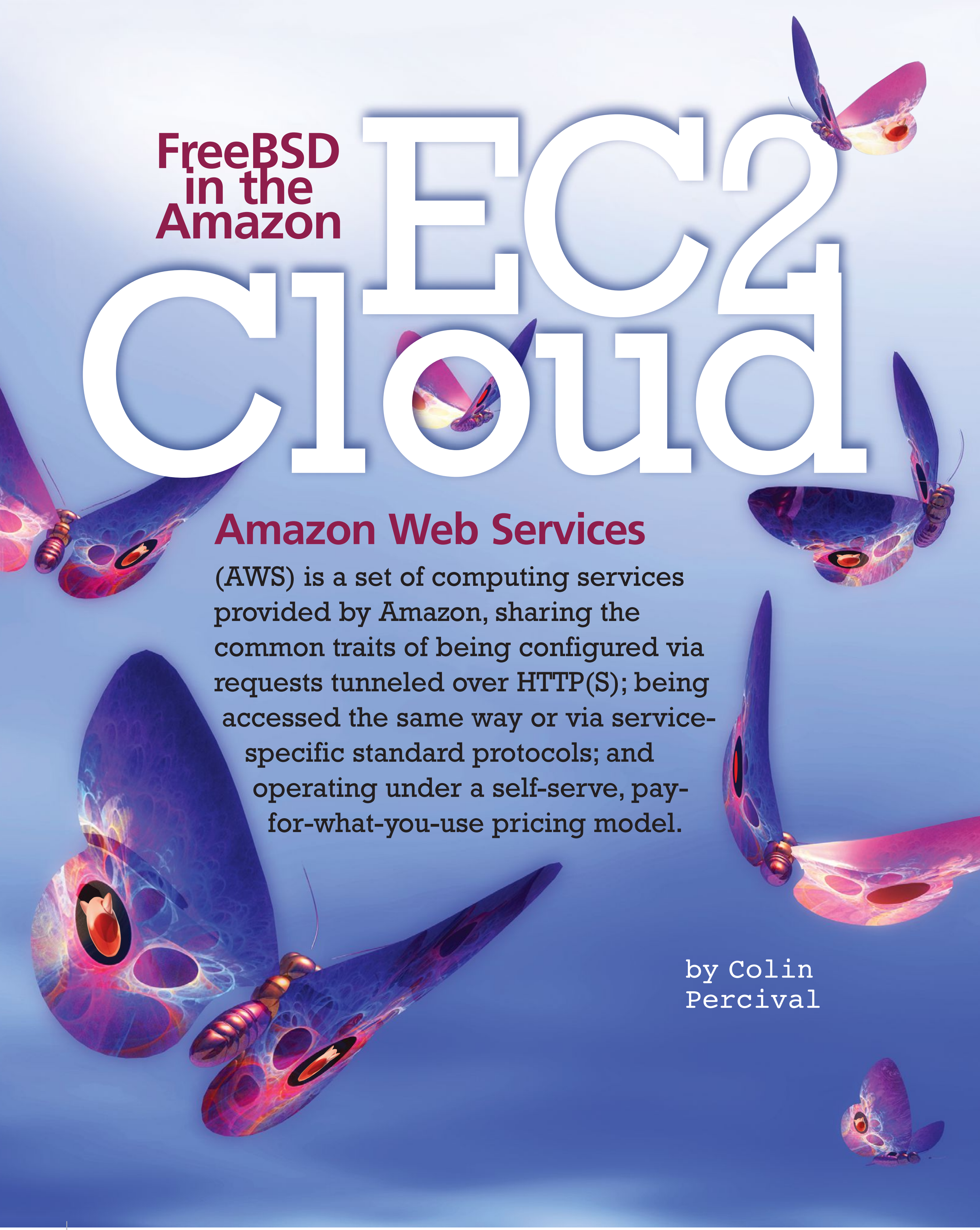
FreeBSD in the Amazon EC2 Cloud



Amazon Web Services

(AWS) is a set of computing services provided by Amazon, sharing the common traits of being configured via requests tunneled over HTTP(S); being accessed the same way or via service-specific standard protocols; and operating under a self-serve, pay-for-what-you-use pricing model.

by Colin Percival

Drawing on Amazon's experience building and operating computing infrastructure for their retail operations, AWS was the first entrant into the modern cloud computing industry and remains the largest player. Amazon provides dozens of services, but the most widely used ones can be placed into four major categories:

