Quali | Torque

Torque Blueprint Automation
Best Practices Guide

Ansible | Terraform | Shell Grains

Quali Torque Documentation

Torque Blueprint Automation
Best Practices Guide

Ansible | Terraform | Shell Grains

Quali Torque Documentation

Table of Contents

1. Introduction and Guiding Principles

This guide establishes the standard practices for building Torque blueprints that orchestrate Ansible, Terraform, and Shell grains. The goal is to create a consistent, reusable, and testable automation library that scales from individual developer environments to production deployments.

Ansible-First Philosophy:

This guide treats Ansible as the default grain type for both infrastructure provisioning and configuration management. Terraform is recommended for cloud-native stateful resources (AWS, Azure, GCP) where its state tracking provides clear advantages, but Ansible is fully capable of provisioning infrastructure through vendor APIs and device modules.

Core Principles:

- Ansible is the primary tool for infrastructure managed through vendor APIs and device modules. Policies, profiles, device configuration, and service management should all be driven by Ansible playbooks.
- Ansible grains support full lifecycle management through the on-destroy section, making them suitable for provisioning resources that need cleanup on environment teardown.
- Terraform is recommended for cloud-native stateful objects (VMs, storage, networking in AWS/Azure/GCP) where drift detection and plan/apply workflows add value.
- Shell grains handle utility tasks: validation, glue scripts, health checks, API calls, and lightweight automation.
- Blueprint inventory-file sections provide dynamic, per-grain inventory that eliminates static inventory files.
- Lower-level (leaf) grains expose verbose inputs for maximum flexibility. Higher-level (composite) blueprints abstract complexity with fewer, opinionated inputs.
- Every grain should include default inputs suitable for local/CI testing without requiring manual parameter entry.

2. Repository Structure Standard

Each Torque-enabled repository should follow a predictable folder layout that separates assets by grain type and organizes them by logical grain name.

2.1 Master Repo Layout

```
automation/
├── blueprints/                    # Torque discovers blueprints here
│   ├── full-stack-app.yaml
│   ├── simple-web-app.yaml
│   ├── database-cluster.yaml
│   └── health-check-only.yaml
├── layouts/                       # Torque layout YAMLs
│   └── default-layout.yaml
├── ansible/                       # Ansible playbooks (primary)
│   ├── provision-app-server/      # Grain: provision an app server
│   │   ├── playbook.yaml
│   │   ├── teardown.yaml
│   │   └── requirements.yaml
│   ├── configure-web-server/      # Grain: configure nginx/apache
│   │   ├── playbook.yaml
│   │   ├── requirements.yaml
│   │   └── roles/
│   │       ├── nginx-setup/
│   │       └── monitoring-agent/
│   ├── deploy-application/        # Grain: deploy app code
│   │   ├── playbook.yaml
│   │   └── teardown.yaml
│   └── health-check/
│       └── playbook.yaml
├── terraform/                     # Terraform modules
│   ├── aws-vpc/
│   ├── aws-ec2-instance/
│   └── rds-database/
├── shell/                         # Shell scripts (utilities)
│   ├── validate-connectivity/
│   ├── wait-for-ready/
│   └── export-report/
└── README.md
```

2.2 Key Structural Rules

Rule	Rationale
One folder per grain under each technology directory	Ensures Torque auto-discovery works correctly and each grain is independently referenceable
Blueprints live only in /blueprints	Torque requires this exact folder name (case-sensitive) with .yaml extension

Layouts in /layouts	Torque scans this specific folder for layout definitions
requirements.yaml alongside each Ansible playbook	Torque auto-installs Galaxy dependencies from the module root
teardown.yaml for on-destroy lifecycle	Referenced by the on-destroy section for lifecycle cleanup
terraform.tfvars.example in each TF module	Documents default values for testing without the blueprint
README.md in every grain folder	Self-documentation for new team members

2.3 Using External Repos as Sources

Existing Ansible collections or shared playbook repositories can be connected to Torque as separate repository stores. Blueprints reference them via the store name, avoiding duplication.

```
# In the blueprint, reference an external repo:
source:
  store: shared-ansible      # Name given when onboarding the repo
  path: playbooks/deploy-app.yaml
```

> **Tip:** Onboard shared repos in Torque under Settings > Repositories. All blueprints can then reference their playbooks directly.

