Synechron

A Comprehensive Guide to **Continuous Compliance-as-Code** in the Cloud

Empowering Application Teams in Heterogenous Cloud Environments

Authors:

Ian Tivey
ian.tivey@Synechron.com

Paul Jones
Paul.Jones@synechron.com

Mark Wong
Mark.Wong@synechron.com

David Sewell
David.Sewell@synechron.com

March 2020

Abstract

Public cloud adoption in Financial Services is maturing with several leading organisations adopting a broad set of services from multiple cloud service providers.

To take full advantage of the services on offer, application developers need direct access to the cloud provider's management interfaces, which challenges the traditional, established lines of responsibility between infrastructure teams and application teams.

This blurring of roles is creating anxiety for the teams responsible for enabling the cloud (typically the "Cloud Centre of Excellence") - upon which the burden of compliance is falling - and those responsible for governing the cloud - who are challenged with new operational and SDLC patterns.

As financial services firms prepare for application teams to take on additional compliance responsibilities in this new model, and as the services underpinning the applications become more heterogenous in nature, there is a demand for a more accurate mechanism to describe how controls should behave in the cloud - both in a generalised way and in the context of the individual cloud services and architectures being implemented.

The shape of the control landscape needs to evolve to spread responsibilities across multiple teams, tools, services and third parties. Using the analogy of an apartment block to model the different responsibilities inside our organisation, we present a comprehensive model for managing compliance "as code" - with traceability from the canonical sources of compliance requirements through to the manifestation of those requirements in the cloud and reporting of the effificacy of the controls we have in place.

To address the need for a consistent mechanism for expressing control objectives, we describe our favoured approach of using Behaviour Driven Development (BDD) – a way of using natural language to express complex system requirements, allowing technical and non-technical stakeholders to agree how a system should behave if implemented correctly.

We then address how these objectives can be automatically and regularly tested as part of a Continuous Integration and Continuous Deployment (CICD) pipeline. These tests do not replace dedicated tools and cloud native services for monitoring and compliance reporting. Instead, they provide for a common specification and reporting layer, giving us transparency, traceability and confidence that these services have been configured correctly and are meeting the desired control outcomes. Continuous testing also gives us assurance that the CSP has not made service or platform changes which affect the compliance and security posture or control behaviour.

By combining these concepts, we can create a data model for end-to-end lineage from the underlying provenance of our controls, to control objectives, to behavioural specifications and finally the result of implementation tests and output of continuous compliance monitoring tools.

We present a practical example using Cucumber, a polyglot BDD testing framework, to specify and test controls of object storage services. The code for this example can be found on the Synechron GitHub account at **https:// github.com/ Synechron/compliance-as-code-whitepaper**

Contents

Introduction

Decentralising Controls in a Polycloud PaaS Model

Financial services organisations are beginning to adopt a "polycloud" model – integrating a heterogenous set of PaaS services from multiple providers into their IT ecosystem.

The approach of handling cloud provisioning through a homogenous abstraction layer – whilst attractive from the perspective of control – has been proven to hinder the progress of this shift in strategy, with the abstraction layer unable to keep pace with the demand for access to new cloud services or the feature velocity of the cloud services that have already been integrated.

The emerging approach towards controlling heterogenous polycloud environments is to decentralize the implementation of controls. By combining the controls provided by the CSP control plane with both a provisioning toolchain and monitoring tools to detect and handle non-conformance (an approach commonly referred to as control "guardrails") firms are gaining confidence in their ability to control the risks of making the native APIs of the CSPs directly available to application developers.

Towards Automation & an SDLC for Controls

Given the requirement to embed hundreds of controls across several service providers - with traceability to policies and regulations - the only viable way to manage their deployment and management is via automation.

With a growing number of implementation teams and multiple control authors, there is a high risk of fragmentation, with different interpretations and implementations of the same set of requirements. The esult is inconsistent control outcomes and a greater burden on control owners and the teams responsible for gating and auditing the controls landscape.

To derive greater consistency in control outcomes, pioneering organisations are looking towards best practices from software engineering to build, test, deploy, manage and report on the controls that have been implemented. Tracing the provenance of controls to their underlying regulatory and legal requirements is also extremely important in highly regulated financial services firms. These firms have a need to provide a structured, auditable evidence trail - both to avoid audit activity firedrills and to provide continuous assurance and transparency across polycloud infrastructure estates for internal operational risk teams.

In this paper we discuss:

- How the shared responsibility model is fragmenting with this shift in strategy
- A reference model for implementing cloud resources and controls
- Techniques for building and validating the implementation of controls
- How the reference model can be implemented in your organization using these techniques

Throughout this paper we build on the following terminology:

- Infrastructure-as-code: referring to the use of software engineering practices to deploy and configure cloud resources
- Compliance-as-code: referring to the use of software engineering practices for implementing and validating the efficacy of controls deployed in the cloud
- Continuous compliance-as-code: referring to the regular validation of controls

Revisiting the Shared Responsibility Model

Customer Accountability & Provider Responsibility

For a number of years, the public cloud Shared Responsibility Model has been the starting point for discussing the segregation of roles and responsibilities between the CSP and the Customer. AWS[1] describes the shared responsibility model as:

- CSP responsibility - "Security of the Cloud"
- Customer responsibility - "Security in the Cloud"

Despite the handover of responsibilities to the cloud provider, financial services cloud-related regulations make it very clear that a financial institution cannot abdicate its accountability for the security and operational resilience of the overall system and data.

- **FCA FG 16/5**[2] - "Firms retain full accountability for discharging all of their responsibilities under the regulatory system and cannot delegate responsibility to the service provider."
- **FFIEC Outsourcing Technology Services**[3]- "As with all outsourcing arrangements FI management can outsource the daily responsibilities and expertise; however, they cannot outsource accountability.
- **OCC Bulletin 2013-29**[4]- "A bank's use of third parties does not diminish the responsibility of its board of directors and senior management to ensure that the activity is performed in a safe and sound manner and in compliance with applicable laws."
- **MAS Outsourcing Guidelines**[5]- Institutions are ultimately responsible and accountable formaintaining oversight of (Cloud Services) and managing the attendant risks of adopting (Cloud Services), as in any other form of outsourcing arrangements.

Practically, this means high levels of due diligence need to be undertaken with respect to the responsibilities outsourced to the service provider. Financial institutions need to develop mechanisms to thoroughly evaluate and keep on top of changes to each of the services without overly restricting the agility benefits of adopting services at this level.

[1]https://aws.amazon.com/compliance/shared-responsibility-model

[2]https://www.fca.org.uk/publication/finalised-guidance/fg16-5.pdf

[3]https://ithandbook.ffiec.gov/media/274841/ffiec_itbooklet outsourcingtechnologyservices.pdf

[4]https://www.occ.gov/news-issuances/bulletins/2013/bulletin-2013-29.html

[5]https://www.mas.gov.sg/-/media/MAS/Regulations-and-Financial-Stability/Regulatory-and-Supervisory-Framework/RiskManagement/Outsourcing-Guidelines_Jul-2016-revised-on-5-Oct-2018.pdf

