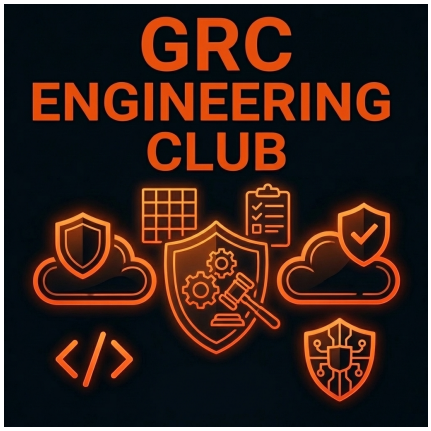

CGE-P

Study Guide

Certified GRC Engineer — Practitioner

GRC Engineering Club Training Academy

Version 1.0 | 2026

www.grcengclub.com

HOW TO USE THIS STUDY GUIDE

Overview of the Certification

The Certified GRC Engineer — Practitioner (CGE-P) certification validates your ability to design, implement, and maintain governance, risk, and compliance engineering solutions. This certification bridges traditional GRC practices with modern cloud-native technologies, emphasizing Infrastructure as Code (IaC), Policy-as-Code (PaC), and continuous authorization frameworks.

Who Should Take This Exam

This certification is ideal for cloud engineers, DevOps practitioners, compliance engineers, security architects, and GRC professionals looking to modernize their compliance practices. No prior GRC certification is required.

Prerequisites

No formal prerequisites are required. However, familiarity with basic cloud concepts, infrastructure fundamentals, and general security principles will enhance your learning experience. Some hands-on experience with infrastructure tools (Terraform, Kubernetes, cloud platforms) is beneficial but not mandatory.

Recommended Study Timeline

Timeline	Activities	Estimated Hours
Week 1	Domains 1–2 (Foundations, IaC)	6–8 hours
Week 2	Domains 3–4 (PaC, CI/CD)	6–8 hours
Week 3	Domains 5–6 (Cloud-Native, OSCAL)	6–8 hours
Week 4	Domain 7, Practice Exam, Portfolio Review	6–8 hours

How to Use This Guide with the Video Course

This study guide complements the CGE-P video course. Use the guide as your primary reference document, working through each domain section in order. Watch the corresponding video lectures, then return to this guide for review and practice questions. The practice exam and portfolio guide provide final preparation before the certification exam.

EXAM OVERVIEW

Exam Format

Attribute	Details
Total Questions	60 questions
Question Types	50 Multiple Choice + 10 Scenario-Based
Time Limit	90 minutes
Pass Score	72% (43/60 correct answers)
Passing Score Points	43 out of 60 questions
Retake Policy	You may retake the exam after 14 days
Format	Computer-based (online proctored)

Domain Breakdown and Question Distribution

Domain	Topic	Weight	Questions	Study Focus
1	GRC Foundations	15%	~9	Maturity models, frameworks, core concepts
2	Infrastructure as Code	20%	~12	Terraform, compliance patterns, evidence
3	Policy-as-Code	15%	~9	OPA/Rego, policy integration, governance
4	CI/CD for GRC	15%	~9	Pipelines, security testing, evidence management
5	Cloud-Native Security	15%	~9	Monitoring, validation, continuous authorization
6	OSCAL & cATO	10%	~6	Component models, continuous authorization
7	Capstone & Portfolio	10%	~6	End-to-end workflow, integration

IaC Portfolio Requirement

In addition to passing the exam, you must submit an Infrastructure as Code portfolio demonstrating your ability to implement GRC engineering solutions. The portfolio includes Terraform configurations, OPA policies, a CI/CD pipeline, and comprehensive documentation. See the IaC Portfolio Guide section for detailed requirements and grading rubric.

Scoring and Certification

Exam Score: You receive a scaled score from 0–100. A score of 72 or higher signifies passing.

Portfolio Grade: Your portfolio is graded on implementation quality, documentation, and demonstration of GRC engineering principles.

Certification Awarded: Upon passing both the exam and portfolio review, you earn the Certified GRC Engineer — Practitioner (CGE-P) credential.

DOMAIN 1: GRC ENGINEERING FOUNDATIONS

15% of Exam — Approximately 9 Questions

What You Need to Know

GRC Engineering represents a modern evolution of traditional Governance, Risk, and Compliance practices. Rather than treating compliance as a periodic checkbox exercise, GRC Engineering integrates compliance controls directly into infrastructure design, deployment pipelines, and continuous operations. You'll learn the foundational maturity model that guides this transformation, the key frameworks that define compliance requirements, and the three pillars that make GRC Engineering effective.

