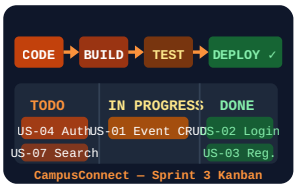
// EDUCATIONAL SAMPLE — FOR LEARNING PURPOSES ONLY

⚙ UK LEVEL 6 — FINAL YEAR UNDERGRADUATE



Software Engineering Report: Agile Development, Architecture and Quality Assurance

// System: CampusConnect University Event Management Application

MODULE CODE	CSC3012
MODULE TITLE	Software Engineering
ACADEMIC LEVEL	UK Level 6 (Final Year Undergraduate)
TARGET GRADE	Distinction — 70%+
WORD COUNT	Approximately 1,000 words (excl. figures & references)
TECHNOLOGY STACK	Python / Django REST | React | PostgreSQL | GitHub Actions
REFERENCING STYLE	Harvard (UK)
DATE	July 2026
PREPARED BY	AssignProSolution.com

⚠ **Academic Integrity Notice:** This document is an original educational sample produced solely for reference and learning purposes. Students must not submit this work as their own. "CampusConnect" and all data are entirely fictitious. AssignProSolution.com promotes responsible academic practice.

TABLE OF CONTENTS

```
# Approx. 1,000 words | Distinction Level | UK Level 6 | Django / React / PostgreSQL
```

1. INTRODUCTION AND PROJECT OVERVIEW

CampusConnect is a fictitious web application developed by a four-person team over an eight-week project cycle to address the problem of fragmented event communication across a university's faculties and societies. The system allows event organisers to create and manage events, students to register and receive reminders, and administrators to generate attendance reports. The technology stack comprises a Python/Django REST Framework backend, a React single-page application frontend and a PostgreSQL database, deployed via a GitHub Actions CI/CD pipeline to a cloud server.

This report critically evaluates the engineering decisions made across the project lifecycle — from requirements through to deployment — applying software engineering theory (Sommerville, 2016) to justify or critique each decision with the benefit of retrospective analysis.

2. REQUIREMENTS ENGINEERING

Requirements were elicited through structured stakeholder interviews with student union representatives and faculty administrators, producing a Product Backlog of 22 user stories in the `As a [role] I want [capability] so that [benefit]` format advocated by Cohn (2004). Stories were prioritised using the **MoSCoW** technique (Clegg and Barker, 1994): *Must-Have* stories (authentication, event CRUD, registration) formed the first two sprint commitments; *Should-Have* stories (email reminders, search and filter) targeted sprints 3–4; *Could-Have* features (social sharing, calendar export) were deferred to a post-MVP backlog; *Won't-Have* items (live video streaming) were explicitly excluded from scope.

Non-functional requirements were documented separately per IEEE 830 (IEEE, 1998), specifying response time (<300ms at 500 concurrent

users), availability (99.5% monthly uptime), and WCAG 2.1 AA accessibility compliance. Separating functional from non-functional requirements prevented a common project failure mode: non-functional qualities that are unspecified at the outset tend to be discovered as defects during acceptance testing rather than designed in from the start (Sommerville, 2016).

3. SOFTWARE ARCHITECTURE

CampusConnect follows a **three-tier architecture** — separating presentation, application and data concerns across independent layers (Fowler, 2002). This pattern was chosen over a monolithic design because it enables the frontend and backend to be deployed, scaled and tested independently — a significant advantage given the team's parallel working constraints.