Core infrastructure services

- Elastic Compute Cloud (EC2)–Virtual machines
- Simple Storage Service (S3)–Durable data storage with ~50 ms response time
- Glacier–Low-cost durable data storage with ~4 hour response time
- DynamoDB–High-performance durable NoSQL data store
- Simple Queue Service (SQS)–Reliable message queueing [1]

Edge location services

- Route 53–Authoritative DNS
- CloudFront–HTTP(S) content distribution network

Managed software services

- ElastiCache–Memcached or Redis
- Relational Database Service (RDS)–MySQL, PostgreSQL, Oracle, or SQL Server
- Elastic Map Reduce (EMR)–Hadoop

AWS usability services

- Identity and Access Management–Create subaccounts with specified privileges
- CloudFormation–Scripted deployment of a set of AWS resources
- CloudWatch–Gathering and monitoring of AWS and custom metrics
- CloudTrail–AWS API call logging

All of these services are accessed via API calls made using a pair of AWS "access keys." An "Access Key ID," which is sent as part of the requests to identify the account, and a "Secret Access Key," which is used to create a cryptographic signature that authenticates the API request [2]. Amazon provides libraries for several programming languages to facilitate use of these APIs in addition to a command line utility and a web interface that can be used to manage many services. This article is concerned only with the Elastic Compute Cloud service, but readers interested in other AWS services are encouraged to consult Amazon's excellent documentation [3].

[1] While message queuing may seem like esoteric functionality, having reliable queues can be a great boon to designers of distributed systems where nodes might die and leave work half-done if it is not retained in a reliable queue.

[2] In addition to the fully privileged "root" access keys for an account, the AWS Identity and Access Management service can be used to create restricted "users" with access keys able to issue only a specified set of requests.

[3] http://aws.amazon.com/documentation/

[4] In keeping with Amazon's low-overhead, self-serve model, credit cards are the only payment method accepted except for US government users and companies spending over $5,000/month.

[5] For production use, you would probably want to use the Identity and Access Management to create a "User" which has a restricted set of privileges, and then create a key pair for that subaccount. For simplicity, we'll use the "root" account keys in this article.

Getting Started with AWS

Before you can use EC2 (or any other AWS services) you will need to create an AWS account. Point a web browser at http://aws.amazon.com/ and click on the "Sign Up" button. You can then log in with an existing amazon.com account or createa new one.

Once you've created or logged into an amazon.com account, you will be asked to provide your name and contact details, accept the AWS Customer Agreement, provide a credit card for billing [4], and complete an "identity verification" process consisting of receiving a phone call and entering a four digit PIN to activate your account for Amazon Web Services.

Now that you have an AWS account, you will need keys for accessing it [5]. Point a web browser at https://console.aws.amazon.com/iam/home?#security_credential and select "Access Keys" and "Create New Access Key." Make a note of the Access Key ID and the Secret Access Key and create a file named `~/.aws/config` containing the following lines (you will need to create the `~/.aws/` directory first):

```
[default]
aws_access_key_id = <Your Access Key ID goes here>
aws_secret_access_key = <Your Secret Access Key goes here>
region = us-east-1
```

Finally, you need software for accessing the AWS services. In this article we're going to use the AWS Command Line Interface (http://aws.amazon.com/cli/). To install this on your FreeBSD system run:

`# make -C /usr/ports/devel/awscli install clean` or

`# pkg install awscli` as root.

[6] For customers who can commit to long-term use, Amazon also offers "reserved instances," which are akin to buying hardware and then paying a heavily discounted rate for power and networking. These can offer substantial savings over the standard "on-demand" hourly rates for users with consistent usage patterns.

[7] For more details about AWS regions, see http://aws.amazon.com/about-aws/globalinfrastructure/regional-product- services/ .

[8] When EC2 launched, it only supported Xen "PV" mode. Amazon added limited support for HVM in order to support Windows, but did not retrofit their existing systems to support it.

[9] Users who will be launching a large number of identical instances (e.g., spinning up web and application servers based on fluctuating traffic levels) often create customized AMIs with code and configuration preinstalled. In such cases there will not necessarily be any need to log in via SSH, of course.

