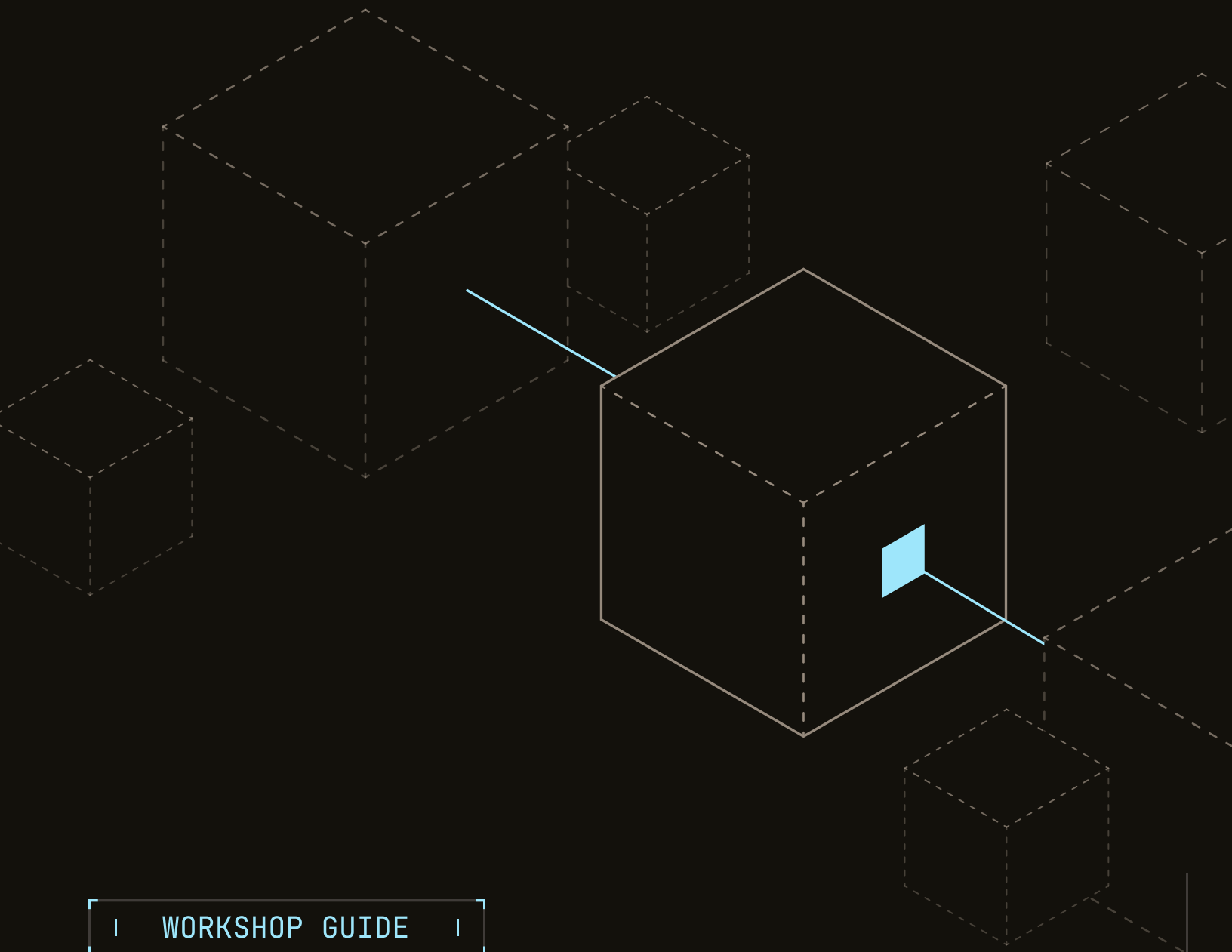
StackGen



WORKSHOP GUIDE

How to Accelerate Infrastructure Provisioning for AI-Native Enterprises

Table of Contents

What You'll Build 2

Step 1: Create Your Project and Authenticate 3

Step 2: Publish Terraform Modules to the Catalog 4

Step 3: Set Up a Governance Policy That Enforces Itself 6

Step 4: Build the Golden Template AppStack 7

Step 5: Convert to a Reusable Template 7

Step 6: Enable AI-Driven Provisioning 8

Step 7: Deploy and Verify 9

What Platform Engineers Get From This Setup 9

How This Compares to Other Approaches 10

Key Takeaways 10

Going Further: Onboard Existing Terraform Projects 11

Further Reading 12

Most platform teams are running the same playbook they wrote three years ago: developer opens a ticket, platform engineer interprets it, hand-crafts Terraform, waits for a review, deploys. Repeat for every environment, every service, every team. It worked when you had five services. It doesn't work when you have fifty.

The teams moving away from that model aren't doing it because self-service is a cool idea. They're doing it because platform engineers were spending 60–70%+ of their sprint capacity on interrupt-driven fulfillment requests instead of building the platform itself.

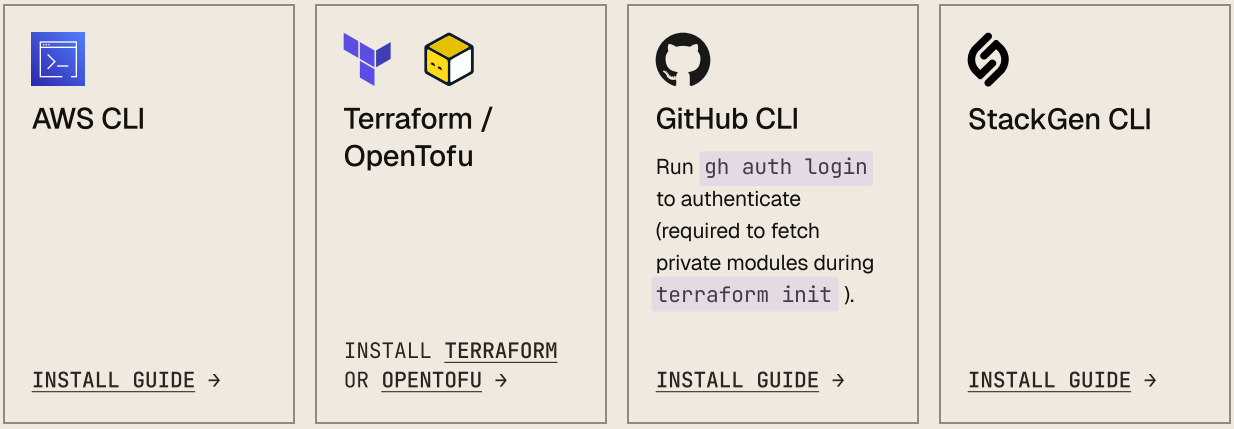
