

# Centralising Test Data Generation to Reduce False Positives in Critical Government Document Systems

---

Matthew Alessandro  
Industrial Partner: DVLA

Programme: BSc Applied Computing  
Module: Industrial Based Research

# Driver and Vehicle Licensing Agency

- The DVLA currently manages ~40 million vehicles and ~50 million driver records in Great Britain
- Caseworkers process applications, medical reviews, and enforcement actions daily
- A routine part of casework is sending letters to drivers (licensing decisions, medical outcomes, legal obligations)
- Drivers Medical Group alone sends approximately 200,000 letters per month across 1,000+ letter types
- Getting these letters wrong can have serious consequences!



## Driver & Vehicle Licensing Agency

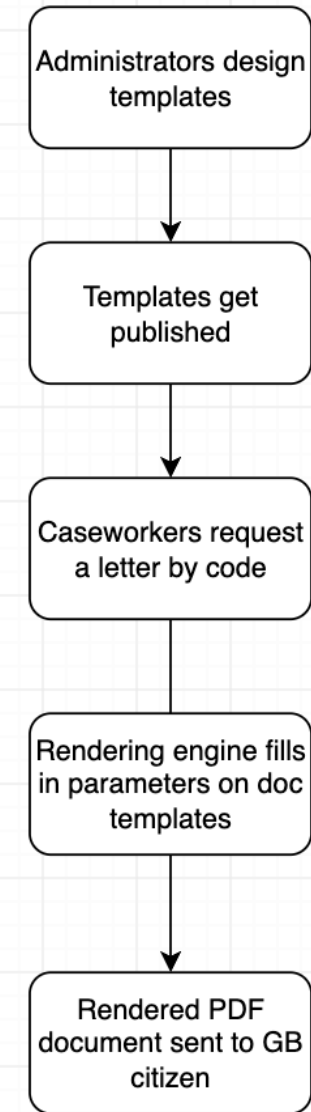


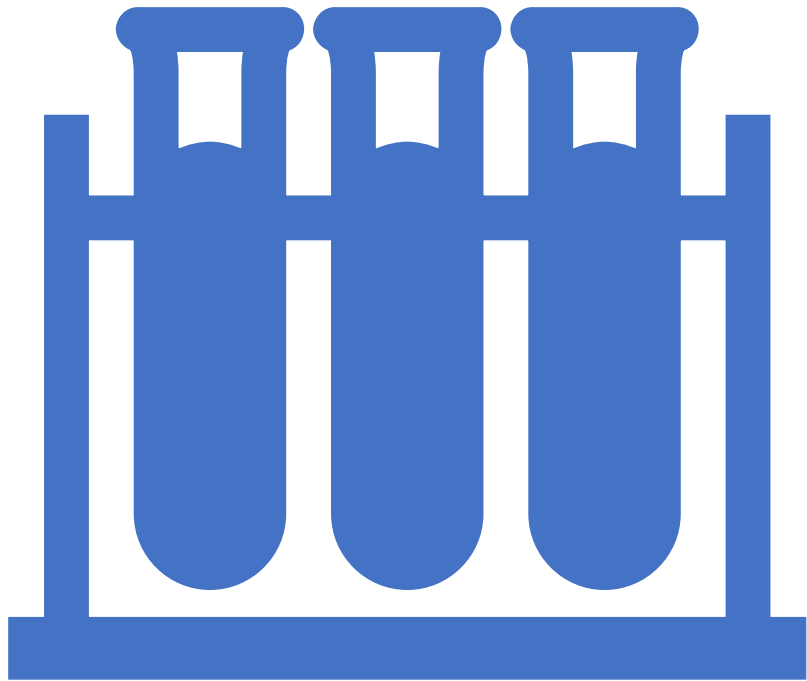
# What is the Document Generation Utility (DGU)?

DVLA's cloud-native system for generating casework letters and documents at scale. It replaced the legacy approach of using Microsoft Word macros on a shared file server (CASP).

