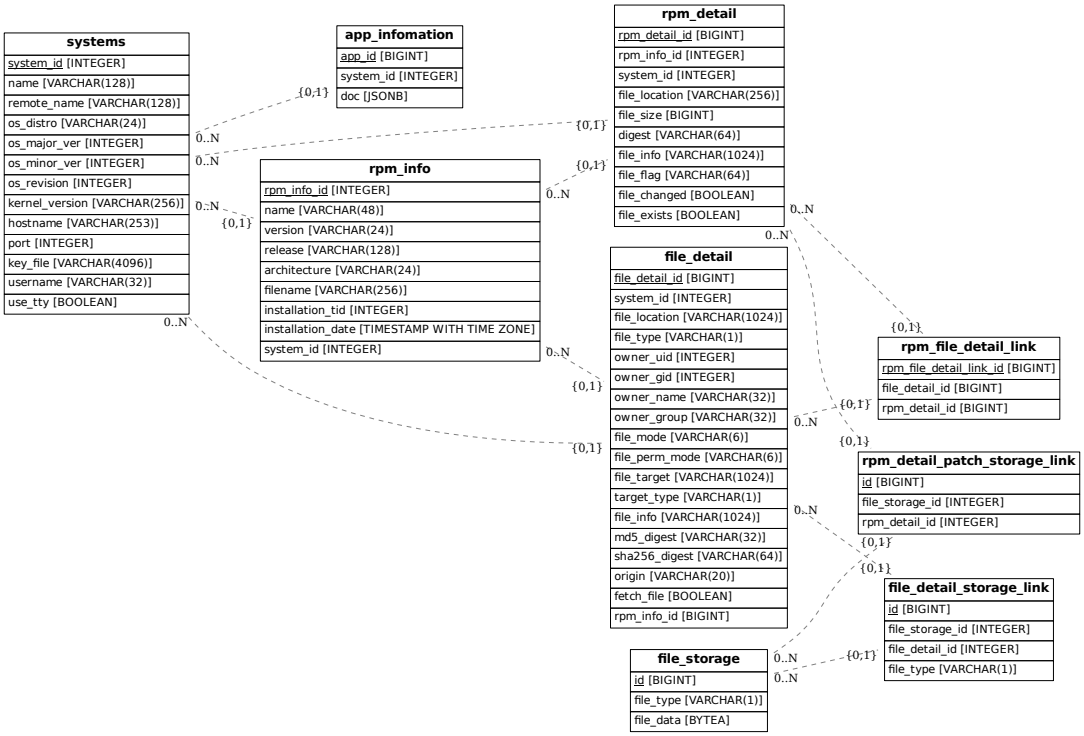

Figure 5: Deployment Model Relational Schema

The three tables *rpm_file_detail_link*, *rpm_detail_patch_storage_link*, and *file_detail_storage_link* each represent relations among the other tables.

Finally, the *app_information* table was included as an extension point to hold JSON data to support heuristic execution; however, it is was not needed for the rules implemented in the prototype.

2.3.4 Generate

IaC tools perform two main functions: First, a VM is *provisioned*, using an API provided by the virtualization management framework to allocate a new VM with a base operating system. Second, the newly created machine is *configured* by adding additional software and modifying configuration files to complete the system creation process.

The *Generate* component uses the populated *Deployment Model* to automate the generation of the IaC artifacts needed to provision and configure each node in the target system.

The prototype implementation of the Generate component creates artifacts for the Ansible[6] IaC tool. The prototype implementation used Ansible features that are comparable to other leading IaC tools such as Chef[7] and Puppet,[8] so the implementation could be easily ported to generate artifacts for another IaC tool. We chose Ansible because its YAML scripting language provides a very simple and direct syntax, allowing easy debugging of the artifact generation logic.

Each node in the source system model is considered an Ansible *role*, so the artifacts for each host are stored in subdirectories of the Ansible `roles` directory. Within the role subdirectory, each task for that role is implemented as a separate function in a separate file. For example, the package manager configuration is a separate file, and it is tagged so that it can be executed separately if needed. This modularity helps the DRAT user understand what packages are being installed on the target system, what configuration files are templated, and what additional files and directories are being created outside the scope of the package manager.

