

# Shenkar College of Engineering

Software Engineering Department

# NavSight

## *Beyond GPS: Precision Navigation in GPS-Denied Environments*

Software Engineering B.Sc. Final Project: Project Book

### AUTHORS

Roey Ben Harush · 315676163

Tamir Sobuh · 206029084

Morad Zubidat · 208156828

### SUPERVISORS

Mr. Amit Dunskey · Dr. Dvir Ross

Approved by: Amit Dunskey

Date: \_\_\_\_\_

Approved by: Dvir Ross

Date: \_\_\_\_\_

7/20/19  
[Handwritten signatures and dates]

*A thesis submitted in partial fulfilment of the requirements for the degree of B.Sc. in Software Engineering.*

July 2026

# Table of Contents

|                                                                                    |    |
|------------------------------------------------------------------------------------|----|
| <b>Glossary</b>                                                                    | 4  |
| Generic Terminology                                                                | 4  |
| NavSight-Specific Terminology                                                      | 6  |
| <b>1 Introduction</b>                                                              | 14 |
| 1.1 Problem Description                                                            | 14 |
| 1.2 Motivation                                                                     | 16 |
| 1.3 Goals                                                                          | 19 |
| 1.4 Overview of the NavSight Approach                                              | 19 |
| 1.5 Usage Scenarios                                                                | 20 |
| 1.6 Audience                                                                       | 21 |
| <b>2 Literature Survey</b>                                                         | 22 |
| 2.1 Problem Survey                                                                 | 22 |
| 2.2 Solution Survey                                                                | 24 |
| 2.3 Discussion and Conclusions                                                     | 26 |
| <b>3 Technological Survey</b>                                                      | 28 |
| 3.1 Application Platform: React Native → Native Android (Kotlin + Jetpack Compose) | 28 |
| 3.2 Vision and Math Core: C++17 + OpenCV over JNI                                  | 28 |
| 3.3 Fusion Filter: Complementary Filter → Error-State EKF with MSCKF Updates       | 29 |
| 3.4 Monocular Depth: MiDaS Retained After Benchmarking Depth-Anything              | 29 |
| 3.5 Maps and Routing: Google APIs → OpenStreetMap Held Locally                     | 30 |
| 3.6 Map Matching: OSRM /match → On-Device Viterbi (LocalMatcher)                   | 30 |
| 3.7 Test Tooling: Deterministic Replay Harness and CI Scoring                      | 31 |
| 3.8 Summary of the Chosen Stack                                                    | 31 |
| <b>4 Requirements and Specification</b>                                            | 33 |
| 4.1 Functional Requirements                                                        | 33 |
| 4.2 Non-Functional Requirements                                                    | 36 |
| 4.3 Specification: The Scope of the NavSight Project                               | 37 |
| 4.4 Deviations from the Signed SRS                                                 | 38 |
| <b>5 System Architecture</b>                                                       | 39 |
| 5.1 High-Level System Architecture                                                 | 39 |
| 5.2 Data Design                                                                    | 42 |
| 5.3 Processes                                                                      | 43 |
| 5.4 The Life-Cycle View of the System                                              | 44 |
| 5.5 Roles                                                                          | 46 |
| 5.6 Deployment and Build Configuration                                             | 46 |
| <b>6 System Design</b>                                                             | 48 |
| 6.1 Use Cases                                                                      | 48 |
| 6.2 Structural Design: Modules and Classes                                         | 50 |
| 6.3 Data Design                                                                    | 54 |
| 6.4 Interactions                                                                   | 56 |

|                                                                |            |
|----------------------------------------------------------------|------------|
| 6.5 Software Architecture Pattern .....                        | 60         |
| 6.6 Concurrency and Threading Model.....                       | 61         |
| <b>7 Implementation .....</b>                                  | <b>63</b>  |
| 7.1 Algorithm Design .....                                     | 63         |
| 7.2 Interfaces (UI Walkthrough) .....                          | 71         |
| 7.3 Development Environment .....                              | 74         |
| 7.4 Programming Languages .....                                | 74         |
| 7.5 Limitations of the System .....                            | 74         |
| 7.6 Installation and Configuration.....                        | 74         |
| 7.7 Information Security and Privacy .....                     | 75         |
| 7.8 Risk Management.....                                       | 75         |
| 7.9 Exceptions Management.....                                 | 76         |
| 7.10 Version Control .....                                     | 77         |
| 7.11 Project Management .....                                  | 77         |
| 7.12 Code .....                                                | 80         |
| <b>8 System Validation .....</b>                               | <b>81</b>  |
| 8.1 Proof of Concept: Early Feasibility .....                  | 81         |
| 8.2 Validation Methodology and Test Environment.....           | 82         |
| 8.3 Functional Validation: Executed Test Cases .....           | 83         |
| 8.4 Performance Evaluation.....                                | 89         |
| 8.5 Accuracy Experiments .....                                 | 89         |
| 8.6 Requirement Coverage Matrix .....                          | 93         |
| 8.7 Development Evolution and Bug Resolution .....             | 94         |
| 8.8 Real-World Case Study: Route A Under Active Jamming.....   | 95         |
| 8.9 Validation Conclusion.....                                 | 98         |
| <b>9 Summary, Evaluation, Conclusions and Future Work.....</b> | <b>100</b> |
| 9.1 Summary .....                                              | 100        |
| 9.2 Evaluation and Conclusions .....                           | 100        |
| 9.3 Future Work.....                                           | 102        |
| 9.4 Closing Statement .....                                    | 102        |
| <b>10 References .....</b>                                     | <b>104</b> |
| <b>Appendix A: Background Material .....</b>                   | <b>106</b> |
| <b>Appendix B: Test Data and Environment .....</b>             | <b>109</b> |
| <b>Appendix C: Validation Procedures .....</b>                 | <b>110</b> |
| <b>Appendix D: Calibration Procedure .....</b>                 | <b>111</b> |
| <b>Appendix E: JNI Interface Reference .....</b>               | <b>112</b> |
| <b>Appendix F: Offline OSM Asset Format .....</b>              | <b>114</b> |
| <b>תקציר .....</b>                                             | <b>116</b> |
| <b>הכרת תודה .....</b>                                         | <b>117</b> |

# Glossary

This glossary defines the vocabulary used throughout the book, generic navigation and computer-vision terms first, then the names this project gave its own parts. It is placed ahead of the lists of figures and tables so that every term is defined before it is used.

## Generic Terminology

Table A: Generic terminology used throughout this book.

| Term                             | Definition                                                                                                                                     |
|----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| GNSS                             | Global Navigation Satellite System, the family of satellite constellations (GPS, GLONASS, Galileo, BeiDou) used for positioning.               |
| Jamming / Spoofing               | Radio interference that masks GNSS signals / counterfeit signals that induce a false position.                                                 |
| Multipath                        | GNSS error caused by signals reflecting off buildings before reaching the receiver.                                                            |
| VIO                              | Visual-Inertial Odometry, estimating ego-motion by fusing camera and IMU measurements.                                                         |
| Dead Reckoning (DR)              | Advancing a position estimate from speed and heading alone, without external fixes.                                                            |
| Trilateration                    | Fixing a position from measured <i>distances</i> to known transmitters, as GNSS does.                                                          |
| VPS (Visual Positioning Service) | Google's cloud service that localizes a phone by matching its camera view against pre-mapped street imagery.                                   |
| ARCore / ARKit                   | Google's and Apple's on-device augmented-reality frameworks, which track device motion but depend on platform services for global positioning. |
| SLAM                             | Simultaneous Localization and Mapping: building a map of an unknown environment while                                                          |

| Term                       | Definition                                                                                                                                                     |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                            | simultaneously tracking the sensor's position within it. ORB-SLAM is the best known monocular implementation.                                                  |
| Loop closure               | Recognizing a previously visited place and correcting the accumulated drift back to zero. It requires the trajectory to revisit somewhere it has already been. |
| IMU                        | Inertial Measurement Unit, the accelerometer + gyroscope package.                                                                                              |
| EKF / error-state EKF      | Extended Kalman Filter; the error-state form estimates corrections to a nominal state, better conditioned for orientation.                                     |
| MSCKF                      | Multi-State Constraint Kalman Filter, a VIO technique that uses feature tracks across a sliding window of camera poses as filter updates [2].                  |
| KLT                        | Kanade-Lucas-Tomasi feature tracking via pyramidal optical flow.                                                                                               |
| Preintegration             | Summarizing IMU samples between camera frames into a single relative-motion constraint [12].                                                                   |
| AHRS / Madgwick            | Attitude and Heading Reference System; Madgwick's filter is a computationally cheap orientation estimator [3].                                                 |
| ZUPT                       | Zero-velocity update, a filter correction applied when the device is detected stationary.                                                                      |
| Monocular depth estimation | Predicting per-pixel scene depth from a single camera image with a neural network.                                                                             |
| IPM / ground plane         | Inverse Perspective Mapping, using the known height of the camera above a planar road to recover metric scale.                                                 |

| Term           | Definition                                                                                                                      |
|----------------|---------------------------------------------------------------------------------------------------------------------------------|
| HMM / Viterbi  | Hidden Markov Model; the Viterbi algorithm finds the most probable hidden-state sequence, here, the road segments traveled [4]. |
| Map matching   | Aligning a measured trajectory to a road network.                                                                               |
| OSM / Overpass | OpenStreetMap, the open, freely licensed world map database [14]; Overpass is its query API.                                    |
| R-tree         | A spatial index structure enabling fast nearest-neighbor queries over geometric data.                                           |
| JNI / NDK      | Java Native Interface / Native Development Kit, the mechanism by which the Kotlin app calls the C++ core.                       |
| $\chi^2$ gate  | A statistical test that rejects a measurement whose residual is improbably large under the filter's own uncertainty.            |
| RANSAC         | Random Sample Consensus, robust model fitting that tolerates outliers by voting over random minimal subsets.                    |

## NavSight-Specific Terminology

Table B: NavSight-specific terminology.

| Term                | Definition                                                                                                                                               |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| The dot             | The raw VIO trajectory point, NavSight's internal, unconstrained position estimate.                                                                      |
| Rail ball           | The displayed position: the dot's motion projected onto the OSM road graph by the map matcher, so it rides the road like a ball on a rail.               |
| Vote / zero-witness | The two-sided speed-measurement taxonomy: road points whose flow is significant and coherent <i>vote</i> a speed; points that could have seen motion but |

| Term                    | Definition                                                                                                                               |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
|                         | did not <i>witness</i> zero. $\geq 5$ votes $\rightarrow$ median speed; $\geq 5$ zero-witnesses $\rightarrow$ exactly 0.                 |
| Inertial bridge         | The bounded accelerometer-based speed propagation used between visual measurements (full trust $\leq 6$ s).                              |
| Per-gait K              | The per-motion-mode scale calibration (walk / run / vehicle) applied to visual speed.                                                    |
| VIO health chip         | The UI status element summarizing calibration and tracking confidence.                                                                   |
| Locked way / rail bonus | Map-matcher terms: the road currently followed and the transition bonus for staying on it, giving road-following stability at junctions. |
| Mount height $h$        | The calibrated height of the phone's camera above the road plane, the single geometric constant behind metric speed.                     |
| Match window            | The sub-sampled window of recent raw-trajectory points submitted to the matcher each tick.                                               |
| Replay harness          | The desktop build of the unmodified native engine that re-processes recorded rides deterministically for A/B validation.                 |
| Event counters          | -180 in-engine atomic counters serialized into session telemetry, the observability backbone of validation.                              |

## List of Figures

Figure 1.1: How GNSS fails in a city. Blocked signals (denial), reflected signals locking the receiver onto delayed echoes (multipath), and a low-power transmitter drowning out or counterfeiting the satellite signal (jamming / spoofing).

Figure 1.2: The competitive landscape. Mainstream navigation and micromobility apps depend on GNSS and connectivity; visual positioning services depend on the cloud; academic SLAM is self-contained but not built for riders. The self-contained, rider-oriented quadrant is where NavSight sits.

Figure 2.1: The smartphone IMU. Left: a MEMS 6-axis inertial sensor (InvenSense MPU-6050 on a breakout board; photo: Nevit Dilmen, Wikimedia Commons, CC BY-SA 3.0). Right: the Android sensor axes on the device body, accelerometer and gyroscope measure specific force and angular rate about x, y, z.

Figure 5.1: The NavSight system architecture: nested tiers from the Android application down through the JNI bridge to the native processing core and the map layer.

Figure 5.2: The session state machine: launch and permissions, calibration, the tracking composite state with its two degraded sub-modes (inertial-only and visual-only), suspension, and session end.

Figure 6.1: Use-case diagram. The Rider's session, calibration, and search flows; the Developer/Validator's recording, debugging, and replay flows; and the system-initiated extensions (degrade, recover, re-latch) of the tracking flow.

Figure 6.2: The SDD-era class diagram of the core path. As-built renames: NativeLib → NativeBridge (object with 47 active external methods; one legacy frame-path method is retained commented out), VisionModule → VioEngine/Tracker, and MainActivity's state-holder role moved into NavSightViewModel; the VisionOutput/IMUPreintegrator structure is unchanged.

Figure 6.3: The SDD activity diagram of a session: permission gate, calibration loop, the real-time tracking loop with its two degraded branches (DR-only on low-light/blur; visual-only on IMU error), pause/resume, and session summary. The as-built flow matches, with pause realized as OS-lifecycle suspension plus explicit Reset (§5.4).

Figure 6.4: The SDD sequence diagram for a single frame. As-built mapping: :MainActivity → SensorRepository (via CameraUi's analyzer), :NativeLib → NativeBridge/native-lib.cpp, :VisionModule → VioEngine::processFrame → Tracker::processFrame; the fusion step drawn inside VisionModule is the EKF inside Tracker.

Figure 6.5: The SDD component diagram: Android Runtime (Presentation, Application Logic, Orchestrator) and Native Runtime (JNI Interface, Processing Core with Vision / IMU / Sensor Fusion). The as-built system preserves this shape exactly and adds the Tier-2 map stack of Figure 5.1.

Figure 7.1: The error-state EKF pipeline. A 15-DOF core with self-calibration (19-DOF total), IMU propagation, the three measurement-update families, and the  $\chi^2$  acceptance gate.

Figure 7.2: The ground-plane (IPM) speed estimator: per-pixel depth from the calibrated mount height, per-point speed with its own noise floor, the two-sided vote taxonomy, and the inertial bridge.

Figure 7.3: HMM map matching: candidate segments as HMM states, the emission and transition scoring with the shipped constants, Viterbi decoding, junction steering, and the graph-constrained rail ball.

Figure 7.4: The live application at standstill before a ride. Speed is locked at 0 km/h, VIO chip green, the purple position ball resting on the road graph.

Figure 7.5: A rear-camera frame during a validation ride, showing the amber ground-plane sampling region and per-point optical-flow arrows (green: template-matching reference; red: production KLT).

Figure 7.6: Project timeline: specification, proof of concept, engine, productization, and validation phases against the signed milestones.

Figure 8.1: Per-frame feature-tracking time on-device ( $n = 98$ ) against the 43 ms camera frame interval and the 200 ms SDD budget; the bulk of the distribution sits well below the frame interval itself.

Figure 8.2: Speed vs time on the primary ride. Thin lines are the raw 1 Hz / per-frame signals, bold lines the 5 s means; red  $\times$  marks the 20 of 91 GPS reference samples rejected as physically implausible.

Figure 8.3: The end-of-ride standstill, zoomed to 5 km/h, the estimate never exceeds 0.9 km/h and locks at exactly 0.0 for 27 frames; the display reads 0 km/h throughout.

Figure 8.4: The validation route in local metric coordinates about the start point, the health-verified GPS track (93 fixes, median accuracy 4 m) that serves as the reference for Figure 8.2 and the  $\rho = 0.91$  ratio.

Figure 8.5: Travelled-distance accuracy on the two independently referenced routes: 93% against the map-measured route, 91% against health-verified GPS.

Figure 8.6: Offline A/B on identical recorded input; the inertial bridge adds +24% integrated distance through turns.

Figure 8.7: Route A ground truth. The route measured with the Google Maps distance tool (1.28 km), a jamming-immune reference.

Figure 8.8: NavSight's matched position over the full Route A ride, colored by time. The track stays on the road throughout, and the return leg after the U-turn re-traces the same road.

Figure 8.9: Cumulative travelled distance on Route A. The jammed GPS reference accumulates 1,705 m on a route that truly measures 1,280 m, while NavSight integrates 1,195 m.

Figure 8.10: Speed channels on Route A. The ground-plane (IPM) speedometer and the raw dot speed track each other closely, while the jammed GPS trace rides visibly above both, consistent with the +33% distance inflation it accumulates over the ride.

## List of Tables

Table A: Generic terminology used throughout this book.

Table B: NavSight-specific terminology.

Table 1.1: The navigation landscape on two axes, external dependence vs. rider focus.

Table 2.1: Surveyed VIO systems compared on the axes that matter on a phone.

Table 3.1: Technology decisions and the rejected alternatives, with the measured reason each alternative lost.

Table 5.1: The streaming data paths and their producers and consumers.

Table 5.2: The bundled offline-search assets.

Table 6.1: Application-layer classes and their responsibilities.

Table 6.2: Native-core components and their responsibilities.

Table 6.3: The five families of the JNI contract.

Table 6.4: The VioData per-frame state snapshot.

Table 6.5: One camera frame as a twelve-message exchange.

Table 6.6: The threading model: every thread, its rate, and what it must never block.

Table 7.1: The 19-dimensional error state, the classical 15-DOF VIO core plus four self-calibration rows.

Table 7.2: Risk register, likelihood, impact, and the shipped mitigation for each identified risk.

Table 7.3: Project milestones with signed evidence.

Table 8.1: The validation environment.

Table 8.2: The thirteen executed system-level test cases with their measured evidence.

Table 8.3: Requirement-coverage matrix, every Chapter 4 requirement traced to the test cases that validate it.

Table 9.1: Original targets versus delivered results.

# Abstract

Modern navigation depends almost entirely on Global Navigation Satellite Systems (GNSS), an umbrella term for the various satellite navigation constellations, including the U.S.-operated GPS, Russia's GLONASS, Europe's Galileo, and China's BeiDou. GNSS could potentially fail exactly where it is needed most: in dense urban canyons, in tunnels, and under radio interference. In the project's own operating region this is not a theoretical concern, testing was carried out in Haifa under real, ambient GPS jamming, where the GPS receiver was observed to freeze for over a minute while moving and to inflate a measured route's length by 33%.

NavSight is a smartphone-only navigation system that keeps a rider's position on the map without depending on GNSS. It is a native Android application (Kotlin / Jetpack Compose) built around a C++ visual-inertial core accessed over JNI: an error-state Extended Kalman Filter fuses KLT optical-flow camera tracking with IMU preintegration; heading is maintained by a gyro-primary Madgwick AHRS with a startup compass alignment and a weak, disturbance-gated compass correction thereafter; metric speed is measured from optical flow over the road plane at a calibrated camera height, bridged inertially through visual dropouts; and an on-device hidden-Markov map matcher (Viterbi-decoded, Newson–Krumm style) constrains the displayed position to the OpenStreetMap road graph held locally on the device, destination search runs against an Israel-wide index bundled in the APK, and road geometry around the rider is cached on-device so previously visited areas work with no connectivity. Positioning itself never uses a network or a satellite signal.

The system was validated on real urban rides under active GNSS jamming on a Samsung Galaxy S21 Ultra. It achieved 91–93% travelled-distance accuracy against two independent references, a jamming-immune map-measured route and health-screened GPS; median moving-speed agreement within 3.5% of a health-verified GPS reference; an exact 0 km/h standstill readout; and a median per-frame tracking cost of 15.4 ms, thirteen times under the 200 ms real-time budget. On the route where jammed GPS overstated the distance by 33%, NavSight stayed within 7% of the map-measured truth. All 13 executed system test cases pass on the validated build.

**Keywords:** Computer Vision, Visual-Inertial Odometry, Dead Reckoning, GNSS-Denied Navigation, Map Matching, Android.

# Acknowledgments

We would like to thank our supervisors, Mr. Amit Dunskey and Dr. Dvir Ross, for their guidance, technical review, and steady support throughout the year, and the Software Engineering faculty at Shenkar College of Engineering for the knowledge and tools that made this project possible.

# 1 Introduction

This chapter presents the problem NavSight addresses (§1.1), the motivation behind it (§1.2), the project's goals (§1.3) and approach (§1.4), and the people it serves (§1.5–§1.6). The vocabulary used throughout the book is defined in the Glossary, which precedes this chapter in the front matter.

## 1.1 Problem Description

Nearly every navigation product in daily use (car navigation, ride-hailing, delivery logistics, micromobility apps) is built on a single foundation: Global Navigation Satellite Systems (GNSS), whether GPS, GLONASS, Galileo, or BeiDou. A GNSS receiver fixes its position by trilateration from the signal travel time of satellites 20,000 km overhead. Under open sky the result is accurate to a few meters, updated every second, effectively free. It is one of the great infrastructure successes of the past decades, and it is fragile in three specific, well-understood ways.

**Denial.** The satellite signal arriving at street level is extraordinarily weak (on the order of  $10^{-16}$  W). Anything substantial between receiver and sky (a tunnel roof, an underground parking structure, the concrete mass of a dense city block) blocks it outright. Urban canyons add a subtler failure: tall façades reflect the signal, so the receiver locks onto delayed echoes (multipath) and reports positions that are plausible but tens of meters wrong. These are precisely the environments where a rider most needs to know which turn to take.

**Jamming.** Because the legitimate signal is so weak, drowning it out takes very little transmit power. Radio interference, whether deliberate area denial or incidental noise, prevents the receiver from locking at all, and it affects whole regions simultaneously, with no advance warning and nothing an individual user can do about it.

**Spoofing.** The most misleading of the three failures: counterfeit GNSS signals feed the receiver a false but internally consistent position. The system fails while looking perfectly healthy, the map continues to move, smoothly, in the wrong location.

Figure 1.1 illustrates the three failure modes on one street scene.

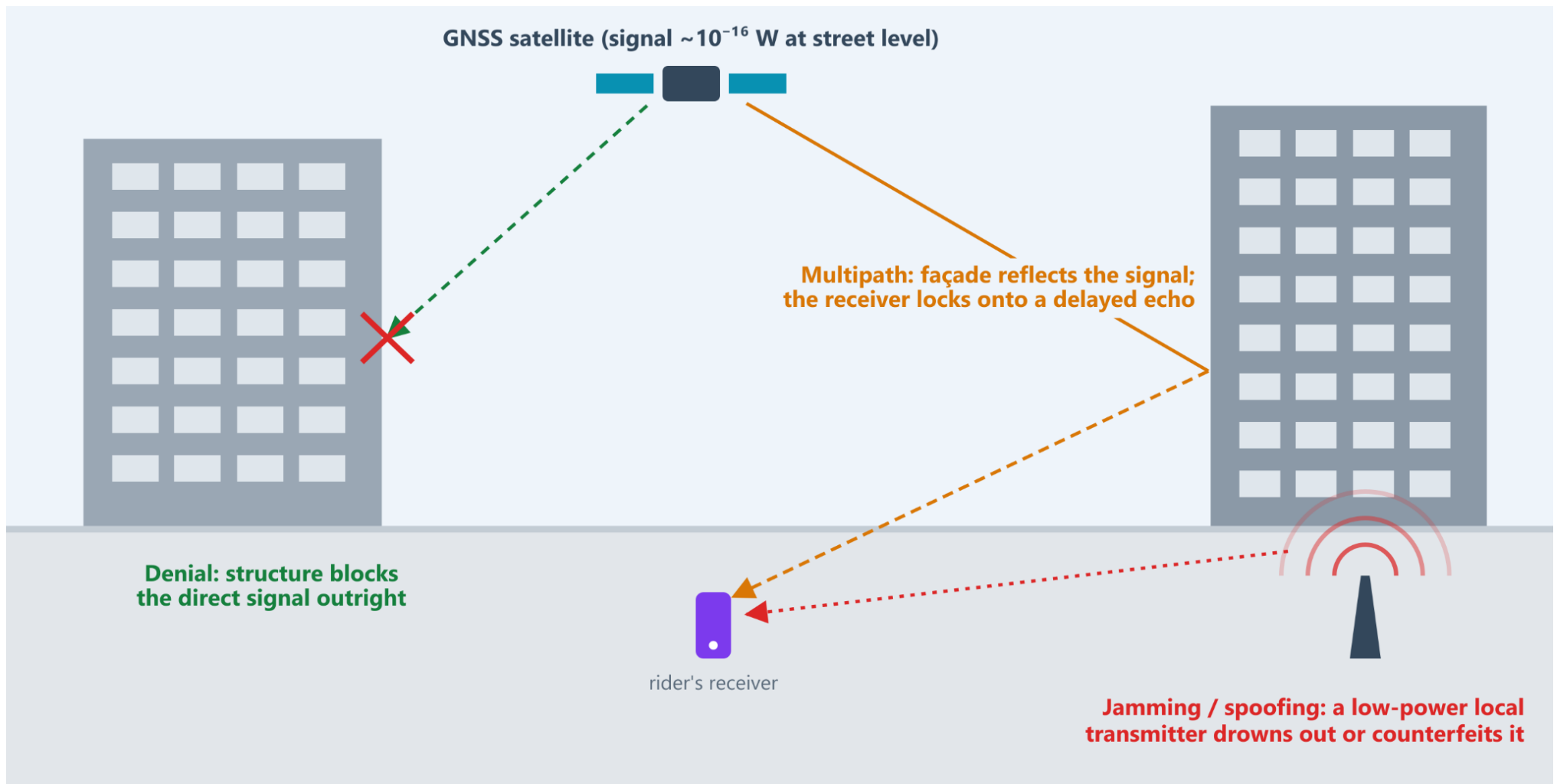


Figure 1.1: How GNSS fails in a city. Blocked signals (denial), reflected signals locking the receiver onto delayed echoes (multipath), and a low-power transmitter drowning out or counterfeiting the satellite signal (jamming / spoofing).

