

Packer A Visual Guide

Machine Image Fundamentals for Systems Engineers From HCL Templates to Secure Build Pipelines

Reiner Braun

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with Packer, 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 when learning Packer for the first time. Each chapter assumes the essential vocabulary and mental models established earlier, so the sequence builds the necessary background knowledge step by step.
- Review the **What You'll Be Able to Do** section at the beginning of each chapter before reading the details. It states the chapter's practical learning direction and helps you recognize which ideas deserve the most attention.
- Use the diagrams to understand relationships, processing flows, responsibility boundaries, state changes, and structures that are difficult to picture from prose alone. The surrounding text remains the authoritative explanation.
- Run the code and hands-on examples yourself whenever a safe test environment is available. Before each command, confirm the stated prerequisites; afterward, compare the observed result with the expected result.
- Use each **Chapter Summary** and the accompanying **Review Questions** to confirm that you understand and can recall the essential fundamentals before moving forward.
- Treat advanced-topic overviews as signposts rather than material to memorize now. Their purpose is to help you recognize technologies and patterns that may become relevant in later study or practical work.

Table of Contents

Introduction

How to Use This Book

Part I: The Big Picture and First Success

Chapter 1: What Packer Does and Why Teams Use It

- The problem Packer solves
- The assembly-line model
- Image automation and immutable infrastructure
- Where Packer fits among other tools
- When Packer is a good fit
- A practical team flow
- Misconceptions that cause design mistakes

Chapter 2: The Packer Build Lifecycle

- Core vocabulary
- The lifecycle step by step
 1. Load and evaluate the template
 2. Locate the required plugins
 3. Ask the builder to create temporary resources
 4. Wait for the guest and establish communication
 5. Run provisioners in order
 6. Finalize and capture the artifact
 7. Perform any post-processing
 8. Clean up and report the result
- What runs on the Packer host and what runs in the guest
- Following one hypothetical Linux build
- Reading logs by phase
- Failure boundaries
- Temporary resources and final resources

Packer turns a version-controlled build definition into a reusable machine image. A **machine image** is a preconfigured operating-system template from which a cloud or virtualization platform can launch new machines. Packer creates a temporary machine, connects to it, applies build-time configuration, captures the result, and reports an **artifact**, meaning the final image or file produced by the build.

A **Packer template** is the configuration that declares how this work should happen. A **base image** is the starting machine image that Packer modifies, and **provisioning** is the build-time process of installing and configuring software inside the temporary machine.

The essential mental model is simple:

```
graph TD; A[trusted base image] --> B[Packer template + provisioning logic]; B --> C[temporary build machine]; C --> D[configured and verified machine]; D --> E[new image artifact]; E --> F[deployment by another tool or platform];
```

This text establishes the foundation needed to read modern templates written in **HashiCorp Configuration Language (HCL)**, a declarative language that describes configuration rather than a step-by-step program. It also explains how to run a build safely, identify where failures occur, and create reusable image definitions without hiding important platform differences.

Part I: The Big Picture and First Success

Chapter 1: What Packer Does and Why Teams Use It

| What You'll Be Able to Do

Packer automates image creation; it does not deploy or continuously manage the machines created from those images. This boundary is the most important fact to understand before learning its syntax.

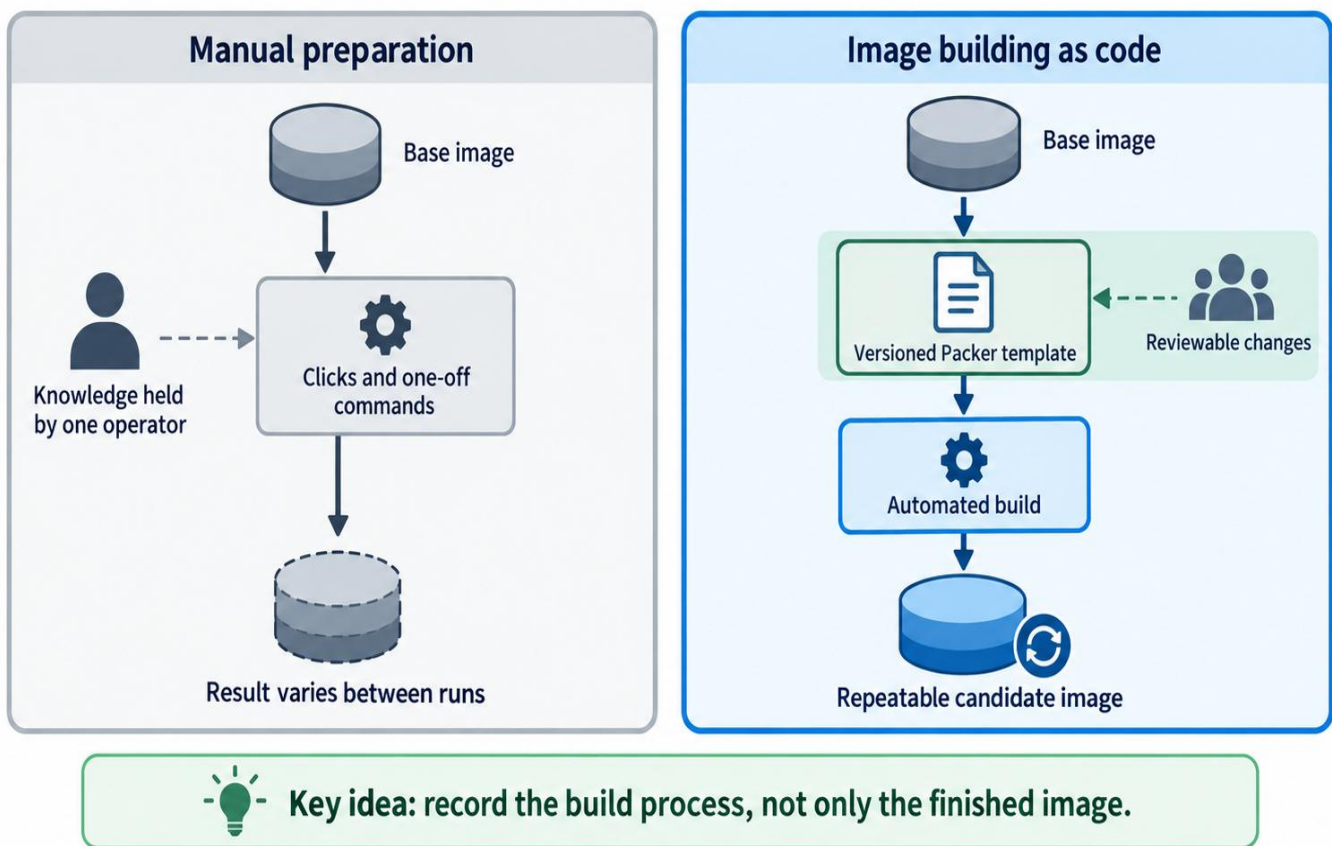
The problem Packer solves

A server can be prepared manually: launch a virtual machine, log in, install packages, copy configuration files, enable services, and then take a **snapshot**, meaning a point-in-time copy of its disk state. That approach may work once, but it produces an undocumented result that is difficult to reproduce. The engineer who created it may remember the steps, while everyone else sees only the finished image.

Packer changes the process from **manual image preparation** to **image building as code**. The build definition can be stored in source control, reviewed, run again, and improved like other software.

Chapter 1: What Packer Does and Why Teams Use It

1-1 Manual Work vs Image Building as Code



001

Figure 1-1: Manual image preparation versus image building as code

Machine images take platform-specific forms. Examples include an **Amazon Machine Image (AMI)** in **Amazon Web Services (AWS)**, an Azure managed image, a Google Compute Engine image, and a virtual disk or template used by a **hypervisor**, which is software that runs and manages virtual machines.

A **golden image** is an organization-approved machine image that contains a known operating-system baseline, standard packages, configuration, and controls. “Golden” does not mean perfect or permanent; it means that the image has passed the organization’s approval process for a defined purpose.

Consider a team that needs 40 web servers. Without an image pipeline, every server might install the web server package during first boot. That introduces repeated downloads, longer startup times, and more opportunities for package repositories or scripts to

behave differently. With Packer, the team installs and configures the common software once during an image build. Each deployed server starts from the same prepared baseline.

The result is consistency at launch time. Runtime values that differ between environments, such as a database password or the address of a production service, should normally be supplied after launch rather than baked into the reusable image.

