2. Change the <group number> in the heading above to your group number
3. Change the <list of group members> to be your group members
4. Change the name of this file to replace the 'N' in the current 'GroupN' with your group number
5. Submit on Blackboard – note ONE SUBMISSION is sufficient for your whole group.

1. Whole Team

"All the contributors to an XP project – developers, business analysts, testers, etc. – work together in an open space, members of one team…."

How was this applied in your team?

Our team implemented the "whole team" concept through scheduled face-to-face sessions in Phoenix Hall. We maintained a consistent weekly schedule: Tuesdays (1-3 PM), Thursdays (3-7:30 PM), and Fridays (4-7:30 PM). This helped in achieving about 10 hours of in-person collaboration every week. We established specific roles during these sessions, rotating members between frontend/backend development, testing, and research works. When implementing the Multi-Modal Path Finder component, for example, we arranged our workspace to have backend developers (working on Neo4j graph algorithms) sitting adjacent to frontend developers (implementing the route visualization), enabling immediate feedback loops.

Benefits as experienced in your team

This approach significantly helped increase our productivity, especially for complex components like the Route Deviation Detector, where on the spot discussions and feedbacks reduced debugging time greatly. This also enabled the team to clarify a lot of questions about API designs for Route Manager and Route Provider services and helped in creating proper design agreements between the frontend and backend developers . KT was also evident when implementing the CI/CD pipeline; members new to GitHub Actions got to understand it through direct observation of more experienced teammates. Our workspace layout facilitated pair programming, allowing pairs to easily consult with others when facing challenges with Flutter's event driven concepts, state management or Neo4j Cypher queries.

Drawbacks as experienced in your team

Absences, such as when Edmund left the team and when Abhishikt was ill for a week, disrupted our collaborative dynamic and made maintaining communications challenging. The Phoenix Hall location created 45+ minute commutes for three team members. The open environment sometimes caused distractions, particularly when multiple pairs discussed complex cases simultaneously. Two members who preferred quiet concentration struggled during active brainstorming sessions for the sustainability scoring algorithm implementation, requiring occasional breaks to maintain productivity.

Lessons learned We learned that in-person collaboration was ultimately increasing the team's productivity. Face-to-face interaction resulted in faster problem resolution compared to asynchronous communication. When collaboratively designing the Route Scorer service's architecture, whiteboard sketches and immediate feedback prevented several potential integration issues. The ability to observe body language during technical discussions helped identify confusion about component responsibilities before it affected implementation. While we incorporated some remote flexibility, our Phoenix Hall sessions formed the cornerstone of our successful development cycle, with measurably higher code quality and feature completion rates during weeks with full attendance.

2. Planning Game

“Planning is continuous and progressive. Every two weeks, for the next two weeks, developers estimate the cost of candidate features, and customers select those features to be implemented based upon cost and business value”

How was this applied in your team? We implemented continuous planning through structured two-week iterations, starting November 6, 2024. Each sprint began with a 2-hour planning session where features were discussed, broken down into feasible stories, and story points were estimated using the modified Fibonacci scale (1-13) in JIRA. For example, the Transit Service integration with Neo4j was initially estimated to be an 8 pointer due to its complexity. Discussions were made on how to break this huge feature into smaller independent stories that can be properly tested and delivered to the consuming teams. The dependencies were also marked to make sure that the teams are aware of when each feature / api is available for consumption by other members of the team.

Tasks were organized hierarchically, through JIRAs epics and each of the epics contained stories for implementation both from frontend and backend side. This approach was chosen so that one feature can be completed across all dependencies and consumers, tested and delivered. This helped a lot in making project management easy as we got to understand where exactly we are with a feature across all the members. For instance, the Map Service and Location components were prioritized early (Sprint 2) as they were dependencies for Route Provider implementation. We maintained a sprint board in JIRA with "To Do," "In Progress," and "Done" columns, reviewing it during our Tuesday sessions. Our backlog were also updated weekly with implementation insights and changing requirements.

Benefits as experienced in your team Our planning approach enabled remarkable adaptability when our architectural needs shifted, for example, when we decided to implement Neo4j for route graph storage rather than PostgreSQL in Sprint 3, we quickly reprioritized tasks. Regular estimation reviews prevented scope creep; our initial Route Scorer implementation was scoped down from 13 to 8 points by focusing on core functionality first. Complex features like the Multi-Modal Path Finder were successfully delivered through incremental implementation across three sprints. Team coordination improved noticeably after Sprint 2 when we standardized our JIRA workflow, resulting in approximately 20% higher velocity in subsequent sprints.

Drawbacks as experienced in your team Estimation disagreements consumed valuable planning time, particularly for technically complex components like Real-Time Notification System integration, where estimates ranged from 5 to 13 points due to varying technical familiarity. As we do not have a UX reference and the frontend UI was completely designed by the developers on the fly, our prioritization occasionally reflected technical interest rather than user value, Flutter UI improvements received high priority despite the backend route finding functionality being more critical to core functionality. Mid-sprint scope adjustments occurred twice, most notably when we discovered unexpected complexity in Neo4j query optimization during Sprint 4, requiring us to reassign resources and adjust deadlines.