For this project, the problem was not hypothetical. Development and validation took place in Haifa during a period of persistent, wide-area GNSS jamming. On instrumented rides, the phone's GPS receiver was observed to freeze for 61 seconds while the vehicle was moving, and to inflate the measured length of a 1,280 m route by 33% through fix scatter, errors that make GNSS unusable not only for navigation but even as a measurement reference (Chapter 8 quantifies both failures from recorded telemetry). A navigation system for such an environment cannot merely tolerate GNSS outages; it must work as if satellites do not exist.

## 1.2 Motivation

Three observations motivated NavSight.

**GPS failure means no navigation.** When the position is lost, mainstream navigation apps stop dead or silently drift; the rider is left guessing at every junction. For a pedestrian this is an annoyance. For an e-scooter rider moving at 30 km/h in traffic, glancing at a frozen or teleporting map is actively dangerous. The failure arrives exactly when attention is scarcest.

**The sensors already exist but are underused.** Every modern smartphone carries a camera, an accelerometer, and a gyroscope, together sufficient, in principle, to measure ego-motion continuously: the camera sees how the world streams past, the IMU feels every acceleration and rotation. Robotics has fused these two sources for two decades under the name visual-inertial odometry [1][8]. Consumer navigation software leaves this capability almost entirely untapped: the phone in a rider's mount is pointing its camera at the road the whole ride, and nothing uses that information.

**No reliable offline fallback exists today.** The closest prior art to NavSight is not another navigation app but the camera-based positioning services: Google's Visual Positioning Service (VPS), which localizes a phone by matching its camera view against pre-mapped street imagery, and the ARCore and ARKit motion-tracking frameworks. They belong in this comparison because they are the only widely deployed systems that position a phone from what its camera sees, which is exactly NavSight's premise. Their limitation is architectural: all three depend on cloud services and pre-mapped imagery databases, so they stop working as soon as the data connection does. It is worth separating two failure causes that are often confused, because they are independent: GNSS denial, jamming, and spoofing corrupt the *satellite* signal, while a dead SIM or an unreachable server removes the *network*. A rider can suffer either one alone. NavSight is built to keep working under both at once.

Mainstream navigation apps assume both are healthy. The nuance worth stating is that they do not all fail identically: Google Maps carries its own dead-reckoning fallback, so when the satellite fix disappears it keeps advancing an estimated position from the phone's motion

sensors, and it continues to show a plausible location in places where Waze simply stalls. That estimate is still inertial-only, so it degrades with time and distance rather than being corrected by anything external, which is precisely the gap NavSight closes. Micromobility apps (Lime, Bird) assume a healthy satellite fix outright. Academic SLAM systems are self-contained but sized for robots and research rigs, not for a phone sharing its processor with a camera pipeline and a UI. Mapping the landscape on two axes (reliance on external signals versus optimization for urban riders), the quadrant that is simultaneously *self-contained* and *rider-oriented* is empty:

Table 1.1: The navigation landscape on two axes, external dependence vs. rider focus.

| Solution                                               | Needs external signals / cloud                                            | Optimized for micromobility riders |
|--------------------------------------------------------|---------------------------------------------------------------------------|------------------------------------|
| Google Maps, Apple Maps, Waze                          | Yes (GNSS + connectivity; Google Maps dead-reckons briefly without a fix) | Partly                             |
| Lime, Bird, Tier apps                                  | Yes (GNSS)                                                                | Yes                                |
| Google Visual Positioning Service (VPS), ARCore, ARKit | Yes (cloud + pre-mapped imagery)                                          | No                                 |
| Academic SLAM (ORB-SLAM, ...)                          | No                                                                        | No (robots / research rigs)        |
| <b>NavSight</b>                                        | <b>No</b>                                                                 | <b>Yes</b>                         |

Figure 1.2 plots that landscape. NavSight targets exactly the empty quadrant: a fully self-contained visual navigation system for urban micromobility, running entirely on the rider’s phone.

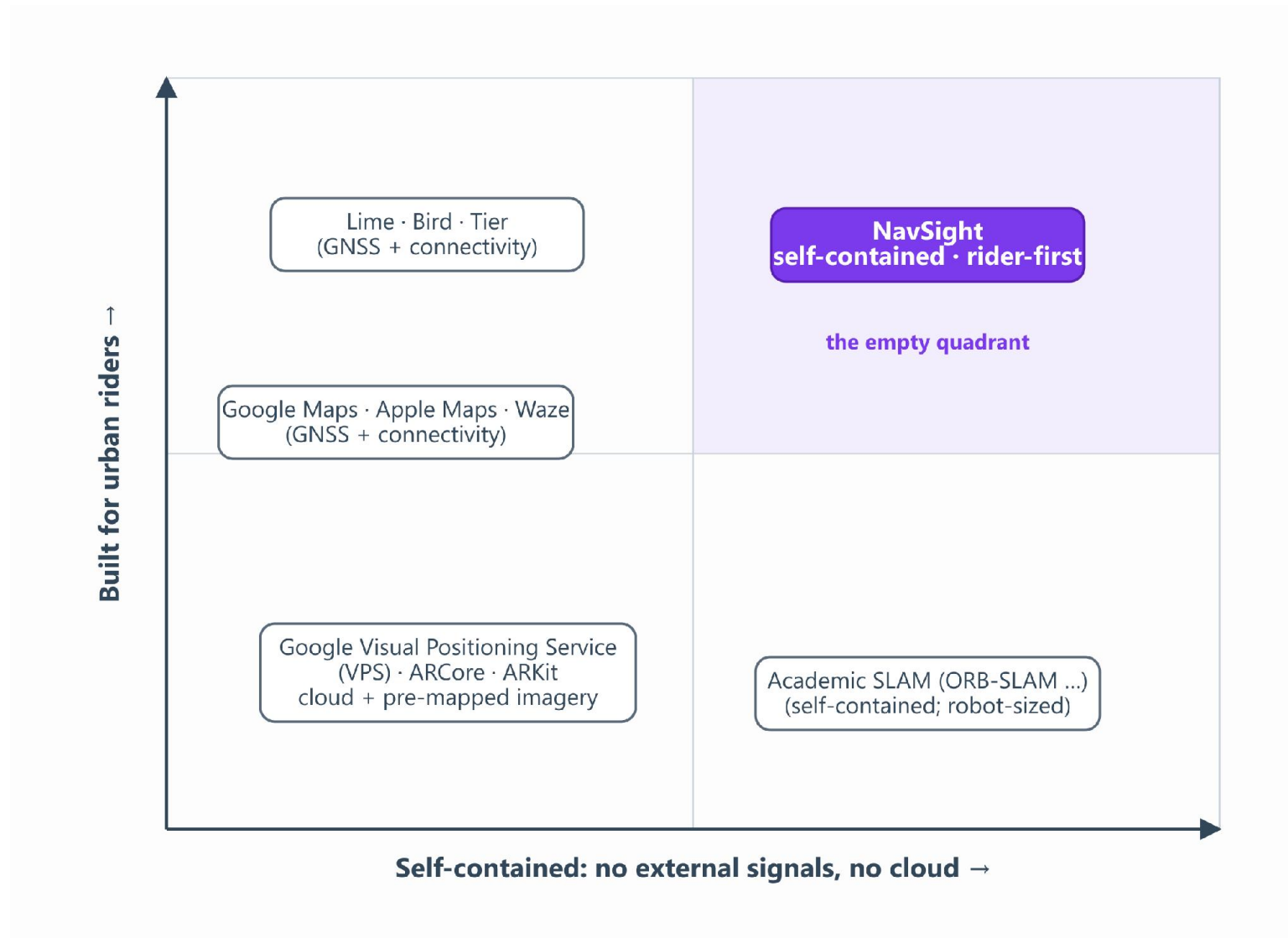


Figure 1.2: The competitive landscape. Mainstream navigation and micromobility apps depend on GNSS and connectivity; visual positioning services depend on the cloud; academic SLAM is self-contained but not built for riders. The self-contained, rider-oriented quadrant is where NavSight sits.

## 1.3 Goals

The project set out to:

- **Develop a hybrid visual–inertial navigation system** combining visual odometry and dead reckoning for real-time position tracking without GNSS. Neither source suffices alone (vision fails in blur and darkness, inertial data drifts within seconds) but their failure modes are complementary.
- **Maintain continuous localization through visual degradation.** Motion blur, poor lighting, and partial obstruction are the *normal* operating regime of a handlebar-mounted camera, not edge cases; the system must bridge them automatically and honestly report its confidence.
- **Operate fully offline:** no satellite signal ever, and no network round-trip for any navigation-critical function, tracking, matching, destination search, and routing all run against on-device data. Road geometry for a *never-visited* area is fetched once and cached (§3.5); previously visited areas work with the radio dead.
- **Achieve real-time performance on a consumer smartphone,** with a processing budget of under 200 ms per frame; a budget set in the SRS and carried through every design decision.
- **Demonstrate practical feasibility for micromobility** on real urban routes, validated against jamming-immune and health-screened references rather than against the very GNSS the system is designed to replace.

## 1.4 Overview of the NavSight Approach

NavSight is a native Android application with a C++ computation core. The description of the pipeline, end to end is as follows:

1. **Capture.** The rear camera (640×480 working resolution; configured at 30 fps, with the analysis pipeline sustaining ~23 fps) and the IMU (accelerometer + gyroscope) stream into the native core over a JNI bridge.
2. **Track.** A visual front-end tracks image features frame-to-frame (Shi-Tomasi corners under pyramidal KLT optical flow), while IMU samples are preintegrated between frames.
3. **Fuse.** An error-state Extended Kalman Filter fuses the two streams, maintaining position, velocity, orientation, and sensor biases with honest uncertainty; heading is maintained by a gyro-primary Madgwick AHRS, aligned to the compass at startup, then

corrected only through a weak, disturbance-gated complementary term that a magnetically noisy street cannot hijack (§7.1.2).

4. **Measure speed metrically.** The road is a plane at a calibrated height below the camera; de-rotated optical flow over that plane yields meters-per-second directly. A two-sided vote/zero-witness scheme reads a true standstill as exactly 0 km/h, and a bounded inertial bridge carries speed through visual dropouts.
5. **Constrain to the map.** A hidden-Markov map matcher, decoded with the Viterbi algorithm, constrains the displayed position to a locally held OpenStreetMap road graph (destination search uses an index bundled in the APK, and road geometry around the rider is cached on the device) so the position the rider sees is always on a road, advanced by the estimated speed and steered through junctions by gyro-relative heading.
6. **Present.** A Jetpack Compose UI renders the map, the matched position, live speed, distance, and tracking-health feedback designed to be read in a glance from a handlebar mount.

Everything above runs on the device, and none of it depends on a signal from outside it. Two ambient inputs are used opportunistically, never structurally: the magnetometer aligns heading (a startup snap plus the weak gated correction of step 3), and a single bootstrap GPS fix, when the receiver has one at all, anchors the session's local frame to world coordinates so the map is drawn in the right place. GPS is never fed to the positioning solution, and the system tracks identically without it (§8.2). Chapter 5 develops this pipeline into a full architecture; Chapter 7 gives each algorithm with its actual constants.

## 1.5 Usage Scenarios

### Scenario 1, Commuting by e-scooter under jamming.

- **Background.** Adam commutes daily by e-scooter across Haifa. Since the start of regional GNSS jamming, his navigation app freezes mid-ride or teleports his position across the map; twice he has missed a turn watching a stale screen.
- **Objective.** Follow his position on the map continuously for the whole ride, regardless of GNSS health.
- **Process overview.** Adam mounts the phone on the handlebar, opens NavSight, and holds still for a few seconds while the app calibrates. The status chip turns green ("Ready"); he starts the session and rides. The purple position ball follows the road as he moves, taking the correct branch at junctions and tracking his speed. At a stop light the speed reads exactly 0 km/h. At his destination he stops the session and reviews distance and duration.

- **Expected outcome.** A continuous, on-road trace of the entire ride with distance accuracy in the 90%+ range, without a single usable GPS fix.

### Scenario 2, Riding through visual degradation.

- **Background.** Noa rides at dusk through a tree-covered street; the camera image is dark and motion-blurred in stretches.
- **Objective.** Keep tracking through short visual dropouts without user intervention.
- **Process overview.** When the visual speed measurement loses confidence, NavSight switches automatically to its inertial bridge: speed is propagated from the accelerometer within a bounded trust window, and the position ball continues advancing along the matched road. When image quality recovers, visual measurement resumes seamlessly. The status indicator reflects the degraded mode throughout.
- **Expected outcome.** No frozen position and no teleport, a bounded, honest estimate that recovers automatically.

### Scenario 3, Finding a destination with no connectivity.

- **Background.** Sara is visiting Haifa; her prepaid data ran out, and GPS has been jammed for a week. She needs to reach a pharmacy across the neighborhood.
- **Objective.** Search for the destination and follow a route with the radio effectively dead.
- **Process overview.** Sara opens NavSight and types the pharmacy's name into the "Where to?" bar. The offline geocoder, an Israel-wide place index bundled inside the app, returns the match instantly; the router computes a route over the locally cached road graph, and the route polyline appears on the map. She rides; the matched ball tracks her progress along the route.
- **Expected outcome.** Search, routing, and live positioning, all without a network round-trip or a satellite fix.

## 1.6 Audience

- **End users:** e-scooter riders, cyclists, and pedestrians in dense urban environments where GNSS is unreliable. No technical expertise is assumed; the interface is minimal and readable outdoors, and every safety-relevant state is color-coded rather than textual.
- **Academic stakeholders:** Shenkar College of Engineering and the project supervisors, who guided and evaluated the work as a Software Engineering B.Sc. final project.
- **Industry:** developers of micromobility platforms, robotics, and AR navigation, for whom NavSight demonstrates that GNSS-independent positioning is feasible on unmodified consumer hardware, no LiDAR, no stereo rig, no RTK base station.

## 2 Literature Survey

This chapter surveys the research landscape NavSight builds on: why GNSS-free positioning is hard (§2.1), how the literature attacks each difficulty and which of those attacks NavSight adopts (§2.2), and where NavSight sits in that landscape (§2.3). The survey is deliberately scoped to techniques that are candidates for a *consumer phone in real time*; each subsection closes with the position NavSight took and why.

### 2.1 Problem Survey

**Inertial drift.** In principle, integrating accelerometer readings twice yields position. In practice, consumer MEMS inertial sensors carry bias and noise that this double integration amplifies quadratically with time: a modest  $0.3 \text{ m/s}^2$  accelerometer bias, typical for a phone, becomes  $1.8 \text{ m/s}$  of velocity error after six seconds and tens of meters of position error within tens of seconds (by a full minute, the same bias alone integrates to hundreds of meters). Gyroscopes fare better but still drift in yaw, the one axis gravity cannot observe. The literature is unambiguous: pure inertial dead reckoning on consumer hardware is only useful as a *bridge* between other measurements, never as the backbone [8]. Figure 2.1 shows the sensor in question: the entire inertial instrument of a modern phone is a few-millimeter MEMS die, whose price in bias and noise is exactly the drift above.

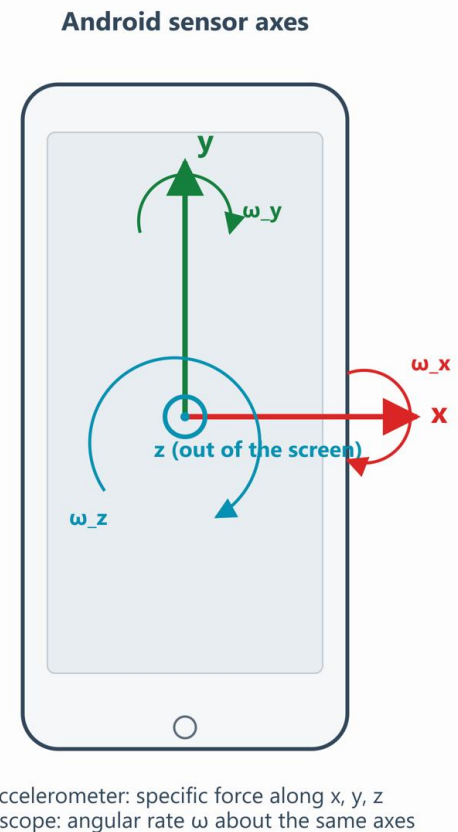
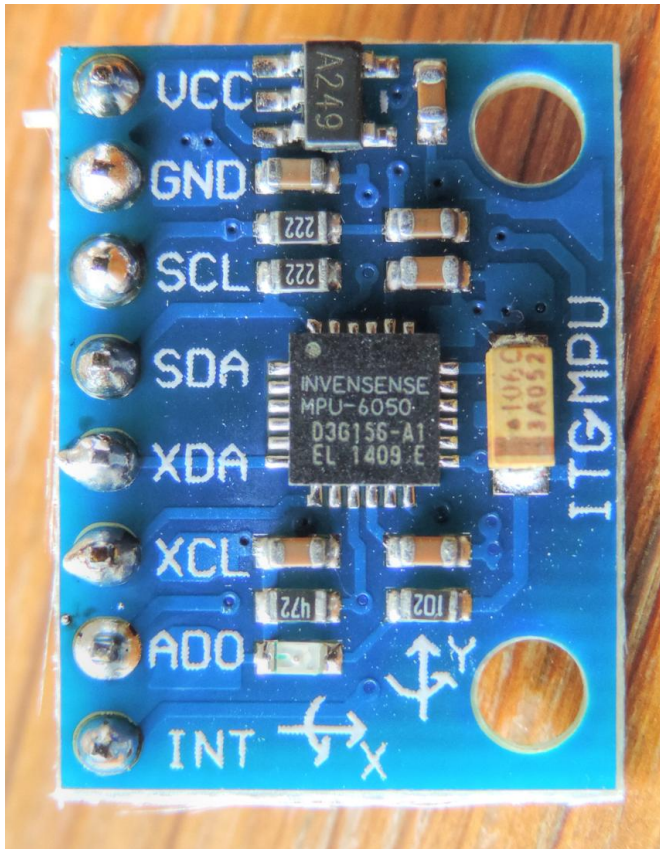


Figure 2.1: The smartphone IMU. Left: a MEMS 6-axis inertial sensor (InvenSense MPU-6050 on a breakout board; photo: Nevit Dilmen, Wikimedia Commons, CC BY-SA 3.0). Right: the Android sensor axes on the device body, accelerometer and gyroscope measure specific force and angular rate about  $x, y, z$ .

**Monocular scale ambiguity.** A single camera observes motion only up to an unknown scale factor: the image stream of a slow ride past nearby objects and a fast ride past distant ones can be geometrically identical [1]. The epipolar geometry between two frames yields the direction of translation but not its magnitude. Every monocular navigation system therefore needs an independent metric anchor: a stereo baseline, known object sizes, IMU excitation observed through a filter, a learned depth prior, or scene structure assumptions such as a ground plane at known height.

**Visual degradation.** Urban riding delivers motion blur (exposure time  $\times$  angular rate), abrupt exposure transitions (sun  $\leftrightarrow$  shadow  $\leftrightarrow$  tunnel), low light, repetitive or feature-poor texture (fresh asphalt), and dynamic occluders (vehicles, pedestrians) that violate the static-scene assumption underlying visual odometry. Feature trackers fail exactly when the rider moves fastest. The consequence for system design is that the *fusion architecture*, not the tracker alone, must carry the estimate through dropouts [1].

**Unbounded error growth without an external anchor.** Odometry of any kind (visual, inertial, or fused) accumulates error with distance traveled; the estimate is a sum of small relative motions, each slightly wrong. Without some externally referenced correction, drift grows without bound. The classical answer is loop closure: recognize a previously visited place

and snap the accumulated error to zero. The practical answer for road vehicles, exploited by every turn-by-turn navigator since the 1990s, is subtler and cheaper: the vehicle is *on a road*, and the road network is known [4].

## 2.2 Solution Survey

**Visual-inertial odometry: filtering vs. optimization.** Two families dominate the literature. *Filter-based* systems, rooted in the MSCKF of Mourikis & Roumeliotis [2] and carried forward by OpenVINS [6], maintain a compact probabilistic state (pose, velocity, biases) and update it with feature-track constraints across a sliding window of camera poses. The MSCKF's key insight is that features need not enter the state at all: when a track ends, its constraint on the *camera poses that observed it* can be applied directly, with the feature's own position projected out via a null-space operation. Cost per frame is bounded and predictable. *Optimization-based* systems such as VINS-Mono [7] instead solve a nonlinear least-squares problem over a keyframe graph, re-linearizing repeatedly; they buy accuracy at the price of a heavier and less predictable compute profile, usually justified on robots with dedicated compute. *Our approach:* a filter, an error-state EKF with MSCKF-style feature updates, because a phone sharing its CPU with camera capture, depth inference, and UI rendering needs bounded, predictable per-frame cost. Chapter 8 shows the entire tracking pipeline running at a median 15.4 ms per frame, which no optimization-based backend on this hardware could have promised.

**Full SLAM systems.** Simultaneous Localization and Mapping systems such as ORB-SLAM [9] additionally build and maintain a persistent landmark map, enabling relocalization against it and loop closure over it. The cost is significant: a bag of-words vocabulary for place recognition (tens of megabytes resident), landmark bookkeeping, and periodic bundle adjustment. On the benefit side, loop closure bounds drift, *when trajectories revisit places*, which a commuting rider's one-way trip often never does. *Our approach:* NavSight deliberately does **not** ship SLAM-style mapping; the SRS scoped it out from the start, and the project's own measurements (Chapter 8, §8.7) showed that for a road vehicle, the map-matching anchor supplies the drift correction loop closure would otherwise provide, at a fraction of the memory and compute. A lightweight ORB-based relocalization of the *visual front-end* (not of a persistent map) is retained for recovery after tracking loss.

**Orientation estimation.** Madgwick's gradient-descent AHRS [3] delivers stable orientation from gyroscope + accelerometer at negligible cost, a single gradient-descent step per sample against the gravity direction, and its gyro-primary character makes it robust to the magnetic disturbances that plague urban environments (vehicles, tram rails, power infrastructure), which corrupt any magnetometer-trusting filter precisely where NavSight operates. *Our*

*approach*: Madgwick AHRS as the heading backbone, gyro-primary by construction: the compass snaps the initial alignment and afterwards contributes only a weak, disturbance-gated complementary correction ( $\tau \approx 5$  s on a clean field, rejected outright past  $35^\circ$  of gyro disagreement), with the road tangent used as a further small, bounded correction when confidently matched to a straight road (Chapter 7, §7.1.2). The EKF maintains its own orientation for fusion purposes, but the *user-facing* heading is deliberately the Madgwick output.

**Map matching.** Newson & Krumm’s hidden-Markov formulation [4] is the standard for snapping noisy trajectories to a road network, developed originally for GPS traces and robust to exactly the noise patterns that break naive nearest-road snapping. Road candidates near each measurement are scored by an emission probability (perpendicular distance under a Gaussian); consecutive candidates are scored by a transition probability comparing great-circle distance to route distance along the graph; the Viterbi algorithm then decodes the jointly most probable road sequence, so a momentary excursion toward a parallel street is outvoted by the trajectory’s history. *Our approach*: an on-device implementation of exactly this model, extended in three rider-specific ways (road-following (“rail”) bonuses that stabilize the matched way at junctions, gyro-steered branch decisions, and deliberately conservative wrong-road behavior (wait for a confident re-match rather than jump)) run entirely against locally held OSM road data (Chapter 7, §7.1.6). Where Newson & Krumm matched recorded traces offline, NavSight matches a live trajectory every  $\sim 500$  ms on the phone.

**Monocular depth estimation.** MiDaS [5] established robust cross-dataset monocular depth prediction: a network trained on a mixture of datasets with a scale-and-shift-invariant loss, transferring zero-shot to unseen scenes, at sizes that run on mobile accelerators. Successors such as the Depth-Anything family improve accuracy and offer metric variants at materially higher compute cost. *Our approach*: MiDaS small on TensorFlow Lite, used to weight optical flow by relative depth in the speed channel that drives the raw trajectory; the newer, heavier models were benchmarked on the target device and rejected on measured latency; the full trade study, with numbers, is §3.4.

**Learned inertial odometry.** A newer research direction regresses velocity directly from raw IMU streams with a neural network (e.g., RNIN-VIO-style models), sidestepping vision entirely; published results are strongest for pedestrian motion, where gait periodicity gives the network structure to exploit. *Our approach*: surveyed but not adopted. Available training data is pedestrian-centric while NavSight’s primary mode is riding; the engineering risk did not fit the project window; and the geometric ground-plane speed measurement met the accuracy targets without a learned component. It remains on the future-work list (§9.3) for exactly the pedestrian modes where the literature is strongest.