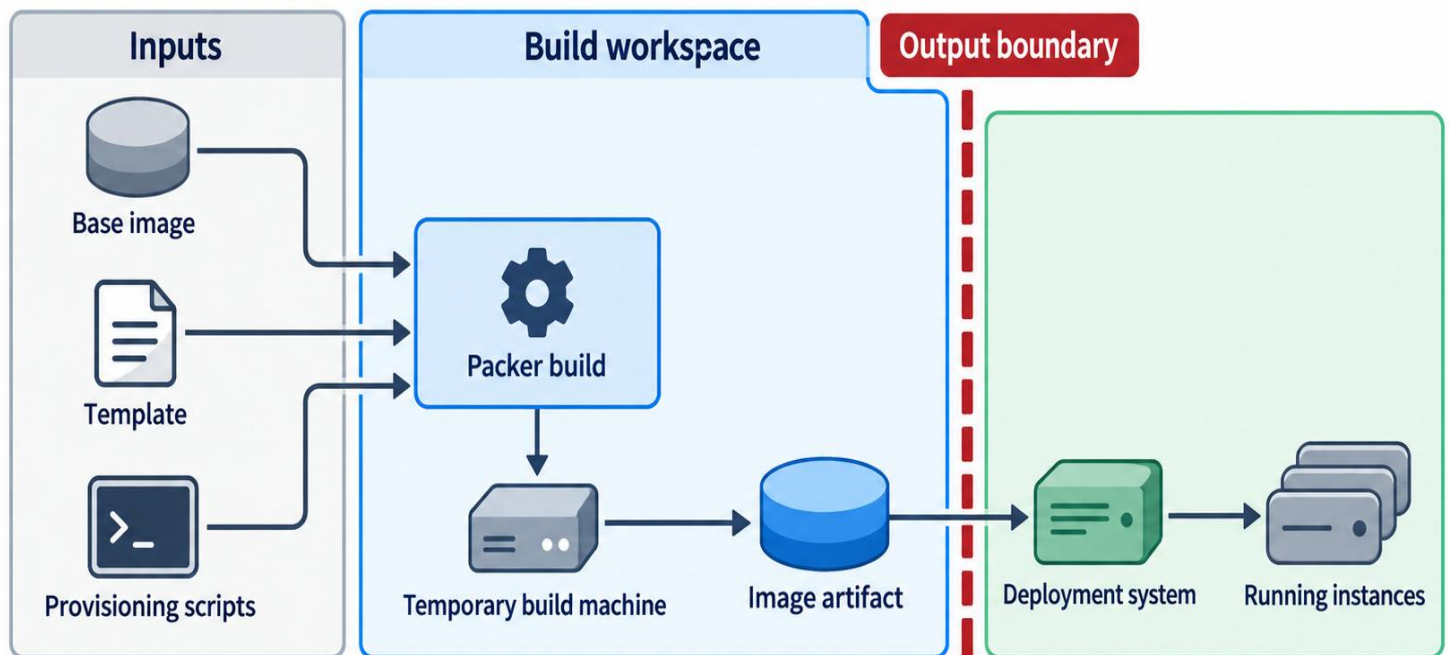
The assembly-line model

Packer is easier to understand when compared with an assembly line:

- The **template** is the production plan.
- The **builder** is the platform-specific machinery that creates a temporary machine and captures an artifact.
- The **temporary machine** is the workbench on which changes are made.
- The **provisioners** are the ordered installation and configuration steps.
- The **artifact** is the finished image or file produced by the build.

Chapter 1: What Packer Does and Why Teams Use It

1-2 The Packer Assembly Line



Key idea: Packer creates the image; another system deploys it.

002

Figure 1-2: Packer as an assembly line with a clear output boundary

The assembly line does not decide how many finished products a warehouse should ship to customers. In the same way, Packer creates the image but does not decide how many production instances should run.

Image automation and immutable infrastructure

Immutable infrastructure is an operating model in which a system is replaced with a newly built version instead of being repaired repeatedly in place. “Immutable” describes the management approach, not a technical guarantee that the running operating system cannot change.

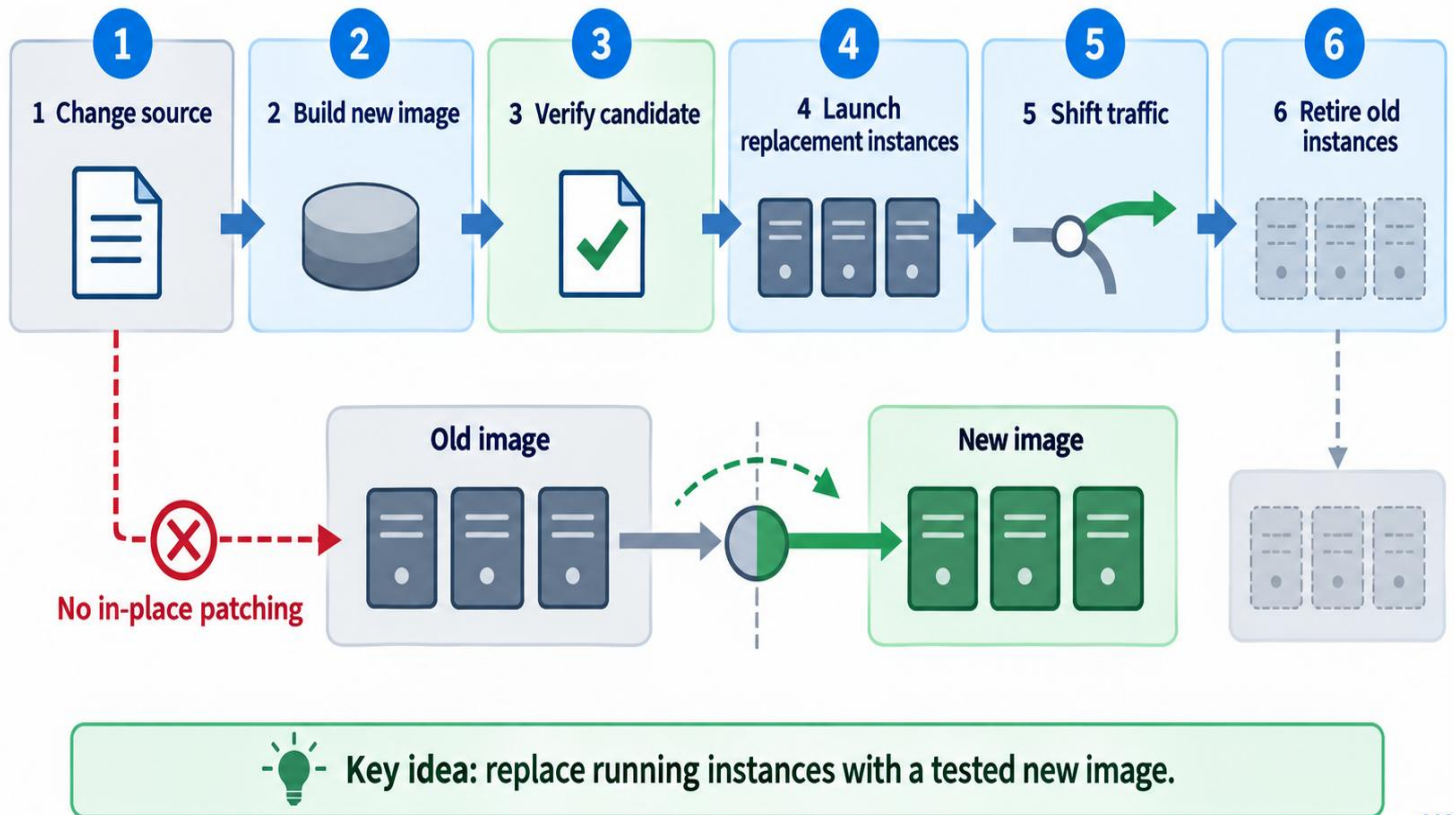
A typical update works as follows:

1. Change the Packer template or provisioning script.

2. Build a new image.
3. Test and approve that image.
4. Replace old instances with instances launched from the new image.
5. Retire the old instances after the replacement is healthy.

Chapter 1: What Packer Does and Why Teams Use It

1-3 Immutable Infrastructure Update Cycle



003

Figure 1-3: Immutable infrastructure update cycle

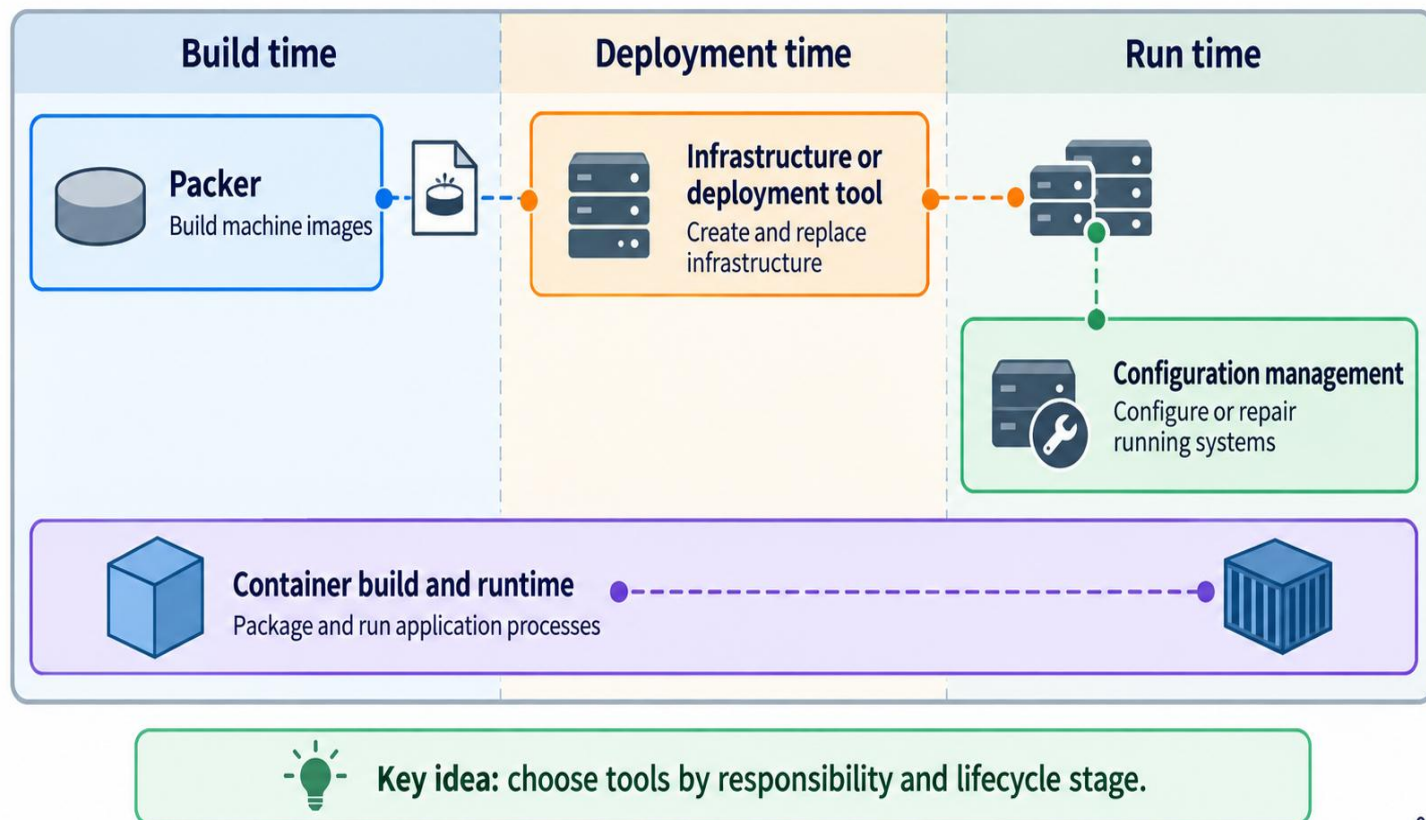
This approach reduces **configuration drift**, which is the gradual difference that develops when nominally identical machines receive different manual changes. Packer does not prevent all drift after deployment, but it provides a known and repeatable starting point.

