AWS Architecture Best Practices

Verified 2026-11-09

Build replaceable AWS resources — Well-Architected pillars, naming scopes and patterns, tags, account isolation, drift, and AI-assisted operations.

Contents

Mental model 3

The six pillars 3

Pets versus cattle 5

Name scopes 7

A naming pattern 8

Memorable or generated names 10

Tags 12

Namespace isolation 13

IaC, change sets and drift 15

AI-assisted operations 17

Further reading 19

Mental model

A cloud resource is a disposable output of code, not a machine you look after. If you can delete any resource and get an identical one back from a template, then scaling, recovery, new environments and audits are all the same operation: run the code again. Names, tags and accounts are the interface that people and automation use to find and govern those outputs, so design them as carefully as an API.

The six pillars

Pillar	The question it asks
Operational excellence	Can you change and run it safely?
Security	Who can do what, and can you prove it?
Reliability	Does it recover without you?
Performance efficiency	Are resources the right kind and size?

Pillar	The question it asks
Cost optimization	Do you pay only for what delivers value?
Sustainability	Is the footprint as small as it can be?

The AWS Well-Architected Framework reviews every workload against these six pillars. They trade off against each other, so decide per workload which ones win: a prototype favors cost, a payment system favors reliability and security. The rest of this sheet is the operational groundwork that several pillars share: automate changes, make them small and reversible, and scale horizontally.

```
Operations as code      templates, not consoles
Small, frequent changes easy to review and revert
Scale horizontally      many small, not one big
Test recovery           break it on purpose
Stop guessing capacity  measure, then autoscale
```

> **Tip:** Run a Well-Architected review when a design changes, not once before launch. The framework expects architectures to evolve.

Pets versus cattle

Aspect	Pet	Cattle
Created	By hand, in the console	By code, from a template
Name	Unique and memorable	Systematic or generated
When it fails	You repair it	You replace it
Scaling	Buy a bigger one	Add more of them
Config changes	Applied in place	Shipped as a new version

A pet is a resource someone built by hand and now has to keep alive, because nobody can rebuild it exactly. Cattle are interchangeable: created programmatically,

updated by redeploying, and replaced instead of fixed. Default to cattle. Treat every pet as a risk to document or eliminate.

```yaml
Resources:
  Orders:           # no TableName: generated
    Type: AWS::DynamoDB::Table
    Properties:
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: pk
          AttributeType: S
      KeySchema:
        - AttributeName: pk
          KeyType: HASH
```

> **Gotcha:** CloudFormation cannot run an update that replaces a custom-named resource, because the old and new copies would need the same name at once. Omit the name, or change it in the same update.

Name scopes

Unique across	Examples
All accounts in a partition	S3 general purpose bucket names
All CloudFront distributions	Alternate domain names (CNAMEs)
One account, all Regions	IAM users, groups, roles
One account, one Region	Lambda functions, RDS instances

A name only has to be unique inside its scope, and the scope differs per service. Regional resources can reuse a name in every Region, which keeps multi-Region stacks symmetrical. Global ones cannot: an IAM role named by a template deployed to two Regions collides with itself. Route 53 follows DNS: a hosted zone has the same name as its domain.

```
Resources:
  Logs:
```

```
    Type: AWS::S3::Bucket
    Properties:
      BucketNamespace: account-regional
      BucketNamePrefix: acme-logs
# acme-logs-111122223333-eu-west-1-an
```

> **Warning:** Delete a bucket in the shared global namespace and another account can create the same name and receive requests meant for yours. Empty and keep buckets you still reference, or use the account regional namespace, where only your account can own the name.

A naming pattern

Segment	Example values
Organization or project	`acme`, `shop`
Environment	`dev`, `staging`, `prod`
Workload or service	`orders`, `auth`
Resource role	`api`, `queue`, `db`

Segment	Example values
Region, only if global	`euw1`, `use1`

Pick one order of segments and use it everywhere, so a name can be parsed by a script and guessed by a human. Lowercase letters, digits and hyphens are the only characters every service accepts: S3 rejects uppercase and underscores, and an RDS identifier must start with a letter. Keep names short. S3 buckets and RDS identifiers stop at 63 characters, Lambda function and IAM role names at 64.

```
<project>-<env>-<workload>-<role>[-<region>]

shop-prod-orders-api          Lambda function
shop-prod-orders-db           RDS instance
shop-prod-orders-events       SQS queue
shop-prod-deploy-euw1         IAM role, per Region
```

> **Gotcha:** Never put customer names, secrets or other sensitive data in a name. Bucket names appear in URLs, and names leak into logs, ARNs and error messages.