Table 2.1 places the surveyed VIO approaches side by side on the axes that mattered for a phone, with NavSight's choice in the final row:

Table 2.1: Surveyed VIO systems compared on the axes that matter on a phone.

| System / family | State size                        | Persistent map          | Loop closure              | Compute profile                    | Resident memory         | Suited to a phone?             |
|-----------------|-----------------------------------|-------------------------|---------------------------|------------------------------------|-------------------------|--------------------------------|
| MSCKF [2]       | Compact (poses + biases)          | No                      | No                        | Bounded, predictable               | Low                     | Yes, the basis NavSight adopts |
| OpenVINS [6]    | Compact (filter + sliding window) | Optional                | Optional                  | Bounded                            | Low–medium              | Yes (research grade)           |
| VINS-Mono [7]   | Keyframe graph                    | Yes                     | Yes                       | Heavy, variable (re-linearization) | Medium                  | Marginal on shared CPU         |
| ORB-SLAM [9]    | Landmark map + graph              | Yes                     | Yes                       | Heavy + BA                         | High (vocabulary + map) | No, memory/compute too high    |
| NavSight        | Error-state EKF + MSCKF updates   | No (road graph instead) | No (map matching instead) | Bounded (15.4 ms median)           | Low                     | Purpose built for it           |

The pattern is consistent: capability in the research literature is bought with state size, memory, and compute, and the phone can afford none of the three in quantity. NavSight's response was not to shrink a heavy system but to replace its two most expensive features, the persistent map and loop closure, with a cheaper external structure the road environment already provides: the road graph.

## 2.3 Discussion and Conclusions

The literature offers a spectrum from lightweight (AHRS + dead reckoning) to heavy (full SLAM with loop closure), with a large body of work at each point. NavSight's position is deliberate and, in each dimension, argued from the phone's constraints:

- a **filter-based VIO core** sized to a hard real-time budget, rather than an optimization backend sized to a robot;

- a **geometric metric-scale anchor**, the road plane at a calibrated height, rather than stereo hardware the phone doesn't have or a learned scale the hardware couldn't run in budget;
- a **road-graph constraint** on the displayed position as the drift anchor, rather than loop closure that assumes revisits and charges tens of megabytes for the privilege.

The combination is mutually reinforcing. The EKF and the ground-plane estimator produce locally accurate motion; the HMM matcher converts “locally accurate” into “globally on the road”; the locally held OSM graph makes the whole loop independent of any signal or server; and each component's failure is covered by another's strength, vision dropouts by the inertial bridge, inertial drift by visual measurement, both by the road constraint. Chapter 8 shows this stack holding 91–93% distance accuracy in an environment where GNSS itself failed by +33%, evidence that for GNSS-denied road navigation, a well-fused lightweight stack beats both of its heavier alternatives on the hardware people actually carry.

## 3 Technological Survey

Where Chapter 2 surveyed *methods*, this chapter surveys *technology choices*, and, in the spirit of an honest engineering record, documents not only what was chosen but what was tried, measured, and abandoned along the way. Several of the project's most consequential decisions were reversals of its own earlier choices; each is presented with the evidence that forced it.

### 3.1 Application Platform: React Native → Native Android (Kotlin + Jetpack Compose)

The original proposal planned a cross-platform React Native application for Android and iOS, with OpenCV bridged in as a native module, an attractive plan on paper, promising two platforms for one codebase. It was revised early, for three reasons that compounded.

First, latency architecture. The per-frame path, camera capture → native processing → UI update, crosses the JavaScript bridge in React Native, adding serialization and copies precisely where the 200 ms budget is tightest; a navigation display that lags its rider is worse than useless. Second, the computational core was always going to be C++/OpenCV regardless of UI framework; a native Android app reaches it through one thin JNI layer instead of two bridging layers (JS→Java→C++). Third, the validation reality: one reference device (Samsung Galaxy S21 Ultra), one team, one year: the cross-platform premium bought nothing that would ever be exercised.

The delivered app is native Android [16]: Kotlin throughout, a Jetpack Compose declarative UI, CameraX for capture with an ImageAnalysis analyzer feeding frames directly to the bridge, and a ViewModel-centered architecture (§6.5). iOS remains out of scope, as the SRS's non-goals had in fact already stated, the proposal and the SRS disagreed, and the SRS won.

### 3.2 Vision and Math Core: C++17 + OpenCV over JNI

All per-frame computation (feature tracking, filtering, speed estimation) lives in a C++17 core built with the Android NDK against the OpenCV 4.5.3 Android SDK (vendored into the repository, so every checkout builds identically). The Kotlin layer feeds camera frames and IMU batches across a single JNI bridge and receives a compact per-frame state object back; the frame path passes the camera's Y and UV planes as direct ByteBuffers, so image data crosses the boundary without a copy.

Two properties motivated the split beyond raw speed. The hot loop stays free of JVM allocation and garbage-collection pauses, an occasional 20 ms GC pause is invisible in an app but catastrophic in a 43 ms frame budget. And the same core, compiled unchanged for desktop, runs inside the replay harness (§3.7), the property that made deterministic offline validation

possible. A JVM-side engine would have required either duplicating the algorithms or dragging Android onto the desktop.

### 3.3 Fusion Filter: Complementary Filter → Error-State EKF with MSCKF Updates

The SRS conservatively specified “a weighted model or complementary filter” for sensor fusion; the simplest thing that could work. The implementation went substantially further: an error-state Extended Kalman Filter over a 15-DOF core motion state (orientation, gyroscope bias, velocity, accelerometer bias, position), later augmented with camera–IMU time-offset and camera-mount rotation calibration states, with IMU preintegration between camera frames and MSCKF-style feature-track updates.

The upgrade was not academic ambition; each step paid for a measured problem. A complementary filter has no concept of uncertainty, so it cannot *reject* a bad measurement, and urban visual data serves bad measurements constantly; the EKF’s  $\chi^2$  gating is what keeps a burst of outlier tracks from corrupting the state. The bias states are what let the filter survive the cheap MEMS sensors of §2.1. The time-offset and mount-rotation states self-calibrate the two constants that field experience showed drifting between rides (mount flex, OS-level timestamp skew). And the filter’s covariance is not bookkeeping; it drives the user-facing tracking- health chip and uncertainty ring, and the  $\chi^2$  gates of every update.

### 3.4 Monocular Depth: MiDaS Retained After Benchmarking Depth-Anything

The depth model weights optical flow by relative scene depth in the speed channel that advances the raw trajectory. MiDaS v2.1 small [5], running on TensorFlow Lite [15], was the incumbent at roughly 30 ms per inference on the target device, throttled to 1 Hz by design.

The newer Depth-Anything family promised *metric* depth, which would have eliminated part of the scale-calibration machinery outright, removed the dependency on calibration warm-up, and simplified the speed pipeline. The promise was worth a proper benchmark on the target SoC (Exynos 2100 / Mali-G78), and the benchmark was decisive in the wrong direction: Depth-Anything V2 small, quantized to INT8, measured **722 ms per inference on CPU and 1,907 ms via NNAPI**, against an inference budget of ~100 ms. Both execution paths were an order of magnitude out; the NNAPI result, slower than CPU, is a reminder that mobile accelerator support for transformer workloads cannot be assumed from spec sheets. The swap was rejected; MiDaS stayed, with an affine fit in disparity space recovering the metric relationship the heavier model would have provided natively. The benchmark, its numbers, and explicit re-open conditions (a distilled mobile variant, or better NNAPI transformer support) are recorded in the project’s study logs and revisited in §9.3.

### 3.5 Maps and Routing: Google APIs → OpenStreetMap Held Locally

The first working map layer used Google's Roads, Directions, and Places APIs, the fastest path to a demo, and a dead end for the product. Three forces pushed against it: every call needs connectivity, which contradicts the offline requirement and is unavailable in exactly the jammed scenarios NavSight targets; rate limits and billing constrain field testing (a day of validation rides is thousands of calls); and an external service is a moving dependency that can change under a finished project.

In May 2026 the map layer was migrated wholesale to OpenStreetMap data held on the device, in two parts with different offline profiles. **Destination search became fully offline:** a build script (`scripts/build_haifa_assets.py`) compiles an Israel-wide OSM place index (~254,000 places with an inverted token index, 16.6 MB packed binary) that ships inside the APK. **Road geometry loads on demand** around the rider's location: fetched once from the OpenStreetMap Overpass service [14], parsed into an immutable in-memory region (segment R-tree, routing graph, roundabout index), and kept in a bounded LRU disk cache with atomic writes, so areas the rider has visited before are served entirely from local storage, and the app boots straight into the last cached region. Positioning never touches the network; local routing runs on-device (Dijkstra) over the cached graph, with an online router available for far-away destinations when connectivity happens to exist. The legacy Google code paths were not deleted but commented out in place, with the migration rationale recorded beside them, the project's standing convention for reversals.

### 3.6 Map Matching: OSRM `/match` → On-Device Viterbi (*LocalMatcher*)

Map matching likewise began as a network service: the OSRM `/match` endpoint, which implements the same Newson–Krumm model server-side. It shared the fate of the Google APIs, with two aggravations of its own: a rate limit that throttled the ~500 ms matcher tick, and a coordinate-count cap that fought the sliding-window design (the matcher wants to resubmit a trailing window every tick, exactly the access pattern a public endpoint penalizes).

It was replaced by *LocalMatcher*, an on-device implementation of the HMM/Viterbi model [4] running directly against the locally held road graph: no network, no rate limits, no payload caps, purely local queries against the packed segment R-tree, and the freedom to extend the model with the rail bonuses, gyro-steered junction decisions, and bounded recovery of §7.1.6, extensions impossible against a fixed public API. The OSRM client remains in the codebase for reference but has no call sites in the shipped app.

### 3.7 Test Tooling: Deterministic Replay Harness and CI Scoring

The single most consequential tooling decision was building a deterministic replay harness: recorded rides (camera frames plus IMU and GPS telemetry, captured by the app's built-in session recorder) are re-run through the *unmodified* native engine compiled for desktop (MSYS2/MinGW). Because the engine is identical to the byte, any two algorithm versions can be compared A/B on identical input, converting “does this change help?” from a ride-to-ride guess into a controlled experiment.

The harness earned its keep repeatedly: the +24% turn-distance contribution of the inertial bridge was isolated this way (Chapter 8, Figure 8.6); a mid-project heading regression was root-caused by *bisection over engine versions on one recorded walk* rather than by tuning; and every candidate fix in §8.7 was validated on recorded data before a wheel turned. Around the harness sit 74 Kotlin unit tests, a C++ unit suite (GoogleTest), and a CI workflow ([.github/workflows/replay.yml](#)) that scores every change against recorded fixtures, a regression net for algorithms, not just for code.

## 3.8 Summary of the Chosen Stack

Table 3.1: Technology decisions and the rejected alternatives, with the measured reason each alternative lost.

| Layer      | Choice                                                                              | Rejected alternative (reason)                                                |
|------------|-------------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| App        | Kotlin + Jetpack Compose, CameraX                                                   | React Native + iOS (bridge latency, single-device reality)                   |
| Core       | C++17 + OpenCV 4.5.3 via JNI/NDK                                                    | JVM-side processing (GC pauses, no replay reuse)                             |
| Fusion     | Error-state EKF + MSCKF updates, self-calibrating $t_d$ and mount rotation          | Complementary filter (no uncertainty, no outlier rejection)                  |
| Depth      | MiDaS v2.1 small on TFLite (-30 ms @ 1 Hz)                                          | Depth-Anything V2 small INT8 (722 ms CPU / 1,907 ms NNAPI vs -100 ms budget) |
| Maps       | OSM held locally (bundled Israel-wide search index + on-device cached road regions) | Google Roads/Directions/Places (online, quotas, moving dependency)           |
| Matching   | On-device Viterbi HMM (LocalMatcher) with rail bonuses                              | OSRM /match (online, rate/coordinate caps, unextendable)                     |
| Validation | Deterministic replay harness + CI scoring + 74 Kotlin tests + C++ suite             | Ride-to-ride comparison only (uncontrolled)                                  |

## 4 Requirements and Specification

This chapter states what NavSight must do (functional requirements, §4.1) and how well (non-functional requirements, §4.2), delimits the project scope (§4.3), and, because a project book should be an honest record, lists explicitly where the delivered system deviates from the signed SRS and why (§4.4). Every requirement carries an ID (FR-*x.y* / NFR-*x.y*) and a priority (**Must** / **Should** / **Could**); Chapter 8 (§8.6) traces each ID to the test case(s) that validate it. Requirements marked (*as-built*) formalize capabilities added beyond the signed SRS; they are requirements the delivered system actually meets, stated so that the validation chapter can audit them.

### 4.1 Functional Requirements

#### FR-1 Camera-Based Motion Tracking

The visual half of the odometry: turning the camera stream into frame-to-frame motion evidence.

- **FR-1.1 (M)** The system shall continuously capture video frames from the device's rear camera during an active session.
- **FR-1.2 (M)** The system shall detect and track visual features between consecutive frames.
- **FR-1.3 (M)** The system shall estimate the relative displacement (translation and rotation) of the device from tracked feature correspondences.
- **FR-1.4 (M)** The system shall reject unreliable frames (excessive blur) and outlier correspondences (geometric verification) before they influence the estimate.

*Example:* riding down Savyon St., the tracker maintains a few hundred road- and façade-texture points; between two consecutive frames the inlier correspondences imply a 0.47 m forward translation and a 1.2° right yaw, which pass geometric verification and enter fusion.

#### FR-2 Inertial Dead Reckoning

The inertial half: what keeps the estimate alive when the camera cannot.

- **FR-2.1 (M)** The system shall read the smartphone's accelerometer and gyroscope continuously during an active session.
- **FR-2.2 (M)** The system shall maintain motion estimates from inertial data alone when visual tracking is unavailable or degraded, for a bounded bridging interval.

- **FR-2.3 (M)** The system shall fuse visual and inertial measurements into a single continuous position estimate.

*Example:* entering a tree-shadowed stretch at dusk, visual speed votes vanish for 3.5 s; the inertial bridge propagates speed from body-forward acceleration and the displayed position keeps advancing along the road, flagged as bridged, until votes return.

### FR-3 Sensor Fusion

- **FR-3.1 (M)** The system shall combine visual and inertial data in a probabilistic filter that weighs each measurement by its uncertainty and rejects statistical outliers.
- **FR-3.2 (M)** The fusion state shall be updated at a minimum frequency of 30 Hz (satisfied by IMU-rate propagation with per-frame visual corrections).
- **FR-3.3 (S)** The filter shall expose its own uncertainty (position covariance, tracking quality) to the application layer for user-facing health feedback.

### FR-4 Path Visualization and Session Control

- **FR-4.1 (M)** The system shall render the user's position and path on a 2-D map view that updates dynamically as new estimates are computed.
- **FR-4.2 (M)** The system shall let the user start, pause/reset, and terminate a tracking session.
- **FR-4.3 (M)** The system shall display live session metrics: current speed, distance traveled, and tracking-health status.
- **FR-4.4 (S)** The display shall be legible outdoors at riding distance, with a night mode for low-light conditions.

### FR-5 Offline Operation

The requirement that defines the product; everything else serves it.

- **FR-5.1 (M)** Position tracking shall function without internet connectivity and without GNSS reception at all times.
- **FR-5.2 (M)** All sensor processing shall execute locally on the device.
- **FR-5.3 (M)** Destination search shall be fully offline (bundled index); map matching and local routing shall be offline for any area whose road data is present in the on-device cache (populated automatically when the area is first visited with connectivity).

### FR-6 Initialization and Calibration

- **FR-6.1 (M)** On first use the system shall request camera and motion-sensor permissions with clear explanations.

- **FR-6.2 (M)** The system shall guide the user through a hold-still calibration of a few seconds and indicate readiness when complete.
- **FR-6.3 (M)** The system shall persist calibration data, including the camera mount height used for metric scale and per-motion-mode scale factors, across sessions.
- **FR-6.4 (S)** The system shall assist mount-height entry with a GPS-derived height suggestion when a healthy fix happens to be available, cross-checked before acceptance.

*Example:* Adam mounts the phone, holds still for four seconds; gravity is estimated, gyro/accel biases settle, the chip walks through its calibration states to green. His handlebar mount height (1.05 m) was stored on first setup and reloads silently.

### FR-7 Error Handling and Feedback

- **FR-7.1 (M)** The system shall detect unreliable visual conditions (low light, motion blur, feature-poor scenes) and reflect them in a visible tracking-health indicator.
- **FR-7.2 (M)** The system shall survive camera interruptions (app backgrounded, screen off) without corrupting the session, and resume cleanly.
- **FR-7.3 (M)** The system shall recover automatically from tracking loss, including re-acquisition of the visual front-end.

### FR-8 Session Management

- **FR-8.1 (M)** The system shall record per-session telemetry locally (pose, sensor, and event-counter streams) sufficient to reproduce the session offline.
- **FR-8.2 (S)** The system shall present a session summary (distance, duration) on completion and support exporting the traveled path.

### FR-9 Map-Matched Positioning (*as-built: see §4.4*)

The drift anchor: the capability that converts locally accurate odometry into a globally correct on-road position.

- **FR-9.1 (M)** The system shall constrain the displayed position to the road network of locally held OpenStreetMap data.
- **FR-9.2 (M)** At junctions, the system shall select the outgoing road using the rider's gyro-relative heading.
- **FR-9.3 (M)** The displayed position shall be constrained to the road graph and shall never jump across the map; any map-derived correction fed back to the underlying trajectory shall be gated on sustained, high-confidence agreement (match confidence  $\geq 0.6$  held for several consecutive ticks).

- **FR-9.4 (S)** The system shall support destination search over a bundled offline geocoding index and route computation over the locally held road graph.
- **FR-9.5 (S)** The system shall handle roundabouts explicitly (entry detection, traversal, exit, and re-entry hysteresis) rather than treating them as generic junctions.

*Example:* approaching a Y-fork at 28 km/h, the accumulated gyro-relative heading since the pre-junction anchor reads +9°; the left branch's tangent differs by 31°, the right's by 6°; the ball takes the right branch. Because the ball rides the graph, even a wrong branch keeps it on a real road rather than teleporting across the map; a matcher-supervised re-acquisition onto the correct road is implemented but conservatively left off by default (§7.1.6), pending an on-device re-record to validate it.

## 4.2 Non-Functional Requirements

### NFR-1 Performance

- **NFR-1.1 (M)** Per-frame visual processing shall complete within 200 ms on the reference device.
- **NFR-1.2 (M)** The system shall sustain the camera's native frame rate without accumulating a frame backlog.
- **NFR-1.3 (M)** Heavy computation shall run in native code (C++) on dedicated threads, never on the UI thread.

### NFR-2 Reliability and Accuracy

- **NFR-2.1 (M)** Accumulated distance error shall not exceed 10% over 200 m under typical urban riding conditions.
- **NFR-2.2 (M)** The system shall recover from short-term visual data loss (occlusion, blur, exposure transients) without user intervention.
- **NFR-2.3 (M)** The system shall resume tracking after a temporary freeze or a background/foreground cycle.
- **NFR-2.4 (S)** A true standstill shall be displayed as exactly 0 km/h, no phantom creep from sensor noise.

### NFR-3 Compatibility

- **NFR-3.1 (M)** The application shall support Android 10+ (API 29+); the as-built minimum is API 24, and the validated configuration is Android 15 on a Samsung Galaxy S21 Ultra (Exynos 2100).
- **NFR-3.2 (M)** The native core shall target ARM64 (arm64-v8a) as the primary ABI.

#### NFR-4 Usability

- **NFR-4.1 (M)** The interface shall be minimal, glanceable, and readable outdoors, with a night mode for low-light riding.
- **NFR-4.2 (M)** Tracking status shall be communicated by color-coded indicators, not by text a rider must read while moving.

#### NFR-5 Maintainability

- **NFR-5.1 (M)** The UI layer (Kotlin) and the processing core (C++) shall be separated behind a single narrow JNI interface.
- **NFR-5.2 (M)** The project shall be maintained under Git version control with a regression suite (unit tests plus deterministic replay scoring) exercised on every change.
- **NFR-5.3 (S)** The engine shall be instrumented (event counters, structured logging) so that field behavior is auditable from recorded telemetry.

#### NFR-6 Security and Privacy

- **NFR-6.1 (M)** All camera and sensor data shall be processed on-device; no visual or sensor data shall ever be uploaded or shared.
- **NFR-6.2 (M)** Session recordings shall be stored locally under the application's storage and be deletable by the user.

### 4.3 Specification: The Scope of the NavSight Project

#### Inside the scope:

- A native Android application performing real-time visual-inertial position tracking on a smartphone.
- Metric speed and travelled-distance estimation from camera + IMU only.
- Map display, map matching, destination search, and routing over locally held OpenStreetMap data (validated region: Haifa).
- Calibration flows (hold-still, mount height) and tracking-health feedback.
- Field validation on real urban routes under GNSS-denied conditions, plus a deterministic offline replay/regression platform.

#### Outside the scope:

- SLAM mapping or 3-D environment reconstruction.
- iOS or cross-platform support.
- Cloud services, backends, or any off-device data processing.

- Turn-by-turn voice guidance and audio navigation.
- External sensor hardware beyond the phone itself.

## 4.4 Deviations from the Signed SRS

The signed SRS (November 2025) predates most of the engineering. The delivered system deviates from it in five places, each deliberate, each an upgrade or a correction rather than a shortfall:

1. **Fusion filter.** The SRS specified “a weighted model or complementary filter” (SRS 3.1.3); the implementation is an error-state EKF with MSCKF-style updates, strictly stronger, chosen for principled uncertainty handling and outlier rejection (§3.3).
2. **Feature tracking method.** The SRS suggested descriptor matching (“e.g., ORB or FAST”); the implementation tracks Shi-Tomasi corners with pyramidal KLT optical flow, which is cheaper per frame and better suited to high-frame-rate tracking. ORB descriptors are used only for relocalization after tracking loss.
3. **Map-matched positioning (FR-9) is an addition.** The SRS promised only a raw path visualization; the delivered system adds map matching over locally held OSM data as the primary drift anchor. The SRS’s open issue 9.4 anticipated “loop detection or periodic re-centering” for drift compensation; road-graph matching is the mechanism that actually shipped, cheaper, and effective on first-visit routes where loop closure by definition cannot fire.
4. **Platform clauses dropped.** The SRS’s mentions of iOS 14+ and of encrypted communication “with any backend” are vestigial: the product is Android-only by its own non-goals, and no backend exists, nothing leaves the device.
5. **Frame-rate wording.** The SRS asked for “at least 30 FPS”; the as-built camera configuration delivers its native ~23 fps at the chosen working resolution, and the real requirement (no backlog, per-frame budget met with an order-of-magnitude margin) is stated as NFR-1.2 and validated in Chapter 8.

# 5 System Architecture

This chapter describes NavSight's structure at the system level: the layered architecture (§5.1), the data that flows through and rests in it (§5.2), the main runtime processes (§5.3), the operational life-cycle (§5.4), and the actors involved (§5.5). Chapter 6 refines this view into concrete components and interactions; Chapter 7 fills in the algorithms.

## 5.1 High-Level System Architecture

NavSight is organized as a **four-tier layered architecture**. Each tier has a single role and communicates only with its neighbors, which keeps the performance-critical native code isolated, the UI responsive, and the whole engine reusable outside the app (the desktop replay harness drives Tiers 3–4 directly).

**Tier 1: Presentation Layer (Kotlin, Jetpack Compose).** Renders the map view with the matched position ball, the live speed header, the path trace, the tracking-health chip, night mode, and the calibration and debug surfaces. The layer is purely reactive: it observes a single UI state exposed by the ViewModel and issues user intents (start, reset, search, calibrate) back as method calls. No composable touches a sensor, a socket, or the JNI boundary.

**Tier 2: Application Logic Layer (Kotlin).** The orchestrator. It manages permissions and lifecycle; configures CameraX (640×480 working resolution at the camera's native ~23 fps) and the IMU listeners (registered at `SENSOR_DELAY_GAME`, nominally ~50 Hz); feeds raw sensor data down to the native core and receives per-frame `VioData` state back. The same tier runs the entire map layer (the road-region manager with its on-device cache, the Viterbi map matcher, the rail-ball display position, routing, and offline geocoding) and folds everything into the one observable UI state Tier 1 renders. The map layer's placement here is deliberate: the VIO core below knows nothing about roads (§5.1's second property).

**Tier 3: Native Interface Layer (JNI Bridge).** A deliberately narrow marshalling layer. It converts camera frames into native images (the Y and UV planes cross as direct byte buffers, zero-copy), batches IMU samples into native queues, keeps heavy processing on a dedicated background VIO thread so the UI thread never blocks on computation, and returns one compact `VioData` snapshot per frame. Everything else it exposes is configuration, calibration persistence, and diagnostics (§6.3).

**Tier 4: Native Processing Core (C++17, OpenCV).** The engine. A visual front-end (Shi-Tomasi detection + pyramidal KLT tracking with geometric verification), IMU preintegration with Madgwick attitude, the error-state EKF with MSCKF feature updates and zero-velocity updates, the ground-plane metric-speed estimator with its inertial bridge, and ORB-based

relocalization for recovery after tracking loss. The core is instrumented throughout with event counters, ~180 atomic counters serialized into every session's telemetry, which is what made the field behavior auditable during validation. Figure 5.1 shows the four tiers and the boundaries between them.