Lessons learned We recognized planning as the critical foundation for our system's architecture, which comprised five microservices and a mobile application. Our collaborative estimation sessions, though sometimes inaccurate, fostered a shared technical understanding that proved valuable during implementation. JIRA and Confluence became essential synchronization tools, with daily visits from all team members. We found that our velocity stabilized after Sprint 3, allowing for more accurate forecasting. Ultimately, our planning approach balanced flexibility with structure, enabling us to adapt to technical discoveries while maintaining consistent progress toward our system goals.

3. Customer Tests

“As part of selecting each desired feature, the customers define automated acceptance tests to show that this feature is working”

NOTE – This principle cannot be applied in your team, as you do not have a customer on your team. If you have an opinion on whether this would have been beneficial, please add it here.

This principle couldn't be directly applied in our project since we lacked an external customer to define acceptance criteria. Instead, we internally assumed this role during our Tuesday planning sessions, where we collectively defined acceptance criteria for each user story. For example, for the "Find Routes Between Locations" feature, we specified criteria like "system must return at least two alternative routes" and "sustainability score must be calculated for each route option." These criteria were then translated into around 30 test cases across our codebase, including the TransitServiceTest class which contains 8 test cases covering various route finding scenarios. While our technical tests verified implementation correctness, they sometimes missed user experience expectations, our detailed test cases for the Route Provider services correctly verified algorithm behavior but didn't adequately test ease-of-use aspects. Having actual customer-defined tests would have likely improved our focus on user-facing concerns like route presentation clarity and simplified the sustainability score visualization, areas where we spent additional development cycles after initial implementation. The disconnect between technical correctness and user expectations was particularly evident in our Flutter UI implementation, where we had to refactor the route display component twice to improve usability despite all tests passing.

4. Simple Design

“The team keeps the design exactly suited for the current functionality of the system. It passes all the tests, contains no duplication, expresses everything the authors want expressed, and contains as little code as possible”.

How was this applied in your team? We initially applied simple design when creating our architectural diagrams in October 2024, focusing on essential components needed for route finding and visualization. Our functional architecture document started with 3 core components but expanded to 6 as additional features were proposed. This expansion was evident in the Route Manager module, which grew from a simple API gateway to include active route tracking, deviation detection, and real-time rerouting. Similarly, our use cases expanded from an initial 10 to 27 detailed scenarios, each requiring sequence diagrams that took 3-4 hours to create. This complexity was reflected in our code, the TransitService class grew to over 400 lines, handling multiple responsibilities. Following feedback from our November 15th meeting with Professor Clarke, we refocused during our Thin Slice implementation (November 22-29), prioritizing a functioning end-to-end flow using the Location Service, Route Provider, and a simplified Flutter UI that displayed basic route information without advanced filtering.

Benefits as experienced in your team Starting with a simpler design allowed faster onboarding and understanding. New team members could pitch in to the discussion of the core system flow within a 30-minute walkthrough. Our eventual simplification efforts during the Thin Slice implementation helped us achieve the setting up of inter-communicating micro services within two weeks. This pushed us forward to continue focusing on critical paths like location selection and basic route finding instead of trying to get a complex structure from scratch. The simplified functional architecture also reduced initial development blockers; the data models and modules for core entities like Location, Stop, and Route were analysed and designed in half a sprint by focusing only on the essential attributes.

Drawbacks as experienced in your team The gradual accumulation of features resulted in significant documentation overhead, creating and maintaining 27 complex flows. This led to occasional confusion regarding component boundaries; for example, responsibility for handling route deviations shifted between the Route Manager and client-side code twice during development. Our expanding scope also created technical debt, particularly in the Route Provider service where we initially implemented four different algorithms before settling on A* for path finding, leaving unused code that required cleanup. Balancing ambitious goals with realistic capacity caused tension in Sprint 4 planning, where we had to postpone the Real-Time Notification System implementation to focus on core routing functionality.

Lessons learned We discovered that maintaining simple design requires active discipline throughout the project lifecycle. When we compared the code complexity metrics between our initial components and later additions, the difference was striking, classes developed under our simplified approach had 30% fewer code smells according to SonarQube analysis. The feedback from Professor Clarke proved invaluable in refocusing our efforts on essential functionality. Moving forward, we have established clearer criteria for evaluating new features, asking whether each component truly adds value to the core user experience before implementing it. The experience reinforced that simplicity in design enables clearer implementation paths, better maintainability, and faster adaptation to changing requirements.

5. Pair Programming

“All production software is built by two programmers, sitting side by side, at the same machine”.