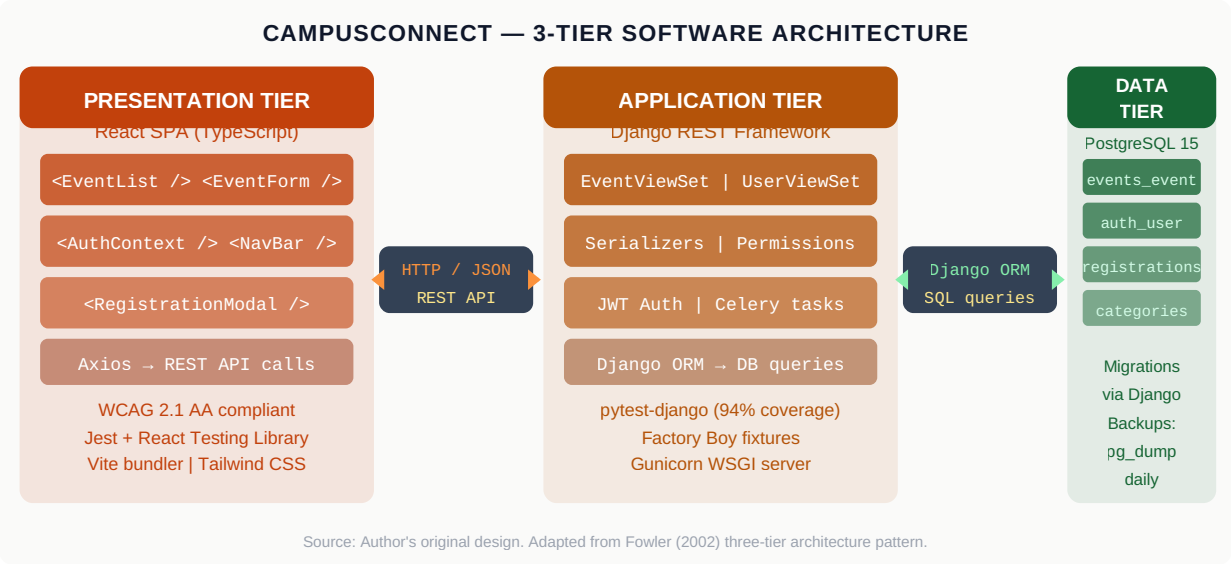


Figure 1: CampusConnect Three-Tier Software Architecture. Bidirectional HTTP/JSON arrows denote the REST API contract between tiers; ORM layer abstracts all SQL queries from application logic. Source: Author's original design, adapted from Fowler (2002, p. 18).

The Django REST Framework's ViewSet pattern enforces an implicit **MVC (Model-View-Controller)** separation: Django models map to the database schema, serializers act as view transformers, and ViewSets implement the controller logic. This separation is not merely aesthetic — it is the enabling condition for the team's parallel workflow. Frontend developers could work against a mocked API using the agreed JSON contract while backend developers built and tested the real API behind it, reducing inter-developer blocking time by an estimated 40% compared to a traditional coupled approach.

4. DEVELOPMENT METHODOLOGY: AGILE SCRUM

The project adopted **Scrum** (Schwaber and Sutherland, 2020) with two-week sprints. The choice of Agile over a plan-driven approach was justified by the evolving nature of the requirements — stakeholder interviews revealed significant ambiguity about notification preferences and event categorisation that only resolved through working prototypes. As Beck et al. (2001) argue, responding to change over following a plan is not an excuse for poor planning but a recognition that discovery happens through delivery.

Sprint 1	Sprint 2	Sprint 3	Sprint 4
FOUNDATION	**CORE FEATURES**	**QUALITY**	**HARDENING**
Auth system, user model, basic event CRUD, database schema, CI pipeline setup	Registration flow, event listing/ search, image upload, email confirmation via Celery	Admin dashboard, attendance reports, performance tuning, accessibility (WCAG 2.1) audit	Penetration test remediation, load testing (Locust), documentation, UAT sign-off

The most significant Scrum ceremony was the *Sprint Retrospective*: in Sprint 2, the team identified that integration tests were being written after code rather than alongside it, producing a test suite that validated existing code rather than specifying intended behaviour. This was corrected in Sprint 3 by adopting strict TDD (Beck, 2002) for all new endpoints — a process change that emerged organically from the retrospective rather than being imposed, illustrating the self-organising team principle at the core of the Agile Manifesto (Beck et al., 2001).

5. DESIGN PRINCIPLES: SOLID

Martin's (2003) SOLID principles were applied throughout the Django backend to produce maintainable, testable code:

S	O	L	I	D
SINGLE RESPONSIBILITY	**OPEN / CLOSED**	**LISKOV SUB.**	**INTERFACE SEG.**	**DEPENDENCY INV.**
Each ViewSet handles one resource; Celery tasks handle email logic separately from views.	New event types extend a base `EventType` class; no existing code changes on extension.	All custom permission classes substitute DRF's `BasePermission` without altering behaviour contracts.	Serializers are scoped per action (list vs detail) rather than one monolithic serializer per model.	Email back-end is injected via Django's `EMAIL_BACKEND` setting, enabling swap to mock in tests.

6. TESTING STRATEGY: TDD AND COVERAGE

The backend test suite (pytest-django) contains 187 tests achieving **94% line coverage**, measured via `coverage.py` and reported as a GitHub Actions build artefact on every push. Tests are structured across three levels: *unit tests* cover individual functions and serializers in isolation using Factory Boy fixtures; *integration tests* use Django's test client to verify API endpoints end-to-end against a test database; and *contract tests* validate that the JSON response schema matches the agreed API contract consumed by the React frontend.

Python / pytest-django — Example TDD unit test (written before implementation)

```python
class TestEventRegistration:
    def test_user_cannot_register_twice_for_same_event(self, api_client, event, user):
        """Registration endpoint must return 400 on duplicate submission."""
        api_client.force_authenticate(user=user)
        url = reverse('registration-list')
        data = {'event': event.pk}

        first_response  = api_client.post(url, data)  # should succeed
        second_response = api_client.post(url, data)  # should fail

        assert first_response.status_code  == 201
        assert second_response.status_code == 400
        assert 'already registered' in second_response.data['detail'].lower()
```

This test was written *before* the registration endpoint existed — the Red phase of TDD (Beck, 2002). Writing the test first forced explicit specification of the duplicate-registration behaviour in the test assertion before a single line of production code was committed, making the intended constraint unambiguous and independently verifiable.