# NavSight - System Architecture

No GPS / no network in the hot path

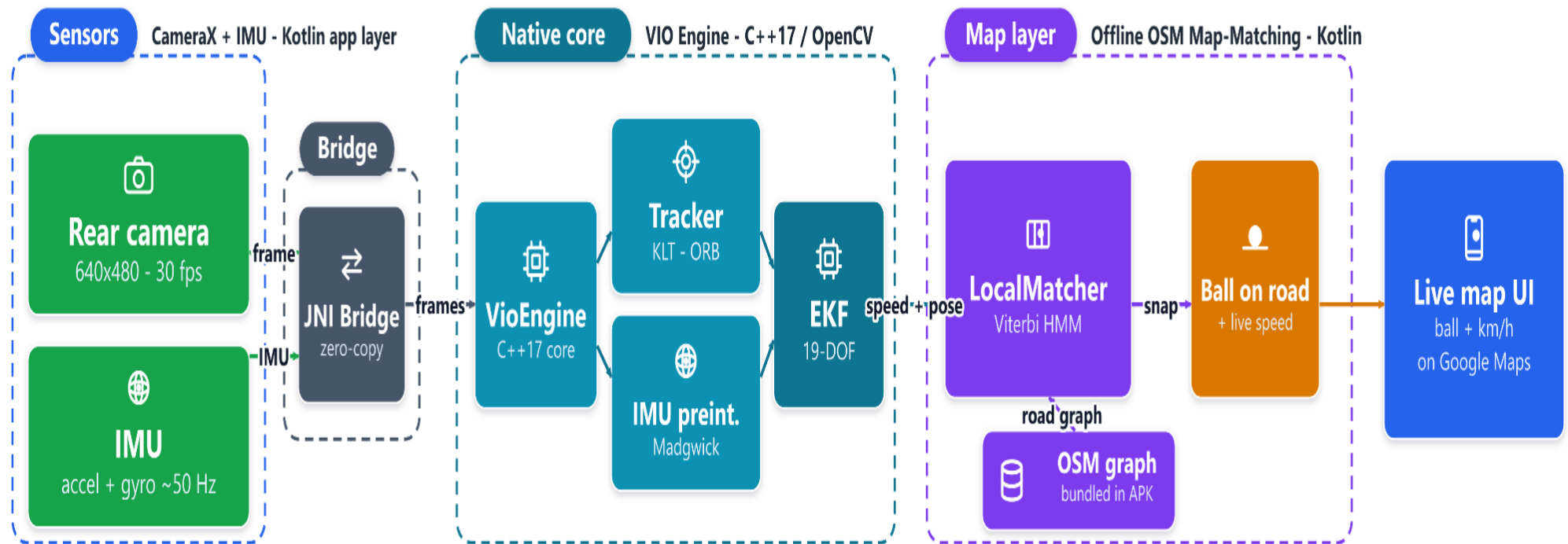


Figure 5.1: The NavSight data path: sensors in the Kotlin app layer cross the JNI bridge into the native C++ VIO engine, whose speed and pose the map layer snaps to the road graph before the live map draws it.

Two properties of this decomposition carried the project:

- **The narrow waist.** Everything crossing Tier 3 is small and explicit: frames and IMU samples down, one state struct back. This is what made the deterministic replay harness possible: the same Tier 4 binary runs on-device and on a desktop, fed the same recorded inputs, producing the same outputs to the byte.
- **The map layer lives above the engine.** The VIO core knows nothing about roads; the matcher consumes the core's raw trajectory ("the dot") and produces the road-constrained display position ("the rail ball"). The two concerns evolve independently; each can be validated in isolation; and when the map layer *does* talk back to the engine, it does so through two explicit, bounded feedback setters (§6.4.3), not through shared state.

## 5.2 Data Design

### Streaming data (transient).

Table 5.1: The streaming data paths and their producers and consumers.

| Stream            | Producer → Consumer        | Content                                                                                   |
|-------------------|----------------------------|-------------------------------------------------------------------------------------------|
| Camera frames     | CameraX → JNI → Tracker    | YUV_420_888 → grayscale, 640×480, timestamped (ns)                                        |
| IMU samples       | SensorManager → JNI queues | accel + gyro triplets, timestamped (ns)                                                   |
| VioData per frame | Core → ViewModel           | global position (m), orientation, speed, tracked points, scale, health counters (§6.3)    |
| Matcher tick      | ViewModel → LocalMatcher   | sliding window of recent raw-trajectory points (-15 sub-sampled from 150)                 |
| Matched state     | LocalMatcher → UI state    | road-constrained position, matched way, confidence                                        |
| Map feedback      | ViewModel → Core           | road-heading hint + gated cross-track position correction (both live and bounded, §7.1.6) |

### Resident data (persistent).

- **Bundled OSM search index (16.6 MB, APK asset).** An Israel-wide geocoding index (~254,000 places) with an inverted token index for offline destination search,

compiled by `scripts/build_haifa_assets.py` into a versioned little-endian binary format:

Table 5.2: The bundled offline-search assets.

| Magic  | File              | Contents                                                                |
|--------|-------------------|-------------------------------------------------------------------------|
| NSGC   | haifa_geocode.bin | places {id, lat, lng, name} +<br>string pool + inverted token<br>index  |
| (none) | haifa_assets.json | manifest: schema version,<br>bounding box, entity<br>counts, build date |

Two further formats, NSSG (packed road segments for the R-tree) and NSGR (routing-graph vertices and edges), are defined and implemented in the same asset pipeline for pre-packing road regions; in the validated build, road geometry arrives through the dynamic path below instead.

- **Road-region cache (on-device, LRU).** Road geometry is fetched once around the rider's location, parsed into an immutable in-memory region (a segment R-tree for nearest-road queries, a routing graph for Dijkstra, and a roundabout index) republished atomically on region change, and persisted in a bounded LRU disk cache with atomic temp-file-rename writes (crash-safe by construction, and OS-evictable without harm). Previously visited areas load without connectivity, and the app boots into the most recently used region.
- **Calibration store.** A schema-versioned store holding the gravity-aligned rotation and gyro/accelerometer biases, plus the camera mount height  $h$  (the metric-scale anchor), per-motion-mode scale factors (per-gait K), and the MiDaS scale factor, all persisted across sessions so a returning rider starts calibrated.
- **Session telemetry.** Per-frame JSON records (pose, IMU, GPS when present, speed channels, event counters) written locally, plus optional recorded camera frames; these recordings are the input to the replay harness and the source of every figure in Chapter 8. Paths can be exported as GPX.

There is no relational database and no server-side state: circular buffers hold transient sensor data in native RAM; preferences and calibration use local storage.

## 5.3 Processes

Three loops run concurrently, at three different rates, meeting only at well-defined hand-offs:

1. **The frame loop (per camera frame: ~23 Hz, a ~43 ms period).** A frame arrives → grayscale conversion and contrast normalization → blur gate → KLT tracks existing features and re-detects to maintain coverage → the IMU preintegrated since the previous frame is consumed → the EKF propagates and applies its updates (essential-matrix pose, MSCKF on lost tracks, zero-velocity when stationary) → ground-plane speed votes are collected (speed, or exact zero, or inertial bridge) → the raw trajectory advances → VioData returns to the app layer. Median measured cost: 15.4 ms (Chapter 8).
2. **The IMU loop (per sensor event: ~50 Hz nominal, a ~20 ms period).** Samples are timestamped and enqueued; preintegration accumulates them on-manifold between frames; the Madgwick AHRS maintains attitude continuously and is the system's heading source.
3. **The matcher tick (~2 Hz, every ~500 ms).** The ViewModel feeds the latest raw trajectory window to the LocalMatcher; candidate segments within 30 m are scored (emission: perpendicular distance under  $\sigma_z = 20$  m; transition: route-length consistency plus same-way and rail bonuses) and Viterbi-decoded; the display ball advances along the matched road by the estimated speed, with junction branches chosen by gyro-relative heading; and, on a confident straight-road match, a bounded road-heading hint flows back down to the engine, joined, once the same confident match has held for a  $\geq 3$ -tick streak, by the gated cross-track position correction to the raw trail (§7.1.6).

## 5.4 The Life-Cycle View of the System

A session moves through seven states, all exercised during validation (Chapter 8, T10). Figure 5.2 shows the state machine as specified at SRS time; the delivered system implements it with one refinement noted below.

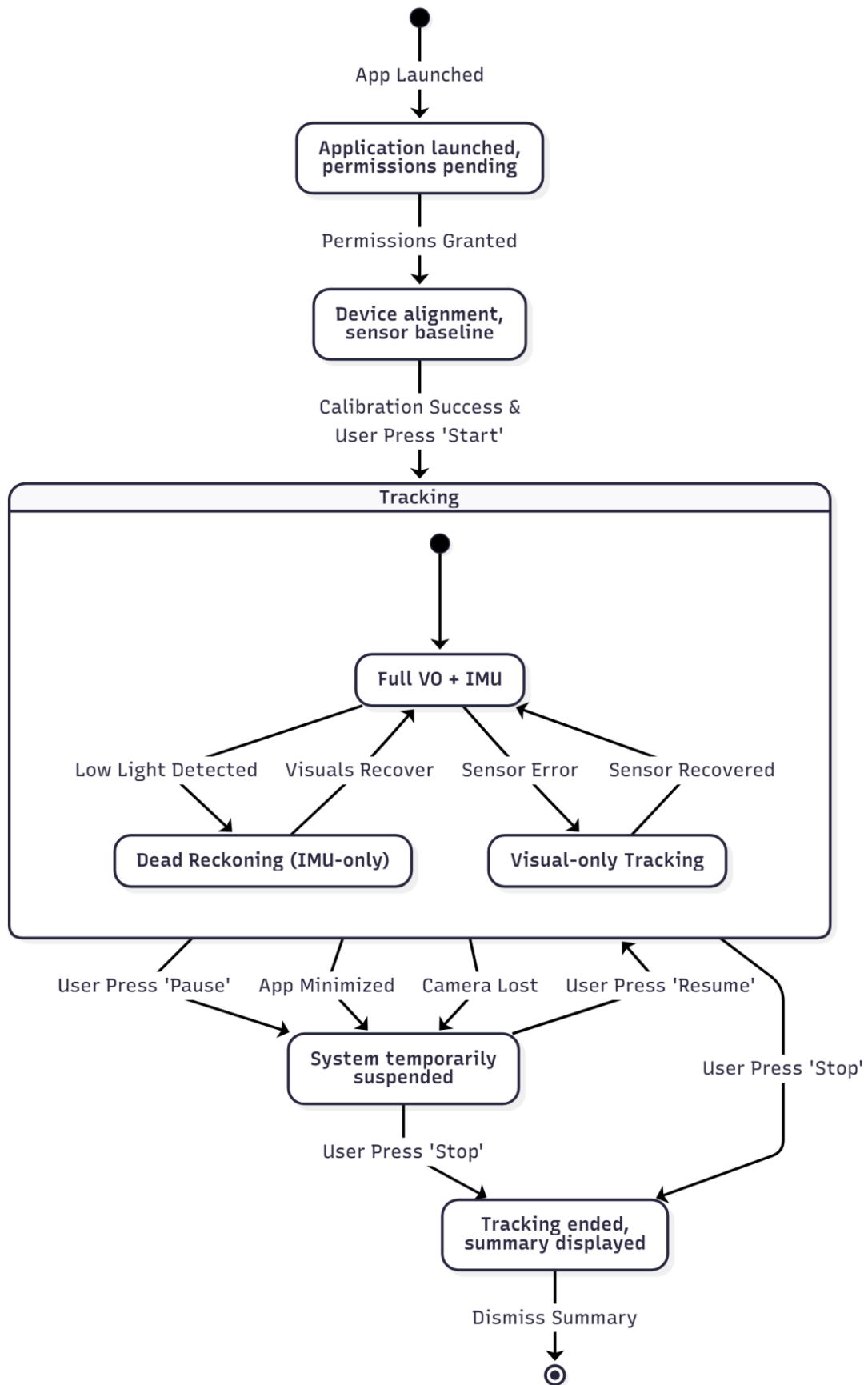


Figure 5.2: The session state machine: launch and permissions, calibration, the tracking composite state with its two degraded sub-modes (inertial-only and visual-only), suspension, and session end.

1. **Launch & permissions.** Camera and sensor permissions are requested with rationale; the native library loads.

2. **Calibrating.** The user holds the device still; gravity is estimated, biases settle, the health chip reports progress.
3. **Ready.** Calibration has converged; the map shows the start position.
4. **Tracking.** The full loop of §5.3 runs; the ball follows the road.
5. **Degraded (bridged).** Visual measurement drops out (blur, darkness); the inertial bridge carries speed within its bounded trust window; the chip reflects the mode. The complementary degradation, IMU anomaly with healthy vision, is absorbed by the filter itself: the  $\chi^2$  gate simply rejects the anomalous inertial evidence, so the state machine's visual-only sub-mode is the filter's natural behavior rather than an explicit switch.
6. **Recovered.** Visual tracking resumes (including ORB relocalization after a hard loss); normal tracking continues.
7. **Stopped / reset.** The session ends; a summary (distance, duration) is shown; telemetry is finalized on disk.

The as-built refinement: what the SRS drew as a user-facing “Paused” state is implemented as suspension-and-clean-resume around OS lifecycle events (app minimized, camera lost) plus an explicit user Reset, the same guarantees, bound to the events Android actually delivers.

## 5.5 Roles

- **Rider (primary actor).** Starts, follows, and ends sessions; performs the hold-still calibration once per mount; searches destinations. Everything else (degradation, bridging, recovery, matching) is automatic, by design: a rider in traffic cannot be a systems operator.
- **Developer / validator (secondary).** Uses the debug panel, the session recorder, and the desktop replay harness; consumes telemetry, event counters, and CI replay scores. This role exists because reproducible validation was a first-class requirement (NFR-5.2, NFR-5.3), not an afterthought, the same build that navigates records the evidence of how well it navigated.

## 5.6 Deployment and Build Configuration

NavSight ships as a single Android application package. The build (Gradle with the Android plugin) targets `compileSdk 34` with `minSdk 24`, and the native core is compiled by CMake through the NDK against the OpenCV 4.5.3 Android SDK, vendored in-repo as a Gradle submodule (`:sdk`) and resolved by CMake's `find_package`, so every checkout builds identically, with `libopencv_java4.so` shipped per ABI alongside the engine. The APK bundles three categories of asset that together make the offline promise concrete: the native `libnavsight.so`

engine (per ABI, ARM64 primary), the MiDaS TFLite model and the ORB vocabulary, and the OSM geocode index under `assets/osm/haifa`.

The debug build type carries two deliberate conveniences that a store build would drop: a `-osm` version-name suffix and an `.osm` application-ID suffix, so the offline-migration build installs *alongside* an earlier build rather than overwriting it, the two appear as separate launcher entries with separate data, which is exactly what a team validating a migration on the same phones needs. The validated build reported throughout this book is **v1.0-osm**. Because the native core is also compiled for desktop (MSYS2/MinGW) from the same sources, “deployment” has two targets, not one: the phone, where the engine navigates, and the desktop replay harness, where the identical engine is held to account (Appendix C).

## 6 System Design

Chapter 5 described NavSight in tiers; this chapter descends to the concrete components, who owns whom, what crosses the JNI boundary, and how the pieces interact frame by frame. The design was documented twice in the project's life: once in the SDD (February 2026), whose diagrams are reproduced here, and once by the delivered code, which refined names and added the map-matching stack. Where the two differ, the prose describes the as-built system and the captions note the mapping.

### 6.1 Use Cases

The system has one primary actor, the **Rider**, whose use cases are: *grant permissions*, *calibrate* (hold-still), *start a session*, *follow position on the map*, *search a destination* (offline geocoder), *request a route*, *reset/stop a session*, and *review the session summary*. A secondary **Developer/Validator** actor uses: *record a session* (frame + telemetry capture), *inspect the debug panel*, and *replay a recording* on the desktop harness. Internal (system-initiated) use cases (*degrade to inertial bridging*, *degrade to visual-only*, *recover tracking*, *re-latch the rail after a maneuver*) extend the tracking flow without user involvement.

Three SDD-era use cases (UC-1 start-and-track, UC-2 degraded-mode fallback, UC-3 session summary) map one-to-one onto the flows above; validation exercised all of them (Chapter 8, T10) plus the map-matching flows the SDD predates (T5, T6). Figure 6.1 draws the actors and flows together.

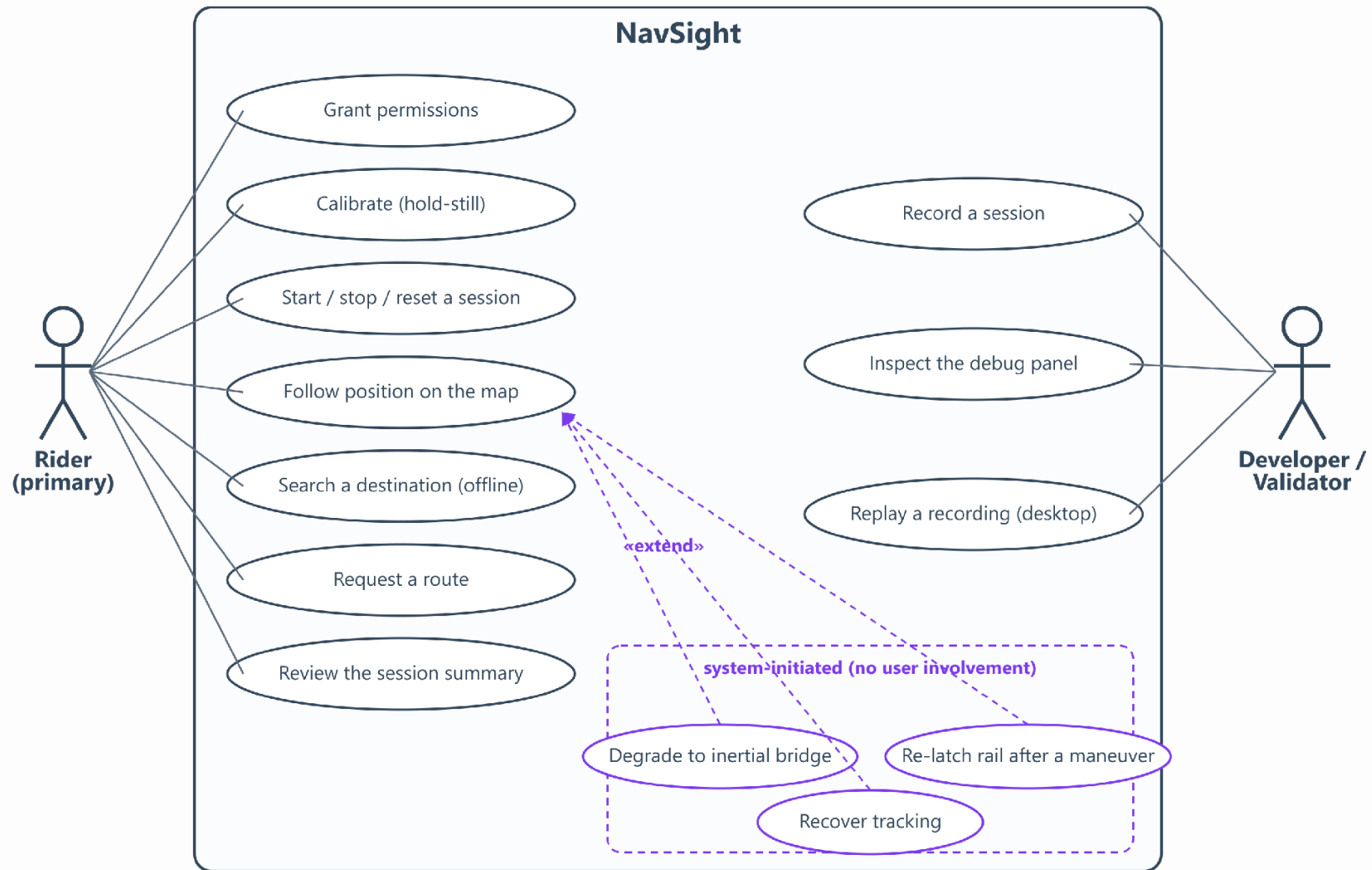


Figure 6.1: Use-case diagram. The Rider's session, calibration, and search flows; the Developer/Validator's recording, debugging, and replay flows; and the system-initiated extensions (degrade, recover, re-latch) of the tracking flow.

## 6.2 Structural Design: Modules and Classes

### 6.2.1 Android Application Layer (Kotlin)

One activity hosts everything: `MainActivity` installs the crash logger, gates on camera + location permissions, and hands the screen to `Compose`. The composition centers on two long-lived objects:

- **NavSightViewModel**, the single state holder. It owns the sensor pipeline (`SensorRepository`), the map-matching stack (`LocalMatcher`, `ManeuverStateMachine`, `GraphRailDot`, with `RoadSnapper` as a last-resort fallback), and ~25 observable UI state fields (VIO state, path history, speed, matched position, navigation state). Its per-frame entry point `handleVioUpdate()` consumes each `VioData`, maintains the path and speed displays and the covariance ring, persists evolving calibration, and ticks the map matcher every ~500 ms.
- **SensorRepository**, owns `CameraX`, the IMU listeners (registered at `SENSOR_DELAY_GAME`), GPS (reference/anchor only), and the MiDaS `DepthEstimator` (throttled to 1 Hz). It forwards IMU samples and camera frames to the native bridge and exposes the returned state as flows; it also feeds the session recorder when recording is enabled.

Around them sit focused single-purpose classes (Table 6.1):

Table 6.1: Application-layer classes and their responsibilities.

| Class                               | Responsibility                                                                           |
|-------------------------------------|------------------------------------------------------------------------------------------|
| NavigationManager                   | Route requests: local Dijkstra over the cached graph; online router for far destinations |
| RoadRegionManager + RegionDiskCache | On-demand road regions; bounded LRU disk persistence with atomic writes                  |
| OsmDataLayer + GeocodeIndexReader   | Bundled Israel-wide offline search index                                                 |
| OfflineGeocoder, OsmRouter          | Destination search; graph routing                                                        |
| LocalMatcher                        | The on-device HMM/Viterbi matcher (§7.1.6)                                               |
| ManeuverStateMachine                | Free-road / on-roundabout / mid-road-U-turn state; releases the rail on maneuvers        |
| GraphRailDot                        | The graph-constrained display position ("ball in a maze")                                |
| CalibrationStore                    | Versioned persistence of the IMU/extrinsics calibration                                  |
| DepthEstimator                      | MiDaS v2.1 small on TensorFlow Lite                                                      |
| SimulationFrameRecorder             | Replay-harness capture (timestamped frames on a bounded queue)                           |
| CrashLogger, GpxExporter            | Native-crash post-mortems; path export                                                   |
| Compose UI files                    | MapScreenUi, CameraUi, calibration/debug/search surfaces                                 |

Figure 6.2 reproduces the SDD class diagram with the as-built renames noted in its caption.

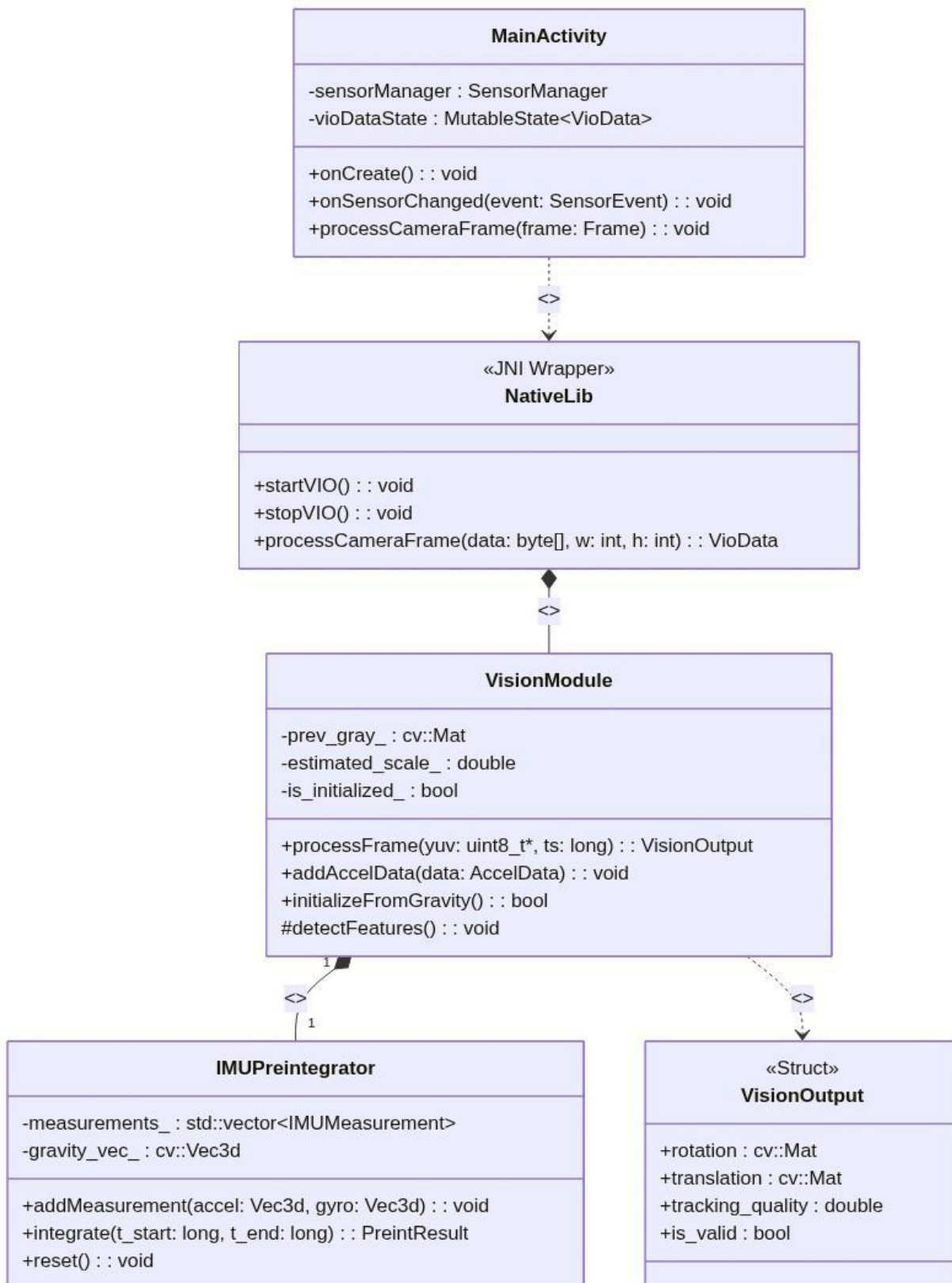


Figure 6.2: The SDD-era class diagram of the core path. As-built renames: *NativeLib* → *NativeBridge* (object with 47 active external methods; one legacy frame-path method is retained commented out), *VisionModule* → *VioEngine/Tracker*, and *MainActivity*'s state-holder role moved into *NavSightViewModel*; the *VisionOutput/IMUPreintegrator* structure is unchanged.