3. Grain Design Philosophy: Layered Granularity

Grains should be designed in two tiers to balance flexibility with ease of use. This layered approach lets power users access every knob while giving most consumers a simple interface.

3.1 Low-Level (Leaf) Grains

These are the building blocks. They map closely to a single Ansible playbook, a single Terraform module, or a single shell script. They expose all possible inputs so they can be composed into different higher-level blueprints.

- **Ansible leaf grain (provisioning):** provision-app-server: accepts app_name, instance_type, region, vpc_id, security_groups, ssh_key, tags, and more.
- **Ansible leaf grain (configuration):** configure-web-server: accepts http_port, ssl_enabled, max_connections, log_level, custom_config_path.
- **Terraform leaf grain:** aws-ec2-instance: exposes instance_type, ami_id, subnet_id, key_name, security_group_ids, tags.
- **Shell leaf grain:** validate-connectivity: accepts target_host, port, timeout_seconds, retry_count, protocol.

3.2 High-Level (Composite) Blueprints

These compose multiple leaf grains into an end-to-end workflow. They expose only the inputs the consumer cares about and wire internal defaults for the rest.

- **Example:** simple-web-app: exposes only app_name, environment (dev/staging/prod), and size (small/medium/large). Internally it maps 'large' to a powerful instance type, high memory, and the correct load balancer config.

Aspect	Leaf Grain	Composite Blueprint
Input count	Many (10-30+)	Few (3-7)
Defaults	All inputs have test defaults	Maps user choices to leaf defaults
Audience	Platform engineers, grain authors	Developers, QA, self-service users
Reusability	High: used across many blueprints	Medium: purpose-built stacks
Testing	Unit-tested independently	Integration-tested as a stack

4. Ansible Grain Best Practices

Ansible is the primary automation grain. It handles both infrastructure provisioning through vendor APIs and post-provision configuration management. This section covers the foundational patterns.

4.1 Playbook Structure per Grain

```
ansible/configure-web-server/
├── playbook.yaml           # Main playbook entry point
├── requirements.yaml       # Galaxy dependencies
└── roles/
    ├── nginx-setup/
    │   └── tasks/main.yaml
    └── monitoring-agent/
        └── tasks/main.yaml
```

4.2 Requirements File

Torque auto-detects a requirements.yaml in the same directory as the playbook and installs the listed collections before execution.

```
# requirements.yaml
collections:
  - name: community.general
  - name: torque.collections
```

4.3 Exporting Outputs from Ansible

Use Torque's built-in export_torque_outputs module to pass results from an Ansible grain to downstream grains or blueprint outputs. The task running this module must target localhost.

```
- name: Export outputs to Torque
  torque.collections.export_torque_outputs:
    outputs:
      app_url: "{{ deploy_result.url }}"
      app_version: "{{ version }}"
      server_ip: "{{ ansible_host }}"
  delegate_to: localhost
  run_once: true
  tags: always
```

4.4 Grain Inputs

Inputs provided to Ansible grains are passed as extra-vars via a JSON file. Torque creates the file at /var/run/ansible/inputs/inputs.json and runs the playbook with --extra-vars pointing to it. Your playbook variables can directly reference the input names.

```
# Blueprint grain section:
inputs:
  - app_name: '{{ .inputs.app_name }}'
  - http_port: '{{ .inputs.http_port }}'
  - environment: '{{ .inputs.environment }}'

# In the playbook, reference directly:
- name: Deploy application
  template:
    src: app.conf.j2
    dest: /etc/{{ app_name }}/config.yml
  vars:
    listen_port: "{{ http_port }}"
```

> **Tip:** Install the torque.collections module locally for testing: ansible-galaxy collection install torque.collections

5. Ansible for Infrastructure Provisioning and Lifecycle

Ansible is not limited to configuration management. With vendor-specific collections, Ansible can provision infrastructure, create cloud resources, configure network devices, and manage services. When combined with the Torque Ansible grain's on-destroy section, Ansible grains support complete lifecycle management: create on deploy, remove on teardown.

5.1 The on-destroy Section

The on-destroy section is a key feature of the Ansible grain that enables full lifecycle control. It mirrors the main grain structure and specifies a separate playbook that runs when the Torque environment is terminated.