Where Packer fits among other tools

Several infrastructure tools may participate in the same delivery process. They are complementary when each tool has a clear responsibility.

Chapter 1: What Packer Does and Why Teams Use It

1-4 Tool Responsibility Boundaries



004

Figure 1-4: Responsibility map for Packer, deployment tools, configuration management, and containers

Tool category	Primary responsibility	Relationship to Packer
Packer	Build machine-image artifacts	Produces the image that other systems consume
Infrastructure provisioning, such as Terraform	Create and change infrastructure resources	Launches instances, groups, networks, or services that reference an image
Configuration management, such as Ansible, Chef, or Puppet	Configure operating systems and applications	Can be called during a Packer build or used after deployment
Deployment or orchestration services	Release workloads and manage rollout behavior	Select and deploy an approved image
Dockerfile and container-build tools	Build container images, which package a filesystem and launch metadata, and their layers	Solve a related packaging problem, but with a container-specific model
Cloud-provider image-building services	Run image pipelines inside one provider	May replace or complement Packer when a provider-native workflow is preferred

Packer can call a configuration-management tool during a build. For example, a team can use Packer to create the temporary virtual machine and Ansible to install the application prerequisites inside it. Packer remains responsible for the image lifecycle, while Ansible remains responsible for system configuration.

Terraform and Packer are often confused because both use HCL-like configuration. Their outcomes are different:

- Packer produces an image artifact such as `ami-0123456789abcdef0`.
- Terraform can use that identifier to create running infrastructure.

Building an image and deploying an image are separate operations. Keeping them separate makes it possible to test and approve one exact artifact before it is used in an environment.

When Packer is a good fit

Packer is a strong choice when several of the following conditions are true:

- New machines should start from a standardized operating-system baseline.
- Boot-time installation is slow, unreliable, or repeated across many instances.
- The team needs reviewable and repeatable image builds.
- Several environments or platforms should follow the same general image process.
- Security patches or standard agents must be included in a controlled base image.
- Image identifiers must be handed to a separate deployment process.

Chapter 1: What Packer Does and Why Teams Use It

1-5 When Packer Is a Good Fit

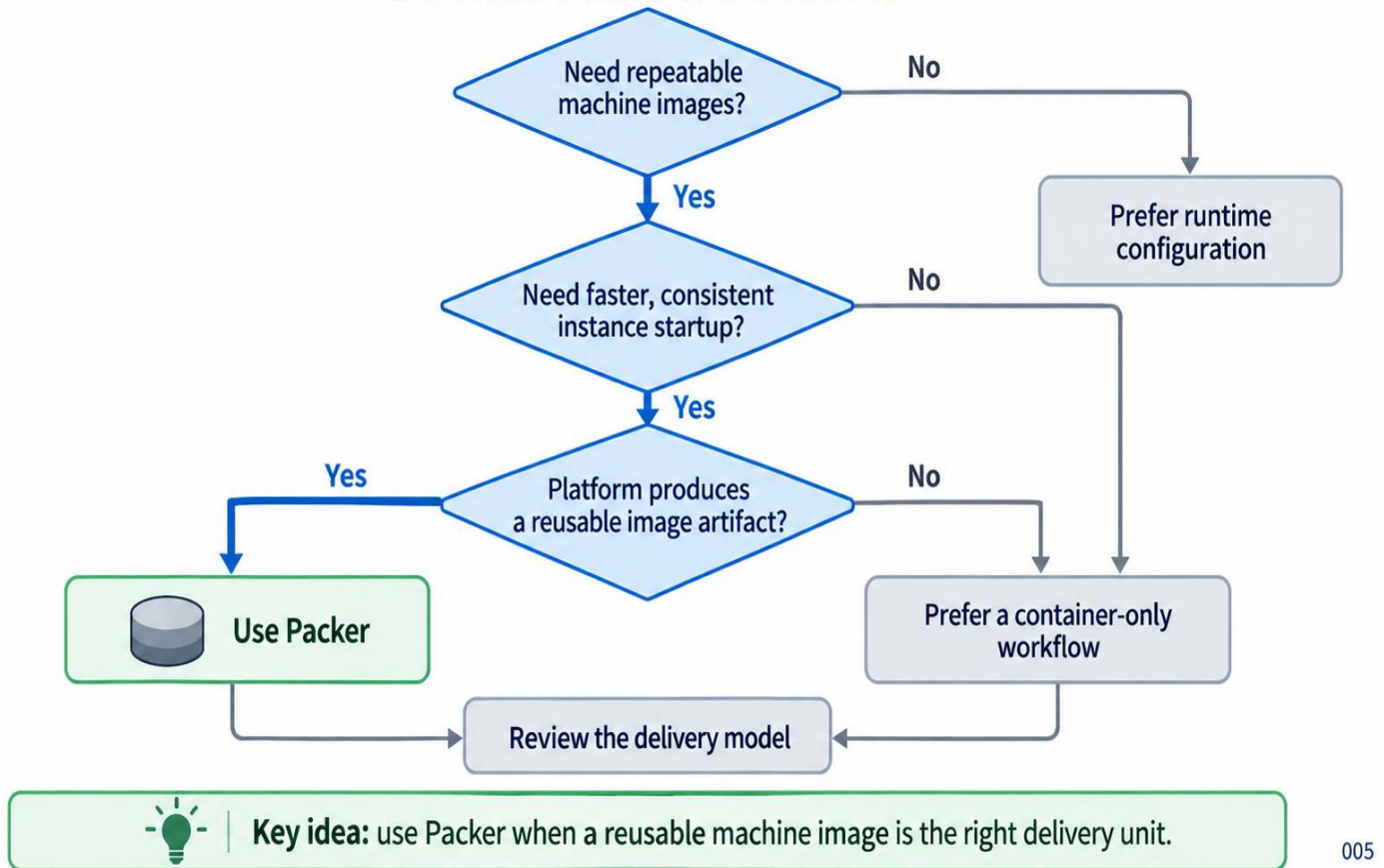


Figure 1-5: Decision path for when Packer is a good fit

Packer may be unnecessary when an unmodified provider image is sufficient, the workload is distributed only as a container, or all required configuration is intentionally performed by a managed runtime service. A very small development environment may also be simpler with startup scripts alone.

Packer is not a substitute for runtime configuration when the value belongs to a specific deployment. Examples include a node identity, a production-only endpoint, a short-lived service credential, or membership in a cluster formed after boot.

