Learning Outcome 4: Evaluate Limitations Using Statistical Methods

4.1 Gaps in Testing Process

Despite comprehensive testing, several gaps were identified where additional testing would provide greater confidence:

Gap 1: Algorithm Path Coverage (A Pathfinding)*
Description: While the A* pathfinding algorithm is tested for typical scenarios (path found, no-fly zones avoided, optimal route), certain internal decision paths remain unexplored.
Specific Gaps:

- Early termination conditions (when open list becomes empty without finding goal)
- Heuristic function edge cases (when heuristic equals actual distance)
- Tie-breaking in priority queue (multiple nodes with equal f-cost)
- Very large graphs (1000+ nodes, complex constraints)

Current Coverage: PathfindingServiceTest includes 11 tests covering 40% of algorithm branches
Impact: Undiscovered bugs in rare edge cases could surface during deployment with unusual input distributions
Mitigation Potential: 3-4 additional targeted tests would likely increase branch coverage from 40% to 65%+

Gap 2: External Service Resilience
Description: DroneAvailabilityService makes calls to external drone databases. Testing assumes these services always respond correctly.
Specific Gaps:

- Service timeout (slow response, not error)
- Partial response (incomplete data set)
- Service degradation (returns stale data)
- Network failures between components
- Rate limiting (429 Too Many Requests)

Current Coverage: Mocked dependencies return successfully; failure modes not tested
Impact: Production deployment could fail when external services degrade
Evidence Location: DroneAvailabilityServiceTest uses MockedDroneRepository – doesn't test actual network calls
Mitigation Potential: WireMock library could simulate external service failures; 5-6 new tests would cover resilience

Gap 3: Multi-Drone Coordination Logic
Description: DeliveryPathService handles complex multi-drone scenarios. Not all combinations of constraints are tested.

Specific Gaps:

- All drones overbooked (no feasible solution possible)
- Conflicting drone capabilities for delivery requirements
- Three+ drones with overlapping availability
- Recalculation when preferred drone becomes unavailable mid-route
- Drone payload reduction during multi-stop routes

Current Coverage: 12 service tests cover typical multi-drone scenarios
Impact: Edge cases in production could result in suboptimal or infeasible route assignments
Evidence Location: DeliveryPathServiceTest – currently tests most common scenarios
Mitigation Potential: Parameterized tests with 10+ constraint combinations would increase confidence significantly

Gap 4: Controller Layer Branch Coverage
Description: Controller classes reach only 51% instruction and 42% branch coverage. This is lower than services (68% instruction).
Specific Gaps:

- Error condition branches
- Null checking in request validation
- Response formatting edge cases
- HTTP status code conditions

Current Coverage: ApiControllerMockitoTest includes 25 tests with mocked services
Impact: Untested error handling paths could produce incorrect HTTP responses
Evidence Location: Figure in Evidence Document shows controller layer at 51% instruction coverage
Mitigation Potential: 10-15 additional error condition tests would likely reach 70%+ branch coverage in controllers

Gap 5: Load and Concurrent Request Handling
Description: No tests for system behavior under load. Sequential request handling verified; parallel requests untested.
Specific Gaps:

- 100+ concurrent requests to pathfinding endpoint
- Memory usage under load (memory leaks possible)
- Thread safety in shared services
- Race conditions in multi-access scenarios
- Request queuing and timeout behavior

Current Coverage: Tests run sequentially; no load testing framework integrated
Impact: System could fail or degrade unexpectedly when handling production traffic
Evidence Location: Maven test execution is sequential; no stress testing tools in pom.xml

Mitigation Potential: JMeter or Gatling integration would allow load profile testing; 5-10 load scenarios

4.2 Target Coverage/Performance

Based on identified gaps, target levels were established for future testing enhancements:

Coverage Targets by Layer

Layer	Current	Target	Gap	Priority
Services (algorithms)	68% instr, 48% branch	80% instr, 70% branch	+12 instr, +22 branch	HIGH
Controllers (endpoints)	51% instr, 42% branch	75% instr, 65% branch	+24 instr, +23 branch	HIGH
Models (data objects)	71% instr, 62% branch	85% instr, 75% branch	+14 instr, +13 branch	MEDIUM
Overall System	60% instr, 38% branch	80% instr, 70% branch	+20 instr, +32 branch	HIGH

