Best Practices

Language-agnostic Best Practices

Sources	Practice	Summary /Extracts
S22 S09 S08 S14	Codify Everything (4)	All infrastructure specifications should be explicitly coded in the configuration files, which act as the single truth of source of the infrastructure specifications. Seamless deployment of the infrastructure sole based on the configuration files. no manual adjustments should be needed (ideally) Codify all the infrastructure things!
S14 S29 S22 S4 S6 S7 S13 S61 S19 S23 S26 S27 S28 S32 S33 S8	Versioning Everything (16)	A tool to manage the database changes with appropriate versioning practices These configuration files will be version-controlled. Because all configuration details are written in code, any changes to the codebase can be managed, tracked, and reconciled. provide an audit trail for code changes, provide the ability to collaborate, peer-review, and test IaC code before it goes live. Code branching and merging best practices - increase developer collaboration and ensure that updates to your IaC code are properly managed. track all the changes in your infrastructure environment. rolling back configuration changes to a previous working configuration in case of a problem. Any infrastructure change should be triggered by a change in the Git repo Configuration change tracking Pull Request (PR) reviews Every single configuration, documentation, test case and script needs to be checked-in This disqualifies binary files such as large VM images. easily define and clone configurations and make changes seamlessly and consistently. Documentation is always current because it is in the code and stored in a central repository
S22 S18 S8 S28	Modularity (5) Configuration Composition (1) Configuration Discovery (1)	break down your infrastructure into separate modules or stacks then combine them in an automated fashion. greater control over who has access to which parts of your infrastructure code (different teams and personals for different codes) limits the amount of changes that can be made to the configuration Align with microservices approach (infra config for microservice) The theory here is good, by isolating the key logic into modules you can reduce complexity and duplication. IaC should present itself as flexible blocks that can be assembled instantly if and when the requirements arise group infrastructure configuration into reusable components.
S22 S7 S20 S28 S9	Code as Documentation (7)	your source code is your documentation. need to include many, if any, additional instructions for users. Your IaC code will serve as documentation in itself. Avoid *infrastructure-instructions* manual inconsistency. However, setup instructions and diagrammatic representations of your architecture are useful for bringing other employees up to speed on the system and sharing knowledge Restrict your documentation! •Avoid written documentation, since the code itself will document the state of the machine. This is particularly powerful because it means, for the first time, that infrastructure documentation is always up to date.
S32	Decentralized Repository (1)	Single repository for the entire stack. Ability to couple application releases with infrastructure or configuration changes, therefore adding: Ability to provision ad-hoc infrastructure for temporary use during deployment pipeline. Ability to push application releases along with required infrastructure configuration changes (such as JVM example above).
28 S7 S9 S22 S6 S61 S27 S17	Testing (12)	Use static code analysis Apply testing to infrastructure in the form of unit testing, functional testing, and integration testing. An array of test types – unit, regression, integration and many more – should be performed. Automated tests can be set up to run each time a change is made to your configuration code. Security of your infrastructure should also be continuously monitored and tested Error replication - errors in code - automated creation - multiple machines with errors. Audit trail of changes are needed Enforcing policies with unittests For example, With infrastructure as code we can make sure a password policy is actually enforced, e.g., Write a unittest that verifies the configured password length. Make it mandatory to run on all servers create tests which examine your IaC setup to ensure it is secure integration testing gives the ability to gain greater confidence that a change being made to an environment won't

		break anything before the changes are applied to production or a system effectively operating as production. Redeployment overimtine uncover issues. As we identify issues, we can add testing for them into our suite of automated tests.
S28	Incremental Configuration (1)	Many packages will already be on the system in their default state. Instead of duplicating the default state in your code, you can only define an incremental change.
S28	Configuration template (1)	
S28	Configuration Data Source (1)	Keep configuration data (e.g., user names, server ips) in a suitable data storage (database) Useful when number of managed items exceeds certain amount
S28	Reproducible Image (1)	Antipattern: Golden Image Manually crafted base infrastructure server image that nobody dares or knows how to change. Pattern: Reproducable Images Operating system distributions (*.iso). Base provider images. Packer can create images for many virtualization software and cloud providers. Docker can build and package containers as images for distribution
S21 S31 S22 S10	Immutable Infrastructure (5)	In order to make any update or fix the software, new servers built from the base image with the modifications are deployed and the old servers are deprovisioned. Advantages: By eliminating the need for patching and in-place server upgrades, immutable infrastructure simplifies maintenance by eliminating corner cases and inconsistencies in the deployed server footprint. No more one-offs and configuration drift No need for administrative ports (ssh, rdp) to be kept open on servers restricts the impact of undocumented changes to your stack.
S19 S28	Environment Template (2)	Templates for various levels (staging and production, etc) of our infrastructure there can be multiple stacks (instances) of the same template Define a template from which you can create a fully working environment. It gives scaling. It gives isolation. It gives flexibility. Be as specific as possible about the infrastructure requirements, including network bandwidth and storage I/O operations per second, if possible. This is often overlooked
S28	Package Application for Deployment (1)	Package your application in the most appropriate format that is ready for the most hasslefree deployment. Publish it to an artifact repository (Maven, RubyGems, Yum, Apt...). Artifact repository serves as a layer of isolation between pipelines. Reduces the amount of code needed on later stages of configuration management.
S28	Infrastructure Query Language (1)	Language or API that allows to query your infrastructure state (real time or last available report). Collect the data generated from from IaC systems
S28	Secret Isolation (3)	Secrets should be injected on the very last stage of "deploying" your code. Secrets should not be in code
S28	Encrypted Secret (1)	all stored secrets must be encrypted Decryption password is shared through a different channel.
S28 S7 S60	Collaborate (3)	Enable collaboration around infrastructure configuration and provisioning, most notably between dev and ops. Do not keep your updates only to yourself. Share them back. Discourage a private fork of a community module code reviews are a must for those embracing the notion of treating their infrastructure as code.
S28	Metrics as Code (1)	Pattern: Metrics as Code Metrics that your application provides evolve with your application. New components, new endpoints, new KPIs... Keep monitoring configuration close to the code! Or make it autodiscoverable and visible! Configuring and collecting metrics Monitoring software has configuration files and/or an API that can be programmed. There a plenty of libraries that allow

Ansible Best Practices

Source	**Practice**	**Summary**
Project		
S46 S51 S57	Use the standard Ansible project structure (3)	
S46	Use semantic versioning (1)	
S46 S48	Use native YAML syntax and conventions (6)	

S55 S52 S54 S50		
S50 S57	Minimal Consistent quoting (2)	Decide for one quoting style and use it consistently: double quotes (dest: "/etc/some.conf"), single quotes (dest: '/etc/some.conf') plus decision if you quote things that don't need it (dest: /etc/some.conf). Keep in mind that dest: {{ var }} is not possible (must be quoted), and that mode: 0755 (chmod) will give an unexpected result (no octal number support), so recommended practice is of course Use quotes only if necessary • String starts with a YAML control character (–, {, }, [,], *, &, ?, |, >, !, %, `, #, @, :) `- file:` `path: "{{ my_path }}"` `mode: 0644` • String contains colon followed by space `- debug:` `msg: "Path: {{ my_path }}"` • String is a boolean value (e.g. yes, false, ...) which should be preserved `- copy:` `dest: "{{ my path }}"` `content: "yes"` • If unsure, use yamllint or test it `ansible all -i localhost, --connection local -m debug -a 'msg={{xxx}}' -e '{xxx: @asd}'`
S55 S53 S51	Vaults for Storing Secrets (3)	The more you use Ansible, more tasks become automated. You will reach a point, that things such as SSL con}guration, database passwords needs to be automated. This is where Ansible provides 1234 - name: install apache yum: name: httpd state: latest 1234 - name: install the latest version of apache yum: name: httpd state:latest the vault (http://docs.ansible.com/ansible/playbooks_vault.html) module so you could safely save these data and automate it without any risks. Vault lets you encrypt important data that can be safely saved in your version control system or transfer to another system. It is most likely that you will have a password or certificates in your repository. It is not a good practise to put them in a repository as plain text. You can use ansible-vault to encrypt sensitive data. You can refer to postgresql-password.yml in group variables to see the encrypted file and postgresql-password-plain.yml to see the plain text file, commented out. To decrypt the file, you need the vault password, which you can place in your root directory but it MUST NOT be committed to your git repos To keep sensisitve data in your playbooks and roles secret, use ansible-vault. There's extensive documentation from Ansible with good examples so I won't cover this topic here.
Playbook		
S46	Decompose top-level playbooks by their roles (1)	2019 Best Practices — Ansible Documentation https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html 6/11 The idea here is that we can choose to confgure our whole infrastructure by "running" site.yml or we could just choose to run a subset by running webservers.yml. This is analogous to the "– limit" parameter to ansible but a li,le more explicit:
S46	Separate infrastructure configuration from application deployment (1)	Deployment vs Confguration Organization The above setup models a typical confgura on topology. When doing mul- er deployments, there are going to be some addi onal playbooks that hop between ers to roll out an applica on. In this case, 'site.yml' may be augmented by playbooks like 'deploy_exampledotcom.yml' but the general concepts can s ll apply. Search this site 18/6/2019 Best Practices — Ansible Documentation https://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html 8/11 Consider "playbooks" as a sports metaphor – you don't have to just have one set of plays to use against your infrastructure all the me – you can have situa onal plays that you use at diferent mes and for diferent purposes. Ansible allows you to deploy and configure using the same tool, so you would likely reuse groups and just keep the OS confgura on in separate playbooks from the app deployment.
S47	Use tags only for speeding and debugging play execution (1)	The problem is that tagging every task in main.yml would be cumbersome, error prone, and clutter the code unnecessarily.
Role		
S46 S54	Use the standard role directory structure (2)	
S50	Parameterized roles (1)	Do not install multiple services from a single role (e.g. Nginx + php-fpm) ○ Instead, use separate them into well parameterized roles (one for Nginx, second for php-fpm)
S50	Wrapper roles (1)	• Useful for *wrapping roles* ○ Role which changes variables of an other role ○ Can also define additional tasks on top the other role
S46 S49	Use roles to group related tasks (4)	In Ansible, roles allow you to group related tasks and all their variables and dependencies into a single, self-contained, portable entity. Grouping your tasks into roles is one of the best ways to maximize the power of Ansible's modularity
S50 S51	Use role documentation templates (2)	create documentation for role - sections : name, description, examples, variables, dependencies, author, licence When documenting roles, it's best to use the template created by ansible-galaxy init. There you have to describe the role and its function, list and explain the variables used, the needed dependencies and provide examples. I always try to add some more documentation of the variables in the form of a table, providing the variable name, the default value and a explanation of the variable:

S50 S54 S57	Test Roles with an emulated environment (3)	Test roles with Vagrant You should be able to perform unit testing in each role in a CI model A convenient way to test roles is by using containers. This will allow you to test the role across multiple distributions Virtual machines are recommended to test roles when the role is performing low level actions like bootloader setting, kernel parameters, firewall settings
S49 S55 S54 S51	Use Ansible Galaxy to find and share Roles (5)	
Task		
S48 S50 S55 S52 S51 S57 S46	Name tasks (10)	It is possible to leave off the 'name' for a given task, though it is recommended to provide a description about why something is being done instead. This name is shown when the playbook is run Always name your plays and tasks. Adding a name with a human meaningful description better communicates the intent to users when running a play. Consider this example play and its standard Ansible output A Task name should indicate its purpose very clearly. If someone else runs our playbook, the name needs to indicate the purpose and the action of the task very clearly to the End Users.
S51	Use variables in task names (1)	Try to be expressive when writing task names. Include as much information as necessary. A good way to do this is to use variables in your task names. For example if you want to determine the host a task is currently running against, you can include a variable in your task name.
S51	Omit superfluous information in task names (1)	One thing you don't have to do, is to include the name of the role in the task-name.
S51	Specify module defaults in tasks (1)	Again there are two reasons for this. The first is of technical nature: When you don't explicitly declare the owner and group of the file, the owner will be the user that executed Ansible. That's something that is not always desirable and can be easily avoided by being explicit. The second reason is more of an organizational or „people-reason". When people use your playbook or role, they may not always know the defaults of the modules you use or what you want to achieve with the tasks. When being explicit in your tasks, there's less room for guessing and interpretations.
S54 S46	Always mention module state (3)	The 'state' parameter is optional to a lot of modules. Whether 'state=present' or 'state=absent', it's always best to leave that parameter in your playbooks to make it clear, especially as some modules support addi143onal states. Define the state parameter. In some modules this could be: present, latest, absent, etc.
S46	Use a valid 4-digit octal value or symbol for a File Mode parameter (1)	
S57	Do not ignore failed Tasks (2)	The ignore_errors setting swallows all errors, even ones you may not be expecting, and you risk leaving your host in a broken or unstable state.
S57	Handle errors with Built-ins handlers (1)	You can achieve less-brittle error handling and better control over how your tasks succeed or fail using the following Ansible built-ins failed_when, changed_when, wait_for, etc,
S57	Avoid skipping tasks (1)	because it might hurt idempotency. What if your Ansible playbook adds a cronjob based on a boolean variable, and later you change the value to false? Using when: my_bool (value now changed to no) will skip the task, leaving the cronjob intact even though you expected 18/6/2019 Andreas Sommer – I'm a software engineer – Blog – Ansible best practices https://andidog.de/blog/2017-04-24-ansible-best-practices 8/22 it to be removed or disabled. Here's a slightly more complicated example: I had to set up a service that should be disabled by default until the developer enables it (because it would log error messages all the time unless the developer had established a required, manual SSH tunnel).
S54	Verify service state (1)	Verify that the service you started is actually running! Because you declared it in a playbook does not mean that it is working. You could do this in your playbooks by using "uri", "waitforconnection" or any other validation method
S57	Use sudo only where necessary (1)	The command failed, so I used sudo command and it worked fine. I'm now doing that everywhere because it's easier. It should be obvious to devops people, and hopefully also software developers, how very wrong this is. Just like you would not do that for manual commands, you also should not use become: yes globally for a whole playbook. Better only use it for tasks that actually need root rights. The become flag can be assigned to task blocks, avoiding repetition. Another downside of "sudo everywhere" is that you have to take care of owner/group membership of directories and files you create, instead of defaulting to creating files owned by the connecting user.
S51	Secure logging (1)	If you use the template-module and there are passwords or other sensitive data in the Øle, you do not want these to be shown in the Ansible output. That's what the no_log-option is for. If added to a task, the output will not be logged. To keep sensisitve data in your playbooks and roles secret, use ansible-vault. There's extensive documentation from Ansible with good examples so I won't cover this topic here.
Module		
S49	Do not use non-idempotent modules (1)	Shell commands are less likely to be idempotent.
S49 S48 S55 S51	Use task-specific modules instead of general modules (6)	Where possible, use Ansible's task-speciÕc built-in modules rather than the shell or command modules. Shell commands are less likely to be idempotent. Shell commands will always run and will always report "changed," unless you're diligent about using changed_when . Many modules are designed to be operating system agnostic, which also helps you write more reusable code. Run commands are what we collectively call the command , shell , raw and script modules that enable users to do commandline operations in different ways. They're a great catch all mechanism for getting things done, but they should be used sparingly and as a last resort. The reasons are many and varied. The overuse of run commands is often a symptom of TL;DR in Ansible and common amongst those just becoming familiar with Ansible for automating their work. They use shell to ßre off a bash command they already know without stopping to look at the Ansible docs. That works well enough initially, but it undermines the value of automating with Ansible and sets things up for problems down the road. The most important thing to consider is that these run commands have little logic to them and no concept of desired state like a typical Ansible module. That shell that succeeded the ßrst time you ran your play may fail the next time when something already exists. That's unless you ignore_errors on that task. But how do you catch a real error like wrong permissions? Now you have to register the result of that ßrst command and follow it with another task that implements conditional logic to check if an error occurred in the ßrst and handle it. This one should be obvious, but for people that come from a classic admin-background and are new to Ansible it