A practical team flow

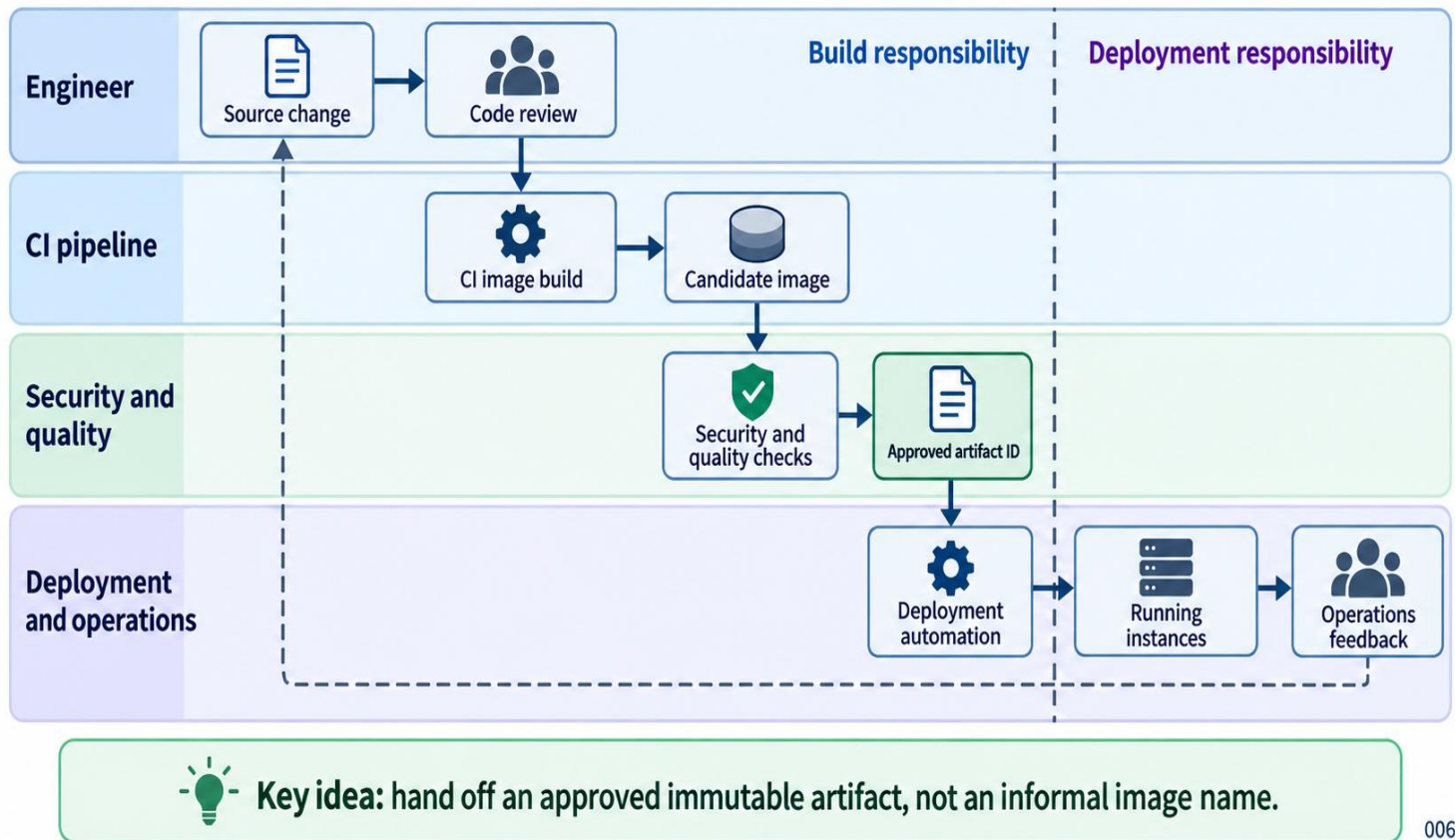
A common image-delivery flow has distinct stages:

1. The team chooses a trusted operating-system base image.
2. A Packer template defines the temporary machine and the resulting image.

3. Provisioning logic installs packages and applies common configuration.
4. The build verifies the expected software and service state.
5. Packer captures the image and returns an artifact identifier.
6. The artifact is tested and approved through the team's normal process.
7. A separate deployment system launches machines from the approved artifact.

Chapter 1: What Packer Does and Why Teams Use It

1-6 Team Image Delivery Flow



006

Figure 1-6: Team image-delivery flow from source change to deployment

The important handoff is between steps 5 and 7. Packer creates a candidate artifact. Another process decides whether, where, and how that artifact should run.

Misconceptions that cause design mistakes

“Packer manages my servers after deployment.” It does not. When a Packer process finishes, it is no longer controlling instances launched later from the image.

“Packer replaces configuration management.” It may run shell, PowerShell, Ansible, Chef, Puppet, or other provisioning logic. It coordinates the image build but does not make every configuration language unnecessary.

“A successful Packer build means the image is ready for production.” A successful build proves that the configured build steps completed and an artifact was produced. Approval, security checks, boot tests, and operational readiness are separate concerns.

“The image should contain every value the application needs.” A reusable image should contain stable software and baseline configuration. Environment-specific secrets and identities generally belong outside the image.

The foundation for all later work is therefore: **Packer creates a repeatable image artifact from declared inputs, and another system deploys that artifact.**

Chapter Summary

- Packer converts reviewed build definitions into reusable machine-image artifacts.
- Image creation, artifact approval, deployment, and runtime configuration are separate responsibilities.
- Packer is most useful when a team needs a consistent, repeatable machine baseline rather than one-off manual preparation.

Review Questions

1. What is the difference between building an image and deploying an image?
2. How does an immutable-infrastructure update reduce configuration drift?
3. When would an unmodified provider image or runtime configuration be simpler than introducing Packer?

Chapter 2: The Packer Build Lifecycle

| What You'll Be Able to Do

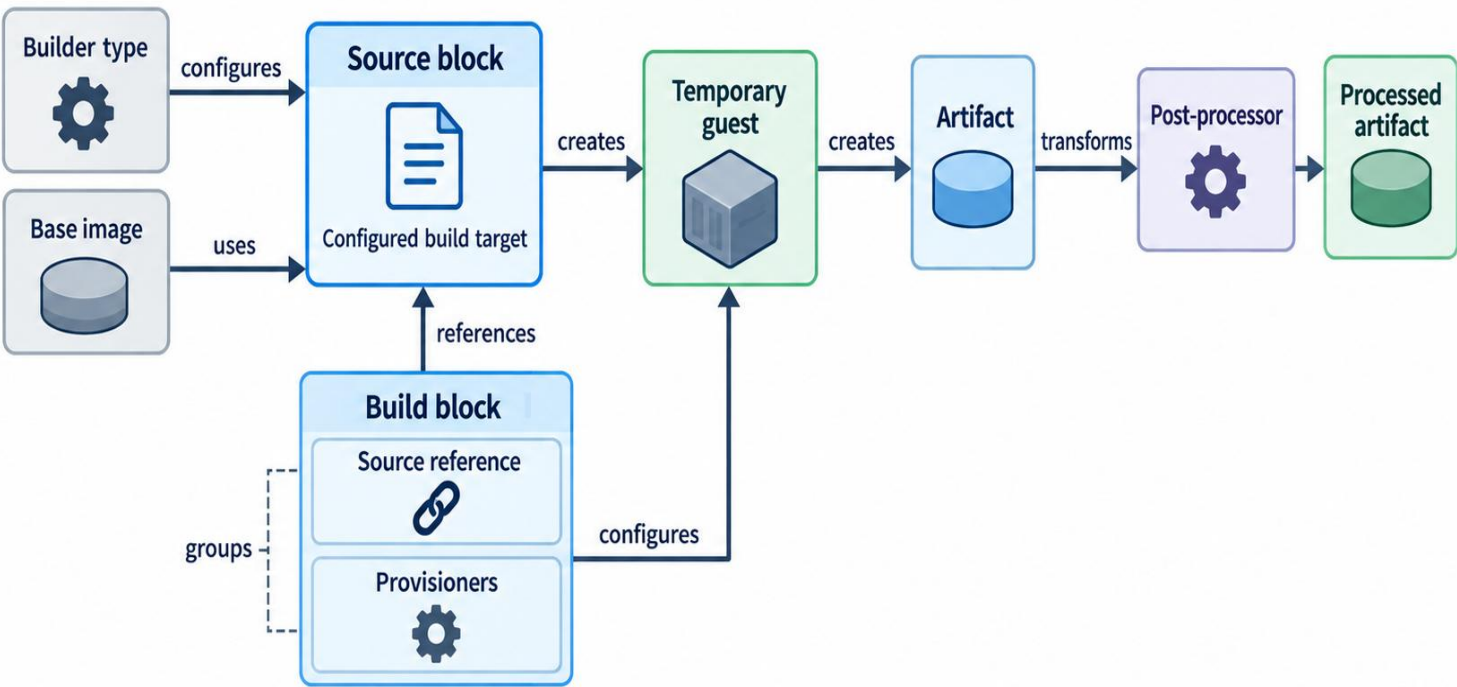
Every Packer build follows a recognizable sequence: load configuration, prepare required components, create a temporary machine, connect, provision, capture an artifact, and clean up. When a build fails, identifying the failed phase is usually more useful than reading the entire log as one undifferentiated stream.

Core vocabulary

Term	Meaning
Template	One or more HCL files that describe inputs, sources, and build steps
Plugin	An executable extension that adds platform-specific or workflow capabilities to Packer
Builder	A platform-specific component that creates a temporary build environment and produces an artifact
Source	A reusable configuration block for one builder type
Data source	A read-only query that retrieves external information for use in the template
Communicator	The connection mechanism Packer uses to upload files and run commands in the temporary machine
Provisioner	A build-time step that installs or configures software in the temporary machine
Artifact	The result of a successful builder, such as a cloud image ID or a virtual disk file
Post-processor	A step that acts after an artifact is created, such as transforming it or recording metadata, which is descriptive information about the build or artifact

Chapter 2: The Packer Build Lifecycle

2-1 Core Packer Vocabulary



Key idea: configured source plus build steps produce an artifact.