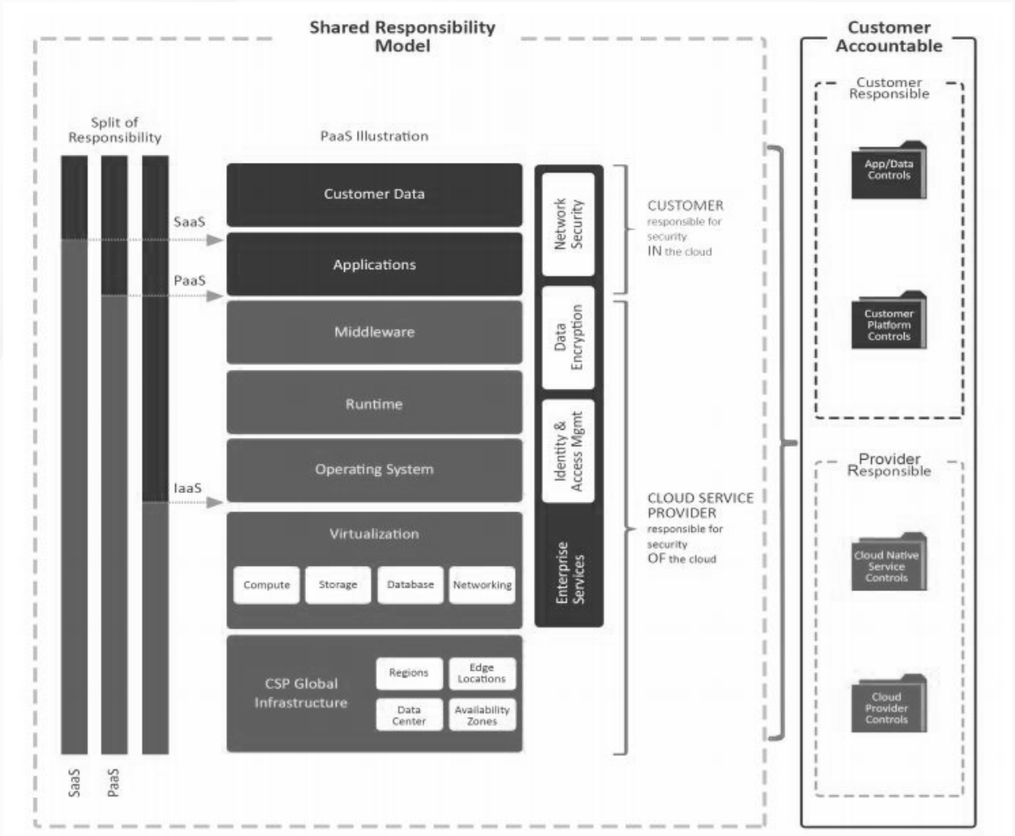

Figure 1 : Shared Responsibility Model

Deconstructing the Shared Responsibility Model

In this section, we deconstruct the Shared Responsibility Model, both in order to describe how control responsibilities become more distributed and heterogeneous as PaaS services and additional service providers are onboarded, and to facilitate the design of an efficient governance model for adopting public cloud services.

Cloud Native Service Controls

Cloud Native Service Controls are the service-aligned controls for which the CSP takes responsibility. At a macro level, more responsibilities are given to the CSP as we move from IaaS to PaaS and then to SaaS (as illustrated in Figure 1). With such a variety of PaaS services on offer, there is a huge variation in the responsibilities assumed by the CSP and in the maturity of controls across the different services. It cannot be assumed that controls that exist for one service be identical, or even exist, for another - both within a single CSP's offerings or across clouds.

Cloud Provider Controls

Cloud Provider Controls are those which are relatively similar across the gamut of services provided by a CSP. Typically, a single periodic assessment will cover controls at this layer. Examples are Data Centre controls, HR controls for CSP staff, Service Management and Contractual controls.

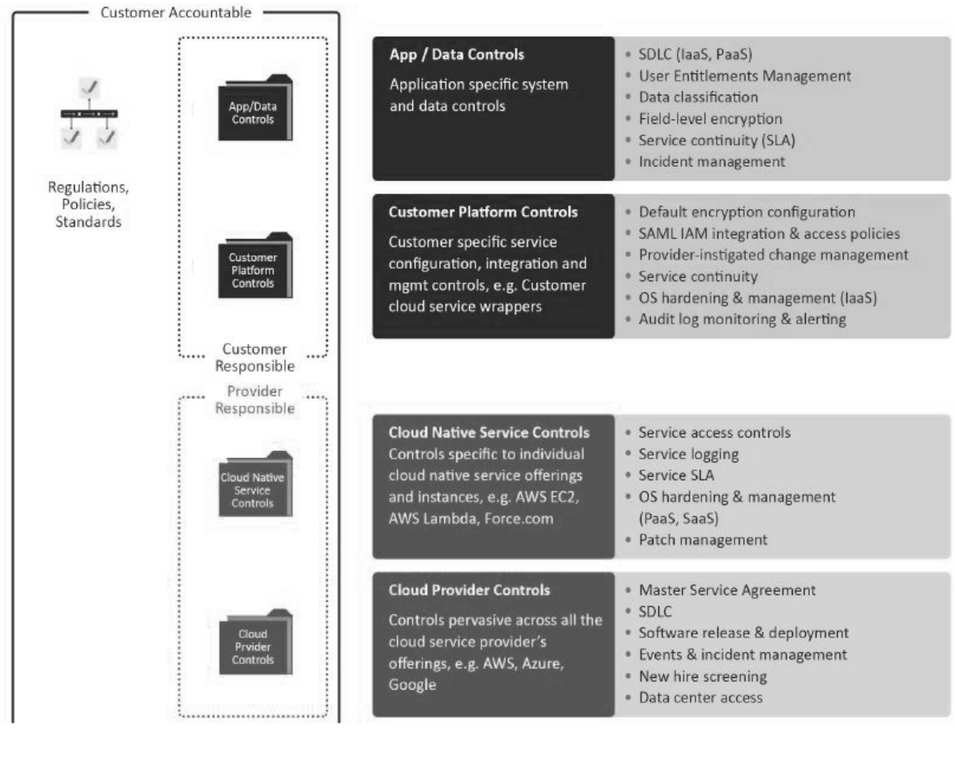

Figure 2: Typical Controls at Each Layer of the Shared Responsibility Model

Customer Platform Controls

Customer Platform Controls are implemented by the customer at a platform level – the customer counterpart to the Cloud Native Service Controls. They are implemented by the customer either using the tools and services provided natively by the CSP or by integrating third party, open source and homegrown tools (often a combination). These can be implemented monolithically, but as firms are becoming more PaaS-aware they are becoming increasingly specific to each service. As we will discuss later in this paper, financial services firms need to be prepared for application teams to take on more responsibility for the "customer platform" as PaaS services are adopted.

Application & Data Controls

Application & Data Controls are the controls implemented specifically for the application - in application code, application processes and by manipulation of the data associated with the application. While many of these will be controls that application teams also need to deal with on-premise there are other controls which on-premise infrastructure and middleware platforms have historically provided transparently but now need to be handled by the application itself (high availability, for example).

> "Financial services firms need to be prepared for application teams to take on more responsibility for the "customer platform" as PaaS servicesare adopted."

Breaking Down Responsibilities within the "Customer Platform"

As we consider how to effectively enable direct developer access to native CSP APIs across a large organisation, the Customer Platform needs to be architected to provide scale benefits in both the delivery of shared integrated services and common overlays for managing risks.

Customer Platform Reference Architecture

There are typically multiple contributors to the Customer Platform across several disciplines in the organisation. The objectives of a modern Customer Platform are to:

- Provide as much transparent autonomy as possible to application teams to manage their application cloud environment
- Provide guardrails to control the boundaries of acceptable risk
- Provide benefits of scale through provision and management of shared and homogenous foundational services

Using the analogy of an Apartment Block (see Figure 3 - Reference architecture for the "Customer Platform"), we can model the different responsibilities inside of our organisation where:

- **"Tenants"** are analogous to "application teams", responsible for the "furnishings" in their "apartment"
- **"Apartments"** have "fixtures and fittings", with more stringent security and compliance
- requirements than furnishings. These need to be managed, or at a minimum signed off, by qualified experts
- **"Resident services"** provide scalable services shared between tenants
- The **"building"** is the foundation for integration of any public cloud service provider
- A **"leasing agent"** is responsible for managing the supply and demand of tenancy services in our fluid public cloud environment

Using this model to improve organisational understanding, we can make determinations of which parts of the organisation are responsible for implementing and deploying any specific control, and responsible for reporting and continuous assessment of compliance.