3 main components:

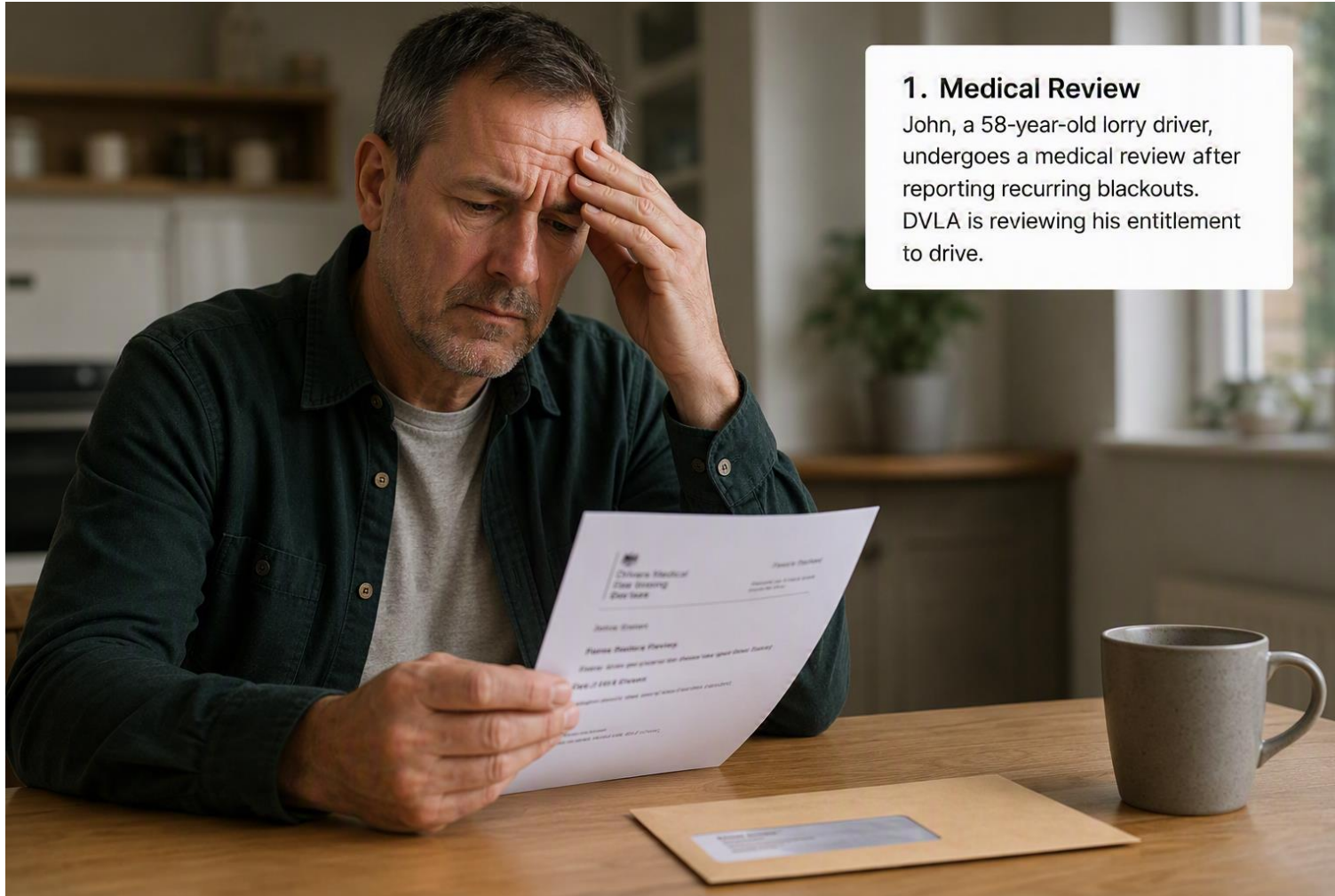
1. **Template Admin UI** – where caseworkers create, manage and publish letters.
2. **Document Generation Admin API** – the backend that stores template metadata, parameters, versions, and file references.
3. **Document Generation Consumer API** – the endpoint that other DVLA systems (Dynamics) call to render a letter.





Why is it important we  
test DGU?

# Scenario: Incorrectly Declared Fit to Drive



*Note: this scenario is entirely fictional but shares the impact of what could go wrong if a lack of testing/poor testing was in place for a letters system.*

## 2. Incorrect Letter Sent

Due to an error in the letter generation process, John receives a letter stating he is fit to drive with no restrictions.



Driver & Vehicle  
Licensing  
Agency

Date: 12 April 2024

Driver Licence Number:  
JOHN SMITH  
AB12 3CD45 6EF

Mr John Smith  
123 Main Road  
Anytown  
AA1 1AA

Dear Mr Smith

### Your medical review

We are pleased to advise that you have been assessed as

**FIT TO DRIVE.**

You may continue to drive with no restrictions.