Figure 2-1: Relationship among source, builder, build, provisioner, post-processor, base image, and artifact

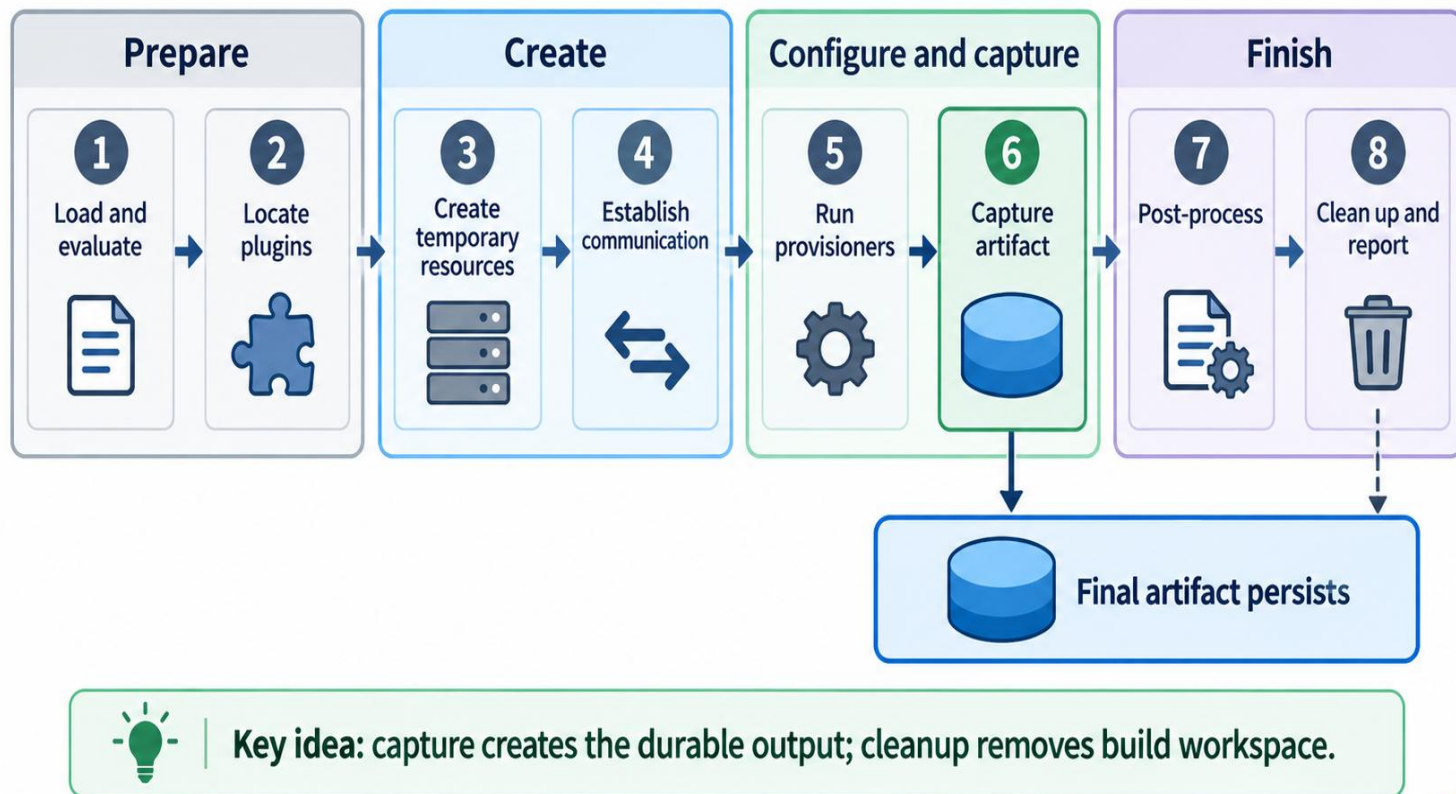
A source and a base image are not the same thing. A source block is Packer configuration. One argument inside that source may identify or search for the base image.

The lifecycle step by step

A complete build evaluates the template, locates plugins, creates a temporary machine, establishes communication, runs provisioners, captures the artifact, performs any post-processing, and cleans up temporary resources while retaining the final artifact.

Chapter 2: The Packer Build Lifecycle

2-2 Complete Packer Build Lifecycle



008

Figure 2-2: Complete Packer build lifecycle

1. Load and evaluate the template

Packer reads the selected `.pkr.hcl` files, evaluates **expressions**—HCL constructs that reference or compute values—checks required inputs, and assembles the build configuration. Errors at this stage include malformed HCL, missing required variables, invalid references, and unsupported arguments.

No temporary virtual machine should be needed merely to discover a syntax error. This is why formatting and validation belong early in the workflow.

2. Locate the required plugins

Packer determines which external plugins the template requires. A plugin supplies platform-specific behavior that the Packer core does not implement directly. For example, the Amazon plugin supplies the `amazon-efs` builder, named for Amazon Elastic Block Store (EBS), to create an AWS AMI.

If a plugin is absent or does not satisfy the declared **version constraint**, which is a rule describing acceptable versions, the build stops before it can call the target platform.

3. Ask the builder to create temporary resources

The builder contacts the target platform and prepares a temporary build environment. An AWS builder might create an **Amazon Elastic Compute Cloud (EC2) instance**, which is a cloud virtual machine, along with a temporary key pair and a **security group**, which is a firewall policy attached to cloud resources. A local virtualization builder might create a virtual disk and start a local virtual machine.

The exact resources are builder-specific, but their purpose is the same: provide a machine that can be changed and then captured.

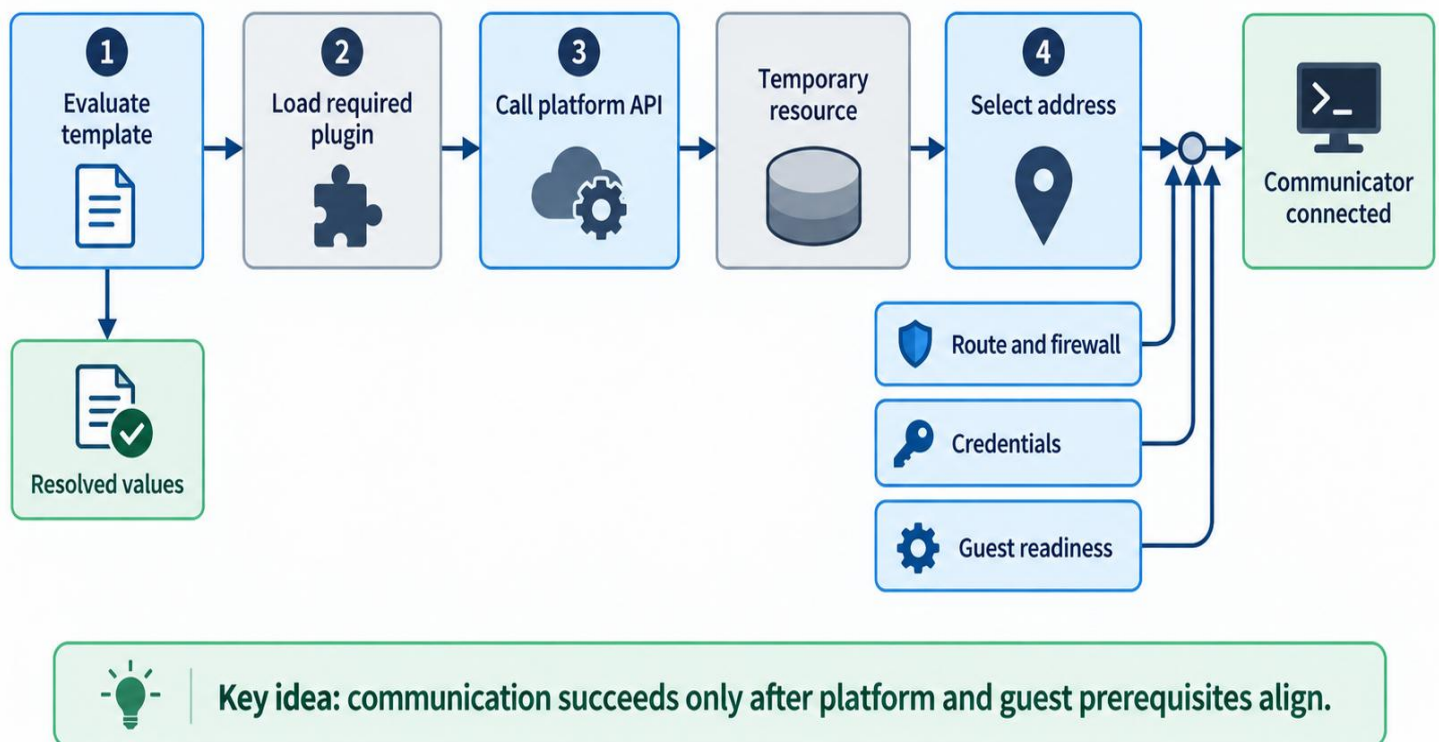
4. Wait for the guest and establish communication

Starting a virtual machine does not mean the operating system is immediately ready. Firmware, boot loaders, the kernel, initialization services, networking, and remote-access services must all become available.

The communicator then establishes a connection. **Secure Shell (SSH)** is an encrypted remote-access protocol commonly used for Linux, while **Windows Remote Management (WinRM)** is Microsoft's remote-management service for Windows. A failed connection can be caused by routing, a firewall, an incorrect port, an incorrect user, invalid credentials, or a guest that has not finished booting.

Chapter 2: The Packer Build Lifecycle

2-2 Template to Reachable Guest



009

Figure 2-3: From template evaluation to a reachable guest

5. Run provisioners in order

Provisioners upload files and execute commands inside the temporary machine. Their order matters. A configuration file cannot be installed before its destination package creates the required directory, and a service cannot be verified before it has been installed.

