Using tools to create an environment in the Cloud.

ASU Knowledge Enterprise advances research, innovation, strategic partnerships, entrepreneurship and international development.

The **Research Technology Office** provides specialized technology solutions and services that enable, unburden and protect ASU's research community and sponsored projects.

Research Computing is the academic supercomputing facility at Arizona State University, providing high performance and high-throughput computing environments to support research and data needs.

#1 in the U.S. for innovation

#1 ASU #2 Stanford #3 MIT

– U.S. News & World Report, 2016 and 2017

ASU Knowledge Enterprise

Arizona State University

Arizona State University
HARC/CRSE2 Team

PI: Doug Jennewein (Sr. Director, Research Computing)
CRSE: Chris Kurtz (Sr. System Architect)

Science Driver: Dr. Jay Oswald
Grad Students: Omid Euclid & Jing Hu

Who I am: Chris Kurtz

- Senior Systems Architect for ASU Knowledge Enterprise
  (I design research technology systems and infrastructure)
- Formerly Director of Public Cloud Infrastructure
  for ASU Central IT (University Technology Office)
- Co-PI for NSF Campus CyberInfrastructure (CC*)
  Grant for Research Network/Science DMZ
- Cloud Research Support Engineer for HARC/CRSE2
- System Architect by choice, System Admin by “training”
  and Network Admin by “insufficient reluctance”
- Technologist, Cosplayer, Geek

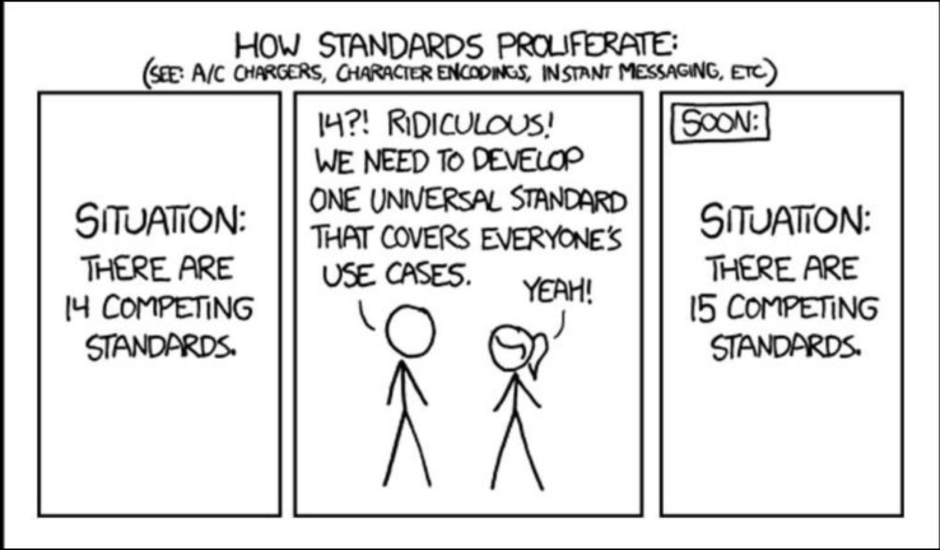
(Anything I say is my opinion and mine alone, yadda yadda)

“Using tools to create an environment in the Cloud.”

Industry standard and cloud neutral

Use tools that are not designed by the cloud providers (looking at you, CloudFormation) that can work everywhere.

(And you’re not subject to the provider killing/modifying them)


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

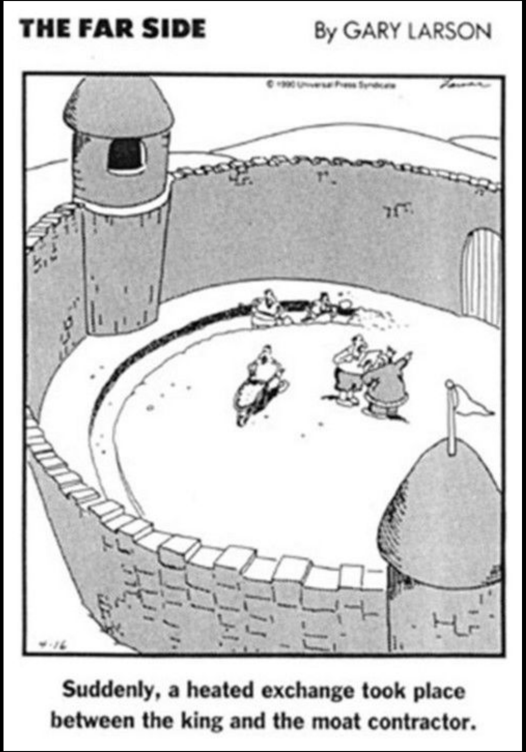
“Using industry-standard cloud-neutral tools to create an environment in the Cloud.”