		often is not: Ansible is batteries-included and comes with more than 1000 modules to help manage systems. Most times it's not needed (nor useful!) to fall back to shell commands instead of using modules.
S49 S51	Use template or copy modules instead of lineinfile or blockinfile modules (3)	
S55	Reloading instead of restarting services upon configuration changes (1)	Reloading a service does not cause the service being inactive, it simply reloads the configuration. When you edit a configuration rather creating a handler which restarts the service, make it reload. For example, after adding a new virtual host instead of restarting, you could simply reload apache and it will not cause any service disruption. Reload: Does not stop the service, it reloads the updated configuration Restart: Stops and then Starts the service. Your users might experience a slight downtime.
Variable		
S53 S54	Name variables consistently (2)	If you want to maintain your code, keep the name consistency between your plays, inventories, roles and group variables. Use the name of the roles to separate different variables in each group. For instance, if you are using the role nginx under webservers play, variables that belong to nginx should be located under group_vars/webservers/nginx.yml. What this effectively means is that group_vars supports directory and every file inside the group will be loaded. You can, of course, put all of them in a single file as well, but this is messy, therefore don't do i Variables should be unique to each role, descriptive and a name convention should be used, i.e: mysql_dnsname1
S53 S52 S51 S57 S54 S50 S48	Prefix variables with context (8)	Meaningful Variable Name As a best practice, we recommend prefxing variables with the source or target of the Task/Play it represents. You can also dramatically improve the readability of your plays for a bit of extra verbosity with using human-meaningful names that communicate their purpose and usage to others or even yourself at a later date. There are some things you should consider when writing variables for your roles. The first thing is that you should prefix them with the name of the role. This makes it easier to know where the variable is used. Role variables should be preÕxed with the role name (e.g. mysql_database_name) because Ansible has no concept of namespaces or scoping these variables only to the role. This helps Õnding out quickly where a variable comes from. In contrast, host groups in Ansible are a way to scope variables so they are only available to a certain set of hosts. As a best practice, we recommend preßxing variables with the source or target of the data it represents. Preßxing variables is particularly vital with developing reusable and portable roles.
S46 S56 S53	Logically group variables (3)	However, soon as your project starts to get bigger, and more you spread variables here and there, more problems you will encounter. Therefore it is good practice to keep all your variables in one place, and this place happen to be group_vars. se the name of the roles to separate different variables in each group. For instance, if you are using the role nginx under webservers play, variables that belong to nginx should be located under group_vars/webservers/nginx.yml. What this effectively means is that group_vars supports directory and every file inside the group will be loaded.
Inventory		
S46 S57 S55	Different inventory for different environments (4)	As also men oned above, a good way to keep your staging (or tes ng) and produc on environments separate is to use a separate inventory fle for staging and produc on. This way you pick with -i what you are targe ng. Keeping them all in one fle can lead to surprises! Tes ng things in a staging environment before trying in produc on is always a great idea. Your environments need not be the same size and you can use group variables to control the differences between those environments. Place environment specific variables in its own directory. Have different inventory for different environments. (this is important for multiple environment configurations)
S46 S57	Use dynamic inventory for abstracting dynamic environments (3)	If you are using a cloud provider, you should not be managing your inventory in a sta c fle. See Working With Dynamic Inventory (intro_dynamic_inventory.html). This does not just apply to clouds – If you have another system maintaining a canonical list of systems in your infrastructure, usage of dynamic inventory is a great idea in general. Who needs to fiddle around carefully in check mode every time you change a production system, if there's a staging environment which can bear a downtime if something goes wrong? Dynamic inventories can help separate staging and production in the most readable and — you guessed it — dynamic way. Separate environments like test, staging or production of course have different properties like IP addresses and networks Host and domain names (FQDN) Set of hosts. Production software may be distributed to multiple servers, while your staging may simply be installed on one server or virtual machine. Other values Ideally, all of these should be specified in variables, so that you can use different values for each environment in the respective inventory, but with consistent variable names. In your roles and playbooks, you can then mostly ignore the fact that you have different environments — except for tasks that e.g. should not or only run in production, but that should also be decided by a variable (→ when: not is_production).
S46	Group definitions of hosts based on their roles (1)	host groups in Ansible are a way to scope variables so they are only available to a certain set of hosts. We're somewhat repeating ourselves with this tip, but it's worth repeating. A system can be in multiple groups. See Working with Inventory and Working with Patterns. Having groups named after things like webservers and dbservers is repeated in the examples because it's a very powerful concept. This allows playbooks to target machines based on role, as well as to assign role specific variables using the group variable system.

Configuration Data		
S49 S55	Configuration file template (2)	Use templates Try to avoid using hardcoded variables and use Templates

Chef Best Practices

Source	Practice	Summary
Global Level		
S41	Use the standard Chef directory structure (1)	
S41	Use standard layout and content organization for each element (1)	
S41	Semantic versioning (1)	Use Semantic Versioning on any cookbook that is uploaded to Chef Server or published to Supermarket. Every cookbook declares a public API, and thus it is appropriate to use SemVer. If you aren't using Test Kitchen, this means bumping the patch version every time you try a new change. You should be using Test Kitchen. When creating a new cookbook, move quickly to create a 1.0 version. You should definitely not spend more than a week creating 0.x.x releases. Cookbooks rapidly become dependent on each other, so you should acknowledge that you have a public interface by issuing a 1.0 release.
S41	Prefer optimistic over pessimistic version constraints (1)	Prefer using optimistic version constraints in your cookbook dependencies rather than pessimistic version constraints especially in community cookbooks. A single pessimistic version constraint in a deep transitive dependency is enough to block usage of new cookbooks that are, in many cases, perfectly safe to use. It's common practice in the Chef community to bump the major version of a cookbook when dropping support for old versions of chef-client, but otherwise keep the API mostly intact. Use an optimistic version constraint (like >= 1.4.0) if you require particular functionality that is somewhat recent. Use a blocking version constraint (like <= 3.0.0) if you depend on a cookbook but are not yet compatible with recent versions. Only use pessimistic constraints (like ~> 1.0) on dependencies in cookbooks that are private and not likely to have other cookbooks depend on them.
S41	Never decrement the version of a cookbook (1)	Never ever decrement the version of a cookbook. Failure to adhere is a violation of Rule #2 in SemVer 2.0. Chef-client will always use the highest-numbered cookbook that is available after considering all constraints. If Chef Server knows about a cookbook with a higher number than the one you just uploaded, then your code is not going to get run. Do not add a version constraint in your test environment to work around this; it will definitely bite you later on. Your build system should fail the build if the cookbook version has not been incremented beyond the last uploaded cookbook. This matters even more if you're publishing to Supermarket.
S41	Git repository per each cookbook (1)	Create a new git repository for each cookbook. This is known as The Berkshelf Way.
S66	Name elements uniformly for their system and component (1)	Name things uniformly for their system and component. For example: attributes: node['foo']['bar'] recipe: foo::bar role: foo-bar directories: foo/bar (if specific to component), foo (if not). For example: /var/log/foo/bar.
S66	Do not use hyphen in cookbook and custom resource names (1)	
S66	Never use hyphens in the names of cookbooks with resource providers (1)	
S66	Use single-quoted strings for values not being interpolated (1)	
S66	Use double quoted strings for variables used in whitespace arrays (1)	
S66 S45 S41	Use Ruby Style for Ruby-specific Code (3)	Based on the community driven Ruby Style Guide, Rubocop is a command-line tool that performs static code analysis of all files within your cookbook, Define arrays on multiple lines, and put a comma at the end of every line. This is one of the best features of Ruby.
S66	Prefer strings over Ruby symbols (1)	
Cookbook		
S66	Use the standard Chef cookbook structure (1)	
S43 S66	Name cookbook with context (2)	Suggested naming conventions: – Cookbook – for the application it manages Use a short organizational prefix for application cookbooks that are part of your organization. For example, if your organization is named SecondMarket, use sm as a prefix: sm_postgresql or sm_httpd.
S44	Generic cookbooks (1)	The idea of a generic cookbook is to focus on specific tasks that the service would provide, and nothing more. In its ideal form, it should only install necessary packages and provide an LWRP (Lightweight Resource/Provider) for available actions (e.g. enabling site in nginx). You should avoid configuring any service that is not strictly related to the particular cookbook (e.g. monitoring, firewall etc.) or at least make it optional and do not include in your installation recipes.
S42 S40	Wrapper cookbooks (5)	Pattern: Set custom attributes in a wrapper cookbook

S63 S44 S66		Roles are not versioned. Cookkbooks are. By setting your custom attributes in a wrapper cookbook and pinning environments to specific cookbook versions, you can roll out attribute changes to dev, then test, then staging, then prod. A common solution to this problem is to use a wrapper cookbook. Wrapper cookbooks make use of the dependency feature we covered in Part One to allow us to depend on and include multiple recipes, similar to the way Sam used the run_list parameter in their role in Part Two. Unlike roles, however, cookbooks are versionable, and multiple versions of the same cookbook can be stored on a Chef Server. This not only allows us to more easily track the development history of cookbooks we create, but provide a means to more safely evaluate the changes we make. Chef environments can be configured with cookbook version constraints, where only the defined versions will be consumed by that environment, even if a newer version has been made available. Wrap when you need to change something Simply put, a wrapper cookbook is just a regular cookbook that includes recipes from other cookbooks. Common use cases for wrapper cookbooks include: Modifying the behavior of a community cookbook from the Chef Supermarket Bundling several base cookbooks into a single cookbook Version controlling a node's run list and attribute definitions
S44 S40	Role cookbooks (2)	In Shelly Cloud, we use the concept of 'role cookbooks' instead of the built-in roles structure in Chef. In basic terms, this means creating a wrapper cookbook for each role in your infrastructure, which then aggregates a generic cookbook and includes it in your nodes instead of roles. There are numerous advantages of using this approach, such as: cookbooks are versionable creates another layer for some additional organization-specific logic (e.g. some searches) can be distributed just like any other cookbook preserves site cookbooks not edited within our repository
S43	Lightweight Application cookbooks (1)	Application Cookbook • One recipe per deployable component • Define attributes and configurations needed for application – Attributes like application name – Property file(s) • Should be fairly lightweight • Should depend on one or more library cookbooks
S41	Create separate lightweight cookbooks for sharing attributes (1)	If you have two cookbooks with many dependencies that need to obtain a shared attribute, consider moving that attribute into a new lightweight cookbook or a role if it will be consistent across environments. This helps to avoid gigantic dependency chains that Berkshelf will struggle to resolve. Do not put it into an environment file if you expect it to be the same in all environments.
S41	Use a base cookbook for configurations relevant for each node (1)	Use a base cookbook for the essentials that are needed on every single node. If you have something that isn't needed on every node, it does not belong in base. A good example of what goes in a base cookbook is users and SSH configuration. Only add your base cookbook to Chef roles, never to a node run_ _list. Nodes should have only one item in the run list, and it should be a role. Application cookbooks should never depend on the base cookbook. Always add your base cookbook to the end of your run_list. Adding it at the end instead of the beginning helps to ensure that application cookbooks don't have hidden dependencies on your base cookbook. Try hard to minimize the number of cookbook dependencies used by your base cookbook.
S41	Use providers instead of recipes for writing reusable cookbooks (1)	Use providers instead of recipes when writing reusable cookbooks. Providers have the benefit that they can be instantiated multiple times within a single Chef run. This comes up more often than you would expect.
S44 S41	Use Berkshelf to manage dependencies (2)	As a consequence of generic cookbooks design, distributed across multiple repositories, we do not need (and even do not want) to edit the cookbooks directly inside our main workspace repository. Using a berkshelf helps to better organize your repository, and keep it clean from site cookbooks that, if manually edited, can lead to a huge mess and unexpected behavior in future.
S41	Test cookbooks with Test Kitchen (1)	Use Test Kitchen. Include a .kitchen.yml file with every cookbook. Make sure your cookbook converges locally before promoting or publishing your cookbook. This helps with http://12factor.net/dependencies.
S41	Always test against the latest version of Chef (1)	Always test against the latest version of Chef. You might want to pin a specific version of Chef to make Kitchen run faster, but remember to keep it updated.
S45	Create unit and integration tests for recipes and resources with ChefSpec and InSpec (1)	ChefSpec is a framework provided as part of the ChefDK to help create and execute unit tests for your cookbooks. An extension of RSpec, a behavior driven development (BDD) framework for Ruby, ChefSpec allows you to quickly test resources and recipes as part of a simulated chef-client run.
Recipe		
S41	Prefix private recipes with underscore (1)	Private recipes should start with an underscore. The absence of an underscore is an indicator that the recipe is meant for use in run lists, or included from recipes in other cookbook
S43	Recipe per managed component (1)	One recipe per deployable component
S63	Declarative recipes (2)	keep recipes declarative Keep the recipe declarative try to move logic elsewhere Everything action :nothing and notify is a extremely hard to follow Are kind of like GOTO statements what do we do instead? Use libraries and helpers Keep the recipe declarative try to move logic elsewhere
S41	Do not use the default recipe (1)	Don't use the default recipe (leave it blank). Instead, create recipes called server or client (or other)