Amazon Elastic Compute Cloud (EC2)

The *Amazon Elastic Compute Cloud* (EC2) is, at its core, rent-by-the-hour virtual machines running under the Xen hypervisor. When EC2 was first launched in 2006, the idea of renting "virtual servers" was already well established, as many companies offered such services using either FreeBSD jails or virtualization systems such as Xen, but such offerings typically involved paying for a month or more at a time, and the role of virtual servers was limited to being a cheaper alternative to dedicated servers. By introducing hourly billing for virtual machines and providing an API for quickly and automatically provisioning new virtual machines, EC2 made available an entirely different benefit of virtualization–flexibility. Rather than renting servers for months at a time [6], EC2 is designed to allow users to scale up and down rapidly in response to load–hence the "Elastic" in "Elastic Compute Cloud."

Launching an EC2 *instance*–or indeed any sort of virtual machine–requires specifying three important parameters: Where to launch the virtual machine, what type of virtual machine to launch, and what should run on the virtual machine. Amazon Web Services, including EC2, is split into 8 Regions: North Virginia, Oregon, North California, Ireland, Singapore, Tokyo, Sydney, and São Paulo [7]. With the exception of some "support" services like billing and authentication systems, each region functions independently of the others. Consequently, in addition to the obvious benefit of reducing latency by placing servers close to systems with which they will be communicating, EC2 regions can also provide benefits for redundancy and regulatory compliance (e.g., ensuring that EU data never leaves the EU). Regions have names like "us-east-1".

Within each AWS Region, there are two or more EC2 *Availability Zones*. These exist for the purpose of making redundant systems possible: Amazon does its best to ensure that any failures (power, networking, etc.) can only affect a single availability zone. Availability zones have names like "us-east-1a".

When EC2 first launched, it only offered a single type of virtual machine, but over the years their offerings have expanded to 38 types, with names like ***m3.medium***, ***c3.xlarge***, or ***i2.8xlarge*** (in general, <family>.<size>). While there is not space in this article to provide details about all of them, there is an important technical detail concerning the virtualization technology—FreeBSD

Name	# CPUs	RAM	Solid state disks	Price (us-east-1 region)
t2.micro	10%	1.0 GB	none	$0.013/hour
t2.small	20%	2.0 GB	none	$0.026/hour
t2.medium	2 x 20%	4.0 GB	none	$0.052/hour
m3.medium	1	3.75 GB	4 GB	$0.070/hour
m3.large	2	7.5 GB	32 GB	$0.140/hour
m3.xlarge	4	15.0 GB	80 GB	$0.280/hour
m3.2xlarge	8	30.0 GB	160 GB	$0.560/hour
c3.large	2	3.75 GB	32 GB	$0.105/hour
c3.xlarge	4	7.5 GB	80 GB	$0.210/hour
c3.2xlarge	8	15.0 GB	160 GB	$0.420/hour
c3.4xlarge	16	30.0 GB	320 GB	$0.840/hour
c3.8xlarge	32	60.0 GB	640 GB	$1.680/hour
r3.large	2	15.0 GB	32 GB	$0.175/hour
r3.xlarge	4	30.5 GB	80 GB	$0.350/hour
r3.2xlarge	8	61.0 GB	160 GB	$0.700/hour
r3.4xlarge	16	122.0 GB	320 GB	$1.400/hour
r3.8xlarge	32	244.0 GB	640 GB	$2.800/hour
i2.xlarge	4	30.5 GB	800 GB	$0.853/hour
i2.2xlarge	8	61.0 GB	1600 GB	$1.705/hour
i2.4xlarge	16	122.0 GB	3200 GB	$3.410/hour
i2.8xlarge	32	244.0 GB	6400 GB	$6.820/hour

requires support for Xen "HVM" mode, which the older EC2 instance types only provide in combination with a Windows license [8]. Most people will want to use one of the 21 instance types belonging to the "current generation" M3 ("general purpose"), C3 ("high CPU"), R3 ("high RAM"), I2 ("high I/O"), or T2 ("low cost") families–which provide better price/performance levels than the older instances anyway. See chart on page 6.

Note that with the exception of the amount of SSD storage on the "medium" and "large" sizes, each step size within a family doubles the number of CPUs, amount of RAM, amount of SSD storage, and hourly price. Conversely, comparing the xlarge sizes of different families, we see that the "high CPU" family is identical to the "standard" family except with half the RAM and a 25% discount on price. The "high RAM" moves in the opposite direction, with double the RAM of "standard" instances and a price 25% higher; and "high I/O" instances are "high RAM" instances with a tenfold increase in SSD storage–and a further 145% increase in price.