Why on-destroy matters for Ansible provisioning:

- Without on-destroy, resources created by Ansible (cloud objects, DNS records, service registrations) would remain after the environment ends, causing drift and sprawl.
- The on-destroy playbook uses state: absent, method: DELETE, or equivalent cleanup operations.
- The on-destroy section supports its own source, inputs, command-arguments, inventory-file, and pre-ansible-run scripts.

5.2 Provisioning Pattern with Lifecycle

```
grains:
  provision_service:
    kind: ansible
    spec:
      source:
        store: automation-repo
        path: ansible/provision-app-server/playbook.yaml
      agent:
        name: '{{ .inputs.agent }}'
      inventory-file:
        localhost:
          hosts:
            127.0.0.1:
              ansible_connection: local
      inputs:
        - app_name: '{{ .inputs.app_name }}-{{ envId | downcase }}'
        - region: '{{ .inputs.region }}'
        - instance_type: '{{ .inputs.size }}'
        - api_key: '{{ .params.cloud_api_key }}'
      outputs:
```

```
      - server_id
      - server_ip

  # Lifecycle: clean up on environment teardown
  on-destroy:
    source:
      store: automation-repo
      path: ansible/provision-app-server/teardown.yaml
    inputs:
      - server_id: '{{ .grains.provision_service.outputs.server_id }}'
      - api_key: '{{ .params.cloud_api_key }}'
    inventory-file:
      localhost:
        hosts:
          127.0.0.1:
            ansible_connection: local
```

5.3 When to Use Ansible vs. Terraform for Provisioning

Scenario	Recommended Grain
Vendor-managed infrastructure with Ansible collections	Ansible (vendor-specific collection)
Network device configuration (routers, switches, firewalls)	Ansible (network modules)
AWS/Azure/GCP infrastructure (VPCs, VMs, storage)	Terraform (native provider support, state tracking)
Hybrid: cloud infra + device config	Terraform for cloud + Ansible for device config
Quick API calls or REST operations	Ansible (uri module) or Shell

Guidance: If the resource has an Ansible collection from the vendor, Ansible is a natural fit. If the resource lives in a public cloud provider and benefits from state-based drift detection, prefer Terraform. Both can coexist in the same blueprint.

6. Dynamic Inventory Management with Blueprints

One of the most powerful features of Torque Ansible grains is the ability to define inventory directly in the blueprint YAML. This eliminates static inventory files and enables each grain to target different hosts with different credentials, all driven by blueprint inputs, parameter store values, and outputs from other grains.

6.1 How Dynamic Inventory Works

The inventory-file section of an Ansible grain accepts the same YAML structure as a standard Ansible inventory file. Torque generates the actual inventory file at runtime. Every field supports Torque's templating engine.

Key advantages:

- No static inventory files to maintain or keep in sync across environments.
- Host addresses can come directly from outputs of upstream grains.
- Credentials and connection parameters can be pulled from the Torque parameter store.
- Each grain in a blueprint can define a completely different inventory.
- The same playbook can be reused across many blueprints with different inventories.

6.2 Per-Grain Inventory Pattern

```
# Grain 1: Create resources via API (runs locally)
provision_infra:
  kind: ansible
  spec:
    inventory-file:
      localhost:
        hosts:
          127.0.0.1:
            ansible_connection: local
        vars:
          api_key: '{{ .params.cloud_api_key }}'
          api_secret: '{{ .params.cloud_api_secret }}'

# Grain 2: Configure servers (SSH to provisioned hosts)
configure_app:
  depends-on: provision_infra
  kind: ansible
  spec:
    inventory-file:
      app_servers:
        hosts:
          server1:
            ansible_host: '{{ .grains.provision_infra.outputs.server_ip }}'
```

```
        vars:
          ansible_user: '{{ .inputs.ssh_user }}'
          ansible_become: true
          http_port: '{{ .inputs.http_port }}'
```

6.3 Combining Static and Dynamic Hosts

```
inventory-file:
  all:
    children:
      new_servers:
        hosts:
          new_vm:
            ansible_host: '{{ .grains.infra.outputs.vm_ip }}'
      existing_infra:
        hosts:
          jumpbox:
            ansible_host: 10.0.1.100
    vars:
      ansible_user: '{{ .inputs.ssh_user }}'
      ansible_become: true
```

Important: A single playbook can be reused across many blueprints, each providing different server lists through the inventory-file. The playbook never changes; only the inventory changes per environment.