S41	Use a private common recipe to share resources among sub-recipes (1)	Create a _common.rb recipe if there are multiple sub-recipes that would otherwise define the same resources (eg. a directory). Otherwise you might find yourself running into CHEF-3694 warnings.
Resource		
S66	Specify default resource action (1)	However, if readability of code is desired, such as ensuring that a reader understands what the default action is for a custom resource or stating the action for a resource whose default may not be immediately obvious to the reader, specifying the default action is recommended:
S41	Use guards instead of if statements for delaying resource action execution to runtime (1)	Understand Chef's Two Pass Model. Be aware of the difference between if statements and guards. One of them is handled during the compile phase, and the other during the execute phase.
Resource Provider		
S41	Use inline resources at the start of a provider (1)	Always use_inline_resources at the start of your provider. This is a kind of "strict mode". This is expected to become the default behaviour in Chef 13. "use_inline_resources is used to separate a LWRP's run context from the main run, making it run in an isolated mini-run. You cannot subscribe to / notify resources that are not part of this resource's context."
S41	Always define a :remove action to clean up resources created by the provider (1)	Always define a :remove action to clean up resources created by your provider. Imagine you have written a provider that installs an application into new_resource.prefix. Then somebody who is using your provider in a recipe decides to install to a different prefix. How should they clean up anything left behind? The answer is that they should change the action on the existing provider to :remove and declare a new resource using your provider with the new path. The tombstone resource needs to be left in the configuration long enough for all nodes to converge. Some people can sidestep this by using disposable infrastructure, but many people do not have that luxury
Attribute		
S41	Qualify attributes with cookbook’s namespace (1)	All attributes defined by your cookbook should exist within your cookbook's namespace. Use node[cookbook_name] to refer to attributes within your cookbook's namespace. If you ever copy or rename the cookbook, you won't need to make nearly as many changes. Caveat: cookbook_name is not defined in attribute files, so you need to set it as a local variable if you want to use it.
S41	Define cookbook-independent attributes in roles or environment files (1)	If you need to define attributes that aren't really owned by any particular cookbook, use node['global'] as the namespace and define them in roles or environment files. It is usually preferable to have attributes owned by a cookbook, but sometimes there are exception cases and the exceptions should be clearly marked.
S41	Do not create persistent node attributes (1)	Never use node.set or node.normal. Creating persistent node attributes is almost universally a bad idea. A normal Chef run starts with an empty set of attributes and populates them based on cookbooks, environments, and roles, all of which are visible in the source code. Attributes persisted onto a node are never visible in source code. They can produce confusing behaviour that is difficult to troubleshoot. The stored node attributes will impact all future runs until you manually remove them from the node. "Normal and override attributes are cleared at the start of the chef-client run, and are then rebuilt as part of the run based on the code in the cookbooks and recipes at that time."
S41 S40	Use attributes instead of hard-coded node names in recipes (2)	Avoid hardcoding node names (or patterns to match node names) anywhere in your recipes. Use attributes instead.
S66	Specify the file mode as a quoted 3- 5 character string of its valid octal mode (1)	
S66	Do not use static Unix Style paths (1)	
S41	Copy and use external attributes (1)	If you need to read attributes from another cookbook, copy them into the namespace of your own cookbook. Do this inside attributes/external.rb. Try to minimize the number of external attribute accesses. Use include_attribute to create sections within your external.rb. Although include_attribute is only a hint to the Chef compiler, it makes it much easier to search for dependencies in a code base when you want to refactor a cookbook.
S41	Keep derived attributes in a separate attribute file (1)	Never override other cookbooks' attributes inside your recipes. Always use attributes/override.rb.The best place to override attributes is in attributes/override.rb. Attributes files are normally used to define the inputs to your cookbook. By putting overrides into a specific file, you make it clear that they are not inputs.It is good practice to use include_attribute to include the attribute file that contains the values you are overriding. This provides fail-fast behaviour if the attribute file or the cookbook no longer exist. It also doubly ensures that your override takes precedence over the original value, even if they are at the same precedence level.
S41 S66	Prefer hashes of attributes to arrays of attributes (2)	The arrays are not recommended to define composite attributes as merging attributes of arrays is more tricky than hashes.
Databag		
S41	Name the data bag based on the names of the cookbook use it (1)	Try to name your data bag so that it matches the name of your cookbook. But due to the shared nature of data bags, this is often not possible.

S41 S43	Do not use data bags to store environment specific data (2)	Do not use data bags to store data that is different across environments. Environment files are very good at this already. Having two places to look ends up being very confusing
S43	Use data bags instead of attributes for storing global data (1)	So far, we have covered setting attributes in application cookbooks – Pros: • Forced standard in all environments • Versioned as part of a cookbook – Cons: • Can't change attribute values between environments • Hard to change "quickly and easily" • Data bags are the answer! – Data bag contains 1 data bag item (JSON file) per environment – 1:1 data bag to cookbook (same name even!) – Live configuration data including application version
S41	Avoid using data bags in public cookbooks (1)	Avoid using data bags in public cookbooks. Requiring a certain data bag structure forces people to manage their infrastructure in a certain manner. This is a violation of one of the guiding principles of Chef: you know your infrastructure best. The users cookbook is a big culprit here. It forces users to conform to a certain data structure, which rarely meets the ever-changing and unique demands of an organization.
S41	Use a specific recipe to refer to information from a data bag (1)	Create a data_bag.rb recipe in each cookbook that needs to look up something from a data bag. Put all of your data bag lookups into this recipe. This helps to reduce duplicate lookups.
S41	Minimize data bag lookups (1)	Minimize the number of data bag lookups that occur. A new HTTP request is sent every time you request a data bag.
S41	Always specify resource action in each item of data-bag collection (1)	If you are using your data bag as a collection, each item in the data bag should include an action attribute to explicitly define whether resources should be created or deleted. If you don't specify this, then you have the implicit assumption that all resources in the data bag will result in resource creation. Then you will have a bad time when you want to remove something from the data bag, because remnants will get left behind. Chef is only able to act on the presence of resources, not the absence of them. In order to clean up resources effectively, you need to use tombstones that last long enough for all nodes to finish converging the deletion. The exception is when your collection is used to render a single resource, such as when a template file is populated by all the items of a data bag. In this case, any data bag entries that are removed will automatically disappear from the template.
S63 S41	Prefer secret isolation with a Vault (2)	The bad idea list of Storing secrets: Encrypted databags - Bad YouReally bad are using only one key to give access to a lot of other secret information. Plus this single key is distributed to everynode to get access. does not support key rotation Databags - Really bad node attributes - Really, really, really bad error prone Source control - Terrible Tool: Chef-vault + Manages the secrets in databags + uses client keys to allow access to the vault + If you use auto scaling you cannot know the client key before so you cannot grant access + Good for static scales though Lots of other options like, hashicorp's hashivault, and .... ?? Avoid keeping secrets in Chef. If you must keep them in Chef, then environment files are the natural place for them. But consider using something like Vault instead.
S42	Use Knife to create data bags (1)	A data bag can be created in two ways: using knife or manually. In general, using knife to create data bags is recommended, but as long as the data bag folders and data bag item JSON files are created correctly, either method is safe and effective.
Environment		
S41	Always use uppercase for environment names (1)	Environment names are always uppercase. The name specified within the environment file should be an exact case-sensitive match of the filename. When you specify an environment with chef-zero it looks for a file with exact matching case. If the filename has different case than the usage in your recipes, you will have difficulty with matching and lookups. *(Although admittedly, you should never depend on the name of an environment within your code. Use attributes for that instead.)
S41	Prefer environment files over databags (1)	Environment files are great. Prefer them instead of data bags. Everything in the environment is available at the beginning of the run, whereas data bags require additional HTTP requests.
S43	Use environment files for cookbook versions and Run lists (1)	Use environment files for cookbook versions and run lists Both roles and environmentsc are unversioned, but roles tend to be more dangerous due to more global effect • Both CAN define global or environment-wide attributes • Environment files contain cookbook version to use • Simple to keep cookbook versions and run lists in same place • We do not use roles for these reasons
S41	Use service discovery tools instead of hard-coded IP addresses (1)	Avoid tracking the current addresses of backing services in your environment file. If you must keep them in Chef, environment files are the natural place. But consider using a proper Service Discovery tool like Consul or Zookeeper instead.
S41	Do not use the name of an environment in conditionals (1)	If you need to do something environment-specific, use an attribute instead. Never make conditionals dependent on the name of an environment.
Role and Node		
S41	Use only one entry in the run list for each node (1)	Use only one entry in the run_list for each node. Node run lists do not have the benefit of version control, and become very difficult to maintain once they grow long enough.

