

DevOps with Pulumi

K19G

2026-10-31

Table of contents

DevOps with Pulumi	4
What this book covers	4
How the content is organized	4
Formats	5
Prerequisites	5
Acknowledgments	5
 Intro	 6
 Overview	 7
 Overview	 8
Purpose of this part	8
Example callout	8
Example code block	8
Next steps	8
 Foundations	 9
 Getting Started	 10
 Getting Started	 11
Suggested structure for a chapter	11
Worked example	11
Checklist	11
Images	11
 Projects, Stacks and State	 12
 Projects, Stacks and State	 13
Learning goals	13
Projects: the program and its <code>Pulumi.yaml</code>	14
Stacks: many instances of one program	14
Names	15
Renaming has consequences	15
Removing	15
Outputs: what a stack publishes	15
Stack references: wiring projects together	16

State and backends	17
Drift and refresh	17
Raw state	18
Diagnosis	18
Operational care	18
Practice	19
Recall card	19
Sources	19
Configuration and Secrets	20
Configuration and Secrets	21
Learning goals	21
Configuration: per-stack key/value pairs	21
Keys and namespaces	22
Reading from code	22
Structured values	22
Project-level defaults	23
Provider configuration	23
Secrets	24
Setting	24
Secrets are outputs that remember	24
Where secret-ness ends	25
Secrets providers	25
Diagnosis	26
Operational care	26
Practice	27
Recall card	27
Sources	27
Inputs, Outputs and the Dependency Graph	28
Inputs, Outputs and the Dependency Graph	29
Learning goals	29
Inputs	29
Outputs	30
Getting at the value	30
apply : transform one output	31
Output.all : combine several	31
Helpers: skip the lambda	31
Lifting: reach into nested outputs	32
Two rules that prevent most bugs	32
Resource identity	32
Diagnosis	32
Practice	33
Recall card	33
Sources	33

DevOps with Pulumi

i Note

Replace this preface after you copy the template. Rename the book folder (e.g. `books/My-Topic`); the folder name becomes the Quarto book title when you run `scripts/update-index.sh`.

What this book covers

Summarize the audience, goals, and scope in a few bullets:

- Who this is for
- What you will be able to do
- What is deliberately out of scope

How the content is organized

Source lives under `content/`. Each first-level directory is a **part** in the sidebar:

Folder	Role
<code>content/00-intro/</code>	Orientation and overview
<code>content/01-foundations/</code>	Core material
<code>content/99-appendices/</code>	Optional extras (add when needed)
<code>content/_planning/</code>	Notes only — ignored by the index script

Chapter order is filename order. Prefer numeric prefixes:

```
content/01-foundations/01-getting-started.qmd
content/01-foundations/02-core-ideas.qmd
```

Set sidebar titles with YAML frontmatter:

```
---
title: "Getting Started"
---
```

Formats

This book is published as part of the monorepo library portal:

- **HTML** — primary reading experience (light/dark themes)
- **PDF / EPUB** — generated on full library builds

Prerequisites

List tools, background knowledge, or environment setup here.

Acknowledgments

Add credits here.

Intro

Overview

Overview

This is a sample chapter. Delete or rewrite it after you copy the template.

Purpose of this part

The 00-intro part is a good place for:

- Motivation and learning outcomes
- How to use the book
- Conventions used in later chapters

Example callout



Tip

Prefer a `YAML title:` in frontmatter so `scripts/update-index.sh` can pick a clean sidebar label without parsing the first heading.

Example code block

```
# From the book directory after copy:
bash scripts/update-index.sh
quarto preview
```

Next steps

1. Skim `content/01-foundations/01-getting-started.qmd`
2. Replace sample chapters with real content
3. Re-run `scripts/update-index.sh` whenever you add, remove, or rename files

Foundations

Getting Started

Getting Started

First real content chapter. Use this part for foundational material.

Suggested structure for a chapter

1. Short motivation
2. Core explanation
3. Worked example
4. Exercises or checklist

Worked example

```
# Good chapter file names (lexicographic order = sidebar order)
01-getting-started.qmd
02-core-concepts.qmd
03-practice.qmd

# Optional intermediate insert without renumbering everything
02a-quick-reference.qmd
```

Checklist

- ☐ Folder name uses a numeric prefix (01-foundations)
- ☐ Chapter file has a `title:` in frontmatter
- ☐ Ran `bash scripts/update-index.sh` after structural edits
- ☐ Previewed with `quarto preview` or `./indipub.sh <BookName>` from `books/`

Images

Put assets under `images/` at the book root and reference them relatively:

```
! [Diagram] (../../images/example.png)
```

(Adjust the relative path based on chapter depth.)