The T2 family is different from other EC2 instance types, in that it has "burstable" CPUs rather than dedicated CPUs: Over the long run they can only use a CPU for 10% or 20% of the time, but if a CPU is underused a "balance" will accumulate for up to a day. These instances can be very useful for systems which are mostly idle but occasionally need to rebuild ports or other similarly CPU-taxing work. (See http://aws.amazon.com/blogs/aws/low-cost-burstable-ec2-instances/ for details).

Once you have decided where to launch an instance and what type of instance to launch, you'll need to tell EC2 what software the instance should run. In EC2 jargon, this is an ***Amazon Machine Image*** (AMI) and it consists of a disk image (containing FreeBSD, for example) and some metadata which tells EC2 how to launch the image. AMIs can be created directly from disk snapshots in EC2 or by "re-bundling" a running EC2 instance–but new users will find it easiest to start by using one of the thousands of existing, freely available AMIs.

Because EC2 operates in a "cloud" environment, there are two more considerations required to securely connect to and use a new virtual machine–firewall rules and login credentials [9]. All EC2 instances are launched into "security groups," which is a fancy way of saying that a firewall rule set can be created in advance and applied to a new instance. By default, new security groups allow all outgoing traffic, but block all incoming traffic except responses to outgoing connections [10]. Finally, EC2 provides a mechanism for specifying an SSH public key for logging in to an instance. This key is provided (along with a variety of other metadata) over a special

[10] The firewall is stateful, but not perfectly so. In particular, in some cases it will block incoming ICMP fragmentation required packets, with the effect of breaking Path MTU Discovery. You may wish to add a firewall rule to allow incoming ICMP type 3 code 4 packets.

[11] The opposite applies if you're launching a large number of identical instances between which you will have traffic load-balanced. In that case you should put them all into the same Security Group so that you can adjust the firewalls for all of them at once.

Launching a FBSD EC2 Instance

In this example we will be launching an m3.medium instance costing $0.07/hour. If you forget to shut down the instance, it will continue running and your credit card will be billed roughly $50/month—so don't forget!

If you followed the instructions under "Getting started with AWS," you should now have the awscli port installed and a configuration file in `~/.aws/config`. Before we launch an EC2 instance, we need to create an SSH key we will use to log in to it. Run the following commands:

```
$ ssh-keygen -f ~/.ssh/ec2login
$ aws ec2 import-key-pair --key-name mykey --public-key-material file://`realpath
~/.ssh/ec2login.pub`
```

This will create an SSH key (in the file "ec2login") and send the public part up to EC2, associating it with the name "mykey." (The `file://` mess is because the `awscli` code expects URIs for everything rather than simple paths to files. I suggested changing this, but the author felt that using URIs provided better consistency.)

We're also going to create an EC2 security group. For historical reasons there is a pre-existing "default" group, but it's a good idea to create a new security group when you're launching an instance for a new purpose. That way you can adjust its firewall rules later without affecting other instances [11]. Run the following commands to create a security group and allow SSH connections from your IP address to its instances:

```
$ aws ec2 create-security-group --group-name mygroup --description "my security group"
$ aws ec2 authorize-security-group-ingress --group-name mygroup --protocol tcp --port
22 --cidr <your IP address>/32
```

Now to launch an instance: Normally at this point you would want to look up which AMI you want to launch–there is a list at http://www.daemonology.net/freebsd-on-ec2/ with AMI IDs for different versions of FreeBSD in each AWS region, but for this example we'll use FreeBSD 10.0-RELEASE in the us-east-1 region (which is the default value we selected when we created the ~/.aws/config file)—that AMI is ami-69dae900.

Launching Instance continues next page.

Launching continued

Run the following command to launch FreeBSD 10.0-RELEASE into an m3.medium virtual machine which we will use the mykey SSH key to connect into, and output the instance ID. [12]

```
$ aws ec2 run-instances --image-id ami-69dae900 --instance-type m3.medium --key-
name mykey —security-groups mygroup --output text --query 'Instances[*].InstanceId'
```

Make a note of the instance ID as you'll need it for the rest of the commands here. If you forget it, you can list all your running instance (and get lots of information about them) by running

```
$ aws ec2 describe-instances.
```

Now go make a cup of coffee. We need to kill about five minutes while FreeBSD boots and EC2 collects its console output for us [13].