S42	Avoid setting attributes in roles (1)	Role attributes are a particularly dangerous anti-pattern which can can break your production environment. Imagine this scenario. You have a web_server role with attributes for names and properties of two web sites you need to create on that server. Now imagine you need to split those web sites into two separate server roles, app_server, and blog_server. How do you test it in your dev environment without breaking prod? You can either roll the dice and make the change, hoping your don't break your prod environment or you have to try to override those attributes in environment files (first dev, then test, then staging, etc) and remember to clean them up after you've reached prod. Neither option is really workable. You should use attributes in a wrapper cookbook instead.
S44 S40	Do not use roles for setting a run list (recipes) of node (2)	Chef's Roles seem ideally suited to specifying the run list of all the recipes needed to build the server. However, this pattern suffers from the same problems as using Role Attributes above. If you add or remove recipes from the Role's run_list, that change affects all servers in the role, including Prod. Keep your Roles lightweight. The list of recipes needed to build your application or web server should be kept in an "application" cookbook instead of a Role. For example if you are building a prototypical web server, you Role should look like this:
Configuration Data		
S41	Avoid using the node object within configuration templates (1)	Avoid using the node object within templates. If you need to use attributes in a template, add them to the template resource using the variables parameter.
S41	Never refer to attributes from multiple cookbooks in a configuration template (1)	Never refer to attributes from other cookbooks in your template.

Puppet Best Practices

Source	Practice	Summary \ Extracts
Project		
S67 S38	Use standard Puppet repository layout and content (2)	
S38	Use separate repositories for complex component modules (1)	
S67	Documenting modules with Puppet Strings (1)	
S38	Use a separate profile repository to delegate profile management to application development teams (1)	
S38	Put the Hiera data at the module level in a separate repository (1)	The pattern of managing hieradata in a separate repository is both common and acceptable, though should typically only be done in cases where there is an identified need to separate access or the development cadence. This pattern is described in the hieradata repository best practice.
S67	Use semantic versioning (1)	
S67	Follow the standard order of information when structuring classes and defined resource types (1)	
S67	Use the recommended spacing, indentation, and whitespace for manifests (1)	
S67	Define Array/Hash with multiple elements on multiple lines (1)	
S67	Use backslash as an escape character (1)	
S67	Explicitly specify iteration over arrays or hashes (1)	
S67	Consistent and appropriate quoting of string values (1)	
S67	All resource names or titles must be quoted (1)	
S38 S67	Specify File mode always as a 4-digits octal value (numeric notation) or a string (symbolic notation) (1)	
S67	Avoid complex conditional expressions (1)	
S67	Include defaults for case statements and selectors (1)	
S67	Avoid mixing conditionals with resource declarations (1)	

S38	Use puppet development kit (1)	The PDK is a Puppet maintained open source project for creating, validating, and testing puppet module code. It should be the preferred option for users when talking about creating new modules or writing new puppet code.
S67	Use hash comments (1)	
S61 S62	Use dry run mode (2)	Use dry-runs: Even with the best precautions taken, sometimes your Puppet manifest doesn't do exactly what you expected. Things can get messy at times whenever you actually get to run the Puppet agent to apply your configuration updates on your servers. For example, if it would update a config file and restart a production service this could result in unplanned downtime. Also, sometimes manual configuration changes are made on a server which Puppet would overwrite. To reduce the risk of problems, you can use Puppet's agent in "dry run" mode (also called noop mode, for no operation) using the following options: puppet agent […] –verbose –noop –test By making use of this practice you will be able to see the difference for all files that would modify as well as validate things according to your expectation. It will help Puppet a
S38	Use a supported Puppet architecture, including a Puppet master (1)	Problems with Masterless Puppet A significant proportion of the value Puppet provides as a configuration management tool is centralized control of Puppet managed systems. In the absence of a Puppet master, however, users inevitably either lose the benefits of central control, or must create bespoke re-implementation of services the Puppet master provides. There are significant maintenance costs for architecting and supporting your own solution to problems already solved by an off-the-shelf solution. Additionally, when a Puppet master is not used every agent must be given full access to the source manifests (not required when using a master) as well as all data inputs required to build its Puppet catalog. This either constrains what information can be provided to agents, requires integration of separate tooling to distribute secrets to agents, or results in loss of the "principle of least privilege" security benefit Puppet enjoys through limiting information given to each individual agent to only what it needs to know. Finally, beyond Puppet configuration management through manifests, when not using a Puppet master all other Puppet-provided services are effectively unavailable. This includes but is not limited to orchestration, tasks, and reporting. Masterless is not one of the approved Puppet architectures, which limits the amount of available information there is on how to implement it.
Module		
S67	Use standard Puppet module layout (1)	
S38	Name modules after technologies being managed by them (1)	Modules should be named after the technology it is managing, such as apache or iis and not a name that includes a verb, such as manage_apache or enable_iis.
S38	Keep component modules generic and cohesive (2)	Modules must only contain resources that are related to the issue the module is solving and not resources that do not immediately relate to the problem. E.g. a phpmyadmin module would not contain resources to manage the installation of Apache or MySQL, only resources to provide configuration for these that relates to phpmyadmin
S38	Use a single class for basic modules (1)	A basic module is a module where there is a single class that contains all the resources for that module. When creating basic modules the entry point should be the class named after the module (i.e. the init.pp file). This file can contain all the resources required to implement the solution that the modules addresses. The main class of a module is its interface point and should be the only parameterised class if possible, this allows the module author to control the usage of the entire module with the inclusion of a single class.
S37	Use install config-service pattern for complex modules (1)	ABOUT THE PACKAGE/FILE/SERVICE PATTERN ➢ Problem: your module's init.pp is getting too cluttered because all of your code lives in that one fle ➢ Solution: break out the basic functions of your module into separate classes, generally into a package, config, and service cla This is one of the frst patterns you see when learning Puppet ➢ This is the embodiment of the Single Responsibility and Separation of Concerns principles ➢ Most modules can be broken down into some form of Package, Confg File, and Service management ➢ Use this specifc pattern any time you write a module that manages these things ➢ Keep the spirit of this pattern in mind whenever you write a module that is more than a few lines long ➢ This has the added beneft of allowing us to utilize class containment for cleaner resource ordering
S67	Split modules into public and private classes and defined types to separate configurable behaviors from protected behaviors (1)	Public and private Split your module into public and private classes and defined types where possible. Public classes or defined types should contain the parts of the module meant to be configured or customized by the user, while private classes should contain things you do not expect the user to change via parameters. Separating into public and private classes or defined types helps build reusable and readable code. Help indicate to the user which classes are which by making sure all public classes have complete comments and denoting public and private classes in your documentation. Use the documentation tags "@api private" and "@api public" to make this clear. For complete documentation recommendations, see the Modules section.
S38	Do not define classes and defined resource types within other classes or defined resource types (1)	
S67	Use separate files for all classes and resource type definitions (1)	

