

Sources	Best Practices	Practice Summary /Extracts
S22 S09 S08 S14 S69 S70	Automation	<p>1. Ensure all infrastructure specifications are clearly coded within configuration files, establishing them as the primary source of truth.</p> <p>2. Enable seamless infrastructure deployment by relying solely on the information present in the configuration files, avoiding the need for manual adjustments whenever possible.</p> <p>3. Emphasize the importance of codifying all aspects of our infrastructure to promote reproducibility and minimize human error risks associated with manual configuration.</p>
S14 S29 S22 S4 S6 S7 S13 S61 S19 S23 S26 S27 S28 S32 S33 S8 S69 S70	Version Control	<ul style="list-style-type: none"> - Utilize a version-controlled database change management tool for configuration files. - Enable an audit trail, collaboration, and testing of IaC code through code reviews. - Implement code branching and merging best practices for efficient management of IaC updates. - Track all infrastructure changes and enable easy rollback if needed. - Require infrastructure modifications to go through Git repository changes and PR reviews. - Maintain code, documentation, test cases, and scripts in a central repository for consistency. - Easily define and clone configurations to ensure current and consistent documentation.
S22 S18 S8 S28	Modularity	<p>To simplify management, divide the infrastructure into separate modules or stacks and automate their combination during deployment.</p> <p>Set access controls to regulate who can modify specific parts of the infrastructure code, accommodating different teams and individuals.</p> <p>Enforce configuration discovery to maintain stability and control the number of configuration changes.</p> <p>Adopt a microservices-oriented approach, aligning infrastructure configurations with individual microservices' needs.</p> <p>Promote modularization to reduce complexity and eliminate duplication of key logic.</p> <p>Design IaC with flexible, reusable blocks that can be assembled on-demand for quicker adaptation to changing requirements.</p>
S22 S7 S20 S28 S9	Document the Code	<p>The source code itself serves as comprehensive documentation, eliminating the necessity for extensive additional instructions for users.</p> <p>IaC code acts as self-documentation, reducing discrepancies between infrastructure and written guidance.</p> <p>Written documentation is not a priority since the code continually reflects the current infrastructure state, providing up-to-date documentation.</p> <p>While diagrams and setup instructions aid knowledge sharing, the focus remains on the code as the most accurate and reliable documentation source.</p>
S32	Distributed Repository	<p>Utilizing a unified repository for the entire stack.</p> <p>Aligning application releases with infrastructure or configuration modifications for synchronized deployments.</p> <p>Facilitating the provision of temporary ad-hoc infrastructure during the deployment process.</p> <p>Allowing seamless integration of application releases with essential infrastructure configuration changes.</p>
S28	Gradual Configuration	<p>Rather than replicating default package states, concentrate on specifying incremental modifications in your code.</p>
S28 S69 S70 S71	Configuration Data Source	<p>Opt for a suitable data storage (like a database) to house configuration data, especially when handling numerous items.</p> <p>Avoid hardcoding values in the IaC to ensure easier adaptability and maintenance.</p> <p>Make the IaC parameterized to enable dynamic configuration based on specific requirements.</p> <p>Use absolute paths instead of relative paths in the IaC for precise and dependable file references.</p>
S28 S69 S70	Reproducible Image	<p>Antipattern: Creating server images manually without proper documentation or understanding of modifications.</p> <p>Pattern: Embracing reproducible images based on established operating system distributions (*.iso).</p> <p>Leveraging base provider images to build upon the infrastructure.</p> <p>Utilizing Packer for generating images compatible with multiple virtualization software and cloud providers.</p> <p>Utilizing Docker to build and distribute containers as portable and shareable images.</p>
S21 S31 S22 S10	Immutable Infrastructure	<p>Software updates and fixes involve deploying new servers from modified base images, while removing the old servers.</p> <p>Advantages:</p> <ul style="list-style-type: none"> Immutable infrastructure simplifies maintenance by eliminating patching and in-place upgrades, reducing corner cases and inconsistencies in server deployments. Configuration drift and one-off instances are avoided, leading to a more consistent infrastructure. Security is improved as administrative ports like SSH and RDP are not kept open on servers. The risk of unexpected impacts due to undocumented changes in the stack's configuration is minimized.
