

Google Cloud A Visual Guide

Essential Concepts and Practical Workflows for New Systems Engineers

Reiner Braun

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with Google Cloud, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order on your first pass. The sequence builds the required background step by step: resource organization and responsibility come before identity and networking; those foundations then support compute, data, operations, security, reliability, and automation.

Before reading a chapter in detail, review the **What You'll Be Able to Do** section at the beginning of that chapter. This identifies the practical capability the chapter is intended to build and helps you distinguish the central lesson from supporting detail.

Use the diagrams to understand **relationships, processes, boundaries, and structures** that are difficult to reconstruct from prose alone. Read the surrounding text as well; each diagram supports the explanation rather than replacing it.

Try the code, commands, and hands-on examples yourself whenever you have an appropriate development or training environment. Confirm the active project, account, region, permissions, expected cost, and cleanup method before running a command that creates or changes resources.

Use each chapter's concluding summary and review questions to confirm that you can explain the fundamentals in your own words and apply them to a small practical scenario. Revisit the relevant sections when an answer depends on memorized wording rather than understanding.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as signposts: they show what a concept is for, how it relates to the foundation, and when you may need to study it in greater depth.

Table of Contents

Introduction

How to Use This Book

Part I — Build the Correct Mental Model

Chapter 1 — Google Cloud in One Mental Model

Start with responsibility, not product names

The service-management ladder

Control plane and data plane

Trace one request through the system

Shared responsibility and shared fate

Managed does not mean automatic

Six architectural concerns

A first design for the inventory service

Mistakes that distort the mental model

What to carry forward

Chapter 2 — Resource Hierarchy, Projects, and Locations

Every resource has an administrative address

The organization resource

Folders express administration, not geography

Projects are operational boundaries

Policy inheritance and effective policy

APIs are enabled per project

Billing linkage is a separate relationship

Project design for development and production

Names, labels, and tags solve different problems

Global, regional, zonal, and multi-regional scope

Choose location from requirements

Resource dependencies cross boundaries

Project lifecycle and safe decommissioning

Three checks before touching a resource

Part I — Build the Correct Mental Model

The central idea of Google Cloud engineering is not to memorize products. It is to make a small set of connected decisions correctly: where a resource belongs, which identity may use it, how traffic reaches it, which compute model fits the workload, and who operates each layer. Once those decisions are explicit, individual services become much easier to understand.

This first part of the guide establishes the foundation needed to enter an existing Google Cloud project safely, read its architecture, investigate its configuration, and operate common compute platforms. A small internal inventory service is used throughout. The service receives requests from employees, reads and updates inventory records, and runs scheduled maintenance work. Later design decisions build on this same example rather than introducing unrelated systems.

Chapter 1 — Google Cloud in One Mental Model

Start with responsibility, not product names

Google Cloud is a collection of infrastructure and application services exposed through APIs. An API, or application programming interface, is a defined way for software to request an operation from another system. The Google Cloud console, command-line tools, client libraries, and automation systems all use these APIs, even though their user interfaces look different.

The cloud changes how infrastructure is obtained and operated. Instead of ordering physical servers, installing them in a data center, and waiting for capacity to become available, a team can request resources programmatically. The provider supplies the underlying facilities, hardware, and a range of managed services. The customer still decides how the application should be designed, secured, configured, monitored, and recovered.

This leads to the most useful first question for any service:

Which operational responsibilities does the service remove, and which responsibilities remain with the team?

A service that removes more infrastructure work is often preferable, but only when it still supports the workload's real requirements. Choosing a higher-level service can reduce patching, capacity management, and failure handling. It can also impose constraints on operating systems, networking, protocols, or runtime behavior. The goal is not to choose the service with the most features or the least control. The goal is to choose the **highest-level managed service that fits the workload without forcing an awkward design.**

The service-management ladder

Cloud services can be understood as a ladder. As a team moves upward, Google operates more of the underlying stack and the team focuses more on application behavior. A **virtual machine (VM)** is a software-defined computer with its own guest operating system, while a **container** packages an application and its user-space dependencies as an isolated process. **Kubernetes** is an open-source system for orchestrating containerized workloads.

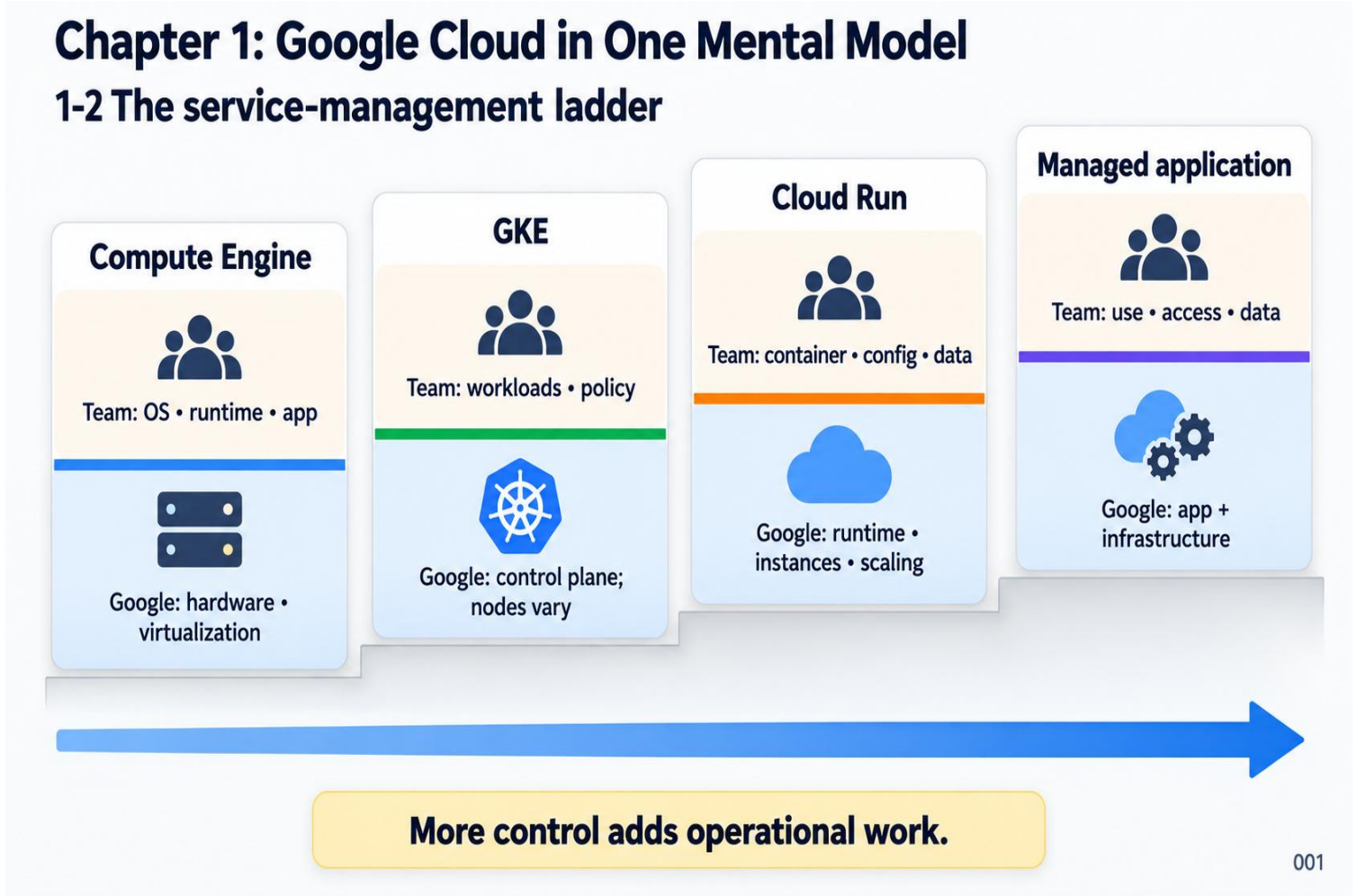


Figure 1-1: The service-management ladder.

Service model	Team primarily manages	Provider primarily manages	Typical Google Cloud example
Infrastructure as a service	Operating system, runtime, application, configuration, and most scaling decisions	Facilities, hardware, virtualization, and core infrastructure	Compute Engine, Google Cloud's VM service
Managed platform	Application, deployment configuration, identity, data, and service limits	Much of the runtime, instance lifecycle, scaling machinery, and platform maintenance	Cloud Run, a managed container application platform
Managed orchestration platform	Workloads, Kubernetes configuration, application reliability, and many cluster policies	Managed Kubernetes control plane and, depending on mode, much of the node infrastructure	Google Kubernetes Engine, Google's managed Kubernetes service
Managed application service	Data use, access, configuration, and business process	Most application and infrastructure operation	A hosted collaboration or business application

These categories are useful, but they are not rigid. A managed Kubernetes service, for example, removes control-plane administration but still requires more platform knowledge than a managed container service. A database service may manage backups automatically, while the customer remains responsible for setting an appropriate backup policy and testing restoration.

An everyday analogy is renting transportation:

- A virtual machine is similar to leasing a car. The provider supplies the vehicle, but the driver chooses the route, handles fuel, and remains responsible for safe use.
- A managed platform is similar to taking a taxi. The passenger states the destination and accepts operating rules, while the provider handles the vehicle and driving.
- A fully managed business application is closer to a delivery service. The customer requests an outcome rather than operating the transportation system.