S62 S61	Use Librarian-puppet tool to manage modules (2)	Use librarian-puppet Managing module dependencies can be a source of headaches, especially when many people are working on Puppet code and they each need to test it on their own computer. Librarian-puppet provides some sanity to the process by automatically managing your module dependencies. You express your dependencies in a file (the “Puppetfile”) and the tool will install, update or remove modules automatically when you run it, always matching what’s specified in the Puppetfile. It’ll even resolve and install the modules’ own dependencies (what we would call transitive dependencies) and detect compatibility issues. Using librarian-puppet on the Puppetmaster also allows for easier deployments: no need to install and manage your modules manually. With librarian-puppet, a deployment usually goes with two simple steps: Sync your main sources with your code repository (ex: git pull) Run librarian-puppet to synchronize your installed Puppet modules
S38	Metadata should include hard module dependencies (1)	Each module must contain a metadata.json file containing information, version, dependencies and intended operating systems for the module.
S38	Readme file should include soft dependencies (1)	Each module must contain a README.md in valid markdown format, containing at the minimum a description of the module and how to use the module.
S38 S67	Must define metadata for each module (2)	Each module must contain a metadata.json file containing information, version, dependencies and intended operating systems for the module.
S38	Use the puppet metrics collector module to collect metrics (1)	We recommend using the puppet_metrics_collector module to collect metrics. By default, the module stores metrics in /opt/puppetlabs/puppet-metrics-collector, and the data can be easily archived and shared with Support.
S38	All classes, defined types, and custom extensions should have associated testing (1)	
S38	Use the archive module to copy a large directory structure into place on an agent (1)	To copy a large directory structure into place on an agent, use the puppet/archive module. The module allows you to send a compressed tarball to the agent and extract it into place. You can combine this with recursive_fle_permissions to ensure permissions are managed on the fles a[er extrac☙on.
Class		
S67	Parameterized classes (1)	
S67	List required parameters before optional parameters (1)	
S67	Resources should be grouped in a manifest by their logical order (1)	
S67	Only declare multiple resources in a single block if they share a default set of options (1)	
S67	Include data types of input parameters (1)	
S36 S38	Don not use class inheritance (2)	We do NOT recommend using inheritance anywhere else in Puppet and for any other reason because there are better ways to achieve what you want to do INSTEAD of using inheritance. Inheritance is a holdover from a scarier, more lawless time
Resource		
S67	Names in resource type definitions must have a unique variable (1)	Because defined resource types can have multiple instances, resource names must have a unique variable to avoid duplicate declarations.
S37	Wrap resources to add new functionality to an existing resource (1)	Resource Wrapper: adding functionality to code you don’t own
S67	“ensure” attribute should be the first attribute specified (1)	
S67	“”*” attribute should be the last attribute specified (1)	
S67	Do not use “*” attribute multiple times in one resource body (1)	
S67	Declare symbolic links by using an “ensure” attribute and a “target” attribute (1)	
S67 S38	Use recursive file permissions resource type for managing permissions, owner, and/or group (1)	
S67 S38	Use recursive file resource type (recurse =>true) only for managing a small number of files (1)	
Task		
S38	Parameterized tasks (1)	Tasks should be parametrized rather then have hard-coded values in them.
S67	Use one input method for receiving input data (1)	
S38	Use task plans to orchestrate multiple tasks into a workflow (1)	Tasks shall not execute multiple puppet apply commands in a single task. Plans should be created with multiple tasks or puppet resource should be used in simple transitional state changes.
S38	Write puppet code for installing task	Tasks should not install their own prerequisites, puppet code should be written for dependencies (e.g.

	prerequisites or dependencies (1)	gem, pip). Tasks should focus on execution of just the action desired. Pre-requisites should be considered long term state and thus managed with puppet code.
S38	Minimize reimplementation of platform independent logic by reusing Puppet resources and facter (1)	Tasks should use puppet resource and facter as much as possible to minimize reimplementation of platform independent logic.
S67	Use task metadata to validate inputs and control its execution (1)	
S38	Use the simplest available languages to implement tasks (1)	Tasks should be written in the simplest available language on a given platform. (e.g. bash on Linux or powershell on Windows)
Role and Profile		
S38 S39	Name profiles according to the technology stack being modeled (2)	Profiles shall be named according to the specific technology being modeled. in Puppet code (e.g. profile::wordpress, profile::mysql, profile::iis, profile::java)
S38	Profiles applicable to a specific OS may be named accordingly (1)	Unique implementations of a technology stack may have independently namespaced Profiles (e.g. profile::ssh::server, profile::ssh::client) so long as those unique implementations are managed independently of one another. If both implementations are typically managed together, then a single Profile is sufficient.
S38	Name roles with node/machine type (1)	Roles shall be named generically according to machine "type" (e.g. role::application_server, role::database_server, role::middleware_host).
S38	Do not name Roles after specific technologies (1)	
S38 S39 S20	Modularize Puppet code with roles and profiles (3)	
S38 S39	Parameterized profiles (2)	Profiles may be parameterized to provide an API to the implementation of a technology stack.
S39	Use a single profile wrapper module defining all profiles (1)	
S38 S39	Use sub-profiles for specializations (2)	
S38 S39	Hide sub-profiles with a super profile (2)	
S38	Prefer role-per-node (1)	
S38	Roles shall not contain resources (1)	
S38	Use Hiera lookups and business logic within a role only for dynamic declaration of profiles (1)	Hiera lookups may be included in Roles only if the lookup aids in the declaration of Profiles
Variable		
S67	Use variables if repetitive values within a manifest (1).	
S67	Prefer hardcoded values for non-repetitive and shared values (1)	
S67	Only use numbers, lowercase letters, and underscores for variable names (1)	
S67	Explicitly specify absolute namespaces for variables (1)	
S67	Refer to facts using the $facts hash (1)	When referencing facts, prefer the $facts hash to plain top-scope variables (such as $::operatingsystem). Although plain top-scope variables are easier to write, the $facts hash is clearer, easier to read, and distinguishes facts from other top-scope variables.
Hiera		
S67 S38 S58	Use Hiera to separate configuration data from code (3)	
S38 S37	Use Hiera hierarchy only for addressing operating system differences (2)	
S67	Use Hiera data to set parameter defaults of classes (1)	Adding default values to the parameters in classes and defined types makes your module easier to use. Use Hiera data in your module to set parameter defaults.
S67	Use Hiera lookups for separate private data from code in classes (1)	
S38	Use Hiera to assign classes to nodes (1)	
S67	Avoid using calls to Hiera functions in public modules (1)	You should avoid using calls to Hiera functions in modules meant for public consumption, because not all users have implemented Hiera. Instead, we recommend using parameters that can be overridden with Hiera.

S36	Do not do Hiera lookups in component modules (1)	Do NOT do Hiera lookups in your component modules! This is something that's really only RECENTLY been pushed. When Hiera was released, we quickly recognized that it would be the answer to quite a few problems in Puppet. In the rush to adopt Hiera, many people started adding Hiera calls to their modules, and suddenly you had 'Hiera-compatible' modules out there. This caused all kinds of compatibility problems, and it was largely because there wasn't a better module structure and workflow by which to integrate Hiera. The pattern that I'll be pushing DOES INDEED use Hiera, BUT it confines all Hiera calls to a higher-level wrapper class we call a 'profile'. The reasons for NOT using Hiera in your module are: By doing Hiera calls at a higher level, you have a greater visibility on exactly what parameters were set by Hiera and which were set explicitly or by default values. By doing Hiera calls elsewhere, your module is backwards-compatible for those folks who are NOT using Hiera Remember – your module should just accept a value and use it somewhere. Don't get TOO smart with your component module – leave the logic for other places.
Security		
S61 S62	Separate secrets from code with Hiera (2)	Today Data security is the key concern for any organization. Some data always needs to be kept safe. For example, for maintaining the data security it may be required to put in your Puppet code in passwords, private keys, SSL certificates and so on. Don't put Puppet code in version control unless you're absolutely aware of the risks you're taking while doing so. If you're already a bit familiar with Hiera which is a Puppet tool, you know why it's a good idea to separate your data from your Puppet manifests. In case you aren't familiar, though, using Hiera leases you write and use reusable manifests and modules. After you store your organization-specific data in Hiera, Puppet classes can request the data they need from your Hiera data store. Hiera allows you to store data about your servers and infrastructure in YAML or JSON files. From usage, you'll see that most data in Hiera files is not confidential in nature… so should we refrain from using version control for Hiera files just because of a few elements that are unsafe? CERTAINLY NOT!!!!!!!!!!!!
S38	Use facts from the trusted facts array ($trusted) (1)	All Puppet users using the Node Classifier and using rules should use variables from the trusted facts array (`$trusted`). These facts are included in the node's certificate or generated on the master and, as such, cannot be changed on the node.
S38	Explicitly state if task parameters are sensitive (1),	Sensitive parameters Use the Sensitive data type to mask parameters that should not be displayed in logs. When you pass a value to a Sensitive parameter, Bolt automatically masks the value before the plan is run
S38	Use of policy-based autosigning of Puppet certificates (1)	
Configuration Data		
S51	Use configuration templates (1)	Don't patch config files, manage them with template
S38 S67	Use template reading functions that can validate the templates being read (2)	The template function can take any number of additional template files, and will concatenate their outputs together to produce the final string.