1.1 GRC Engineering vs. Traditional GRC

Traditional GRC: Compliance through manual processes, periodic audits, and after-the-fact evidence collection. Controls are often separate from infrastructure decisions.

GRC Engineering: Compliance by design. Controls are embedded in infrastructure code, continuously validated through automated pipelines, and evidence is generated as a byproduct of operations. Shift-left philosophy: catch issues at development time, not in production.

1.2 The GRC Engineering Maturity Model

Level	Characteristics	Compliance Approach
1: Reactive	Audit is a once-a-year fire drill, evidence collected after the fact, no single owner	Compliance happens to the team, not from it
2: Documented	Policies named, controls listed, owners on a spreadsheet, evidence still manual	Tidier than Reactive, still not engineered
3: Automated	IaC with compliance baked in, PaC enforced in CI, evidence generated by pipelines	Audit prep shrinks from weeks to days
4: Continuous	Every merge is an audit event, drift triggers alerts, compliance dashboards on live telemetry	cATO becomes possible
5: Adaptive	System self-tunes, controls self-heal, risk is a real-time signal	Aspirational; very few orgs operate here

1.3 Three Pillars of GRC Engineering

The three pillars are required together. Two out of three is not a working program.

Pillar 1: Automate — Infrastructure as code (IaC) and policy as code (PaC). If a control isn't expressed as code, it isn't a control. Manual steps either disappear, become audited exceptions, or have an explicit owner.

Pillar 2: Evidence — Machine-verifiable receipts that come out of the build. Hash-signed, timestamped, attributable. Evidence is generated as a byproduct of building software, not gathered by hand.

Pillar 3: Continuous — Every merge is an audit event. Controls are checked on every commit, drift triggers alerts in near real-time, and auditors can query the system directly.

1.4 Compliance Framework Landscape

Know the big five. Your customers usually pick which one applies; you build controls and map them once.

Framework	Scope	Key Focus
NIST CSF 2.0	Common risk language across industries	Govern, Identify, Protect, Detect, Respond, Recover
NIST 800-53	Control catalog (especially federal)	Security and privacy control catalog
ISO 27001	International enterprise / B2B	Information security management system (ISMS)
SOC 2	B2B SaaS trust commitments	Trust Services Criteria (security, availability, confidentiality, etc.)
FedRAMP	Cloud services to the US federal government	Authorization of cloud offerings
CMMC	Defense industrial base contractors	Cybersecurity maturity levels (1–3)
NIST RMF	Risk management process	Prepare, Categorize, Select, Implement, Assess, Authorize, Monitor
OSCAL	Machine-readable controls and evidence	Catalog, Profile, Component, SSP, Assessment models
Zero Trust	Architecture principle	Never trust, always verify (identity, device, network)
FISMA	U.S. federal information security law	Requirements for federal IT systems

1.5 The GRC Engineer Role and Career Path

GRC Engineers combine cloud architecture, DevOps practices, and compliance expertise. Responsibilities include designing control frameworks, implementing policy-as-code solutions, automating evidence collection, and enabling continuous authorization. Career progression includes roles such as GRC Architect, Compliance Automation Engineer, and Security Engineering Manager.

Key Terms and Definitions

Control: A safeguard or countermeasure that prevents, detects, or mitigates security or compliance risks.

Evidence: Auditable artifacts demonstrating that a control exists and functions as intended.

Assessment: The process of evaluating whether controls meet compliance requirements.

Authorization: The formal approval to operate a system or service.

Continuous Authorization (cATO): Ongoing authorization based on real-time monitoring and evidence.

Drift Detection: Identifying when deployed infrastructure deviates from intended/coded state.

Control Validation: Automated verification that controls function correctly.

Sample Study Question — Domain 1

Question: Your organization uses Terraform to define infrastructure, OPA for policy validation, and automated monitoring for compliance drift detection. Which maturity level does this approach represent?

A) Level 1: Reactive

B) Level 2: Documented

C) Level 3: Automated

D) Level 4: Continuous

Answer: C) Level 3: Automated

Explanation: The organization demonstrates Level 3 (Automated) characteristics: IaC (Terraform), Policy-as-Code (OPA), and integrated pipeline validation. Level 4 (Continuous) would additionally require every-merge audit events, drift alerts in near real-time, and live compliance dashboards driven by telemetry, which are not explicitly mentioned.

DOMAIN 2: INFRASTRUCTURE AS CODE

20% of Exam — Approximately 12 Questions

What You Need to Know

Infrastructure as Code (IaC) is the foundation of GRC Engineering. By defining infrastructure declaratively in code, you create auditable, version-controlled, and reproducible infrastructure. This domain covers Terraform fundamentals, compliance patterns in IaC, evidence collection from infrastructure definitions, and best practices for maintaining secure and compliant infrastructure at scale.