Projects, Stacks and State

Projects, Stacks and State

Every Pulumi command you run answers three questions before it touches a cloud API: *which program* (the project), *which copy of it* (the stack), and *where is the record of what already exists* (the backend and its state). Get those three straight and the rest of Pulumi is “write a program that declares resources”. Get them wrong and you deploy staging’s code into production’s state.

i What was and was not run

The authoring box had no `pulumi` CLI and the run rules forbid installs, so none of the commands below were executed. Command lines, file names and behaviour follow the current Pulumi documentation. The CLI listings are reproduced from that documentation, not captured locally, and are marked as such. Examples use Python. Every other Pulumi language has an equivalent call, listed in the docs.

Learning goals

- Read a `Pulumi.yaml` and say which runtime and entrypoint it selects
- Create, select, rename and remove stacks, and explain fully qualified stack names
- Export stack outputs and consume them from another project with a `StackReference`
- Choose between Pulumi Cloud and a DIY backend, and know what `pulumi refresh` does and does not do

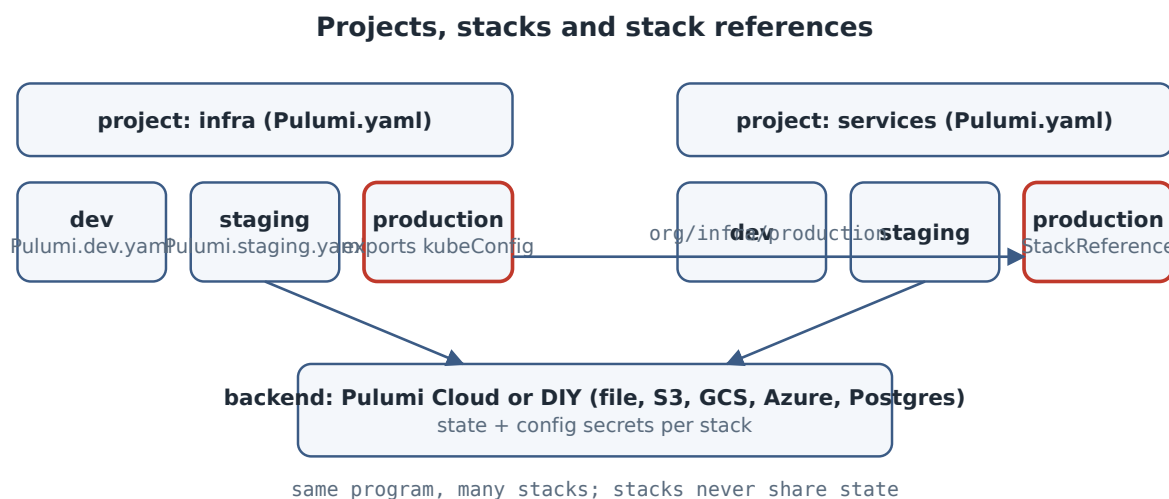


Figure 1: Two projects, three stacks each, linked by stack references; every stack has its own state in the backend.

Projects: the program and its Pulumi.yaml

A **project** is any directory containing a `Pulumi.yaml`. At run time Pulumi walks up from the current directory to the nearest one, and that file decides the project. `pulumi new` creates it.

```
name: webserver
runtime: python
description: A minimal Pulumi program.
```

- `name` is the project name. It also namespaces configuration keys (next chapter).
- `runtime` is one of `nodejs`, `python`, `go`, `dotnet`, `java`, `yaml` or `hcl`.
- Each runtime finds its entrypoint by its own convention: `__main__.py` (or `setup.py`) for Python, `package.json` for Node, the build tool for Go, .NET and Java, `.tf` files for HCL.

The file name must start with a capital P. Either `.yaml` or `.yml` works.

For compiled languages you can skip the build at deploy time by pointing at a prebuilt binary:

```
name: my-project
runtime:
  name: go
  options:
    binary: mybinary
```

Paths. File paths inside a program are relative to the *working directory*. If the program must find files next to `Pulumi.yaml` regardless of where it was launched, use `pulumi.get_root_directory()` and convert the result to a path relative to the working directory, so plans don't differ between machines.