How was this applied in your team? We implemented pair programming according to a structured 10-week rotation schedule established in our project development plan. Our rotation assigned four specific pairs per week with a designated observer role, documented in a detailed pair programming schedule matrix. After Edmund's departure in week 3, our team of 8 members formed perfectly balanced pairs. During our Phoenix Hall sessions, each pair shared a single development machine, for instance, when implementing the Deviation Detection service, Naveen and Hariharan worked on one laptop, with responsibility alternating every 45 minutes between writing code and doing real-time code and logic reviews. We also implemented specific protocols for hybrid scenarios: when Evander couldn't attend due to illness during week 5, he connected via Teams screen-sharing with Savio who was taking care of pipelines for the development environment, maintaining collaboration even if they are not working side by side. Each pair is always defined with a specific agenda to complete or follow up. Example, during the 4th week , Naveen and Edmund focused on the Location Service API implementation while Devansh and Abhishikt worked on the Route Scorer calculation algorithms.

Benefits as experienced in your team Pair programming delivered measurable quality improvements, modules developed in pairs showed approximately 40% fewer code smells in SonarQube analysis compared to code written individually during initial prototype phases. When implementing the complex Neo4j path-finding algorithm in the TransitService class (76 lines of traversal logic), the navigator spotted three potential infinite loop conditions that the driver had overlooked, preventing bugs that would have been difficult to identify through testing alone. Knowledge transfer was particularly evident when Abhishikt (with transportation domain expertise) paired with Guoqiang (with strong graph algorithm skills) to implement the multi-modal route construction logic, the resulting haversineDistance calculation correctly weighted walking distances against transit options, something neither could have optimized individually. Our coding standards compliance improved by approximately 75% when measured against our linting rules, compared to around 30% adherence in early, non-paired code samples.

Drawbacks as experienced in your team Despite its benefits, our pair programming approach faced practical challenges. When Edmund absconded for three sessions in week 6, we had to temporarily reassign pairs, disrupting established workflows. Speed mismatches created occasional tension, during the Flutter UI implementation, significant time was spent synchronizing understanding between mobile development experts and those newer to the framework, sometimes extending development time by 20-30% compared to solo work. The constant peer review aspect initially created discomfort for two team members who were used to independent work, requiring additional encouragement during the first two weeks. We also observed diminishing returns for straightforward tasks, implementing simple entity classes didn't benefit much from pairing and consumed twice the developer resources.

Lessons learned Our pair programming experience yielded concrete insights about effective implementation. Code review metrics showed that features developed through pairing had fewer critical issues during code review compared to our initial, individually-developed prototypes. The regular rotation between pairs, switching every two weeks per our schedule, prevented knowledge silos. By project completion, each member had worked directly with 6-7 others, creating redundant expertise across all system components. Our tracking showed higher reported satisfaction (8.5/10 average versus 5/10) when members worked in pairs rather than individually. We conducted intermediate retrospective sessions in order to understand the psychology of the group. This helped us address any issues that arise during pair programming and handle disputes and misunderstandings in a professional setup. We found the driver/navigator role alternation, which we timed at 1-hour intervals, maintained energy and focus throughout our 5-hour Thursday sessions. Based on these outcomes, our retrospective notes from the final week indicated unanimous agreement to continue pair programming in future projects, with refinements to allow flexibility for simpler, well-defined task implementations..

6. Test-Driven Development

“The programmers work in very short cycles, adding a failing test, then making it work”

How was this applied in your team? We implemented TDD through a structured workflow documented in our coding standards document. Before implementation, each pair wrote JUnit tests for backend components and Flutter/Mockito tests for frontend features. Failing tests guided our implementation of a 76-line findRouteBetweenStops() method. When developing the RouteScorerService, we first created testEmptyRoutes(), testRoutesWithOnlyWalkSegment(), and testRoutesWithTransitSegment() to verify scoring behavior across different route compositions. Our Flutter mobile application followed the same approach, before implementing location tracking, we wrote widget tests verifying that coordinate changes properly triggered UI updates. During our Tuesday sessions, pairs would typically write tests and implement it, then perform refactoring the following day after ensuring all tests passed.

Benefits as experienced in your team TDD significantly reduced our manual testing cycles, the Route Deviation Detector component had only 2 reported bugs despite containing complex conditional logic with 5 different deviation scenarios. Our test coverage metrics, tracked via SonarQube, showed 62% coverage for critical path components compared to 15% in early modules developed without strict TDD. The RouteScorerServiceTest exemplified how tests served as living documentation, when Zihan joined that pair in week 7, he could understand the scoring logic by examining the test cases instead of studying the implementation details directly. When refactoring Neo4j query performance in Sprint 5, having 6 comprehensive test cases allowed us to verify that our 30% performance improvement didn't compromise the intended behaviour. Integration between components was notably smoother; the RouteManager and RouteProvider services integrated with only minor adjustments despite being developed by different pairs.

Drawbacks as experienced in your team Several practical challenges were faced during implementation. Four team members without prior TDD experience required a few days to adjust to the workflow, creating an initial productivity dip during Sprints 2 and 3. Testing Flutter UI components proved particularly difficult, the MapView widget required complex mocking of Google Maps dependencies, consuming hours of development time for tests alone. For simple entity classes, writing tests first felt inefficient, adding 30-40% overhead for minimal benefit. We also encountered scenarios where tests passed but missed user experience aspects, our RouteResponse presentation tests verified data correctness but didn't capture usability issues that emerged during actual navigation testing. The suite became temporarily fragile during week 9 when we refactored the underlying Neo4j data model, requiring additional hours of test maintenance.