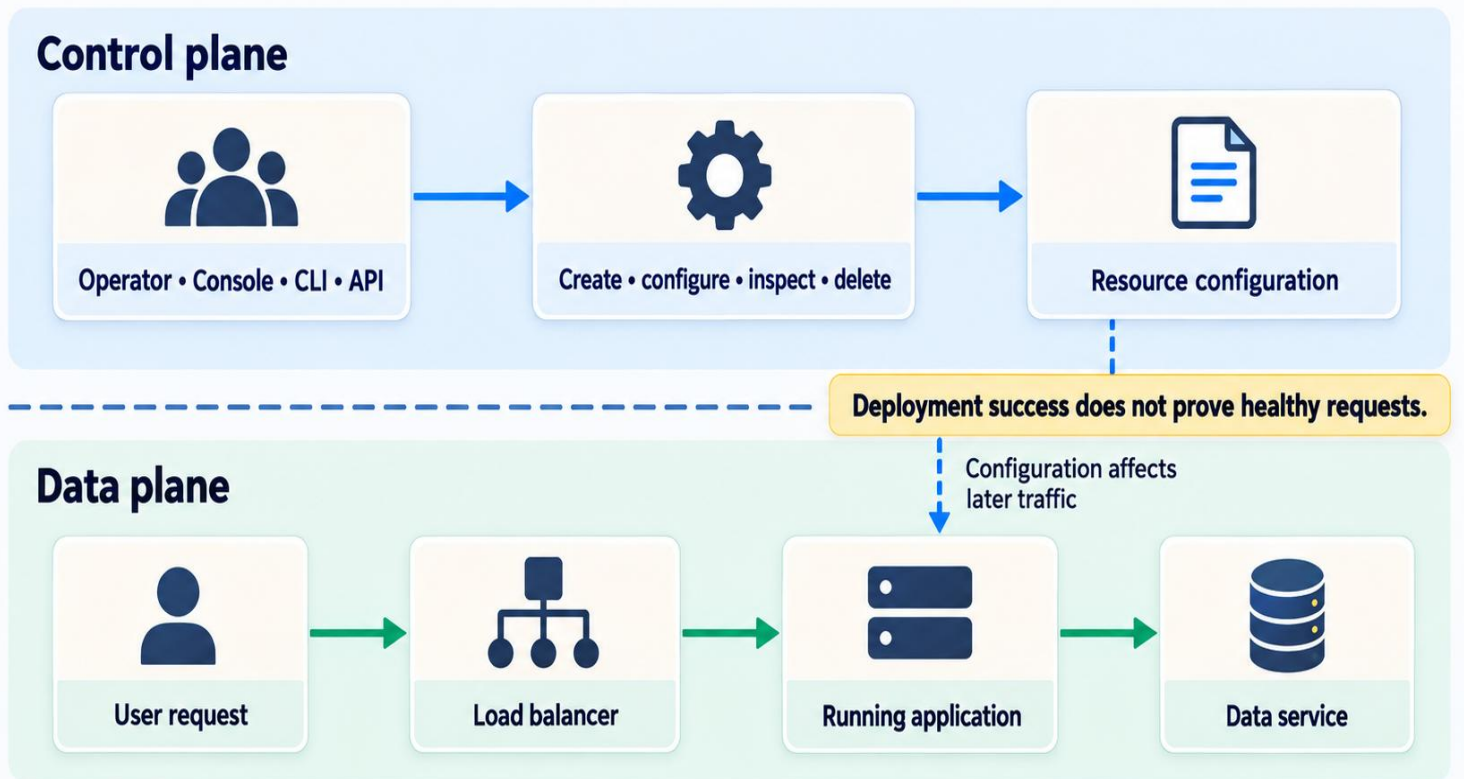
The analogy has limits, but it illustrates the tradeoff: **more control normally brings more operational work.**

Control plane and data plane

Most cloud systems can be viewed as two interacting planes.

Chapter 1: Google Cloud in One Mental Model

1-3 Control plane and data plane



002

Figure 1-2: Control plane and data plane.

The **control plane** is the set of APIs and processes used to create, configure, inspect, and delete resources. Creating a virtual machine, changing an **Identity and Access Management (IAM)** policy—which controls who may perform actions—or deploying a new Cloud Run revision (an immutable configuration version) is control-plane activity.

The **data plane** is the path through which the running system handles application traffic and data. An **HTTPS** request—web traffic protected by **Transport Layer Security (TLS)**—reaching the inventory service, a database query, or an object being read from storage is data-plane activity.

Consider a restaurant. Updating the menu, hiring staff, and changing opening hours are control-plane actions. Preparing and serving a customer's meal is data-plane work. A control-plane change can affect later data-plane behavior, but the two are not the same operation.

This distinction is practical during incidents. If a deployment command fails, the problem may be in the control plane: permissions, API availability, a quota (a service limit on permitted use), or invalid configuration. If deployment succeeds but users receive errors, the problem is more likely in the data plane: routing, application health, dependency access, or exhausted capacity.

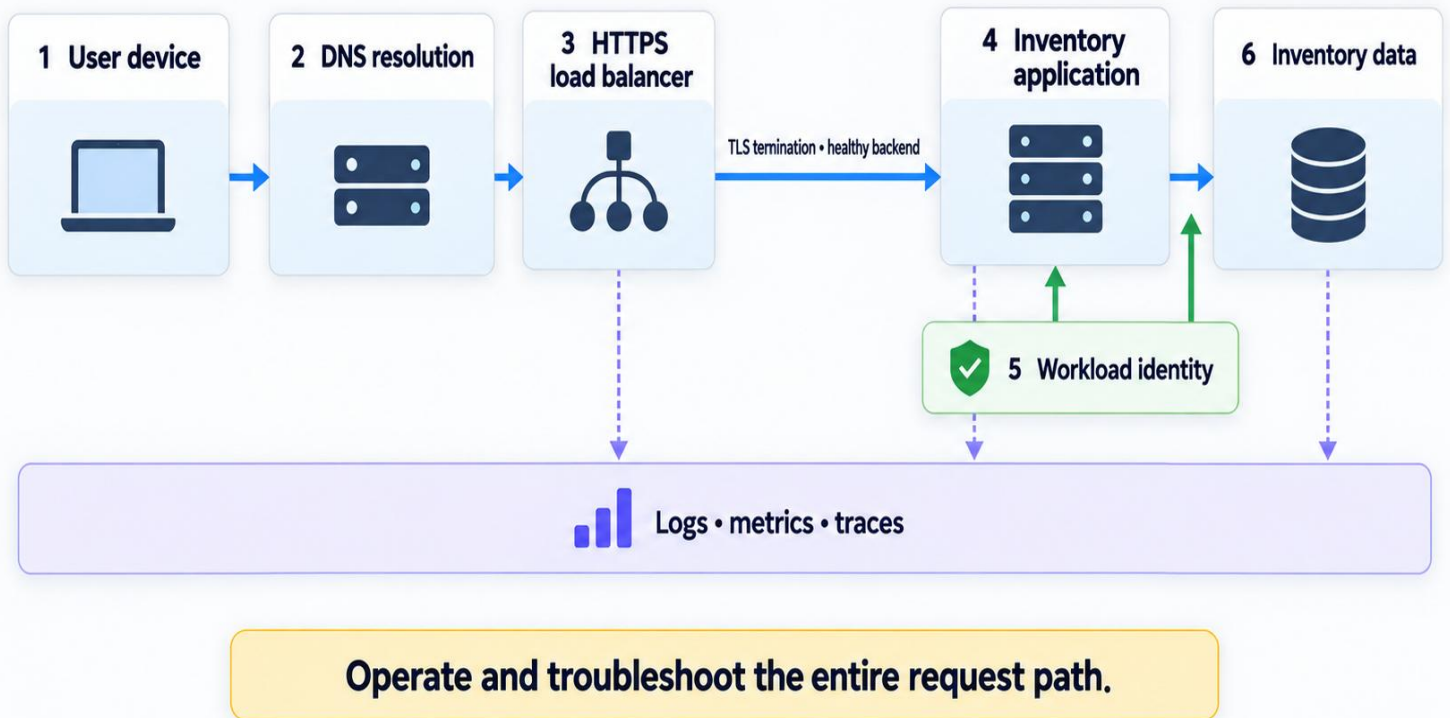
It is also possible for one plane to be healthy while the other is not. An application may continue serving existing traffic during a temporary control-plane issue, while configuration changes are unavailable. Conversely, the console may show a healthy resource even though the application is returning invalid results. **Resource existence is not evidence of application correctness.**

Trace one request through the system

A simple request to the inventory service may follow this path:

Chapter 1: Google Cloud in One Mental Model

1-4 Trace one request through the system



003

Figure 1-3: Trace one request through the system.

1. A user's device resolves a domain name through the Domain Name System, or **DNS**, which translates names such as `inventory.example.com` into a destination.
2. A load balancer accepts the connection, terminates Transport Layer Security, or **TLS**, and chooses a healthy backend.
3. A compute service runs the application code.
4. The application authenticates to a data service using a **workload identity**, which represents application software rather than a person.
5. The data service reads or changes inventory records.
6. Logs, metrics, and traces record what happened so that operators can understand normal and abnormal behavior. A **log** records an event, a **metric** records a numeric measurement over time, and a **trace** follows a request across components.

Every step contains a design decision. DNS records can be wrong. A load balancer can route to the wrong backend. An application can use an overprivileged identity. A database can be available but overloaded. Logs can exist but omit the information needed for diagnosis.

The cloud provider operates substantial parts of this path, but **the team remains responsible for the end-to-end result**. Managed infrastructure does not remove the need to define who may call the service, what constitutes health, how much load dependencies can sustain, or how failure should be handled.

Shared responsibility and shared fate

The **shared-responsibility model** divides security and operational duties between the cloud provider and the customer. Google is responsible for the security and operation of the cloud's underlying facilities and infrastructure. The customer's responsibilities depend on the service used and commonly include identities, access policies, data classification, network exposure, application code, configuration, and recovery objectives.

