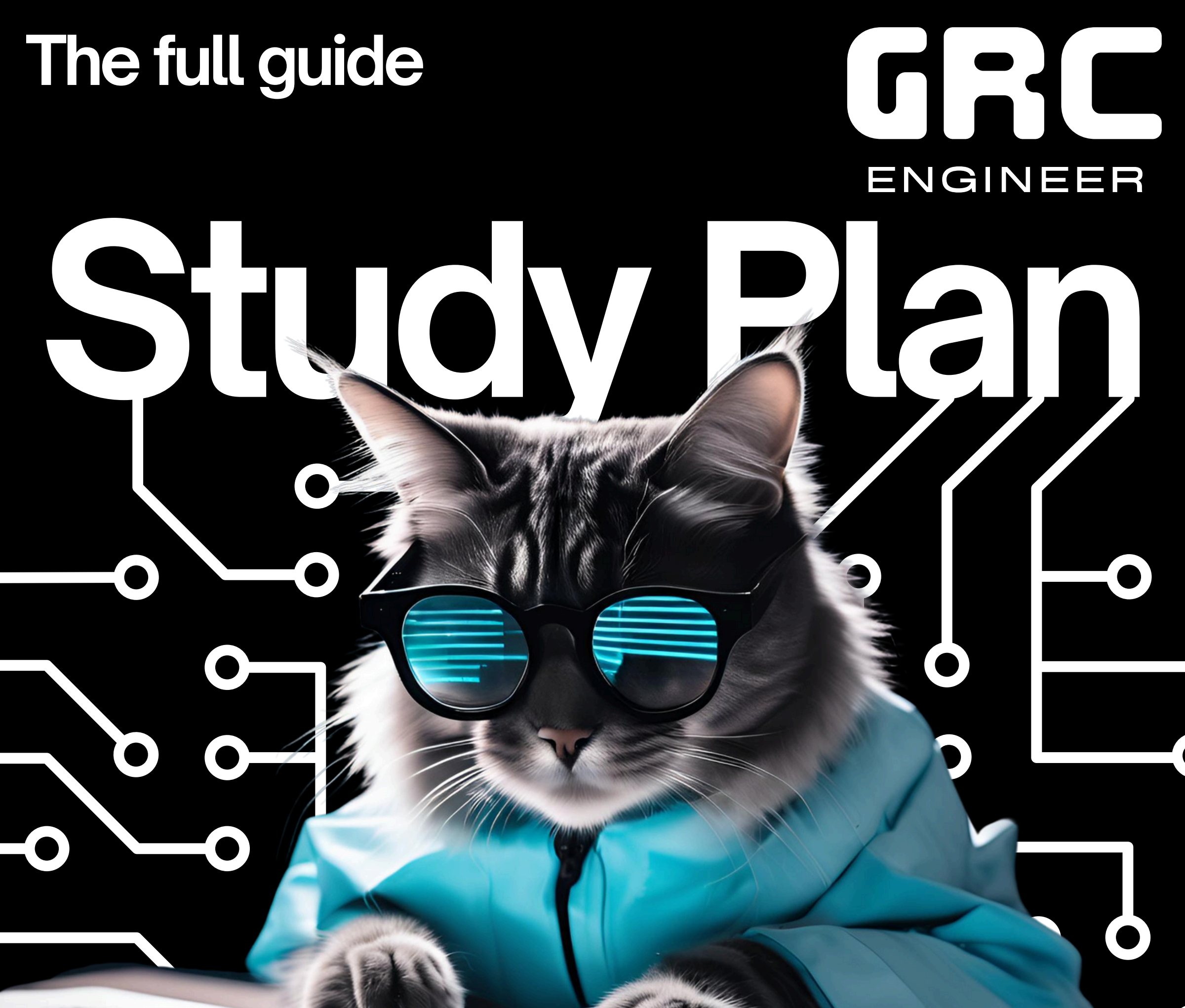
The full guide

GRC ENGINEER

# Study Plan



From spreadsheets to code in 180 days

# Introduction:
# Spreadsheets to Code in 180 days

GRC ENGINEER

Why This Guide Exists
If you work in GRC, you've probably noticed a growing gap. While engineering teams deploy code continuously and infrastructure changes by the minute, many GRC teams still rely on screenshots and spreadsheets. This disconnect isn't just inefficient—it's becoming unsustainable.