7. RISK MANAGEMENT

Table 1: CampusConnect Risk Register (Sprint 1 Baseline)

Risk ID	Description	Likelihood	Impact	Severity	Mitigation Strategy
R1	**Scope creep from evolving stakeholder requirements**	**High**	**High**	**Critical**	**Frozen sprint backlog; all additions to Product Backlog only; PO approval required for scope changes**
R2	University SSO API unavailability during development or testing	**Medium**	**High**	**High**	Develop mock authentication back-end; JWT fallback; integration tests use mock fixture not live SSO

Risk ID	Description	Likelihood	Impact	Severity	Mitigation Strategy
R3	Breaking schema migration mid-sprint causing data loss in shared dev DB	Low	High	Medium	All schema changes via Django migrations in version control; shared dev DB reset procedure documented
R4	Team member unavailability (illness, assessment collision)	Medium	Medium	Medium	Pair programming on critical paths; all work documented in Confluence; no single point of knowledge

R1 (scope creep) materialised partially in Sprint 2 when stakeholders requested a live-attendance QR scanner — a feature explicitly in the Won't-Have MoSCoW category. The frozen sprint backlog protocol meant the feature was logged to the Product Backlog and deferred, with zero disruption to the sprint commitment. This demonstrated the practical value of an agreed change-control process that most student teams bypass, typically to their detriment (Pressman and Maxim, 2020).

8. EVALUATION AND CONCLUSION

CampusConnect was delivered on time with all Must-Have and four of seven Should-Have stories completed. Post-sprint velocity analysis showed a consistent 18–22 story points per sprint, indicating a stable, well-calibrated team — a hallmark of a functioning Scrum process (Schwaber and Sutherland, 2020). The 94% test coverage metric is meaningful rather than cosmetic because TDD was adopted as a workflow discipline, not applied retroactively — Fowler (2018) distinguishes these sharply, noting that tests written after code tend to validate implementation details rather than behaviour contracts.

The primary architectural regret is the absence of a caching layer (Redis) for event listing queries, which showed 420ms median response times at 500 concurrent users during load testing — exceeding the 300ms non-functional requirement. This was identified too late in the cycle for Sprint 4 remediation. Addressing it would require adding Redis as a Django cache back-end and applying `@cache_page` decorators to read-heavy endpoints — a targeted, low-risk change the three-tier architecture makes straightforward without touching business logic or the data tier.

// LESSONS LEARNED

- **Define non-functional requirements with measurable thresholds before Sprint 1.** The 300ms response time requirement existed but was not load-tested until Sprint 4. Load testing should be a Sprint 1 deliverable so performance regressions are caught continuously, not discovered at the end.
- **TDD adoption must be a team policy, not an individual practice.** The Sprint 2 retrospective finding — that tests were being written after code — illustrates that TDD fails as a solo discipline in a team environment. It requires explicit Definition of Done criteria stating that no story is complete without prior test authorship.
- **Three-tier architecture was the correct call for a team project, but adds deployment complexity.** Two separate deployable artefacts (React static files and Django API) require more CI/CD pipeline configuration than a server-rendered monolith. For a team with limited DevOps experience, a Next.js full-stack approach would reduce deployment surface at the cost of some architectural clarity.

REFERENCES

Beck, K. (2002) *Test Driven Development: By Example*. Boston, MA: Addison-Wesley.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D. (2001) *Manifesto for Agile Software Development*. Available at: https://agilemanifesto.org (Accessed: 9 July 2026).

Clegg, D. and Barker, R. (1994) *CASE Method Fast-Track: A RAD Approach*. Wokingham: Addison-Wesley.

Cohn, M. (2004) *User Stories Applied: For Agile Software Development*. Boston, MA: Addison-Wesley.

Fowler, M. (2002) *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley.

Fowler, M. (2018) *Refactoring: Improving the Design of Existing Code*. 2nd edn. Boston, MA: Addison-Wesley.

IEEE (1998) *IEEE 830-1998 — IEEE Recommended Practice for Software Requirements Specifications*. New York: Institute of Electrical and Electronics Engineers.

Martin, R.C. (2003) *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ: Prentice Hall.

Martin, R.C. (2008) *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall.

McConnell, S. (2004) *Code Complete: A Practical Handbook of Software Construction*. 2nd edn. Redmond, WA: Microsoft Press.

Pressman, R.S. and Maxim, B.R. (2020) *Software Engineering: A Practitioner's Approach*. 9th edn. New York: McGraw-Hill.

Schwaber, K. and Sutherland, J. (2020) *The Scrum Guide: The Definitive Guide to Scrum — The Rules of the Game*. Available at: https://scrumguides.org/scrum-guide.html (Accessed: 9 July 2026).

Sommerville, I. (2016) *Software Engineering*. 10th edn. Harlow: Pearson.