You're back and caffeinated? Good, now you can run

```
$ aws ec2 get-console-output --output text --instance-id <Your Instance ID goes
here> | more
```

and you will see FreeBSD's usual boot-time kernel output. Scroll down, though, and you'll see a few things you don't get in a standard FreeBSD install. First, `freebsd-update` runs, downloading and installing critical updates to FreeBSD—important, since with a virtual machine in the "cloud" we can't exactly follow the usual security advice to log in and apply security updates before exposing a new server to the internet! After that, you'll see the `pkg` bootstrap running and the `awscli` package being downloaded and installed. We will see later how to change the set of packages installed on first boot. Finally, we see FreeBSD rebooting. This is done if `freebsd-update` installed any updates or any `rc.d` scripts were added by installed packages, so that we will have a system running up-to-date code with all enabled services.

It's interesting to see what FreeBSD is doing, but there's really only one vital reason to read the console output: To see the fingerprints of the SSH host keys. These host keys are generated when the system first boots, and–again, because this is in the cloud–we need to use this out-of-band mechanism to ensure that when we connect to the system we're SSHing into the right box. In addition to being printed when the keys are generated, the fingerprints are printed again with a prefix of `ec2:` in order to make them easily extracted by automatic tools [14].

Speaking of SSH, it's time to use it. Run:

```
$ aws ec2 describe-instances --output text --query
'Reservations[*].Instances[*].PublicIpAddress' --instance-id <Your Instance ID
goes here>
```

to get the public IP address of your instance. Then run:

```
$ ssh -i ~/.ssh/ec2login ec2-user@<Instance IP address goes here>
```

SSH should prompt you to confirm the SSH host fingerprint. You can compare this against the value you saw a moment ago in the console output.

You should now be logged in to your EC2 instance as the default unprivileged "ec2-user" user. To become root, simply run 'su' (there is no root password set in the standard image). This system will now behave like any other FreeBSD 10.0-RELEASE system. When you've finished looking around, log out.

Now unless you want to pad Amazon's profits, you should destroy your EC2 instance. Run the following commands:

```
$ aws ec2 modify-instance-attribute --block-device-mappings '[ { “DeviceName”:
“/dev/sda1”, “Ebs”: { “DeleteOnTermination”: true } } ]' --instance-id <Your
Instance ID goes here>
$ aws ec2 terminate-instances --instance-ids <Your Instance ID goes here>
```

The first command is necessary because the default behavior is to retain a copy of the boot disk of a terminated instance (costing $0.50/month for the standard 10-GB instance boot disk).

If you want clean up your AWS account completely, you can also delete the security group you created using the '`aws ec2 delete-security-group`' command and delete the SSH key using the '`aws ec2 delete-key-pair`' command, but these are not essential–Amazon doesn't charge anything for security groups or SSH keys.

management interface. In FreeBSD, it is code in the `sysutils/ec2-scripts` port which reads this data and arranges for user logins.

Elastic Block Store, Elastic IP Addresses, Elastic Load Balancer (oh my!)

As a platform for creating virtual machines, EC2 is useful enough, but there are several other services that add important functionality to EC2. First, ***Elastic Block Store*** (EBS), which makes it possible to create and attach virtual disks to EC2 instances, second, ***Elastic IP Addresses*** (EIP), which make it possible to reserve an IP address and assign it to an instance or move it between instances, and third, ***Elastic Load Balancers*** (ELB), which provide an easy mechanism for load balancing traffic between a pool of instances.

Elastic Block Store would probably have been better named EC2 Storage Area Network. With an API call you can create a virtual disk of between 1 GB and 1 TB, known as an ***EBS Volume***, within a specified EC2 availability zone. Another API call can be used to attach a volume to an EC2 instance. Because these volumes are accessed over Amazon's network rather than being physically connected to the boxes hosting EC2 instances, it is easy to detach a volume from one EC2 instance and attach it to another. This can help for migrating data to a new instance, or–as often happens to FreeBSD developers–if a system is accidentally rendered unbootable [15].

Elastic Block Store comes in three flavors: "Magnetic" (formerly known as "Standard"), "Provisioned IOPS," and "General Purpose." Magnetic volumes are cheap, but as the name implies they have typical "spinning iron oxide" performance levels–typically 50 to 100 I/O operations per second on 128 kB blocks. Provisioned IOPS volumes, in contrast, allow you to specify a target I/O rate of between 1 and 4,000 I/O operations per second on 16 kB blocks, but are more expensive, especially for large reserved I/O rates. General Purpose volumes fall into a middle ground: They can burst up to 3,000 I/O operations per second, but provide a performance baseline of 3 I/O operations per second per GB of allocated space. It is important to remember however that because EBS is accessed over the network, all flavors are slower than the SSD storage which is attached directly to EC2 instances–there is an unavoidable trade-off between optimizing for performance and optimizing for usability.