If you've spent a Friday untangling a state file conflict while three developers wait on their environments — or explained to your security team why a policy violation made it to prod — you'll appreciate this setup: a self-service AWS infrastructure workflow built with StackGen, combining a published Terraform module catalog, self-enforcing governance policies, a golden-path AppStack template, and an AI layer that generates environments from plain English. The whole setup takes under 15 minutes.

60–70%+	<15 min	90 sec
spent on interrupt-driven fulfillment requests	modules, governance, golden path template, AI skill	what used to take 3 days and a slack thread

What You'll Build

COMPONENT	WHAT IT DOES
Terraform module catalog	App Runner, DynamoDB, ECR, Secrets Manager — published once, reused by every developer
Governance policy	Security rule enforced at plan time — before apply runs, before anything touches production
Golden-path AppStack template	A containerized app pattern developers instantiate with variables, not tickets
AI Skill (Aiden AI Copilot)	Generates a fully-configured AppStack from a plain-English developer request

Prerequisites:



// VERIFY IF ALL TOOLS ARE INSTALLED:

```
terraform --version
stackgen version
gh --version
aws sts get-caller-identity
```

Step 1: Create Your Project and Authenticate

Log in to your StackGen instance, open **Enterprise Configuration → Settings → All Projects**, and create a new project. Use a consistent naming convention — your GitHub handle works well (e.g., `jdoe-project`). This project is your isolated workspace — all AppStacks, modules, and governance policies live here.