Chapter 1: Google Cloud in One Mental Model

1-5 Shared responsibility and shared fate

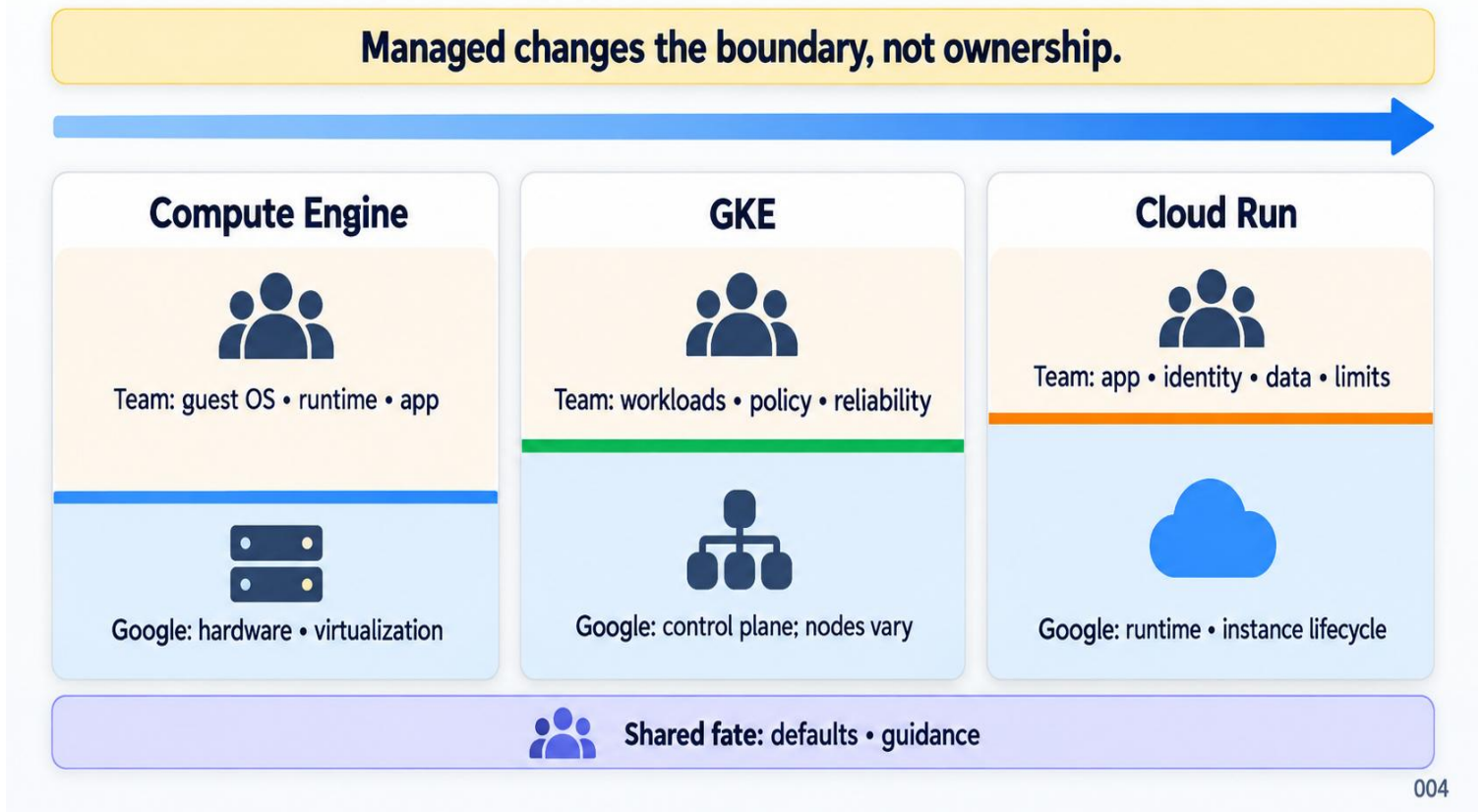


Figure 1-4: Shared responsibility and shared fate.

The exact boundary moves as the service model changes:

- On a virtual machine, the customer normally manages the guest operating system, installed software, patch timing, host-level configuration, and application runtime.
- On Cloud Run, Google manages the server infrastructure and runtime platform, while the customer manages the container, application behavior, identity, network settings, scaling limits, and dependencies.
- On Google Kubernetes Engine (GKE), Google manages important Kubernetes infrastructure, but the customer still owns workload manifests, resource requests, policies, application probes, and application-level reliability.

Google also describes a **shared-fate** approach: the provider supplies secure defaults, guidance, and capabilities that help customers achieve reliable outcomes rather than merely drawing a boundary around responsibility. This support is valuable, but it does not transfer ownership of the customer's architecture or data decisions.

A useful rule is: **never infer a safety property from the word “managed.”** Verify the actual behavior. Ask whether backups are enabled, which locations are used, what the service-level objective—a measurable target for service reliability or performance—covers, how access is granted, and what happens when a dependency fails.

Managed does not mean automatic

A managed service reduces undifferentiated administration. It does not make design choices on behalf of the application owner.

For example, a managed compute service may automatically add instances when traffic rises. That is useful only if the application can start correctly, the maximum instance limit is sufficient, and the database can accept the resulting connection volume. Automatic scaling can move a bottleneck rather than remove it.

Similarly, a service may replicate data across hardware. Replication improves resilience to some failures, but it does not necessarily protect against accidental deletion, malicious change, or a defective application update. Those risks require independent recovery controls and tested operational procedures.

The following questions expose the remaining responsibilities:

- What identity does the workload use, and which permissions does it have?
- Which network paths are open, and why are they required?
- Where does durable state live?
- What happens when an instance, zone, region, or dependency is unavailable?
- How are releases rolled back?
- Which signals show user impact?
- Who receives an alert, and what action can that person take?
- Which resources continue to incur cost when the application is idle?

A service is operationally suitable only when the team can answer these questions.

Six architectural concerns

The [Google Cloud Well-Architected Framework](#) organizes architecture guidance into six concerns. They should be considered together because improving one concern can affect another.

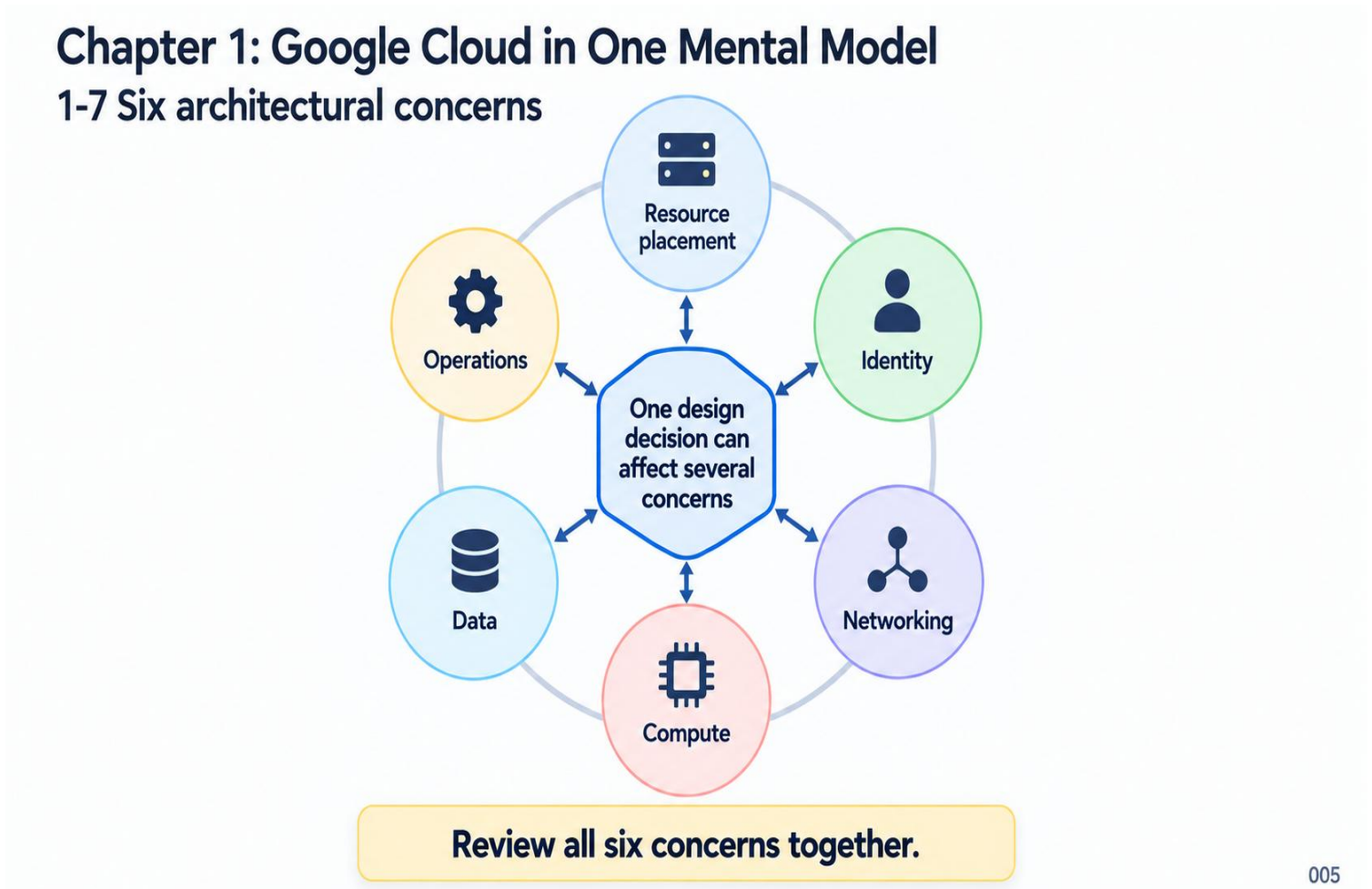


Figure 1-5: Six architectural concerns.

Concern	Foundational question
---------	-----------------------

Concern	Foundational question
Operational excellence	Can the team deploy, observe, change, and troubleshoot the system safely?
Security, privacy, and compliance	Are identities, data, software, and network paths protected according to their risk?
Reliability	Does the system continue to meet its purpose during expected failures and recover from larger failures?
Cost optimization	Does spending reflect useful business value, with waste and unexpected growth detected?
Performance optimization	Does the system meet latency, throughput, and capacity needs efficiently?
Sustainability	Does the design use resources efficiently and avoid unnecessary consumption?

These are not stages completed once. A design may be secure but too expensive, reliable but impossible to operate, or fast under normal load but unable to recover safely. Architecture is the process of balancing these properties against requirements.