Packer treats a nonzero command exit status as a failure unless the provisioner is explicitly configured otherwise. Correct exit behavior is therefore part of the build contract.

6. Finalize and capture the artifact

After provisioning succeeds, the builder performs the platform-specific capture operation. A cloud builder may stop the temporary instance, create snapshots, register an image, wait for the image to become available, and apply tags. A local builder may shut down the virtual machine and retain its disk or export files.

This phase changes a configured temporary machine into a reusable artifact.

7. Perform any post-processing

A post-processor receives a completed artifact or its metadata. This stage is intentionally separate from guest configuration. It may package an artifact, calculate a **checksum**, which is a value used to detect file changes, import the artifact elsewhere, or record build information.

Post-processing is introduced here only as a lifecycle phase. Its detailed behavior belongs after the core image-building model is understood.

8. Clean up and report the result

Packer asks the builder to remove temporary resources that are no longer needed and prints the artifacts produced by successful builds. The final output normally includes a platform-specific identifier or path.

Cleanup is not the same as deleting the completed artifact. The temporary instance should disappear; the newly created image should remain.

Chapter 2: The Packer Build Lifecycle

2-2 Provision to Report and Cleanup

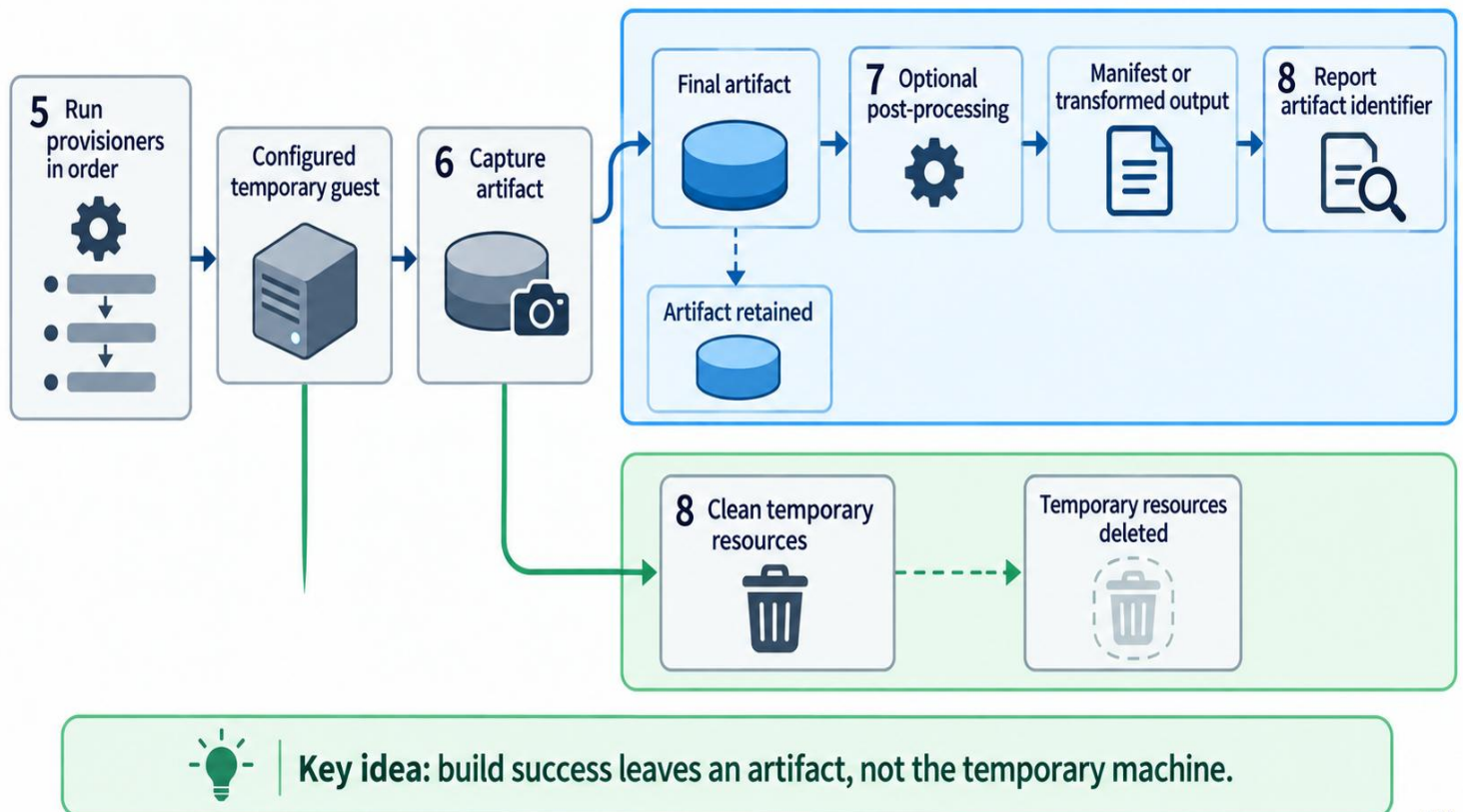
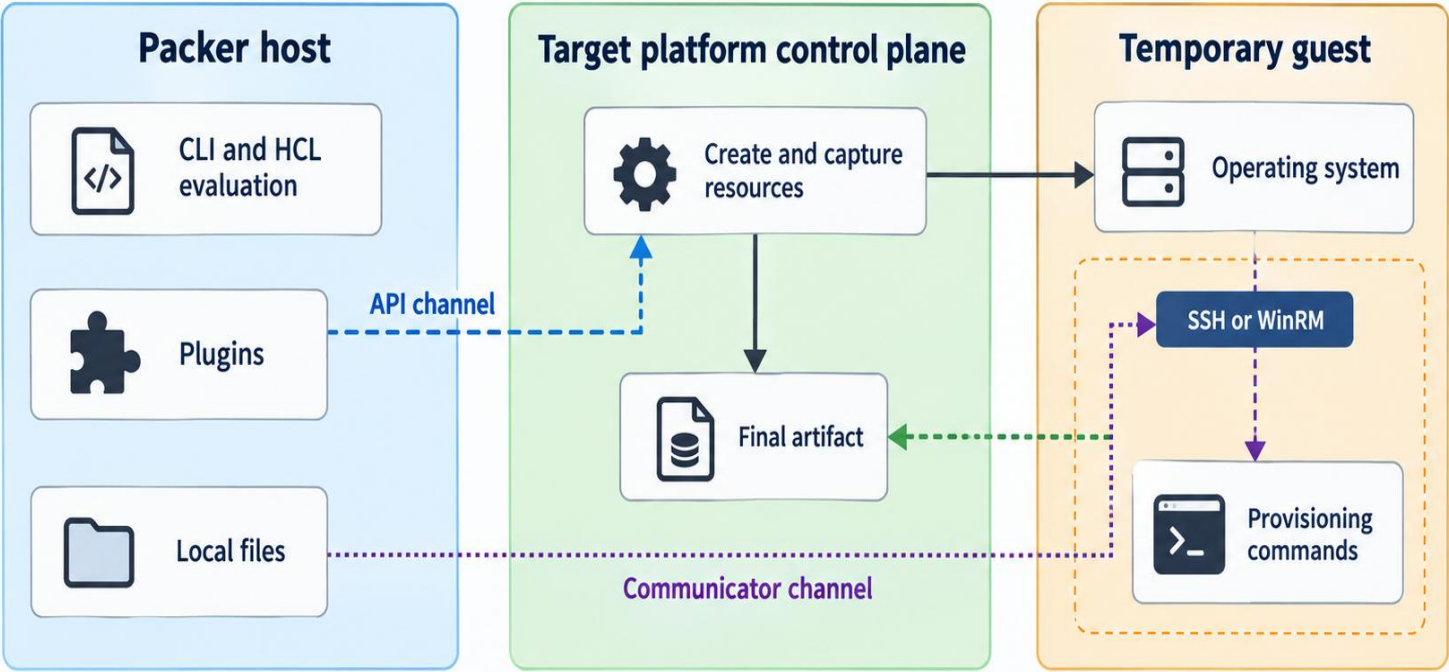


Figure 2-4: From ordered provisioning to artifact reporting and cleanup

What runs on the Packer host and what runs in the guest

The **Packer host** is the machine on which the `packer` command runs. It may be an engineer's workstation or an automation runner. The **guest** is the temporary operating system being prepared. An **application programming interface (API)** is a defined way for software such as a Packer plugin to request actions from another system.

Chapter 2: The Packer Build Lifecycle
2-3 Host, API, and Guest Boundaries



Key idea: orchestration, platform control, and guest execution are separate.

011

Figure 2-5: Packer host, target platform API, and guest execution boundary

Activity	Usually runs on the Packer host	Usually runs in the temporary guest
Parse HCL and evaluate most expressions	Yes	No
Load plugins and call cloud or hypervisor APIs	Yes	No
Open SSH or WinRM connections	Yes	Receives the connection
Upload scripts and files	Sends	Receives
Install packages and configure services	No	Yes
Execute a <code>shell-local</code> provisioner, which runs a command on the Packer host	Yes	No
Capture the machine image	Builder coordinates from the host	Guest is shut down or captured as required

This distinction explains many confusing failures. If `curl` works inside the guest but the Packer host cannot reach the guest's SSH address, provisioning still cannot begin. Conversely, the Packer host may reach the guest while the guest cannot reach a package repository.