This study plan is your roadmap from traditional GRC to GRC Engineering. Whether you're a compliance analyst looking to automate evidence collection or a GRC manager wanting to modernize your program, this guide will help you build the technical skills you need.

**What You'll Learn**
This guide takes you through ten core areas to build your GRC Engineering baseline.

**How It Works**

Each chapter includes
- Clear explanations in GRC terms
- Practical examples you can use immediately
- Real code that solves actual GRC problems
- Specific projects to build your skills
- Resources for deeper learning

**Prerequisites**
You don't need to be a programmer to start this journey. You just need:
- Basic comfort with computers
- Understanding of GRC concepts
- Willingness to learn technical skills
- Access to a computer where you can install tools

**Getting Started**
1. Read one chapter per week
2. Complete the practical projects
3. Build your own versions of the examples
4. Share your learnings with the community

Let's transform how we do GRC—one script at a time.

# Chapter 1:
# GRC Architecture Foundations

GRC ENGINEER

Traditional GRC was built for static infrastructure - servers in data centers, quarterly changes, and manual documentation. Today's reality is different: infrastructure scales automatically, containers live for hours, and configurations are pushed through code. This fundamental shift breaks traditional compliance approaches.

Modern infrastructure means your entire production environment might be recreated daily. While engineering teams deploy continuously, GRC often operates on annual cycles. By the time traditional processes document an environment, that environment has changed multiple times.

The solution isn't just automating existing processes - it's rethinking how GRC works in a modern engineering organization. Evidence collection becomes part of deployment pipelines, control testing happens continuously, and documentation lives in code.

**Quick Start Project**
*Infrastructure Mapping*

- Document one critical system's infrastructure components
- Track how often each component changes over a week

**Practical Applications**
- Map compliance requirements to cloud services
- Track control effectiveness across dynamic infrastructure
- Design automated evidence collection points

**Moving Forward**
Focus first on understanding how your organization builds and deploys software. The automation opportunities will become clear once you understand the infrastructure.

**Resource**
AWS Well-Architected Framework - Start here for a solid understanding of modern infrastructure principles.

# Chapter 2: Evidence Automation Basics

GRC ENGINEER

Most GRC teams spend hours taking screenshots, organizing files, and updating spreadsheets. This manual work isn't just time-consuming - it's prone to errors and impossible to scale. Let's start automating with simple tools you already have.

The command line might seem intimidating, but simple commands can save hours of manual work. Instead of clicking through folders to find recent evidence, one command can find all files modified in the last quarter.

Basic automation starts with organizing your evidence systematically. Clear folder structures and consistent naming make automation possible.

**Quick Start Project:**
*Evidence Organisation*

1. Create a structured evidence folder
2. Write a basic script to organize files

```bash
#!/bin/bash
# evidence-organizer.sh
# Simple script to organize evidence files by date and type

# Set up directory structure
EVIDENCE_DIR="./evidence"
DATE=$(date +%Y-%m)

# Create directories
mkdir -p "$EVIDENCE_DIR/$DATE/access-reviews"
mkdir -p "$EVIDENCE_DIR/$DATE/configs"
mkdir -p "$EVIDENCE_DIR/$DATE/logs"

# Move files to appropriate directories
mv *access*.pdf "$EVIDENCE_DIR/$DATE/access-reviews/"
mv *config*.pdf "$EVIDENCE_DIR/$DATE/configs/"
mv *log*.pdf "$EVIDENCE_DIR/$DATE/logs/"

echo "Evidence organized in $EVIDENCE_DIR/$DATE"
```

This script is simple but practical - it organizes evidence files into dated folders by type. It's a real starting point for automation.