Architected and Orchestrated

Whichever tools you select should allow for a reproducible and rebuildable environment.

Infrastructure (and Configuration) as Code!

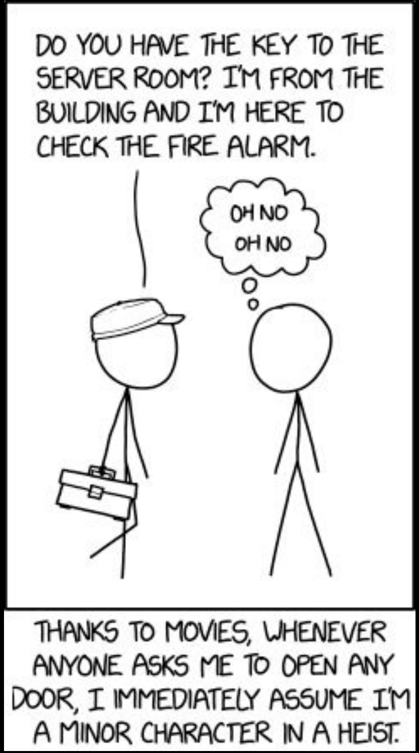
“Nice system you got there… be a real shame if something happened to it without a nice orchestrated way to get back to before someone made a critical error and rm -rf’d /”


THE FAR SIDE
By GARY LARSON
Suddenly, a heated exchange took place between the king and the moat contractor.

“Using Industry-standard cloud-neutral tools to create commonly architected and orchestrated in the Cloud.”

Secure

Reproducibility helps provides safety and security. You can test against an architected/orchestrated environment for compliance, unauthorized changes, etc. It also means recovering from a critical situation is fairly easy and relatively painless.

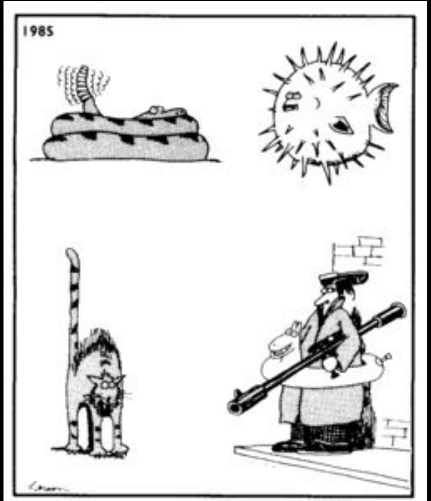

DO YOU HAVE THE KEY TO THE SERVER ROOM? I'M FROM THE BUILDING AND I'M HERE TO CHECK THE FIRE ALARM.
OH NO OH NO
THANKS TO MOVIES, WHENEVER ANYONE ASKS ME TO OPEN ANY DOOR, I IMMEDIATELY ASSUME I'M A MINOR CHARACTER IN A HEIST.

“Using Industry-standard cloud-neutral tools to create commonly architected, orchestrated, and secure environment that can easily be deployed to any of the common cloud providers.”

Any cloud

It’s a lie, we know it. Even when using good tools, every provider calls it something new and different. Some use “CloudFoo”, others acronyms, and some even use decent descriptions.

So when moving from one cloud to another you’re going to need to rewrite half your code. But it will only be half, and you’ll still be secure because your architecture is solid.



