| Sources | Best Practices | Practice Summary /Extracts |
|---|---|---|
| S22<br>S09<br>S08<br>S14<br>S69<br>S70 S85-90 | Automation | 1. Ensure all infrastructure specifications are clearly coded within configuration files, establishing them as the primary source of truth.<br><br>2. Enable seamless infrastructure deployment by relying solely on the information present in the configuration files, avoiding the need for manual adjustments whenever possible.<br><br>3. Emphasize the importance of codifying all aspects of our infrastructure to promote reproducibility and minimize human error risks associated with manual configuration. |
| S14<br>S29<br>S22<br>S4<br>S6<br>S7<br>S13<br>S61<br>S19<br>S23<br>S26<br>S27<br>S28<br>S32<br>S33<br>S8<br>S69<br>S70 | Version Control | - Utilize a version-controlled database change management tool for configuration files.<br>- Enable an audit trail, collaboration, and testing of IaC code through code reviews.<br>- Implement code branching and merging best practices for efficient management of IaC updates.<br>- Track all infrastructure changes and enable easy rollback if needed.<br>- Require infrastructure modifications to go through Git repository changes and PR reviews.<br>- Maintain code, documentation, test cases, and scripts in a central repository for consistency.<br>- Easily define and clone configurations to ensure current and consistent documentation. |
| S22<br>S18<br>S8<br>S28 | Modularity | To simplify management, divide the infrastructure into separate modules or stacks and automate their combination during deployment.<br><br>Set access controls to regulate who can modify specific parts of the infrastructure code, accommodating different teams and individuals.<br><br>Enforce configuration discovery to maintain stability and control the number of configuration changes.<br><br>Adopt a microservices-oriented approach, aligning infrastructure configurations with individual microservices' needs.<br><br>Promote modularization to reduce complexity and eliminate duplication of key logic.<br><br>Design IaC with flexible, reusable blocks that can be assembled on-demand for quicker adaptation to changing requirements. |
| S22<br>S7<br>S20<br>S28<br>S9 | Document the Code | The source code itself serves as comprehensive documentation, eliminating the necessity for extensive additional instructions for users.<br>IaC code acts as self-documentation, reducing discrepancies between infrastructure and written guidance.<br>Written documentation is not a priority since the code continually reflects the current infrastructure state, providing up-to-date documentation.<br>While diagrams and setup instructions aid knowledge sharing, the focus remains on the code as the most accurate and reliable documentation source. |
| S32 | Distributed Repository | Utilizing a unified repository for the entire stack.<br>Aligning application releases with infrastructure or configuration modifications for synchronized deployments.<br>Facilitating the provision of temporary ad-hoc infrastructure during the deployment process.<br>Allowing seamless integration of application releases with essential infrastructure configuration changes. |

| | | |
|---|---|---|
| S28 | Gradual Configuration | Rather than replicating default package states, concentrate on specifying incremental modifications in your code. |
| S28<br>S69<br>S70<br>S71 | Configuration Data Source | Opt for a suitable data storage (like a database) to house configuration data, especially when handling numerous items.<br>Avoid hardcoding values in the IaC to ensure easier adaptability and maintenance.<br>Make the IaC parameterized to enable dynamic configuration based on specific requirements.<br>Use absolute paths instead of relative paths in the IaC for precise and dependable file references. |
| S28<br>S69<br>S70 S85-90 | Reproducible Image | Antipattern:Creating server images manually without proper documentation or understanding of modifications.<br>Pattern: Embracing reproducible images based on established operating system distributions (*.iso).<br>Leveraging base provider images to build upon the infrastructure.<br>Utilizing Packer for generating images compatible with multiple virtualization software and cloud providers.<br>Utilizing Docker to build and distribute containers as portable and shareable images. |
| S21<br>S31<br>S22<br>S10 | Immutable Infrastructure | Software updates and fixes involve deploying new servers from modified base images, while removing the old servers.<br>Advantageantages:<br>Immutable infrastructure simplifies maintenance by eliminating patching and in-place upgrades, reducing corner cases and inconsistencies in server deployments.<br>Configuration drift and one-off instances are avoided, leading to a more consistent infrastructure.<br>Security is improved as administrative ports like SSH and RDP are not kept open on servers.<br>The risk of unexpected impacts due to undocumented changes in the stack's configuration is minimized. |
| S19<br>S28 S85-90 | Environment Template | Develop templates catering to various infrastructure levels like staging and production, enabling the creation of multiple instances from the same template.<br>Construct detailed templates offering complete working environments with scalability, isolation, and adaptability features.<br>Ensure clear and specific specifications for infrastructure needs, including network bandwidth and storage I/O operations per second, to avoid any omissions. |
| S28 | Package Application for Deployment | Prepare your application in the optimal format for hassle-free deployment.<br>Share the packaged application through an artifact repository, such as Maven, RubyGems, Yum, or Apt.<br>The artifact repository serves as a buffer, providing isolation between pipelines and simplifying integration.<br>This approach helps reduce the complexity of code needed in later stages of configuration management. |
| S28<br>S69<br>S70 | Infrastructure Query Language | Use a language or API to query real-time or latest available reports on your infrastructure state.<br><br>Collect data from IaC systems to keep track of the current status of the infrastructure.<br><br>Employ Declarative Language and tools such as Puppet, Terraform, or Pulumi for infrastructure management.<br><br>Be mindful of potential reproducibility problems caused by the current configuration and machine states. |
| S28 | Secret Isolation | Secrets should be injected on the very last stage of "deploying" your code.<br>Secrets should not be in code |
| S28 | Encrypted Secret | all stored secrets must be encrypted<br>Decryption password is shared through a different channel. |