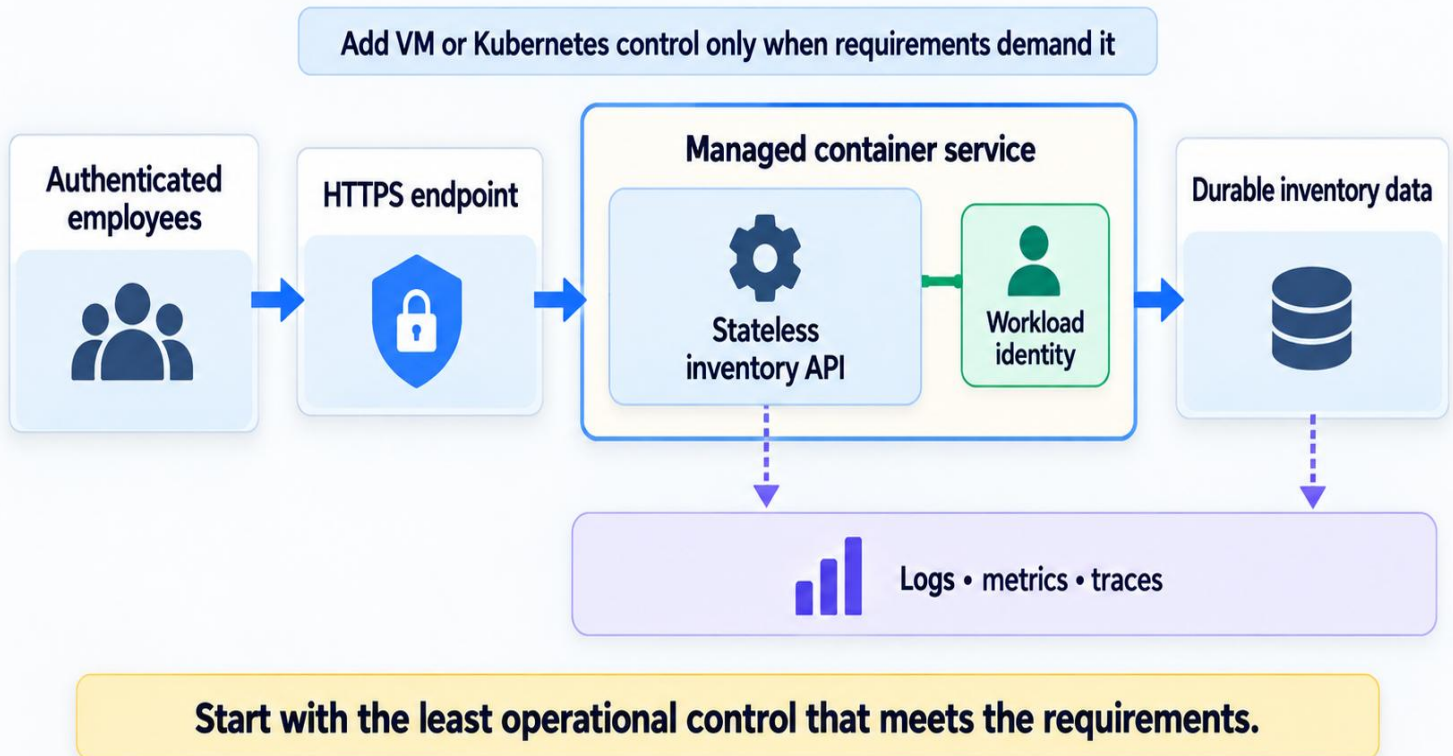
For a beginner, the framework is most useful as a review lens. When examining an unfamiliar resource, do not stop at “What does it do?” Ask “How is it operated, secured, recovered, paid for, and measured?”

A first design for the inventory service

Assume the inventory service starts with these requirements:

Chapter 1: Google Cloud in One Mental Model

1-8 A first design for the inventory service



006

Figure 1-6: A first design for the inventory service.

- Employees access an HTTPS API.
- The application is stateless; durable inventory records are stored outside the compute instance.
- Traffic is modest but varies during working hours.
- The team is small and does not need operating-system customization.

- Access must be limited to authenticated users and services.

A managed container platform is a reasonable starting point because the application can be packaged as a container and does not require host administration. This is not yet a final architecture. The team still needs to choose project boundaries, identities, networking, data services, logging, and recovery controls. The conclusion is narrower: **do not begin with virtual machines or Kubernetes when the requirements do not demand their additional control.**

Another workload in the same organization may need a Windows service, a kernel module, or a vendor appliance. A virtual machine may then be the correct choice. A platform shared by many teams may need Kubernetes APIs and cluster-level policies. GKE may then justify its added complexity. The selection follows requirements rather than fashion.

Mistakes that distort the mental model

“Serverless means there are no servers.” Servers still exist. The provider operates them and exposes a higher-level execution model. The customer still manages code, configuration, identity, dependencies, and limits.

“Managed means highly available by default.” Some managed services provide strong availability characteristics, but the configured location, tier, replication model, client behavior, and application design determine the actual outcome.

“The cloud fixes an unchanged architecture.” Rehosting a system can provide faster provisioning and data-center benefits, but a single fragile instance remains a single fragile instance. Cloud-native reliability comes from explicit design, not from the address of the data center.

“The console is the cloud.” The console is one client of the APIs. Understanding resources, identities, and scope is more durable than memorizing where a button appears.

What to carry forward

Treat every Google Cloud service as a contract of capabilities, constraints, and divided responsibilities. Identify the control-plane actions, trace the data-plane path, and choose the most managed service that meets the workload’s needs. This mental model makes the chapters that follow—resource hierarchy, identity, networking, and compute—parts of one system rather than separate product lists.

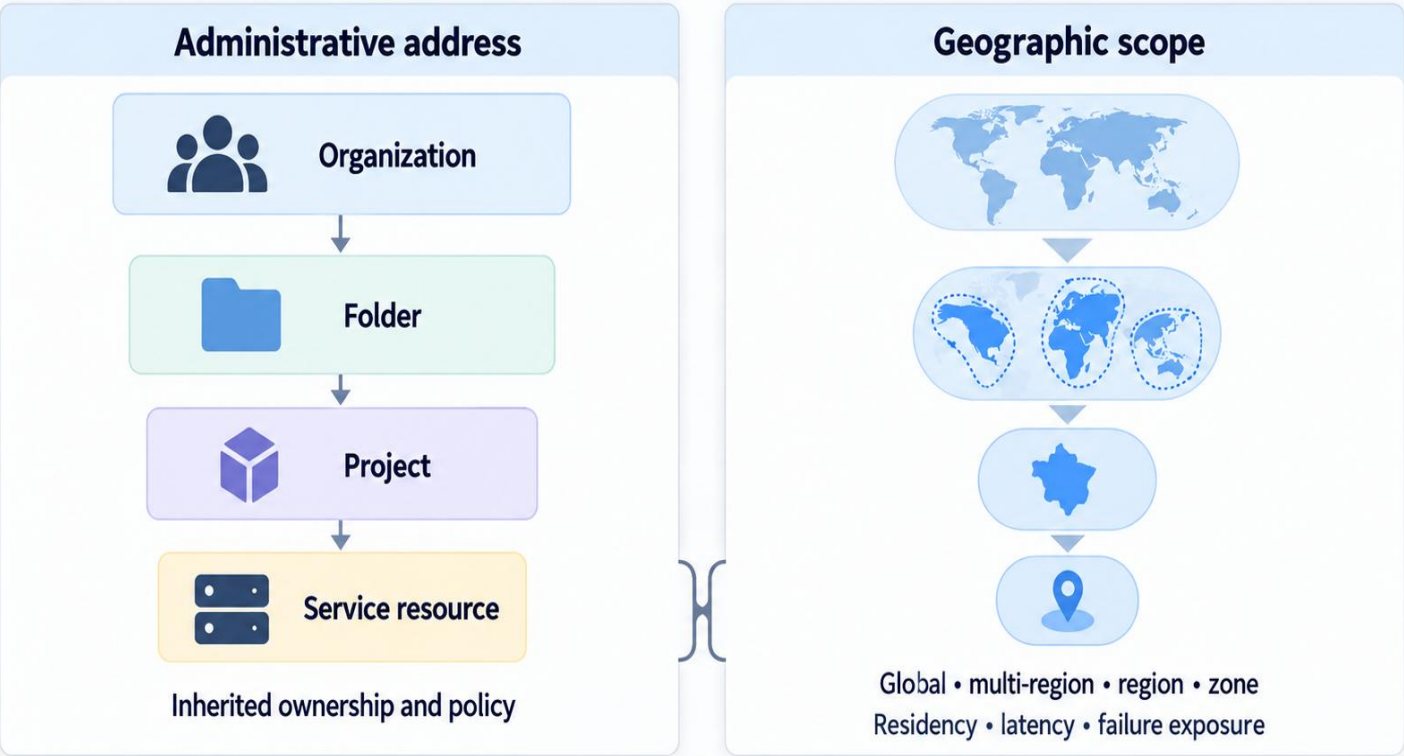
Chapter 2 — Resource Hierarchy, Projects, and Locations

Every resource has an administrative address

A Google Cloud resource is governed by two kinds of placement: its position in the resource hierarchy and its **geographic scope**. The hierarchy determines ownership, inherited policy, and administrative boundaries. Geographic scope determines where data or computation resides, which failures can affect it, and how it communicates with other components.

Chapter 2: Resource Hierarchy, Projects, and Locations

2-1 Every resource has an administrative address



Hierarchy and geography are separate coordinates.

007

Figure 2-1: Every resource has an administrative address.

This distinction prevents a common error. A resource can belong to the correct production project but still be placed in an unsuitable region. It can also be in the preferred region but inherit overly broad access from a parent folder. Both dimensions must be correct.

Google Cloud’s [resource hierarchy](#) usually has four levels:

Level	Purpose	Typical examples of decisions made there
Organization	Top-level administrative container for a company or institution	Organization-wide policies, central administrators, audit configuration
Folder	Optional grouping beneath an organization	Environment, business unit, platform, or delegated-administration boundaries
Project	Fundamental boundary for resource ownership and many operational controls	Enabled APIs, IAM policy, quotas, billing linkage, environment isolation
Service resource	The actual resource used by a service	VM instance, Virtual Private Cloud (VPC) network, Cloud Run service, GKE cluster

Each resource has one parent in this hierarchy. The structure resembles a company filing cabinet: the organization is the cabinet, folders are drawers, projects are case files, and service resources are the documents inside a file. A rule placed on a drawer can affect every file within it.

