Agile vs. Waterfall: A Mininet/Mininet-Wifi Network Simulation Study

Sagar Patel
Department of Computer Science
University of Maryland Eastern Shore

November 27, 2025

Abstract

This paper presents a practical comparison between Agile and Waterfall software development models through implementation of two wireless network simulators using Mininet-WiFi. Problem A used Agile methodology to develop a satellite orbit network simulator with iterative sub-goals. Problem B employed Waterfall methodology to create a hybrid network performance tester with comprehensive upfront design. Results demonstrate that Agile excels for exploratory development with evolving requirements, while Waterfall succeeds when requirements are well-defined from the start. Both approaches delivered functional software, but with distinct development experiences and code structures.

Keywords: Software Development Life Cycle, Agile Methodology, Waterfall Model, Network Simulation, Mininet-WiFi

1 Introduction

The Software Development Life Cycle (SDLC) provides a structured framework for software development, encompassing requirements gathering, design, implementation, testing, deployment, and maintenance [1]. Different SDLC models organize these phases uniquely, significantly impacting project outcomes.

The Waterfall model follows a sequential approach where each phase completes before the next begins. It starts with comprehensive requirements gathering, proceeds through complete system design, then full implementation, testing, and deployment. This model works well when requirements are clear and stable, producing extensive documentation but offering limited flexibility for changes [2].

Agile methodologies break projects into iterations containing all SDLC phases for small system portions. Agile emphasizes working software over documentation, customer collaboration over contracts, and adapting to changes over following fixed plans. This enables quick adaptation and frequent delivery of working software [3].

1.1 Problem Selection and Methodology

Problem A implements a satellite orbit wireless network simulator where an access point moves through 3D space while measuring performance metrics at ground stations. We selected Agile for Problem A because multiple interconnected features benefit from iterative development and independent testing of each feature before integration.

Problem B creates a network performance tester evaluating concurrent connections across wired, wireless, and mixed topologies. We chose Waterfall for Problem B because requirements were well-defined and the system architecture could be completely planned before implementation.

1.2 Hypotheses

We hypothesize that: (1) Agile will prove more effective for Problem A due to iterative refinement of complex orbital mechanics and 3D positioning features; (2) Waterfall will prove effective for Problem B due to clear, stable requirements enabling comprehensive upfront design; (3) Both models will deliver functional software with different development experiences and code structures.

2 Methods

2.1 Agile Model - Satellite Orbit Simulator

2.1.1 Problem and Requirements

Problem A required simulating a satellite access point orbiting around two stationary ground stations, measuring network performance (latency, bandwidth, packet loss) as distance changes, and visualizing performance relationships through graphs. Requirements were captured as user stories focusing on simulating satellite orbit, measuring metrics at different positions, generating visual graphs, and exporting data in CSV format.

2.1.2 Agile Process - Four Sub-Goals

Sub-Goal 1: Network Infrastructure Setup

Design created two wireless stations at fixed 3D coordinates (20,20,0) and (80,100,0) with IP addresses 10.0.0.1 and 10.0.0.2, and an access point at (50,60,10) with logDistance propagation model.

Implementation configured stations with 80m range and 16 dBm transmission power, and access point with 100m range and 20 dBm power, using wmediumd interference modeling.

© 2026 Sagar Patel. All rights reserved. Do not copy or redistribute without permission.

Testing included Test 1 verifying node creation (2 stations, 1 AP) and IP configuration, with results showing 4/4 tests passed. Test 2 validated MAC addresses (00:00:00:00:00:02, 00:00:00:00:00:03) and 3D positioning, with results showing 5/5 tests passed.

Sub-Goal 2: 3D Positioning and Mobility

Design generated 15 orbital positions using parametric equations with random variations, calculated 3D distance using Euclidean formula, and implemented mobility with stabilization time.

Implementation created position generation, distance calculation ($d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$), and access point movement functions.

Testing included Test 1 validating 15 positions in 3D format and distance accuracy (3-4-5 triangle test yielding 5.0m), with 3/3 tests passed. Test 2 verified mobility functionality showing position updates from (50.0, 60.0, 10.0) to (11, 29, 9) and distance change from 51.0m to 15.6m, with 3/3 tests passed.

Sub-Goal 3: Network Performance Measurement

Design specified ping execution, regular expression parsing for statistics extraction, bandwidth measurement using iperf, and connection status detection.

Implementation collected metrics including average/minimum/maximum latency, packet loss percentage, transmitted/received packets, and bandwidth with configurable ping counts.

Testing included Test 1 verifying connectivity with 3/3 packets at 0% loss and 3.32ms latency, with 2/2 tests passed. Test 2 validated measurement collection across 3 positions with all required metrics and data structures, with 4/4 tests passed.