> Generate a **Personal Access Token (PAT)** from Settings → Personal Access Tokens. Save it immediately — you won't see it again.

// CONFIGURE THE STACKGEN CLI:

```
export STACKGEN_TOKEN=<YOUR_PERSONAL_ACCESS_TOKEN>
export STACKGEN_URL="https://your-instance.cloud.stackgen.com"

stackgen login \
 --url https://your-instance.cloud.stackgen.com \
 --project your-project-name
```

// CONFIRM AUTHENTICATION

```
stackgen appstack list
# Returns an empty list for a new project — that's correct
```

Step 2: Publish Terraform Modules to the Catalog

This is where the self-service foundation gets built. Instead of a platform engineer fielding requests and writing Terraform per team, you publish approved modules once, and developers drag them onto a canvas. No Terraform knowledge required on their end. No context-switching required on yours.

// PUBLISH FOUR MODULES FOR A STANDARD CONTAINERIZED AWS APPLICATION:

```
# App Runner — serverless container compute
stackgen upload custom-modules \
 --scope project \
 --name app_runner \
 --provider aws \
 --ref https://github.com/your-org/aws-modules//modules/app_runner@main \
 --version v1.0.0

# DynamoDB — NoSQL application data layer
stackgen upload custom-modules \
 --scope project \
 --name dynamodb_table \
 --provider aws \
 --ref https://github.com/your-org/aws-modules//modules/dynamodb_table@main \
 --version v1.0.0

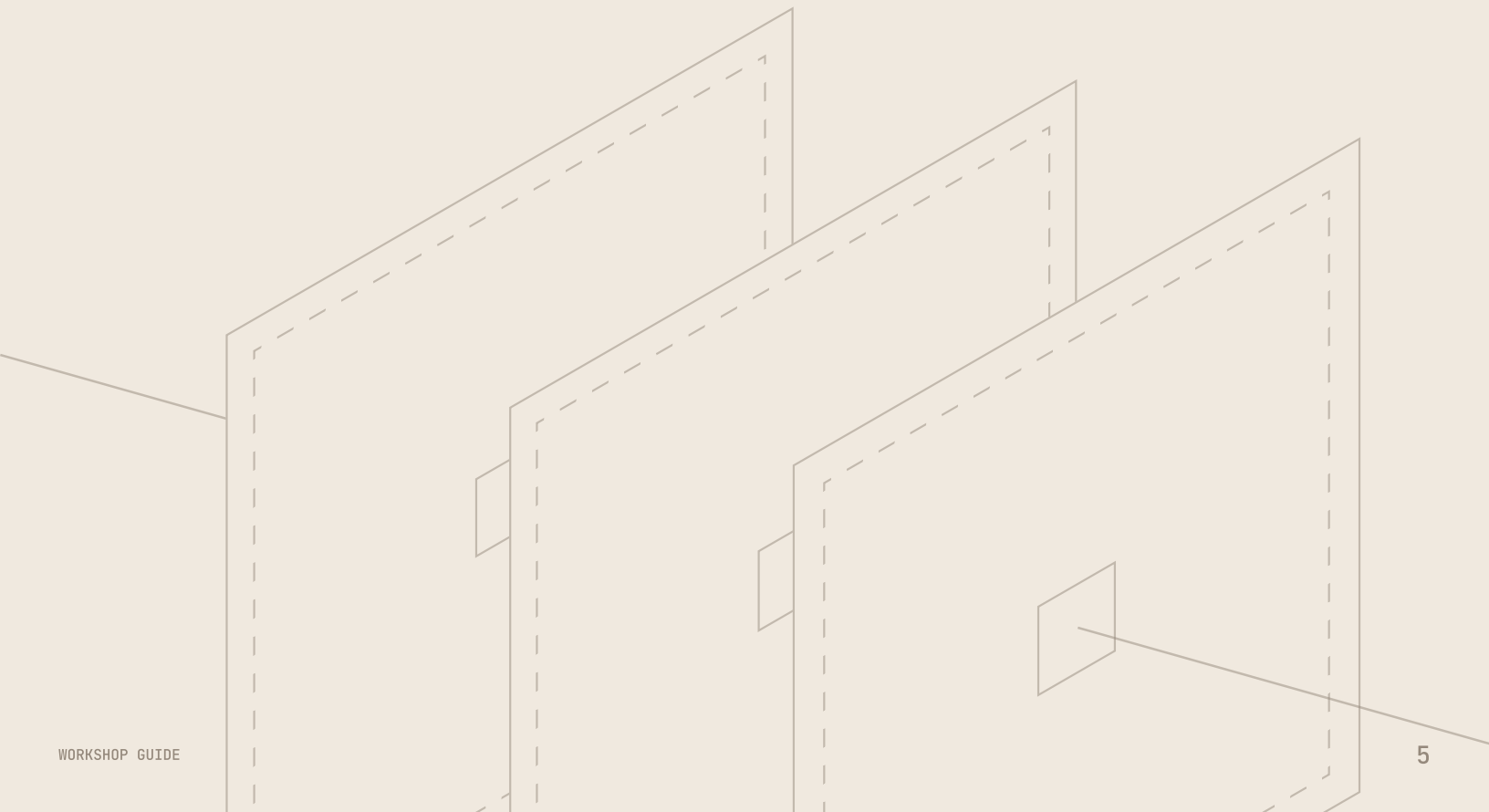
# ECR — container image registry
stackgen upload custom-modules \
 --scope project \
 --name ecr_repository \
 --provider aws \
 --ref https://github.com/your-org/aws-modules//modules/ecr_repository@main \
 --version v1.0.0
```

```
# Secrets Manager — secure credential storage
stackgen upload custom-modules \
 --scope project \
 --name secrets_manager \
 --provider aws \
 --ref https://github.com/your-org/aws-modules//
modules/secrets_manager@main \
 --version v1.0.0
```