Memorable or generated names

Use a memorable name for	Use a generated name for
Organization-wide resources, such as the central log bucket	Anything an IaC stack owns
Resources people reference daily	Resources that scale horizontally
Long-lived resources with a stable purpose	Temporary or ephemeral resources
Names other teams hard-code	Shared development environments

Apply the pets-and-cattle test to names. A fixed name is a promise that the resource is a singleton, and it blocks a second copy of the stack in the same account and Region. A generated name such as `MyStack-Orders-abcdefghijk1` lets one template deploy any number of times. Code should receive the generated name as configuration instead of hard-coding it.

```
Outputs:
  OrdersTable:
    Value: !Ref Orders
    Export:
      Name: !Sub "${AWS::StackName}-orders-table"
```

> **Tip:** Put the human meaning in the stack name and tags. Generated names inherit the stack name as a prefix, so `shop-prod-orders` still shows in the console.

Tags

Tag key	Answers
`Project`	Which workload is this?
`Environment`	Is it safe to delete?
`Owner`	Who do I ask?
`CostCenter`	Who pays?
`ManagedBy`	Which tool may change it?

Names are fixed at creation and limited in length; tags can change and carry the rest. A resource takes up to 50 user tags, with keys up to 128 characters and values up to 256. Keys and values are case sensitive, so `CostCenter` and `costcenter` are two different tags. Write the schema down, keep it in version control, and apply tags from IaC at the stack level, not by hand.

```
aws cloudformation deploy \
  --stack-name shop-prod-orders \
  --template-file template.yaml \
  --tags Project=shop Environment=prod \
```

```
    Owner=orders-team CostCenter=1042 \
    ManagedBy=cloudformation
```

> **Gotcha:** A tag does not appear in billing reports until you activate it as a cost allocation tag. A new key can take up to 24 hours to show up for activation, and up to 24 more to activate.

Namespace isolation

Team size	Isolation
One to a few developers	Prefix per developer in one account
Several environments	One account per environment
Beyond 10–15 developers	Account per developer or team

When developers share an account, give each one a namespace: initials or a username in the stack name,

plus an `Owner` tag so strays can be traced and cleaned up. An account is the stronger boundary. Separate accounts give each environment its own security controls, limit the impact of a mistake, split costs, and keep one workload from using up another's service quotas. At scale, AWS Organizations groups accounts into OUs, Service Catalog offers approved templates, and tag policies standardize keys.

```
dev-alice-orders-api       Alice's Lambda function
dev-bob-orders-api         Bob's Lambda function
dev-alice-orders-db        Alice's RDS instance
dev-orders-api             shared dev
staging-orders-api         shared staging
prod-orders-api            production
```

> **Gotcha:** A tag policy in enforcement mode rejects noncompliant tagging requests, which can block autoscaling and provisioning. Start in reporting mode and review the compliance report first.

IaC, change sets and drift

Practice	Why
Manage every stack resource through IaC	A console edit is invisible to the template
Create a change set before updating	See replacements before they happen
Detect drift regularly	Find manual changes
`DeletionPolicy: Retain` on data	Stack deletion keeps the resource

Infrastructure as code is what makes cattle possible: the template is the source of truth and the deployed stack is a copy of it. Drift is any difference between the two, usually an urgent console fix nobody ported back.

Detect it on a schedule, then either update the template or revert the resource.

```
aws cloudformation deploy \
  --stack-name shop-prod-orders \
  --template-file template.yaml \
  --no-execute-changeset     # review first
aws cloudformation detect-stack-drift \
  --stack-name shop-prod-orders
aws cloudformation describe-stack-resource-drifts
\
  --stack-name shop-prod-orders \
  --stack-resource-drift-status-filters \
    MODIFIED DELETED
```

> **Gotcha:** Drift detection only compares properties the template sets explicitly. A property left at its default can be changed by hand without being reported, so set the ones you care about.

AI-assisted operations

The assistant can	Your guardrail
Generate IaC from your naming standard	Standard lives in a rules file
Recommend tags across resources	Tag policy and code review
Surface drift and unused resources	Read-only role for reviews
Fetch current AWS guidance over MCP	Check claims against the docs

AI coding assistants such as Kiro extend the cattle approach past resource creation. Put your naming pattern and tag schema in the project rules file, `.kiro/steering/*.md` for Kiro or `CLAUDE.md` for Claude Code, so every generated template follows them. The AWS MCP Server lets an agent search AWS documentation and call AWS APIs with your IAM credentials, and marks those requests with the

`aws:ViaAWSMCPService` condition key. Use that key in an SCP to stop agents from deleting production data.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Deny",
    "Action": ["rds:Delete*", "s3:DeleteBucket"],
    "Resource": "*",
    "Condition": {
      "Bool": {"aws:ViaAWSMCPService": "true"}
    }
  }]
}
```

> **Warning:** Keep a human approval step on any change that modifies or removes production resources. An agent runs with your permissions, and a confident wrong plan deletes data as fast as a correct one.

Further reading

- AWS Well-Architected Framework
- Organizing Your AWS Environment Using Multiple Accounts
- Best Practices for Tagging AWS Resources
- S3 general purpose bucket naming rules
- CloudFormation best practices
- Agent Toolkit for AWS User Guide