Lessons learned . Test granularity proved crucial, the most effective tests verified behavior at the service level rather than implementation details. For example, RouteProviderControllerTest focused on validating endpoint responses rather than internal algorithms. We discovered an optimal balance by applying TDD rigorously to complex business logic (Route calculation, sustainability scoring) while using a lighter approach for infrastructure components and simple entities. For Flutter UI testing, focusing on user interactions rather than widget properties significantly improved test resilience. Data collected across our 8 sprints showed that components developed with TDD required approximately 40% less debugging time and had a significant decrease in regressions when modified. This translated to higher velocity in later sprints, our final two sprints delivered features approximately 25% faster than early sprints where TDD discipline was still being established. Based on these measurable benefits, our team committed to maintaining TDD practices but with pragmatic adaptations based on component complexity and criticality.

7. Design Improvement

"Don't let the sun set on bad code. Keep the code as clean and expressive as possible"

How was this applied in your team?
Our team made design improvement a continuous priority by dedicating one pair team specifically to refactoring tasks every week from sprint 4. This ensured that code quality remained high throughout the development process, not just at the end of the project. During our scheduled sessions, these rotating refactoring pairs would review recent code additions, identify areas for improvement, and implement cleaner solutions. We established clear criteria for what constituted "good" versus "bad" code, including readability, appropriate use of design patterns, and adherence to our coding standards etc.,. Additionally, we used SonarQube to identify code smells and technical debt, which helped guide our refactoring efforts. All team members were empowered to flag code that needed improvement, making design enhancement a collective responsibility.

Benefits as experienced in your team
By treating refactoring as an ongoing activity, we maintained a consistently high quality codebase throughout the project lifecycle. This approach prevented technical debt from accumulating and becoming overwhelming later. We found that regular refactoring made our code more maintainable and extensible, which was particularly valuable as we added new features to our Dynamic Sustainable Wayfinding application. The dedicated refactoring pairs often identified opportunities for code reuse, helping us eliminate duplication and create more cohesive modules. Furthermore, the continuous improvement process served as an excellent learning opportunity for less experienced team members, who could observe how code evolved from functional to elegance through thoughtful refactoring.

Drawbacks as experienced in your team
There were no major drawbacks to our design improvement approach. Initially, some team members were concerned that dedicating resources specifically to refactoring might slow down feature development, but we found that the improved code quality actually accelerated development in the long run. Occasionally, there were minor disagreements about whether certain code needed refactoring or was already acceptable, but these discussions typically led to productive conversations about code quality standards.

Lessons learned
We learned that design improvement is most effective when built into day-to-day coding practices rather than treated as a separate phase. The dedicated refactoring pairs helped establish this as a core value in our development process. We discovered that refactoring becomes significantly easier when combined with our comprehensive test suite, which gave us confidence that our improvements didn't break existing functionality. Through this experience, we came to appreciate refactoring not just as a technical necessity but as a valuable practice that improves the overall sustainability of the project and continuously elevates the team's coding standards.

8. Continuous Integration

“The team keeps the system fully integrated at all times”

How was this applied in your team? We implemented continuous integration through GitHub Actions workflows defined in .github/workflows/ci.yml files for each microservice. Our CI pipeline included four sequential stages: build, test, static analysis, and deployment verification, executing within 8-12 minutes on average. For the Maven-based backend services, we configured the pipeline to use Java 17 and run 'mvn clean verify', which executed 94 tests. The Flutter mobile application used a separate workflow that ran 'flutter test' for unit tests and 'flutter build' for both Android and iOS platforms. This was done using a third party client called Bitrise. We established a strict branch protection rule requiring all PRs to pass CI checks and receive at least two approvals before merging, for example, when integrating the Route Provider with Neo4j in week 5, all four passing checks were verified before the code was accepted. Team members operated on a "pull first, push last" protocol, documented in our development guidelines, where everyone began each session by pulling the latest main branch and submitted completed work before ending their session. Our policy treated build failures as blocker issues, with the responsible pair required to either fix the issue immediately or revert the changes.

Benefits as experienced in your team
Continuous integration provided numerous benefits to our Dynamic Sustainable Wayfinding project. It significantly reduced integration issues by catching conflicts early when they were still small and manageable. This prevented the dreaded "integration hell" that often occurs when merging large changes. The automated test runs gave us immediate feedback on the quality of our code, catching regressions quickly. This built confidence in our codebase and allowed us to move faster with new feature development. The practice also improved team coordination, as everyone always had access to the latest working version of the system. This was particularly valuable when developing interdependent components like Multi-Modal Path Finder, ensuring they remained compatible throughout development.