2.1 IaC Fundamentals and Terraform

Infrastructure as Code (IaC) involves writing infrastructure definitions in code files. Terraform is the leading open-source IaC tool that enables you to define, version, and manage infrastructure across cloud providers (AWS, Azure, GCP) through a declarative language (HCL). Key benefits include reproducibility, auditability, version control, and consistency.

2.2 State Files and Resource Tracking

Terraform state files record the deployed infrastructure. The state file is critical for detecting drift, planning changes, and producing evidence. State should be stored in a secure backend (S3, Terraform Cloud) with encryption, versioning, and access controls to prevent unauthorized modifications and provide audit trails.

2.3 Compliance Patterns in Infrastructure

Compliance patterns are reusable IaC configurations that implement specific controls. Examples include: encryption at rest and in transit, network segmentation through security groups, identity and access management (IAM) policies, logging and monitoring configurations, and backup/disaster recovery setups. Modules encapsulate these patterns for consistency.

2.4 Evidence Collection from Infrastructure

IaC naturally produces evidence. Terraform configurations show control implementation. Resource tags link infrastructure to controls. Change logs provide audit trails. Outputs expose compliance-relevant data (subnet IDs, role ARNs, policy documents). Integration with monitoring and CI/CD pipelines generates additional timestamped evidence of continuous compliance.

2.5 Terraform Modules and Best Practices

Terraform modules are reusable containers of configurations. A module encapsulates best practices, reduces duplication, and enforces compliance patterns. Module composition enables organizations to build compliant infrastructure reliably. Modules should be well-documented, versioned, and tested to ensure quality and prevent configuration drift.

Hands-On Practice

Lab 2.3 First Compliant Resource: build an AWS S3 bucket that satisfies SC-28, AC-3, AU-3, AU-6, CM-6 in code. **Lab 2.4** Terraform Modules for Compliance: package the same pattern as a reusable GCP GCS module that emits a compliance attestation output. **Lab 2.5** IaC as Compliance Evidence: stand up an S3 Object Lock vault and a capture-evidence.sh script that uploads signed bundles.

Sample Study Question — Domain 2

Question: Your terraform plan -json output shows the following on a new aws_s3_bucket: encryption: [{}]. The Terraform code wires the bucket to an aws_s3_bucket_server_side_encryption_configuration that references aws_kms_key.bucket.arn. What does the plan output indicate?

A) The bucket has no encryption configured and the policy gate should reject the plan

B) The encryption block exists; the KMS key reference is computed at apply time and therefore omitted from plan JSON

C) The plan output is malformed and Terraform should be rerun with -refresh=true

D) The bucket uses AWS-owned encryption (SSE-S3) by default

Answer: B) The encryption block exists; the KMS key reference is computed at apply time and therefore omitted from plan JSON

Explanation: Unknown values (those resolved during apply) are omitted from terraform plan -json. The empty object means the encryption block is configured but its values reference resources Terraform has not created yet. A correctly written SC-28 Rego policy accepts this case (block exists, not empty-string) and only fails when the encryption resource is missing entirely.

DOMAIN 3: POLICY-AS-CODE

15% of Exam — Approximately 9 Questions

What You Need to Know

Policy-as-Code (PaC) enforces governance rules through machine-readable policies. Open Policy Agent (OPA) and its policy language Rego enable you to define compliance rules that validate configurations, deployments, and runtime behavior. This domain covers OPA/Rego fundamentals, policy integration into CI/CD pipelines, and using policies to prevent non-compliant deployments.

3.1 Open Policy Agent (OPA) Overview

Open Policy Agent (OPA) is a general-purpose policy engine. It evaluates configurations against declarative policies written in Rego. OPA can be integrated into Kubernetes (as an admission controller), Terraform (via Conftest), and CI/CD pipelines. Policies can enforce security, compliance, and operational best practices automatically.

3.2 Rego Policy Language

Rego is a declarative language for writing policies. A Rego rule is a Boolean statement that evaluates to true or false. For example, a rule might check: 'All EC2 instances must have encrypted EBS volumes.' Rego queries evaluate policies against input data (Terraform plans, Kubernetes manifests, JSON configs). Policies are composable and testable.

3.3 Policy Integration into CI/CD

Conftest is a testing framework that integrates OPA policies into CI/CD pipelines. After 'terraform plan', Conftest evaluates the plan against policies. Non-compliant plans are rejected before deployment. This shift-left approach prevents compliance violations in production. Policies become part of the development workflow, similar to code linting.