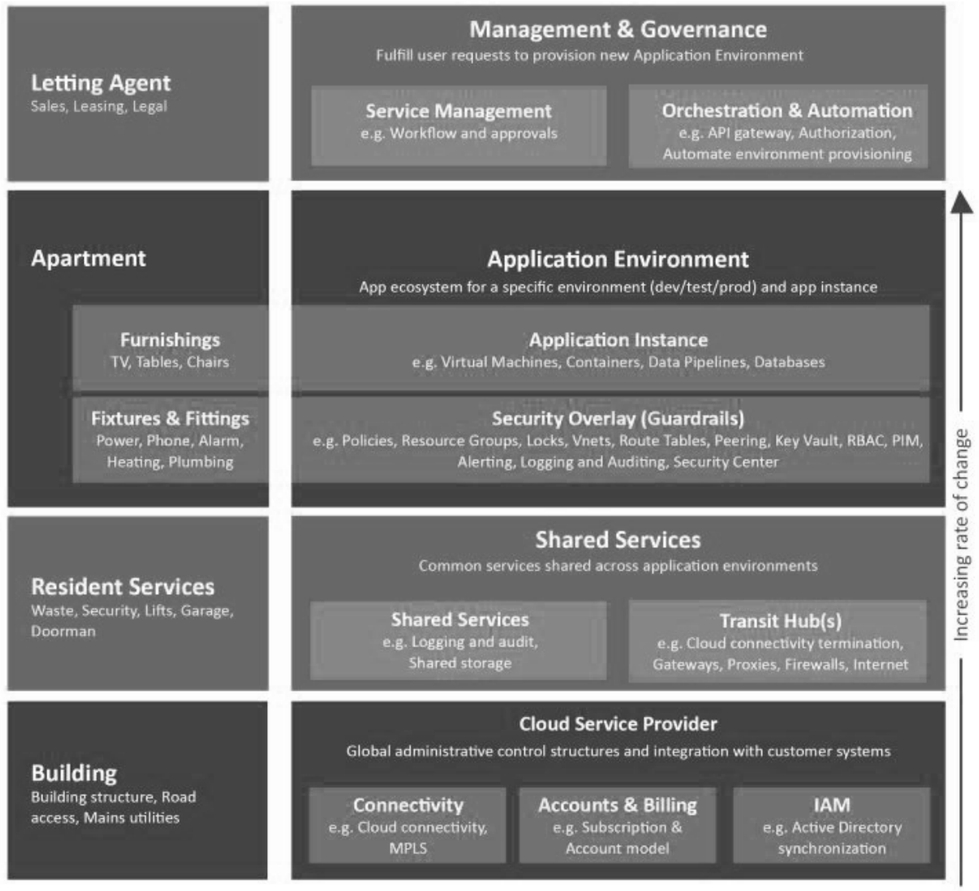

Figure 3: Reference Architecture for the "Customer Platform"

Building

There are services, like gas mains, that provide the fundamental building blocks for access to the cloud. The layout of the billing and accounting structure, identity management and long-line network connectivity typically sit in this layer. To change these services is extremely disruptive and risky and so should be held to the highest standards of service management.

Resident Services

Similar to the Building, there are several shared services which need to be in place to provide core, highly specialised capabilities from which every tenant can benefit. Examples in this category are security log aggregation and analysis, or the transit networking to provide connectivity between different tenants. Central engineering and management of these services provides scale benefits and typically there is very little tenant-specific customisation. Like the building, changing these services is disruptive and risky.

The Apartment Unit

The Apartment Unit is the shell structure into which tenants can deploy furnishings tailored to their specific requirements. There are notable differences between the different cloud service providers in the concrete manifestation of an apartment unit. In AWS this is typically an "Account", in Azure a "Subscription" or "Resource Group" and in Google Cloud Platform a "Project". There may also be hierarchical structures in place from which attributes of the apartment unit are inherited (for which we can use the analogy of a "floor").

The aim is to standardize the look and feel of apartment units (analogous to standard layout 1-, 2- and 3-bed apartments), whilst also making provisions for edge cases which require more specialisation (analogous to the Penthouse).

Furnishings

Like tenants in an apartment unit who are free to choose their furniture and decorations, application teams have the freedom to choose services, the configuration of those services and how they integrate together to optimally support their application.

We can adopt a model where application teams can choose to use curated blueprints to deploy services using tools supported by the cloud team (think "IKEA furniture") or build their own from the ground up using the tools they are most comfortable with (think "custom walnut furniture").

This is not to say that application teams have free rein to do whatever they like. Strict rules still apply and, while guardrails can be put in place and the use of compliance-stamped pre-canned blueprints encouraged, application teams will need to take responsibility for the configuration of resources which they own, obtaining sign-off from the appropriate compliance teams before go-live or the release of material changes and dealing with auditors.

Fixtures & Fittings

In an individual apartment unit, services such as the plumbing, gas outlets and electrical wiring are generally fixed in place. Remodeling these fixtures requires a level of expertise for which, given the risk of getting these wrong, most tenants choose to or are mandated to call in an expert. Regardless of who does the work, in many cases modifications require sign-off from certified professionals before being put to use - the risk of using a badly installed gas stove could be catastrophic, for example.

Similarly in the cloud, much of the tenant-specific networking, roles management, audit logging and policy management require a level of expertise that most application teams don't have embedded into their squads, and the risk of getting these wrong can result in material damage. Application teams can, if they wfeel they have the knowledge, attempt to modify deployment artifacts for these resources by raising a pull request against the appropriate source code repository.

An emerging set of services and automated compliance tools are helping to enforce compliance both in- and out-of-band - supporting the objective of allowing direct developer access to CSP APIs. Each of the major CSPs have native offerings and there are several commercial tools and open source projects which integrate with the CSP APIs in order to achieve similar goals. These "guardrails" should be considered part of the "fixtures and fittings" to ensure that segregation of duties controls are in place.

These "fixtures and fittings" would usually be delivered as part of the "apartment unit" provisioning and any specific tailoring for application teams (e.g. virtual firewall ports) should be applied via code in a manner that is completely reproducible at any point in the future.

Lettings Agent

To deliver these architectural features we need a set of mechanics to handle requests for new apartment units, modifications to existing units and destruction of units which are surplus to requirements. This should be built API-first with any GUI components built on top of that API. These services require back-end integrations with the systems and data sources required for determining whether the request can be fulfilled - such as authentication and authorization, financial management systems, information about data residency and disposal requirements and service usage approvals. They may also be integrated with existing inventory platforms and other data sources.

For the most part, the aim is to have processes which prime these systems so that business-as-usual requests get immediately approved and fulfilled.

Implementing Controls in the "Customer Platform"

Operational Responsibilities

The apartment block abstraction described previously implies a distribution of roles and responsibilities across several different teams. Synechron strongly advocates mandating the use of Infrastructure-as-Code wherever possible, controlled via software development techniques. This not only provides repeatable and auditable outcomes, but allows the delegation of operational activities.

For each layer in the reference architecture, there are three sets of responsibilities we need to consider:

- **Responsible for the code -** The build, test, release and maintenance of the code falls under this team's responsibilities. This team may wholly write the code themselves, be responsible for reviewing and approving pull requests from other development teams, or both.
- **Responsible for instigating the execution of code -** The team which kicks off execution of code to deliver cloud resources. In some cases this might be via an API call which performs several control checks before kicking off a pipeline to deliver the requisite resources.
- **Responsible for compliance -** The team responsible for ensuring the code results in compliant resources, including obtaining sign-off from the appropriate compliance teams.

In some cases the same team will be responsible for two or more of these, in others it will be different teams.

The table below describes the typical responsible party for each layer in the stack. "Teams" is used as a loose definition – the exact organisational party will depend on the organisational operating model being followed.

	Automation Code	Automation Instigation	Automation Compliance
Building	Cloud Platform Teams	Cloud Platform Teams	Cloud Platform Teams
Resident Services	Cloud Platform Teams	Cloud Platform Teams	Cloud Platform Teams
Apartment	Cloud Platform Teams*	Application Teams*	Cloud Platform Teams*
Fixtures & Fittings			
Furnishings	Application Teams	Application Teams	Application Teams
Letting Agent	Cloud Platform Teams	Application Teams	Cloud Governance Teams

*"Apartment" and "Fixtures & Fittings" typically delivered together

Implementing Controls

As most firms aim to "shift-left" in their application development practices, we should follow a similar approach to the cloud by introducing controls as early as practically possible in the development lifecycle to avoid any last minute surprises when trying to push configurations out into Production.

In this architectural model, where controls are federated across different teams, there are two styles of controls implementation (which should be used in combination to provide a comprehensive set of controls across the platform): 1. Control enforcement 2. Control validation

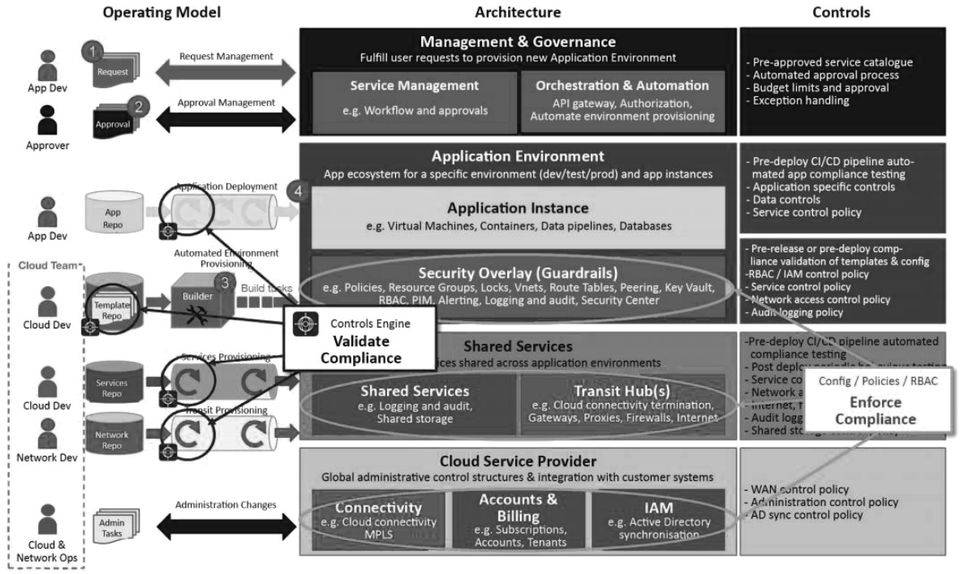

Figure 4: Implementing Controls Across the Different Layers in the Reference Architecture

Control Enforcement

There are several different techniques we can use to enforce compliance in the cloud. A well-rounded Enterprise adoption program will use all of these techniques across different parts of the Customer Platform.