Drawbacks as experienced in your team
Despite its benefits, our CI approach presented some challenges. Environment-specific issues occasionally triggered false build failures, Flutter tests initially failed on the CI environment despite passing locally due to configuration issues on the remote machines, requiring additional hours of troubleshooting in week 4. Maintaining the CI configuration itself consumed development resources; we spent approximately 12 person-hours throughout the project updating workflow files as dependencies evolved. The problems increased when we ran out of credits for our servers and we need to reconfigure deployments from scratch again with a different account etc., updating configurations across all platforms.

Lessons learned
Our experience with continuous integration taught us that the upfront investment in setting up proper automation pays significant dividends throughout the project. We learned to write more modular, testable code since we knew it would be automatically verified. Creating clear branch strategies and merge policies proved essential for managing the flow of changes. We discovered that frequent, smaller integrations were

far preferable to infrequent, larger ones. The practice fostered a team culture of quality and collective ownership, as everyone was invested in maintaining a working build.

9. Collective Code Ownership

“Any pair of programmers can improve any code at any time”

How was this applied in your team?
In our team, we implemented collective code ownership while maintaining clear accountability. During each rotation cycle, within each pair, one person was designated as the primary point of contact (POC) for a specific module of code. This person had deeper knowledge of that module and served as the go-to resource for questions, but crucially, ownership remained collective as any pair could modify any code at any time. Through our rotating pair programming schedule, team members regularly worked on different components of the system, ensuring knowledge distribution. We established comprehensive documentation and coding standards to enable any pair to understand and modify any part of the codebase. When refactoring opportunities were identified, pairs felt empowered to improve the code regardless of who originally wrote it. During our Phoenix Hall sessions, we regularly conducted code walkthroughs to build collective understanding of the entire system. This approach was supported by our GitHub workflow, where pull requests were reviewed by different team members, further spreading knowledge and responsibility across the team.

Benefits as experienced in your team
This shared ownership approach created a strong sense of joint responsibility and notably enhanced our codebase quality. Instead of waiting for the original authors to make changes, any pair could address issues or improve functionality when needed. This prevented bottlenecks and kept development flowing smoothly. There was a significant learning benefit as team members gained familiarity with multiple system components. For example, when implementing the Route Deviation Detector, several pairs contributed improvements based on their experiences with the Location and Map Services, leading to a more robust integration. This cross-pollination of ideas resulted in more innovative solutions and consistent implementation patterns across the application.
Drawbacks as experienced in your team
There were no drawbacks to our collective code ownership approach. Our comprehensive documentation, consistent coding standards, and regular communication effectively addressed the potential challenges that might have arisen.

Lessons learned
We discovered that collective ownership creates a more resilient and flexible team. The practice encouraged everyone to maintain high standards since anyone might work on the code next. It fostered a collaborative mindset where improvements were welcomed rather than seen as criticism of the original implementation. We learned

that this approach reduces project risk by eliminating single points of failure in terms of expertise. The cross-training effect was particularly valuable, as team members developed broader skills and deeper understanding of different aspects of the application.

10. Coding Standard

“All the code in the systems looks as if it was written by a single – very competent – individual”

How was this applied in your team?

Coding standards were consistently applied throughout our Dynamic Sustainable Wayfinding system via several established patterns. We implemented the comprehensive standards outlined in our initial specifications document, which addressed aspects like style guides, language-specific conventions, error handling protocols, and documentation requirements. We always maintained a modular structure influenced heavily by functionality. Architectural consistency was enforced across microservices, with each following identical layered patterns comprising controllers, services, repositories, and entities. Lombok annotations were utilized across entity classes to minimize boilerplate code. Dependency management was centralized in the parent POM file. For our Flutter frontend, we adhered to strict Dart coding conventions. We utilized SonarQube to measure and enforce code quality metrics, including code duplication, maintainability, and reliability. Linting tools were configured for Flutter and SpringBoot frameworks. Documentation style, error handling approaches, and naming conventions were standardized across all modules. Docker configurations were kept uniform, enabling consistent deployment processes. These practices collectively ensured that the codebase appeared to have been written by a single competent developer.

Benefits as experienced in your team

Multiple benefits were realized through the application of consistent coding standards. Developer onboarding was expedited as new team members could quickly comprehend the codebase structure due to recurring patterns. Maintenance efficiency was improved since bug fixes could be more readily applied across services. Code reviews were streamlined, allowing reviewers to concentrate on logical aspects rather than stylistic inconsistencies. Collaboration was enhanced, enabling simultaneous work on different services without diverging implementation patterns. Standardized testing approaches facilitated similar test strategies across services. Due to consistency in data structures and naming conventions, integrations were made easy. The well-defined error handling protocols significantly reduced debugging time, as the system provided clear, actionable error messages.

Drawbacks as experienced in your team

There were no major drawbacks to our coding standards approach. We observed only minimal challenges, such as occasional small styling differences when team members were getting accustomed to the established conventions. Some team members experienced a brief learning curve when adapting to specific coding standards they hadn't previously worked with, particularly with Flutter/Dart conventions or specific Java practices. However, these minor adjustments were quickly overcome through pair programming and code reviews, and did not significantly impact our development process or timeline.