3.4 Control Mapping and Governance

Policies are mapped to compliance controls. For example, a policy 'encryption-required' maps to NIST 800-53 SC-28 (Protection of Information at Rest). This mapping creates auditability: policy violations become control violations, and policy passes generate compliance evidence. Clear mapping enables impact assessment of policy changes.

3.5 Policy Maintenance and Versioning

Policies should be version-controlled, tested, and documented. Policy repositories should include test cases validating both pass and fail scenarios. Policy changes should be reviewed (via pull requests) and impact-assessed before merging to production. Deprecated policies should be phased out with clear communication and transition periods.

Hands-On Practice

Lab 3.3 Writing Compliance Policies in Rego: author SC-28, AC-3, and CM-6 policies with # METADATA blocks against a GCP terraform plan, with passing and failing test fixtures. **Lab 3.4** Integrating PaC with Terraform: wire Conftest into the workflow, add AWS-resource-type variants of the same policies, and demonstrate a blocked plan.

Sample Study Question — Domain 3

Question: A Conftest run against a Terraform plan emits the following: "FAIL [SC-28] aws_s3_bucket.uploads: aws_s3_bucket has no matching aws_s3_bucket_server_side_encryption_configuration. Remediation: add one referencing this bucket." The pipeline expects a non-zero exit code to block the merge. What is the cleanest way to wire Conftest so the failed run still uploads the JSON output as evidence before the job exits?

A) Run conftest test --policy policies plan.json without any error handling; CI will both fail and upload

B) Run conftest test ... --output=json plan.json with || true after the command, then write the output to disk and inspect it in a follow-up step that decides exit code

C) Run conftest test ... --strict and pipe the output to /dev/null

D) Run conftest test inside a shell function that catches non-zero with set +e

Answer: B) Run conftest test ... --output=json plan.json with || true after the command, then write the output to disk and inspect it in a follow-up step that decides exit code

Explanation: Conftest exits non-zero on policy failures, which would abort the CI step and skip the artifact upload. The standard pattern is to suppress the immediate exit with || true so the JSON output is captured, then make the pass/fail decision in a separate step that has access to the captured file. This preserves evidence on failure, which is the whole point of the pipeline.

DOMAIN 4: CI/CD FOR GRC ENGINEERS

15% of Exam — Approximately 9 Questions

What You Need to Know

Continuous Integration/Continuous Deployment (CI/CD) pipelines orchestrate the automated deployment of infrastructure and policies. For GRC Engineers, pipelines are the vehicle for compliance automation. This domain covers pipeline architecture for GRC, security testing within pipelines, evidence collection from CI/CD systems, and approval workflows for compliance controls.

4.1 GRC-Focused CI/CD Pipeline Architecture

A typical GRC pipeline includes: code commit, linting/scanning, policy validation (OPA/Conftest), security testing (SAST/DAST), infrastructure planning (terraform plan), approval gates, deployment, evidence collection, and monitoring. Each stage produces artifacts and logs that serve as compliance evidence. Pipelines should enforce the principle of least privilege and immutability.

4.2 Security Testing in Pipelines

SAST (Static Application Security Testing) scans code for vulnerabilities before deployment. DAST (Dynamic Application Security Testing) tests running applications. Container scanning checks images for known vulnerabilities. Infrastructure scanning validates infrastructure configurations. These tools generate timestamped reports that serve as compliance evidence of security validation.

4.3 Evidence Generation and Audit Trails

CI/CD systems naturally generate evidence. Build logs show who triggered a deployment and when. Test results provide proof of security validation. Policy evaluations show what policies were checked and their results. Timestamps and immutable logs (stored in S3, cloud storage) create auditable trails. Integration with evidence aggregation systems (e.g., OSCAL) centralizes compliance records.

4.4 Approval Gates and Compliance Workflows

Approval gates enforce segregation of duties and compliance checks. Before production deployment, infrastructure changes should be reviewed and approved. Policy violations should trigger notifications and require exception handling. Approval workflows should be audit-logged and align with change management processes. Some organizations implement automated approvals for low-risk changes and manual reviews for high-risk changes.

Hands-On Practice

Lab 4.3 Building a GRC Evidence Pipeline: stand up GitHub OIDC trust to AWS, build a workflow that runs Terraform plan plus Conftest plus tfsec on every PR, and uploads a named evidence artifact. **Lab 4.4** Evidence Management & Chain of Custody: extend the pipeline with Cosign keyless signing and Object Lock vault upload, then build a verify-evidence.sh script that confirms integrity, authenticity, and preservation.