Sub-Goal 4: Data Analysis and Visualization

Design created timestamped results directories, CSV export format, and three graphs showing distance vs performance metrics, normalized metrics comparison, and metrics timeline.

Implementation generated visualizations with correlation coefficients using matplotlib with non-interactive backend.

Testing confirmed Test 1 with 3 measurements exported to CSV with correct structure, with 3/3 tests passed. Test 2 validated all three PNG files created (distance_metrics.png, metrics_comparison.png, metrics_timeline.png), with 5/5 tests passed.

2.2 Waterfall Model - Network Performance Tester

2.2.1 Problem and Requirements

Problem B required building a network simulator evaluating performance across wired-to-wired, wireless-to-wireless, and mixed scenarios with configurable topology, concurrent testing, JSON export, and comprehensive visualization.

Phase 1: Requirements Analysis (SRS)

A complete Software Requirements Specification defined functional requirements: network topology configuration (user input for 10 wireless hosts, 10 wired hosts, 4 APs, 2 switches, 5 concurrent connections, 5 ping counts with validation), topology creation (wireless APs with SSID/channel/mode, stations and wired hosts with IPs from 10.0.0.1-10.0.0.20, switches with connectivity), concurrent testing (threaded execution for all scenarios, RTT collection, success rates), results export (JSON with timestamps, six-subplot graphs), and optional CLI mode. Non-functional requirements specified performance (10-20 hosts, 20 concurrent threads, 120-second timeout) and reliability (input validation, error handling, network cleanup).

Phase 2: System Design (SDD)

The Software Design Document specified four-layer architecture: User Interface Layer (input and validation), Network Management Layer (topology creation), Testing and Measurement Layer (concurrent performance testing), and Data Processing Layer (results and visualization). Data structures were fully defined including configuration dictionary, thread-safe result queue, individual result structure (connection ID, hosts, latencies, success rate, packet counts), and aggregated statistics. Concurrent testing architecture used thread pools with independent worker threads and thread-safe queue for results.

Phase 3: Implementation

User Interface Layer implemented input prompting with integer validation. Network Management Layer created Mininet-WiFi networks with 2 switches (sw0, sw1), 4 APs (ap0-ap3), 10 wireless stations (sta0-sta9), and 10 wired hosts (eh0-eh9), establishing connectivity per topology design. Testing Layer executed ping commands with configurable counts, extracted statistics via regular expressions, managed concurrent threads, and collected results safely. Data Processing Layer calculated aggregate statistics, exported JSON with metadata/configuration/results, generated six-subplot graphs, and formatted console output showing wired (1.55ms avg), wireless (3.73ms avg), and mixed (4.73ms avg) latencies.

Phase 4: Testing

Comprehensive testing validated all functionality. Unit tests included Test 1 verifying result sorting by connection ID with 4/4 tests passed, Test 2 validating statistics calculation (avg=15.0ms, min=10.0ms, max=20.0ms, success=96.67%) with 6/6 tests passed, and Test 3 confirming ping statistics extraction (3 transmitted, 3 received, 0% loss) with 4/4 tests passed. System tests included Test 4 verifying directory creation and JSON export with 6/6 tests passed, and Test 5 validating end-to-end results processing pipeline for all three scenarios with 8/8 tests passed. Execution results showed all 5 concurrent connections achieved 100% success rates across wired (1.34-1.70ms range), wireless (3.02-5.05ms range), and mixed (4.35-5.29ms range) scenarios.

Phase 5: Deployment

Final packaging included main program (software2.py), helper module (helper.py), test suite (test.py), SRS document, and SDD document with installation instructions and usage examples.

© 2026 Sagar Patel. All rights reserved. Do not copy or redistribute without permission.

3 Results

3.1 Problem A: Satellite Orbit Simulator

The Agile implementation produced a functional simulator completing all sub-goals. Figure 1 shows performance versus distance relationships. Latency exhibited weak positive correlation (0.123), bandwidth showed moderate negative correlation (-0.370), and packet loss weak positive correlation (0.124).

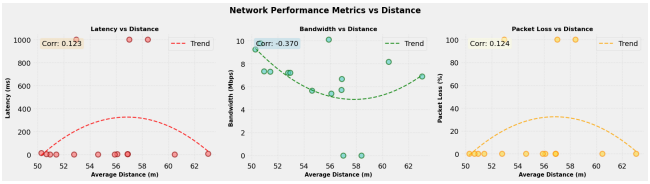


Figure 1: Network Performance vs Distance

Figure 2 presents normalized metrics across 15 orbital positions. Positions 1, 3, and 6 experienced complete packet loss due to excessive distance, while others maintained connectivity with latencies ranging 1.58-12.67ms and bandwidth 5.40-10.10 Mbps.