Following one hypothetical Linux build

Assume a template defines an Ubuntu web-server image for a cloud platform.

1. Packer loads the HCL and resolves the region, base-image selection, instance type, and image name.
2. The cloud plugin authenticates to the provider API.
3. The builder launches a temporary instance from the selected Ubuntu image.
4. The builder determines the address to use and the SSH communicator logs in as the image's default user.
5. A shell provisioner waits for operating-system initialization, installs a web server, enables it, and verifies its configuration.
6. The builder stops the temporary instance and asks the platform to create the new image.
7. The platform returns an image identifier.
8. Packer terminates the temporary instance and removes temporary access resources.
9. The final log reports the new image identifier.

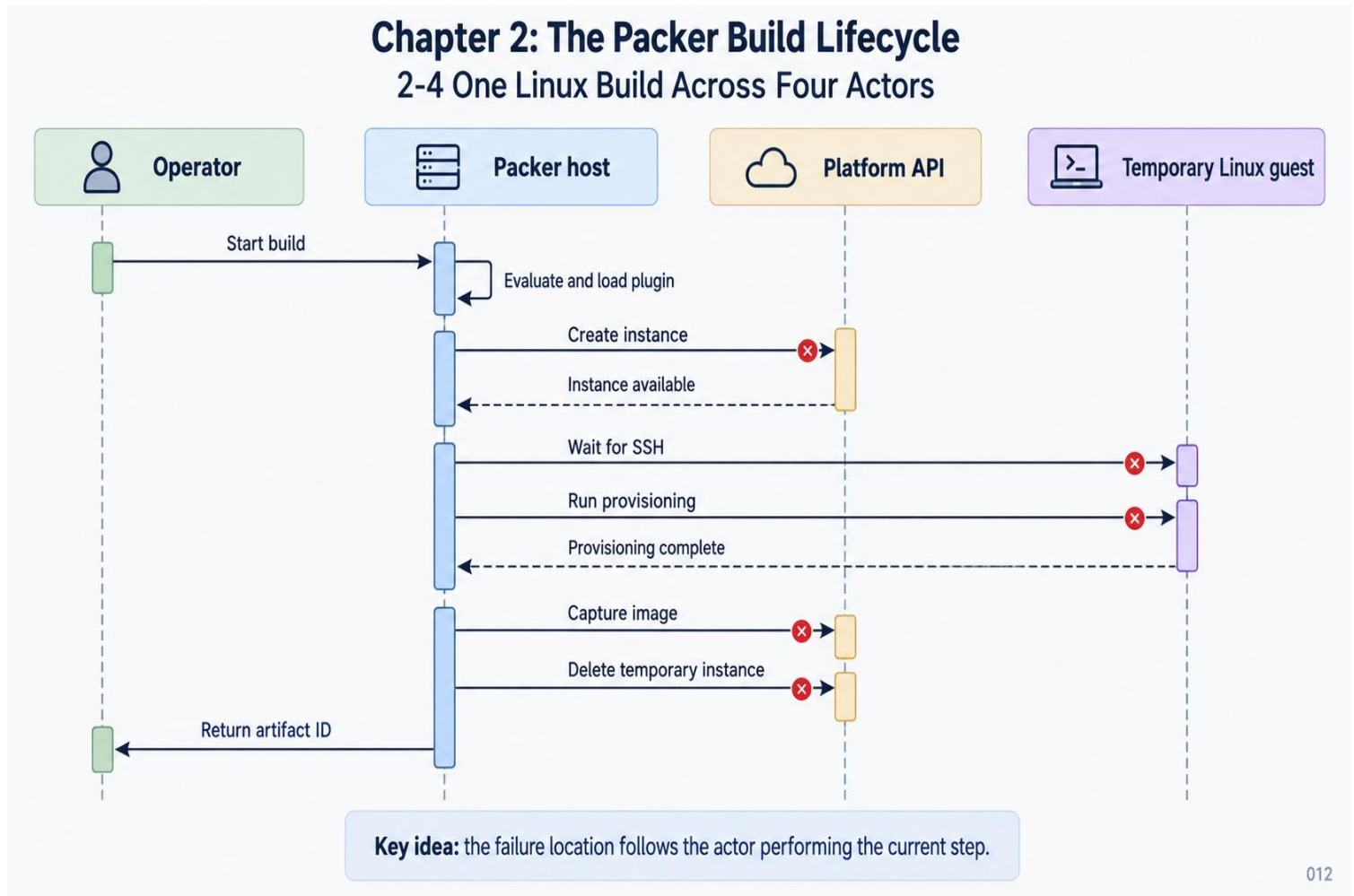


Figure 2-6: One hypothetical Linux build with component ownership at each step

At each step, a different component has primary responsibility. A malformed package command is not a builder problem. A rejected API request is not an SSH problem. A timeout before SSH connects is not evidence that the installation script is wrong.

Reading logs by phase

Packer prefixes log lines with the active build or source name. During a single-source build, these prefixes show which component is speaking. During a multi-source build, they also separate interleaved output.

Look for transition messages rather than starting with individual command lines:

- Template and plugin initialization
- Temporary resource creation
- "Waiting for SSH" or "Waiting for WinRM"
- Provisioner start and command output
- Image or snapshot creation
- Artifact registration