Elastic IP Addresses should have been named Reserved IP Addresses, since that's exactly what they are. Like EBS volumes, Elastic IP Addresses can be allocated, attached to EC2 instances, and moved between EC2 instances. These are necessary for directly public-facing servers, since without those EC2 instances are created with randomly selected IP addresses and there is no way for a new instance to assume the address of an earlier instance which has been shut down.

While Elastic IP Addresses make it possible to put an address into DNS and know that it can be pointed at a new EC2 instance if required, this does not satisfy the needs of companies with large numbers of public-facing servers. Enter Elastic Load Balancers: An Elastic Load Balancer is configured with a pool of EC2 instances–as usual, with API calls to add or remove EC2 instances–and it forwards incoming connections to instances from the pool.

There are many more EC2 features: Virtual Private Cloud, which allows you to configure virtual networks including routing tables, IP subnets, and VPN endpoints; Elastic Network Interfaces, which make it possible to attach multiple addresses to an EC2 instance; CloudWatch, which provides monitoring of EC2 instances; Autoscaling, which makes it possible to automatically launch or shut down instances in response to load. There is not enough space here to write in detail about everything, so we can merely refer the reader to Amazon's excellent website and documentation.

Instance Autoconfiguation Using configinit

While many people may be satisfied with launching a clean FreeBSD system and SSHing in to perform necessary system configuration, especially for one-off servers, there are advantages both in speed and repeatability to having some or all of the configuration process scripted and performed automatically when an EC2 instance first boots. Many Linux systems, including Ubuntu and RedHat, make use of the CloudInit system, which allows a user-data file provided when an EC2 instance is launched to specify a set of commands to be run.

CloudInit is not very well suited for use in FreeBSD for two reasons. First, it uses Python, which is not part of the FreeBSD base system [16], and second, because CloudInit is built around the concept of configuring a system by running commands. In contrast to Linux systems, FreeBSD is far more "configuration file

[12] The default behavior of the AWS CLI is to output JSON containing a very large number of parameters about the request completed. This is useful for programs which speak JSON, but for shell scripts I always use the `--output text` option along with `--query` to specify the particular details I want.

[13] It seems to take a minimum of 3 minutes before console output will be available via the EC2 API, and if you try to read the output too soon, EC2 seems to cache the "no output" response and you need to wait even longer.

[14] One such tool is the author's ec2-known-host script, which populates .ssh/known_hosts with EC2 host fingerprints, available at http://www.daemonology.net/blog/2012-01-16-automatically-populating-ssh-known-hosts.html.

[15] The process in this case is non-trivial, but it is possible to detach the boot disk from an instance, attach it temporarily to another instance so that something can be fixed, and then move it back and reboot the formerly non-booting instance.

[16] It isn't feasible to include Python in standard FreeBSD EC2 images either, since there are several different versions of Python in the FreeBSD ports tree, and the one installed to run CloudInit would inevitably not be the one people would want to use for other purposes.

oriented," and so I decided to write my own system, which I called "configinit." Unlike CloudInit, the configinit system takes the form of a very simple shell script, and is included in FreeBSD EC2 images starting from 10.0-RELEASE. Similar to CloudInit, however, configinit runs early in the boot process when an EC2 instance first launches, downloads the EC2 user data (which is exposed via the same management interface used for ssh public keys), and then processes that file.

There are four types of files that configinit can handle:

• If a file starts with the characters `>/` then the first line minus the leading `>` character will be interpreted as a path, and the rest of the file (everything except the first line) will be written to that location (replacing any existing file).

• If a file starts with the characters `>>/` then the first line minus the leading `>>` characters will be interpreted as a path, and the rest of the file will be appended to that location (creating a new file if no file yet exists at that location).

• If a file starts with the characters `#!` then the file will be executed (this is most likely to be used with shell scripts).

Otherwise, configinit attempts to extract the file as an archive using bsdtar (which automatically detects a wide range of archive and compression formats), and then recurses onto each file in lexicographical order.