If your health changes, you must tell us as it may affect your ability to drive.

Yours sincerely

Driver & Vehicle Licensing Agency

Driver & Vehicle  
Licensing  
Agency

### **3. John gets back on the road**

After being incorrectly approved to drive his HGV, John returns to work





#### **4. The accident**

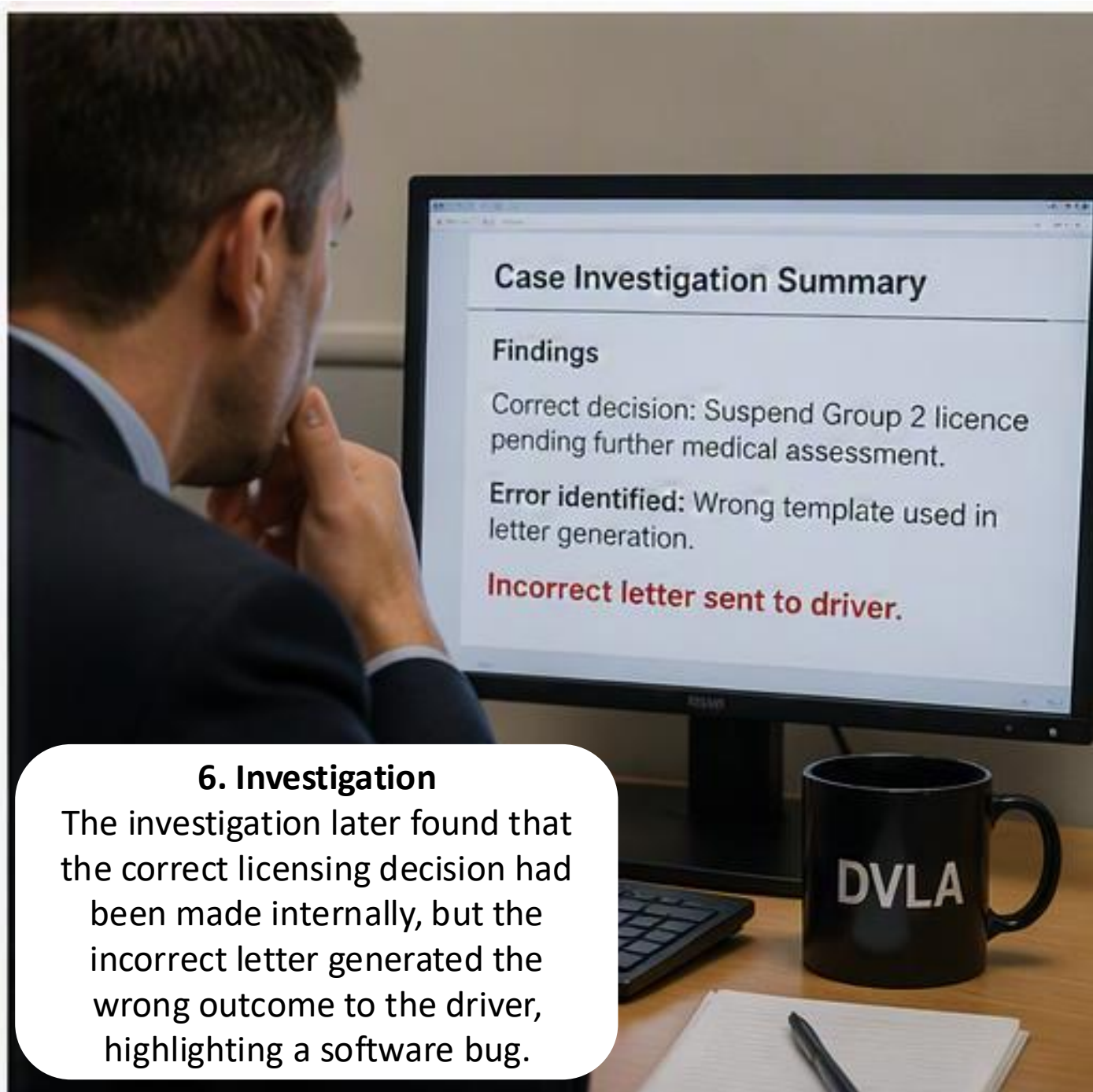
Two weeks later, John suffered a blackout while driving his lorry on a motorway, causing a multi-vehicle collision.