Once uploaded, these modules appear in the StackGen InfraComposer catalog — immediately available to any developer in your project.

AUTOMATE MODULE PUBLISHING

Add `STACKGEN_URL` and `STACKGEN_TOKEN` as GitHub Actions secrets in your module repository. Every merge to main automatically publishes the updated version. One less manual step in the workflow.



Step 3: Set Up a Governance Policy That Enforces Itself

The usual failure mode with self-service is governance: you give developers the keys, someone spins up a DynamoDB table with `PROVISIONED` billing mode, and now you own that cost overrun in the next budget review. StackGen governance policies run at plan time — the deployment fails before it ever reaches production.

Navigate to **Enterprise Configuration → Governance**, create a new configuration, and select AWS services. At Step 4 (Select Security Policies), click + **Add New Policy** and describe the rule in plain English:

> "Enforce that all AWS DynamoDB tables use PAY_PER_REQUEST billing mode."
> StackGen's AI generates the policy structure from that description.

CRITICAL

Check the `resource_type` field in the generated rego policy (typically line 6). It must exactly match the Terraform resource type in your module — for example, `aws_dynamodb_table` . A mismatch means the policy won't evaluate against your actual resources.

// ADD THE AWS PROVIDER BLOCK:

```
terraform {
 required_providers {
 aws = {
 source = "hashicorp/aws"
 version = "~> 5.0"
 }
 }
}

provider "aws" {
 region = "us-east-1"
}
```

Scope the policy to your project in the "Enforce on projects" dropdown, then save. Every AppStack deployment in this project is now validated against this policy before apply runs.

Step 4: Build the Golden Template AppStack

A golden template AppStack is the infrastructure pattern you want developers to use by default — the one that already has the right compute, data, and secrets layers wired together. Developers instantiate it with their own variable values; the underlying architecture doesn't change.

In the StackGen UI, create a new AppStack, select **AWS**, and name it descriptively (`your-name-golden-path`). Using the InfraComposer canvas, drag the following modules from the catalog:

- **App Runner** — containerized compute layer
- **DynamoDB** — application data store
- **Secrets Manager** — API keys and credentials

Name resources consistently (`workshop-app` , `workshop-table` , `workshop-secret`) and save.

> IDE SHORTCUT
>
> If you've installed the **StackGen MCP** server in VS Code, Cursor, or Claude Code, describe the AppStack in your IDE chat — Aiden composes the canvas for you without leaving the editor.

// RUN A PLAN TO VALIDATE THE APPSTACK BEFORE CONVERTING IT TO A TEMPLATE:

```
stackgen provision \
 --appstack-id {your-appstack-id} \
 --project {your-project-id} \
 --environment prod
# Dry run only — no resources created yet
```

Step 5: Convert to a Reusable Template

This is the step that eliminates the ticket queue. Templates are golden path AppStacks with input variables — developers instantiate one with their own values and get a unique environment that inherits all your architectural standards automatically.

1. Click the AppStack name (top-left), check "**Mark as Template**", and save.
2. Click **Add New → Terraform Config → Variables** and create input variables: `app_name` , `table_name` , `secret_name` .
3. Link each variable to its module input: click the (x) icon next to any field and reference `var.app_name` .

Every environment derived from this template now inherits the **same architecture, governance policy, and security posture** — only the variable values differ between teams.

Step 6: Enable AI-Driven Provisioning

The last piece is the natural language layer. Developers who don't know which modules to pick, or don't want to navigate a canvas, can **describe what they need in plain English** — and get a provisioned AppStack back.

Navigate to the StackGen AI console (`your-instance.cloud.stackgen.com/ai/`), create a workspace attached to your project, and connect the StackGen integration with your instance URL and PAT.

Under **Skills**, create a new skill:

SKILL NAME	DESCRIPTION
Containerized Infra Generation	Use this when devs request infra for new containerized apps

SKILL PROMPT

```
When a developer requests infrastructure for a new containerized app, create an
AppStack with a randomly generated name (AWS provider) in the StackGen project
called {your-project-name}. Populate the AppStack using the template called
{your-template-name}. Check the required Terraform input variables and fill
them in with reasonable values for the developer team. Summarize what was
created and provide a URL link to view the new AppStack.
```

Save the skill. Developers can now type:

> "Create infrastructure for my new payment service."

Aiden generates a **fully-configured, governance-compliant AppStack** and hands back a direct link. No ticket, no platform engineer in the loop, no Terraform required. What used to take 3 days and a Slack thread now takes 90 seconds.

NOTE

In environments where ticketing is still required, **Aiden can directly integrate with ServiceNow** to process developer requests.

Step 7: Deploy and Verify

// WITH THE PLAN REVIEWED, DEPLOY:

```
stackgen provision \
 --appstack-id {your-appstack-id} \
 --project {your-project-id} \
 --environment prod \
 --apply
```

// APP RUNNER PROVISIONING TYPICALLY COMPLETES IN 3–5 MINUTES.
TO DESTROY THE ENVIRONMENT:

```
stackgen destroy \
 --appstack-id {your-appstack-id} \
 --environment prod \
 --apply
```

What Platform Engineers Get From This Setup

Once this is running, the platform team's relationship to infrastructure changes. You stop being a fulfillment queue and start being a standards organization.