Stacks: many instances of one program

A **stack** is an isolated, independently configured instance of the project's program, typically `dev`, `staging`, `production`, or one per feature branch. Each stack has its own configuration and its own state. The program is shared.

```
pulumi stack init staging    # create and select
pulumi stack ls             # list stacks of the nearest project
pulumi stack select dev     # change the active stack
pulumi stack                # show the active stack's resources and outputs
```

The commands `config`, `preview`, `up` and `destroy` act on the **active** stack. Pass `--stack` (or `-s`) in scripts and pipelines rather than relying on whatever was selected last.

Names

Stack names may contain letters, digits, hyphens, underscores and periods, and must be unique within a project. A stack can be written three ways:

Form	Meaning
<code>dev</code>	Current user or default organization, nearest project
<code>acme/dev</code>	Organization <code>acme</code> , nearest project
<code>acme/webserver/dev</code>	Fully qualified: organization, project, stack

On a DIY backend the organization part is always the literal `organization`. Backends initialised by a CLI older than v3.61.0 only understand the short form until upgraded with `pulumi state upgrade`.

Renaming has consequences

`pulumi stack rename production` changes the value of `pulumi.get_stack()`. If your program uses the stack name in resource names, the next `pulumi up` proposes **replacements**. Read the preview. Sometimes creating a new stack is the safer operation.

A rename across projects (`acme/old-project/prod` to `acme/new-project/prod`) needs two manual follow-ups: change `name` in `Pulumi.yaml`, and rename project-namespaced keys in `Pulumi.<stack>.yaml` (for example `old-project:database` to `new-project:database`). Skip the second step and the program silently sees those values as unset.

Removing

`pulumi destroy` deletes the stack's resources. `pulumi stack rm` then deletes the empty stack, its history and its `Pulumi.<stack>.yaml`. `pulumi stack rm --force` removes a stack that still has resources, orphaning them. Do not.

Outputs: what a stack publishes

A stack exports values from the top level of the program:

```
import pulumi

pulumi.export("x", "hello")
pulumi.export("o", {"num": 42})
```

From the documentation (not run here), reading them back looks like:

```
$ pulumi stack output x
hello
```

```
$ pulumi stack output --json
{
  "x": "hello",
  "o": {
    "num": 42
  }
}
```

Outputs are effectively JSON-serialised and resolved after the update finishes. Export IDs and names, not whole resource objects, which serialise to large blobs. Secret outputs display as `[secret]` unless you pass `--show-secrets`.

Stack references: wiring projects together

Large systems split into projects by lifecycle and ownership, for example `infra` (network, cluster) and `services` (apps deployed onto it). The pairing is by environment: `acme/infra/production` feeds `acme/services/production`. A `StackReference` reads another stack's outputs:

```
import pulumi
from pulumi import StackReference

env = pulumi.get_stack()
infra = StackReference(f"acme/infra/{env}")

kubeconfig = infra.require_output("kubeConfig")
```

Use the fully qualified name. Pulumi records the dependency between stacks. Choose the reader carefully:

Method	Behaviour	Use
<code>require_output</code>	Fails the deployment if the output is missing	Default choice; turns a typo into an error
<code>get_output</code>	Returns an output wrapping <code>None</code> if missing	Only when absence is expected and handled
<code>get_output_details</code>	Awaitable; returns <code>.value</code> or <code>.secret_value</code> directly	When program logic needs the raw value. Python needs an async entrypoint and SDK 3.254.0 or later

The referenced stack's outputs are its public interface. Renaming an output is a breaking change for every consumer, just like renaming a function.

State and backends

Each stack's **state** is Pulumi's record of the resources it manages. **preview** and **up** diff the program against that record, not against the live cloud. The state lives in a **backend**:

- **Pulumi Cloud** (the default, also self-hostable): `pulumi login` with no argument. Updates go through a transactional API with full history.
- **DIY**: object storage you run (S3 or compatible such as MinIO or Ceph, Azure Blob, GCS), a PostgreSQL database, or the local filesystem. `pulumi login <backend-url>`, for example `pulumi login file://~` for local files.

Avoid typing the URL every time by setting `PULUMI_BACKEND_URL` or a `backend.url` in `Pulumi.yaml`. `pulumi whoami -v` shows the user and the backend in use. Credentials for backends are stored in `~/.pulumi/credentials.json`. Your *cloud* credentials are never stored in state. They stay where the CLI runs.