**M6 CLOSED  
JCT 15-17  
EXPECT DELAYS**

### **5. Impact**

Several people were injured, major road closures occurred, and significant financial costs were incurred.



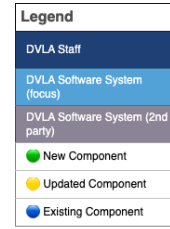


## 6. Investigation

The investigation later found that the correct licensing decision had been made internally, but the incorrect letter generated the wrong outcome to the driver, highlighting a software bug.

Had robust tests existed, this issue would likely have been caught prior to releasing the application

# TESTING DGU

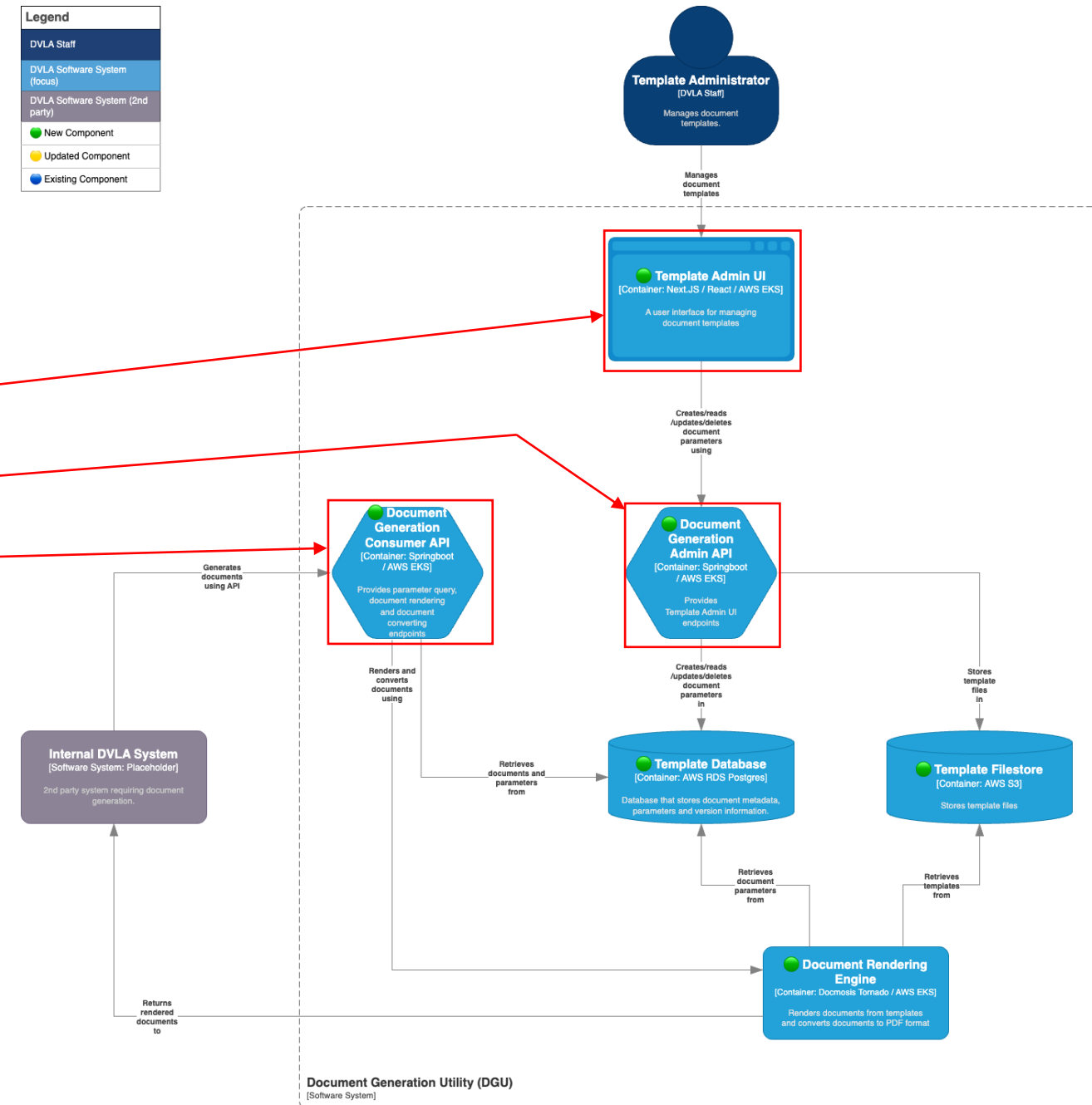


## 3 Individual services to test

- Casework internal UI
- Admin API (43 Endpoints)
- Consumer API (8 Endpoints)

Each service has different test cases but uses the SAME data models

A single realistic test scenario requires 10+ API calls in the right order, threading IDs between them to create an entity relationship flow that represents the desired object for testing.