S19 S28	Environment Template	<p>Develop templates catering to various infrastructure levels like staging and production, enabling the creation of multiple instances from the same template.</p> <p>Construct detailed templates offering complete working environments with scalability, isolation, and adaptability features.</p> <p>Ensure clear and specific specifications for infrastructure needs, including network bandwidth and storage I/O operations per second, to avoid any omissions.</p>
S28	Package Application for Deployment	<p>Prepare your application in the optimal format for hassle-free deployment.</p> <p>Share the packaged application through an artifact repository, such as Maven, RubyGems, Yum, or Apt.</p> <p>The artifact repository serves as a buffer, providing isolation between pipelines and simplifying integration.</p> <p>This approach helps reduce the complexity of code needed in later stages of configuration management.</p>
S28 S69 S70	Infrastructure Query Language	<p>Use a language or API to query real-time or latest available reports on your infrastructure state.</p> <p>Collect data from IaC systems to keep track of the current status of the infrastructure.</p> <p>Employ Declarative Language and tools such as Puppet, Terraform, or Pulumi for infrastructure management.</p> <p>Be mindful of potential reproducibility problems caused by the current configuration and machine states.</p>
S28	Secret Isolation	<p>Secrets should be injected on the very last stage of "deploying" your code.</p> <p>Secrets should not be in code</p>
S28	Encrypted Secret	<p>all stored secrets must be encrypted</p> <p>Decryption password is shared through a different channel.</p>
S28 S7 S60	Collaborate	<p>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</p> <p>code reviews are a must for those embracing the notion of treating their infrastructure as code.</p>
S28	Metrics as Code	<p>Pattern: Metrics as Code Metrics that your application provides evolve with your application. New components, new endpoints, new KPIs...</p> <p>Keep monitoring configuration close to the code! Or make it auto discoverable and visible!</p> <p>Configuring and collecting metrics Monitoring software has configuration files and/or an API that can be programmed. There a plenty of libraries that allow</p>
S69 S70 S71	Dependency Declaration	<p>Dependency pinning → use correct format of the declaration</p> <p>Use a file to declare your minimum needed configurations/libraries</p> <p>Use standard libraries to maximize portability.</p> <p>Check for system-specific dependencies (e.g., operating system, hardware architecture).</p>

Ansible Best Practices		
Source		Practice Summary
Project		
S46 S51 S57	Use the standard Ansible project structure	
S46	Use semantic versioning	
S46 S48	Use native YAML syntax and conventions	
S55 S52 S54 S50	Use native YAML syntax and conventions	
S50 S57	Minimal Consistent quoting	Decide on a uniform quoting style, either double quotes or single quotes, and use it consistently throughout the code. Keep in mind that certain situations, like using variables or octal numbers, may require explicit quoting to ensure proper functionality.
S55 S53 S51	Vaults for Storing Secrets	As you gain experience with Ansible, more tasks can be automated. Eventually, sensitive information like SSL configurations and database passwords will need automation. Ansible offers the Vault module for securely storing and automating such data without risks. Vault allows encryption of important information for safe storage in version control systems or other environments. Storing passwords or certificates in plain text within repositories is not recommended, but ansible-vault enables encryption of sensitive data. The playbook has examples of both encrypted and commented-out plain text files. Decrypting files requires the vault password, which must be stored in the root directory but not committed to git repositories. For keeping confidential data secure within playbooks and roles, use ansible-vault, which is well-documented with helpful examples.
Playbook		
S47	Use tags only for speeding and debugging	The problem is that tagging every task in main.yml would be cumbersome, error prone, and clutter the code play execution unnecessarily.