7. Terraform Grain Best Practices

Terraform grains are recommended for cloud-native stateful resources where drift detection and plan/apply workflows add value.

7.1 Module Structure

```
terraform/aws-ec2-instance/
├── main.tf              # Resource definitions
├── variables.tf         # All input variables with defaults
├── outputs.tf           # All outputs (IPs, IDs, names)
├── versions.tf          # Required providers and versions
└── terraform.tfvars.example
```

7.2 Variables and Outputs Conventions

- Every variable must have a description, a type constraint, and a default value where sensible.
- Use snake_case for all variable and output names.
- Prefix related variables: vm_name, vm_cpu_count, vm_memory_mb.
- Output every piece of information another grain might need.

7.3 State Backend Configuration

Always configure a remote backend in production blueprints. Torque auto-generates unique state file names per environment, appending {environmentId}_{grainName}.tfstate to the key-prefix.

```
grains:
  cloud_infra:
    kind: terraform
    spec:
      source:
        store: automation-repo
        path: terraform/aws-ec2-instance
      backend:
        type: "s3"
        bucket: "my-torque-tf-state"
        region: "us-east-1"
        key-prefix: "cloud/ec2"
```

8. Shell Grain Best Practices

Shell grains are the utility layer. They handle tasks that do not warrant a full Ansible playbook: connectivity checks, quick API calls, report generation, data transformations, and environment validation.

8.1 Script Organization

- Keep scripts small and single-purpose. One script per grain folder.
- Always use the files section to reference scripts from the repo rather than inlining long commands.
- Remember that each command line runs in its own shell. Use semicolons or && to chain commands, or call a script file.
- The deploy activity is mandatory even if you only need destroy. Use 'echo NOTHING-TO-DO' as a placeholder.

8.2 Capturing Outputs

Shell outputs are captured by exporting environment variables. Name the command and declare expected outputs.

```
grains:
  validate_env:
    depends-on: configure_app
    kind: shell
    spec:
      agent:
        name: '{{ .inputs.agent }}'
      files:
        - source: automation-repo
          path: shell/validate-connectivity/check_ssh.sh
      activities:
        deploy:
          commands:
            - name: connectivity_check
              command: >-
                source check_ssh.sh
                '{{ .grains.provision_vm.outputs.vm_ip }}'
                22 30 5
              outputs:
                - connectivity_status
                - response_time_ms
```

9. Input Templating and Defaults Strategy

Torque uses a Liquid-compatible templating engine. Blueprint inputs flow into grain inputs through template expressions. A well-designed input layer is critical for both usability and testability.

9.1 Blueprint-Level Inputs

```
inputs:
  agent:
    type: agent
  environment:
    type: string
    default: dev
    description: Target environment (dev, staging, prod)
  app_name:
    type: string
    default: my-app
  region:
    type: string
    default: us-east-1
    description: Cloud region for deployment
  ssh_user:
    type: string
    default: ubuntu
```

9.2 Using Torque Parameter Store

Sensitive values like API keys, passwords, and private keys must be stored in Torque's parameter store and referenced using the .params template syntax. Never hardcode credentials in blueprints or playbooks.

```
inputs:
  - api_key: '{{ .params.cloud_api_key }}'
  - db_password: '{{ .params.database_password }}'
  - ssh_key: '{{ .params.ssh_private_key }}'
```

9.3 Default Inputs for Testing

Every leaf grain should work out of the box with its defaults. This enables CI pipelines to deploy and test grains without manual parameter entry.

- Blueprint defaults should point to a lab/dev environment.
- Parameter store should have a set of CI-safe credentials scoped to the test space.
- Every input should have a default that produces a valid but safe deployment.

10. Outputs and Inter-Grain Dependencies

Grains communicate through the depends-on directive and the outputs/inputs templating chain.

10.1 Dependency Rules

- Use depends-on at the grain level (sibling of kind: and spec:) to declare execution order.
- A grain can depend on multiple grains: depends-on: grain_a, grain_b
- Only reference outputs from grains listed in depends-on.
- Ansible outputs require export_torque_outputs. Terraform outputs are auto-detected from outputs.tf. Shell outputs use exported environment variables.