The generated artifacts include a list of all packages installed on each node of the target system. This allows the DRAT user to look across nodes and decide which packages can be grouped into a system-wide (or enterprise-wide) base operating system configuration, and those packages can then be moved to a new common role. Additionally, the configuration files are configured to be installed as templates, so the user can easily customize the values for each node.

The generated output artifacts embody best practices for Ansible [2]; for example, roles are divided by hostname, and components are stored in separate output directories. Additionally, each role's tasks are split into smaller pieces. Finally, configuration files are represented as parameterized templates, allowing efficient customization for a particular node or role. Often, automatically generated artifacts (e.g., source code) are not structured for human comprehension and evolution, assuming that humans will not directly modify the generated artifact. In contrast, the IaC artifacts output by the Generator component are structured to be modified by DRAT users, and this application of best practices provides an exemplar starting point for DRAT users (who may not be knowledgeable about IaC technology) to transition to manual evolution of IaC artifacts.

2.4 DRAT Prototype Tool Workflow

This subsection describes the workflow to use the DRAT prototype tool to analyze a source system and create a target system.

[6] https://www.ansible.com
[7] https://www.chef.io
[8] https://puppet.com

The user begins by creating the SSH access configuration file (`systems.conf`) that is used by the *Crawl and Inspect* component. This file contains one line for each node in the source system, with the following information:

- IP address or hostname of the node
- SSH port (the standard SSH port is 22)
- SSH user name
- path to SSH key for the specified user name
- user-defined label for the node

The last item allows the user to put a meaningful label on each node in the source system. This label is included in the Deployment Model.

The DRAT prototype tool supports batch and exploratory workflows. In a batch workflow, the user creates a `systems.conf` file with the complete information for all nodes in the source system, while an exploratory workflow would create a `systems.conf` file that specifies only one source system node (or any subset of the source system nodes).

In either case, the user invokes the DRAT tool, which connects to the node or nodes specified in the `systems.conf` configuration file, and executes the Crawl and Inspect and Analyze components to populate the Deployment Model. We provide a simple tool to inspect how a file or directory was marked in the Deployment Model, and the Deployment Model can be queried using SQL queries from a PostgreSQL client or another analysis package.

After completing Analysis, the user invokes the Generate component to create the IaC artifacts. Execution of the Ansible tool to use the IaC artifacts to create the target system is not included in the DRAT tool, and Ansible must be invoked directly.

3 Results and Conclusions

The DRAT prototype was tested using a source system that contained 10 nodes. SSH access was tunneled through a bastion host, which represents a typical data center or cloud configuration. The source system nodes included web servers, database servers, and custom servers.

The DRAT tool was run on a MacBook Pro laptop computer. Recall that the DRAT tool executes in Docker containers, which can run on MacOS, Windows, and Linux computers. We mention the computer platform we used simply to show that processor and memory requirements are not significant.

We did not make detailed performance measurements, but note that the *Crawl and Inspect* operation typically took a few minutes (less than 10 minutes) per node. *Analyze* and *Generate* took less than one minute per node.

The target system was created using Ansible. The Vagrant[9] open source package was used to provision VMs for the target system. Vagrant creates each new VM, and then Ansible opens an SSH connection to the newly created VM to complete the configuration process based on the IaC artifacts.

The testing results on this system were successful: The DRAT prototype was able to recreate the source system nodes.

Recall from the Introduction (Section 1) that our goal was to answer the following question:

> *What are the challenges and limitations to automatically generate the IaC scripts needed to instantiate a deployment that is identical to an original system deployment, including measures of the amount of manual intervention needed to perform the tasks, and the amount and type of specialized knowledge needed about the system and IaC technology?*

We will divide the challenges into *essential challenges* that are unavoidable and due to the approach we chose and *accidental challenges* that are introduced by the technology that surrounds the solution.