Pre-Built Resources

By only allowing application teams to integrate with pre-built resources in the cloud, we enforce compliance by simply not providing options to configure those resources in a non-compliant manner. These could either be shared services or prescriptive resources dedicated to each application team that are delivered as part of their "Apartment Units."

The resources deployed at the "Building", "Resident Services" and "Fixtures & Fittings" layers fall into this category. Typically, we try to avoid pre-building resources at the "Furnishings" level (although we may provide mechanisms to assist application teams in deploying compliant furnishings).

Preventative Controls

Cloud providers offer services integrated with their control plane that can be configured to constrain or prevent certain user actions and behaviours, with the effect of blocking the non-compliant configuration of a resource.

Preventative controls provide immediate feedback to the developer, leaving the developer fully in control of the resources deployed into their environment. They are our first line of defence in meeting the objective of providing native CSP API access for developers and typically delivered as part of the "Fixtures & Fittings" in every apartment unit.

Detective-Corrective Controls

There are various native cloud provider services, commercial tools and open source projects which can be configured to periodically scan the environment or receive triggers when a resource is created, compare the configuration of resources against encoded policies and take action to correct any non-compliant resources - such as destroying them, quarantining them for human inspection or just alerting for manual intervention.

Detective-corrective controls are less desirable than preventative controls. They can cause unexpected behaviour with deterministic state management tools, such as Terraform, which report successful deployment of resources that are then mutated or destroyed outside of the visibility of the developer's tooling. They still form a major part of our tooling, however, where is it not possible to implement preventative controls.

Wrapped Resources

For middleware services which are complex to build in a compliant manner and which involve multiple control planes (one example being the various flavours of Kubernetes service) we can wrap the multi-stage delivery process in an API or provide it as a native template in one of our supported templating languages to assist development teams in building compliant furnishings.

We can also provide a common set of tests which provide a level of assurance that these complex resources are compliant, which teams can integrate into their "Furnishings" pipeline.

Control Validation

Use of Infrastructure-as-Code allows us to adopt best practices from software delivery by performing code inspections and testing as part of the delivery pipeline to validate that the resources which will be delivered will be compliant.

There are different levels of inspection and testing we can perform to increase our confidence level in the efficacy of our controls and the compliance of deployed resources.

Unfortunately there is no "silver bullet" to validate that our controls are effective: we need to make tradeoffs when choosing how deep we go, how frequently we test and much risk we are willing to take on in the validation of our controls.

As we get into the territory of actively deploying resources to validate our guardrails, the test suite can take a long time to execute and, in some cases, can expose the firm to unacceptable risk if, for example, noncompliant resources are successfully deployed to a Production environment. A complete approach will shift both "left" and "right" in the SDLC process, being both integrated into the feedback a developer receives whilst developing their infrastructure code but also stressed via active "chaos" testing in Production environments.

Peer Review

As part of the development workflow, when a pull request is raised by a developer, it should be peer reviewed and approved before being merged. Reviewers/approvers could be developers in the same development team and/or control partners in compliance teams.

Fail Fast

Before trying to run potentially time-consuming integration tests, we can do as much as possible to fail fast. We can list our test code and deployment artifacts for hygiene and validity before we try to execute them. Cloud Providers provide a JSON schema against which we can validate that not only is our JSON syntactically correct, but also semantically correct. There are also open source[6] and commercial[7] tools which can be used by developers to validate their code against the guardrails they will face in deployment.

Code Analysis

When new code is checked-in to our source repository we can automatically inspect it against specific requirements and reject the pull request if any requirements are violated. We might choose to check for test coverage and sufficently low numbers of "code smells", or use tools which are closely coupled to our modelling language to define configuration rules. We might also automatically suggest (with a bot) the most appropriate reviewer of a pull request based on the code that is being edited.

Resource Inspection

After we deploy resources, both in our non-production and production environments, we can pull the resultant configuration of those resources via the CSP's APIs, parse and inspect it for specific compliance requirements.

Tools such as Open Policy Agent (OPA) can be used to declare validation logic and inspect the JSON configurations returned in the CSP API.

Active Resource Deployment

We can test the efficacy of preventative and detective-corrective controls by attempting to deploy resources and perform actions which both violate and comply with our policy boundaries and make assertions based on the outcome of those tests. We can do this on a regular basis to ensure that any changes made by the cloud provider have not altered the effect of our guardrails, which would otherwise be opaque.

This type of test is particularly effective at testing the efficacy of tools which scan the environment and perform actions on non-compliant resources.

[6]https://github.com/open-policy-agent/conftest

[7]https://www.hashicorp.com/sentinel/

Penetration and Vulnerability Testing

Once resource and configuration artifacts are deployed we can perform penetration tests to determine the efficacy of network controls. We can also attempt to exploit known platform-level attack vectors, such as attempting to hijack identities via the CSP's metadata service.

Chaos Engineering

Chaos Engineering started as a mechanism for actively testing the efficacy of high availability architectures by pseudo-randomly destroying resources and logically taking large parts of the system offline, even in Production environments. It now covers a broader set of activities, such as injecting malformed messages onto the wire, regularly attempting to exploit known vulnerabilities and deploying rogue resources and software into the environment to mimic a bad actor.

Testing of control implementations in non-production environments is likely to be "cleaner" than in the production environment, so for the highest level of confidence we should aim to perform tests which actively violate the controls in Production – either continuously, or periodically.

Democratising Compliance

The blueprint approach is a technique that can be used to facilitate compliance and consistency in the way "furniture"-level resources are deployed. In this model, pre-approved templates and workflow patterns can be encoded and held in a common code repository.

Rather than having a single team take on the responsibility for creating and maintaining them, an "innersource" model, where blueprint artifacts are managed as an open source project but for internal use, allows any member of the firm to take on responsibility for maintaining compliant resource definitions by raising pull requests against the codebase.

Accountability can be retained by security and compliance teams by having them be part of the governance process, responding to and approving pull requests and making decisions about when changes are merged into the master branch.

Running such an inner-source model requires that all blueprint definitions in the repository have comprehensive automated tests, which we will discuss in more detail later in this paper, and that application teams leveraging these templates are also able to run regression tests against their own environment. In a sense, the tests are more important than the deployment artifacts – having high quality tests in a shared repository allows development teams to decide on their own tools for managing their resources, with controls evidence generated by a common set of trusted tests generating a common set of compliance artifacts.

Having high quality tests in a shared repository allows developmentteams to decide on their own tools for managing their resources, with controls evidence generated by a common set of trusted tests generating a common set of compliance artifacts.

Implementing the Reference Architecture

Defining Common Control Objectives

We have long espoused the need[8] to invest in a common set of control objectives which are fit for purpose for Cloud. Where there are multiple regulatory bodies and industry standards to which we must adhere, a common control set provides a reference against which cloud services can be sourced, onboarded, configured and integrated. These common controls may form a dedicated set of objectives for the cloud or be folded into the firm's existing, general IT frameworks. Having a common controls reference is becoming increasingly important with the adoption of PaaS services, with control responsibilities for the Cloud Platform fragmenting across the firm.

Having a common controls reference is becoming increasingly important with the adoption of PaaS services, with control responsibilities for the Cloud Platform fragmenting across the firm.

Mapping this common control set to the underlying legal and standards documents is also an important first step in tracing the provenance of controls implementations to the underlying requirements.