10.2 Output Reference Syntax

Grain Type	Output Reference Pattern
Terraform	{{ .grains.<grain_name>.outputs.<output_name> }}
Ansible	{{ .grains.<grain_name>.outputs.<output_name> }}
Shell	{{ .grains.<n>.activities.deploy.commands.<cmd>.outputs.<out> }}

10.3 Blueprint-Level Outputs

```
outputs:
  app_url:
    value: '{{ .grains.deploy_app.outputs.app_url }}'
  server_ip:
    value: '{{ .grains.provision_vm.outputs.vm_ip }}'
  dashboard:
    kind: link
    value: 'https://dashboard.example.com/env/{{ envId }}'
```

11. Blueprint Examples

11.1 Ansible-Only Blueprint: API-Driven Service Provisioning

This blueprint uses three Ansible grains to create cloud resources via API calls, configure the service, and register it with a service mesh. Each grain has its own inventory. The provisioning grain includes on-destroy for cleanup.

```
spec_version: 2
description: >
  API-driven service provisioning using Ansible only.
  Creates resources, configures the service, and registers
  with service mesh. Cleans up via on-destroy.

inputs:
  agent:
    type: agent
  app_name:
    type: string
    default: my-service
  region:
    type: string
    default: us-east-1
  instance_type:
    type: string
    default: small
  ssh_user:
    type: string
    default: ubuntu

outputs:
  server_ip:
    value: '{{ .grains.provision.outputs.server_ip }}'
  app_url:
    value: '{{ .grains.configure.outputs.app_url }}'
  mesh_status:
    value: '{{ .grains.register.outputs.registration_status }}'

grains:

  # GRAIN 1: Provision via cloud API (localhost)
  provision:
    kind: ansible
    spec:
      source:
        store: automation-repo
        path: ansible/provision-app-server/playbook.yaml
      agent:
```

```
      name: '{{ .inputs.agent }}'
    inventory-file:
      localhost:
        hosts:
          127.0.0.1:
            ansible_connection: local
        vars:
          api_key: '{{ .params.cloud_api_key }}'
          api_secret: '{{ .params.cloud_api_secret }}'
    inputs:
      - app_name: '{{ .inputs.app_name }}-{{ envId | downcase }}'
      - region: '{{ .inputs.region }}'
      - instance_type: '{{ .inputs.instance_type }}'
    outputs:
      - server_ip
      - server_id
    on-destroy:
      source:
        store: automation-repo
        path: ansible/provision-app-server/teardown.yaml
      inputs:
        - server_id: '{{ .grains.provision.outputs.server_id }}'
        - api_key: '{{ .params.cloud_api_key }}'
        - api_secret: '{{ .params.cloud_api_secret }}'
      inventory-file:
        localhost:
          hosts:
            127.0.0.1:
              ansible_connection: local

# GRAIN 2: Configure server (SSH to provisioned host)
configure:
  depends-on: provision
  kind: ansible
  spec:
    source:
      store: automation-repo
      path: ansible/configure-web-server/playbook.yaml
    agent:
      name: '{{ .inputs.agent }}'
    inventory-file:
      app_servers:
        hosts:
          target:
            ansible_host: '{{ .grains.provision.outputs.server_ip }}'
            ansible_user: '{{ .inputs.ssh_user }}'
        vars:
          ansible_become: true
          app_name: '{{ .inputs.app_name }}'
    inputs:
      - app_name: '{{ .inputs.app_name }}'
      - environment: dev
    outputs:
```

```
      - app_url

# GRAIN 3: Register with service mesh (localhost)
register:
  depends-on: configure
  kind: ansible
  spec:
    source:
      store: automation-repo
      path: ansible/deploy-application/playbook.yaml
    agent:
      name: '{{ .inputs.agent }}'
    inventory-file:
      localhost:
        hosts:
          127.0.0.1:
            ansible_connection: local
    inputs:
      - app_name: '{{ .inputs.app_name }}'
      - server_ip: '{{ .grains.provision.outputs.server_ip }}'
    outputs:
      - registration_status
```