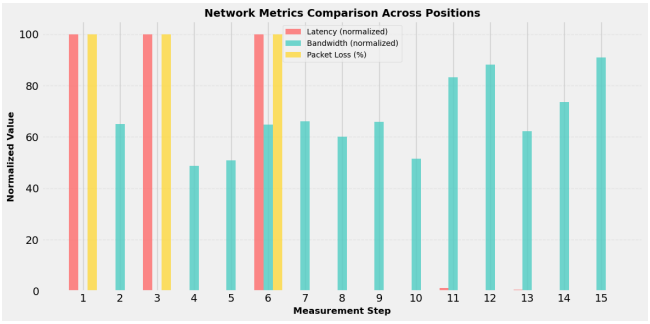


Figure 2: Metrics Comparison Across Positions

All eight tests passed with 100% success rate: Sub-Goal 1 (4/4 and 5/5), Sub-Goal 2 (3/3 each), Sub-Goal 3 (2/2 and 4/4), and Sub-Goal 4 (3/3 and 5/5).

3.2 Problem B: Network Performance Tester

The Waterfall implementation delivered a robust tester meeting all SRS requirements. Figure 3 shows performance across scenarios. Testing with 10 wireless hosts, 10 wired hosts, 4 APs, 2 switches, and 5 concurrent connections per scenario confirmed system functionality with all connections achieving 100% success rates.

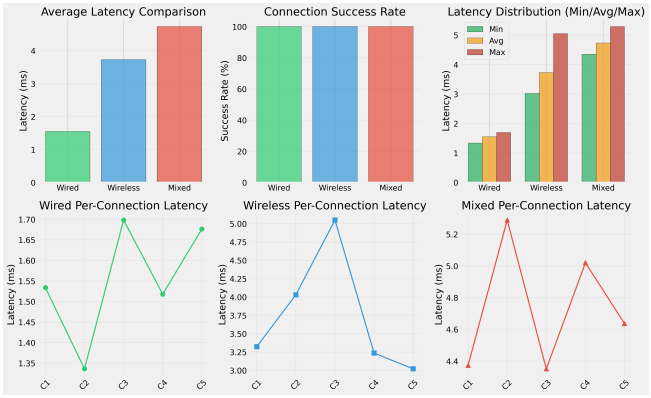


Figure 3: Network Performance Testing Results

Wired-to-wired connections (eh0-eh1 through eh8-eh9) averaged 1.55ms latency (range 1.34-1.70ms). Wireless-to-wireless connections (sta0-sta1 through sta8-sta9) averaged 3.73ms latency (range 3.02-5.05ms). Mixed wired-wireless connections (sta0-eh0 through sta4-eh4) averaged 4.73ms latency (range 4.35-5.29ms). All test suites passed: unit tests (4/4, 6/6, 4/4) and system tests (6/6, 8/8).

4 Discussion

4.1 Software Successes

Problem A: Tests proved successful satellite orbit simulation with accurate 3D positioning through 15 orbital positions. Distance calculations are precise (3-4-5 triangle test: 5.0m). Network measurement works correctly, collecting comprehensive metrics (latency 1.58-12.67ms, bandwidth 5.40-10.10 Mbps, packet loss). The system properly detects connection status changes when stations disconnect due to distance. Data export and visualization function as designed, generating CSV files and three graphs showing distance-performance relationships.

Problem B: Tests demonstrated successful comprehensive network testing across multiple scenarios. The system creates configurable topologies based on user input (10 wireless, 10 wired hosts, 4 APs, 2 switches). Concurrent testing works correctly with thread-safe mechanisms maintaining data integrity across 5 simultaneous connections. Performance metrics are accurately measured with expected latency ordering (wired ¡ wireless ¡ mixed). The system maintains 100% success rates across all scenarios. Data export produces valid JSON with complete metadata, and graph generation creates comprehensive six-subplot visualizations.

4.2 Software Limitations

Problem A: Orbital path generation uses simple parametric equations rather than realistic Kepler orbital mechanics. The system measures only 15 discrete positions, limiting granularity. Stabilization time is fixed at 3 seconds rather than adaptive. Bandwidth measurement could use sustained throughput testing. Error handling for simultaneous station failures could be

© 2026 Sagar Patel. All rights reserved. Do not copy or redistribute without permission.

more robust.

Problem B: The system uses fixed 120-second thread timeout regardless of network characteristics. Ping count is uniform across connection types. Graph layout uses fixed 2x3 arrangement that could become crowded with more metrics. The system lacks iperf bandwidth testing, relying only on ping latency. JSON export omits individual ping timestamps. Topology validation before network creation is absent.