While the typical starting point for Cloud Controls is security, a complete set of controls will cover the full spectrum of control concerns when adopting public cloud:

- Availability, Continuity and Resilience
- Contracts and Legal
- Exit Management
- Human Resources
- Operations and Service Management
- Privacy
- Risk Management Practices
- Security
- Vendor Management

In our experience developing Cloud Control Objectives and implementing cloud controls for several financial services organisations, we advise taking the following into account:

- Each objective is traceable to one or more legal / regulatory requirements and vice versa, supported by other standards and best practice documents
- They address concerns across all control disciplines - security, sourcing, availability, continuity, service management and vendor management
- They acknowledge that the approaches and best practices differ between traditional IT and Cloud Computing
- They are phrased in such a way as to avoid the objective being solved using only one implementation technique, unless absolutely necessary
- They are high level and flexible enough to allow for the different types of service and service providers encountered as cloud services are onboarded
- The total number of objectives is manageable in size. In our experience, 150-200 is a good number to aim for

[8] https://www.synechron.com/insights/whitepapers/cloud-controls-for- financial-services/

Creating Traceable Control Implementations

With our common control objectives in place, mapped back to their various source artifacts, we can then begin to map our objectives forward to concrete implementations of controls, providing end-toend traceability from the originating compliance requirements to the details of how the different parties and teams involved in the delivery of applications and data in the public cloud are going about their responsibilities.

Through our work, we have come to the realization that, as firms bring on multiple cloud providers, adopt heterogenous PaaS services and federate out the responsibility for delivering different controls, there is a layer of precision missing between the high level, unstructured objective statement (which is intentionally broad and open to interpretation) and the concrete implementation.

The result is inconsistency in how control objectives are interpreted (even within the same teams), challenges in communication between implementation, compliance and audit teams, and overly rigid diktat on how individual controls must be implemented. This often manifests itself in multiple spreadsheets with overlapping, conflicting requirements written in their own unstructured or structured ways.

Compliance programs need a conventional way for control owners to structure and maintain detailed control requirements for the teams responsible for implementing and integrating any individual service or part of the platform - a common reference for continuous validation and attestation reporting, with the flexibility to choose and modify how any particular requirement is implemented as the underlying cloud service offerings change.

Our favoured technique for articulating these types of requirements is Behaviour Driven Development (BDD), a technique borrowed from software engineering.

Compliance-as-Code Using Behaviour Driven Design Specifications

Behaviour Driven Development (BDD) is a technique for formalising a shared understanding (between technical and non-technical staff) of how a system should behave.

It is largely facilitated through the use of a domain-specific language (DSL) using natural language constructs. There are a handful of popular BDD DSLs, but in this paper we will focus on Gherkin which is the format for the Cucumber framework[9].

We favour BDD for compliance-as-code because it is particularly effective in complex scenarios with multiple stakeholders. Focusing on behaviours avoids straying into implementation details in our requirements, allowing us to make specifications portable across cloud providers, services and the concrete control implementations.

Each BDD specification forms a “contract” between all of the stakeholders involved in compliance activities and forms the basis for generating auditable evidence.

Each BDD specification forms a “contract” between all of the stakeholders involved in compliance activities and forms the basis for generating auditable evidence. There is, naturally, effort involved in gaining consensus and sign-off by stakeholders on the specifications so, while the implementation artifacts and testing of the controls linked to the requirements in the BDD specifications may change often, the specifications themselves should remain relatively static. Investing up-front in high quality specifications is important to avoid later re-work and churn in re-gaining consensus.

Figure 5 shows the different code artifacts in the compliance-as-code library.

From the BDD specification we build code artifacts, implementing resources which behave according to the specification and tests which attest that the deployment artifacts exhibit the specified behaviours.

As we will see in the following example, when writing the specifications it is important to have sympathy for how tests can be logically implemented. The most effective approach to BDD involves an iterative process between those writing the specifications and those creating the deployment artifacts and the control tests to ensure the specifications result in achievable and robust test coverage.

[9]https://cucumber.io

```
Feature: Object Storage Encryption in flight
    As a data security Control Owner
    I want to ensure that suitable security controls are applied to object storage
    So that my organisation is protected against data leakage via misconfiguration

    Scenario Outline: Only allow secure transfer storage bucket
        Given security controls that restrict data from being unencrypted in flight
        When we provision an object storage bucket
        And http access is "<http_option>"
        And https access is "<https_option>"
        Then creation will "<Result>" with an error matching "<Error Description>"

        Examples:
            | http_option | http_option | Result  | Error Description                                    |
            | enabled     | disabled    | Fail    | Storage Buckets must not be accessible via plain HTTP |
            | enabled     | enabled     | Fail    | Storage Buckets must not be accessible via plain HTTP |
            | disabled    | enabled     | Succeed |                                                      |
```

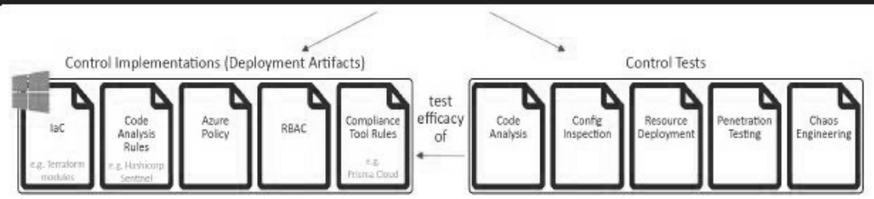

Figure 5 : Code Artifacts in the Compliance-As-Code library (Azure Example)

BDD Basics

Let us describe the fundamental BDD building blocks: Features, Scenarios and Scenario Outlines: Features

Features

A BDD specification starts with a Feature. This is typically something a Product Owner or a Control Owner might write in the form of a User Story.

Scenarios & Scenario Outlines

The behaviours which support the Feature are defined by a series of Scenarios. Each Scenario is expressed using a set of key words followed by natural language. The main key words are:

- **"Given"** – this is intended to describe the state of the system at the beginning of the scenario (the "scene")
- **"When"** – describes the event or action to trigger the scenario
- **"Then"** – the observable desired response to the When triggers

Any of these keys words can have multiple statements under them, using the keywords "And" and "But."

It's important to note that the key words themselves don't apply any particular behaviour to the test implementation, they are there in order to provide structure and convention to the Feature definition.

To avoid repetition, when we want to define identical Scenarios with a range of possible states, triggers and outcomes, we can use a Scenario Outline. In the case of a Scenario Outline we define variables in our "Given"/"When"/"Then" statements and provide a data table which defines values for each of the variables under separate tests.

There are also other Gherkin keywords for describing more complex features – Background, Rule. The full specification can be found at https://cucumber.io/docs/gherkin/reference/

We give an example of Feature and associated Scenarios below.

BDD Example - Object Storage Encryption in Flight

The following example is a Scenario that describes the behaviours a user should experience when controls restricting network traffic to HTTPS are applied to storage buckets. The goal was to write the specification in such a way as to be portable across different Cloud Providers.

As we came to implementing the deployment artifacts and the control validation tests for this scenario we went through a couple of iterations, which we will step through in this example to demonstrate the importance of understanding the underlying cloud implementation when writing the BDD specifications.

Feature

The user story for the feature describes the need for our specification and high level goal:

```
Feature:  Object Storage Encryption in Flight

As a Cloud Security Architect
I want to ensure that suitable security controls are applied to Object Storage
So that my organisation is not vulnerable to interception of data in transit
```

Scenario 1 – A Preventative Control

Initially, we used a Scenario Outline to describe the different HTTP/HTTPS on/off permutations, the expected result under each permutation and the semantics of the error response (not the actual error) that we'd expect to receive from the cloud provider.

```
Scenario Outline:  Prevent Creation of Object Storage Without Encryption in Flight
    Given security controls that restrict data from being unencrypted in flight
    When we provision an Object Storage bucket
    And http access is "<HTTP Option>"
    And https access is "<HTTPS Option>"
    Then creation will "<Result>" with an error matching "<Error Description>"
    Examples:
        | HTTP Option | HTTPS Option | Result  | Error Description                                    |
        | enabled     | disabled     | Fail    | Storage Buckets must not be accessible via plain HTTP |
        | enabled     | enabled      | Fail    | Storage Buckets must not be accessible via plain HTTP |
        | disabled    | enabled      | Succeed |                                                      |
```

Whilst this is a trivial example, it demonstrates that BDD brings a level of precision to our control definitions that our control objectives do not (nor should be designed to do), yet the requirements remain quite readable by anyone vaguely familiar with object storage. A typical control objective linked to this requirement would be, "Encrypt All Sensitive Information in Transit"[10].

In writing this example, we had an Azure Storage Account implementation in mind. When we wanted to write tests for the same specification in AWS, we realized that the specification required some changes to be portable across the two clouds.

[10]CIS Controls Version 7

Scenario 2 – Adding a Detective Control

Azure has simple options to toggle HTTP and HTTPS on Storage Accounts and Azure Policy has the capability to outright prevent the creation of Storage Accounts which do not meet the policy's requirements.