The organization resource

An **organization resource** represents the top of an enterprise's Google Cloud hierarchy. It is commonly associated with a managed Google Workspace or Cloud Identity domain. Google Workspace is Google's managed productivity and collaboration suite, while Cloud Identity provides centrally managed identities and device-related capabilities without requiring the full productivity suite.

The organization is the natural place for controls that should apply broadly. Examples include restricting which geographic locations may be used, preventing certain risky configurations, or delegating administrative roles to a central platform team.

An organization resource is not simply a label for projects. It establishes an administrative root. Policies applied there can flow to folders, projects, and resources below it. For this reason, organization-level changes require careful review: a small policy change can affect many systems.

A personal experiment may exist outside an organization, depending on how the account and project were created. Enterprise work should normally use organization-owned projects so that the company retains administrative control when individuals change roles or leave.

Folders express administration, not geography

A **folder** groups projects and other folders so that policy and administration can be delegated. A folder does not contain application traffic or data in the way that a network or storage service does. It is an administrative construct.

Common folder strategies include grouping by:

- Environment, such as `development`, `test`, and `production`.
- Business unit, such as `retail`, `finance`, and `operations`.
- Platform function, such as `shared-infrastructure` and `application-teams`.
- A combination of these, using nested folders where the additional hierarchy has a clear administrative purpose.

There is no universal folder tree. The right structure follows ownership and policy needs. A small organization may need only a production folder and a non-production folder. A large enterprise may need business-unit delegation beneath central guardrails.

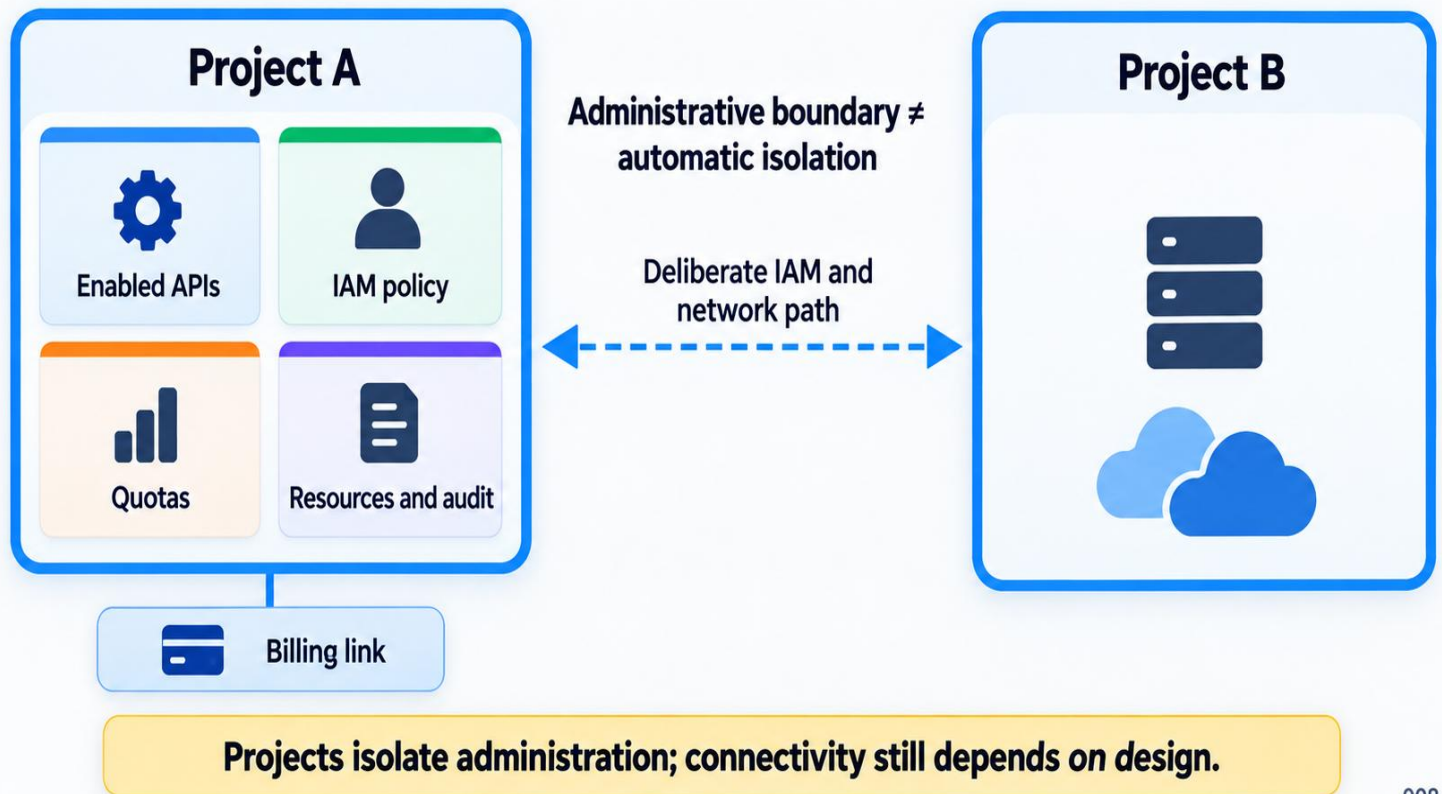
Do not create deep folder structures merely to make the hierarchy look organized. Every level should answer a useful question such as "Who administers these projects?" or "Which policy applies to this group?"

Projects are operational boundaries

A **project** is the fundamental container for most Google Cloud resources. It is more than a billing folder. A project commonly defines or influences:

Chapter 2: Resource Hierarchy, Projects, and Locations

2-4 Projects are operational boundaries



008

Figure 2-2: Projects are operational boundaries.

- Which service APIs are enabled.
- Which identities have access and at what scope.
- Which quotas and limits apply.
- Which billing account pays for usage.
- Which resources are discovered and managed together.
- Which audit records and operational settings apply.
- The trust boundary between one environment or application and another.

Projects provide useful isolation, but they are not impermeable. Resources in different projects can communicate when networking and permissions allow it. A user can have access to multiple projects. Centralized services can be shared across projects. **A project is a strong administrative boundary, not an automatic network air gap.**

Each project has three identifiers:

Identifier	Important property	Example form
Display name	Human-readable and changeable; not necessarily unique	Inventory Production
Project ID	Globally unique identifier chosen or generated at creation; normally treated as permanent	inventory-prod-7f3a
Project number	Numeric identifier assigned by Google; used by some services and identities	123456789012

Engineers often see the project ID in commands, URLs, resource names, and logs. The project number appears in generated service identities and some API relationships. The display name is convenient for people but should not be treated as a stable programmatic identifier.

Because a project ID may appear in externally visible names, automation, and audit records, choose it as though it will be long-lived. Avoid embedding confidential information or an employee's name. Renaming the display name later does not change the project ID.

Policy inheritance and effective policy

Policies applied to an organization or folder can be inherited by descendants. **Inheritance means that a child can be affected by policy set on a parent even when that policy is not visibly authored on the child itself.**

Chapter 2: Resource Hierarchy, Projects, and Locations

2-5 Policy inheritance and effective policy

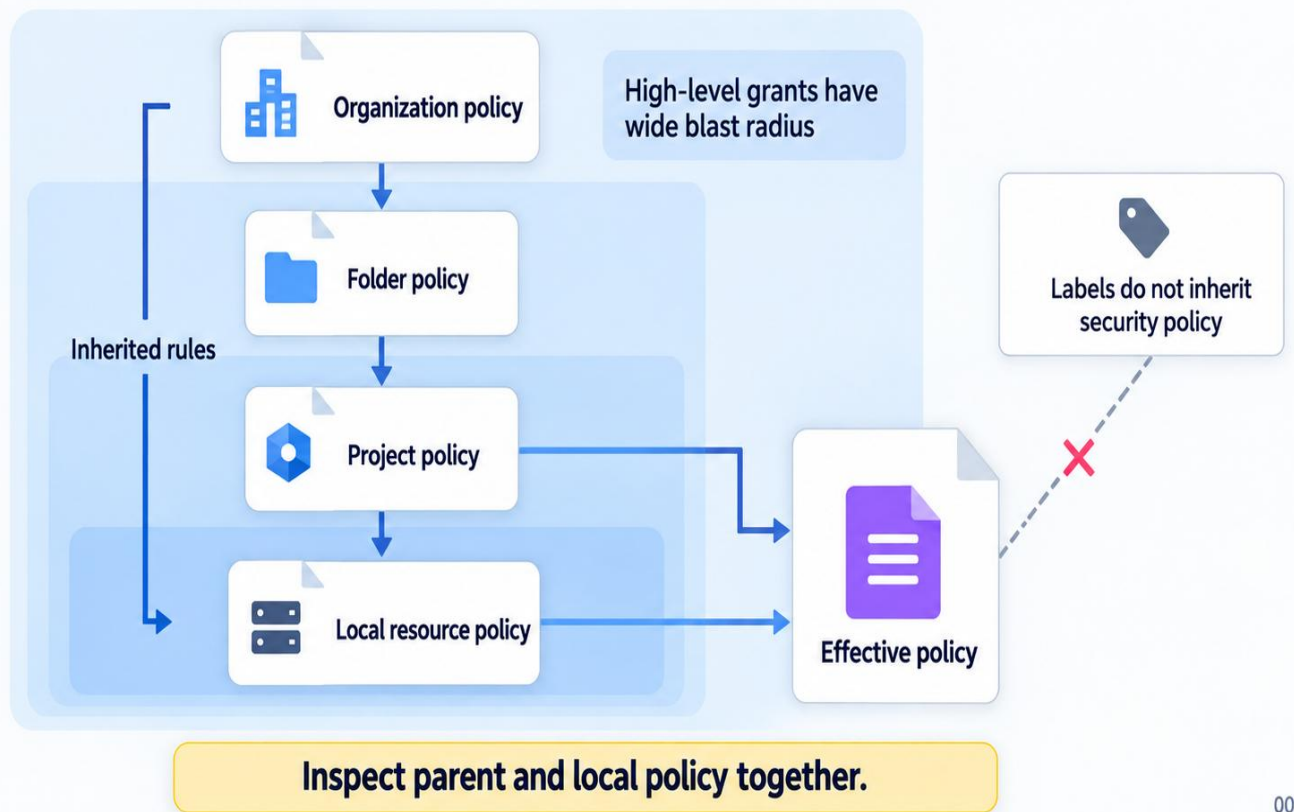


Figure 2-3: Policy inheritance and effective policy.