Bad Practices

Language-Agnostic Bad Practices

Source	Practice	Summary
S28	Data as code (1)	(Configuration) Data has a different lifecycle. It's more dynamic. Example 1: use your provisioning tool to define organization users. Example 2: manifest that lists all your 500 servers
S28	Fancy configuration file copying (1)	To configure package X, you keep all configuration files it needs within your "code". You use provisioning tool abstractions to copy every single file onto the target system
S28	Not treating IaC as Code (1)	Code must be in Version Control. Lack of experience with new tool may Code Reviews. Yes, there are tools for Static Code Analysis even for IaC products. Unit testing does not make a lot of sense for IaC, but Integration Testing does. Applying all the above techniques gives the best QA result for any code.
S28	Ignoring styling guidelines (1)	Each tool/language out there has one. Nobody canceled clean code teachings. Reading, writing and eventually merging code is always easier if people follow the same formatting and styling.
S28	Automation over documentation (1)	It's quite common that Ops team have been given or have created a bunch of documents describing procedures for system operations. Code can do better! It happens that writing those documents take as much time as writing and testing code that implement the same guide lines. Automating procedures can reduce the amount of documentation needed or eliminate the documentation completely.
S28	Private fork of a community module (1)	There is a lot of code out there. Private fork may work as a shortterm solution. Do not keep your updates only to yourself. Share them back. It's better to create a wrapper. This simplifies upgrades. And tracebilty.
S18	Version control by copy and paste (1)	There are a couple of problems with this approach though, firstly, moving changes between environments is what I call version control by copy and paste. By which I mean when a new change is ready to be released it must be copied from the dev file to the staging file and up to the production code. This opens you up to all sorts of mistakes and problems, it also makes automated releases tricky.
S28	Non-Reproducible image (1)	Manually crafted base infrastructure server image that nobody dares or knows how to change.
S31 S6	Non reproducible environments (2)	Recently I worked for a customer to help him transforming his IT towards Continuous Integration. To do so, it was necessary to create a new environment to run system tests and verify the quality of the release. To implement the test automation including service virtualization was a doable challenge and worked as expected. But creating a working new environment for the applications we had in scope was almost impossible. The knowledge how to setup a suitable infrastructure and the application got lost. In addition to that, complicated and slow operation processes made it even worse. The Once a machine is created via an IaC workflow, it should not suffer intervention outside of an automated, aligned, and compliant maintenance workflow. Manual or external updates (even if just security patching) may result in configuration drifting which in time has the potential of producing massive non-compliance or even service failure.
S31	Mixed-up environments (2)	Each environment (Development, Test, Integration, ...) you use during your development process or as a part of your deployment pipeline should have a well-defined purpose. When companies start to use environments for different purposes or do not have clear rules here, chaos starts to take over. The worst thing you can do is to use one environment for manual and automated tests, because this will lead to failing automated tests due to issues with the test data or to inefficient manual tests. Because you can either set up an environment to run automated tests or to provide the right setup for manual testing - both is not possible.
S28	Postponing secret isolation (1)	"It's OK for now" does not really work! It creates a culture of security being not so important! It may alienate your Dev and Ops teams, because they can't share code due to hardcoded secrets!

Ansible Bad Practices

Source	Practice	Summary
S65	Overuse of comments (1)	Overuse of comments Ansible is declarative for a reason. Your code should document itself. Tasks should have descriptive names that explain what is

		happening in that task. Overuse of comments, however well-meaning, leads to congested code that requires more maintenance if changes need to be made. It also sets a dangerous precedent and other developers may start commenting with important information that will become buried in the code. Keep your README updated and your Tasks names sharp.
S65	Mixing YAML syntax styles (1)	Pick one! Yes, your code works but it's bad form and super annoying. If you're going to indent parameters, don't suddenly start equating them. Here's an example of this behavior for your entertainment.
Playbook		
S47	Single one-size-fits-all playbook (2)	Single playbook A single playbook relates to the first Sin again, but also applies to more focused playbooks where you only deploy one thing. Splitting your playbooks between various logically related roles will fasten your deployments. Again, why running ssh key distribution, storage cluster deployment, web stack, middlewares and application when you just change the color of a button in your web app ? Split your playbook in related parts that reflects your stack architecture. They will be faster and easier to use
S50 S48 S47	Including business logic in the playbooks (3)	No roles (tasks only) Well, this is obvious. Even if you don't want to share, make roles and strive for code reuse. Reused code will save you time of course, but it is also battle tested since it is used more frequently. Tasks-only playbook can be used for a quick hit and run, solving a transient problem that doesn't offer any code reuse opportunities. I also try to avoid tasks along roles in playbooks: this hurts the abstraction level you manage to build using roles. When thinking in terms of roles, you don't need to think about the nitty gritty details of the roles when reading your playbooks. If your roles are thouroughly tested, you can read your infrastructure in seconds. Add tasks to the mix, and you loose this superpow Do not put logic into plays - use role instead Ansible is a desired state engine by design. If you're trying to "write code" in your plays and roles, you're setting yourself up for failure. Our YAML-based playbooks were never meant to be for programming.
Role		
S50 S47 S65	Multiple responsibilities roles (3)	Nginx role will install and configure nginx. Nothing else. It won't create DNS entries, trim logs, add a ftp server or anything. It just installs nginx. Period. Is your role doing too much? When writing a role, keep in mind what the role must accomplish. Once you start expanding beyond the express purpose for the role, you have code that could install superfluous software/configurations on yours and other systems. Use role dependencies to combat role sprawl. For example: You want to write a role to install MySQL. You include all the necessary bits and pieces to get MySQL installed on multiple Linux distributions but you remember that one of your use-cases requires a JDBC driver. The little red devil on your shoulder says "Just add a task to the role! Who's it going to hurt?" The little angel pops up on the other shoulder and says "That's not an appropriate 6/13/2020 Organizing Ansible - Oteemo https://oteemo.com/2018/02/22/organizing-ansible/ 3/6 place to install that driver! That driver should be a dependency for whatever software requires it!" Besides the fact that you have 2 Do not install multiple services from a single role (e.g. Nginx + php-fpm) ○ Instead, use separate them into well parameterized roles (one for Nginx, second for php-fpm)
S50	Defining dependencies between atomic roles within roles (1)	• Do not define dependencies between atomic roles ○ List atomic roles in the Playbook instead ○ Dependencies in the atomic role should be for inheritance only (*wrapping roles*)
S65 S54	Non Idempotent roles/modules (2)	Failing to strive for idempotency If you aren't focusing on writing Idempotent roles, you should be. Most people run into this pitfall when using certain modules like Command. When using command alone, Ansible will always run the command and mark the task as changed even if nothing has actually changed on the host. Here is a simple example: I want to enable the optional repo if it's disabled. If I do it like this, the task will show as "changed" no matter what.
Task and Handler		
S51	Superfluous information in task names (1)	One thing you don't have to do, is to include the name of the role in the task-name. That's done automatically. See here:
S54	Restart services without using a handler (1)	Do not restart services without using a handler. Services restarts should always be done with a handler!
S54	Chaining handlers (1)	Do not chain handlers! If you do, tasks may fail if a previous handler fails
Module		
S54	Using general modules shell, command, raw, and script instead task specific modules (2)	When you develop playbooks and roles, avoid using the following: shell , command , raw , and script . Other modules should be used instead If you can't avoid using one of this modules, test what you are executing and ensure that it is idempotent If you are using shell tasks as a handler, ensure that the task calling the handler comes from a module that is idempotent

Inventory		
S47	Per-host fine grained variables in inventories (1)	when you try to encompass your whole infrastructure, you start to think, inheritance, variables overriding and refining. And while doing this, you add considerable complexity to your inventories. It is very hard to track down variables definitions when you overrides them in group_vars/some_group , group_vars/all , hosts_vars/machine , role defaults, … Now this can get even worse when you use the hash_behavior: merge Ansible configuration setting: it introduces more confusion, and makes your Ansible work potentially unshareable with people using hash_behaviour: replace . Since I am guilty on this one, it is time to make some apologies. Sorry folks. Michael DeHaan did not like it, and he was right.
Variable		
S57	Hardcording variable (1)	The database name preÕx ourprefix_ seems to be a hardcoded string. First of all, this led to a bug — privileges are not correctly applied to the user in the second task because the preÕx was forgotten. The hardcoded string could be an internal variable (mark those with an underscore!) deÕned in the defaults Õle roles/mysql/defaults/main.yml : _database_name_prefix: 'ourprefix_' # comment describing why it's hardcoded , and must be used wherever applicable. Whenever the value needs changing, you only need to touch one location.