Justification for Targets:

- **80% Instruction Coverage:** Industry best practice for critical systems; every major code path executed
- **70% Branch Coverage:** All decision paths evaluated; catches condition-related bugs
- **80% for Services:** Algorithm correctness critical; high coverage reduces deployment risk

- **75% for Controllers:** API responses must be reliable; high coverage ensures correct error handling

Performance and Execution Targets

Metric	**Current**	**Target**	**Rationale**
Unit Test Execution	~4 seconds	<5 seconds	Rapid feedback for developer iteration
Integration Test Execution	~2.7 seconds	<10 seconds	Allows frequent full suite runs
Total Suite Execution	~6.7 seconds	<15 seconds	CI/CD pipeline can run on every commit
Avg Per-Test Time	47 ms	<50 ms	Consistent performance
Test Startup Time	~1 second	<2 seconds	Spring context loading

Load Testing Targets

Scenario	**Target**	**Justification**
Concurrent Requests	Support 50+ concurrent	Typical web service load

Response Time (p95)	<200 ms	User-acceptable latency
Memory Usage	<500 MB under load	Server resource constraints
Thread Safety	Zero race conditions	Data integrity requirement
Failure Recovery	Auto-recover within 30s	Resilience expectation

Defect Metrics Targets

Metric	**Target**
Defect Density	<1 defect per 1000 lines of code
Test Yield	>95% of defects caught during testing
Bug Escape Rate	<5% of bugs reaching production
Mean Time to Fix	<2 hours for P1 bugs

4.3 Comparing Testing to Targets

Coverage Comparison: Actual vs. Target

Layer	Actual	Target	Status	Gap
Services	68% instr, 48% branch	80% instr, 70% branch	Below	-12%, -22%
Controllers	51% instr, 42% branch	75% instr, 65% branch	Below	-24%, -23%
Models	71% instr, 62% branch	85% instr, 75% branch	Below	-14%, -13%
Overall	60% instr, 38% branch	80% instr, 70% branch	Below	-20%, -32%

Analysis: Coverage falls short of targets, particularly in branch coverage (38% vs. 70% target). Service layer performs best at 68% instruction; controller layer weakest at 51% instruction. The 20-32 percentage point gaps indicate substantial opportunities for improvement through additional test development.

Performance Comparison: Actual vs. Target

Metric	Actual	Target	Status
Unit Test Time	~4 seconds	<5 seconds	Met
Integration Test Time	~2.7 seconds	<10 seconds	Met

Total Suite Time	~6.7 seconds	<15 seconds	Met
Avg Per-Test	47 ms	<50 ms	✅ Met

Analysis: All execution time targets met. Test suite is fast enough for frequent CI/CD runs. Performance is not a limiting factor.

Defect Detection Comparison

Metric	**Actual**	**Target**	**Status**
Total Defects Found During Testing	3-4 (bean loading, mock config, minor edge cases)	>10 expected	Below expectations
Test Pass Rate	100%	95-99% (after fixes)	Exceeds
Bugs Escaped to Production	0 (at submission)	<5% expected	Meets
Undetected Issues in Testing	~2-3 (uncovered branches)	0 ideal	Small gaps remain

Analysis: Test suite is effective at catching integration issues but cannot reveal bugs in untested code paths. The 100% final pass rate indicates all discovered issues were resolved.

Comparison Summary

Dimension	Assessment	Notes
Coverage	Below Targets	Service layer strong; controllers need improvement
Execution Speed	Exceeds Targets	All timing goals met
Defect Detection	Effective	Found and fixed all integration issues
Overall Quality	Good, with Gaps	Solid foundation; enhancement opportunities clear

Quantified Gap Analysis

Achieving target coverage levels would require approximately:

- **Service Algorithm Tests**: +15-20 additional tests (gap: -22% branch)
- **Controller Error Tests**: +20-25 additional tests (gap: -23% branch)
- **Integration Scenarios**: +10-15 additional tests (gap: load, resilience)
- **Total Enhancement**: 45-60 additional tests to reach targets

Effort Estimate: 2-3 additional weeks of focused testing could close identified gaps.