## 6.2.2 Native VIO Core (C++)

Ownership is a simple tree. VioEngine is a thin facade owning the IMUPreintegrator and the Tracker; the Tracker owns everything else (Table 6.2):

Table 6.2: Native-core components and their responsibilities.

| Component            | Responsibility                                                                                                                                                                                                                                              |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Tracker              | The engine driver: runs the whole per-frame pipeline (§6.4.2), owns the modules below.                                                                                                                                                                      |
| TrackKLT             | Pyramidal Lucas–Kanade optical flow + 5-point RANSAC essential-matrix verification.                                                                                                                                                                         |
| EKFState             | The error-state Kalman filter. IMU error state: rotation, gyro bias, velocity, accel bias, position (15 DOF), plus camera–IMU time-offset and camera-mount rotation calibration states (19 total); up to 15 pose clones (6 DOF each) support MSCKF updates. |
| IMUPreintegrator     | On-manifold preintegration between frames <b>and</b> the Madgwick AHRS, the deliberate heading source (the EKF’s yaw is not user-facing).                                                                                                                   |
| UpdaterMSCKF         | Null-space MSCKF update over lost feature tracks: triangulation, Jacobians, $\chi^2$ gate with Huber weighting.                                                                                                                                             |
| UpdaterZeroVelocity  | Stationarity detection (joint IMU + optical-flow $\chi^2$ gate) driving zero-velocity updates.                                                                                                                                                              |
| GroundPlaneEstimator | Legacy ground-plane overlay estimator (every third frame; read-only with respect to the EKF), retained for the camera-screen overlay. The shipped IPM <i>speed</i> runs per-frame inside the Tracker (updateGroundFlowSpeed, §7.1.4).                       |
| FeatureManager       | Feature detection, IDs, lifecycle, keyframe ORB descriptors (relocalization), MSCKF candidate queues.                                                                                                                                                       |
| InertialInitializer  | Startup state machine (wait-stationary → wait-motion → ready, 15 s timeout).                                                                                                                                                                                |

| Component     | Responsibility                                                                                                 |
|---------------|----------------------------------------------------------------------------------------------------------------|
| EventCounters | -180 atomic counters serialized into session telemetry, the observability backbone used throughout validation. |

Modules for loop closure, SLAM landmarks, windowed bundle adjustment, pose-graph optimization, and visual-inertial scale estimation exist in the source tree but are compile-time gated off and never execute; they are not part of the shipped design (§9.3 discusses their future).

## 6.3 Data Design

**The JNI contract (the “narrow waist”).** The bridge exposes 47 native methods, but they fall into five small families (Table 6.3):

Table 6.3: The five families of the JNI contract.

| Family        | Representative methods                                                                                                    | Purpose                                                                                                                     |
|---------------|---------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| Lifecycle     | startVIO, stopVIO, resetVIO                                                                                               | Engine creation, teardown, pose reset                                                                                       |
| Hot path      | processCameraFrameDirect, processAccelerometer, processGyroscope                                                          | The per-frame and per-sample data feeds                                                                                     |
| Configuration | setIntrinsics, setCameraHeight, setInitialHeading, seedMadgwickYaw, setDepthMap, setMidasScaleK                           | Calibration constants and seeds                                                                                             |
| Map feedback  | setRoadHeadingHint, setMapPositionCorrection, setMapPositionEnabled                                                       | Map→VIO legs: heading hint + gated cross-track position correction (both live, §7.1.6), plus the position leg’s kill switch |
| Diagnostics   | getPositionCovariance, getFusedSpeedMps, getGroundFlowSpeedMps, overlay snapshot getters, nativeGetEventCountersJsop<br>n | Health, overlays, telemetry                                                                                                 |

The per-frame hot path is one call: `processCameraFrameDirect(...)` takes the camera’s Y and UV planes as direct byte buffers (zero-copy), image geometry and strides, and the frame timestamp, and returns a single `VioData` object. The full method-by-method reference is Appendix E.

**VioData (native → Kotlin, once per frame; Table 6.4):**

*Table 6.4: The VioData per-frame state snapshot.*

| Group         | Fields                                                                                                                   |
|---------------|--------------------------------------------------------------------------------------------------------------------------|
| Pose          | x, y, z (m from start), orientation, raw visual-odometry echoes                                                          |
| Quality       | trackingQuality, trackedFeatures, totalFeatures, isInitialized, pose flags                                               |
| Speed & scale | estimated scale, fused speed channels (via dedicated getters)                                                            |
| Overlay       | trackedPoints (2-D feature coordinates for the camera overlay)                                                           |
| Diagnostics   | flow statistics, inlier count, step counter/frequency, heading (Madgwick, the user-facing value), camera-IMU time offset |

**Persisted structures.** The calibration store (schema-versioned) holds the gravity-aligned rotation and gyro/accel biases; separate preferences hold the camera mount height, per-mode scale, and MiDaS scale factor. The bundled OSM search index and the road-region disk cache were specified in §5.2; the in-memory region is an immutable `DynamicRoadRegion` (segment R-tree + routing graph + roundabout index) republished atomically on region change, so no consumer ever observes a half-loaded map.

## 6.4 Interactions

### 6.4.1 Activity View: A Ride Session

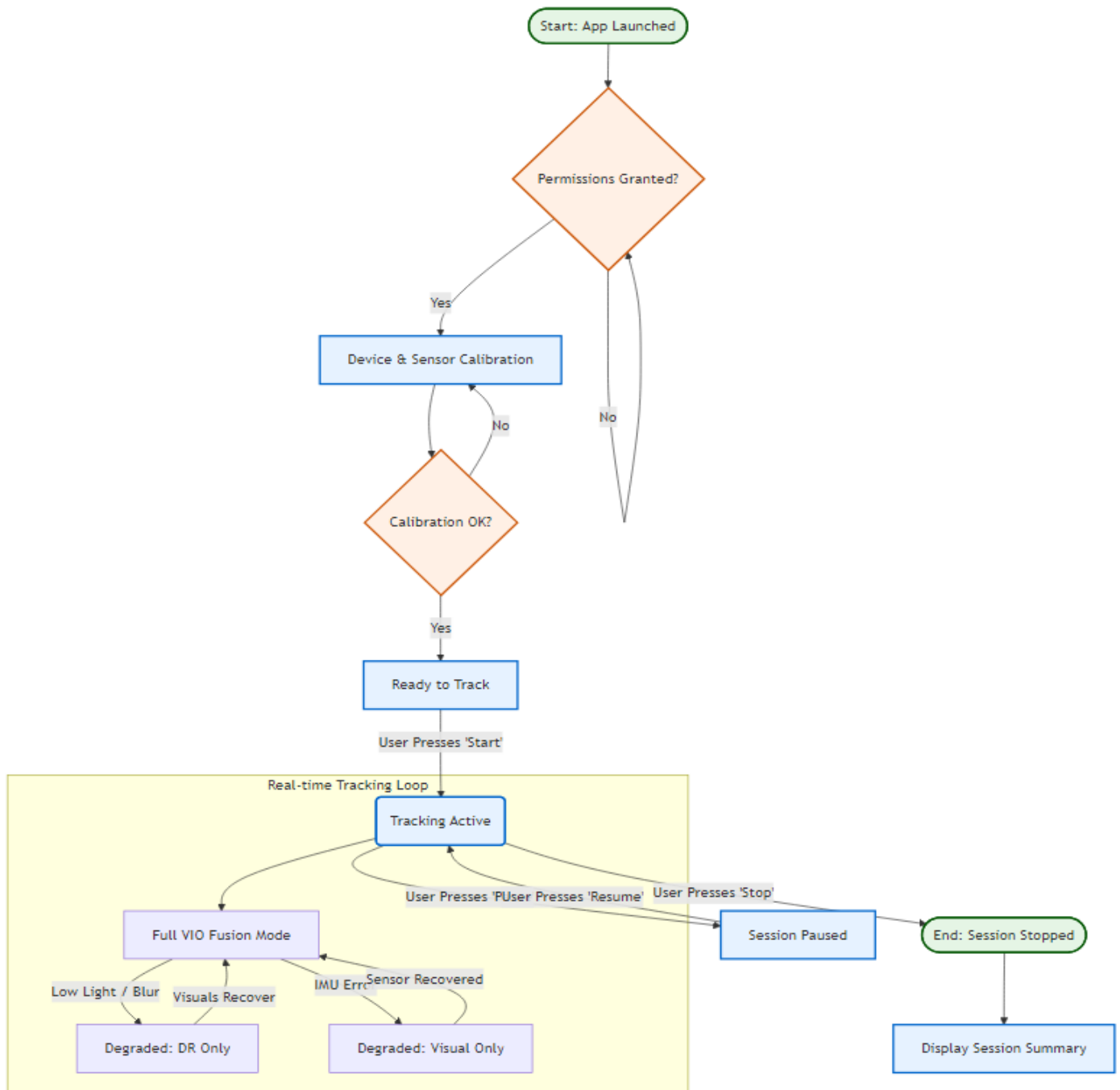


Figure 6.3: The SDD activity diagram of a session: permission gate, calibration loop, the real-time tracking loop with its two degraded branches (DR-only on low-light/blur; visual-only on IMU error), pause/resume, and session summary. The as-built flow matches, with pause realized as OS-lifecycle suspension plus explicit Reset (§5.4).

Figure 6.3 traces the same flow the rider experiences: launch → permission gate → native library load → hold-still calibration (gravity estimate, bias settle; retry loop until converged) → *Ready* → user starts → the tracking loop of §5.3 runs with its degraded branches and automatic recovery → user stops → session summary → telemetry finalized.

## 6.4.2 Sequence: One Camera Frame

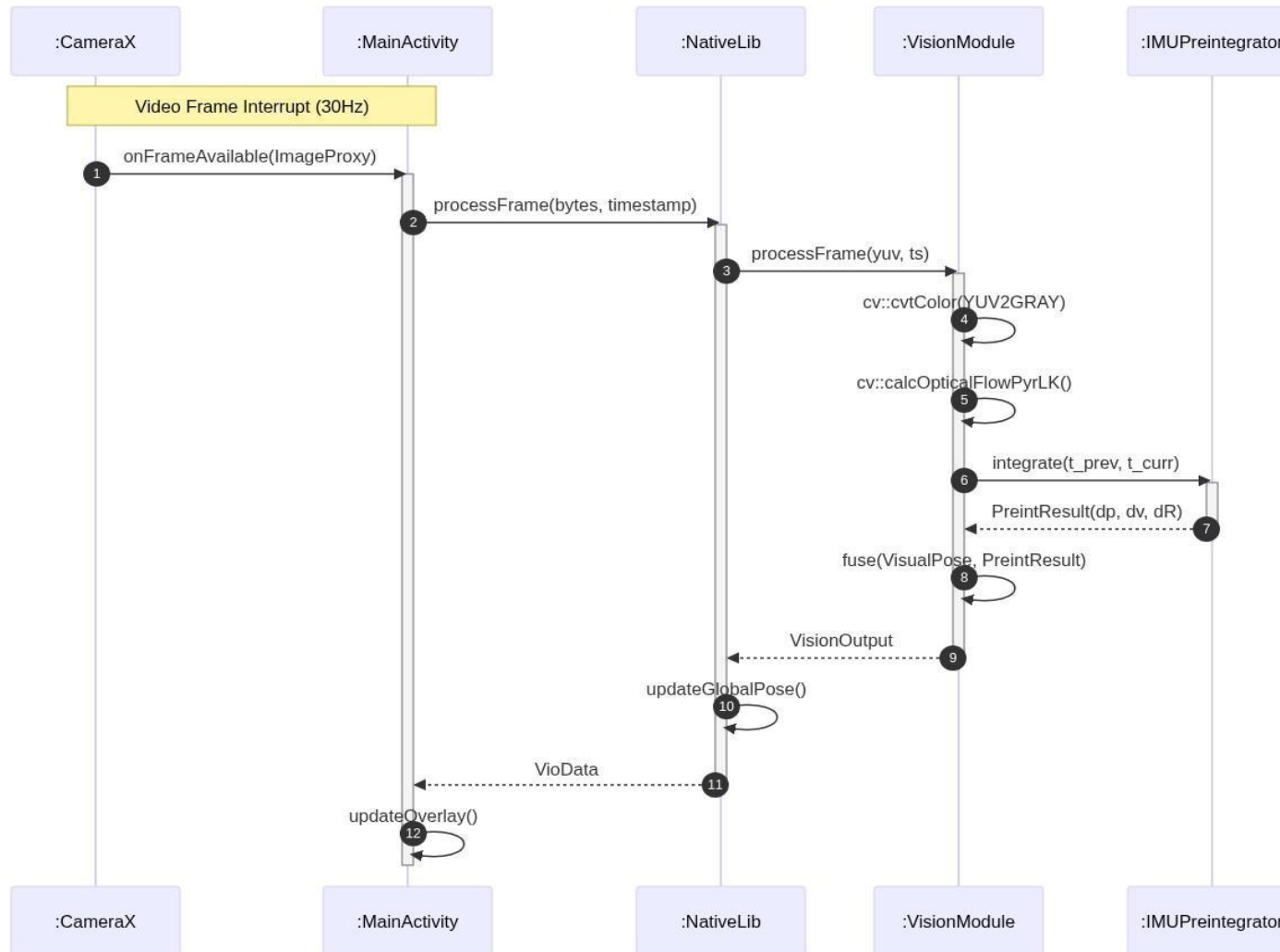


Figure 6.4: The SDD sequence diagram for a single frame. As-built mapping: **:MainActivity** → *SensorRepository* (via *CameraUi*'s analyzer), **:NativeLib** → *NativeBridge/native-lib.cpp*, **:VisionModule** → *VioEngine::processFrame* → *Tracker::processFrame*; the fusion step drawn inside *VisionModule* is the EKF inside *Tracker*.

As-built, the frame of Figure 6.4 proceeds as follows:

1. CameraX delivers a frame to the analyzer (CameraUi) → `SensorRepository.processCameraFrame()`.
2. `NativeBridge.processCameraFrameDirect(...)` marshals the planes; the native dispatcher assembles the grayscale image and calls `VioEngine::processFrame`, which delegates to `Tracker::processFrame`.
3. The Tracker runs the pipeline: contrast normalization (CLAHE) and blur gate → KLT tracking of existing features (+ re-detection to maintain grid coverage) → motion gates and stationarity check (ZUPT) → essential-matrix pose with RANSAC → EKF propagation (consuming the IMU preintegrated since the last frame) and updates (MSCKF on lost tracks, zero-velocity when stationary) → ground-plane speed votes → output assembly.
4. The returned `VisionOutput` is converted to `VioData` and handed back to Kotlin; `NavSightViewModel.handleVioUpdate()` updates the UI state (speed EMA, path, covariance ring), and every ~500 ms triggers the matcher tick (§6.4.3).

At message granularity, one frame is the following exchange (Table 6.5, the as-built refinement of the SDD's twelve-message sequence):

Table 6.5: One camera frame as a twelve-message exchange.

| # | From → To                       | Message                                               | Effect                                             |
|---|---------------------------------|-------------------------------------------------------|----------------------------------------------------|
| 1 | CameraX → SensorRepository      | <code>onFrame(ImageProxy)</code>                      | Frame delivered at ~23 fps                         |
| 2 | SensorRepository → NativeBridge | <code>processCameraFrameDirect(y, uv, ..., ts)</code> | Zero-copy planes cross JNI                         |
| 3 | NativeBridge → VioEngine        | <code>processFrame(yuv, w, h, ts)</code>              | Grayscale assembled                                |
| 4 | VioEngine → Tracker             | <code>processFrame(frame)</code>                      | Pipeline entry                                     |
| 5 | Tracker → Tracker               | CLAHE + blur gate                                     | Normalize; reject smeared frames                   |
| 6 | Tracker → TrackKLT              | <code>track(prev, cur)</code>                         | KLT correspondences                                |
| 7 | TrackKLT → Tracker              | inliers (5-pt RANSAC)                                 | Verified pose direction                            |
| 8 | Tracker → IMUPreintegrator      | <code>integrate(t_prev, t_cur)</code>                 | Preintegrated $\Delta R$ , $\Delta v$ , $\Delta p$ |

| #  | From → To                         | Message                  | Effect                                         |
|----|-----------------------------------|--------------------------|------------------------------------------------|
| 9  | Tracker →<br>EKFState             | propagate() +<br>updates | MSCKF / ZUPT<br>applied under $\chi^2$<br>gate |
| 10 | Tracker →<br>GroundPlaneEstimator | speed votes              | Metric speed or<br>zero-lock                   |
| 11 | Tracker →<br>NativeBridge         | VisionOutput             | Global pose<br>assembled (Z-<br>up→Y-up)       |
| 12 | NativeBridge →<br>ViewModel       | VioData                  | UI state updated; matcher<br>tick armed        |

### 6.4.3 Sequence: Matcher Tick and Map Feedback

1. `ManeuverStateMachine.tick()` runs first with the current position, heading, and roundabout index; it tracks maneuver state (free road / on-roundabout / mid-road U-turn) and, on a maneuver event, releases the matcher's road lock so the rail can re-latch cleanly instead of fighting the maneuver.
2. The `ViewModel` builds the match window (the last 150 raw-trajectory points subsampled to ~15, plus the current position) and calls `LocalMatcher.match(window, region)`: candidate segments within 30 m are scored by the HMM (emission = perpendicular distance under  $\sigma_z = 20$  m; transition = route-length consistency with  $\beta = 6$  m, plus same-way and rail bonuses) and Viterbi-decoded (§7.1.6 gives the code). Matches with confidence  $\geq 0.30$  are accepted.
3. `GraphRailDot`, the “ball in a maze”, either acquires onto the matched road or advances along the graph by the estimated speed (capped at 15 m/s), choosing junction branches by gyro-relative heading. Its position is what the rider sees; if the rail is not acquired, the matcher's point (or, failing that, a simple perpendicular road-snap) is displayed.
4. **Feedback legs (map → VIO):** on a confident straight-road match (confidence  $\geq 0.6$ ) the road tangent is pushed down as a bounded heading hint (`setRoadHeadingHint`). Once the same high-confidence match has held for a  $\geq 3$ -tick streak, the cross-track part of the (ball – raw VIO) error follows as a position correction (`setMapPositionCorrection`) that native bleeds into the raw trajectory at  $\leq 0.05$  m per frame, live by default, and individually toggleable through its own setter (`setMapPositionEnabled`) rather than shared state, so it can be disabled in isolation if a ride ever shows it misbehaving (§7.1.6). The stronger matcher-supervised re-acquisition remains **off by default**.

## 6.5 Software Architecture Pattern

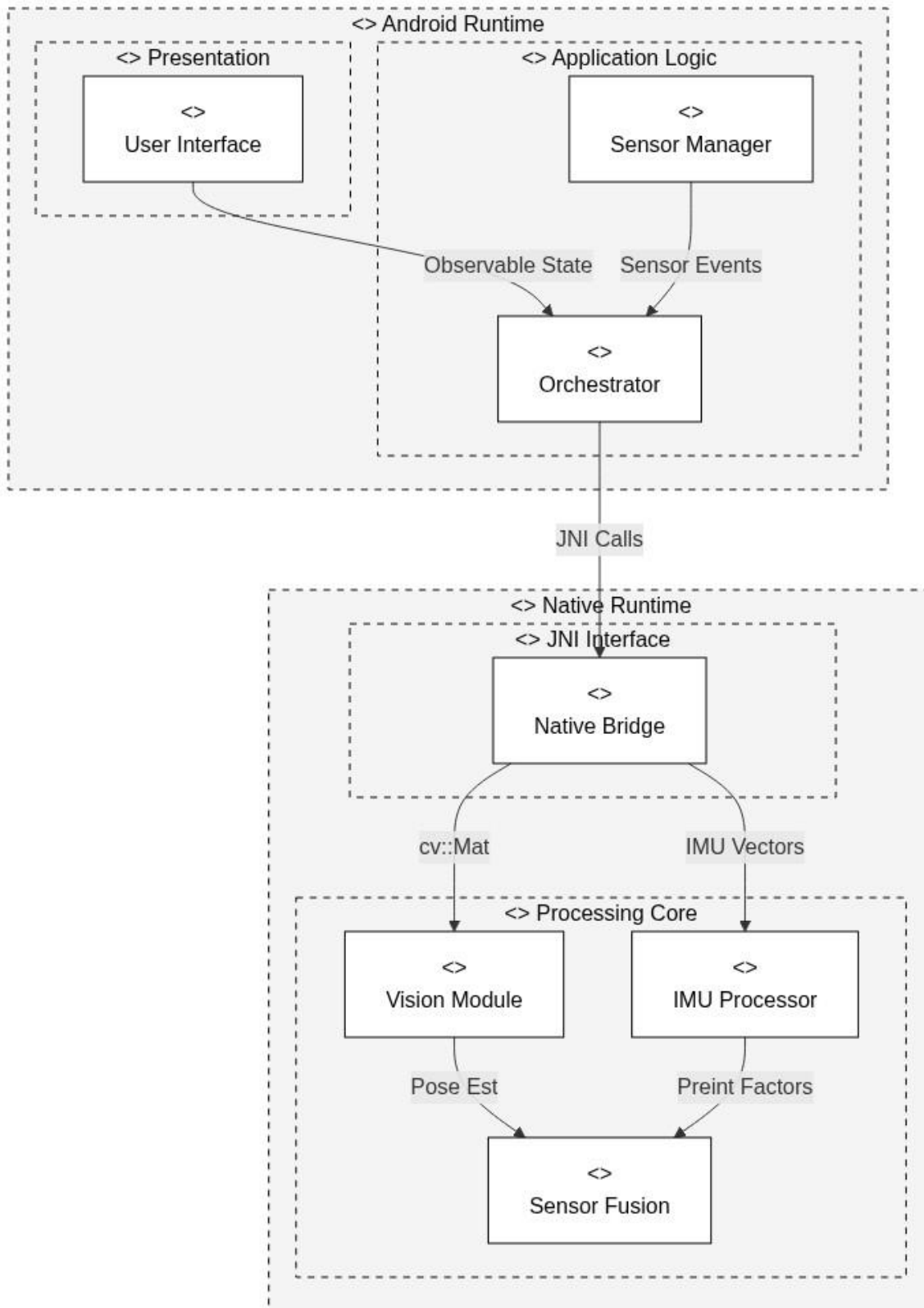


Figure 6.5: The SDD component diagram: Android Runtime (Presentation, Application Logic, Orchestrator) and Native Runtime (JNI Interface, Processing Core with Vision / IMU / Sensor Fusion). The as-built system preserves this shape exactly and adds the Tier-2 map stack of Figure 5.1.

Two patterns govern the code (Figure 6.5). Across tiers, it is a **layered pipeline**: each layer consumes its lower neighbor through one narrow interface (Compose observes the ViewModel; the ViewModel calls the bridge; the bridge calls the engine), which is what allowed the engine to be lifted unchanged into the desktop replay harness. Within Tiers 1–2, it is **MVVM**: Compose views are pure functions of NavSightViewModel state; user intents flow back as method calls; the Model is the VioData/matcher state the lower tiers produce. The discipline paid off measurably, UI redesigns late in the project (June 2026) touched no engine code, and engine changes were validated in replay with no UI involved.

## 6.6 Concurrency and Threading Model

A real-time navigation app is a concurrency problem as much as an algorithm problem: sensors, camera, depth inference, matching, and rendering all run at different rates, and none may block the UI. NavSight resolves this with a small set of clearly-owned threads (Table 6.6):

Table 6.6: The threading model: every thread, its rate, and what it must never block.

| Thread / context  | Rate      | Work                                                       | Must not block |
|-------------------|-----------|------------------------------------------------------------|----------------|
| Main (UI)         | display   | Compose rendering; ViewModel state; matcher tick (~500 ms) | ever           |
| CameraX analyzer  | ~23 fps   | delivers frames; hands planes to the JNI bridge            | the camera     |
| Native VIO thread | per frame | the entire Tracker pipeline (§6.4.2)                       | the analyzer   |
| Sensor callback   | ~50 Hz    | IMU samples forwarded to the native queues                 | the sensor bus |
| Depth (coroutine) | 1 Hz      | MiDaS inference off the hot path                           | the frame loop |

Three design rules keep this safe. First, **the native VIO thread is the only place heavy computation runs**: the JNI bridge marshals a frame and returns immediately with the latest available VioData, so a slow frame never stalls the camera or the UI. Second, **IMU ingestion is decoupled from consumption**: sensor callbacks only enqueue timestamped samples; the frame loop drains and preintegrates them at frame boundaries, so the ~50 Hz sensor rate and the ~23 fps frame rate never contend. Third, **the matcher runs on the main thread but only every ~500 ms** and on a sub-sampled window (~15 points), so its cost is negligible and its results need no cross-thread synchronization with the render pass. The map region, when it

changes, is published as an *immutable* object and swapped atomically, a reader mid-tick either sees the whole old region or the whole new one, never a half-built graph.