Role		
S50 S69 S70	Parameterized roles Parametrizing scripts	List required parameters before optional parameters
S46 S49	Use roles to group related tasks	Roles in Ansible provide a way to bundle related tasks, variables, and dependencies together in a single, self-contained, and portable unit. Leveraging roles is an effective method to fully utilize Ansible's modular capabilities.
S50 S51	Use role documentation templates	When documenting Ansible roles, use the template provided by ansible-galaxy init. Include a description of the role's purpose and function, along with usage examples. List and explain the variables used in the role, preferably in the form of a table with variable name, default value, and explanation. Specify the dependencies required for the role to work correctly. Mention the role's author and provide information about the role's license.
S50 S54 S57 S69 S71	Test Roles with an emulated environment	In a CI model, ensure unit testing is performed for each role. Containers are a convenient choice for testing roles across multiple distributions. For low-level actions, such as bootloader setting and firewall configurations, use virtual machines for testing. Conduct thorough and consistent testing on various platforms to validate the role's functionality.
S49 S55 S54 S51	Use Ansible Galaxy to find and share Roles	
Task		
S48 S50 S55 S52 S51 S57 S46	Name tasks	While you can leave out the 'name' field for tasks, it's advised to include a description of the task's purpose. The 'name' field is visible during playbook execution. Always name plays and tasks with descriptive and human-readable information to improve communication with users when they run the playbook. Task names should be clear and specific, enabling easy understanding for end-users and other team members executing the playbook.
S51	Use variables in task names	When naming tasks, aim to be expressive and informative by incorporating all relevant details. Improve task names with the use of variables to provide additional context and clarity. Including variables to identify the current host the task is executing against adds descriptive value to the task name.
S51	Specify module defaults in tasks	One reason is technical: If file ownership isn't explicitly declared, it defaults to the user executing Ansible, which might not always be desirable. Being explicit helps avoid this issue. The second reason is organizational: When others use your playbook or role, they might not be aware of module defaults or your intentions. Being explicit in tasks reduces confusion and ensures a clear understanding of your playbook's goals
S54 S46	Always mention module state	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 additional states.
S57	Do not ignore failed Tasks	Define the state parameter. In some modules this could be: present, latest, absent, etc. 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	Avoid skipping tasks	Applying conditional statements like "when" in Ansible playbooks can affect idempotency. For instance, changing a boolean variable used to add a cronjob might not remove the cronjob as intended. In more intricate situations, a service set to be disabled by default may persist until manually enabled by a developer.
S54	Verify service state	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	The command failed, so I used the 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.
Module		
S49	Do not use non-idempotent modules	Shell commands are less likely to be idempotent.

S49 S48 S55 S51		<p>Shell commands are less likely to be idempotent.</p> <p>Shell commands will always run and will always report "changed," unless you're diligent about using <code>changed_when</code>.</p> <p>Many modules are designed to be operating system agnostic, which also helps you write more reusable code.</p> <p>Run commands are what we collectively call the <code>command</code>, <code>shell</code>, <code>raw</code> and <code>script</code> modules that enable users to do</p> <p>command line 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.</p> <p>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.</p> <p>Ansible sets things up for problems down the road.</p> <p>The most important thing to consider is that these run commands have little logic to them and no concept of desired</p> <p>state like a typical Ansible module.</p> <p>That <code>shell</code> that succeeded the first time you ran your play may fail the next time when something already exists.</p> <p>That's unless you ignore <code>_errors</code> on that task. But how do you catch a real error like wrong permissions? Now you have to register the result of that first command and follow it with another task that implements conditional logic to check if an error occurred in the first and handle it.</p> <p>This one should be obvious, but for people that come from a classic admin-background and are new to Ansible it</p>