**Practical Applications:**

- Organize evidence files automatically
- Find evidence files by date and type
- Create consistent folder structures

**Moving Forward**
Start with organizing one type of evidence. Once you have a system working, expand to other types.

**Resource**
Command Line Crash Course - Practical introduction to command line basics.

# Chapter 3: Python Fundamentals

GRC ENGINEER

Python has become the go-to language for GRC automation because it's readable and powerful. You don't need to become a developer - you just need enough Python to automate repetitive tasks.

Instead of manually checking spreadsheets and copying data, Python can read files, extract information, and generate reports automatically. It turns hours of manual work into seconds of computer time.

Start with simple scripts that solve real problems. Focus on automating one manual task completely before moving to more complex projects.

**Quick Start Project**
*Control Status Checker*

1. Create a script to read control status from a CSV
2. Generate a simple report

```python
import csv
from datetime import datetime, timedelta

def check_control_status(csv_file):
    # Track controls needing review
    needs_review = []

    # Read the CSV file
    with open(csv_file, 'r') as file:
        reader = csv.DictReader(file)

        # Check each control
        for row in reader:
            last_review = datetime.strptime(row['last_review_date'], '%Y-%m-%d')
            days_since_review = (datetime.now() - last_review).days

            # Flag controls not reviewed in 90 days
            if days_since_review > 90:
                needs_review.append({
                    'control_id': row['control_id'],
                    'owner': row['control_owner'],
                    'days': days_since_review
                })

    # Generate simple report
    print("Controls Needing Review:")
    for control in needs_review:
        print(f"Control {control['control_id']} - "
              f"Owner: {control['owner']} - "
              f"Days since review: {control['days']}")

# Use the function
check_control_status('controls.csv')
```

This script solves a real GRC problem - finding controls that need review. It's simple but immediately useful.

**Practical Applications**

- Check control review status automatically
- Generate compliance status reports
- Process evidence file metadata

**Moving Forward**
Start with scripts that read and analyze your existing data. Build familiarity before moving to more complex automation.

**Resource**
Automate the Boring Stuff with Python - Perfect introduction to practical Python automation.

# Chapter 4: SQL & Data Management

GRC ENGINEER

Your risk register, control inventory, and vendor assessments probably live in spreadsheets. As your program grows, you'll find yourself creating more complex formulas and eventually a spreadsheet of spreadsheets that takes forever to open. SQL solves this by making complex questions easy to answer.

Think about common GRC questions: "Which high-risk vendors haven't been reviewed this quarter?" or "Which controls have no evidence in the last 90 days?" In spreadsheets, these require complex formulas. In SQL, they're straightforward queries.

The power of SQL isn't just in storing data - it's in asking questions that would be impossible in spreadsheets.

**Quick Start Project**
*Control Database*

1. Create a simple control tracking database
2. Write basic queries for common lookups

```sql
-- Create a simple control tracking table
CREATE TABLE controls (
    id VARCHAR(50),
    name TEXT,
    framework VARCHAR(50),
    owner VARCHAR(100),
    last_review_date DATE,
    risk_rating VARCHAR(20)
);

-- Find high-risk controls not reviewed in 90 days
SELECT id, name, owner, last_review_date
FROM controls
WHERE risk_rating = 'HIGH'
  AND last_review_date < CURRENT_DATE - INTERVAL '90 days'
ORDER BY last_review_date;

-- Count controls by framework and risk
SELECT framework, risk_rating, COUNT(*) as control_count
FROM controls
GROUP BY framework, risk_rating
ORDER BY framework, risk_rating;
```

These queries solve real GRC problems - finding overdue reviews and understanding your control landscape.

**Practical Applications**

- Track control testing history
- Monitor vendor assessments
- Analyze risk trends

**Moving Forward**
Start by converting one critical spreadsheet to a database. Focus on the questions you ask most often.

**Resource**
SQLBolt - Learn SQL with Interactive Exercises - Hands-on SQL learning with immediate feedback.

# Chapter 5: API Integration

GRC ENGINEER

Most modern security tools have APIs - ways to programmatically access their data. Instead of logging into dashboards and taking screenshots, you can write code that pulls this information automatically. This is where real evidence automation begins.