The hottest shared structure (the engine's global pose, read by the diagnostics getters while the VIO thread writes it) is guarded by a dedicated mutex, and the getters snapshot under it rather than holding it. Around it, each native module keeps its own small lock for the cross-thread hand-offs it owns (overlay buffers, the depth-map exchange, calibration snapshots); the governing rule is not "one lock" but *no shared mutable state without a single named owner*, everything else is message-passing or immutable publication. The payoff showed up in validation as the *absence* of a class of bug (no tearing, no half-updated overlays, no UI jank traceable to the engine) across every validation day (T10).

# 7 Implementation

This chapter documents how NavSight was actually built: the algorithms with their real constants (§7.1), the user interface (§7.2), the engineering environment and practices (§7.3–§7.10), and how the project was run (§7.11). Everything here describes the validated build (v1.0-osm); where a constant appears, it is the constant in the shipped code, not a design-time intention.

## 7.1 Algorithm Design

### 7.1.1 Visual Front-End: KLT Tracking

Each frame is converted to grayscale and contrast-normalized (CLAHE [19], adaptive histogram equalization that keeps texture visible across the frame's bright and shadowed halves), then gated on blur before any tracking: a frame too smeared to track is cheaper to skip than to mis-track. Shi-Tomasi corners [10] are maintained on a spatial grid for even coverage; without the grid, corners cluster on the nearest high-contrast object and the pose estimate degenerates. Existing features are tracked with pyramidal Lucas–Kanade optical flow [11]: the pyramid lets large inter-frame motions be found coarse-to-fine, essential at riding speed where a feature can travel forty pixels between frames. Tracked correspondences are then verified geometrically with a 5-point RANSAC essential-matrix fit [18]: inliers survive, while outliers (a passing car, a pedestrian, a mis-track) are recycled. The tracker's job is deliberately narrow: produce reliable frame-to-frame point correspondences for everything downstream, and know when it cannot.

### 7.1.2 Inertial Pipeline: Preintegration and Madgwick Heading

Between consecutive frames, IMU samples are preintegrated on-manifold [12]: rotation accumulates on the  $SO(3)$  manifold, and velocity and position accumulate under the rotated specific force:

$$\Delta R_{ij} = \prod_{k=i \dots j-1} \exp(\omega_k \Delta t) \text{ (rotation on } SO(3))$$

$$\Delta v_{ij} = \sum_k \Delta R_{ik} a_k \Delta t \text{ (velocity increment)}$$

$$\Delta p_{ij} = \sum_k (\Delta v_{ik} \Delta t + \frac{1}{2} \Delta R_{ik} a_k \Delta t^2) \text{ (position increment)}$$

The result is a single motion constraint per frame interval, relative to the pose at the start of that interval rather than to any global frame, and independent of the absolute starting state. The filter therefore consumes one preintegrated measurement instead of dozens of raw samples, and re-linearization never requires re-integration.

In parallel, a Madgwick AHRS [3] runs at sensor rate and is, deliberately, the system's heading source. The compass snaps yaw once at startup, and from then on contributes only through a weak, disturbance-gated complementary term: on a magnetically clean field the fused yaw is pulled gently toward the compass with time constant  $\tau \approx 5$  s (per-tick gain  $k = \Delta t / \tau$ ), and a compass sample is rejected outright whenever it disagrees with the gyro by more than  $35^\circ$ , the signature of the urban magnetic disturbance, near vehicles and infrastructure, that corrupts magnetometer-*trusting* filters. Heading is thus gyro-primary: the gyro drives every tick and a disturbed compass is simply ignored, yet stays honest on clean fields instead of being trusted once and then abandoned. The EKF maintains its own orientation for fusion, but the user-facing heading is the Madgwick output; the two agree in steady state, and the separation means a transient fusion upset never swings the displayed compass. On confident straight-road matches, heading is additionally nudged toward the road tangent  $\theta_{\text{road}}$ , with the  $\pm 180^\circ$  ambiguity resolved travel-aligned ( $\Delta = \text{wrap}_{\pm}(\theta_{\text{road}} - \theta)$ , choosing  $\arg \min(|\Delta|, |\Delta \pm 180^\circ|)$ ) and the correction applied only when the residual is short of perpendicular ( $|\Delta| < 90^\circ$ ). There is deliberately no tighter alignment threshold: the target is the locked road's *own* tangent, trusted by road identity; a genuinely crossing road never reaches the nudge, because the ball is never confidently matched onto one.

### 7.1.3 Error-State EKF and MSCKF Updates

The filter estimates an error state of 19 dimensions (Table 7.1):

Table 7.1: The 19-dimensional error state, the classical 15-DOF VIO core plus four self-calibration rows.

| Rows  | Error state       | Meaning                                      |
|-------|-------------------|----------------------------------------------|
| 0–2   | $\delta\theta$    | rotation error (rad)                         |
| 3–5   | $\delta b_g$      | gyroscope bias error (rad/s)                 |
| 6–8   | $\delta v$        | velocity error (m/s)                         |
| 9–11  | $\delta b_a$      | accelerometer bias error (m/s <sup>2</sup> ) |
| 12–14 | $\delta p$        | position error (m)                           |
| 15    | $\delta t_d$      | camera-IMU time-offset error (s)             |
| 16–18 | $\delta\phi_{bc}$ | camera-mount rotation error (rad)            |

The first fifteen rows are the classical VIO core; the last four let the filter *self-calibrate* its two most fragile constants (the timestamp skew between camera and IMU, and the exact rotation of the phone in its mount), both of which field experience showed drifting between rides. Up to 15 historical pose clones (6 DOF each) extend the state for MSCKF updates [2]: when a feature track ends, the feature is triangulated against the clones that observed it and applied as a null-

space-projected update; the feature's own position is marginalized out analytically, so the state never grows with scene content. Every update passes a  $\chi^2$  gate with Huber weighting: for a measurement residual  $\mathbf{r}$  with innovation covariance  $\mathbf{S} = \mathbf{H} \mathbf{P} \mathbf{H}^T + \mathbf{R}$ , the update is accepted only when the Mahalanobis distance falls under the gate threshold,

$$\mathbf{r}^T \mathbf{S}^{-1} \mathbf{r} < \chi^2_{\alpha, d} \text{ (gate on } d \text{ degrees of freedom at confidence } \alpha \text{)}$$

so a measurement improbable under the filter's own covariance is down-weighted or rejected outright, which is what keeps one burst of outlier tracks from corrupting a whole ride. A zero-velocity detector (joint IMU + optical-flow gate) locks the filter during standstill; gravity alignment at initialization fixes the world frame. The filter's covariance is not internal bookkeeping; it drives the user-facing uncertainty ring, the tracking-health chip, and every gate above.

Figure 7.1 summarizes the filter as a pipeline: IMU propagation feeding the three measurement updates (MSCKF visual, gravity alignment, ZUPT), all passing the  $\chi^2 = 22.5$  gate before the bounded state reaches the map matcher.

## NavSight - 15-DOF Error-State EKF Pipeline

tightly-coupled visual-inertial fusion, ENU Z-up world frame

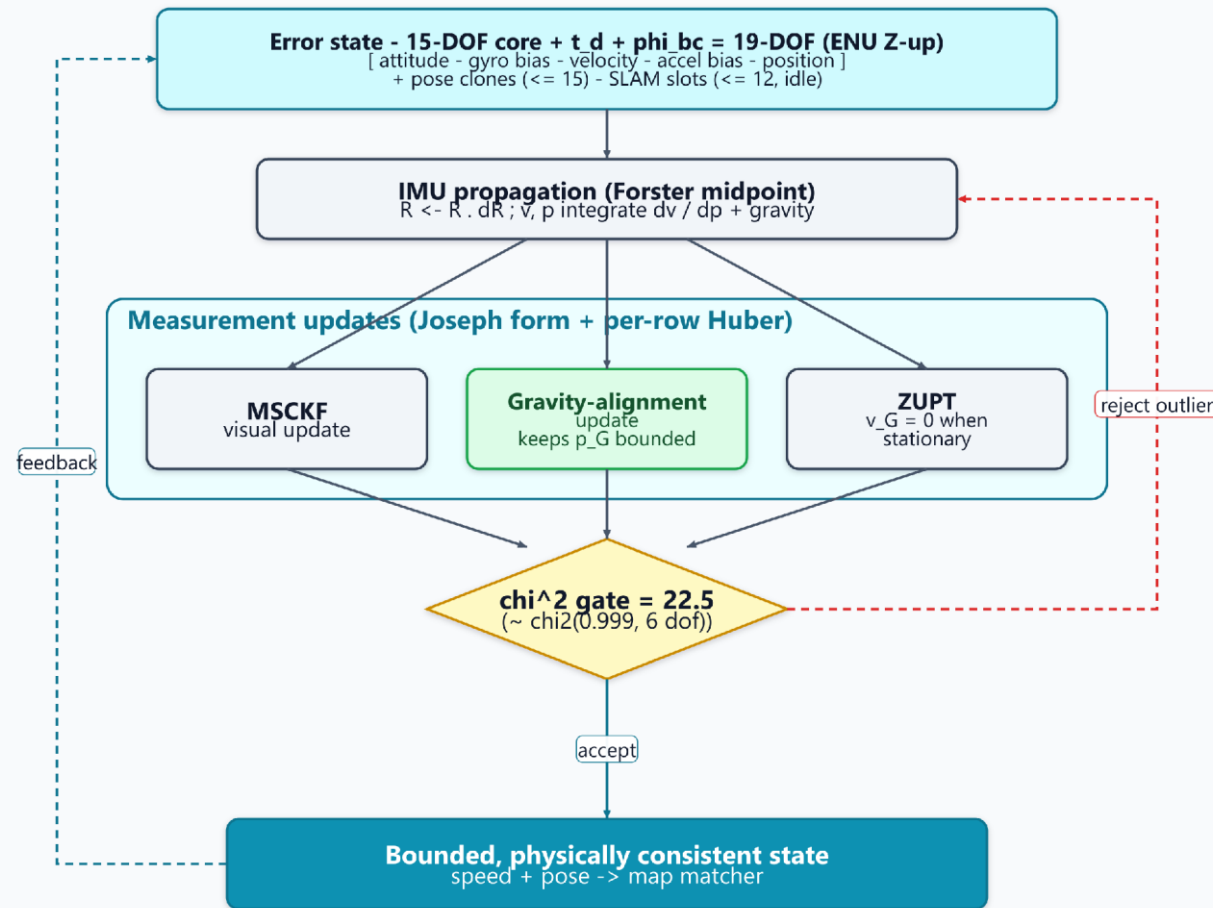


Figure 7.1: The error-state EKF pipeline. A 15-DOF core with self-calibration (19-DOF total), IMU propagation, the three measurement-update families, and the  $\chi^2$  acceptance gate.

#### 7.1.4 Metric Speed: the Ground-Plane (IPM) Estimator

Monocular vision has no scale; NavSight's scale anchor is geometric; the road is a plane at the calibrated mount height  $h$  below the camera. With  $\mathbf{n}$  the unit up-vector in the camera frame (from filtered gravity) and  $\mathbf{r}_i = (x', y', 1)$  the ray of road pixel  $i$ , the depth, the predicted flow, the least-squares speed, and the per-point noise floor are:

$$Z_i = h / (\mathbf{n} \cdot \mathbf{r}_i) \text{ (depth of road pixel } i)$$

$$a_i = (\text{ufwd} - \mathbf{r}_i \cdot \text{ufwd}, z) / Z_i \text{ (image flow per unit forward speed)}$$

$$v_i = -(\mathbf{f}_i \cdot \mathbf{a}_i) / (\mathbf{a}_i \cdot \mathbf{a}_i) \cdot (1/\Delta t) \text{ (least-squares speed from flow } \mathbf{f}_i; \text{ forward motion drives road flow opposite } \mathbf{a}_i, \text{ hence the sign)}$$

$$\sigma_{v,i} = (\sigma_{px} / f_x) / (|\mathbf{a}_i| \cdot \Delta t), \quad \sigma_{px} = 0.5 \text{ px (per-point resolution floor, noise amplifies with distance, } \propto Z)$$

Every road point thus reports both a speed *and its own noise floor*, the resolution below which that point cannot distinguish motion from tracker jitter. The decision logic built on this is two-sided by design. A point **votes** its speed only if  $v_i > 3\sigma_{v,i}$  *and* its flow is forward-coherent (within a  $45^\circ$  cone of the expected direction); a point whose measured speed is below its own floor, when that floor is tight enough to mean something ( $3\sigma_{v,i} \leq 1 \text{ m/s}$ ), is a **zero-witness**, positive evidence of stillness, not mere absence of motion. Five votes yield the median speed; failing that, five zero-witnesses yield *exactly zero*. This is why a standing phone displays a clean 0 km/h: a one-sided magnitude gate would rectify 25–34% of random-direction tracker noise into positive speed, and the two-sided taxonomy is what eliminates that phantom creep (validated as NFR-2.4, test T3).

**Inertial bridge.** Between visual measurements, speed propagates from body-forward linear acceleration; the mean specific force minus gravity, projected on the horizontalized optical axis:

$$v_k = v_{k-1} + a_{\text{fwd}} \cdot \Delta t \text{ (predict, every frame)}$$

$$v_k \leftarrow (1-\alpha) \cdot v_k + \alpha \cdot \text{median}\{v_i\}, \quad \alpha = 0.15 \text{ (correct, frames with } \geq 5 \text{ votes)}$$

Prediction alone is trusted for at most 6 s, a  $0.3 \text{ m/s}^2$  accelerometer bias integrates to  $1.8 \text{ m/s}$  over that window, after which decay resumes rather than letting the bias masquerade as speed. Chapter 8 isolates this bridge's contribution by controlled A/B on identical replayed input: +24% integrated distance through turns (Figure 8.6), the difference between a ball that dies in every corner and one that carries through it.

Figure 7.2 lays the whole channel out, from per-pixel ray to the three possible outcomes of a frame (median speed, exact zero, or bridged prediction).

## NavSight - Speed Estimation (Ground-Plane IPM + Inertial Bridge)

metric speed from optical flow; no learned depth in the speed path

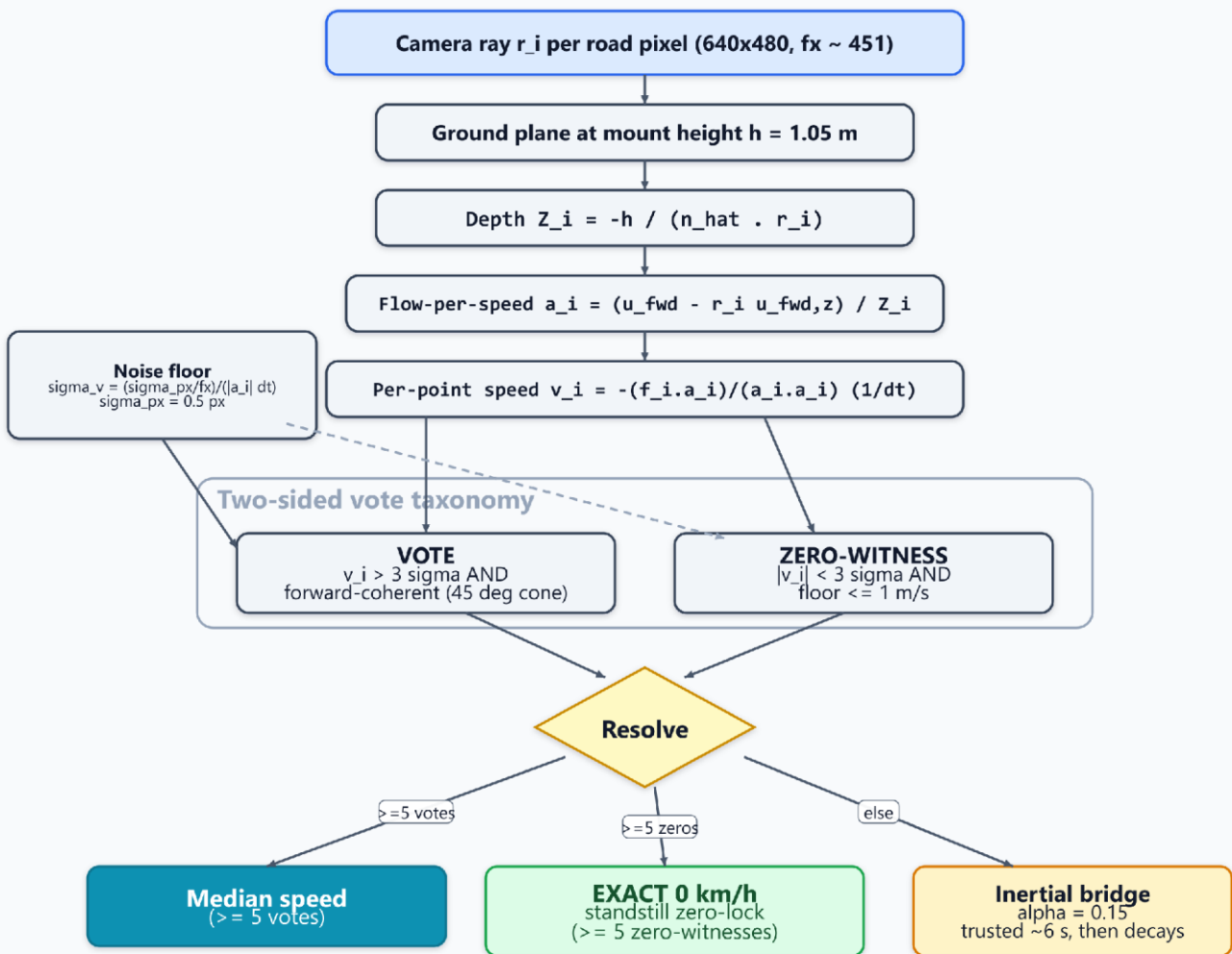


Figure 7.2: The ground-plane (IPM) speed estimator: per-pixel depth from the calibrated mount height, per-point speed with its own noise floor, the two-sided vote taxonomy, and the inertial bridge.

### 7.1.5 Depth-Weighted Flow and Per-Gait Scale

A second speed channel drives the raw trajectory (“the dot”) that feeds the map matcher: optical flow weighted by relative monocular depth from MiDaS [5] (DepthEstimator, TensorFlow Lite [15], 256×256 output, throttled to 1 Hz; an affine fit in disparity space relates the network’s scale-and-shift-ambiguous output to usable inverse depth). Weighting flow by depth matters because a nearby bush streaming past at high angular rate and a distant building barely moving carry very different metric information; depth is what converts pixels-per-frame into a consistent speed observable.

Because flow-derived speed still carries a mode-dependent scale (a walking phone bobs, a riding phone glides) a cadence-based gait classifier selects a per-mode calibration factor  $K$  (walking, running, vehicle) that is learned during travel and persisted across sessions. The history behind this design is instructive and told in §8.7: a single shared  $K$  measurably failed across gaits before the per-mode split. When the visual channel drops out entirely, translation falls back to the bridged estimate, on a representative validation ride the live-session engine counters recorded in the V&V report show the fallback engaging 36 times, keeping the dot moving through every dropout, and ORB relocalization re-anchoring the front-end 4 times.

### 7.1.6 Map Matching: the HMM Rail-Follower

The matcher implements the Newson–Krumm HMM [4] on-device. Every ~500 ms the application layer submits a window of the recent raw trajectory (the last 150 points sub-sampled to ~15, plus the current position, enough history to give Viterbi a story, few enough points to be effectively free). Candidate road segments within 30 m of each window point are scored and the jointly most probable road sequence decoded. The scoring core, adapted for print from LocalMatcher.kt (the two-branch trellis is flattened and helper signatures simplified; the constants and scoring structure are exactly as shipped):

```
//      Viterbi      over      per-point      candidates      (LocalMatcher.kt)
for              (c              in              layer)          {
    val emit = -0.5 * (c.offsetM / SIGMA_Z) * (c.offsetM / SIGMA_Z) // log-Gaussian
    emission
    val railB = if (railWayId != null && c.wayId == railWayId) RAIL_BONUS else 0.0
    // transition from each previous-layer node: route-length consistency
    val dRoute = haversineM(pn.cand, c) // along-candidate distance
    val trans = -abs(dGc - dRoute) / BETA // Newson-Krumm consistency
    val sameWay = if (pn.cand.wayId == c.wayId) SAME_WAY_BONUS else 0.0
    score = pn.score + trans + sameWay + emit + railB
}
// confidence ~ how tightly the chosen path hugs the roads
val confidence = exp(-meanOffsetM / SIGMA_Z).coerceIn(0.0, 1.0)
```

with the shipped constants:  $\sigma_z = 20$  m (emission spread, VIO position noise plus OSM way accuracy),  $\beta = 6$  m (transition scale), same-way bonus 0.7, rail bonus 1.2, candidate search radius 30 m, at most 5 candidates per point, minimum 4 matchable points. A window point with no candidate is an off-road gap that restarts the trellis rather than forcing a match. Matches with confidence  $\geq 0.30$  are accepted; the emission/transition balance means a momentary drift toward a parallel street is outvoted by the window's history, while the **rail bonus**, awarded to candidates on the currently locked way, is what keeps the matcher from re-snapping to a

parallel road on every wobble. That lock is not absolute: the maneuver state machine (free road / on-roundabout / mid-road U-turn, with roundabout re-entry hysteresis) releases it on genuine maneuvers so the rail re-latches cleanly.

The displayed position, the **rail ball**, then advances *along the graph* by the estimated speed ( $s_{k+1} = s_k + v_k \Delta t$ , capped at 15 m/s), choosing junction branches by gyro-relative heading from the pre-junction anchor:  $b^* = \arg \min_b |\theta_b - (\theta_{\text{anchor}} + \Delta\theta_{\text{gyro}})|$ . Because the ball is constrained to graph edges, it can never leave the road network or jump across the map; the on-road guarantee is structural, not a correction applied after the fact. Figure 7.3 summarizes the chain from EKF pose to rail ball.

### NavSight - HMM Map Matching

Newson & Krumm (2009), Viterbi-decoded over offline OSM

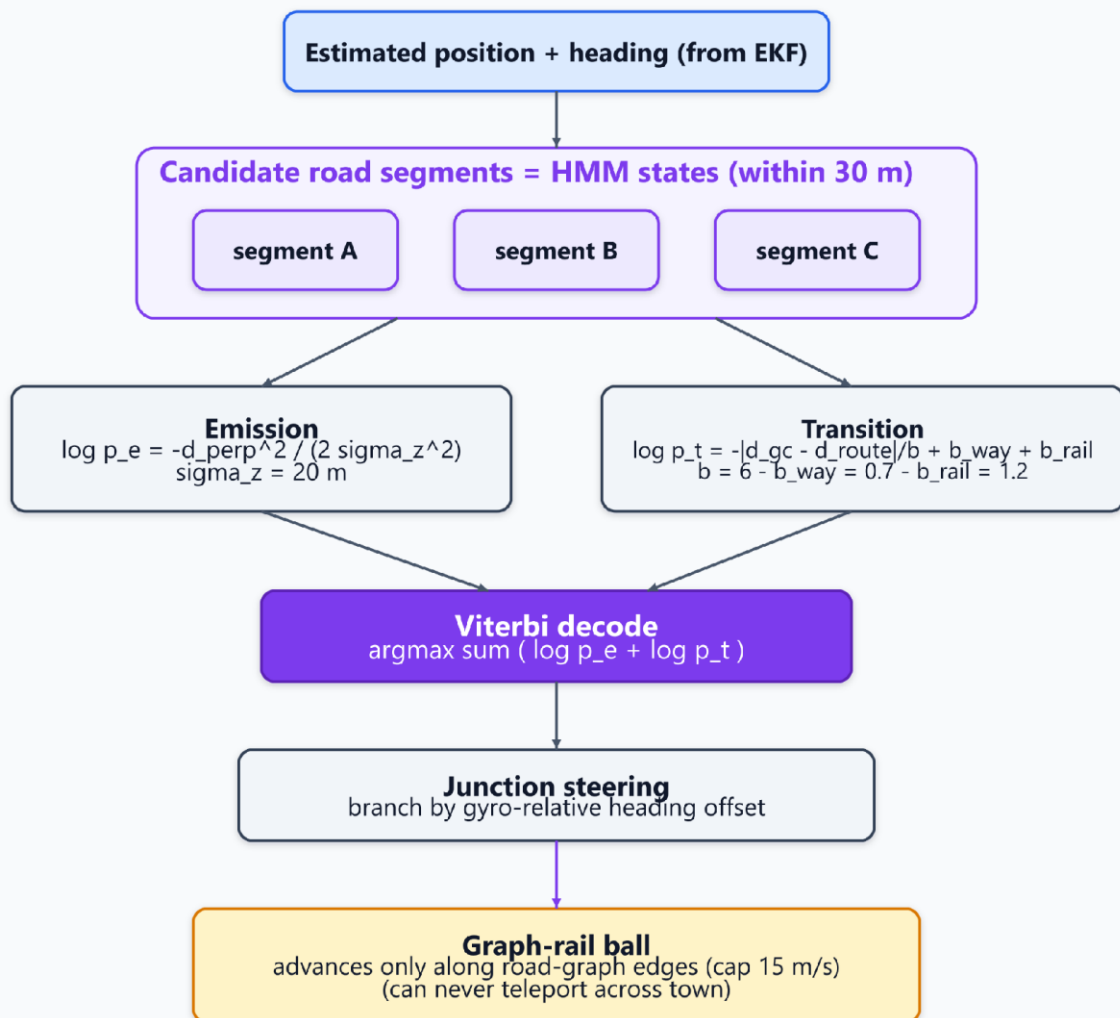


Figure 7.3: HMM map matching: candidate segments as HMM states, the emission and transition scoring with the shipped constants, Viterbi decoding, junction steering, and the graph-constrained rail ball.