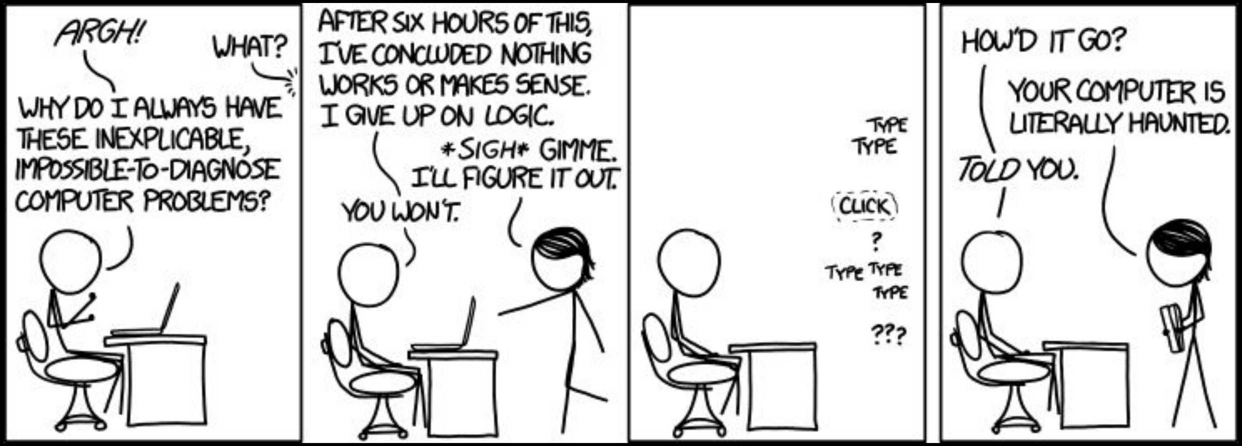
How nature says, "Do not touch."

So…

Using Industry standard cloud-neutral tools to create commonly architected, orchestrated, and secure environment that can easily be deployed to any of the common cloud providers.

Whew!

But what does that all **mean?**


ARGH!
WHAT?
WHY DO I ALWAYS HAVE THESE INEXPLICABLE, IMPOSSIBLE-TO-DIAGNOSE COMPUTER PROBLEMS?
AFTER SIX HOURS OF THIS, I'VE CONCLUDED NOTHING WORKS OR MAKES SENSE. I GIVE UP ON LOGIC.
SIGH GIMME. I'LL FIGURE IT OUT.
YOU WON'T.
TYPE TYPE
CLICK
?
TYPE TYPE TYPE
???
HOW'D IT GO?
YOUR COMPUTER IS LITERALLY HAUNTED.
TOLD YOU.

Industry standard tools to Architect and Orchestrate

Terraform

Terraform: build (& maintain) infrastructure

- Infrastructure as Code: CLI/IDE/Source control friendly
- Provider neutral (dozens and dozens of supported providers)
- Infrastructure neutral (support for many appliances)
- Free (almost certainly for anything you'll ever do)
- Designed for multiple admins
  (with some caveats about state, which Azure solves nicely)

ASU

Industry standard tools for a secure environment

Terraform

Terraform: for security

- Auditable:
  - Why did this change? (Infrastructure as Code)
  - Who did this change? (git blame)
- Change Control:
  - “The reason you have a change control window at 3am is because there is an admin at the keyboard, and they make mistakes. At 3am that mistake is less likely to be catastrophic to our users.” (Nameless Engineer)
  - Scheduled
  - Architected/Orchestrated.

Terraform Examples

```
provider "azurerm" {
  version = "=1.44.0"
}
```

First you define one or more “providers” that provide the platform for the resources you will create. The version is the version of the TF provider module. Version 2 is newer and a complete rewrite (you should use this like using Py3 vs Py2)

```
resource "azurerm_resource_group" "rc" {
  name     = "rc-rg"
  location = "West US"
}
```

Resource Groups are blocks of related infrastructure underneath a provider. These are nested inside each other (e.g. a Resource Group for Azure would contain resources such as networks, and be contained in a provider substantiation.)

```
resource "azurerm_virtual_network" "rc_network" {
  name                = "rc-Vnet"
  address_space       = ["10.0.0.0/24"]
  location            = azurerm_resource_group.rc.location
  resource_group_name = azurerm_resource_group.rc.name
}
```

Resources are contained in Resource Groups, and reference those RGs by name.

Terraform Examples con’t

```
resource "azurerm_virtual_network_gateway" "rc" {
  name                = "test"
  location            = azurerm_resource_group.rc.location
  resource_group_name = azurerm_resource_group.rc.name

  type     = "Vpn"
  vpn_type = "RouteBased"

  active_active = false
  enable_bgp    = false
  sku           = "Basic"

  ip_configuration {
    public_ip_address_id          = azurerm_public_ip.rc.id
    private_ip_address_allocation = "Dynamic"
    subnet_id                     = azurerm_subnet.rc.id
  }
}
```

Note the references to other variables in the hierarchy. As a specific example, changing a single variable (azurerm_resource_group.rc_location) can change the entire Azure Region.

This is an Azure-specific VPN, but you can see the information is generalized enough that many defaults could be set infrastructure wide and simply referenced.