# Problem Summary

Each test pack defines its own interpretation of valid data, creating inconsistency across the test suite.

Maintaining three separate test packs increases the risk of divergence as developers make independent changes over time.

When API contracts change, updating a single, unified test pack is straightforward. Maintaining three separate implementations makes even minor contract alignments more complex and time-consuming.

## Result:

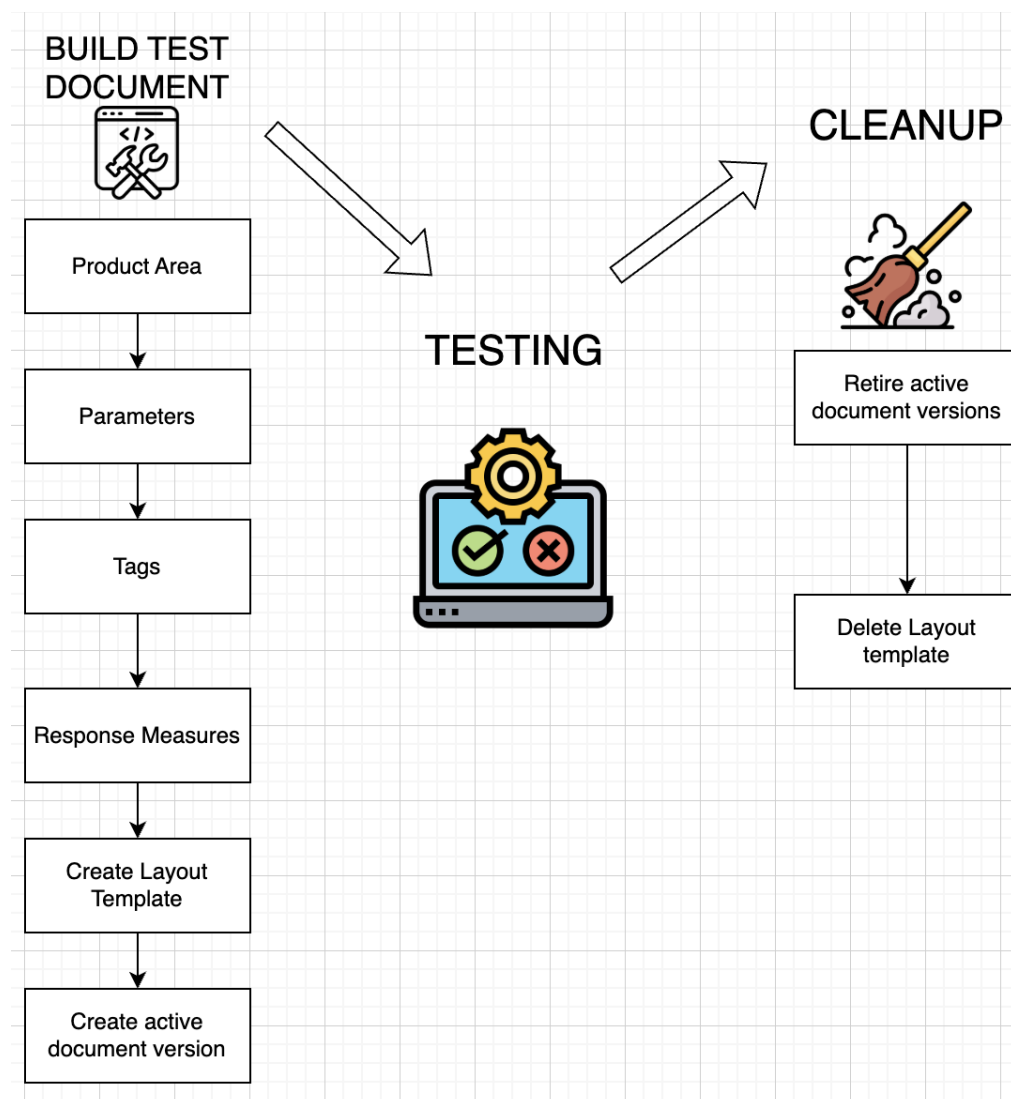
- Test data Inconsistency
- Missing proper relationships (documents without product areas)
- Unrealistic values (hardcoded data that never exercises real code paths)