Sample Study Question — Domain 4

Question: Your GitHub Actions workflow uses Cosign keyless signing via cosign sign-blob. The job fails with "failed to get OIDC token: Get http://...: 403 Forbidden." Which fix correctly addresses the root cause?

A) Add COSIGN_EXPERIMENTAL=1 as an env var on the run step

B) Move the cosign-installer step to run after the configure-aws-credentials step

C) Add permissions: id-token: write at the workflow or job level

D) Use --key cosign.key with a pre-generated long-lived key pair instead

Answer: C) Add permissions: id-token: write at the workflow or job level

Explanation: Cosign keyless requires GitHub to mint an OIDC token for the runner; that mint requires the workflow to declare permissions: id-token: write. Without it, the API call to Sigstore Fulcio is unauthenticated and returns 403. COSIGN_EXPERIMENTAL was the old flag (no longer required). Falling back to a long-lived key pair defeats the keyless pattern.

DOMAIN 5: CLOUD-NATIVE SECURITY & MONITORING

15% of Exam — Approximately 9 Questions

What You Need to Know

Every cloud provider's security catalog can be mapped to five functional families: Identity, Logging, Detection, Secrets, and Posture. Once you know the families, the vendor names are vocabulary on top. This domain covers the AWS, Azure, and GCP implementations, the chains by which they feed each other (CloudTrail to Config to Security Hub on AWS; Org Policy at the API on GCP; Management Group hierarchy on Azure), and the NIST 800-53 control families each native service produces evidence for.

5.1 The Five Functional Families

Identity: federated SSO, role-based access, just-in-time elevation. Logging: structured records of every API call, integrity-validated. Detection: managed services that score anomalies and policy violations. Secrets: managed key and secret stores. Posture: configuration recorders and aggregators that produce a normalized findings feed. Native services in each family come pre-integrated with IAM, billing, and audit logs, which is leverage you don't get from third-party tools.

5.2 The Rosetta Stone (AWS / Azure / GCP)

Family	AWS	Azure	GCP
Identity	IAM Identity Center, IAM	Entra ID, PIM	Cloud Identity, IAM, BeyondCorp
Logging (audit)	CloudTrail	Activity Log + Diagnostic Logs	Cloud Audit Logs (4 streams)
Detection	GuardDuty	Defender for Cloud, Sentinel	Security Command Center, Chronicle
Secrets / keys	Secrets Manager + KMS	Key Vault	Secret Manager + Cloud KMS
Posture	AWS Config + Security Hub	Azure Policy + Defender CSPM	SCC + Org Policy Service

5.3 AWS Chain and Control Mapping

CloudTrail is the raw truth: every service writes to it. AWS Config records resource configuration changes; Security Hub aggregates findings from Config, GuardDuty, and Inspector into one normalized stream. Control mapping: CloudTrail produces AU-2 / AU-12 evidence, log-file validation adds AU-10. Config feeds CM-2 / CM-6 / CM-8 (baseline, settings, inventory). GuardDuty feeds SI-4 / IR-4. Security Hub aggregator addresses CA-7 (continuous monitoring). Watch-out: Config bills per item recorded plus per rule eval, and many Security Hub controls require Config to be on; some org-managed accounts SCP-block Config in non-management accounts.

5.4 GCP Identity-First Baseline

GCP enforces at the API. Organization Policy Service rejects non-compliant resource creation calls before they succeed ("FAILED_PRECONDITION: constraint iam.disableServiceAccountKeyCreation"). Workload Identity Federation replaces long-lived service-account JSON keys (the #1 GCP breach vector) with short-lived OIDC tokens minted by GitHub or another trusted identity provider. Data Access logs are off by default in GCP and have to be enabled per service (storage.googleapis.com, cloudkms.googleapis.com, iam.googleapis.com); turning them on is the most common audit finding remediation.

Hands-On Practice

Lab 5.2 AWS Security Services Baseline: deploy CloudTrail (multi-region, log-file-validation) and Security Hub (NIST 800-53 standard) via Terraform, then capture findings as JSON evidence. **Lab 5.4** GCP Security Services Baseline: enforce Org Policy at the API, replace service-account keys with Workload Identity Federation, and enable Data Access logs explicitly per service.

Sample Study Question — Domain 5

Question: Your team is enabling AWS Security Hub and subscribing to the NIST 800-53 standard in a sandbox account. Many critical findings appear with the message "AWS Config should be enabled and use the service-linked role for resource recording." On apply, terraform fails with "AccessDeniedException ... explicit deny in a service control policy." What is the correct interpretation?