Imagine a company rule that all office doors must lock automatically. A team does not need to repeat the rule on every room; the building policy applies throughout. Similarly, an IAM role granted at a folder can provide access to projects below that folder. An organization policy can constrain configurations across many projects.

This creates two operational lessons:

1. Inspecting only the project's local policy may not explain its effective behavior.
2. Granting broad access at a high level can affect far more resources than intended.

When an action is unexpectedly allowed or denied, investigate parent policies as well as local configuration. The effective policy is the result of all applicable rules, not just the nearest visible rule.

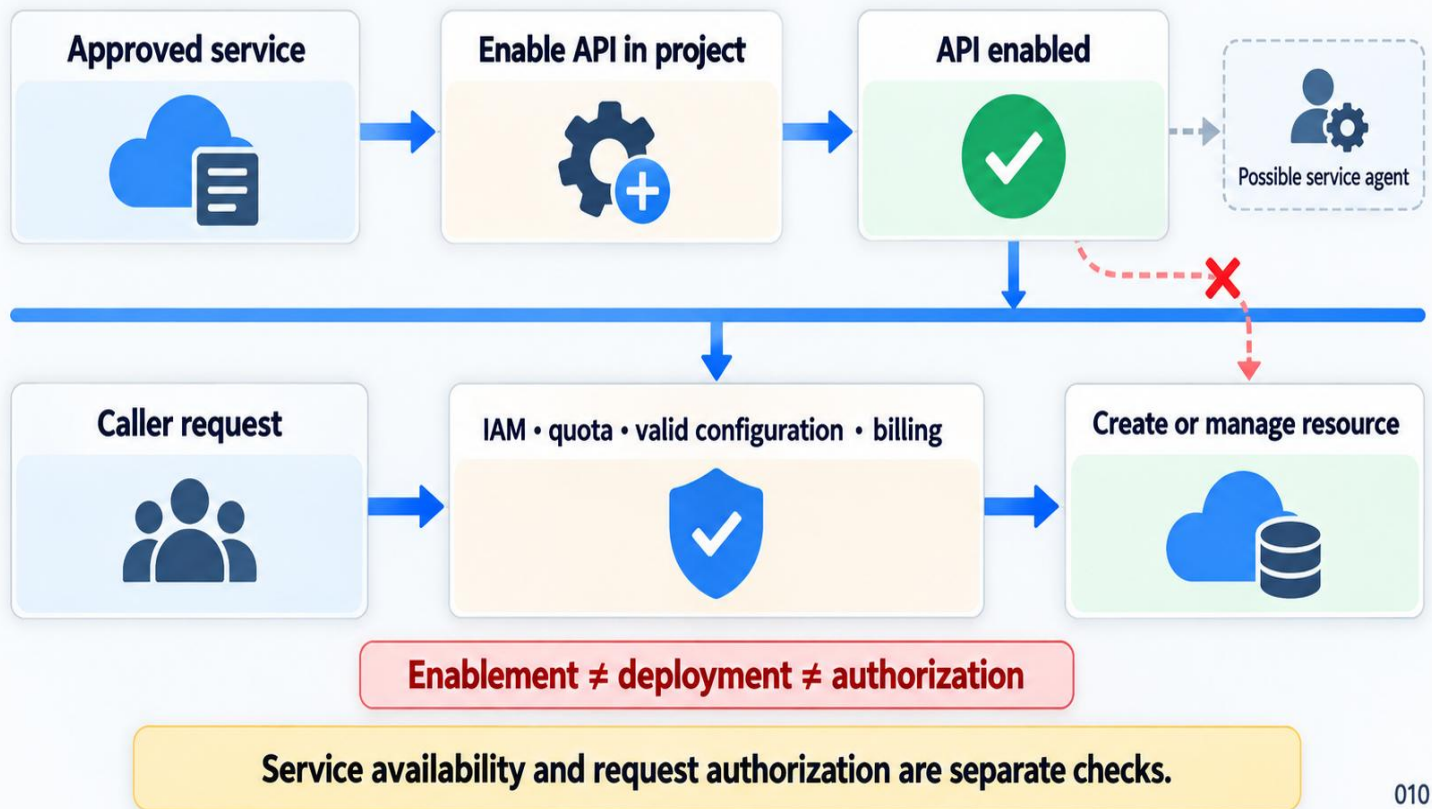
Folders and labels must not be mistaken for equivalent security mechanisms. A folder can participate in policy inheritance. A label is metadata used for organization and reporting; it does not itself prevent access.

APIs are enabled per project

A Google Cloud service is generally used through a corresponding service API. Before a project can create or manage certain resources, the relevant API must be enabled in that project. Enabling an API makes the service available to the project; it does not create the service's resources and does not grant a user permission to operate it.

Chapter 2: Resource Hierarchy, Projects, and Locations

2-6 APIs are enabled per project



010

Figure 2-4: APIs are enabled per project.

For example, enabling the Compute Engine API establishes the service relationship needed to create virtual machines. The engineer still needs IAM permissions, quota, valid configuration, and billing where the service requires it.

Service enablement belongs in repeatable project setup rather than relying on an engineer to click through prompts. In an established environment, it may be controlled by a platform team or infrastructure automation. Enabling unnecessary APIs is not normally the largest security risk by itself, but minimizing and reviewing enabled services makes the project easier to govern and understand.

Some APIs depend on other APIs or create service agents. A **service agent** is a Google-managed service account used by a Google Cloud service to act within a customer project. Removing its permissions without understanding the dependency can break the service.

Billing linkage is a separate relationship

A **Cloud Billing account** pays for chargeable usage from linked projects. It is not a parent in the resource hierarchy. A project belongs under an organization or folder and is separately linked to a billing account.

This separation is useful. A company may have several billing accounts for legal entities, contracts, or cost ownership while maintaining one administrative hierarchy. Conversely, projects in different folders may use the same billing account.

Changing or removing a billing link can affect paid services, but it does not move the project in the hierarchy. Moving a project to another folder changes its inherited administration, but it does not automatically change who pays for it.

Always examine hierarchy and billing as two independent relationships.

Project design for development and production

For the inventory service, a simple enterprise hierarchy could be:

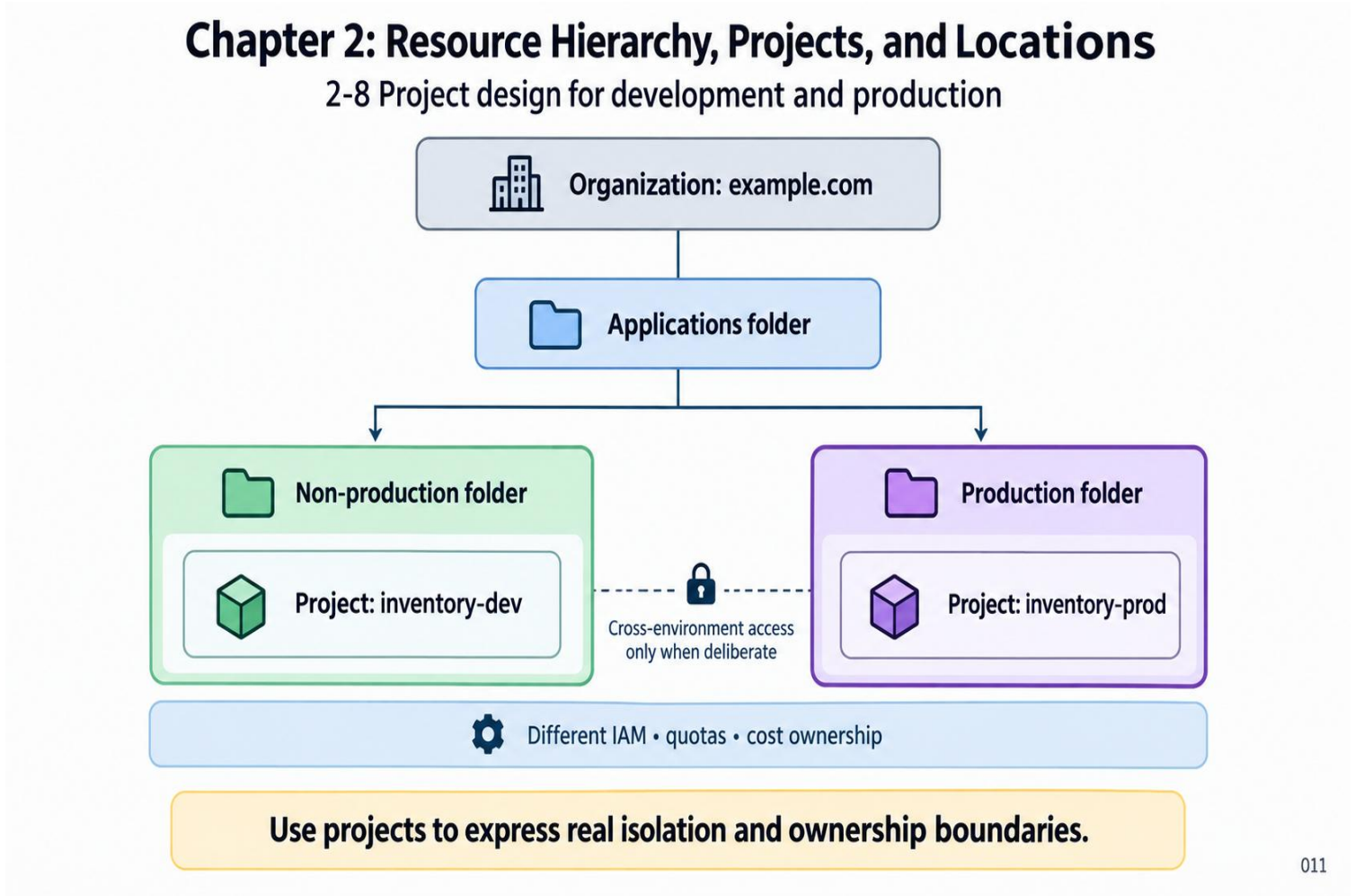


Figure 2-5: Project design for development and production.

```
Organization: example.com
├─ Folder: Applications
│   ├── Folder: Non-production
│   │   └─ Project: inventory-dev
│   └─ Folder: Production
│       └─ Project: inventory-prod
```

The development and production environments use separate projects. This separation reduces accidental cross-environment changes, allows different access policies, isolates many quotas, and improves cost attribution. Production operators can be granted access without automatically receiving the same power in every development project, and developers can experiment without being able to modify production resources.