## Consequences:

**False positives** - tests pass when they should fail

**Wrong letters sent out the door** - incorrect letters being sent out due to weak test quality/coverage

## Example 'test setup' dependency chain for a layout template document



# The recipe book analogy (Why consistency is important)



# What does the research say?

## Test effectiveness depends on test data quality

- Jia & Harman [2011]: Mutation testing measures whether tests detect faults, but only if test data exercises realistic conditions
- Zhu et al. [1997]: "The choice of test data is crucial to the effectiveness of testing"

## Bad test data gives false confidence

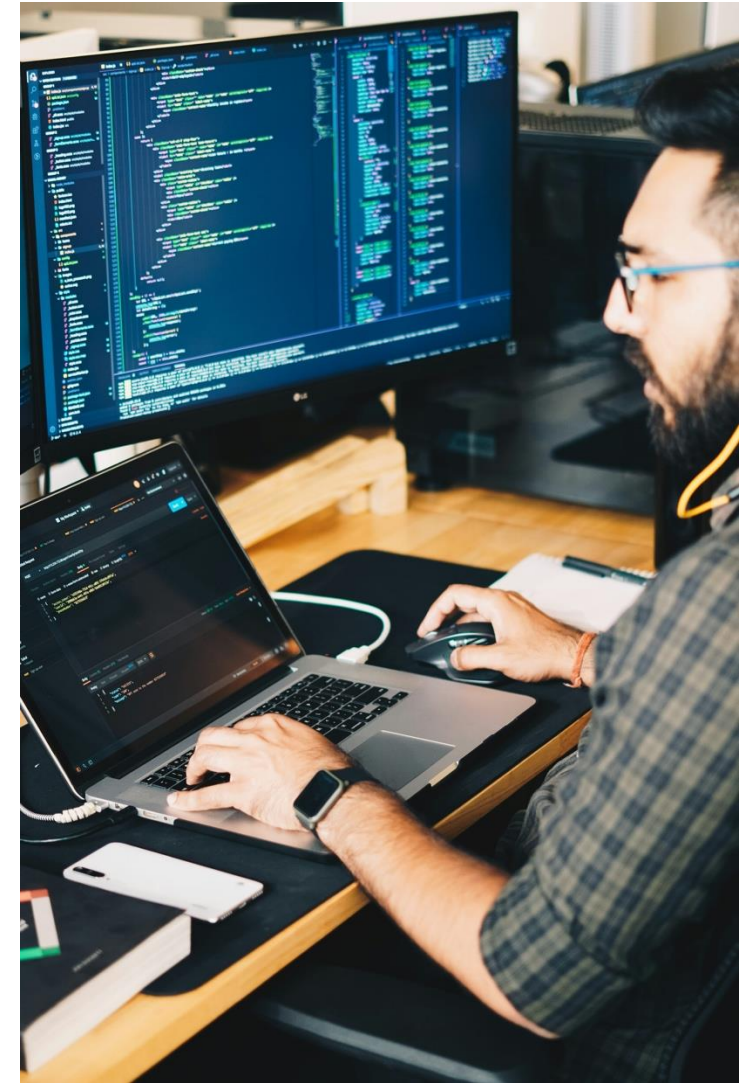
- Spadini et al. [2018]: "Test smells" like mystery guest (disconnected data) correlate with post-release defects
- Newman [2021]: In microservices, when test data diverges between services sharing a domain model, integration defects go undetected

## Existing tools don't solve the full problem

- Meszaros [2007], FactoryBot, Faker Maker: Address single-entity generation but don't enforce cross-entity constraints or hierarchical relationships

## The Builder pattern is the right fit

- Gamma et al. [1994]: Separates complex object construction from representation
- Freeman & Pryce [2009]: "Make it trivial to create valid test data and difficult to create invalid data"
- Luo et al. [2014]: Random/non-deterministic test inputs cause flakiness
- The gap: No empirical evidence linking test data inconsistency across distributed test packs to false positive rates in production-critical systems



# Aim and Objectives

Aim: Investigate impact of inconsistent test data and evaluate a centralised gem as intervention