With a DIY backend you take on access control, encryption at rest, backups and auditing yourself. Checkpoint history lives under `.pulumi/history/`, but blob storage is not transactional, so some partial failures cannot be recovered automatically the way Pulumi Cloud recovers them.

Moving a stack between backends is `pulumi stack migrate` (CLI v3.254.0 or later). Log in to the target and name the source:

```
pulumi login                                # target: Pulumi Cloud
pulumi stack migrate file://~ my-app-production
```

It copies the latest checkpoint and configuration, re-encrypts secrets with the target's secrets provider, and leaves the source untouched.

Drift and refresh

Pulumi does not query every resource on every run. That is a deliberate choice for speed, predictability and explicit control. A console edit is invisible until you run:

```
pulumi refresh                             # update state from the cloud; changes no infrastructure
pulumi preview --refresh                    # refresh, then show what up would do
pulumi up --refresh                         # refresh, then converge to the program
```

After a refresh, the next **up** reverts out-of-band changes unless you also change the program to keep them.

Raw state

`pulumi stack export --file stack.json` and `pulumi stack import --file stack.json` give you the JSON checkpoint for inspection or emergency repair. An incorrect import can orphan resources. Export first, keep the copy, and prefer the `pulumi state` subcommands for targeted edits.

Diagnosis

Symptom	Likely cause	Check
<code>up</code> acts on the wrong environment	Active stack was not the one you assumed	<code>pulumi stack ls</code> (the <code>*</code> marks the active one); pass <code>--stack</code>
CLI says no project (<code>Pulumi.yaml</code>) was found	Running outside the project tree	<code>ls Pulumi.yaml</code> in the directory or a parent
Preview proposes replacing everything after a rename	Stack name used in resource names	Revert the rename, or accept replacement deliberately
Config values silently missing after rename StackReference fails	Project-namespaced keys still carry the old name Wrong qualified name, missing permission, or output not exported	Inspect <code>Pulumi.<stack>.yaml</code> <code>pulumi stack output -s org/project/stack</code>
Stacks “disappeared” Plan ignores a console change	Logged in to a different backend No refresh	<code>pulumi whoami -v</code> <code>pulumi preview --refresh</code>

Operational care

- One project per lifecycle and owner. One stack per environment. Do not encode environments as `if` statements on the stack name for anything bigger than sizes and counts.
- Commit `Pulumi.yaml` and the `Pulumi.<stack>.yaml` files of long-lived stacks (secrets in them are encrypted). Ephemeral stacks usually do not commit theirs.
- In CI, always pass `--stack` with the fully qualified name, and run `preview` in pull requests and `up` on merge.
- Tag stacks (`pulumi stack tag set environment production`) so Pulumi Cloud can group them. Avoid the reserved prefixes `pulumi:`, `gitHub:` and `vcs:`.
- On DIY backends: version the bucket, restrict who can write, and back it up. That bucket *is* your inventory.

Practice

1. Sketch the project and stack layout for a system with a shared network, a database, and three services, across dev and production. How many stacks, and which references?
2. Write the `StackReference` line a `services` program needs to read `vpcId` from the matching `infra` stack, and justify `require_output` over `get_output`.
3. List what you must do yourself if you choose an S3 DIY backend instead of Pulumi Cloud.

Recall card

- Project = directory with `Pulumi.yaml` (name, runtime). Stack = one configured instance with its own state.
- `org/project/stack` is the full name. DIY backends use the literal `organization`.
- `pulumi.export` publishes; `StackReference.require_output` consumes.
- Backend = Pulumi Cloud or DIY. `refresh` syncs state from reality, explicitly. `stack migrate` moves between backends.

Sources

- Pulumi, [Projects](#)
- Pulumi, [Stacks](#)
- Pulumi, [State and backends](#)

Configuration and Secrets

Configuration and Secrets

A stack differs from its siblings in exactly two places: its state and its **configuration**. Configuration is how one program becomes a `t3.micro` in dev and a `m7i.large` in production without a single `if`. Some of that configuration is a password, and Pulumi's **secrets** system makes sure the password is encrypted in every place it travels to, not just the file you typed it into.

i What was and was not run

`pulumi` was not available on the authoring box and installs were not allowed, so no commands here were executed. Commands, file layouts and listings follow the current Pulumi documentation. Listings such as `[secret]` masking are reproduced from the docs and marked as such. Examples use Python.