Lessons learned
Standards enforcement was automated through linting and code quality tools to reduce the manual review burden. Comprehensive style guides were documented early to prevent misunderstandings. A balance was struck between consistency and pragmatism, allowing for innovation when standard patterns were insufficient. Periodic review sessions were conducted to evolve standards as the team gained experience. Starter templates were created for common components to accelerate development while maintaining consistency. Knowledge sharing sessions were held more frequently to ensure uniform understanding. A formal process for documenting necessary deviations from standards was established. Standards were built incrementally rather than attempting to implement all guidelines simultaneously. Finally, we found that incorporating consistent fault tolerance mechanisms across the codebase significantly improved system reliability.

11. Metaphor

"The team develops a common vision of how the program works"

How was this applied in your team? Our team embraced the concept of a system metaphor by viewing our Dynamic Sustainable Wayfinding application as a knowledgeable local guide who knows all transportation options and environmental conditions. This metaphor helped us conceptualize how different components should interact and what their responsibilities were. During our initial planning sessions at Phoenix Hall, we created comprehensive diagrams illustrating the system's architecture and workflow, which served as our common reference point. We established a consistent vocabulary for key components like the "Multi-Modal Path Finder," "Route Scorer," and "Location and Map Services" that everyone used when discussing the system. Our functional and technical architecture documents helped solidify this shared understanding, providing clear component descriptions that aligned everyone's mental model of how the system worked. Regular whiteboarding sessions during our in-person meetings reinforced this common vision and allowed us to refine our shared understanding as the project evolved.

Benefits as experienced in your team Having a common understanding of our system significantly improved communication efficiency within the team. When discussing, team members could quickly reference specific components or interactions without lengthy explanations. This shared mental model helped us identify potential issues earlier, as everyone understood how their work fit into the larger system. New features could be designed more cohesively since the entire team had a clear picture of the existing architecture. Our consistent terminology also streamlined documentation, making it more accessible and reducing ambiguity. The visual representations we created served as excellent onboarding tools, helping team members get up to speed quickly when rotating to different parts of the codebase.

Drawbacks as experienced in your team Not having a formal metaphor occasionally led to minor misunderstandings about how certain components should interact, particularly as the system grew more complex. Some newer concepts, like the sustainability scoring mechanism, took longer to become part of our shared understanding since they weren't anchored to a concrete metaphor. At times, team members had slightly different interpretations of how certain edge cases should be handled, requiring additional discussion to align our vision.

Lessons learned We learned that while a formal metaphor isn't always necessary, having a consistent and well-documented shared understanding is crucial for team alignment. Visual representations proved extremely valuable in establishing this common vision. We found that regularly revisiting and updating our system diagrams helped keep everyone synchronized as the project evolved. Creating a glossary of key terms early in the project would have further strengthened our shared vocabulary. Overall, we recognized that investing time in building a common vision pays dividends throughout the project by reducing misunderstandings and enabling more cohesive development.

12. Sustainable Pace

“The team is in it for the long term. They work hard, at a pace that can be sustained indefinitely. They conserve their energy, treating the project as a marathon rather than a sprint”

How was this applied in your team?
Our team implemented sustainable pace through structured, consistent work sessions that respected team members' other commitments. We established regular meeting times (Tuesdays 1-3 PM, Thursdays 3-7:30 PM, and Fridays 4-7:30 PM) at Phoenix Hall, creating a commitment that enabled everyone to plan their schedules effectively. These fixed time blocks ensured we made continuous progress without burning out. We deliberately avoided late-night coding marathons or last-minute rushes by setting realistic iteration goals and tracking our velocity. Our pair programming rotation system distributed the workload evenly, preventing any single team member from becoming overwhelmed. We also maintained a healthy backlog with tasks prioritized based on value and complexity, allowing us to work steadily toward project completion rather than rushing to meet arbitrary deadlines.

Benefits as experienced in your team
The sustainable pace approach yielded several significant benefits for our project. Team morale remained consistently high throughout the development cycle, as members weren't subjected to extreme pressure or unreasonable hours. The quality of our code stayed high because we weren't cutting corners due to time pressure. Our consistent schedule facilitated better work-life balance, allowing team members to fulfill academic and personal responsibilities alongside the project work. Decision-making improved as we weren't making rushed choices under duress. Additionally, our velocity became more predictable over time, enabling more accurate planning and expectation setting for future iterations.

Drawbacks as experienced in your team
While our sustainable pace approach was generally successful, we did face some challenges. Occasionally, the fixed schedule made it difficult to respond to unexpected technical obstacles that required additional attention. Some team members initially felt that the measured pace might limit how much we could accomplish within the project timeframe. Coordination across different time zones sometimes created scheduling complexities for remote participants. Additionally, we observed that momentum on certain complex features could be disrupted when sessions ended at their scheduled time, requiring additional context-switching in subsequent sessions to regain flow.