The map also feeds back into the engine, in a deliberately staged way. Two bounded legs are **live in the validated build**. The first is the heading hint: on a confident straight-road match (match confidence  $\geq 0.6$ , ball acquired, not on a roundabout) the road's local tangent is offered to the engine, which applies a small fractional nudge natively for residuals short of perpendicular ( $|\Delta| < 90^\circ$ , §7.1.2). The second is the position-correction leg (`setMapPositionCorrection`): under the same conditions held for a  $\geq 3$ -tick streak, the *cross-track* part of the (ball – raw VIO) error is bled into the raw trajectory at  $\leq 0.05$  m per frame (ignored beyond a 15 m divergence, suspended while the magnetometer is actively fusing). Enabled by default, it slowly pulls the drifting raw trail back toward the road, while the *displayed* ball, being graph-constrained, never depends on it. One stronger correction is **implemented but disabled by default**: the **matcher-supervised wrong-road re-acquisition** (`MATCHER_SUPERVISION_ENABLED`), which would re-acquire the ball onto the matcher's road after a sustained  $\sim 25$  m divergence from a confident match, held off because it regressed a heading-degraded ride in offline replay, awaiting an on-device re-record before it is trusted. This staging is deliberate: the shipped system leans on the *structural* on-road guarantee plus gated, bounded feedback, and holds the aggressive re-acquisition back until it is individually validated, the same conservative discipline that disabled the SLAM stack (§9.3).

### 7.1.7 Offline OSM Data Pipeline

`scripts/build_haifa_assets.py` compiles an Israel-wide OpenStreetMap [14] place index ( $\sim 254,000$  places with an inverted token index in a 16.6 MB little-endian blob (format NSGC, §5.2)) bundled in the APK for fully offline destination search. Road geometry loads on demand around the rider: fetched from Overpass, parsed into an immutable in-memory region (segment R-tree for nearest-road queries, routing graph, roundabout index) that is republished atomically, and persisted in a bounded LRU disk cache with atomic temp-file-rename writes. Local routing is Dijkstra's shortest-path algorithm [20] over the region graph; far destinations can use an online router when connectivity exists. The pipeline also implements packed formats for road segments (NSSG) and routing graphs (NSGR), ready for pre-packing regions into the APK, the path §9.3 identifies for removing the first-visit connectivity need.

## 7.2 Interfaces (UI Walkthrough)

The main screen is designed to be glanceable from a handlebar mount: one number (speed), one color (health), one ball (position). Figure 7.4 shows it at standstill before a ride: the slim header carries the live speed (reading exactly **0 km/h**, the zero-witness lock of §7.1.4) beside the green VIO health chip (calibrated and converged); below it sits the “Where to?” offline search bar; the purple rail ball rests on the road graph at the start position; and the footer shows the distance-tracked counter (0 m), the session Reset control, and the three-state night-mode

toggle (Auto). Tapping the health chip opens the calibration sheet (mount height, calibration status detail); a session ends in a summary of distance and duration.

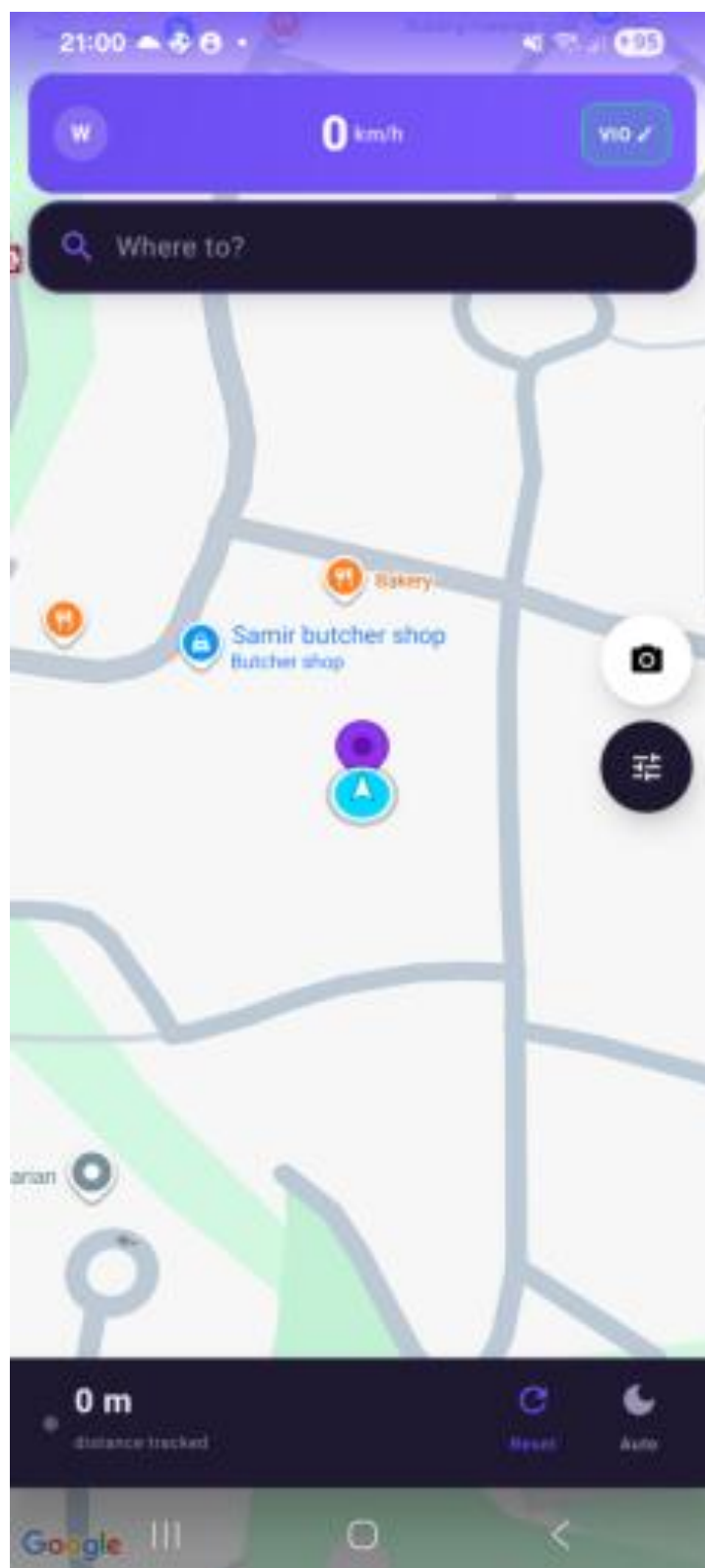


Figure 7.4: The live application at standstill before a ride. Speed is locked at 0 km/h, VIO chip green, the purple position ball resting on the road graph.

Figure 7.5 shows the system's own view of the road during a validation ride: the grayscale working frame with the amber outline marking the ground-plane sampling region whose depth

is known from the calibrated mount height. Each green arrow is the verified frame-to-frame motion of one road-texture point (template-matching reference); the red arrow beside it is the production KLT tracker's measurement of the same point; their agreement is the measured flow that §7.1.4 converts into metric speed. Lane markings, asphalt grain, and shadow edges serve as texture; points off the road (parked cars, trees) are excluded by the mask.



*Figure 7.5: A rear-camera frame during a validation ride, showing the amber ground-plane sampling region and per-point optical-flow arrows (green: template-matching reference; red: production KLT).*

The remaining surfaces follow the same design language: a rationale-first permission flow at first launch; the hold-still calibration screen driving the chip's state progression; offline destination search results with the computed route polyline on the map; and a grouped debug panel (engine counters, matcher state, speed channels) that served as the developer surface behind Chapter 8's validation work.

## 7.3 Development Environment

Android Studio with the NDK/CMake toolchain (compileSdk 34); the OpenCV 4.5.3 [13] Android SDK vendored in-repo so every checkout builds identically; MSYS2/MinGW GCC for the desktop replay-harness build; Python 3 (NumPy/Matplotlib) for the asset pipeline, telemetry analysis, and the validation-report generator; GitHub for hosting and CI.

## 7.4 Programming Languages

Kotlin (application layer, Compose UI), C++17 (VIO core, chosen over C++14 for structured bindings and constexpr improvements used throughout the engine), Python (asset pipeline, validation tooling). The language boundary is the architecture boundary: no algorithmic code in Kotlin, no UI concern in C++.

## 7.5 Limitations of the System

- Road data must be cached (first visit to an area needs connectivity); the bundled index covers destination search only.
- Metric scale depends on the calibrated mount height; a badly wrong height skews speed proportionally until recalibrated.
- The camera must see the road: extreme darkness or full occlusion degrades to the bounded inertial bridge ( $\leq 6$  s full trust), after which speed decays by design rather than drifting.
- Wrong-road divergences wait for a confident re-match (the maneuver state machine releases the rail lock; the supervised re-acquisition leg ships disabled) rather than jump, bounded honesty over unbounded guessing.
- One reference device is validated (Samsung S21 Ultra); other devices need their own camera intrinsics calibration (Appendix D).

## 7.6 Installation and Configuration

1. Clone the repository and open it in Android Studio (the OpenCV Android SDK is vendored; no separate download).

2. Build with `gradlew assembleDebug`; the debug build type carries the `-osm` version suffix and its own application ID (validated build: `v1.0-osm`).
3. The offline search index ships in `app/src/main/assets/osm/haifa/`; to rebuild it from fresh OSM data run `python scripts/build_haifa_assets.py`.
4. Install to the device (`adb install`), grant camera + location permissions on first launch.
5. Mount the phone, set the camera height in the calibration sheet (default rig: handlebar mount), and complete the hold-still calibration.

For the desktop replay harness: build `tests/cpp` under MSYS2/MinGW and point it at a recorded session directory; CI runs the same replay scoring on every push (`.github/workflows/replay.yml`). Appendix C gives the step-by-step validation procedures.

## 7.7 Information Security and Privacy

All camera frames, IMU data, and position estimates are processed and stay on-device; nothing is uploaded, and there is no backend to upload to. Session recordings live under app-private storage and are user-deletable. Network access is limited to map-data acquisition (Overpass road tiles, optional online routing), never sensor data; the permission set is exactly camera + location, each requested with its rationale. One development aid is worth noting honestly: the engine embeds a small read-only HTTP debug viewer used during development; it exposes no sensor data externally beyond the debug overlay state and is slated for removal in a store-distributed build.

## 7.8 Risk Management

Table 7.2: Risk register, likelihood, impact, and the shipped mitigation for each identified risk.

| Risk                          | Likelihood | Impact | Mitigation (as built)                                                         |
|-------------------------------|------------|--------|-------------------------------------------------------------------------------|
| IMU drift / calibration error | High       | High   | EKF bias states + ZUPT + hold-still calibration + persisted calibration store |
| Poor lighting / motion blur   | High       | Medium | Blur gate + CLAHE + inertial bridge + zero-witness standstill lock            |
| Monocular scale error         | Medium     | High   | Ground-plane geometric scale (calibrated $h$ ) + per-                         |

| Risk                             | Likelihood | Impact | Mitigation (as built)                                                                           |
|----------------------------------|------------|--------|-------------------------------------------------------------------------------------------------|
|                                  |            |        | gait K + depth-weighted channel                                                                 |
| Long-run drift without GNSS      | High       | High   | HMM map matching as external anchor + bounded map→VIO feedback legs                             |
| Performance bottleneck on device | Medium     | High   | Native C++ core, zero-copy JNI, background VIO thread; measured median 15.4 ms/frame            |
| Battery drain                    | Medium     | Medium | Power trim of non-contributing subsystems (~50 MB RAM + per-frame work saved), lazy diagnostics |
| Permission denial                | Low        | High   | Rationale-first permission flow, graceful retry                                                 |
| Wrong-map data (OSM errors)      | Low        | Medium | Bounded corrections; off-road gaps restart the trellis instead of forcing a match               |
| GPS jamming (environment)        | Certain    | ,      | The product premise: no GNSS dependence; GPS used only as a health-verified reference           |

Each mitigation in the right-hand column is a shipped mechanism referenced elsewhere in this book, not an intention, the table doubles as an index into §7.1 and Chapter 8.

## 7.9 Exceptions Management

Permission denial routes to an explanatory retry screen. Camera interruption (backgrounding, screen off) pauses ingestion and resumes cleanly, preserving session state. Sensor anomalies and native faults are caught at the JNI boundary; a crash logger persists native stack traces for

post-mortem. Tracking loss triggers staged recovery (inertial bridging, then ORB relocation, then re-initialization) with the health chip reporting each stage. Map-data failures degrade gracefully: a missing or corrupt asset disables map matching and leaves raw-VIO tracking functional rather than crashing (the loader returns a failure result by contract, it never throws).

## 7.10 Version Control

Git on GitHub. Work flowed through feature branches into the team’s integration branch and then master; every behavior-affecting change was validated against recorded rides in the replay harness before merging, and regressions were hunted by *bisection over engine versions on identical replayed input* rather than by tuning, several fixes in Chapter 8’s log cite the exact commits this process produced.

## 7.11 Project Management

Table 7.3: Project milestones with signed evidence.

| Milestone                  | Date           | Evidence                           |
|----------------------------|----------------|------------------------------------|
| Project proposal           | Autumn 2025    | Proposal document                  |
| SRS signed                 | 16/11/2025     | Signed SRS                         |
| SDD signed                 | 02/02/2026     | Signed SDD                         |
| Proof of Concept presented | Winter 2025–26 | POC presentation + results         |
| Alpha approved             | 09/04/2026     | Signed Alpha form                  |
| Beta approved              | June 2026      | Signed Beta form + demo videos     |
| V&V report approved        | 22/06/2026     | Signed V&V report (build v1.0-osm) |
| Project book               | July 2026      | This document                      |

**Development narrative.** The project ran in the phases the milestones above mark, each with a distinct character. The *specification* phase (proposal → SRS → SDD) fixed the four-tier architecture and the 200 ms budget that every later decision was measured against. The *proof-of-concept* phase answered the feasibility question (§8.1) and, in doing so, exposed the two gaps, drift and monocular scale, that defined the rest of the work. The long *engine* phase built and repeatedly corrected the VIO core; this is the phase the bug-resolution log (§8.7) chronicles, and it is where the replay harness earned its keep, turning what would have been guesswork into bisection. The *productization* phase, roughly Alpha (April) through Beta (June), shifted the centre of gravity from “does it track?” to “does it track *on a road*”: it is when the online map services were replaced by the offline OSM stack, when the per-gait scale calibration replaced a

single shared factor, and when the power trim disabled the subsystems measurement showed were dead weight. The final *validation* phase produced the supervisor-approved V&V report this chapter and Chapter 8 draw on. Figure 7.6 lays these phases on a timeline.

## NavSight: project phases and signed milestones

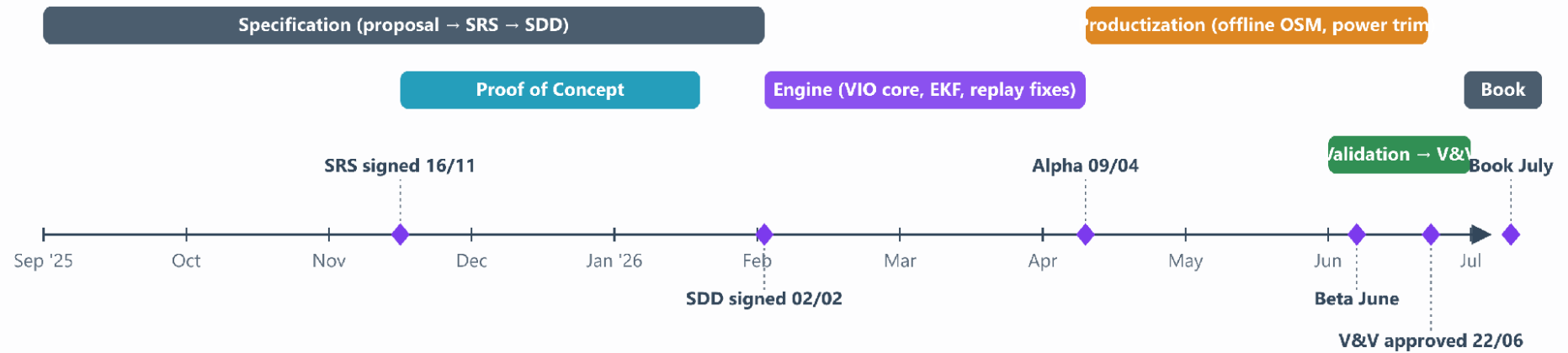


Figure 7.6: Project timeline: specification, proof of concept, engine, productization, and validation phases against the signed milestones.

**Team roles (as built):** Morad Zubidat, engine: VIO core, EKF/sensor fusion, calibration, integration testing. Roey Ben Harush, application and UI: Compose interface, camera pipeline, UX. Tamir Sobuh, map matching: OSM data layer, matcher, routing. Shared: architecture decisions, field validation, documentation. The division follows the architecture (§5.1): one owner per tier boundary, with the JNI contract and the matcher tick as the negotiated interfaces between them. Weekly supervisor syncs; code review on integration; validation rides as a team activity; the rider, by rotation, was also the test operator.

## 7.12 Code

The complete source (Android app, C++ core, asset pipeline, replay harness, and the validation-report generator that produced Chapter 8's figures) is maintained in the project's Git repository, together with the recorded validation telemetry under `tests/sims/`. Appendix B catalogs the datasets; Appendix E documents the JNI surface.

**Repository:** <https://github.com/morad-z/NavSight1>

The repository is shared with the supervisors; read access covers the full history, including the commits cited in the bug-resolution log of §8.7.

# 8 System Validation

NavSight was validated in two stages: an early **proof of concept** that established feasibility before the full system existed (§8.1), and the **full system validation** that measured the finished product against its requirements (§8.2 onward). This chapter covers both: the environment and methodology (§8.2), the executed test cases (§8.3), performance (§8.4), the accuracy experiments (§8.5), traceability back to Chapter 4’s requirements (§8.6), the engineering evolution behind the results (§8.7), a real-world case study under active jamming (§8.8), and conclusions (§8.9). The numeric results and figures come from the archived ride telemetry and are regenerated by a single script (`scripts/make_validation_report_pdf.py`); the Route A case-study views (Figures 8.7–8.10) are archived captures cross-checked against the map-matching distance report. The validated build is **v1.0-osm**, **git 7b333ac**, and the underlying V&V report was reviewed and approved by the supervisor on 22/06/2026.

## 8.1 Proof of Concept: Early Feasibility

Before committing to the full four-tier system, the team built a proof of concept whose sole purpose was to answer the project’s riskiest open question: *can a phone’s camera and IMU, fused on-device, produce a usable trajectory in real time at all?* Everything else (map matching, the offline OSM stack, the polished UI, the error-state EKF’s calibration states) was deferred until this feasibility was demonstrated.

**Objective.** Demonstrate accurate movement tracking with the device held parallel to the ground (the handlebar-mount orientation), displaying the trajectory on-screen in real time.

**Test scenario.** The device was held horizontally while the operator walked forward, backward, left, and right. The system interpreted the camera feed and IMU data to compute relative movement and drew the resulting path live.

**Pipeline under test.** The proof of concept exercised the classical monocular VIO front-end that survives, largely unchanged, in the shipped engine (§7.1.1): Shi-Tomasi feature extraction to find high-contrast corners, Lucas–Kanade optical flow to compute each anchor point’s inter-frame velocity, and RANSAC to statistically reject outlier correspondences, the “VIO pipeline” that all later work was built on top of.

**Success criteria.** Correct movement-direction detection; real-time path updates; and tracking continuity during normal walking, with no manual re-initialization.

**Results.** The proof of concept successfully demonstrated VIO tracking with the phone in the horizontal orientation. The on-screen visualization correctly reflected movement direction in all four walked directions, and the path updated in real time without stalling, validating

implementation feasibility and clearing the way for the full system. Two findings from this stage shaped everything after it: first, that the front-end held up at frame rate on the target hardware (the seed of the 15.4 ms median measured later, §8.4); and second, that raw monocular VIO alone drifts and carries no metric scale, the twin gaps that motivated the ground-plane speed estimator (§7.1.4) and the map-matching anchor (§7.1.6). The proof of concept proved the core was possible; the rest of the project made it accurate.

## 8.2 Validation Methodology and Test Environment

Table 8.1: The validation environment.

| Item        | Detail                                                         |
|-------------|----------------------------------------------------------------|
| Device      | Samsung Galaxy S21 Ultra (SM-G998B), Exynos 2100, Mali-G78     |
| OS / build  | Android 15 (API 35) · NavSight v1.0-osm, git 7b333ac           |
| Native core | C++17 / NDK, OpenCV 4.5.3, error-state EKF                     |
| Test area   | Haifa urban roads (e-scooter) + indoor walks                   |
| Telemetry   | Per-frame JSON: pose, IMU, GPS, speed channels, event counters |

**Reference strategy (the defining constraint.** Validation ran under *real, ambient GNSS jamming*) the exact environment NavSight is built for, which makes GPS itself unusable as ground truth. Two independent reference types were therefore used: **map-measured route distances** (Google Maps distance tool, jamming-immune), and **health-verified GPS**, used only on rides whose GPS passed per-ride screening (median fix accuracy 4 m, no freezes or jump artifacts). On rides that failed screening, GPS was observed to freeze for 61 s while moving and to inflate a measured route by +33%, quantified in §8.8.

**Instruments.** (1) *Deterministic offline replay*: recorded frames + IMU re-run through the unmodified native engine, enabling controlled A/B comparison of algorithm versions on byte-identical input. (2) *Automated suites*: 74 Kotlin unit tests, a C++ unit suite, and CI replay scoring on every change (.github/workflows/replay.yml). (3) *Event counters*: ~180 in-engine counters serialized into every session’s telemetry, making internal behavior auditable after the fact.

**Error metrics.** The two headline metrics are the distance ratio and the windowed speed RMSE, defined as:

$$\rho = D_{\text{nav}} / D_{\text{ref}} \text{ (travelled-distance ratio)}$$

$$\text{RMSE}_v = \sqrt{(\text{mean}_w (\bar{v}_{\text{nav},w} - \bar{v}_{\text{ref},w})^2)} \text{ (over aligned 5 s windows } w)$$

$$\sigma_{v,\text{ref}} \approx \sqrt{2} \cdot \sigma_{\text{pos}} / \Delta t \approx 5.7 \text{ m/s (GPS speed-reference noise floor)}$$

GPS speed references are conditioned before comparison in two steps: samples whose implied acceleration exceeds physical capability against both neighbours ( $|dv/dt| > 3 \text{ m/s}^2$ ) are rejected as position-jitter artifacts (22% of samples on the primary ride), and both signals are averaged over the same non-overlapping 5 s windows. The third equation above is the reason the conditioning is unavoidable: differencing two 4 m position fixes one second apart produces a speed sample with  $\sim 5.7 \text{ m/s}$  ( $\approx 20 \text{ km/h}$ ) of noise, so even after window-averaging a  $\sim 4\text{--}6 \text{ km/h}$  reference floor remains, a property of consumer GNSS, not of the estimator, and the bound on any measurable speed agreement (which is why the 8.9 km/h RMSE of §8.3 is reported *at* that floor, not above it).

### 8.3 Functional Validation: Executed Test Cases

Thirteen system-level test cases were executed on the validated build, spanning the tracking core (T1–T4), the map layer (T5–T6), degradation and recovery (T7), operations (T8–T12), and the regression platform itself (T13). Each case’s evidence column cites the measured artifact behind the verdict (a figure, a counter, or a telemetry value) so every PASS is auditable from the archived data. All 13 **PASS**.

Table 8.2: The thirteen executed system-level test cases with their measured evidence.

| ID | Case                                      | Result | Evidence                                                                                                                                |
|----|-------------------------------------------|--------|-----------------------------------------------------------------------------------------------------------------------------------------|
| T1 | GPS-denied live tracking (jamming active) | PASS   | Continuous pose/path with zero GNSS dependence, incl. through 61 s GPS-frozen segments                                                  |
| T2 | Speed accuracy, cruise                    | PASS   | $\rho = 0.91$ ; RMSE 8.9 km/h at the reference’s own 4–6 km/h floor; bias $-1.9 \text{ km/h}$ ; medians 36.3 vs 35.1 km/h (within 3.5%) |

| ID | Case                            | Result | Evidence                                                                                                                                                                |
|----|---------------------------------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| T3 | Speed accuracy, standstill      | PASS   | Displayed 0 km/h throughout; estimate never exceeded 0.9 km/h, with 27 frames locked at exactly 0.0 <sup>1</sup>                                                        |
| T4 | Distance vs map-measured route  | PASS   | 1,195 / 1,280 m = 93.4% on route A, ridden under active jamming                                                                                                         |
| T5 | On-road constraint              | PASS   | Displayed position is graph-constrained; it cannot leave the road network or jump across the map (structural, verified by the rail-ball trajectory and logged counters) |
| T6 | Roundabout entry/traversal/exit | PASS   | Session started on a roundabout: detected at t = 0, clean exit at t = 9 s, no spurious re-entry                                                                         |
| T7 | Fallback & recovery             | PASS   | 36 inertial fallbacks on the 2026-06-04 ride; +24% turn distance from the inertial bridge in offline A/B; 4 ORB relocalizations                                         |
| T8 | Calibration flow                | PASS   | Chip states + calibration sheet verified on-device; GPS auto-height hardened (Doppler-vs-position cross-check)                                                          |

| ID  | Case                          | Result | Evidence                                                                                                                                             |
|-----|-------------------------------|--------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| T9  | Real-time budget              | PASS   | Median 15.4 ms tracking (13× under the 200 ms budget), worst frame 78 ms                                                                             |
| T10 | Life-cycle & UI on Android 15 | PASS   | All seven life-cycle states exercised; no crashes across validation days                                                                             |
| T11 | Heading correctness           | PASS   | Compass-validated against NOAA declination reference; road-tangent correction and backwards-start detection verified                                 |
| T12 | Power-aware operation         | PASS   | Non-contributing subsystems disabled behind documented flags (~50 MB RAM + per-frame work saved); lazy diagnostics; verified by init logs + counters |
| T13 | Regression platform           | PASS   | 74/74 Kotlin tests; C++ suite; deterministic replay + CI scoring on fixtures                                                                         |

<sup>1</sup> The source V&V report's test table printed 50 locked frames while its Figure 4 caption said 27; regenerating the figure from the archived telemetry for this book confirms **27**; the figure caption was correct, and the value is used here.