APIs turn manual evidence collection into automated workflows. Instead of periodic screenshots, you get continuous data collection. More importantly, you can combine data from multiple tools to create comprehensive compliance reports.

The key is starting simple - connecting to one tool and pulling basic data before building complex integrations.

**Quick Start Project**

*AWS Security Check*

1. Connect to AWS Security Hub
2. Pull basic compliance data

```python
import boto3
from datetime import datetime, timedelta

def check_security_findings():
    # Connect to Security Hub
    securityhub = boto3.client('securityhub')

    # Get findings from last 7 days
    response = securityhub.get_findings(
        Filters={
            'CreatedAt': [{
                'Start': datetime.now() - timedelta(days=7),
                'End': datetime.now()
            }],
            'Severity': [{
                'Eq': ['HIGH']
            }]
        }
    )

    # Generate simple report
    print("High Severity Findings Last 7 Days:")
    for finding in response['Findings']:
        print(f"Title: {finding['Title']}")
        print(f"Resource: {finding['Resources'][0]['Id']}")
        print(f"Status: {finding['Workflow']['Status']}")
        print("---")

# Run the check
check_security_findings()
```

This script provides real value - automatically checking AWS security findings instead of manual dashboard reviews.

**Practical Applications**

- Automated evidence collection
- Security status monitoring
- Compliance reporting

**Moving Forward**

Start with one security tool's API. Master basic data collection before attempting complex integrations.

**Resource**

AWS Security Hub Documentation - Great example of security tool API usage.

# Chapter 6: Infrastructure as Code

GRC ENGINEER

Instead of documenting how systems should be configured, Infrastructure as Code (IaC) lets you write code that enforces those configurations. This means your security controls become code that can be version-controlled, tested, and automatically validated.

When auditors ask about security settings, you can show them the exact code that enforces those settings. More importantly, you can be confident those settings are consistently applied across your infrastructure.

Think of IaC as making your security requirements executable - instead of hoping systems stay compliant, you encode compliance into their creation.

**Quick Start Project**
*Security Baseline*

1. Create a basic AWS security baseline
2. Add compliance tags and security rules

```
# Basic AWS security baseline with compliance tags
resource "aws_security_group" "compliant_web" {
  name = "compliant-web-access"
  description = "SOC 2 compliant web access"

  # HTTPS only
  ingress {
    from_port = 443
    to_port = 443
    protocol = "tcp"
    cidr_blocks = ["10.0.0.0/8"]
  }

  # Explicit egress rules
  egress {
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Compliance tags
  tags = {
    Environment = "Production"
    Compliance = "SOC2"
    Control_ID = "CC6.1"
    Evidence_Contact = "grc@company.com"
  }
}

# Enable encryption by default
resource "aws_s3_bucket_public_access_block" "compliant_bucket" {
  bucket = aws_s3_bucket.data.id

  block_public_acls = true
  block_public_policy = true
  ignore_public_acls = true
  restrict_public_buckets = true
}
```

This code doesn't just document security requirements - it enforces them.

**Practical Applications**

- Automated security control implementation
- Consistent configuration management
- Compliance validation

**Moving Forward**
Start with one simple security control in code. Build confidence before tackling complex configurations.

**Resource**
Terraform Getting Started Guide - Best introduction to Infrastructure as Code.

# Chapter 7: Continuous Compliance

GRC ENGINEER

Traditional compliance is like an annual exam - lots of preparation, intense stress, then forgetting everything until next year. Continuous compliance integrates testing into your daily operations, catching issues before they become audit findings.

By adding compliance checks to your deployment pipeline, you prevent non-compliant configurations from reaching production. Instead of finding problems during audits, you catch them automatically during deployment.

This shift from periodic to continuous validation fundamentally changes how we approach compliance.