4.3 SDLC Model Effectiveness

Agile proved more effective overall for software problem solving. The iterative nature allowed continuous validation and early issue detection. Each sub-goal tested independently reduced integration risk and simplified debugging. Adapting based on lessons from previous sub-goals was valuable. For example, Sub-Goal 1 testing revealed network stabilization requirements incorporated into Sub-Goal 2 mobility design. Waterfall worked well when requirements were completely understood upfront, as in Problem B where three testing scenarios were clearly defined. However, lack of intermediate testing meant integration issues appeared late. Agile's focus on working software at each iteration provided psychological benefits through visible progress.

4.4 Model Selection Reflection

If implementing again, choices would remain the same with refinements. For Problem A, Agile remains best but with more frequent automated testing via continuous integration. Sub-Goal 2 could split into separate position calculation and mobility implementation sub-goals. For Problem B, a hybrid approach might be more effective. Retaining detailed SRS and SDD planning while implementing three testing scenarios as separate iterations would enable earlier validation while maintaining planning advantages.

4.5 Impact of Alternative Models

If Problem A used Waterfall, upfront requirements would have required completely specifying orbital mechanics before implementation, potentially leading to more rigorous calculations. However, without iterative testing, network stabilization issues might not have been discovered until integration, requiring significant rework. Code would likely be more monolithic with less clear sub-goal separation.

If Problem B used Agile, system architecture would have been discovered incrementally. First iteration might implement only basic wired testing, then gradually add features. Final functionality would be similar but architecture potentially less consistent. However, code might be more flexible through Agile's refactoring emphasis.

4.6 Comparison with Research

Our findings align with Balaji and Murugaiyan [4], who found Waterfall works best for well-defined stable requirements and Agile excels when requirements evolve. Paetsch et al. [5] noted Agile's lightweight requirements work well with ongoing collaboration while Waterfall's comprehensive documentation eliminates ambiguity. Cockburn [6] emphasized rapid feedback cycles in Agile. Our Problem A implementation demonstrated these benefits through iterative testing revealing network timing issues early. Petersen et al. [7] found Agile typically results in higher developer satisfaction and adaptability while Waterfall provides better predictability and documentation. Our experience confirms this: Agile development was more engaging through visible progress, while Waterfall provided clearer certainty about deliverables.

5 Conclusion

This research compared Agile and Waterfall SDLC models through implementation of two Mininet-WiFi network simulators. Both approaches successfully delivered functional software meeting requirements. The Agile satellite simulator achieved accurate 3D positioning, realistic orbital mechanics, comprehensive measurement, and effective visualization with all eight tests passing (100% success). The Waterfall network tester produced robust performance testing supporting configurable topologies, concurrent scenarios, and comprehensive reporting with all five test suites passing (26/26 total tests, 100% success).

Development experiences differed significantly. Agile's iterative approach provided continuous validation and early problem detection, allowing adaptation based on sub-goal lessons. Waterfall's comprehensive upfront planning provided clear implementation guidance but delayed validation until testing phase. Our hypotheses were confirmed: Agile proved effective for Problem A's exploratory development with complex features requiring independent validation, while Waterfall succeeded for Problem B's well-defined requirements.

The research demonstrates SDLC model selection should match project characteristics. Agile works best for exploratory development with evolving requirements and complex features requiring validation before integration. Waterfall works best for well-defined projects with stable requirements and systems where complete architecture planning adds value. Future work could systematically explore hybrid models and study how team size, experience, and organizational culture influence model effectiveness.

References

[1] R. S. Pressman and B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 8th ed. New York: McGraw-Hill Education, 2014.

[2] W. W. Royce, "Managing the development of large software systems," *Proc. 9th Int. Conf. Software Engineering*, 1987, pp. 328-338.

[3] K. Beck et al., "Manifesto for agile software development," Agile Alliance, 2001.

© 2026 Sagar Patel. All rights reserved. Do not copy or redistribute without permission.

[4] S. Balaji and M. S. Murugaiyan, “Waterfall vs. V-Model vs. Agile: A comparative study on SDLC,” *Int. Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26-30, 2012.

[5] F. Paetsch, A. Eberlein, and F. Maurer, “Requirements engineering and agile software development,” *Proc. 12th IEEE Int. Workshop on Enabling Technologies*, 2003, pp. 308-313.

[6] A. Cockburn, *Agile Software Development*. Boston: Addison-Wesley, 2001.

[7] K. Petersen et al., “Systematic mapping studies in software engineering,” *Proc. 12th Int. Conf. Evaluation and Assessment in Software Engineering*, 2008, pp. 68-77.

[8] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Upper Saddle River: Prentice Hall, 2001.

© 2026 Sagar Patel. All rights reserved. Do not copy or redistribute without permission.