PLATFORM ENGINEERS	**Building standards** Expand the module catalog with additional golden paths — classic VMs, data platforms, web apps with SQL. Use terraform-importer to onboard existing Terraform projects without rewrites (see Going Further below). Every new pattern you publish becomes immediately available to every developer in the project.
DEVOPS ENGINEERS	**Managing deployments** Connect the StackGen MCP server to VS Code or Cursor so developers never leave their editor to request infrastructure. Set up GitHub Actions for automatic module publishing on every merge.
ENGINEERING LEADERS	**Tracking compliance** StackGen governance policies enforce standards at plan time across every AppStack in your project. Audit trails show which template generated each environment and when — useful when a compliance question surfaces at renewal time.

How This Compares to Other Approaches

APPROACH	TIME TO FIRST ENVIRONMENT	GOVERNANCE	DEVELOPER EXPERIENCE
Manual Terraform	Hours–days	Manual review	CLI only, high expertise required
Backstage + Terraform	Days to set up, minutes at runtime	Template-level	Portal UI, moderate expertise required
StackGen AppStacks	Set up in ~15 min, minutes at runtime	Policy-enforced at plan time	Canvas UI + natural language via Aiden

> ### The distinction that matters most in practice
>
> Governance enforced **at plan time (before apply)** versus governance enforced through review processes (after the code is written, before the PR merges). For teams that have been burned by audit findings on manually-deployed infrastructure, that's the difference that drives the decision.

Key Takeaways

- **StackGen's AppStack Templates** let platform teams encode golden paths once — developers instantiate them with variables, getting environments that inherit architecture and security posture automatically.
- **StackGen Governance Policies** are generated from natural language and enforced at plan time, blocking non-compliant infrastructure before it reaches production.
- **Aiden AI Copilot** generates fully-configured, policy-compliant AppStacks from a plain-English developer request — no Terraform knowledge required.
- The full setup — module publishing, governance, golden path template, AI skill — takes under 15 minutes.

Going Further: Onboard Existing Terraform Projects

If your team already has Terraform projects created from the community version of Terraform, you don't need to rewrite them. The `terraform-importer` CLI reads an existing Terraform source directory and creates a StackGen AppStack from it automatically — no manual canvas work required.

// CLONE THE WORKSHOP EXAMPLES REPO TO TRY IT:

```
git clone https://github.com/your-org/aws-workshop-modules
```

// THEN IMPORT ONE OF THE EXAMPLE PROJECTS:

```
# Import an API backend project
terraform-importer import appstack -d \
 --project {your-project-uuid} \
 --input-tf-source-dir ./examples/api_backend \
 --appstack-cloud-provider aws \
 --appstack-name "api-backend"

# Import a classic VM project
terraform-importer import appstack -d \
 --project {your-project-uuid} \
 --input-tf-source-dir ./examples/classic_vm \
 --appstack-cloud-provider aws \
 --appstack-name "classic-vm"
```

NO CLI REQUIRED

If you have the StackGen MCP server installed in VS Code (GitHub Copilot), Cursor, or Claude Code, just describe the import in your IDE chat, and Aiden takes care of the entire process.

Once imported, the AppStack lives in your project catalog alongside everything you built in the steps above. From there, you can **mark it as a template, attach governance policies, and make it self-serviceable** for other teams — the same workflow, applied to code that already exists.

Further Reading

StackGen CLI Reference
Full command reference for stackgen provision, upload, and destroy

DOCS →

InfraComposer Canvas Guide
Visual infrastructure composition

DOCS →

What Is Infrastructure from Code?
StackGen's approach to intent-driven infrastructure provisioning

BLOG →

Aiden AI Copilot for Platform Engineering
How AI agents accelerate infrastructure delivery

BLOG →

Ready to build your self-service platform?

Get started with StackGen or book some time with our team.
We're happy to walk through your setup.

Get started →