Chef Bad Practices

Source	Practice	Summary
Cookbook		
S40 S63	Changing a community cookbook (2)	However, much as installing the chef-client cookbook is a near-universal best practice, so too is directly editing a community cookbook a near-universal anti-pattern. Consider what happens when a new version of the cookbook is released. It may have updated functionality that Sam will want to take advantage of, but to do so, they'll first need to re-apply their update, as they will with each subsequent release Use the community cookbook as is - and pin the version (a "quality pin") Wrap when you need to change something
Recipe		
S63 S43	God recipes (2)	Recipes should be lightweight and very focused Need to make shorter recipes no God functions! It hides logic keep it simple
S63	Complex branching logic (1)	Using action :nothing and notifying later is a -w/ recipes with many resource calls (for example: action:nothing, w/ notify later on), it's difficult to follow due to the branching logic.
S63	Numerous resource notifications (1)	Everything action :nothing and notify is a extremely hard to follow Are kind of like GOTO statements
Attribute		
S63 S44	Avoid Conditionals (on node attributes) with many branches (2)	Conditionals that use node attributes creating a lot of code branches This is untestable
S63	Customize all the things using attributes (1)	Trying to customize all the things using attributes Attributes are global variables so using them often does not allow you to use them in a reduced scope. Using too many is a code smell Old pattern: an attribute for every single configurable thing. New pattern: Model the configuration you need and then... teach chef to create the configurations for you Node attributes are leaky... passwords are problematic because they're insecure. Sprawl with attributes: Too many attributes... indicator that your cookbook does too many things.
Role		
S42	Setting a server's run list in a role (2)	Anti-Pattern: Setting a Server's Run List in a Role Chef's Roles seem ideally suited to specifying the run list of all the recipes needed to build the server. However, this pattern suffers from the same problems as using Role Attributes above. If you add or remove recipes from the Role's run_list, that change affects all servers in the role, including Prod.
S42	Using role attributes (1)	Role attributes are a particularly dangerous anti-pattern which can can break your production environment. Imagine this scenario. You have a web_server role with attributes for names and properties of two web sites you need to create on that server. Now imagine you need to split those web sites into two separate server roles, app_server, and blog_server. How do you test it in your dev environment without breaking prod? You can either roll the dice and make the change, hoping your don't break your prod environment or you have to try to override those attributes in environment files (first dev, then test, then staging, etc) and remember to clean them up after you've reached prod. Neither option is really workable. You should use attributes in a wrapper cookbook instead.
Security		
S63	Storing secrets as Encrypted data bags (1)	The bad idea list of Storing secrets: Encrypted databags - Bad YouReally bad are using only one key to give access to a lot of other secret information.

		Plus this single key is distributed to everynode to get access. does not support key rotation Databags - Really bad node attributes - Really, really, really bad error prone Source control - Terrible

Puppet Bad Practices

Source	Practice	Summary
Project		
S36 S30	Everything in separate repositories (2)	Everything in Separate Repos In the example above, I've mentioned a "profile" class. A common pattern when dealing with Puppet code is the Roles and Profiles Pattern. The idea is that you assign one "Role" per server and the role is made up of individual bite-sized "Profiles". Roles and profiles are just you or your companies custom Puppet modules that make use of upstream modules or Puppet resources to configure systems. A common and problematic pattern I've seen is maintaining an SCM repo for roles, an SCM repo for profiles, an SCM repo for the "control repo" and a separate SCM repo for hieradata. This can then turn into multiple branches of each and before you know it you're maintaining 12 versions of a diverging code base. It's common to see this when people have followed some "best practice" blog post without fully thinking things through. You also often see this when developers created the code initially. They love git branches and complexity. Especially if trying to bring Gitflow to Puppet code. Here's my top tip when deciding how to organize your repos: KISS - Have no more than one repo for your products Puppet code. Multiple repos and branches leads to managing multiple very different code bases, different module versions in each environment, merge tasks, complex git fixing… a nightmare So, there is a way to fix that problem, of course, you just have a separate configuration management branch for every cluster you run in production, and restrict developers to running code on only the systems they manage. Now they can only shoot their own team in the foot, but great, now how do you manage the common infrastructure code? Maybe you use git submodules? It becomes a hairy mess really quickly.
Module		
S34 S36	Monolithic modules repository (2)	Monolithic modules Directory Quite often I see repos where people have puppet module install'd straight into the `modules directory or they've downloaded a module and extracted it there. The whole repo including their own modules mixed in with upstream modules is then committed to source control. This pattern has a few problems: 1. you don't know what is a locally developed module and what is an upstream module 2. there's no way of easily seeing what versions of modules are deployed 3. it adds a lot of extra code to your Puppet repository Although this way works and you know that your module versions are pinned, tools are out there that make it much easier to manage your Puppet modules such as librarian-puppet and r10k. Each model has its pros and cons, but we tend to recommend one module per repository for the following reasons: •Individual repos mean individual module development histories •Most VCS solutions don't have per-folder ACLs for a single repositories; having multiple repos allows per-module security settings. •With the one-repository-per-module solution, modules you pull down from the Forge (or Github) must be committed to your repo. Having multiple repositories for each module allow you to keep everything separate
S34	Everything in general Manifests (1)	In many beginners tutorials you get taught to put all your code in manifests such as site.pp or nodes.pp . Although, it's not the best idea to manage your infrastructure in this way it's actually a reasonably good way to very easily and simply bootstrap cloud instances. with separate manifests based on server type (web.pp , app.pp , lb.pp etc). These can than be applied using cloudinit to create an immutable bootstrapped node.
Class		
S38	Imperative logic in classes (1)	imperative statements (such as the resource type \textsc{exec} to run ad-hoc OS commands) can break one of the key philosophies of the language: the declarative configuration model
S58	Hardcoding variables (1)	assigning a variable. In the example below, if you opened up the Puppet manifest and changed the owner from 'root' to 'puppet', that would be considered hardcoding the value: Hardcoding has a negative connotation because typically, when someone would hardcode a value in a script, it represented a workaround where a data item is injected into the code — and mixing data and code means that your code is no longer as generic and extensible as it once was.
Task		
S38	Task managing a state already managed by puppet (1)	Tasks should not manage a state that is already managed by puppet code on the system. ● Exceptions would be if puppet runs are incorporated into the task plan(Not yet a feature in PE tasks) and puppet is expected to undo an ad-hoc configuration change as part of the automation timeline (e.g. starting a service after a db migration).
S38	Imperative tasks managing ad-hoc configurations (1)	Imperative Tasks shall not be used for ad-hoc configuration management. The exception would be when a configuration option is changed before a transition state. An example would changing

		a configuration file, stopping a service, then reverting the configuration change after restarting a service.
S38	Tasks re-implementing existing puppet functionality (1)	Tasks should not reimplement existing functionality of puppet or the puppet tasks framework. Feedback from the task shall always make its way back to the Puppet Enterprise console. Ex. Don't implement a different logging framework or host filtering logic within a task rather than using PQL.
S38	Not using Puppet plans for executing multiple tasks (1)	Tasks shall not execute multiple `puppet apply` commands in a single task. Plans should be created with multiple tasks or `puppet resource` should be used in simple transitional state changes.
Role and Profile		
S38	Roles named after specific technologies (1)	Roles shall not be named a[er specifc technologies (e.g. role::tomcat , role::jboss) because the specifcity is in direct confict with the purpose of “Profle” wrapper classes.
S38	Roles containing resources (1)	Roles shall not contain resources - only declara□ons of Profles and condi□onal logic.
S38	Profiles named after generic server types (1)	Profles shall not be named a[er generic server types (e.g. profile::web_server , profile::db_server) because the lack of specifcity is in direct confict with “Role” wrapper classes used for classifca□on by server type.
S38	Profiles containing resources from component modules (2)	Profles shall not contain resources from Component Modules. Only resources containing site-specifc data shall be declared inside Profles to preserve modularity
Environment		
S34	Using multiple Puppet environments for a single infrastructure (1)	Puppet environments are a powerful thing, but I don’t believe you should confuse Puppet environments with Application environments. You should aim to manage your infrastructure in a single Puppet environment: "production". If you arrange your hieradata hierarchy sensibly you can manage the differences in configuration in a single branch. Puppet branches should be used when you need to test big changes or new features out in a controlled way (of course you’re developing and testing on Vagrant). Create a branch like "new_feature", develop it locally testing it on Vagrant then test out the changes on a suitable Tim Birkett © 2018 Latest Posts GitHub HubPress Tim Birkett Read more posts by this author. Read More 0 Comments pysysops 1 Login t Tweet f Share Sort by Best LOG IN WITH OR SIGN UP WITH DISQUS Name Start the discussion… ? Be the first to comment. ✉ Subscribe d Add Disqus to your siteAdd DisqusAdd ﬀ Disqus' Privacy PolicyPrivacy PolicyPrivacy Recommend piece of infrastructure by running puppet apply --environment new_feature . Don’t forget your security and perf testing at this point ;) if everything looks good, open a PR, get it reviewed, merged to production and delete the branch
Security		
S38	Using facts from the unsecure facts array (1)	Using facts from the standard facts array ($facts) is discouraged as these variables are generated on the node and therefore can be changed leading to a possible change in classification. From a security and operational perspective this is not desirable as it could be used to get configuration the node should not have, potentially including sensitive data(passwords, etc).
S38	Developing an autosign system from scratch (1)	Rolling something yourself. Completely wri□ng an autosign system from scratch is discouraged as it is a lot of work given that there are already well-tested, extensible solu□on being used in produc□on at many customers. If however an autosign system was wri,en from scratch it should have the following a,ributes:
Configuration Data		
S34	Configuration data in code (1)	When writing Puppet code it’s sometimes tempting to hard-code things like IP addresses or node specific things.This works, but the code isn’t re-usable. If you deploy to a different network or DC are your DNS servers still the same? To improve re-usablility, change the variable to be a class parameter with an optional default value:Then you can specify environment (network, node, DC) specific configuration in Hiera: Now you avoid multiple classes or writing case or if {…} else {…} logic in your class file.
S38	Using data-in-modules in the control repository (1)	Use data in modules in the control repo. This should be discouraged due to access restric□ons not providing any addi□onal capabili□es and does add addi□onal complexity that can confuse the loca□on of where to add data for the profles layer.