# *Exercise 2.5: Fitness Functions*

At PETech, while we are going to build the Engineering Platform on AWS, we know that the merger with Alltech is pending completion. AllTech is 100% on Google Cloud, and they don't even have an AWS account. We know that when we build the platform for PETech, we have to focus on our immediate customers but make architectural decisions that allow us to change and adapt the platform, such as potentially for other clouds in the future. One area of high importance is our custom Platform APIs.

How might we write a fitness function that ensures our platform APIs are implemented in a cloud-agnostic way?

1. Using the sample API provided in the (C3 Repo)[Todo, Link] - Write a fitness function that ensures our API keeps cloud-specific features and operations in isolated interfaces that don't affect our service logic
2. How might we expand this fitness function to work for all of our Platform APIs, not just this one?

After some debate amongst the team, we've decided that test-driven development is a rule we want to adopt and use across all of our platform's custom software.

1. Write an Architectural Decision Record that captures this decision. Include reasons, alternatives, and details.
2. Write a fitness function that will fail if someone checks in a new Service Layer without tests associated with it.

As we've seen throughout this chapter, Observability and Monitoring Data are at the core of every decision we make. Consider how we might write ADRs and Fitness Functions that capture this.

1. How might we write a fitness function that would fail if a new API Feature gets checked in without any monitors? Feel free to use a specific observability tool to write your answer and then compare it against the answer in the back for similarities.
2. Consider how a data-driven approach might change the dynamics of the team's interactions with other stakeholders and executives at the company. What tactics can you use to debate the merits of a new feature request using our ADRs, Fitness Functions, and observation-driven decision-making? Consider how these techniques remove emotions and assumptions from these sorts of debates.

## Solution

PE Tech is building an Engineering Platform on AWS. However, with the pending merger with AllTech, which operates entirely on Google Cloud, the platform must be adaptable to different cloud environments. This necessitates designing cloud-agnostic APIs that can work across various cloud providers while also adhering to test-driven development (TDD) principles for reliability and quality. Observability and monitoring are crucial for ensuring platform stability and for guiding decisions based on data.

**Decision:**

1. **Cloud-Agnostic APIs:**
    - Implement platform APIs to isolate cloud-specific features within dedicated interfaces, keeping the main service logic cloud-agnostic.
    - Write fitness functions that ensure APIs follow this pattern.
2. **Test-Driven Development:**
    - Adopt TDD for all platform custom software.
    - Write fitness functions that enforce tests for new service layers.
3. **Monitoring and Observability:**
    - Ensure each new API feature includes appropriate monitoring and observability tools.
    - Write fitness functions that enforce this requirement.

**Alternatives Considered:**

- **Cloud-Specific APIs:** Building APIs specific to AWS would simplify implementation but reduce flexibility.
- **No TDD Enforcement:** Relying on manual code reviews for quality assurance, but risking missed test coverage.
- **Ad-hoc Monitoring:** Allowing teams to decide monitoring independently, risking inconsistencies.

**Justification:**

- **Future-Proofing:** Cloud-agnostic APIs ensure easier migration and integration with different cloud providers.
- **Reliability:** TDD ensures code reliability and catches defects early, improving software quality.
- **Proactive Monitoring:** Consistent observability ensures platform stability and allows data-driven decision-making.

**Fitness Functions:**

**Cloud-Agnostic APIs:**

- Ensure platform APIs are implemented in a cloud-agnostic way by isolating cloud-specific features:

```
layeredArchitecture()
  .layer("CloudSpecific").definedBy("..cloudspecific..")
  .layer("Service").definedBy("..service..")

  .assertLayer("Service").mayOnlyBeAccessedByLayer("Handler")
  .assertLayer("CloudSpecific").mayOnlyBeAccessedByLayer("Service")
```

**Test-Driven Development:**

- Fail if a new service layer is checked in without associated tests:

```
serviceLayerWithoutTests := // Logic to find services without tests
assert serviceLayerWithoutTests.empty()
```

**Monitoring and Observability:**

- Fail if a new API feature lacks monitoring:

```
newAPIFeaturesWithoutMonitors := // Logic to find features without monitors
assert newAPIFeaturesWithoutMonitors.empty()
```