Learning goals

- Set, read and structure stack configuration, and explain key namespaces
- Choose between `get`, `require` and the `*_secret` variants, and between stack-level and project-level config
- Explain how secret-ness propagates through outputs, and where it does not (resource IDs, `apply` callbacks)
- Pick and change a secrets provider, including for DIY backends

Configuration: per-stack key/value pairs

Values live in `Pulumi.<stack>.yaml` next to `Pulumi.yaml`, and you **commit** that file. The values drive the program, so they belong in version control alongside it.

```
pulumi config set aws:region us-west-2      # provider namespace
pulumi config set instanceCount 3           # project namespace (implicit)
pulumi config get instanceCount
pulumi config                               # list; --json for machines
cat id_ed25519.pub | pulumi config set publicKey # stdin: no shell escaping
```

`config set` overwrites an existing value **without warning**. Review the diff of `Pulumi.<stack>.yaml` before committing.

Keys and namespaces

Keys are `[namespace:]name`. With no colon, the namespace is the project name. That is how `aws:region` and your own `region` coexist. In the stack file the result looks like:

```
config:
  aws:region: us-west-2
  webserver:instanceCount: "3"
```

In code, `pulumi.Config()` reads the project namespace and `pulumi.Config("aws")` reads another one. Reusable components should read their *own* namespace (`pulumi.Config("mylib")`) so they never collide with the consuming project.

Reading from code

```
import pulumi

config = pulumi.Config()
name = config.require("name")           # missing -> deployment stops with an explanatory
count = config.get_int("instanceCount") or 1 # missing -> None, so default it
db_password = config.require_secret("dbPassword") # Output, marked secret

aws_region = pulumi.Config("aws").require("region")
```

Use `require` for anything without a sensible default. A typo in a key should fail the preview, not quietly create one instance in production. Configuration is **read-only** at run time. Writing config from code is the job of the Automation API.

Environment variables also reach the program (`os.environ`), but configuration is per-stack, versioned and visible to collaborators, so prefer it.

Structured values

`--path` treats the key as a path into an object:

```
pulumi config set --path 'data.active' true
pulumi config set --path 'data.nums[0]' 1
pulumi config set --path 'data.nums[1]' 2
```

```
config:
  proj:data:
    active: true      # booleans stay booleans
    nums: [1, 2]      # integer-like values become integers
```

```
data = pulumi.Config().require_object("data")
active = data["active"]           # a plain dict, not a Config
```

`require_object` returns a plain dictionary. Index into it. You cannot call `require` on a nested value.

Project-level defaults

Values every stack shares go in `Pulumi.yaml`, and any stack can override them:

```
# Pulumi.yaml
config:
  aws:region: us-east-1
  name: BroomeLLC
  data:
    value:           # structured project-level values need this wrapper
    active: true

# Pulumi.dev.yaml: overrides for one stack
config:
  aws:region: us-east-2
  myproject:name: MopLLC
```

Watch the two syntaxes. The stack file uses `project:key` with structures nested directly. The project file uses bare `key` and wraps structures in `value:`. Project-level config is edited by hand (`config set` doesn't write it) and holds **plaintext only**.

The project file can also **type-check** stack values:

```
config:
  name:
    type: string
    description: Base name for resources.
    default: BroomeLLC
  subnets:
    type: array
    items:
      type: string
```

A stack that sets `name` to a number then fails at `pulumi preview`. Type specifications don't yet cover structured config.

Provider configuration

A provider takes settings from three places: `pulumi config set aws:region ...`, its environment variables, or constructor arguments. Only the **default** provider reads stack config. An explicit

`aws.Provider(...)` you create reads nothing from config, so pass its arguments explicitly. To forbid implicit default providers altogether:

```
config:
  pulumi:disable-default-providers: ["aws", "kubernetes"] # or ["*"]
```

`pulumi:tags` in config applies stack tags on every `up` or `refresh`. Removing a key from config does *not* delete the tag from the stack.

Secrets

Setting

```
pulumi config set --secret dbPassword 'S3cr37'
```

The stack file now holds ciphertext:

```
config:
  myproject:dbPassword:
    secure: AAABAI1lW0ewSuZ1FJxw/+Rpw6BNqTUvGJ3008WkpL2hB4aPyS7UU68=
```

Quote values containing `$`, `!` or `#`, or pipe them from a file, so the shell doesn't change them first. Secrets work inside structured config too: `pulumi config set --path --secret 'endpoints[0].token' ....`

The inverse: the CLI tries to detect values that *look* like tokens and stops you. When it guesses wrong, use `--plaintext`.