- Resource cleanup
- Final artifact summary

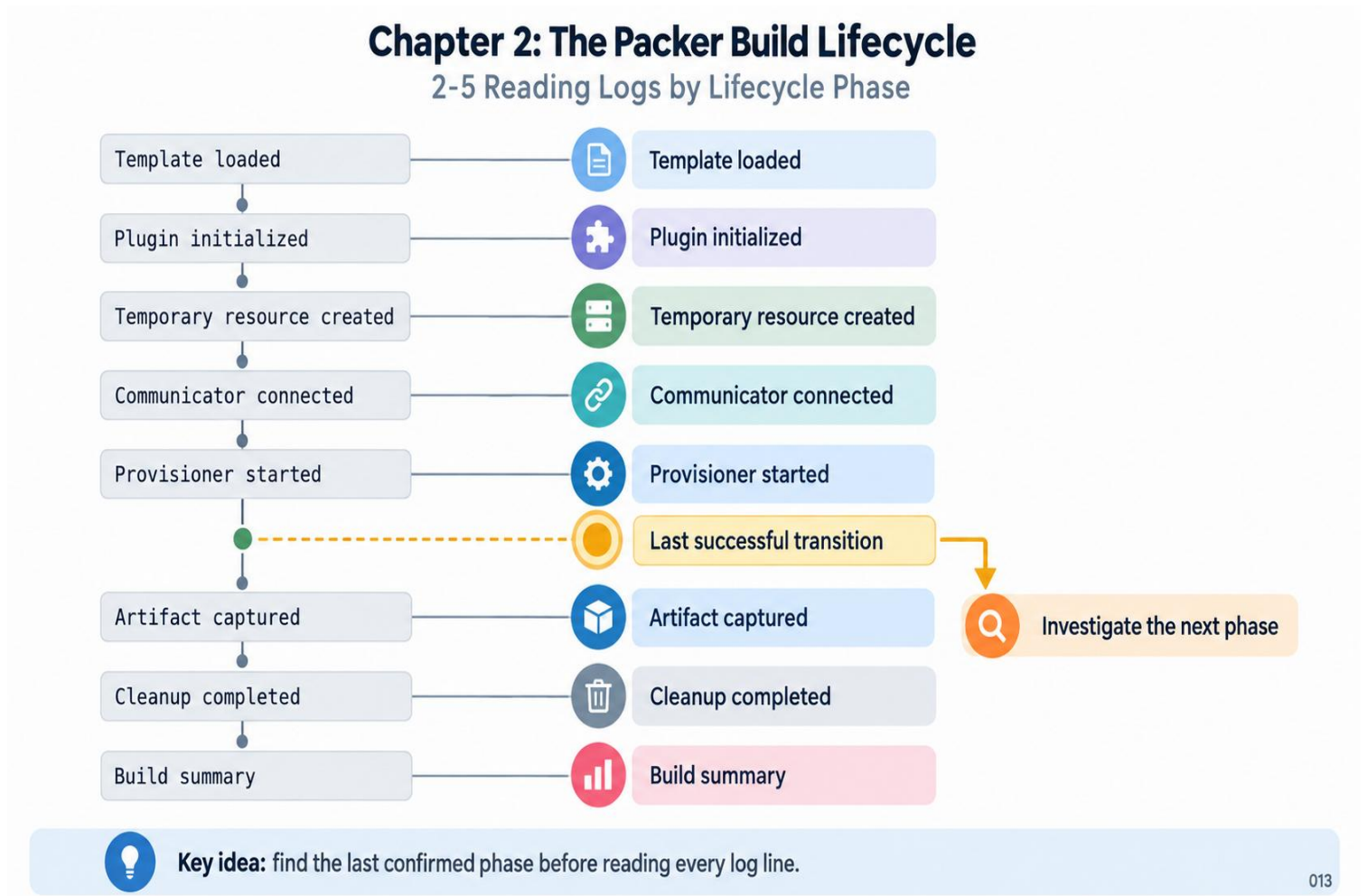


Figure 2-7: Mapping log transitions to lifecycle phases

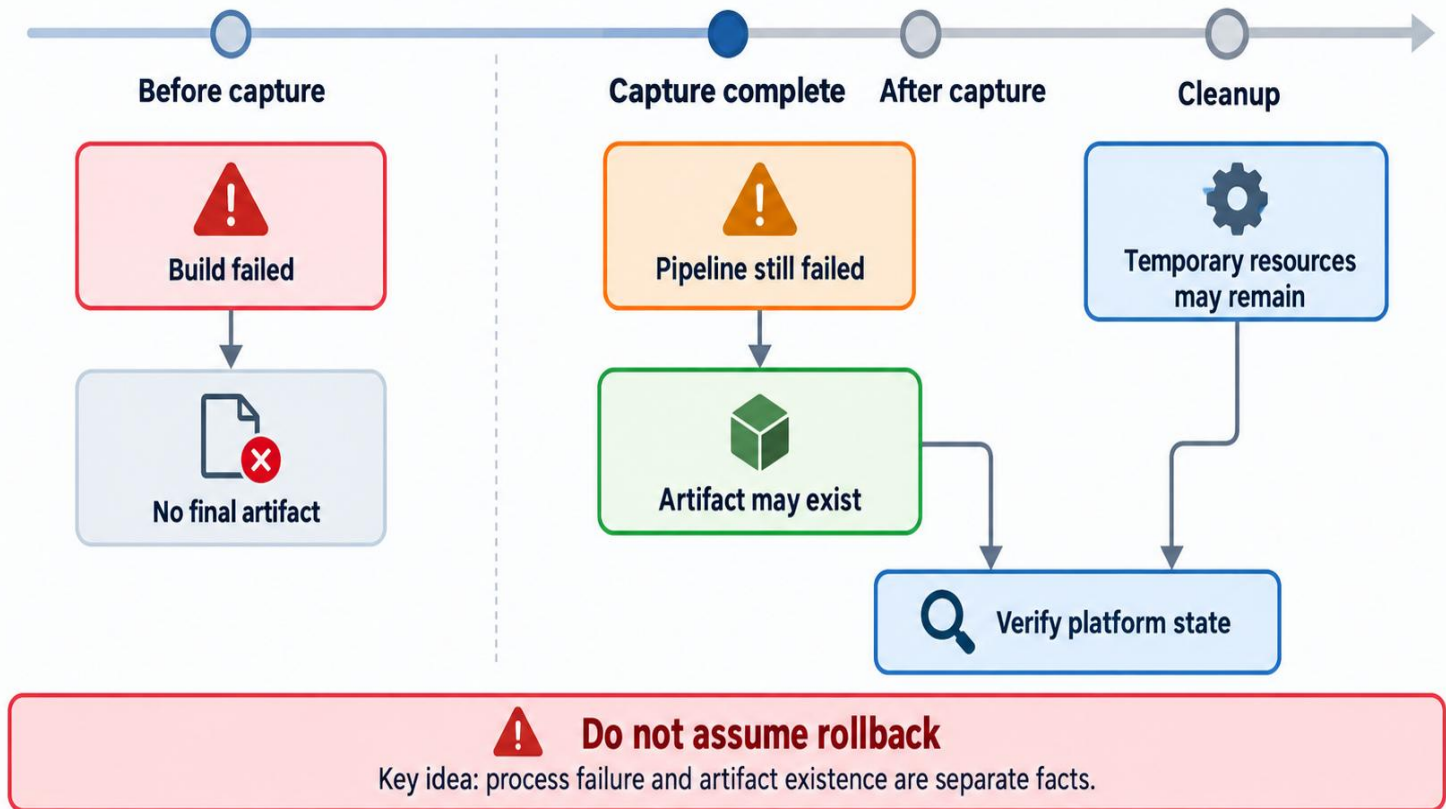
The last successful transition is a strong clue. If the log never reaches “Waiting for SSH,” the problem occurred before communication. If SSH succeeds and the first package command fails, the network path from Packer to the guest was already proven.

Failure boundaries

Failed phase	Typical symptoms	Primary area to inspect
Configuration loading	Parse errors, unknown variables, invalid references	HCL and input values
Plugin preparation	Missing plugin, incompatible version	<code>required_plugins</code> and initialization
Infrastructure creation	API denial, quota error, invalid network ID	Provider credentials and platform settings
Communication	SSH or WinRM timeout, authentication failure	Addressing, routes, firewalls, user, credentials, guest readiness
Provisioning	Nonzero exit status, missing file, permission error	Script logic, privilege, package state, ordering
Artifact capture	Name collision, snapshot or registration error	Builder arguments and platform image rules
Post-processing	Artifact exists but later step fails	Post-processing configuration and external destination
Cleanup	Temporary resources remain	Interrupted execution, permissions, provider state

Chapter 2: The Packer Build Lifecycle

2-6 Failure Boundaries and Build State



014

Figure 2-8: Failure boundaries and resulting build state

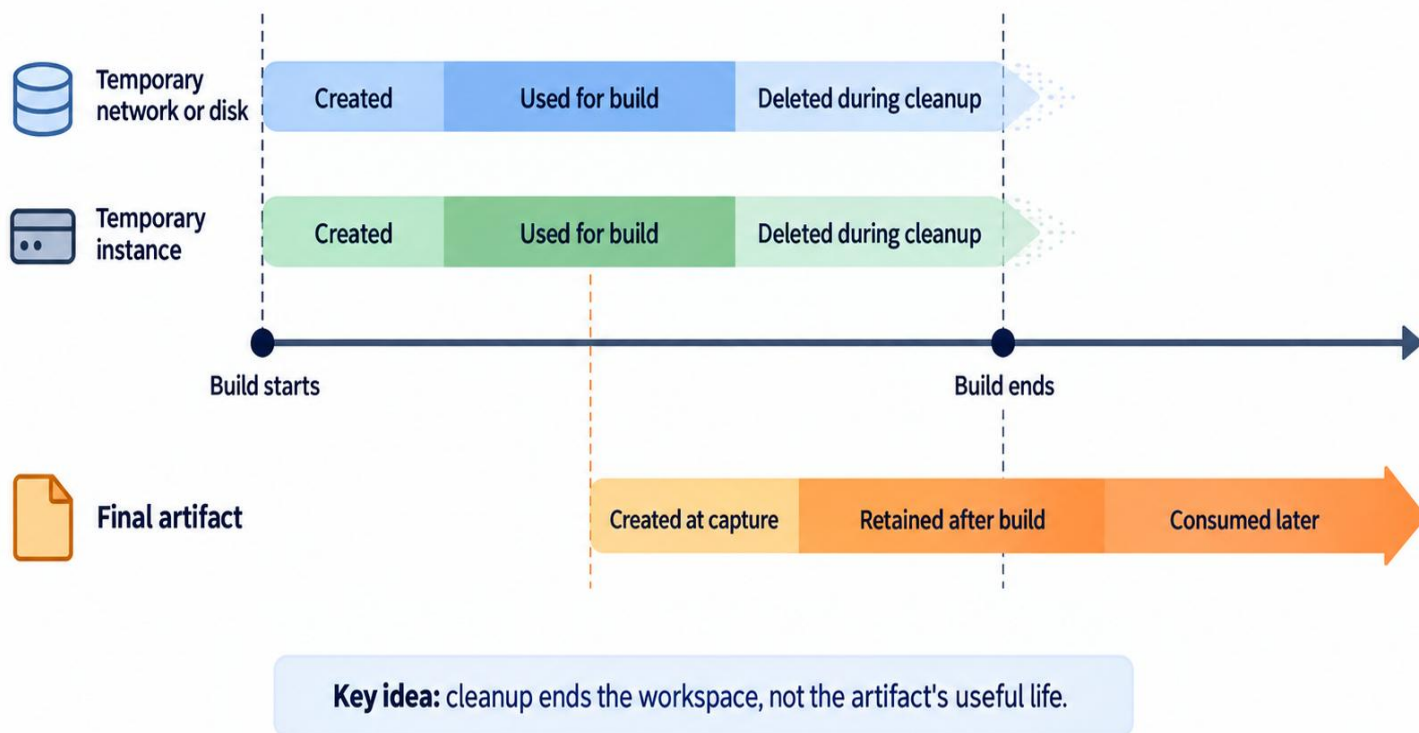
A build may fail after an artifact has already been created. Therefore, a failure message does not always mean “nothing exists,” and a retry should not be started blindly. Before retrying, determine whether the platform now contains an image, snapshot, temporary instance, or conflicting name from the previous attempt.

Temporary resources and final resources

A successful cloud build commonly leaves the intended image and its backing storage while deleting the build instance and temporary access resources. A local virtualization build commonly leaves the exported disk or virtual-machine files while deleting transient runtime state.

Chapter 2: The Packer Build Lifecycle

2-7 Resource Lifetimes



015

Figure 2-9: Temporary resource lifetime versus final artifact lifetime

Interrupted builds deserve special attention. Packer normally attempts cleanup, but the host can lose power, the process can be killed, or the provider API can reject a cleanup request. The engineer must know which resource names and tags identify the build's temporary objects.

The lifecycle can be summarized as **prepare, connect, change, capture, and clean**. That sequence becomes the map for every template and every log.

Chapter Summary

- A Packer build moves through identifiable phases from configuration loading to artifact reporting and cleanup.
- The Packer host, platform API, and temporary guest perform different parts of the work.
- Locating the last successful phase is the fastest way to narrow a failure and understand which resources may remain.

Review Questions

1. Which lifecycle phases occur before Packer can run a provisioner inside the guest?
2. What work happens on the Packer host, and what work happens inside the temporary guest?
3. Why can a failed command still leave temporary resources or even a completed artifact?