## Objectives:

- Critically analyse existing test packs, producing an inconsistency taxonomy
- Quantify the false positive rate from invalid test data
- Design and develop a shared Ruby gem (Builder pattern + Faker Maker)
- Evaluate via mutation score improvement (target: 5%+)



# Methodology

How I will approach this:

**Philosophy:** Positivist (objective, measurable data)

**Approach:** Deductive (hypothesis first, then test it)

**Time Horizon:** Cross-sectional (before vs after snapshot)

**Strategies:** Literature review + Software development + Experiment

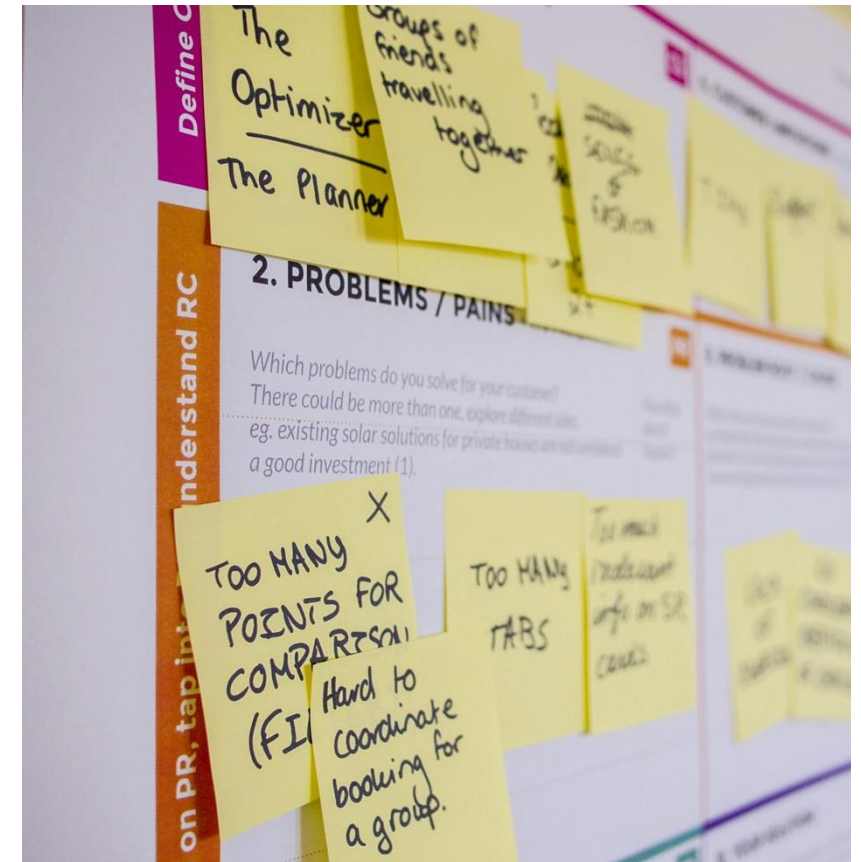
## **Four phases:**

1. Baseline audit - classify inconsistencies and document API examples with Bruno client
2. Impact measurement - change the validation rules for example, (all related tests should fail but it would be interesting to see how many still pass)
3. Implementation - build the gem (FDD testing referring to API schema)
4. Evaluation - re-measure, compare before/after (Did the gem help reduce issues?)

## **Variables:**

Independent Variable: Test data generation approach (distributed vs centralised)

Dependent Variable: Mutation score, false positive rate, constraint coverage



# Proposed solution – Builder Pattern

---

## Why Builder pattern?

- Step-by-step construction of complex objects
- Fluent chainable interface with human readability
- Enforces domain constraints at build time
- Different representations from same construction process
- Papermill acts as the "Shared recipe book"

### Before

```
product_area = {name: "Test Product Area Name",
  description: "This is a fake description",
  owner: "DVLA"
  updatedBy: "Joe Bloggs"}

response = HTTParty.post(url: Settings.test.url, body: product_area.as_json, headers: Settings.test.headers)

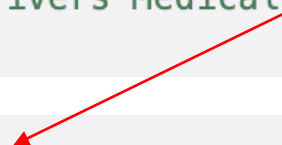
parsed_response = response.simple_symbolize

# Now ready to test.
```

### After

```
DVLA::Papermill.generate(:product_area)
  .with(name: 'Drivers Medical')
  .build
```

```
=> {
  name: "Drivers Medical",
  description: "Voluptatem nulla possimus vero ad et omnis tempora odio modi.",
  owner: "DL SQUAD",
  updatedBy: "Test user"
}
```