As documented (not run here), listings and accidental prints are masked:

```
$ pulumi config
KEY                                VALUE
aws:region                        us-west-1
dbPassword                        [secret]
```

Secrets are outputs that remember

A secret is an `Output` with a flag. Pulumi encrypts it wherever it is persisted, and **propagates** the flag. Anything computed from a secret with `apply` or `Output.all` is also secret. Resource properties built from it are encrypted in state.

```
import pulumi
import pulumi_aws as aws

cfg = pulumi.Config()
```

```
param = aws.ssm.Parameter(
    "a-secret-param",
    type="SecureString",
    value=cfg.require_secret("my-secret-value"),    # encrypted in state too
)
```

Ways to make a value secret:

Source	How
Config	<code>get_secret</code> / <code>require_secret</code>
An existing value	<code>pulumi.Output.secret(value)</code>
A resource's outputs	<code>ResourceOptions(additional_secret_outputs=[</code>
Derived values	Automatic via <code>apply</code> / <code>Output.all</code> on a secret

Reading a `--secret` key with plain `require` hands you the plaintext and **drops** the marking downstream. Always use the secret getters.

Where secret-ness ends

- **Inside `apply`.** The callback receives plaintext. Pulumi marks the *return value* secret, but cannot stop the callback from printing or writing it to a file.
- **Resource IDs.** A resource's `id` is always stored in plaintext, and `additional_secret_outputs=["id"]` does nothing. `random.RandomString` exposes its `result` as its ID too, so its value lands unencrypted in state. Use `random.RandomPassword` and mark `result` secret.
- **Outside the stack.** Secret-ness covers one stack's state. Sharing the same credential across stacks or teams is a job for Pulumi ESC environments, which a stack imports with an `environment:` list in its stack file. Explicit stack config wins over imported values.

Secrets providers

By default with Pulumi Cloud, each stack gets its own Cloud-managed key and each value a nonce. Choose another provider when you use a DIY backend or your organisation mandates its own KMS:

```
pulumi stack init prod --secrets-provider="awskms://alias/pulumi-prod?region=us-east-1"
pulumi stack init prod --secrets-provider="gcpkms://projects/P/locations/L/keyRings/R/crypto
pulumi stack init prod --secrets-provider="azurekeyvault://myvault.vault.azure.net/keys/pulu
pulumi stack init prod --secrets-provider="hashivault://pulumi-prod"
pulumi stack init prod --secrets-provider=passphrase
```

- `awskms` accepts an ID, alias or ARN, an optional `profile=`, and (CLI v3.41.1+) `context_<k>=<v>` encryption context you can condition IAM policies on.
- `hashivault` reads `VAULT_SERVER_URL` and `VAULT_SERVER_TOKEN`.

- **passphrase** derives the key from a passphrase you supply (typically through `PULUMI_CONFIG_PASSPHRASE`). Lose it and the stack’s secrets are unrecoverable.

The provider’s public bits (**encryptionsalt** for passphrase, **encryptedkey** for KMS-style providers) are written into `Pulumi.<stack>.yaml` and are safe to commit. To switch an existing stack:

```
pulumi stack change-secrets-provider "awskms://alias/pulumi-prod?region=us-east-1"
pulumi preview      # expect: no changes
```

This re-encrypts config and state. If you’d rather not commit stack files at all, `pulumi config refresh` rebuilds them from the last deployment.

Diagnosis

Symptom	Likely cause	Fix
“Missing required configuration variable”	Key not set for <i>this</i> stack, or wrong namespace	<code>pulumi config -s <stack>;</code> check the <code>project:</code> prefix
Value set, program says unset	Set under a different stack, or project renamed	Confirm the active stack; rename keys in the stack file
Password visible in <code>stack export</code>	Read with require instead of require_secret , or carried in an <code>id</code>	Use the secret getters; avoid ID-bearing resources
CLI refuses a harmless value as “looks like a secret” “passphrase must be set” or <code>decrypt</code> fails in CI	Secret detection heuristic Passphrase or KMS access missing in the runner	<code>--plaintext</code> Provide <code>PULUMI_CONFIG_PASSPHRASE</code> or KMS credentials
Preview fails with a type error	Project-level type spec	Fix the stack value’s type

Operational care

- Commit `Pulumi.yaml` and `Pulumi.<stack>.yaml`. Ciphertext is meant to be committed. Never commit plaintext you meant to mark secret, because git history keeps it.
- Treat `pulumi config set` in pipelines as a code change. Prefer reviewing the stack file in a PR to mutating config during deploys.
- Use one KMS key or ESC environment per trust boundary (prod versus non-prod), and restrict decrypt rights accordingly.