On AWS, AWS Config works by detecting the creation of non-compliant resources and then taking action.

Because the two implementations are different – preventative vs detective - we decided to create a separate "detective" scenario, written to be portable across other CSPs and third party tools which work in detective mode:

```
@preventative
@csp.azure
Scenario Outline: Prevent Creation of Object Storage Without Encryption in Flight
    Given security controls that restrict data from being unencrypted in flight
    When we provision an Object Storage bucket
    And http access is "<HTTP Option>"
    And https access is "<HTTPS Option>"
    Then creation will "<Result>" with an error matching "<Error Description>"
    Examples:
        | HTTP Option | HTTPS Option | Result  | Error Description                                  |
        | enabled     | disabled     | Fail    | Storage Buckets must not be accessible via plain HTTP |
        | enabled     | enabled      | Fail    | Storage Buckets must not be accessible via plain HTTP |
        | disabled    | enabled      | Succeed |                                                    |
@detective
@csp.azure
@csp.aws
Scenario: Delete Object Storage if Creation of Object Storage Without Encryption in Flight is Detected
    Given there is a detective capability for creation of Object Storage with unencrypted data transfer enabled
    And the capability for detecting the creation of Object Storage with unencrypted data transfer enabled is active
    When an Object Storage bucket is created with unencrypted data transfer enabled
    Then the detective capability detects the creation of Object Storage with unencrypted data transfer enabled
    And the detective capability enforces encrypted data transfer on the Object Storage Bucket
```

We have used "@tags" to annotate each Scenario, indicating that one Scenario is applicable to preventative controls and the other is applicable to detective controls. These tags can be used in the implementation of Cucumber tests associated with this scenario.

BDD Principles

When writing BDD Scenarios, there are some key principles to keep in mind:

Don't include implementation details in Scenarios

BDD forms a contract across the firm and if the BDD specification requires significant re-work, should a provider modify their approach or release something better, then there is a lot of work involved in reestablishing the contract.

Example: rather than "Given an Azure Policy is in place…", which is a very specific implementation of the control, use "Given security controls are in place…", which gives flexibility in how those controls are implemented.

Focus on semantics, not specifics

Example: error messages from CSPs and services will vary and may change over time. If error messages in your BDD Feature are semantically correct, they can be mapped to concrete CSP error messages in tests implementations.

Create a tag namespace for annotating Scenarios within a Feature

Examples: "@csp.gcp" indicates a scenario which applies to Google Cloud Platform. "@service.aks" indicates a scenario which applies to Azure Kubernetes Service. "@preventative" indicates a scenario that describes a preventative behavior. With these tags in place, not only do the Features have additional metadata, but we can choose to only execute logical tests against, for example, GCP or AKS, to suit our needs.

Be vivid

This one comes from the Gherkin reference site[11], in reference to the Background keyword. We find it useful across the entire feature specification - use colourful names, and try to tell a story. The human brain keeps track of stories much better than it keeps track of names like "User A", "User B", "Site 1", and so on.

Test Implementation

With our BDD Features agreed, we can begin implementing Cucumber tests. Our example tests are written in Go using the official Cucumber BDD framework for Golang, "Godog" from Data Dog[12].

It is beyond the scope of this whitepaper to detail how tests for above Scenario can be implemented.

Detailed write-ups on how we have implemented tests can be found on **http://medium.com/Synechron**

The code for our tests can be found on **https://github.com/Synechron/compliance-as-code-whitepaper**

Test Output

The output of our tests aligns to the originating BDD Feature, showing which steps passed, which failedand which were skipped.

Test steps might be skipped if a preceding steps fails, if a Feature is defined but a physical implementation is not.

Because the test output is machine-readable JSON, we can choose to visualise it in different ways. We've chosen to use the off-the-shelf Cucumber HTML Reporter[13], which produces output like that shown in Figure 6.

[11] https://cucumber.io/docs/gherkin/reference/

[12] https://github.com/cucumber/godog

[13] https://github.com/gkushang/cucumber-html-reporter

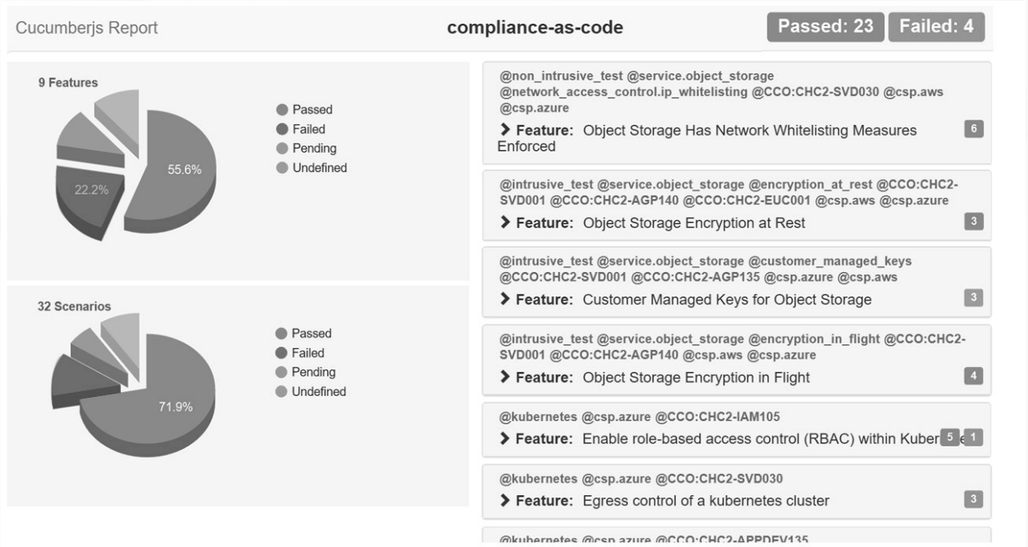

Figure 6 : Example Test Output Using Cucumber html Reporter

Putting it all Together: Compliance-as-Code

With a library of Common Control Objectives, BDD specifications, Infrastructure as Code deployment artifacts and Cucumber tests for the CSPs and cloud services we have onboarded, we have a comprehensive model for managing compliance "as code" - with traceability from the canonical sources of compliance requirements through to the manifestation of those requirements in the cloud and reporting of the efficacy of the controls we have in place.

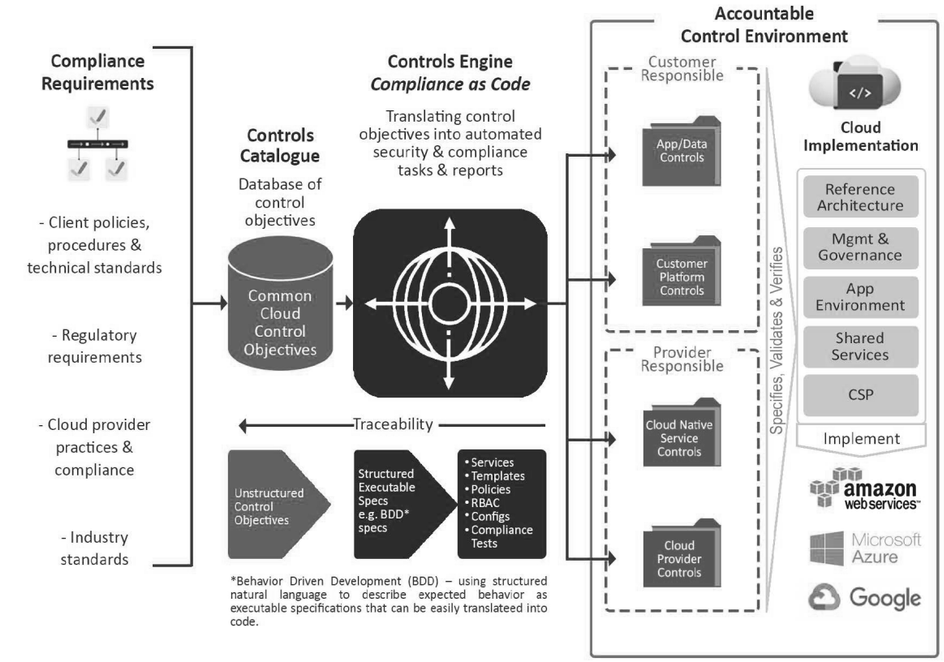

Figure 7: End-to-End Control Traceability

Let's look at how all of these artifacts tie together to provide a comprehensive approach towards Compliance-as-Code.

A Compliance-as-Code Development Pipeline

In our reference architecture (see Figure 4) we visualized several pipelines for managing different parts of the Customer Platform.