| | | |
|---|---|---|
| S28<br>S7<br>S60 | Collaborate | 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<br>code reviews are a must for those embracing the notion of treating their infrastructure as code. |
| S28 | Metrics as Code | Pattern: Metrics as Code Metrics that your application provides evolve with your application. New components, new endpoints, new KPIs...<br><br>Keep monitoring configuration close to the code! Or make it auto discoverable and visible!<br>Configuring and collecting metrics Monitoring software has configuration files and/or an API that can be programmed. There a plenty of libraries that allow |
| S69<br>S70<br>S71 S99-110 | Dependency Declaration | Dependency pinning → use correct format of the declaration<br>Use a file to declare your minimum needed configurations/libraries<br>Use standard libraries to maximize portability.<br>Check for system-specific dependencies (e.g., operating system, hardware architecture). |

**Ansible Best Practices**

| **Source** | | **Practice Summary** |
|---|---|---|
| **Project** | | |
| S46<br>S51<br>S57 | Use the standard Ansible project structure | |
| S46 | Use semantic versioning | |
| S46<br>S48 | Use native YAML syntax and conventions | |
| S55<br>S52<br>S54<br>S50 | Use native YAML syntax and conventions | |
| S50<br>S57 | Minimal Consistent quoting | Decide on a uniform quoting style, either double quotes or single quotes, and use it consistently throughout the code.<br>Keep in mind that certain situations, like using variables or octal numbers, may require explicit quoting to ensure proper functionality. |
| S55<br>S53<br>S51 | Vaults for Storing Secrets | As you gain experience with Ansible, more tasks can be automated.<br>Eventually, sensitive information like SSL configurations and database passwords will need automation.<br>Ansible offers the Vault module for securely storing and automating such data without risks.<br>Vault allows encryption of important information for safe storage in version control systems or other environments.<br>Storing passwords or certificates in plain text within repositories is not recommended, but ansible-vault enables encryption of sensitive data.<br>The playbook has examples of both encrypted and commented-out plain text files.<br>Decrypting files requires the vault password, which must be stored in the root directory but not committed to git repositories.<br>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<br>S69<br>S70 | Parameterized roles<br>Parametrizing scripts | **List required parameters before optional parameters** |
| S46<br>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.<br>Leveraging roles is an effective method to fully utilize Ansible's modular capabilities. |
| S50<br>S51 | Use role documentation templates | When documenting Ansible roles, use the template provided by ansible-galaxy init.<br>Include a description of the role's purpose and function, along with usage examples.<br>List and explain the variables used in the role, preferably in the form of a table with variable name, default value, and explanation.<br>Specify the dependencies required for the role to work correctly.<br>Mention the role's author and provide information about the role's license. |
| S50<br>S54<br>S57<br>S69<br>S71 | Test Roles with an emulated environment | In a CI model, ensure unit testing is performed for each role.<br>Containers are a convenient choice for testing roles across multiple distributions.<br>For low-level actions, such as bootloader setting and firewall configurations, use virtual machines for testing.<br>Conduct thorough and consistent testing on various platforms to validate the role's functionality. |
| S49<br>S55<br>S54<br>S51 | Use Ansible Galaxy to find and share Roles | |
| **Task** | | |
| S48<br>S50<br>S55<br>S52<br>S51<br>S57<br>S46 | Name tasks | While you can leave out the 'name' field for tasks, it's Advantageised to include a description of the task's purpose.<br>The 'name' field is visible during playbook execution.<br>Always name plays and tasks with descriptive and human-readable information to improve communication with users when they run the playbook.<br>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.<br>Improve task names with the use of variables to provide additional context and clarity.<br>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.<br><br>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<br>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.<br><br>Define the state parameter. In some modules this could be: present, latest, absent, etc. |
| S57 | Do not ignore failed Tasks | 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 S112-119 | Avoid skipping tasks | Applying conditional statements like "when" in Ansible playbooks can affect idempotency.<br>For instance, changing a boolean variable used to add a cronjob might not remove the cronjob as intended.<br>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.<br><br>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.<br>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<br>would not do that for manual commands,<br><br>you also should not use become: yes globally for a whole playbook. Better<br>only use it for tasks that actually need root rights. The become flag can be assigned to task blocks, avoiding repetition.<br><br>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** | | |
| S112-119 S49 | Do not use non-idempotent modules | Shell commands are less likely to be idempotent. |
| S49<br>S48<br>S55<br>S51 | | Shell commands are less likely to be idempotent.<br><br>Shell commands will always run and will always report “changed,” unless you’re diligent<br>about using changed_when .<br>Many modules are designed to be operating system agnostic, which also helps you<br>write more reusable code.<br>Run commands are what we collectively call the command , shell , raw and script modules that enable users to do<br><br>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.<br><br>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.<br><br>Ansible and sets things up for problems down the road.<br>The most important thing to consider is that these run commands have little logic to them and no concept of desired<br><br>state like a typical Ansible module.<br>That shell that succeeded the first time you ran your play may fail the next time when something already exists.<br>That’s unless you ignore_errors on that task. But how do you catch a real error like wrong permissions?<br>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.<br><br>This one should be obvious, but for people that come from a classic admin-background and are new to Ansible it |