Lessons learned
From our experience with sustainable pace, we learned that consistency is key to long-term productivity. We found that shorter, more focused sessions often produced better results than longer ones where fatigue might set in. Maintaining a well-organized backlog was essential for ensuring that we always had appropriately sized tasks ready for each session. Regular retrospectives helped us adjust our pace when necessary, finding the right balance between progress and sustainability. We also discovered that celebrating small victories along the way helped maintain motivation during the extended project timeline. Overall, we concluded that a sustainable pace is not about working less, but about working smarter, ensuring that the effort we put in consistently translates to quality output over the entire project duration.

13. Overall Project

Benefits as experienced in your team
Implementing XP practices in our project provided several tangible benefits. The pair programming approach improved our code quality, for instance, when implementing the complex Route Scorer algorithm, the navigator caught several edge cases that the driver had overlooked, preventing potential bugs from entering production code. Our test-driven development practices proved invaluable when integrating the Multi-Modal Path Finder with external transportation APIs; we had a failing test that identified an incompatibility with the Dublin Bus API format before it became a larger issue. The regular face-to-face sessions at Phoenix Hall fostered strong team cohesion, particularly evident when we faced a major architectural decision regarding how to handle real-time transportation updates. Instead of drawn-out debate via messaging, we resolved it efficiently during one Thursday session through collaborative whiteboarding. Our continuous integration setup caught integration issues early, notably when two pairs were simultaneously working on the Location Services and Route Deviation Detector components, a potential conflict was identified immediately upon push rather than becoming a larger problem later.

Drawbacks as experienced in your team
Despite the overall positive experience, we encountered some challenges with our XP implementation. Coordinating schedules for our fixed in-person sessions occasionally proved difficult, especially during midterm periods when team members had competing academic priorities. There was a learning curve associated with practices like TDD for those team members who hadn't previously worked this way, which initially slowed our velocity during the first two iterations. For example, during the implementation of user navigation on mobile, we spent significant time debating the appropriate test strategy before writing any production code. Additionally, while pair programming generally improved code quality, we found it wasn't always the most efficient approach for simple, straightforward tasks like implementing standard UI components or basic CRUD operations. In these cases, the overhead of pair programming sometimes felt unnecessary compared to the value it provided.

Lessons learned
We learned that adapting XP practices to fit our specific context was more effective than rigid adherence to textbook definitions. For example, our hybrid approach to collective code ownership, where one person in each pair served as the point of contact for specific modules, balanced accountability with flexibility. We discovered that investing time in proper test infrastructure early paid dividends later; our comprehensive test suite for the Route Manager component allowed us to refactor it confidently when performance issues emerged. We found that physical co-location during our Phoenix Hall sessions was irreplaceable for complex problem-solving and team building, the solutions we developed for the sustainability scoring algorithm emerged from face-to-face interaction that would have been difficult to replicate remotely. Perhaps most importantly, we learned that XP isn't just a collection of practices but a holistic approach to software development that creates a positive feedback loop: practices like pair programming and TDD improved code quality, which reduced the need for lengthy debugging sessions, which maintained our

sustainable pace, which kept morale high, which in turn improved our collaboration and code quality further.

14. Analysis of the use of Generative AI tools in your project, as experienced in your team

Strengths of Gen AI tools as experienced in your team across the whole development life cycle (where relevant). Please give explicit examples.

Throughout the development lifecycle of our project, generative AI tools provided substantial benefits across multiple phases. During the initial design phase, we used ChatGPT to help draft our functional architecture documentation, particularly for refining the component descriptions. For example, when defining the Real-Time Notification System's responsibilities in our architecture document, AI suggestions helped us comprehensively understand and analyse all notification types (real-time push notifications, background fetch notifications, and socket-based events).

In the implementation phase, GitHub Copilot significantly accelerated our development by suggesting code patterns and implementations. When creating the TransitService class that interfaces with Neo4j for route finding, Copilot helped generate the complex Cypher queries needed for graph traversal. For instance, in our findRouteBetweenStops method, Copilot suggested the *apoc.algo.aStar* algorithm implementation that became central to our path-finding logic. Similarly, when implementing the haversineDistance calculation for geographic distances, AI generated the mathematical formula correctly, saving research time.

For data model creation, AI tools helped maintain consistency across our numerous entity classes. When developing model classes like Location, Stop, Route, and Trip, Copilot suggested appropriate JPA annotations and relationship mappings that aligned with our database schema. For example, the StopTime entity with its composite key implementation benefitted from AI-generated code suggestions that handled the IdClass implementation correctly.

During testing, we used AI to understand best practices of writing tests and generate test cases for complex components. AI helped create comprehensive test scenarios including edge cases like "LOOP_DETECTED_IN_PATH" etc for testing out the validity of the paths generated by us. This improved not just our test coverage significantly, but also showed us that such problems are very common in routing usecases and always needs to carefully considered in solution designing..

For our Flutter mobile application, AI proved invaluable in setting up platform-specific configurations. When implementing Google Maps integration in the mobile apps's especially with gradle management and manifest changes in Android, AI provided the correct implementation pattern for initializing the Maps SDK.