This is one reasonable example, not a universal rule. A small low-risk application may use fewer projects. A regulated application may need separate projects for networking, security tooling, data, and workloads. The project count should be justified by isolation and ownership, because every additional project also adds configuration and lifecycle overhead.

Two common strategies illustrate the tradeoff:

Strategy	Advantage	Cost or risk
Project per environment	Clear separation between development, test, and production	Related components are spread across projects and need deliberate cross-project access
Project per application and environment	Strong ownership, cost attribution, and blast-radius control	Larger project estate and more automation needed for consistent setup

Avoid one permanent “everything” project for unrelated production systems. It makes access, quotas, billing, and decommissioning difficult to separate. Also avoid creating a new project for every minor resource when no distinct boundary is gained.

Names, labels, and tags solve different problems

A naming convention helps people recognize a resource’s purpose and environment. A name such as `inv-prod-api-mig` is more useful than `instance-group-1`, but names should remain readable and should not attempt to encode every property. Properties change; names often persist.

A **label** is a key-value pair attached to supported resources for organization, filtering, and cost attribution. Examples include `environment=prod`, `application=inventory`, and `cost_center=operations`. Labels are useful only when teams apply them consistently.

A **tag** in Resource Manager is a centrally governed key-value association that can be used by supported policy and access-control features. Tags are designed for governance relationships, whereas labels are primarily resource metadata. The word “tag” is also used by some individual services for other mechanisms, such as network tags on virtual machines. Those service-specific tags are not the same as Resource Manager tags.

Use the mechanisms according to their function:

- Use names for human recognition.
- Use labels for inventory, filtering, and cost reporting.
- Use Resource Manager tags when a supported policy needs governed metadata.
- Use folders and projects for administrative boundaries and inherited policy.

Metadata does not become a security boundary merely because it contains the word `production`.

Global, regional, zonal, and multi-regional scope

Google Cloud resources have location scopes. The main categories are:

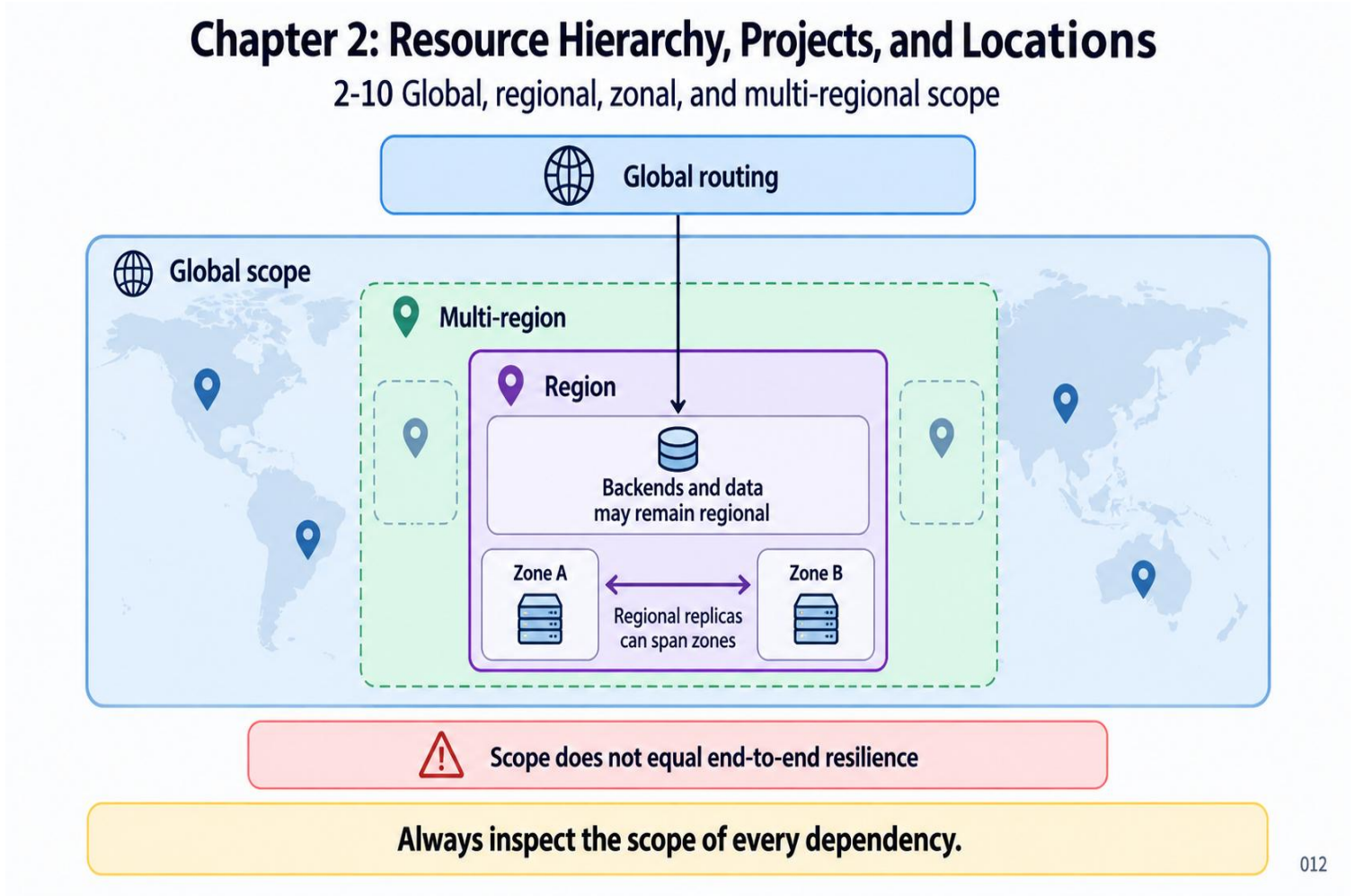


Figure 2-6: Global, regional, zonal, and multi-regional scope.

Scope	Meaning	Failure consideration
Zonal	Resource is placed in one zone within a region	A zone disruption can directly affect it
Regional	Resource spans or is managed within one region	Better protection from a single-zone issue, but exposed to regional disruption
Multi-regional	Service places or manages data across a defined large geography	Behavior depends on the specific service's replication and consistency model
Global	Resource is not tied to one region in its control or routing scope	Backends and data can still be regional or zonal

A **region** is a geographic area such as a metropolitan area, and a **zone** is an isolated deployment area within a region. Zones in the same region have low-latency connectivity, but they are distinct failure domains. A regional architecture can place replicas across zones so that a single-zone failure does not remove all capacity.

A global resource does not make an application globally resilient. A global load balancer can route worldwide, but if every backend and the only database are in one region, that region remains a critical dependency. Similarly, a VPC network is global, while its subnets and many attached workloads are regional or zonal.

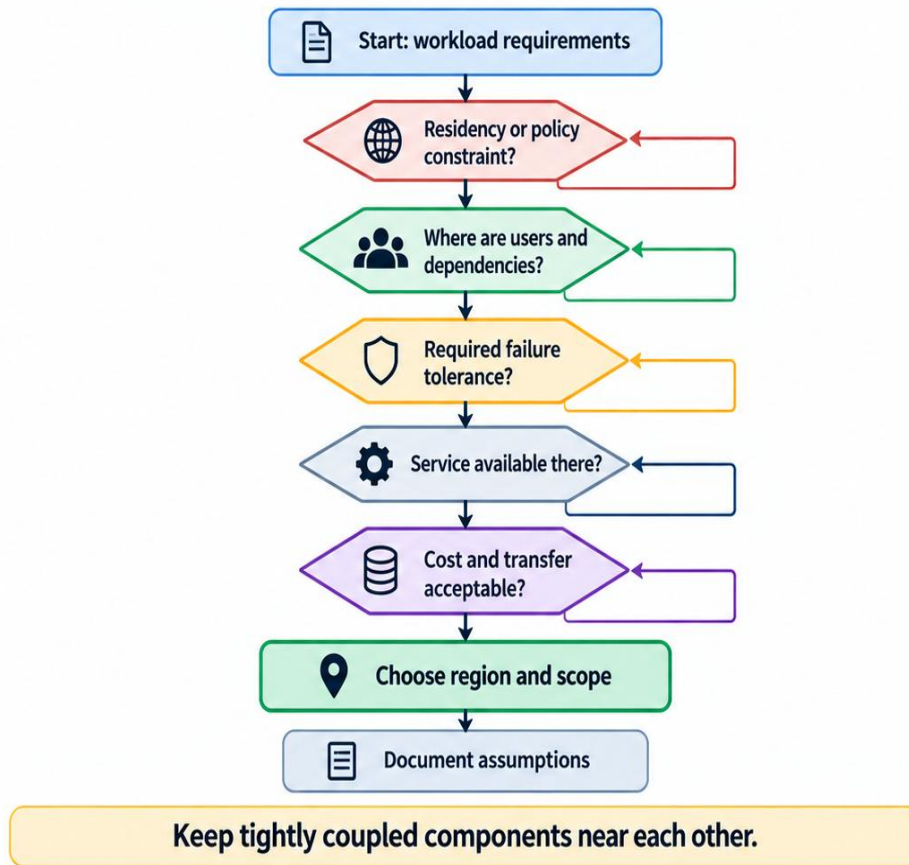
Multi-region is also not a universal synonym for “automatically survives everything.” Each service defines how data is replicated, how requests are routed, what consistency is provided, and what failures are covered. Those service-specific details must be verified during architecture design.

Choose location from requirements

Location selection should follow explicit constraints rather than habit. Evaluate at least these factors:

Chapter 2: Resource Hierarchy, Projects, and Locations

2-11 Choose location from requirements



013

Figure 2-7: Choose location from requirements.