A) The Terraform code is wrong; AWS Config is enabled implicitly when Security Hub is on

B) An organization-level SCP centrally manages AWS Config; Config recorders cannot be created in this account, and Security Hub is reporting that gap as its own evidence

C) Security Hub itself requires elevated permissions and cannot be enabled in a non-master account

D) The Terraform provider version is too old and does not support AWS Config

Answer: B) An organization-level SCP centrally manages AWS Config; Config recorders cannot be created in this account, and Security Hub is reporting that gap as its own evidence

Explanation: AWS Config is often centralized at the org level; an SCP denying config:* in non-management accounts is a common pattern. Security Hub reporting Config-related findings is correct and useful: the account is reporting its own posture gap. The fix is a process gap (request Config from the org admin), not a code gap.

DOMAIN 6: OSCAL & CONTINUOUS AUTHORIZATION

10% of Exam — Approximately 6 Questions

What You Need to Know

OSCAL (Open Security Controls Assessment Language) is a standardized format for defining controls, assessments, and evidence. OSCAL enables interoperability between compliance tools and streamlines the production and consumption of compliance artifacts. This domain covers OSCAL structure, component models, using OSCAL for evidence aggregation, and integration with continuous authorization frameworks.

6.1 OSCAL Overview and Structure

OSCAL is a project of NIST that provides machine-readable formats for security controls, assessments, and findings. OSCAL documents are JSON/XML and follow a hierarchical structure: catalogs (control definitions), profiles (control selections), component definitions (how components implement controls), assessment plans, results, and pointers (evidence references). OSCAL enables automation of control mapping, assessment, and reporting.

6.2 Component Models and Control Mappings

OSCAL Component Models describe how a system component (e.g., an RDS database, a Kubernetes cluster) implements controls. A component model maps controls to the component's configuration and generates evidence. For example, a component model for RDS might map NIST SC-28 (Protection at Rest) to RDS encryption settings. Component models enable automated compliance reporting.

6.3 Evidence Aggregation with OSCAL

OSCAL Assessment Results documents aggregate evidence. Results include control findings, observations, and evidence references (pointing to timestamps, logs, reports). Tools generate OSCAL results from monitoring, testing, and validation systems. OSCAL standardization enables aggregation of evidence from heterogeneous sources—compliance becomes composable.

6.4 Continuous Authorization with OSCAL

OSCAL's assessment results and monitoring integration enable continuous authorization. Systems continuously produce OSCAL results showing control status. Authorization decisions are made on current evidence rather than historical snapshots. OSCAL's machine-readable format enables automated decision-making: if all controls are compliant, authorization is granted; if controls fail, access is revoked. This automates cATO.

Hands-On Practice

Lab 6.1 Introduction to OSCAL: install compliance-trestle, scaffold a Component Definition for one of your earlier Terraform modules, populate it with implemented-requirements citing NIST controls (SC-28, AC-3, AU-3, CM-6), wire link href values to signed evidence in the Lab 2.5 vault, and pass trestle validate.

Sample Study Question — Domain 6

Question: You author an OSCAL component-definition.json by hand. You hand-write UUIDs that look like “00000000-0000-0000-0000-000000000001” for the component, control-implementation, and each implemented-requirement. trestle validate fails with “string does not match regex” on every UUID. Which is the cleanest fix?

A) Replace the UUIDs with sequential integers (1, 2, 3) — OSCAL accepts integers if it accepts UUIDs

B) Generate v4 UUIDs (e.g., python3 -c 'import uuid; print(uuid.uuid4())') and replace each one; v4 is the only format the OSCAL schema accepts

C) Strip the dashes from the UUIDs — the schema rejects the dashes, not the digits

D) Set --skip-validation when running trestle so the schema check is bypassed

Answer: B) Generate v4 UUIDs (e.g., python3 -c 'import uuid; print(uuid.uuid4())') and replace each one; v4 is the only format the OSCAL schema accepts

Explanation: OSCAL strictly requires v4 UUIDs (^[0-9A-Fa-f]{8}-...-[45][0-9A-Fa-f]{3}-[89ABab][0-9A-Fa-f]{3}-[0-9A-Fa-f]{12}$). Hand-written all-zero UUIDs do not match. The fix is to generate real v4 UUIDs once per UUID. Skipping validation is not the fix; the document still needs to validate when an assessor checks it.

DOMAIN 7: APPLIED GRC ENGINEERING

10% of Exam — Approximately 6 Questions

What You Need to Know

Applied GRC Engineering is the capstone. The candidate inherits a deliberately non-compliant workload (a small AWS-deployed application provided as a public starter repo) and assembles four layers around it: a Terraform compliance baseline, an OPA policy suite, a GitHub Actions evidence pipeline, and an OSCAL component definition. The same skills the multiple-choice exam tests, this time end-to-end, on a real cloud account. The capstone is graded on integration depth, not resource count.