Each of these pipelines can be implemented as a common set of tasks, shown in Figure 8.

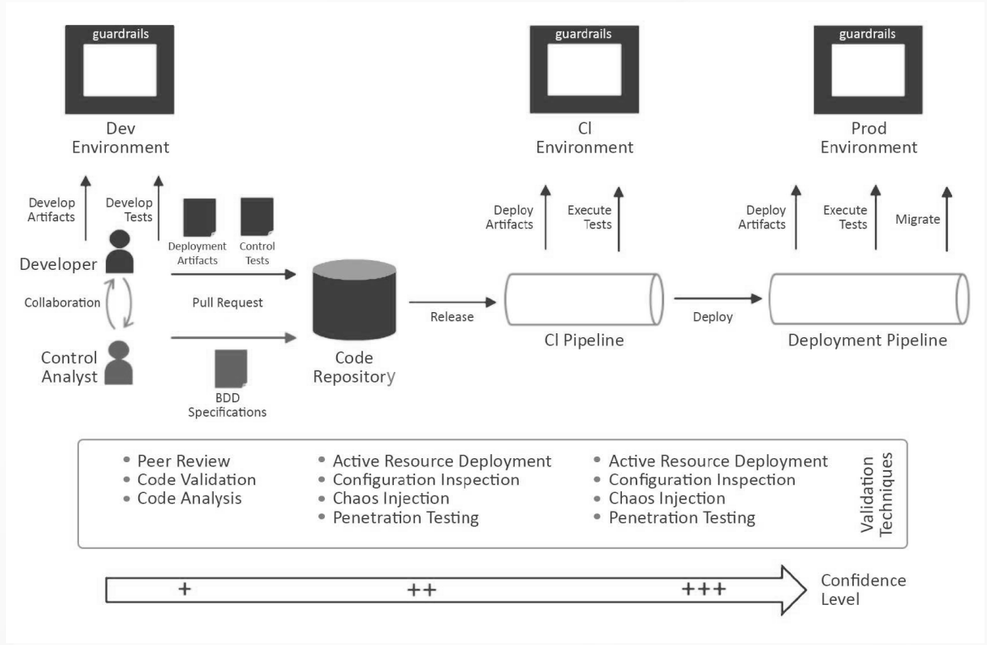

Figure 8: Compliance-as-Code Development Pipeline

Enforcing compliance - via guardrails in the environment - and implementing validation as far "left" as possible in the pipeline provides developers with early feedback for rapidly fixing issues and ensuring deployment artifacts have a high probability of being compliant before the code is released.

Extending validation testing of our implementations as far "right" as possible and performing increasingly intrusive validation tests ("chaos engineering") provides increasing levels of confidence that our controls are effective.

Moving from left-to-right in Figure 8:

Writing Features, Deployment Artifacts and Tests

Controls Analysts and Developers work together to curate BDD specifications, deployment artifacts and control tests in a development environment. Where appropriate, the development environment should have guardrails in place to enforce compliance, providingearly feedback on the changes required to the deployment artifacts to meet compliance requirements, or to highlight the changes required to the guardrails to facilitate the service being onboarded.

Incorporating Changes into the Core Codebase

When changes to BDD specifications, deployment artifacts and test code are ready, a pull request is raised against the main code branch (requesting that new changes are incorporated into the core codebase). At a minimum, the pull request should be subject to peer review. In addition it can be subject to code validation and analysis, as we described earlier.

Continuous Integration

Both before and after a merge, the proposed changes will kick off a continuous integration (CI) pipeline which deploys the artifacts in the repository and executes the associated control validation tests. If the tests pass (and our other hygiene and quality gates are met) then the code is ready to be merged and, in a Continuous Delivery model, ready for deployment in the Production environment.

With layered controls providing "defence in depth," it can be difficult to test specific controls deeper in the control stack. For example, if IP whitelisting is blocking access to a resource it will prevent testing of other controls applied to that resource. In our CI environment we should aim to test all of the controls.

Continuous Testing

In the absence of a code release, the CI pipeline may be periodically kicked off to provide continuous testing of the efficacy of the controls. Because the cloud providers are continually making opaque changes to the platform, this can provide alerts to changes which would otherwise go undetected.

Release

When ready, the deployment pipeline releases the deployment artifacts. As part of the deployment process we may also run through the full suite of control validation tests to give us confidence that any differences between CI and Production haven't affected our controls, although some tests may be omitted for risk reasons (for example, attempting to open SSH port 22 to the internet on the virtual firewall).

Once the tests have passed, if we are following a "blue/green" or "canary" style deployment we can migrate the Production environment over to the newly deployed resources and retire the previous version.

Control Validation ("Smoke") Tests

Post-deployment, we can also periodically or continually perform control validation tests to ensure entropy hasn't caused any of the controls or resources in our environment to become non-compliant.

A Practical Compliance-as-Code End-to-End Example - Enforcement Controls

In this section we present a practical example of how enforcement controls can be delivered as code when delivering "apartment units" to development teams.

As discussed earlier in this paper, these enforcement controls form the "fixtures and fittings" of the apartment, allowing us to meet our stated goal of allowing developers to access the CSPs' native APIs with appropriate guardrails in place to manage compliance risks.

Figure 9 shows the architecture of the delivery pipeline.

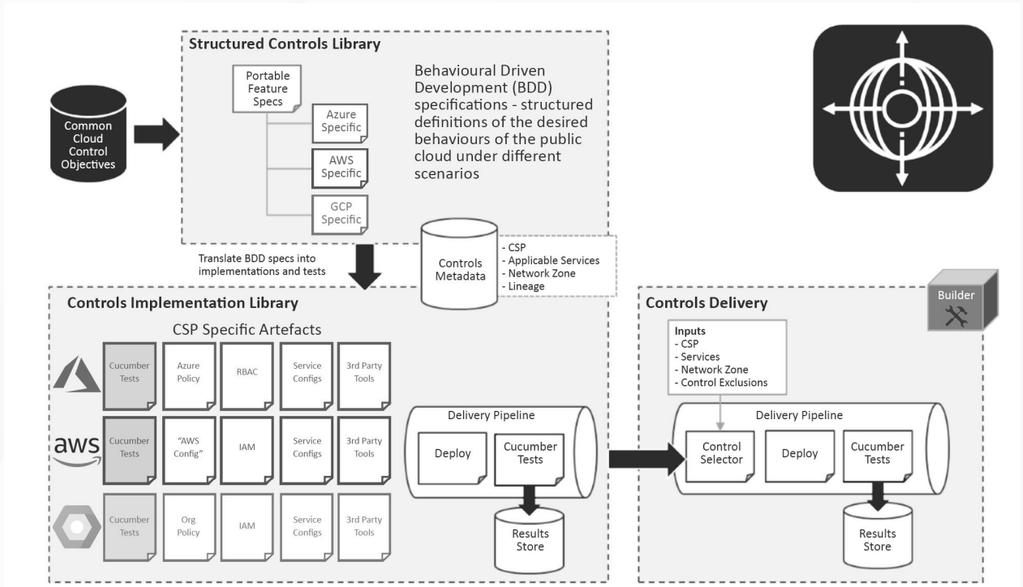

Figure 9: Enforcement Controls Delivery PIpeline

Structured Controls Library

The "Structured Controls Library" is the library of BDD specifications. Each specification has a series of metadata associated with it, such as:

- **Lineage:** the upstream Cloud Control Objective(s) implemented by the specification.
- **CSPs:** to which the specification applies. Whilst the aim is to write the specifications in a portable manner, it is not always practical to do so due to notable differences in the CSP service architecture.
- **Applicable Services:** for cases where specifications are written for specific services. Typically, this will be due to secondary control planes, such as a Kubernetes service.
- **Network Zone:** there are notable differences in the controls in an Internet-facing zone which accepts client connections versus virtual network zones where databases are hosted.
- **Data Sensitivity:** stronger or weaker controls for greater or lesser sensitivities of data.

Controls Implementation Library

The "Controls Implementation Library" is the collection of Deployment Artifacts and Cucumber tests for the "apartment" and "fixtures and fittings", plus the associated metadata/attributes. The different types of artifact and testing strategies that can be found are discussed earlier in this paper.

The underlying implementation of each control and associated tests will be specific to each CSP and their individual service offerings. There will also be specifics across other dimensions such as data sensitivity, network zone and system criticality.

It is beyond the scope of this whitepaper to go into detail on how this can be achieved.
Please refer to **https://medium.com/Synechron** for detailed technical write-ups and **https://github.com/Synechron/compliance-as-code-whitepaper** for code examples.