The most straightforward use of this functionality is to add content to FreeBSD's "master configuration file," `/etc/rc.conf`. The configinit code instructs the `rc` system to reload that file in order to ensure that settings changed will be reflected later in the boot process. In addition to the standard FreeBSD configuration options (e.g., `sshd_enable="YES"`, which appears in the `rc.conf` file in EC2 images), there are a few useful options for the packages which are preinstalled on EC2 images:

```
firstboot_pkgs_list="list of packages"
```

will cause those packages to be downloaded

Setting Up Drupal at Instance Launch Using configinit

Now we're going to use configinit to launch a system with Drupal installed. The same caveat applies here as before–EC2 instances cost money, so make sure you don't forget to clean up after you've finished exploring

First, let's create a new SSH key pair for the instance we're going to launch. Although we're not going to SSH into the instance right now, if you were planning on using this you would need to be able to SSH in later to perform upgrades (of both FreeBSD and the packages required by Drupal) and to perform backups. Run the following commands:

```
$ ssh-keygen -f ~/.ssh/drupal
$ aws ec2 import-key-pair --key-name drupal --public-key-material file://`realpath ~/.ssh/drupal.pub`
```

We also need to create a new EC2 security group (aka. firewall rule set). In this case we want to open up port 80 to the entire world, while there's no need for SSH access. If and when you need to SSH into this instance later, you can use `authorize-security-group-ingress` to add a rule allowing incoming traffic on port tcp/22. Run the following commands:

```
$ aws ec2 create-security-group --group-name drupal --description "drupal demo security group"
$ aws ec2 authorize-security-group-ingress --group-name drupal --protocol tcp --port 80 --cidr
0.0.0.0/0
```

Now for the configinit data, rather than retyping the data run the following command to fetch a Drupal configinit file:

```
$ fetch http://freebsd-ec2-dist.s3.amazonaws.com/drupal-conf.tar
```

If you look inside that archive you'll see three files named "`rcconf`," "`apache`," and "`drupalinit`." The names aren't important; when configinit extracts the archive and processes the files individually, it will handle them in lexicographical order, but in this case the order doesn't matter. The file "`rcconf`" contains the following lines:

```
>>/etc/rc.conf
ec2_fetchkey_user="drupal-user"
firstboot_pkgs_list="apache22  drupal7  mod_php5  mysql55-server"
apache22_enable="YES"
mysql_enable="YES"
```

The first line means "append the rest of this file to `/etc/rc.conf`," while the remaining lines are rc.conf settings that tell the EC2 boot scripts to set up SSH logins for a user named "drupal-user." Download and install the packages `apache22`, `drupal7`, `mod_php5`, and `mysql55-server`, start `Apache 2.2` and start MySQL.

The file "`apache`" writes to `/usr/local/etc/apache22/Includes/local.conf` the necessary configuration to tell Apache to serve files out of `/usr/local/www/drupal7` (which is where the Drupal port installs its data), and to enable PHP (which Drupal uses).

The file "`drupalinit`" is a shell script which performs steps needed to allow Drupal to work. It sets ownerships to and installed when the EC2 instance first boots, `firstboot_freebsd_enable="NO"` willl disable the launch-time bootstrapping of the pkg system (useful if an instance is being launched into a networking environment where it cannot access the pkg mirrors), `firstboot_freebsd_update_enable="NO"` will disable the launch-time downloading and installation of critical errata and security updates, `ec2_fetchkey_user="username"` will change the name of the user created and configured for SSH access via the public key provided to EC2.

For example, providing the following user-data file

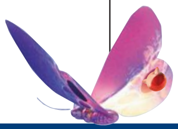
```
>>/etc/rc.conf
firstboot_pkgs_list="apache22 python33"
ec2_fetchkey_user="webmaster"
apache22_enable="YES"
```

will result in Apache 2.2 and Python 3.3 being preinstalled, a user named "webmaster" being created for SSH logins and Apache 2.2 being launched automatically during the boot process.

More sophisticated uses of configinit will generally need to create or edit multiple files. In those cases it will be necessary to create an archive file (e.g., a tarball) containing one file for each required configuration change (see "Setting up Drupal at Instance Launch" for an example).

Future Directions for FreeBSD/EC2

FreeBSD has been available on EC2 in some form since December 2010, and it has been stable enough for production use since mid-2011. A lot of progress has been made since then, and especially in the past year. In October 2012, Amazon announced the M3 instance family, followed in November by the C3 instance family, in December 2013 by the I2 instance family, in April 2014 by the R3 instance family, and in July 2014 the T2 instance family–and between them, this "current generation" of EC2



the "`www`" user so that the Drupal code (running via Apache and mod_php5) can function and creates a mysql database and user for Drupal to use. There's a catch, however. Because configinit runs early in the boot process–by necessity, early enough to add `rc.conf` settings which specify which packages to install–there is no mysql daemon running and the Drupalfiles which need to have ownership changed do not exist yet. To escape this limitation, the `drupalinit` script creates a new rc.d script, which will run after packages are installed and mysql is running–a script which then deletes itself after initializing Drupal data to prevent itself from being run again the next time the system boots.