7.1 The Capstone Scenario

The candidate is the first GRC engineer at a fictional 50-person company. The engineering team has shipped a Patient Intake API with deliberate compliance gaps. The CTO asks the candidate to make it audit-defensible in 30 days without slowing engineering down. Step 0 is the deploy gate: clone the public starter, deploy it to the candidate's own sandbox, confirm a smoke test returns 200. If the candidate cannot deploy a working starter, the capstone cannot proceed.

7.2 The Four Required Layers

Layer 1 — Terraform Baseline: KMS keys, S3 evidence bucket with Object Lock, multi-region CloudTrail, and gap-closing overrides on the inherited workload (move the Lambda into the VPC, add SSE-KMS to the data bucket, tighten IAM, add bucket policy for TLS-only).

Layer 2 — OPA Policy Suite: Five or more Rego policies with metadata blocks. Each policy cites a control ID from the candidate's declared primary framework. Each policy has passing and failing test fixtures.

Layer 3 — GitHub Actions Pipeline: One workflow with five named steps: Plan, Policy Check, Apply, Sign (Cosign keyless via GitHub OIDC), Upload to vault. Two pull requests must exist in the repo's history: one that passed and merged, one that failed the policy gate and was blocked.

Layer 4 — OSCAL Component: A component-definition.json describing what was actually built. Real v4 UUIDs. Implementation statements reference real Terraform resources. Evidence link href values resolve to signed objects in the candidate's vault.

7.3 Framework Selection

The candidate declares a primary compliance framework: HIPAA Security Rule, SOC 2 Trust Services Criteria, or CMMC Level 2. Every policy cites a control ID from that framework. The OSCAL component's control-implementation source points at that framework's catalog (or NIST 800-53 with HIPAA/SOC2 mapping props). The framework choice and its rationale are defended in the WRITEUP.md.

7.4 The Three Common Failure Modes

Too much scope. A 30-resource Terraform that closes 5 gaps cleanly beats a 200-resource Terraform with policies tacked on. **Copy-paste OSCAL.** A component-definition.json that does not actually describe what was built is worse than no OSCAL. Authenticity over completeness. **Unsigned evidence.** A pipeline that produces a plan but does not sign and immutably store it has not demonstrated chain of custody. Reviewers pull a recent run and verify it; if the verification fails, the layer fails.

7.5 Suggested 30-Day Plan

Week 1: design (pick scenario, map controls, sketch repo). Week 2: build the Terraform baseline; apply it once by hand; do not start the pipeline until baseline applies clean. Week 3: write the Rego suite; build the GitHub

Actions workflow; open the green PR and the red PR; add Cosign signing and vault upload. Week 4: author the OSCAL component, validate with trestle, wire evidence URIs to real vault objects, write WRITEUP.md, submit. Break it into weeks. Do not start day 29.

Hands-On Practice

Capstone Brief (Lab 7.1): the comprehensive brief lives in the cert-platform docs and references a public starter workload repo. The candidate forks the starter, deploys it, then assembles all prior labs (2.3-2.5, 3.3-3.4, 4.3-4.4, 5.2 or 5.4, 6.1) into a single capstone repo wrapped around it.

Sample Study Question — Domain 7

Question: A capstone candidate ships a repo with a Terraform baseline, an OPA suite, a GitHub Actions workflow that produces signed evidence, and an OSCAL component-definition.json. The OSCAL component declares NIST 800-53 SC-28 with an evidence link to s3://VAULT/runs/12345/bundle.tar.gz. A reviewer follows the link, downloads the bundle, runs cosign verify-blob, and the signature verifies. Recomputing SHA-256 matches the .sha256 sidecar. The Object Lock retention has not yet expired. Which of the following best describes the layer the reviewer just verified?

A) Only the OSCAL component, because the reviewer started in OSCAL

B) Only the GitHub Actions pipeline, because the bundle was produced by it

C) End-to-end integration of all four layers: OSCAL points at the bundle, the pipeline produced it, the signing in Layer 3 made authenticity verifiable, and the vault from Layer 1 made it tamper-evident

D) Only the Terraform baseline, because the vault is part of the baseline

Answer: C) End-to-end integration of all four layers: OSCAL points at the bundle, the pipeline produced it, the signing in Layer 3 made authenticity verifiable, and the vault from Layer 1 made it tamper-evident

Explanation: The capstone scoring criterion is end-to-end integration. The reviewer's traversal exercises Layer 4 (OSCAL link), Layer 3 (the signing the pipeline performed), Layer 1 (the vault holding the bundle with active retention), and the Layer 2 evidence the bundle contains. A working chain that touches all four layers in one verification is what "end-to-end integration" looks like in practice.