- Rotate by setting the new value with `--secret` and running `up`. Changing the *provider* is a separate operation (`change-secrets-provider`).

Practice

1. Design config for a service with region, instance count, a list of allowed CIDRs, and a database password. Which keys go project-level, which stack-level, and which are secret?
2. A colleague creates a token with `random.RandomString` and marks `result` secret. What is still exposed, and what do you change?
3. Your team moves from Pulumi Cloud to an S3 DIY backend. What happens to secrets, and which provider would you pick?

Recall card

- `Pulumi.<stack>.yaml` holds per-stack config (commit it). `Pulumi.yaml config:` holds shared, plaintext, typed defaults.
- No-colon keys belong to the project. `Config("ns")` reads other namespaces.
- `require` fails loudly, `get` returns `None`, and `*_secret` keeps the marking.
- Secret-ness propagates through outputs, except inside `apply` callbacks and resource IDs.
- Use `--secrets-provider` at `stack init` and `stack change-secrets-provider` later. A passphrase is a key you must not lose.

Sources

- Pulumi, [Configuration](#)
- Pulumi, [Secrets](#)
- Pulumi, [State and backends](#)

Inputs, Outputs and the Dependency Graph

Inputs, Outputs and the Dependency Graph

A Pulumi program looks imperative: Python statements, run top to bottom. The infrastructure it describes is declarative: Pulumi decides what to create, in what order, and what to leave alone. The bridge between the two is a pair of types. **Inputs** are what you give a resource. **Outputs** are values the cloud reports back later. Every confusing moment for new Pulumi users (“why does my bucket name print as an object?”) comes from treating an output as if it were already a value.

i What was and was not run

pulumi and the provider SDKs were not available on the authoring box and installs were not allowed, so these Python snippets were not executed. They follow the current Pulumi documentation and SDK method names.

Learning goals

- Distinguish inputs, resource outputs and stack outputs
- Explain how passing an output into an input creates a dependency edge
- Transform outputs with `apply`, `Output.all`, `Output.format`, `Output.concat` and the JSON helpers, and reach nested values by lifting
- Recognise the “output used as a plain value” bug, and why resources must not be created inside `apply`

Inputs

An input is an argument to a resource, usually one parameter of the underlying cloud API call. Every input of type `Input[T]` also accepts a plain `T`:

```
import pulumi_random as random

my_id = random.RandomId("mine", byte_length=8) # 8 is a plain value used as an input
```

Python offers two spellings for object-shaped inputs: an `...Args` class or a dictionary typed as `...ArgsDict`. Both carry the same fields, and the dictionary form is shorter:

```
import pulumi_aws as aws

repo = aws.ecr.Repository(
    "repo",
```

```

        image_tag_mutability="MUTABLE",
        image_scanning_configuration={"scan_on_push": True},
    )

```

Outputs

An output is a property the provider reports **after** the resource exists, such as an ID, an ARN, an endpoint or a generated password. Outputs behave like futures. During a first `pulumi up` the value is unknown until the cloud answers. During `pulumi preview` it may never be known.

Pass an output straight into another resource’s input and Pulumi records a dependency:

```

import pulumi_aws as aws
import pulumi_random as random

password = random.RandomPassword("password", length=16, special=True)

db = aws.rds.Instance(
    "db",
    instance_class="db.t3.micro",
    allocated_storage=64,
    engine="mysql",
    username="someone",
    password=password.result,      # output -> input: an edge in the graph
)

```

That edge tells Pulumi to create the password before the database, and on **destroy** to delete the database before the password. You wrote no ordering code. A program that wires outputs to inputs gets correct ordering for free, and `depends_on` is only for dependencies that no value carries.

Three similar names refer to different things:

Term	What it is
Input	An argument to a resource; accepts plain values or outputs
Resource output	A property of a resource, known once the provider reports it
Stack output	A value you <code>pulumi.export</code> at the end of a run for the CLI or a <code>StackReference</code>

Getting at the value

Because an output is not yet a value, you can’t use ordinary language operations on it. `print(bucket.id)` or `"https://" + lb.dns_name` doesn’t do what you expect. The SDK supplies the tools instead.

apply: transform one output

```
url = lb.dns_name.apply(lambda name: f"https://{name}")
```

The callback runs when the value is known and receives the plain value. Its return value becomes a new `Output`, which you can pass on as an input or export. Secret-ness and dependencies flow through.