**Quick Start Project**
*Basic Pipeline Check*

1. Add security checks to your deployment process
2. Save results as evidence

```yaml
# .gitlab-ci.yml - Basic compliance pipeline
image: ubuntu:latest

# Define stages in pipeline
stages:
  - security_check
  - evidence_collection

variables:
  EVIDENCE_DIR: "compliance_evidence"

# Run security checks
security_scan:
  stage: security_check
  script:
    # Check for secrets in code
    - curl -sSL
https://github.com/zricethezav/gitleaks/releases/download/v8.8.7/gitleaks_8.8.7_linux_x64.tar.gz |
tar xz
    - ./gitleaks detect --source . --report-path ${EVIDENCE_DIR}/secrets_scan.json

    # Check Terraform configurations
    - curl -sSL https://github.com/aquasecurity/tfsec/releases/download/v1.15.4/tfsec-linux-amd64 -o
tfsec
    - chmod +x tfsec
    - ./tfsec . --format json > ${EVIDENCE_DIR}/terraform_scan.json
  artifacts:
    paths:
      - ${EVIDENCE_DIR}
    expire_in: 30 days

# Collect and store evidence
collect_evidence:
  stage: evidence_collection
  script:
    - mkdir -p ${EVIDENCE_DIR}
    - echo "Pipeline Run: $(date)" >> ${EVIDENCE_DIR}/run_evidence.txt
    - echo "Branch: $CI_COMMIT_REF_NAME" >> ${EVIDENCE_DIR}/run_evidence.txt
    - echo "Commit: $CI_COMMIT_SHA" >> ${EVIDENCE_DIR}/run_evidence.txt
    - echo "Pipeline Status: $CI_JOB_STATUS" >> ${EVIDENCE_DIR}/run_evidence.txt
  artifacts:
    paths:
      - ${EVIDENCE_DIR}
    reports:
      junit: ${EVIDENCE_DIR}/*_scan.json
  rules:
    - when: always  # Run even if previous stage fails
```

This workflow automatically checks for common security issues and maintains evidence of testing.

**Practical Applications**

- Prevent non-compliant deployments
- Maintain continuous evidence trail
- Automate control validation

**Moving Forward**
Start with one simple compliance check in your pipeline. Add more checks as you gain confidence.

**Resource**
GitLab CI/CD Documentation - Comprehensive guide to pipeline automation.

# Chapter 8: Evidence Collection Automation

GRC ENGINEER

Let's build a complete evidence collection system that combines what we've learned about APIs, databases, and automation. The goal is to replace manual screenshot collection with automated, continuous evidence gathering.

Modern evidence collection should be automatic and continuous. When auditors request evidence, you should be able to generate it from data you're already collecting, not start a screenshot marathon.

This is where we combine multiple tools and techniques to create a comprehensive solution.

**Quick Start Project**
*AWS Evidence Collector*

1. Create an evidence collection database
2. Build automated collection script

```python
import boto3
import sqlite3
from datetime import datetime

class EvidenceCollector:
    def __init__(self):
        # Connect to SQLite database
        self.db = sqlite3.connect('evidence.db')
        self.setup_database()

    def setup_database(self):
        # Create evidence table
        self.db.execute('''
        CREATE TABLE IF NOT EXISTS evidence (
            id TEXT PRIMARY KEY,
            control_id TEXT,
            type TEXT,
            content TEXT,
            collected_date TIMESTAMP,
            source TEXT
        )''')

    def collect_iam_evidence(self):
        # Connect to AWS
        iam = boto3.client('iam')

        # Get MFA status for all users
        response = iam.list_users()

        # Process each user
        for user in response['Users']:
            mfa_devices = iam.list_mfa_devices(UserName=user['UserName'])

            # Store evidence
            self.db.execute('''
            INSERT OR REPLACE INTO evidence
            VALUES (?, ?, ?, ?, ?, ?)
            ''', (
                f"mfa_{user['UserName']}",
                'CC6.1',  # Example control ID
                'MFA_Check',
                f"MFA Devices: {len(mfa_devices['MFADevices'])}",
                datetime.now(),
                'AWS IAM'
            ))

        self.db.commit()

    def generate_report(self):
        # Get recent evidence
        cursor = self.db.execute('''
        SELECT control_id, COUNT(*) as evidence_count
        FROM evidence
        WHERE collected_date > datetime('now', '-90 days')
        GROUP BY control_id
        ''')

        print("Evidence Collection Report:")
        for row in cursor:
            print(f"Control {row[0]}: {row[1]} pieces of evidence")
```

This script creates a real evidence collection system - gathering data automatically and storing it for audit needs.