4.4 Steps to Reach Target Levels

A concrete strategy was developed to close testing gaps and achieve target coverage levels:

Phase 1: Algorithm Branch Coverage (Week 1)

Objective: Increase service layer branch coverage from 48% to 65%+

Actions:

1. Analyze JaCoCo reports to identify untested branches in PathfindingService
2. Write specific test cases for:
    - A* algorithm failure cases (no path exists)
    - Heuristic function edge cases
    - Priority queue tie-breaking scenarios
    - Large graph handling (1000+ nodes)
3. Expand DeliveryPathServiceTest with multi-drone constraint combinations
4. Add edge case tests for DroneAvailabilityService (timeout, partial data)

Expected Outcome:

- PathfindingServiceTest: 11 tests → 18-20 tests
- Branch coverage: 48% → 65%
- Effort: 8-10 hours

Phase 2: Controller Layer Enhancement (Week 2)

Objective: Increase controller branch coverage from 42% to 65%+

Actions:

1. Add tests for every error condition in ApiController
2. Test HTTP error status codes (400, 404, 500 scenarios)
3. Test request validation failure paths
4. Test service exception propagation
5. Add null/empty parameter handling tests

Expected Outcome:

- ApiControllerMockitoTest: 25 tests → 40-45 tests
- Branch coverage: 42% → 65%
- Effort: 10-12 hours

Phase 3: Resilience Testing

Objective: Validate system behavior under adverse conditions

Actions:

1. Integrate WireMock for external service simulation
2. Create test scenarios for:
    - Service timeout (slow response, eventual timeout)
    - Partial response (incomplete data)
    - Service unavailability (500 errors)
    - Rate limiting (429 responses)
3. Test graceful degradation (system continues with degraded service)

Expected Outcome:

- New service resilience tests: 10-15 tests
- Documentation of failure modes and recovery
- Effort: 12-15 hours

Phase 4: Load and Concurrency Testing (Week 4)

Objective: Validate system behavior under production-like load

Actions:

1. Set up JMeter or Gatling for load testing
2. Create load profiles:
    - Baseline: 10 concurrent requests
    - Normal load: 50 concurrent requests
    - Peak load: 100+ concurrent requests
3. Monitor metrics: response time, throughput, errors, memory
4. Document performance characteristics

Expected Outcome:

- Load test scenarios: 5-10 scenarios
- Performance baseline established
- Bottlenecks identified (if any)
- Effort: 15-18 hours

Phase 5: Mutation Testing Implementation (Week 5)

Objective: Quantitatively measure test effectiveness

Actions:

1. Integrate PIT (Pitest) mutation testing tool in pom.xml
2. Run mutation tests on core services
3. Identify ineffective tests (mutations not caught)
4. Enhance tests based on mutation score feedback

Expected Outcome:

- Mutation score: baseline (initial measurement)
- Ineffective tests identified and improved
- Effort: 8-10 hours

4.5 Statistical Analysis of Testing Effectiveness

Simple Statistical Metrics Calculated:

Test Coverage Efficiency:

- Lines of code: ~2,500 (estimated)
- Covered lines: ~1,500 (60%)
- Tests per covered line: 143 ÷ 1,500 = 0.095 (9.5 tests per 100 lines)
- Interpretation: Good density; each line covered by ~1 test on average

Defect Detection Rate:

- Integration issues discovered: 3-4
- Integration issues not discovered: 0
- Detection rate: 100% for discovered issues
- Interpretation: Tests effective at catching integration problems

Test Reliability:

- Test pass rate: 143/143 = 100%
- Flaky test rate: 0%
- Interpretation: Tests are stable and repeatable

Coverage Concentration:

- Service layer concentration: 68% ÷ 60% = 113% of average
- Controller layer concentration: 51% ÷ 60% = 85% of average
- Interpretation: Testing concentrated in service logic; controllers undertested

Evidence References

- Figure 4.1: JaCoCo Coverage Exclusions (pom.xml, lines 120-127)
- Figure 4.2: Test Execution Results (target/surefire-reports/TEST-*.txt)
- Figure 4.3: Coverage Metrics by Package (target/site/jacoco/index.html)
- Maven Build Log: Evidence that all 143 tests pass
- Coverage Reports: Full JaCoCo HTML reports showing 60% instruction, 38% branch