Now that we've seen how the Drupal configinit file works, let's see it in action. Run the following command:

```
$ aws ec2 run-instances --image-id ami-69dae900 --instance-type m3.medium --key-name
drupal --security-groups drupal --user-data file://drupal-conf.tar --output text --
query 'Instances[*].InstanceId'
```

Note that the only change between this and launching a standard FreeBSD image is the added `--user-data file://drupal-conf.tar` option. As before, the clumsy `file://` syntax is required due to the URI-centric design of the AWS CLI.

As before, we'll need to wait for FreeBSD to boot and EC2 to collect its console output. After five minutes, run:

```
$ aws ec2 get-console-output --output text --instance-id <Your Instance ID goes here>
| more
```

and you'll see FreeBSD booting, updating itself, downloading and installing the packages we requested, rebooting, launching the Apache and MySQL daemons, and finally you'll see a line

```
Drupal password: <12 characters>
```

which is the (randomly generated) password for the MySQL "drupal" user. Once we have the password, we can proceed with configuring Drupal. Run:

```
$ aws ec2 describe-instances --output text --query 'Reservations[*].Instances[*].
PublicIpAddress' --instance-id <Your Instance ID goes here>
```

to get the public IP address of your instance, and then paste it into your web browser of choice. When prompted for database parameters, tell Drupal to use the database named "drupal," database username "drupal," and the password you obtained from the console output. Congratulations, you now have Drupal running and ready for you to add content to your new website!

As always with EC2 instances, when you've finished you'll want to clean up to avoid paying unnecessary costs. Run the following commands:

```
$ aws ec2 modify-instance-attribute --block-device-mappings '[ { "DeviceName":
"/dev/sda1", "Ebs": { "DeleteOnTermination": true } } ]' --instance-id <Your Instance
ID goes here>
$ aws ec2 terminate-instances --instance-ids <Your Instance ID goes here>
```

instance types make it possible to run FreeBSD across a wide range of hardware profiles without any of the evil hacks needed to run FreeBSD on earlier instance types.

Coming from the other direction, starting with FreeBSD 10.0-RELEASE in January 2014, FreeBSD gained Xen support in the GENERIC kernel configuration, allowing official release binaries to run in EC2 and making it possible to use the FreeBSD Update system for binary updates. The addition of configinit made it possible to configure FreeBSD virtual machines via EC2 user data and scripts are now available that allow anyone to build FreeBSD AMIs [17].

[17] http://www.daemonology.net/blog/2014-02-16-FreeBSD-EC2-build.html

[18] FreeBSD/EC2 already autoconfigures swap space on EC2 ephemeral disks, but most instance types have far more ephemeral disk space than swap space alone can use.

So what's next? This will depend largely on what FreeBSD–and EC2–users ask for, but some possible highlights include:

- Integrating the EC2 AMI building process into the FreeBSD release process and having AMIs built by the FreeBSD release engineering team,
- Providing EC2 AMIs with ZFS or MFS root filesystems,
- Automatically using EC2 ephemeral disks to cache or mirror data on EC2 Elastic Block Store volumes to provide greater performance and/or reliability [18],
- Using Identity and Access Management "roles" to allow an EC2 instance to automatically create and add more Elastic Block Store volumes to itself, allowing for truly "elastic" file-systems, and
- Providing more configinit samples for ready-to-use server setups.

But more than anything else what FreeBSD/EC2 needs right now is more users, more testing, and more feedback. What works? Is there anything that doesn't work? What features would you like to see added? Most of the development is done. Now it's time for the baton to be passed to the wider user community–go forth and use! •

Colin Percival has been a FreeBSD developer since 2004 and was the project's Security Officer from 2005 to 2012. He started struggling to bring FreeBSD to the EC2 environment in 2006, and now uses it extensively in the Tarsnap online backup service, which he founded and continues to run.

RootBSD

Premier VPS Hosting

RootBSD has multiple datacenter locations, and offers friendly, knowledgeable support staff. Starting at just $20/mo you are granted access to the latest FreeBSD, full Root Access, and Private Cloud options.

www.rootbsd.net