**Practical Applications**

- Automated AWS configuration evidence
- Continuous control validation
- Evidence database maintenance

**Moving Forward**
Start with one type of evidence and expand the system gradually. Focus on evidence you collect frequently.

**Resource**
AWS Compliance Documentation - Great guide for automated evidence collection.

# Chapter 9: Compliance Reporting & Dashboards

GRC ENGINEER

Moving from static reports to dynamic dashboards changes how organizations view compliance. Instead of point-in-time snapshots, you provide continuous visibility into your security posture.

Good compliance dashboards don't just show status - they drive action. They help teams understand what needs attention and why. More importantly, they make compliance data accessible and actionable for everyone involved.

Modern reporting should be automated and real-time, pulling directly from your evidence and control databases.

**Quick Start Project**
*Basic Compliance Dashboard*

1. Create a simple web dashboard
2. Display compliance status from your database

```python
import boto3
import sqlite3
from datetime import datetime

class EvidenceCollector:
    def __init__(self):
        # Connect to SQLite database
        self.db = sqlite3.connect('evidence.db')
        self.setup_database()

    def setup_database(self):
        # Create evidence table
        self.db.execute('''
        CREATE TABLE IF NOT EXISTS evidence (
            id TEXT PRIMARY KEY,
            control_id TEXT,
            type TEXT,
            content TEXT,
            collected_date TIMESTAMP,
            source TEXT
        )''')

    def collect_iam_evidence(self):
        # Connect to AWS
        iam = boto3.client('iam')

        # Get MFA status for all users
        response = iam.list_users()

        # Process each user
        for user in response['Users']:
            mfa_devices = iam.list_mfa_devices(UserName=user['UserName'])

            # Store evidence
            self.db.execute('''
            INSERT OR REPLACE INTO evidence
            VALUES (?, ?, ?, ?, ?, ?)
            ''', (
                f"mfa_{user['UserName']}",
                'CC6.1',  # Example control ID
                'MFA_Check',
                f"MFA Devices: {len(mfa_devices['MFADevices'])}",
                datetime.now(),
                'AWS IAM'
            ))

        self.db.commit()

def generate_report(self):
    # Get recent evidence
    cursor = self.db.execute('''
    SELECT control_id, COUNT(*) as evidence_count
    FROM evidence
    WHERE collected_date > datetime('now', '-90 days')
    GROUP BY control_id
    ''')

    print("Evidence Collection Report:")
    for row in cursor:
        print(f"Control {row[0]}: {row[1]} pieces of evidence")
```

This creates a simple but functional compliance dashboard that automatically updates based on your evidence database.

**Practical Applications**

- Real-time compliance visibility
- Automated status reporting
- Evidence tracking

**Moving Forward**
Start with basic metrics you check frequently. Add more complexity as you understand what drives action.

**Resource**
Flask Tutorial - Perfect for building simple web dashboards.

# Chapter 10: Advanced Integration Patterns

GRC ENGINEER

The final step in GRC engineering is creating seamless workflows across your entire security and compliance ecosystem. This means making all your tools work together automatically - from evidence collection to reporting to remediation.

Advanced integration isn't about building complex systems - it's about creating simple, reliable connections between your tools. The goal is to automate entire workflows, not just individual tasks.

This is where we bring everything together into comprehensive solutions.