| | | |
|---|---|---|
| **Configuration Data** | | |
| S49<br>S55 | Configuration file template | Use templates<br>Try to avoid using hard coded variables and use Templates |

| **Source** | **Bad Practice** | **Practice Summary** |
|---|---|---|
| S28 | Data as code | Configuration Data has a different lifecycle. It's more dynamic.<br>Example 1: use your provisioning tool to define organization users.<br>Example 2: manifest that lists all your 500 servers |
| S28 | Fancy configuration file copying | To configure package X, you keep all configuration files it needs within your "code".<br>You use provisioning tool abstractions to copy every single file onto the target system |
| S28 | Not treating IaC as Code | Version Control is essential for managing and collaborating on code effectively.<br>Inexperience with new tools may necessitate more Code Reviews to ensure code quality.<br>Static Code Analysis tools can be applied to IaC products for quality assessment.<br>While Unit Testing may not be ideal for IaC, Integration Testing is significant for verifying system behavior.<br>Incorporating the mentioned techniques provides optimal QA results for any code. |
| S31<br>S6 | Non reproducible environments | After a machine is created using an IaC workflow, it should not undergo manual interventions or external updates.<br>All maintenance should be automated, aligned with the IaC process, and compliant with established standards.<br>Making manual or external updates, including security patching, can lead to configuration drifting, which may eventually cause significant non-compliance or service failures. |

| **Ansible Bad Practices** | | |
|---|---|---|
| **Source** | | **Practice Summary** |
| S65 | Overuse of comments | Overuse of comments Ansible is declarative for a reason. Your code should document itself. Tasks should have descriptive names that explain what is |
| | | - Avoid overusing comments in tasks, as it can result in congested code that becomes challenging to maintain during changes.<br>- Excessive comments can also encourage others to add more comments, leading to important information being buried in the code.<br>- Instead, focus on keeping the README updated and ensuring task names are clear and descriptive to maintain code readability and organization. |
| **Playbook** | | |
| S47 | Single one-size-fits-all playbook | The recommendation against a single playbook also extends to more focused playbooks targeting specific tasks.<br>To expedite deployments, avoid running unrelated tasks together in a single playbook.<br>For instance, when making minor changes like button color, there's no need to execute complex tasks like SSH key distribution or application deployment.<br>Organize your playbook into sections that match your stack architecture for improved speed and simplicity during deployments. |

| | | |
|---|---|---|
| S50<br>S48<br>S47 | Including business logic in the playbooks | Prioritize roles for increased code reuse and efficiency, as reused code is well-tested and reliable.<br>Tasks-only playbooks are useful for solving transient issues without code reusability.<br>Avoid mixing tasks directly in playbooks with roles to maintain a higher abstraction level and improve readability.<br>Thinking in terms of roles allows quick understanding of playbooks without getting lost in the details.<br>Well-tested roles facilitate easy comprehension of infrastructure, but adding tasks can undermine this clarity.<br>Stick to roles for managing logic and desired state as Ansible playbooks are not intended for extensive coding. |
| S65<br>S54 S112-119 | Non Idempotent roles/modules | Make idempotency a key goal while writing roles to ensure consistent and predictable behavior.<br>Some modules, like Command, can cause issues with idempotency, leading to tasks being marked as changed unnecessarily.<br>For instance, when enabling an optional repository, the task might always show as changed, even if the repository was already enabled. |
| S54 | Restart services without using a handler<br>Chaining handlers | Do not restart services without using a handler. Service restarts should always be handled.<br><br>Do not chain handlers! If you do, tasks may fail if a previous handler fails Module |
| **Role** | | |
| S54 S112-119 | task specific modules | Using general modules shell,<br><br>When you develop playbooks and roles, avoid using the following: shell , command, raw, and script instead command , raw , and script .<br><br>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.<br><br>If you are using shell tasks as a handler, ensure that the task calling the handler comes from a module that is idempotent. |