Continuous Integration

The purpose of the Continuous Integration (CI) pipeline is to determine the efficacy of the control implementations in our library, including the configuration of continuous compliance tools which monitor the cloud environments and alert for non-compliance.

Continually running integration tests against the full library of controls will quickly catch and alert to any changes affecting the behavior of our control implementations. Even if the code hasn't changed, running the full suite of tests regularly (e.g. daily or weekly) will capture any behavioural changes made by the Cloud Service Provider, which would otherwise be opaque.

There are a couple of strategies to consider when setting up the CI pipeline – "Isolation" Testing and "Combination" Testing:

Isolation Testing

The layering of controls implemented around a specific service to make it fully compliant presents challenges in testing the full stack of controls around any particular service. For example, if both preventative and detective methods are implemented for the same control it is not possible to test the detective method if the preventative is blocking the creation of a non-compliant resource. IP whitelisting controls may outright block the creation of resources, regardless of any other configuration applied.

We need an environment to cleanly and continually test the individual implementations of each control in the library in an isolated manner, giving us confidence that each individual control is effective.

Combination Testing

As well as testing the individual control implementations, we also need to ensure that there are no conflicts when the full stack of controls across multiple services are combined. This is a non-trivial exercise to get right with preventative controls, because of the "blocking" challenges described above.

With PaaS services issues typically arise around network controls (e.g. user defined routes, virtual firewall rules), access management/roles and encryption and so isolating combinations of these controls in testing may also reveal issues that need to be addressed. Issues will usually manifest themselves in the denial of resource creation (or generation of alerts), which should otherwise be allowed.

Controls Delivery

This is the pipeline that delivers the "Apartment Unit" and "Fixtures and Fittings".

In our experience, there is rarely a "one size fits all" set of guardrails that satisfies the needs of all application teams. In reality, almost every application team requires a slightly different set of guardrails according to the services they are using, the network zone and data classifications. As a result, a significant percentage of tenants in the platform require customisation of the guardrails around their environment.

One way to manage custom requirements is to handle exceptions in the policy code itself and deploy the same policy set everywhere. This makes the process of delivering new "apartment" units relatively simple, but makes the testing of each policy and troubleshooting issues considerably more complex. As new requirements come in existing policies need to be evaluated and modified if new customisations are required.

The alternative is to deploy a custom set of much simpler policies using metadata to automatically create a custom package of deployment artifacts and Cucumber tests for the requirements of any particular "apartment". Attributes we can use for matching control artifacts to apartment units include :

- The target CSP
- Whitelisted services
- Network zone
- Data sensitivity
- Control exclusions – i.e. any controls which should be excluded from this specific apartment for reasons other than the above

Prior to giving the "apartment" a green light for application deployment we run through a set of Cucumber tests to ensure the combination of controls selected does not result in conflicts or other issues. Because of the blocking nature of certain preventative controls, it may not always be possible to run through the full suite of controls in Production - particularly where we have a combination of preventative and detective techniques in place for similar controls. We need to be shrewd in selecting the control tests that are executed for the purpose of validating our deployments to avoid false positives.

Reporting Framework

The purpose of these Cucumber tests is not to replace the compliance dashboards provided by the cloud service providers or third party tools. Our tests, however, should give us confidence that the policies being tracked meet the requirements – if the dashboard indicates everything is green, then we want to be confident that it isn't missing anything non-compliant.

The goal of the reporting for the testing framework is to generate evidence of which tests were run and when, which tests have passed or failed and link the tests back to the originating control objectives and regulatory requirements. Where the results of any tests have changed, we need to be alerted immediately and a response process put in place to mitigate changes in compliance posture.

In our example above we have tagged the BDD feature with the control objectives the feature is linked to. We write the summary results to a database, with a hyperlink to the full test output. This could also be integrated into existing compliance tools, such as RSA Archer or ServiceNow.

Compliance-as-Code for Application Teams

While guardrails can be put in place to put restrictions around the core capabilities of the cloud they can only go so far. Each PaaS service will have it's own mechanism for implementing controls, particularly where a secondary control plane is involved – for example, a managed Kubernetes service is going to be very different to a managed Database service. Even within the same Database service, the different flavours (PostgresDB, MySQL, MSSQL) have their own distinct control planes.

As PaaS services are adopted, financial services firms need to prepare for application teams to take on more responsibility for the controls in and around the services being used for their application platform (i.e. the "furnishings" in their "apartment"). By approaching cloud resource management as an extension of existing software quality control processes, by using the techniques described in this paper at different stages of SDLC, application teams should be well set to handle these additional responsibilities.

Cloud Resource Management Discipline

In the face of often aggressive deadlines for migrating applications to public cloud, application teams who maintain discipline in how they manage their compliance responsibilities around the cloud resources underpinning their application will be in a strong position to obtain the necessary sign-offs for go-live.

We recommend focusing on maintaining discipline around the following items. While these may be viewed as unnecessary activities that might slow a project down, often concerns raised around the handling of compliance requirements can derail projects resulting in firedrills and delays that could otherwise have been avoided.

Enforce Infrastructure as Code

Enforcing the use of Infrastructure as Code in any environment where non-public data is hosted ensures that best practices from software development can be used to control the provision of resources in the Cloud.

Restricting the ability to provision or modify resources in these environments reduces the ability for any individual to bypass pipeline controls, which could otherwise result in non-compliant resources.

Communicate control requirements using BDD

Using Behaviour Driven Design to communicate compliance requirements at the "furnishings" level gives control owners and auditors comfort that requirements are well communicated and understood using a common, easily understood set of semantics.

Practice Test Driven Development for Cloud Resources

The process of writing compliance tests up-front validates the completeness and effectiveness of the BDD control requirement features. Often the process of writing the tests results in changes to the BDD feature to make the requirements more robust.

It also forces teams to consider how compliance could be violated, resulting in a more secure and stable solution overall.

Incorporate controls testing in the “furniture” delivery pipeline

All of the control points mentioned in previous sections can and should be incorporated into the SDLC pipelines used by applications teams for deploying cloud resources. This includes Cucumber tests which generate a continual stream of evidence and ability to “catch” failed tests in continuous integration environments, immediately after deployment and periodically against existing deployments.

Cucumber Tests as Guardrails

In this paper we have presented guardrails as controls which can be implemented in- and out-of-band using various tools. Cucumber Tests should also be considered an essential tool for implementing guardrails in the delivery of an application team’s “furnishings”.

The teams involved in onboarding a new cloud service – often a combination of a public cloud specialist team and the first application teams using the service – should work together to define the BDD feature specifications and build the Cucumber tests. These can then be deployed in the application team’s pipeline and inside the “fixtures and fittings” as continuous tests.

Making these tests available in an “inner source” repository allows subsequent consumers of that service to leverage and extend this work, even if the tooling they want to use for provisioning is different.

Conclusion

Meeting compliance requirements across a heterogenous set of PaaS services from multiple cloud service providers is a major challenge for financial services firms. For IaaS the prevailing approach was to implement controls in a monolithic abstraction layer through which all requests were funneled. For PaaS adoption this approach is no longer viable, with developers needing access to the cloud service providers’ APIs. This necessitates a more federated approach towards how the cloud platform is built and maintained, which we modelled in this whitepaper using the analogy of an apartment block. Cloud-native services, 3rd party tools and open source frameworks are emerging to support this shifting approach. These tools take advantage of the information available through cloud providers’ APIs, putting in place preventative and detective “guardrails” to manage the risk of non-compliant resources being deployed by the application team. A typical “polycloud” environment will have several tools and CSP-native services deployed in an attempt to deploy guardrails and it is important that they are all configured to the same specifications.

Using software development best practices and mandating the use of infrastructure-as-code we can continually test the completeness and efficacy of controls implemented in the cloud platform. We can also automatically deploy the controls required around any application-specific environments, based on a consistent set of attributes, generating a continual stream of auditable evidence along the way.

Our favoured technique for defining and testing controls is Behaviour Driven Design (BDD), which uses a structured natural language to describe the behavior of the system under specific scenarios. This allows both technical and non-technical stakeholders to understand how the system should behave when the different controls are in place. “Cucumber” is a polyglot BDD test framework for writing behavioural tests, integrated into Continuous Integration and Continuous Deployment (CICD) pipelines to give us confidence in the effectiveness of controls across the platform.

Finally, creating a data model for end-to-end lineage - from the underlying requirements for our controls, to control objectives, to behavioural specifications and finally to the capture of implementation test results and the outputs of continuous compliance tools - will help in communicating to internal stakeholders and regulators how effectively the control requirements of the organisation are being met.

www.synechron.com