**Quick Start Project**
*Automated Workflow*

1. Connect evidence collection to reporting
2. Add automated notifications

```python
import boto3
import sqlite3
import slack
from datetime import datetime

class ComplianceWorkflow:
    def __init__(self):
        self.db = sqlite3.connect('evidence.db')
        self.slack = slack.WebClient(token='your-token')

    def check_compliance(self):
        # Get AWS Config status
        config = boto3.client('config')
        rules = config.describe_compliance_by_config_rule()

        # Process results
        for rule in rules['ComplianceByConfigRules']:
            # Store in database
            self.db.execute('''
                INSERT INTO compliance_status
                (rule_id, status, check_date)
                VALUES (?, ?, ?)
            ''', (
                rule['ConfigRuleName'],
                rule['Compliance']['ComplianceType'],
                datetime.now()
            ))

            # Alert on non-compliance
            if rule['Compliance']['ComplianceType'] != 'COMPLIANT':
                self.alert_non_compliance(rule)

        self.db.commit()

    def alert_non_compliance(self, rule):
        message = f"""
        :warning: Compliance Alert
        Rule: {rule['ConfigRuleName']}
        Status: {rule['Compliance']['ComplianceType']}
        Action: Review required
        """

        # Send to Slack
        self.slack.chat_postMessage(
            channel='#compliance-alerts',
            text=message
        )

    def generate_summary(self):
        # Get daily summary
        cursor = self.db.execute('''
            SELECT
                status,
                COUNT(*) as count
            FROM compliance_status
            WHERE date(check_date) = date('now')
            GROUP BY status
        ''')

        return cursor.fetchall()
```

This creates a real workflow that connects compliance checking, alerting, and reporting.

**Practical Applications**

- Automated compliance workflows
- Real-time alerting
- Integrated reporting

**Moving Forward**
Start by connecting two systems you use frequently. Build reliable, simple integrations before attempting complex workflows.

**Resource**
AWS Config Rules - Excellent example of automated compliance checking.

# Conclusion: Your Path to GRC Engineering

GRC ENGINEER

**Your 200-Day Learning Journey**

**Days 1-30: Foundation Phase**
- Week 1-2: Study modern infrastructure concepts
- Week 3-4: Learn basic command line operations
- Week 5-6: Start with Python basics
- Week 7-8: Build your first automation script

**Milestone:** Create a basic evidence organization script

**Days 31-80: Data & Integration Phase**
- Weeks 9-10: Learn SQL fundamentals
- Weeks 11-12: Convert one spreadsheet to a database
- Weeks 13-14: Study API basics
- Weeks 15-16: Build first API integration

**Milestone:** Automated evidence collection from one source

**Days 81-140: Infrastructure Phase**
- Weeks 17-18: Learn Infrastructure as Code basics
- Weeks 19-20: Study cloud security concepts
- Weeks 21-22: Build basic security controls
- Weeks 23-24: Implement continuous validation

**Milestone:** Security controls implemented in code

**Days 141-200: Advanced Implementation**
- Weeks 25-26: Build compliance dashboard
- Weeks 27-28: Implement automated workflows
- Weeks 29-30: Create integrated reporting
- Week 31: Testing and refinement
- Week 32: Documentation and handover

**Milestone:** Complete automated GRC workflow

# Making it work

GRC ENGINEER

### Tips for Success

- Start small but start now
- Automate your most painful tasks first
- Build incrementally - don't try to automate everything at once
- Document what you build
- Share your knowledge with your team

### Common Pitfalls to Avoid

- Trying to automate everything immediately
- Building complex solutions for simple problems
- Forgetting to document your automation
- Not involving other team members
- Neglecting error handling in automation

### Measuring Success

- Time saved through automation
- Reduction in manual tasks
- Improved accuracy in compliance data
- Faster response to audit requests
- Better visibility into compliance status

# Final thoughts

GRC Engineering isn't about turning compliance professionals into software developers - it's about using engineering principles to make GRC more effective, efficient, and reliable.

**Remember:**

- Every automated task compounds over time
- Small improvements add up to significant changes
- Focus on solving real problems
- Keep learning and sharing knowledge

**Your path to GRC Engineering starts now. Take that first step, automate that first task, and keep building from there.**

**Want to continue learning? Check out:**

- GRC Engineer Newsletter
- GRC Engineering Podcast
- LinkedIn Learning Course on GRC for the Cloud-Native Revolution

GRC
ENGINEER

# You enjoyed it?
# Let me know how your journey goes on LinkedIn!

Ayoub Fandi