Similarly, it was very helpful in troubleshooting certain issues when it comes to environment setup and pipeline creations etc.,

Documentation was another area where AI tools shined. Throughout our codebase, we used AI to help generate meaningful fuctional comments that clearly explained method purposes, parameters, and return values. For complex classes like GraphInitializer and TrafficAPI, AI helped craft detailed documentation that made the code more accessible to other team members.

Limitations of Gen AI tools as experienced in your team across the whole development lifecyle (where relevant). Please give explicit examples.

Despite their benefits, generative AI tools exhibited notable limitations throughout our project's lifecycle. During system design, we found that AI suggestions for our architecture lacked domain-specific understanding of transportation systems. When designing our Multi-Modal Path Finder component, AI proposed overly simplistic data structures that didn't account for the complexities of public transit schedules and real-time updates, requiring significant human expertise to correct.

In implementation, we encountered accuracy issues with AI-generated code, particularly for complex algorithms. When developing the RouteScorer service, GitHub Copilot suggested sustainability scoring calculations that appeared plausible but contained logical flaws in how different transportation modes were weighted. For example, it incorrectly applied the same emissions factors to both electric trams and diesel buses. Our extensive test suite fortunately caught these errors before they reached production.

The Neo4j graph database integration presented particular challenges where AI tools fell short. When implementing the GraphInitializer service, Copilot suggested Cypher queries that didn't properly handle relationship properties or maintain bidirectional relationships between nodes. The code for adding TRANSFER edges between stops required complete human rewrites, as the AI-generated version created incorrect graph structures that would have caused routing failures.

For our Flutter frontend, we observed that AI tools struggled with platform-specific nuances. When setting up the Google Maps integration for iOS, Copilot suggested outdated permission handling approaches that were incompatible with iOS 18. Similarly, when implementing the location tracking features, AI-generated code didn't properly handle background location updates or respect battery optimization settings. At times if we tried to generate models with AI giving them sample jsons, instead of creating crisp and clean code, it hallucinates and fails to create simple reusable code, ending up creating complex ~ 3 x 200 lines of code which a human if can properly thought through can complete within 60 lines in a simplistic and logical structure.

Testing revealed further limitations. AI-generated test cases for our TransitService often failed to consider edge conditions specific to transportation systems, such as the handling of transit transfers or service disruptions. When testing the active route management functionality, the AI didn't generate tests for critical scenarios like handling temporary GPS signal loss or sudden transport service cancellations.

Documentation generated by AI often lacked precision and context awareness. JavaDoc comments for complex classes like RouteProviderController sometimes contained generic descriptions that didn't properly explain the component's specific role in our architecture or important implementation details about its interaction with other services.

Perhaps most significantly, we discovered that AI tools struggled with our evolving codebase. When code structures changed significantly, such as our refactoring of the route data model midway through development, AI tools continued to suggest patterns based on the previous implementation, leading to integration issues. This was particularly evident when we modified our Stop entities to align better with GTFS data formats, AI tools persistently suggested the outdated entity relationships.

Lessons learned

We learned that optimal AI utilization is essential and requires a thoughtful, balanced approach. When working with the DAMN system's domain-specific components, like the Transit Service and Route Scorer, we discovered that providing AI tools with detailed context significantly improved the quality of suggestions. For example, when seeking help with Neo4j Cypher queries, including sample data structures and expected outputs resulted in more accurate code generation.

We established an effective review protocol where all AI-generated code underwent peer review before integration into our codebase. This practice proved crucial when implementing the graph-based route finding algorithms, where AI suggestions also contained subtle logical errors that only domain knowledge could help us clearly identify. Our pair programming approach complemented this process well, with the navigator often critically evaluating AI suggestions while the driver implemented the code.

The team noticed that AI tools provided the greatest value for standardized, well-established patterns rather than novel or highly specific implementations. For instance, when generating entity classes like Location, Stop, and Route with standard JPA annotations, AI suggestions were mostly correct and saved considerable time. However, for the complex multi-modal route construction logic or sustainability scoring algorithms, AI proposals required significant modifications to align with our specific requirements.

We discovered that AI tools worked best as assistants rather than drivers of our development process. When we approached AI with specific questions about implementing the Route Deviation Detector, we received helpful guidance; but when we relied on it to design the entire component, the results lacked cohesion with our overall architecture. This reinforced the importance of maintaining human oversight for architectural decisions and complex logic.

Documentation tasks revealed another important lesson: AI-generated content required thorough fact-checking. When using AI to document our APIs or explain complex algorithms in comments, we needed to verify accuracy and completeness, particularly for domain-specific terminology around transportation modes and geospatial concepts.

Perhaps most importantly, we learned to view AI (weather it is GPT, Claude or Copilot) as a tool for amplifying our understanding and productivity rather than replacing it. Our most successful outcomes occurred when team members with strong domain knowledge directed AI tools toward specific goals, leveraging them to accelerate implementation while maintaining architectural integrity and business logic correctness. This balanced approach enabled us to benefit from AI assistance while avoiding its limitations..

AI Tools used : Chat GPT - 4o and 4.5 , Claude 3.7 Sonnet and Copilot.