# Project Planning and Resources

## Timeline: 26 weeks across 8 phases

- Literature Review (W1-3) → Baseline Analysis (W4-6) → Impact Assessment (W7-9)
- Intervention Design (W10-11) → Implementation (W12-16) → Evaluation (W17-19)
- Analysis & Writing (W20-23) → Review & Submission (W24-26)
- Milestones: Interim Viva (W12), Report (W24), Final Viva (W26)

## MoSCoW Prioritisation

- **Must have:** Baseline audit, gem for 4 core entities, before/after measurement
- **Should have:** Full 9 entity builders, property-based tests
- **Could have:** All three test packs integrated, developer feedback
- **Won't have:** Public release, UI-level generation

## Resources

- **Hardware:** Existing DVLA macOS workstation + Docker Desktop
- **Software:** Ruby 4, Cucumber 11, Faker Maker, RSpec, Mutant (all open source or existing)
- **People:** Academic supervisor, Principal QE (Nigel Brookes-Thomas), AC Squad colleagues
- **Budget:** £0
- All tools are open source or already available under MIT license within DVLA infrastructure
- No external licences, hosting, or procurement required



# Impact and Limitations

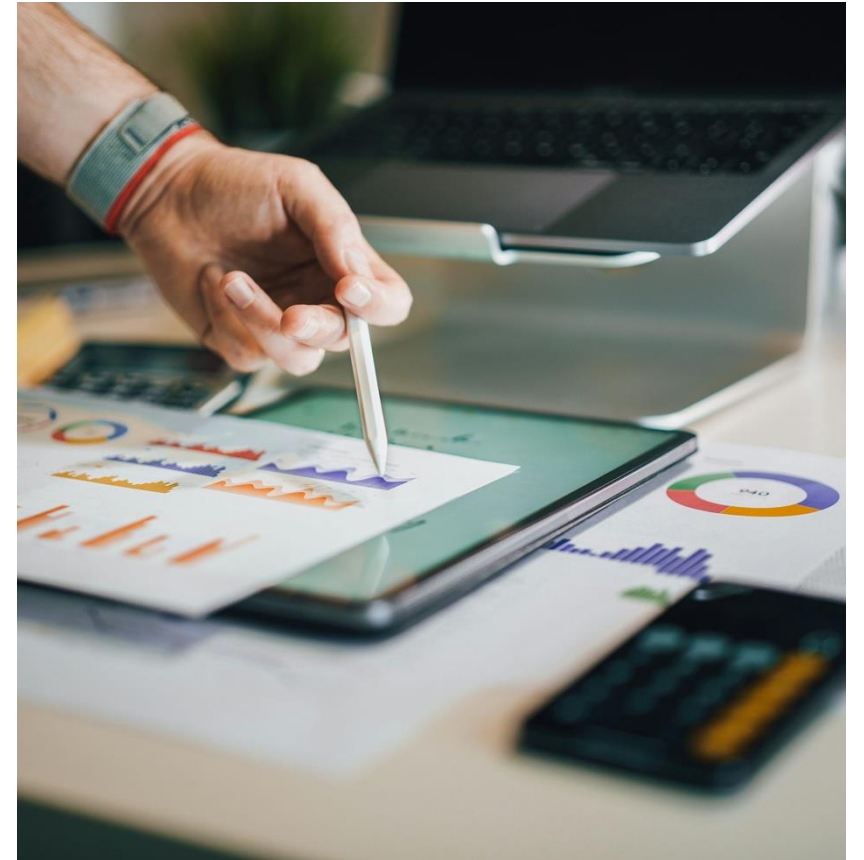
---

## Impact:

- Eliminates duplicated logic across 3 packs
- Prevents false positives through domain constraint enforcement
- Reduces onboarding time for new engineers
- Reusable pattern for other DVLA multi-service systems

## Limitations:

- Single system/organisation
- Evaluation on one test pack





# Summary and Questions

---

## Bringing it all together

- *Problem:* Inconsistent test data → false positives → defects reach production
- *Gap:* No empirical evidence linking test data inconsistency to false positive rates
- *Solution:* Centralised Ruby gem using Builder pattern
- *Evaluation:* Measurable improvement via mutation testing
- *Impact:* Better defect detection for a system serving millions

Questions?