Output.all: combine several

```
import pulumi

conn = pulumi.Output.all(server.name, database.name).apply(
    lambda args: f"Server=tcp:{args[0]}.database.windows.net;initial catalog={args[1]};"
)

# keyword form yields a dict
conn = pulumi.Output.all(server=server.name, db=database.name).apply(
    lambda a: f"Server=tcp:{a['server']}.database.windows.net;initial catalog={a['db']};"
)
```

Helpers: skip the lambda

Most `apply` calls build a string or some JSON. The helpers do that directly and are easier to read:

```
import json
import pulumi

url = pulumi.Output.format("https://{0}/api", lb.dns_name)
arn_path = pulumi.Output.concat(bucket.arn, "/*")

policy = pulumi.Output.json_dumps({
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Principal": "*",
        "Action": ["s3:GetObject"],
        "Resource": [arn_path],          # outputs allowed inside the structure
    }],
})

parsed = pulumi.Output.json_loads(some_json_output)
```

`json_dumps` accepts a structure that has outputs nested inside it. A `json.dumps` inside an `apply` would make you resolve each one by hand first.

Lifting: reach into nested outputs

Properties and indexes can be read straight off an output, with no `apply`. Pulumi “lifts” the access:

```
record_name = cert.domain_validation_options[0].resource_record_name
```

Without lifting this would be `cert.domain_validation_options.apply(lambda o: o[0].resource_record_name)`

Two rules that prevent most bugs

1. Never treat an output as a plain value. Printing it, concatenating it with `+`, using it in an `if`, or formatting it with an f-string outside `apply` gives you the wrapper object, not the value. Python’s `Output` warns when converted to a string, and the result ends up in resource names and tags looking like garbage. Use `apply`, `format`, `concat` or `json.dumps`.

2. Don’t create resources inside `apply` or `all`. Pulumi’s documentation says to avoid it wherever possible. Resources created in a callback don’t appear in `pulumi preview` unless the value is already known, so the preview stops matching what up will do. Reviewers then approve a plan that leaves things out. If the *number* of resources depends on a value, read that value from configuration (known at preview), not from another resource’s output.

Resource identity

Wiring resources means passing *some* identifier, and the inputs differ in which one they want. A resource has four identities: its logical name (the first constructor argument), its physical name at the provider, its physical ID (`resource.id`), and its Pulumi URN. Many inputs want an ARN, an endpoint or a name instead of the ID. A type mismatch at preview usually means you passed the wrong identity. Check the input’s documentation.

Diagnosis

Symptom	Cause	Fix
A name or tag contains text about calling <code>__str__</code> on an <code>Output</code>	Output used in an f-string or <code>str()</code>	<code>Output.format/concat</code> , or build it in <code>apply</code>
Preview shows fewer resources than up creates	Resources created inside <code>apply</code>	Drive counts from config. Create resources at top level
Resources created in the wrong order	The dependency is not carried by any value	Pass the output as an input, or <code>depends_on</code> as a last resort

Symptom	Cause	Fix
Type error passing <code>bucket</code> where a string is expected	Passed the resource, not a property	<code>bucket.id</code> , <code>bucket.arn</code> or <code>bucket.bucket</code> , per the input's docs
JSON policy has a placeholder where an ARN belongs	<code>json.dumps</code> on a structure containing outputs	<code>Output.json_dumps</code>

Practice

1. Build an output URL `https://<dns>/health` from a load balancer's `dns_name` three ways: `apply`, `Output.format` and `Output.concat`. Which reads best?
2. A colleague loops `for i in range(int(count_output))`. Explain why this fails and rewrite it to use `config`.
3. Write an S3 bucket policy that references two buckets' ARNs, using only `Output.json_dumps`.

Recall card

- Inputs accept plain values or outputs. Outputs are future values reported by the provider.
- Output to input means a dependency edge. Ordering for create and destroy is automatic.
- `apply` handles one output, `Output.all` several, and `format/concat/json_dumps/json_loads` are the common shortcuts. Lifting reads nested fields.
- Never treat an output as plain, and never create resources inside `apply`.

Sources

- Pulumi, [Inputs and outputs](#)
- Pulumi, [Accessing single output values with apply](#)
- Pulumi, [Accessing multiple output values with all](#)