FRAMEWORKS PRIMER (FOR THE CAPSTONE)

The capstone requires the candidate to declare a primary compliance framework. The three accepted options below are the ones a US-based GRC practitioner is most likely to encounter. Map every Rego policy and OSCAL implementation statement to the chosen framework's control IDs.

HIPAA Security Rule

Choose this framing when the workload handles PHI. Most-cited control IDs: **164.308(a)(1)** Security Management Process; **164.308(a)(7)** Contingency Plan; **164.312(a)(1)** Access Control; **164.312(a)(2)(iv)** Encryption and decryption; **164.312(b)** Audit Controls; **164.312(d)** Person or Entity Authentication; **164.312(e)(1)** Transmission Security. OSCAL note: there is no NIST-published HIPAA OSCAL catalog; teams typically cite **NIST SP 800-66 Rev. 2** (Implementing the HIPAA Security Rule) as the catalog and reference 164.x sections in props on each implemented-requirement.

SOC 2 Trust Services Criteria

Choose this framing for B2B SaaS trust commitments. Most-cited Common Criteria: **CC6.1** logical access controls; **CC6.3** authorization and least privilege; **CC6.6** boundary protection; **CC6.7** transmission security; **CC7.2** system monitoring; **A1.2** system availability and recovery. OSCAL note: AICPA does not publish an official OSCAL TSC catalog; teams use a community-maintained one or map TSC to NIST 800-53 and cite the latter.

CMMC Level 2 (NIST 800-171)

Choose this framing for federal pilots or DoD contractor work. CMMC L2 inherits 110 NIST 800-171 practices. Most-cited for a small workload: **AC.L2-3.1.1** authorized access enforcement; **AC.L2-3.1.3** information flow control; **AC.L2-3.1.5** least privilege; **AU.L2-3.3.1** generate audit records; **SC.L2-3.13.1** boundary protection; **SC.L2-3.13.8** encryption in transit; **SC.L2-3.13.11** FIPS-validated cryptography for CUI; **SI.L2-3.14.6** system monitoring. OSCAL note: NIST publishes 800-171 Rev. 3 in OSCAL; CMMC's catalog is a profile over 800-171.

How the framework choice flows through the capstone layers

Every Rego policy's # **METADATA** block carries a custom.framework field set to the declared framework and a custom.controls field listing the relevant control IDs. The deny message includes the control ID inline so a developer reading a failed PR sees the exact citation. The OSCAL component-definition.json declares one control-implementation whose source URI matches the chosen framework's catalog. Cross-framework references can live in props on each implemented-requirement so the same control can be cited in HIPAA AND SOC 2 AND CMMC if it satisfies all three.

CAPSTONE PORTFOLIO GUIDE

Portfolio Overview

The capstone portfolio is a required component of CGE-P certification. The candidate inherits a public starter workload, deploys it to their own cloud sandbox, and assembles four layers around it that together produce signed evidence on every push. The same skills the multiple-choice exam tests, this time end-to-end.

Required Components

1. Terraform Baseline: KMS keys, S3 evidence bucket with Object Lock, multi-region CloudTrail, and gap-closing overrides on the inherited workload

2. OPA Policy Suite: Five or more Rego policies with metadata blocks and tests; each policy cites a control ID from the declared primary framework

3. GitHub Actions Pipeline: One workflow with five named steps (Plan, Policy Check, Apply, Sign, Upload). Two pull requests in repo history: one green, one red

4. OSCAL Component: A component-definition.json that validates with trestle and links to real signed objects in the candidate's vault

Plus a five-page **WRITEUP.md** explaining design decisions, framework choice, trade-offs, and an honest list of what the candidate did not get to.

Grading Rubric

Category	Weight
End-to-End Integration	35% — Open a PR; the gate runs; the gate decides whether apply happens; apply triggers signing; signing uploads to the vault. Not four disconnected demos.
Working Evidence Pipeline	30% — Reviewer pulls a recent run and verifies it: Cosign signature against the public Sigstore log, SHA-256 recompute, Object Lock retention check. All three must hold.
Clear Design Reasoning	20% — WRITEUP.md explains why each tool was chosen and what trade-offs were accepted; honest gap reporting.
Framework Alignment	15% — Policies cite the declared framework's control IDs; OSCAL's control-implementation source matches.

Submission Guidelines

Submit via GitHub repository. Include a README with setup instructions and a COMPLIANCE.md mapping controls to implementation. All code must be original work. Repositories must be public for review. Allow 5–7 business days for grading after submission.