3.1 Essential Challenges

The main essential challenge we identified was the specification of analysis heuristics and rules. We incrementally defined and developed rules to handle the cases found in our test system. We found that we were able to easily recognize installed packages, and from the configuration files, extract the data needed to create IaC artifacts. However, this extraction was unique for every package: Although configuration file structure is somewhat regular (most Linux packages use one of a small number of format styles), the values in the configuration file needed to be processed uniquely for each package. We were able to use the common structure to develop a set of helper functions to make the development of the unique parameter extraction more efficient. This challenge could be addressed by a modestly-sized concentrated development effort, or as open source contributions to extend the prototype implementation.

The other essential challenge is recognizing executable files that were not installed by a package manager. We implemented half of Heuristic 3—we used the service manager configuration to identify executable files. We did not automate an approach to identify executable files that might be started by an interactive user; furthermore, we did not automate an approach to analyze each user's PATH environment variable. The convention for changing the PATH variable is to add new directories to the list, so analyzing the list requires backtracking through all changes to the PATH. Some directories added are specified in terms of other environment variables, and we could not find a simple way to dependably recreate the PATH. Additional research is needed here.

[9] https://www.vagrantup.com

3.2 Accidental Challenges

The first accidental challenge relates to authentication on the target system. The VM provisioner (Ansible, in this case) creates a default user identity and installs an SSH certificate for that user that is used by Ansible to authenticate its connection to the VM. We retrieve the `shadow`, `gshadow`, `passwd`, and `group` files from the source system, and the IaC scripts run by Ansible place copies of those files on the target system. The default user identity on the target system must match one of the user identities on the source system; otherwise, the Ansible tool may lose access to the target system during the configuration process.

The second challenge related to network interface configuration. We chose not to propagate the network interface configuration files from the source system to the target system, in order to avoid creating IP address conflicts. The provisioning process automatically configures one network interface. However, if the source system has more than one physical network interface, then the second network interface will not be configured by default by the provisioner, and the DRAT tool does not notify the user.

Finally, there are always challenges to automating system provisioning and configuration, due to the limited observability provided by the partially built system. One particular example that affected us during our integration and test related to network and HTTP proxy configuration. The Ansible IaC tool configures the target system's package manager (in this case, `yum`) and configures the URLs for the package repositories that the package manager will use. These repository URLs can point to public repositories, enterprise repositories, and/or local repositories. In any case, when Ansible starts the package manager, the package manager tries to connect to every configured repository URL. If a repository is not accessible, for example, due to DNS resolution failing or a misconfigured HTTP proxy on the target system, then the package manager exits and Ansible configuration stops, and there is little diagnostic information available to help identify and correct the misconfiguration. While we describe the issue in terms of Ansible and `yum`, it generalizes to other IaC tools and package managers.

3.3 Conclusions

Obviously, more work is needed to evolve this research prototype into a usable product for DoD sustainment organizations and other users; however, we have demonstrated that automated generation of correct IaC artifacts is feasible. The prototype tool can be employed by users with minimal knowledge of the source system to create a target system, and the generated artifacts for the target system provide a starting point for users to begin sustainment and evolution of that system.

References

[1] Agile Alliance. Manifesto for agile software development [online]. 2001. URL: https://www.agilealliance.org/agile101/the-agile-manifesto/ [cited 1 Dec 2018].

[2] Ansible. Best practices [online]. 2018. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html [cited 1 Dec 2018].

[3] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley, 2015.

[4] Paul Clements, Felix Bachman, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, second edition, 2011.

[5] Defense Innovation Board. Ten commandments of software. Draft Working Document Version 0.14, U.S. Dept. of Defense, Washington, DC, USA, April 2018. URL: https://media.defense.gov/2018/Apr/22/2001906836/-1/-1/0/DEFENSEINNOVATIONBOARD_TEN_COMMANDMENTS_OF_SOFTWARE_2018.04.20.PDF [cited 1 Dec 2018].

[6] Mary Ann Lapham. Software sustainment – now and future. *CrossTalk*, page 33, January/February 2014. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a591337.pdf [cited 1 Dec 2018].

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM19-0046