11.2 Two-Grain Blueprint: Terraform + Ansible

This example provisions cloud infrastructure with Terraform and configures it with Ansible. Each grain has a distinct inventory. The Ansible grain depends on Terraform and consumes its outputs.

```
spec_version: 2
description: >
  Provision infrastructure with Terraform and configure it
  with Ansible. Demonstrates dependency and per-grain inventory.

inputs:
  agent:
    type: agent
  app_name:
    type: string
    default: web-app
  instance_type:
    type: string
    default: t3.medium
  ssh_user:
    type: string
    default: ubuntu
  http_port:
    type: string
    default: '8080'

outputs:
  vm_ip:
    value: '{{ .grains.provision_vm.outputs.public_ip }}'
  app_url:
    value: '{{ .grains.configure_app.outputs.app_url }}'

grains:

  # GRAIN 1: Terraform creates the cloud VM
  provision_vm:
    kind: terraform
    spec:
      source:
        store: automation-repo
        path: terraform/aws-ec2-instance
      agent:
        name: '{{ .inputs.agent }}'
      backend:
        type: "s3"
        bucket: "my-torque-tf-state"
        region: "us-east-1"
        key-prefix: "cloud/vms"
      inputs:
```

```
      - instance_name: '{{ .inputs.app_name }}-{{ envId | downcase }}'
      - instance_type: '{{ .inputs.instance_type }}'
    outputs:
      - public_ip
      - instance_id

# GRAIN 2: Ansible configures the new VM
configure_app:
  depends-on: provision_vm
  kind: ansible
  spec:
    source:
      store: automation-repo
      path: ansible/configure-web-server/playbook.yaml
    agent:
      name: '{{ .inputs.agent }}'
    inventory-file:
      web_servers:
        hosts:
          server1:
            ansible_host: '{{ .grains.provision_vm.outputs.public_ip }}'
            ansible_user: '{{ .inputs.ssh_user }}'
        vars:
          ansible_become: true
          http_port: '{{ .inputs.http_port }}'
          app_name: '{{ .inputs.app_name }}'
    inputs:
      - app_name: '{{ .inputs.app_name }}'
      - http_port: '{{ .inputs.http_port }}'
    outputs:
      - app_url
```

12. Naming Conventions and Tagging

Consistent naming prevents confusion across grains and blueprints.

Element	Convention	Example
Blueprint file	kebab-case.yaml	full-stack-app.yaml
Grain name	snake_case	provision_vm, configure_app
Ansible role	kebab-case folder	nginx-setup, monitoring-agent
Ansible variable	snake_case	app_name, http_port
TF variable	snake_case	instance_type, subnet_id
TF output	snake_case	public_ip, instance_id
Shell script	kebab-case.sh/.py	check-ssh.sh, gen-report.py
Torque input	snake_case	environment, app_name
Param store key	snake_case	cloud_api_key
Repo store name	kebab-case	automation-repo

12.1 Resource Tagging Strategy

Tag all resources created through cloud providers or APIs with environment metadata:

```
# In Terraform modules:
locals {
  common_tags = {
    managed_by  = "torque"
    environment = var.environment
    app         = var.app_name
  }
}

# In Ansible playbooks:
tags:
  managed_by: torque
  environment: "{{ environment }}"
  app: "{{ app_name }}"
```

13. Testing and CI/CD Recommendations

13.1 Testing Strategy

Layer	Tool	What to Test
Ansible	ansible-lint + molecule	Playbook syntax, role unit tests
Terraform	terraform validate + plan	Syntax, variable types, plan drift
Shell	shellcheck + bats-core	Script linting, function testing
Blueprint	Torque dry-run / preview	Full grain wiring, dependency resolution

13.2 Default Inputs for CI

Every leaf grain should be independently testable with its default inputs. A blueprint with only defaults should be deployable to a CI/dev environment. Store CI-specific parameters in the Torque parameter store under a test scope.

13.3 Git Branching and Blueprint Testing

- Torque supports connecting repository branches. Use a develop branch for testing before merging to main.
- Each blueprint should be validated in a Torque space connected to the test branch before promotion.
- Use Torque's Environment as Code (EaC) for automated regression: define environment YAMLs in /environments.

13.4 Quick Reference Checklist

- Every Ansible grain: has requirements.yaml, exports outputs via torque.collections, uses on-destroy for provisioning grains
- Every Ansible grain: leverages dynamic inventory-file from the blueprint (no static inventory files)
- Every Terraform module: has variables.tf with defaults, outputs.tf, versions.tf
- Every Shell script: has executable permissions, captures outputs via export, handles errors
- Every Blueprint: has description, default inputs, outputs section, depends-on chain
- Sensitive values: stored in parameter store, referenced via {{ .params.* }}
- Naming: snake_case for code identifiers, kebab-case for files and repos