- **Latency:** users and dependent systems should be close enough to meet response-time needs.
- **Availability:** replicas and dependencies should be placed to tolerate the failures covered by the service objective.
- **Data residency:** laws, contracts, and internal policy may restrict where data is stored or processed.
- **Service availability:** not every product, feature, or machine type is offered in every location.
- **Cost:** prices and network data-transfer charges can differ by location and traffic path.
- **Dependency placement:** frequent communication across regions can add latency, failure modes, and cost.

For the inventory service, suppose most users and the corporate data systems are in Japan. A team might choose a Japanese region as its primary location to reduce latency and simplify residency requirements. This is an example for that environment, not a recommendation for every company. The actual decision must account for the organization's users, policies, recovery design, service availability, and cost.

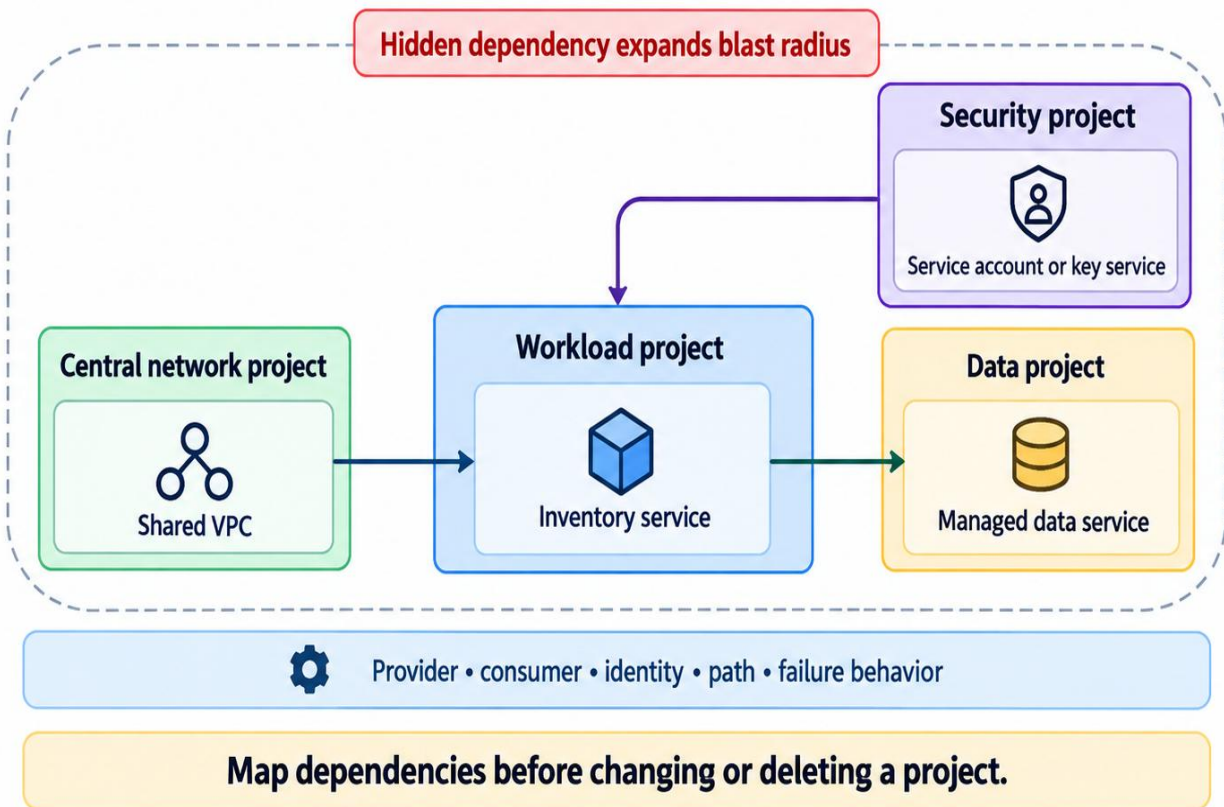
Keep tightly coupled components near one another unless a requirement justifies separation. An application in one region making every database call to another region can experience avoidable latency and network cost. Cross-region replication may be valuable for recovery, but it should be designed as such rather than occurring accidentally.

Resource dependencies cross boundaries

Resources can depend on items in other projects or locations. A workload project may use a VPC hosted by a central network project. A deployment pipeline in one project may impersonate a service account in another. A globally scoped load balancer may use backends in multiple regions.

Chapter 2: Resource Hierarchy, Projects, and Locations

2-12 Resource dependencies cross boundaries



014

Figure 2-8: Resource dependencies cross boundaries.

Cross-boundary designs are normal, but each relationship should be visible and owned. For every dependency, record:

- The providing and consuming teams.
- The resource names and projects involved.
- The identity and permissions used.
- The network path.
- The expected failure behavior.
- The change and decommissioning process.

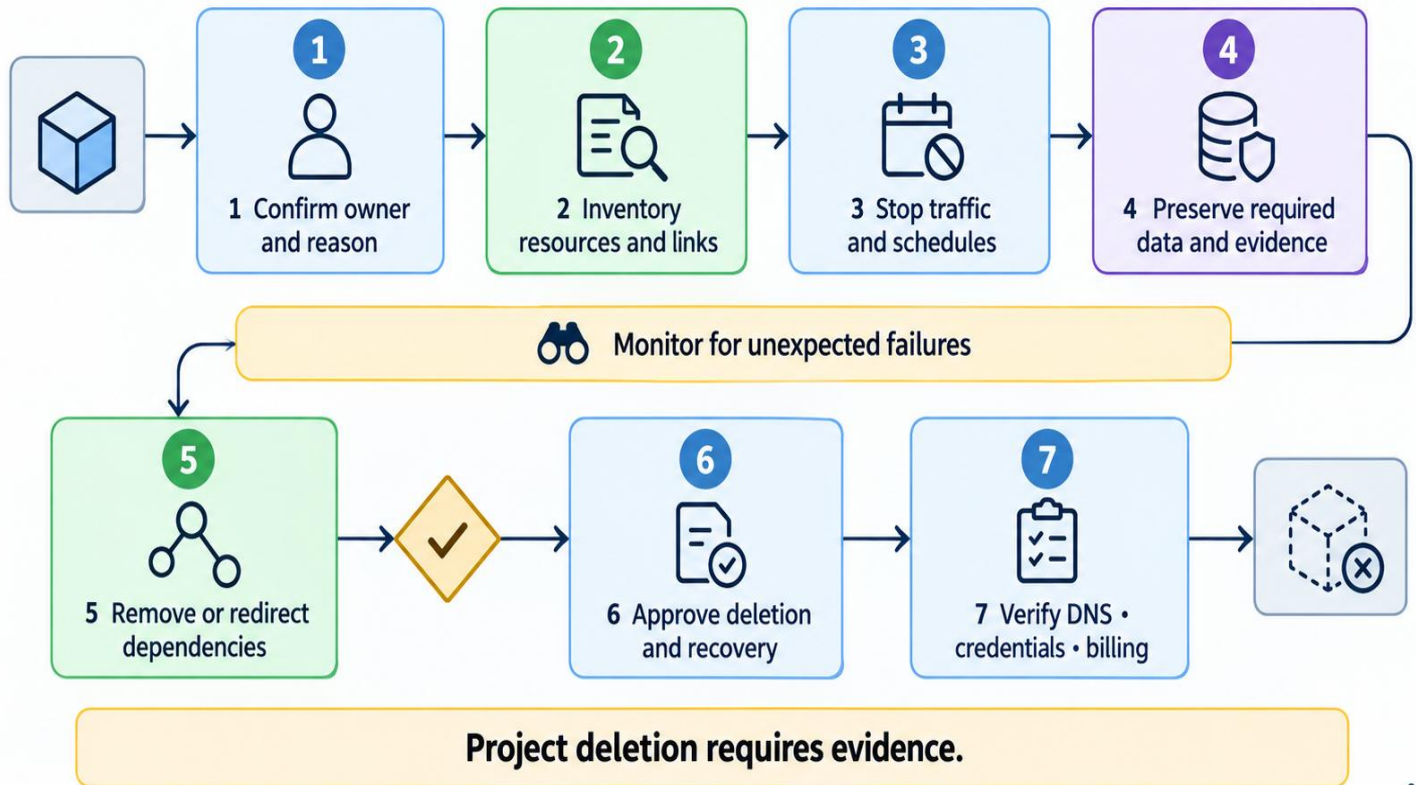
An undocumented shared dependency creates hidden **blast radius**, meaning the scope of systems and users affected by one failure or change. Deleting an “unused” project may break production elsewhere if its DNS zone, service account, network, or image repository is still consumed.

Project lifecycle and safe decommissioning

Project creation is easy; safe project deletion requires evidence. A project may contain data, network endpoints, service identities, cryptographic keys, logs, backups, or shared resources used by other projects.

Chapter 2: Resource Hierarchy, Projects, and Locations

2-13 Project lifecycle and safe decommissioning



015

Figure 2-9: Project lifecycle and safe decommissioning.

A disciplined decommissioning process is:

1. Identify the project owner and approved reason for retirement.
2. Inventory resources, enabled services, IAM bindings, billing activity, and cross-project dependencies.
3. Stop incoming traffic and scheduled work.
4. Preserve data and audit evidence required by retention policy.
5. Remove or redirect dependencies, then monitor for unexpected failures.
6. Apply deletion protection where supported until the review is complete.
7. Delete the project through the approved process and track the limited recovery period.
8. Verify that external references, DNS records, credentials, and billing artifacts are cleaned up.

Project deletion is not a substitute for deleting a single application. It has a much larger blast radius. Conversely, leaving abandoned projects indefinitely creates security, compliance, and cost risk.

Three checks before touching a resource

Before changing any Google Cloud resource, identify:

1. **Hierarchy:** Which organization, folder, and project own it, and which parent policies apply?
2. **Location:** Is it global, regional, zonal, or multi-regional, and what failure exposure follows?
3. **Dependencies:** Which identities, networks, services, and other projects rely on it?

These checks take little time and prevent many high-impact mistakes. The next chapter turns them into a practical working method using the console, Cloud Shell, the Google Cloud command-line interface (CLI), APIs, and credentials.