Variables created are immediately referenceable, you could immediately use azurerm_public_ip.rc.id in a Security Group or ACL

Industry standard tools for configuration

ANSIBLE

Ansible: Configure and maintain the systems

- Configuration as Code: CLI/IDE/Source control friendly
- OS neutral (Any Linux or Linux-like OS. Windows too?)
- Infrastructure neutral (If you can ssh or another interactive shell and have permissions, you can use ansible to maintain it)
- Free (pretty graphs and other things for a cost...or free…)
- Designed for multiple admins

Industry standard tools for a secure environment

Ansible: for security

- Auditable:
  - Why did this change? (Configuration as Code)
  - Who did this change? (git blame)
- Enforced configuration: configuration stays the same but is easy to update infrastructure wide.
- Automate the routine: Add/Remove users, update packages, update config
- (Potentially) InfoSec can check themselves that the environment is patched for CVE-2014-0160 at 4pm on a Friday...

Ansible Examples

```
- hosts: admin
  order: sorted
  gather_facts: no
  roles:
    - common
```

An ansible “playbook” - it specifies the hosts and what “roles” (groups of tasks) to run. Tasks, files, and other data (variables) can easily be organized into roles.

```
- name: CentOS Base
  ini_file:
    path: /etc/yum.repos.conf/CentOS-Base.repo
    section: base
    option: baseurl
    value: https://mirror.domain.com/centos/$releasever/os/$basearch/
  tags: centos_repos
```

A task which sets the baseurl in a Yum repo. This is extremely useful in environments that have heightened security and restrictive outbound firewall rules.

Note that the ini_file function is generic. It works on almost any text ini file divided into sections, with option = value elements inside.

```
- name: sshd config
  copy:
    src: files/etc/ssh/sshd_config
    dest: /etc/ssh/sshd_config
    owner: root
    group: root
    mode: 0600
  notify: restart_sshd
```

Files can easily be distributed from a central source. Note that if this file is changed we execute a “notify” handler to restart sshd.

Ansible Examples con’t

```
- name: Add group to sudoers
  lineinfile:
    dest: /etc/sudoers
    line: '%ansible       ALL=(ALL)       ALL'
    insertafter: '^%wheel'
  when: inventory_hostname in groups[‘sudo’]
```

Ansible can also arbitrarily edit files.

Referencing a “when” clause - this only runs when the condition is met. The condition could be an inventory group as in this case, a playbook variable, a “fact” about the server, or the results of a command run earlier in the playbook.

```
- name: Set admin users authorized_keys
  tags: admin_ssh
  authorized_key:
    state: present
    user: '{{ item.path }}'
    key: "{{ lookup('file', '{{ item.src }}') }}"
  with_filetree: admin-ssh/
```

Name a public ssh key after a user and put it in the admin-ssh on the location ansible is run from and the file is copied to the user’s authorized_keys file in user’s ssh directory. Note the use of the user variable to make this elegant.

What about secrets? Two Suggestions.

ANSIBLE

- Easy to use
- Free (part of Ansible)
- Designed for simple things (passwords, keys, licenses)
- There is still a password that is shared amongst multiple users
- Designed to be used in Ansible

HashiCorp Vault

- Free (probably for all your needs)
- Separate install
- Powerfull (“Enterprise” = $$$)
- Bit of a learning curve
- Designed for multiple users
- Works with anything that can use an API (but designed for Terraform)

ASU

How does this relate to HARC/CRSE2?

- I (the CRSE) will supervise two of Dr. Oswald's Grad Students who will be learning Public Cloud, Terraform, and Ansible.

- The environment they create will directly enable Dr. Oswald to use the Public Cloud (specifically Azure to start) to perform simulations and analysis on an "as needed, as available" basis, leveraging:
  - Elastic resources (spin up/down as needed)
  - Spot instances (unused low cost resources)
  - Flexible payment (pay as needed)

- The Terraform and Ansible scripting will also support general ASU Research Computing users in the future, for "burst to cloud", rapid prototyping, and custom research-driven systems.

- BONUS: The Ansible scripting will also support Dr. Oswald's own local resources, enhancing his efficiency and security in the future.

Special Thanks to:

- Sean Dudley (ASU Research CIO) for the dream job
- IU and HARC Staff
- Doug Jennewein (ASU Research Computing Director) for the guidance in general and specifically the Zork-like (“A maze of twisty corridors, all alike”) maze of grants and proposals

Thank you for listening!