Configuration Data		
S49 S55	Configuration file template	<p>Use templates</p> <p>Try to avoid using hard coded variables and use Templates</p>
Source	Bad Practice	Practice Summary
S28	Data as code	<p>Configuration Data has a different lifecycle. It's more dynamic.</p> <p>Example 1: use your provisioning tool to define organization users.</p> <p>Example 2: manifest that lists all your 500 servers</p>
S28	Fancy configuration file copying	<p>To configure package X, you keep all configuration files it needs within your "code".</p> <p>You use provisioning tool abstractions to copy every single file onto the target system</p>
S28	Not treating IaC as Code	<p>Version Control is essential for managing and collaborating on code effectively.</p> <p>Inexperience with new tools may necessitate more Code Reviews to ensure code quality.</p> <p>Static Code Analysis tools can be applied to IaC products for quality assessment.</p> <p>While Unit Testing may not be ideal for IaC, Integration Testing is significant for verifying system behavior.</p> <p>Incorporating the mentioned techniques provides optimal QA results for any code.</p>
S31 S6	Non reproducible environments	<p>After a machine is created using an IaC workflow, it should not undergo manual interventions or external updates.</p> <p>All maintenance should be automated, aligned with the IaC process, and compliant with established standards.</p> <p>Making manual or external updates, including security patching, can lead to configuration drifting, which may eventually cause significant non-compliance or service failures.</p>
Ansible Bad Practices		
Source		Practice Summary
S65	Overuse of comments	<p>Overuse of comments Ansible is declarative for a reason. Your code should document itself. Tasks should have descriptive names that explain what is</p>
		<p>- Avoid overusing comments in tasks, as it can result in congested code that becomes challenging to maintain during changes.</p> <p>- Excessive comments can also encourage others to add more comments, leading to important information being buried in the code.</p> <p>- Instead, focus on keeping the README updated and ensuring task names are clear and descriptive to maintain code readability and organization.</p>
Playbook		
S47	Single one-size-fits-all playbook	<p>The recommendation against a single playbook also extends to more focused playbooks targeting specific tasks.</p> <p>To expedite deployments, avoid running unrelated tasks together in a single playbook.</p> <p>For instance, when making minor changes like button color, there's no need to execute complex tasks like SSH key distribution or application deployment.</p> <p>Organize your playbook into sections that match your stack architecture for improved speed and simplicity during deployments.</p>
S50 S48 S47	Including business logic in the playbooks	<p>Prioritize roles for increased code reuse and efficiency, as reused code is well-tested and reliable.</p> <p>Tasks-only playbooks are useful for solving transient issues without code reusability.</p> <p>Avoid mixing tasks directly in playbooks with roles to maintain a higher abstraction level and improve readability.</p> <p>Thinking in terms of roles allows quick understanding of playbooks without getting lost in the details.</p> <p>Well-tested roles facilitate easy comprehension of infrastructure, but adding tasks can undermine this clarity.</p> <p>Stick to roles for managing logic and desired state as Ansible playbooks are not intended for extensive coding.</p>
S65 S54	Non Idempotent roles/modules	<p>Make idempotency a key goal while writing roles to ensure consistent and predictable behavior.</p> <p>Some modules, like <code>Command</code>, can cause issues with idempotency, leading to tasks being marked as changed unnecessarily.</p> <p>For instance, when enabling an optional repository, the task might always show as changed, even if the repository was already enabled.</p>
S54	Restart services without using a handler Chaining handlers	<p>Do not restart services without using a handler. Service restarts should always be handled.</p> <p>Do not chain handlers! If you do, tasks may fail if a previous handler fails</p>
Role		
S54	task specific modules	<p>Using general modules <code>shell</code>,</p> <p>When you develop playbooks and roles, avoid using the following: <code>shell</code>, <code>command</code>, <code>raw</code>, and <code>script</code> instead <code>command</code>, <code>raw</code>, and <code>script</code>.</p> <p>Other modules should be used instead If you can't avoid using one of these modules, test what you are executing and ensure that it is idempotent.</p> <p>If you are using <code>shell</code> tasks as a handler, ensure that the task calling the handler comes from a module that is idempotent.</p>