Three of the thirteen deserve elaboration beyond their table rows:

**T5/T6 (the map layer under stress.** T5's claim is architectural, not statistical: the displayed position *cannot* leave the road network, because the rail ball advances along graph edges by construction) the on-road guarantee needs no correction to hold. Of the map→VIO feedback legs (§7.1.6), the heading hint and the tightly gated position-correction leg are live in the validated build (but the position leg acts only on the *raw* VIO trail, never on the displayed ball)

while the matcher-supervised re-acquisition ships disabled; the passing result therefore rests on the structural constraint, independent of any correction. T6 chose the adversarial starting condition on purpose, a session beginning *on* a roundabout, the geometry that had produced the project's worst historical failure (§8.7): the state machine detected the roundabout at  $t = 0$ , tracked traversal, exited cleanly at  $t = 9$  s, and the re-entry hysteresis held.

**T7, degradation as a first-class behavior.** The 2026-06-04 ride was selected for its hostile lighting; the inertial bridge engaged 36 times and the trajectory survived every dropout. The +24% figure beside it comes not from that ride but from the controlled replay A/B (Figure 8.6), the two together separate “it worked in the field” from “this mechanism is why.”

**T13, the platform that validates the platform.** The 74 Kotlin unit tests cover the map stack's pure logic (matcher scoring, geometry, routing readers, maneuver state machine); the C++ suite covers the engine's math (filter updates, preintegration, gating); and the CI replay workflow re-scores recorded fixtures on every push, so an algorithmic regression fails a build before it ever reaches a ride. It is this layering that made the fix history of §8.7 safe to accumulate.

### 8.3.1 Detailed Test-Case Specifications

Each executed case is specified below with its traced requirement(s), objective, setup, procedure, expected result, and the measured actual result. All were run on the validated build (v1.0-osm, git 7b333ac) on the reference device.

**T1, GPS-denied live tracking.** *Requirements:* FR-1, FR-5, R1/R3. *Objective:* confirm continuous positioning with no GNSS dependence whatsoever. *Setup:* Samsung Galaxy S21 Ultra, Haifa urban rides under active ambient jamming; GPS logged for reference only, never fed to the positioning solution. *Procedure:* start a session, ride the full route, and observe pose and on-map path continuity, including intervals where the GPS receiver froze. *Expected:* an unbroken pose stream and on-road path, independent of GNSS state. *Actual:* continuous pose and path across every recorded ride, including a 61 s segment during which the GPS position was frozen while the vehicle moved. **PASS.**

**T2, Speed accuracy, cruise.** *Requirements:* FR-3, NFR-2, R6. *Objective:* quantify moving-speed agreement against a health-verified reference. *Setup:* the 2026-06-12 ride (~99 s) whose GPS passed per-ride health screening (93 fixes, median accuracy 4 m). *Procedure:* compare NavSight speed to the conditioned GPS reference over aligned, non-overlapping 5 s windows; reject reference samples implying  $|dv/dt| > 3 \text{ m/s}^2$ . *Expected:* agreement at the reference's own resolution limit, without systematic bias. *Actual:* distance ratio  $\rho = 0.91$ ; windowed RMSE 8.9

km/h (against a 4–6 km/h reference floor); bias –1.9 km/h; median moving speed 36.3 vs 35.1 km/h, within 3.5% (Figure 8.2). **PASS.**

**T3, Speed accuracy, standstill.** *Requirements:* NFR-2.4, R6. *Objective:* confirm a true standstill reads exactly 0 km/h with no phantom creep. *Setup:* the end-of-ride stop on the 2026-06-12 ride. *Procedure:* hold the vehicle stationary; record the displayed speed and the per-frame estimate through the stop. *Expected:* displayed 0 km/h; estimate bounded below any perceptible speed. *Actual:* the estimate never exceeded 0.9 km/h and locked at exactly 0.0 for 27 frames; the display read 0 km/h throughout the stop (Figure 8.3). **PASS.**

**T4, Travelled distance vs. a map-measured route.** *Requirements:* FR-9, NFR-2, R6. *Objective:* validate integrated distance against a jamming-immune reference. *Setup:* Route A (2026-06-03), a 1,280 m out-and-back with a U-turn, ridden under active jamming; ground truth measured with the Google Maps distance tool. *Procedure:* ride the route; integrate NavSight distance; compare to the map-measured length. *Expected:* integrated distance within the  $\leq 10\%$  drift target. *Actual:* 1,195 m integrated vs 1,280 m measured = 93.4% (7% error); the jammed GPS on the same ride accumulated 1,705 m, +33% (Figures 8.7–8.9). **PASS.**

**T5, On-road constraint.** *Requirements:* FR-9.1, FR-9.3, R9. *Objective:* confirm the displayed position cannot leave the road network or jump across the map. *Setup:* all map-matched rides. *Procedure:* inspect the rail-ball trajectory and the logged counters for any off-road excursion or unbounded jump. *Expected:* the ball stays on graph edges at all times; no teleport. *Actual:* no off-road excursion and no jump observed; the constraint is structural (the ball advances only along graph edges, independent of the gated feedback legs of §7.1.6, which act on the raw VIO trail and not the displayed ball). **PASS.**

**T6, Roundabout entry, traversal, and exit.** *Requirements:* FR-9.5, R9. *Objective:* validate explicit roundabout handling, including the worst starting condition. *Setup:* a session begun on a roundabout, the geometry behind the project's worst historical failure (§8.7). *Procedure:* start on the roundabout, traverse, and exit; observe detection, traversal state, exit, and any spurious re-entry. *Expected:* clean detection, traversal, exit, and no spurious re-entry. *Actual:* detected at  $t = 0$ , traversal tracked, clean exit at  $t = 9$  s, no re-entry, the re-entry hysteresis held. **PASS.**

**T7, Degraded-mode fallback and recovery.** *Requirements:* FR-2.2, FR-7.3, R4/R5. *Objective:* confirm automatic bridging through visual dropouts and recovery afterwards, and quantify the bridge's contribution. *Setup:* the 2026-06-04 ride (hostile lighting); plus an offline replay A/B on identical input. *Procedure:* ride through degraded lighting; count inertial fallbacks and ORB relocalizations; separately, run the recording through the pre- and post-bridge engine versions. *Expected:* automatic bridging with no frozen position; measurable

bridge benefit. *Actual*: 36 inertial fallbacks and 4 ORB relocalizations on the ride; the controlled A/B showed +24% integrated turn distance from the bridge (Figure 8.6). **PASS.**

**T8, Calibration flow.** *Requirements*: FR-6, R7. *Objective*: validate the calibration states, the calibration sheet, and the GPS-assisted mount-height path. *Setup*: on-device sessions. *Procedure*: perform the hold-still calibration; observe chip-state progression and the calibration sheet; exercise the Doppler-vs-position cross-check for auto-height. *Expected*: correct state progression to Ready; hardened auto-height. *Actual*: chip states and calibration sheet verified on-device; GPS auto-height hardened with the cross-check. **PASS.**

**T9, Real-time budget.** *Requirements*: NFR-1, R2. *Objective*: confirm per-frame processing fits the 200 ms budget with margin. *Setup*: on-device per-frame tracking timings (n = 98). *Procedure*: record feature-tracking time per frame; compare to the 43 ms frame interval and the 200 ms budget. *Expected*: median well under budget; no frame exceeding it. *Actual*: median 15.4 ms (13× under budget), maximum 78.2 ms (2.5× margin at the native ~23 fps), no frame over 200 ms (Figure 8.1). **PASS.**

**T10, Life-cycle and UI on Android 15.** *Requirements*: FR-4, FR-8, NFR-3, R8/R11. *Objective*: exercise all seven life-cycle states and the redesigned UI without failure. *Setup*: the reference device on Android 15 across validation days. *Procedure*: drive the app through init/permissions, calibrating, ready, tracking, degraded, recovered, and stopped/reset; capture on-device screenshots. *Expected*: all states reachable; no crashes. *Actual*: all seven states exercised; UI validated by on-device screenshots (Figure 7.4); no crashes across validation days. **PASS.**

**T11, Heading correctness.** *Requirements*: FR-3, R10. *Objective*: validate heading against an external reference and confirm the road-tangent correction and backwards-start detector. *Setup*: a heading-audit session (2026-05-25) referenced to NOAA magnetic declination. *Procedure*: compare Madgwick heading to the declination-corrected reference; verify the road-tangent nudge and the backwards-start detector. *Expected*: heading tracks the reference; correction and detector behave as specified. *Actual*: compass-validated against the NOAA reference; road-tangent correction and backwards-start detection verified. **PASS.**

**T12, Power-aware operation.** *Requirements*: NFR-1, R13. *Objective*: confirm non-contributing subsystems are disabled and diagnostics are lazy. *Setup*: init logs and event counters on the validated build. *Procedure*: verify loop closure, live SLAM, landmark mapping, windowed BA, and VI-scale estimation are gated off; confirm camera-screen diagnostics are lazy-computed. *Expected*: the subsystems inactive; ~50 MB RAM and their per-frame work saved. *Actual*: all confirmed disabled behind documented flags; savings verified by init logs and counters. **PASS.**

**T13, Regression platform.** *Requirements:* NFR-5, R14. *Objective:* confirm the automated regression net is complete and green. *Setup:* the CI environment plus the local suites. *Procedure:* run the Kotlin unit tests, the C++ unit suite, and the deterministic replay scoring on fixtures. *Expected:* all suites pass; replay scoring stable. *Actual:* 74/74 Kotlin tests pass; C++ suite passes; deterministic replay and CI fixture scoring green. **PASS.**

## 8.4 Performance Evaluation

**Real-time budget.** On-device per-frame feature-tracking cost over an  $n = 98$  sample: **median 15.4 ms, maximum 78.2 ms**, against a camera frame interval of 43 ms and the 200 ms budget of NFR-1.1. The median sits  $13\times$  under budget; even the single worst frame retains a  $2.5\times$  margin.

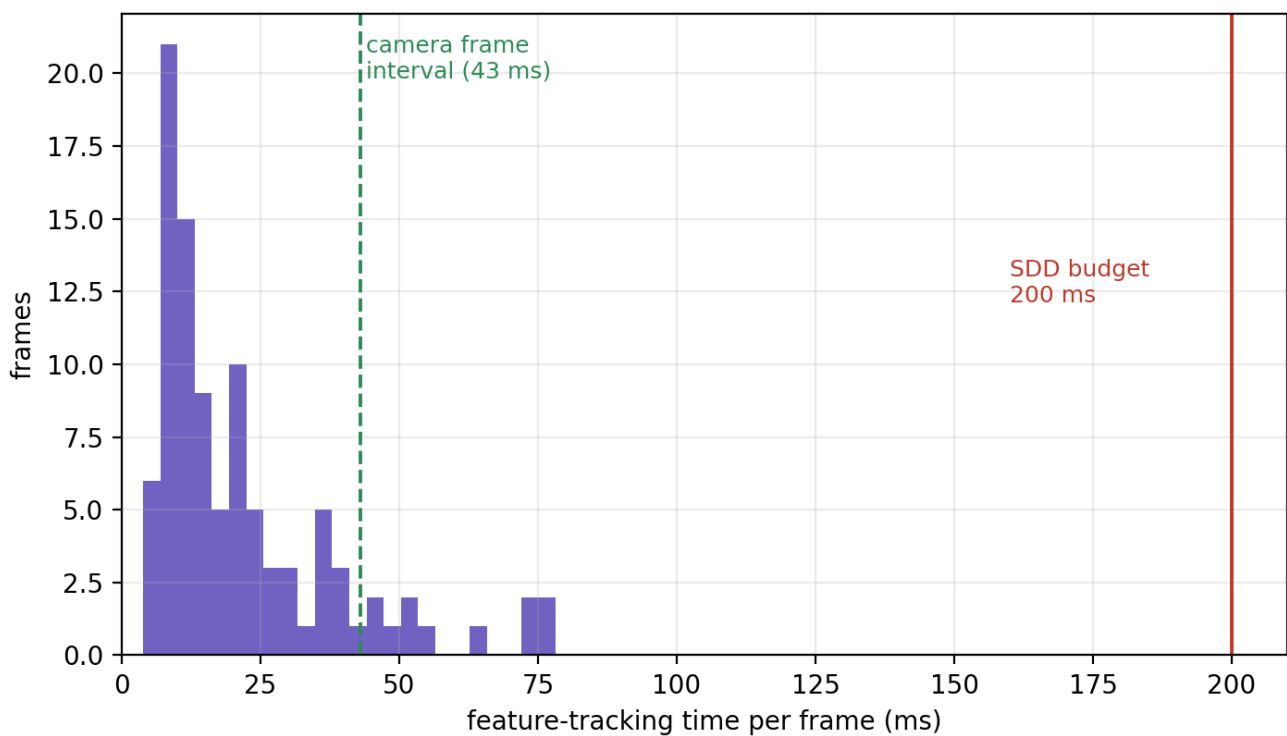


Figure 8.1: Per-frame feature-tracking time on-device ( $n = 98$ ) against the 43 ms camera frame interval and the 200 ms SDD budget; the bulk of the distribution sits well below the frame interval itself.

**Power-aware operation.** Measurement on a validation ride showed several implemented subsystems producing work that no user-facing output consumed; they were disabled behind documented compile-time flags (§8.7), saving  $\sim 50$  MB of RAM and their per-frame compute. Camera-screen diagnostics are lazy-computed. Verified by init logs and event counters (T12).

## 8.5 Accuracy Experiments

**Primary ride (2026-06-12,  $\sim 99$  s urban route, Haifa).** Speed tracks the health-verified GPS reference across the 20–45 km/h band; integrated distance **792 m vs 869 m GPS**  $\rightarrow \rho =$

**0.91.** Windowed speed agreement: **RMSE 8.9 km/h**, at the conditioned reference's own noise floor, with **bias -1.9 km/h**; median moving speed **36.3 km/h vs 35.1 km/h (within 3.5%)**. During the end-of-ride standstill the estimate never exceeded 0.9 km/h, with the display locked at 0 km/h.

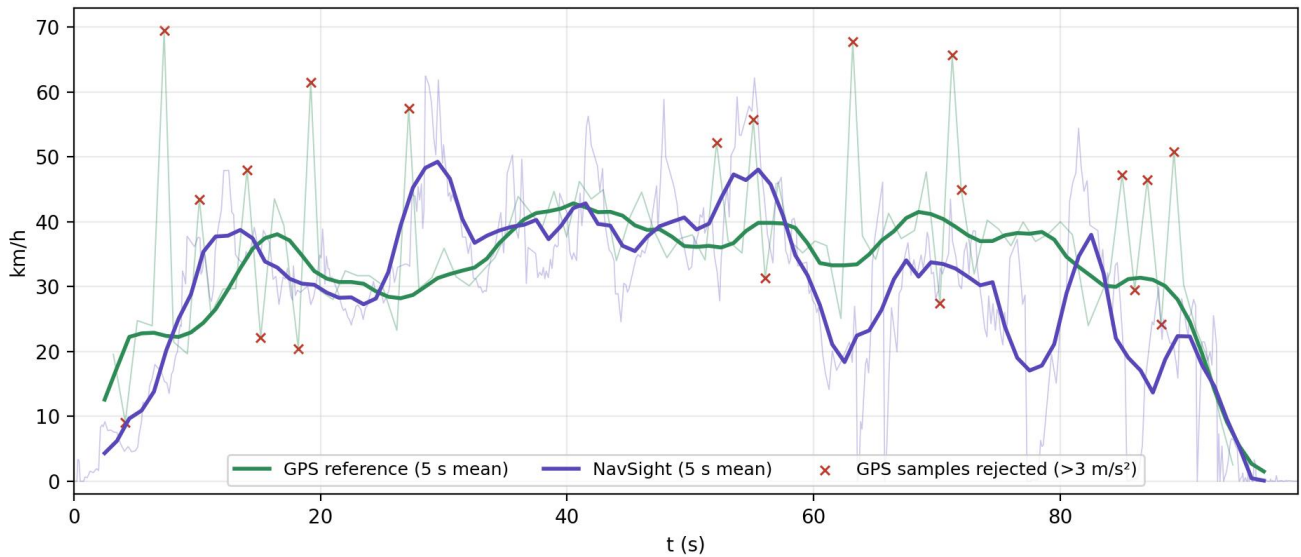


Figure 8.2: Speed vs time on the primary ride. Thin lines are the raw 1 Hz / per-frame signals, bold lines the 5 s means; red × marks the 20 of 91 GPS reference samples rejected as physically implausible.

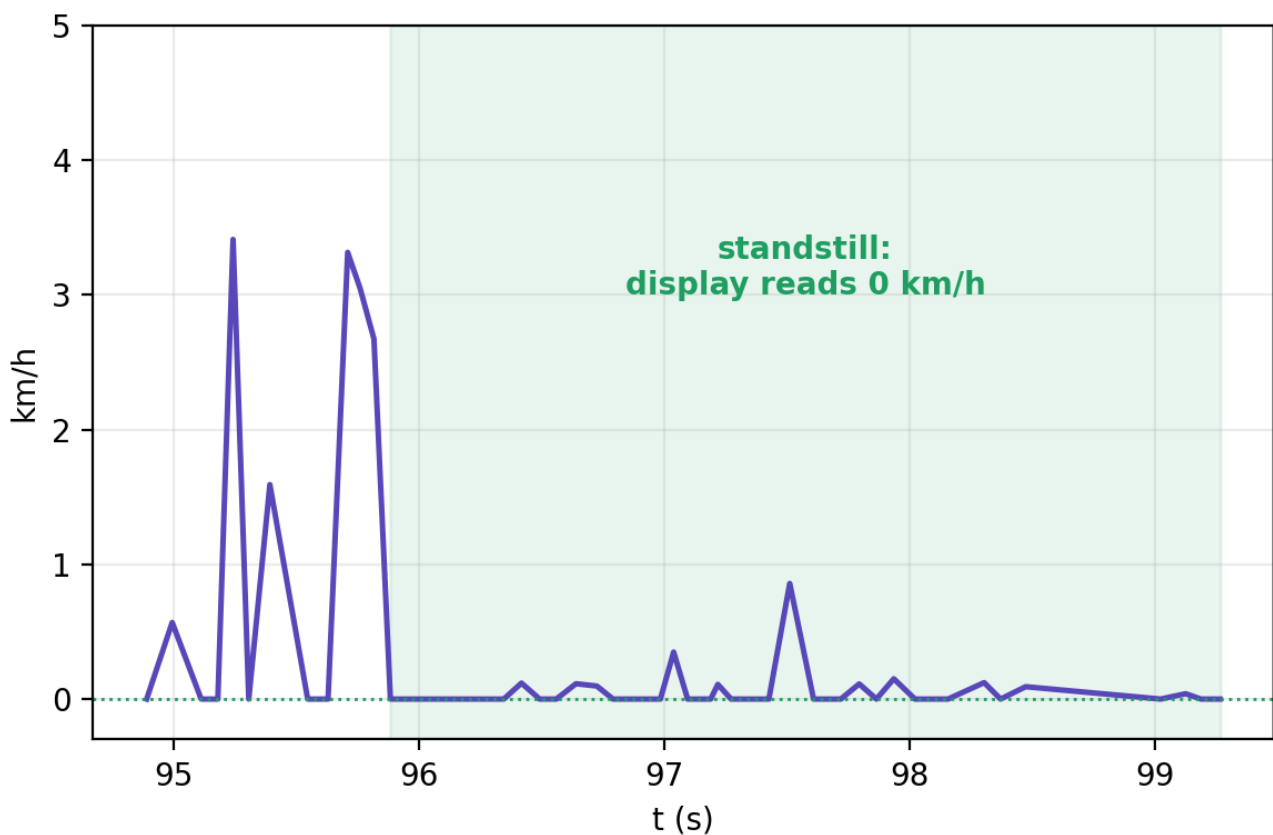


Figure 8.3: The end-of-ride standstill, zoomed to 5 km/h, the estimate never exceeds 0.9 km/h and locks at exactly 0.0 for 27 frames; the display reads 0 km/h throughout.

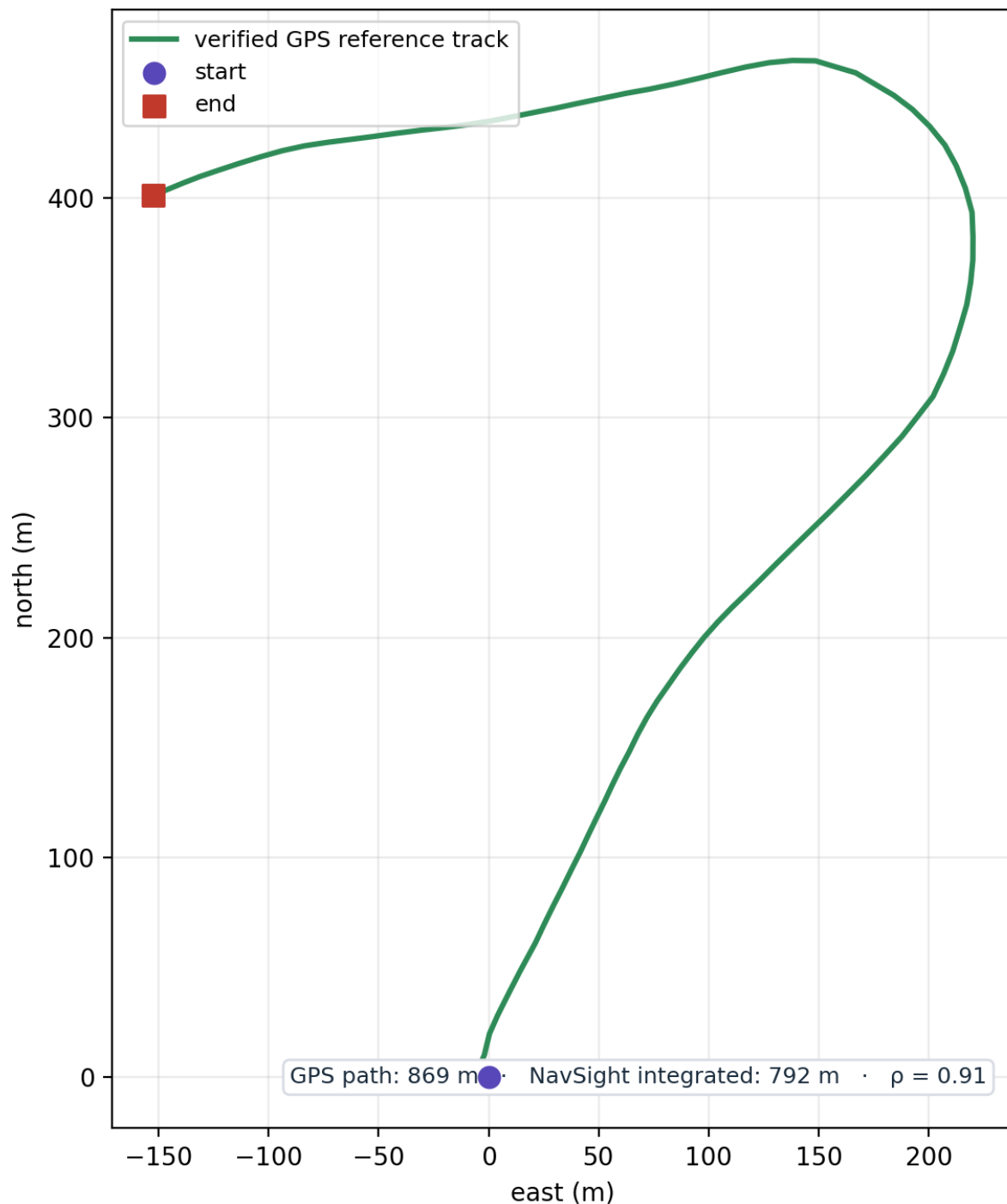


Figure 8.4: The validation route in local metric coordinates about the start point, the health-verified GPS track (93 fixes, median accuracy 4 m) that serves as the reference for Figure 8.2 and the  $\rho = 0.91$  ratio.

**Cross-route distance accuracy.** Two independently referenced routes: Route A (2026-06-03, map-measured 1,280 m): **93%**. Primary ride (2026-06-12, verified-GPS 869 m): **91%**.

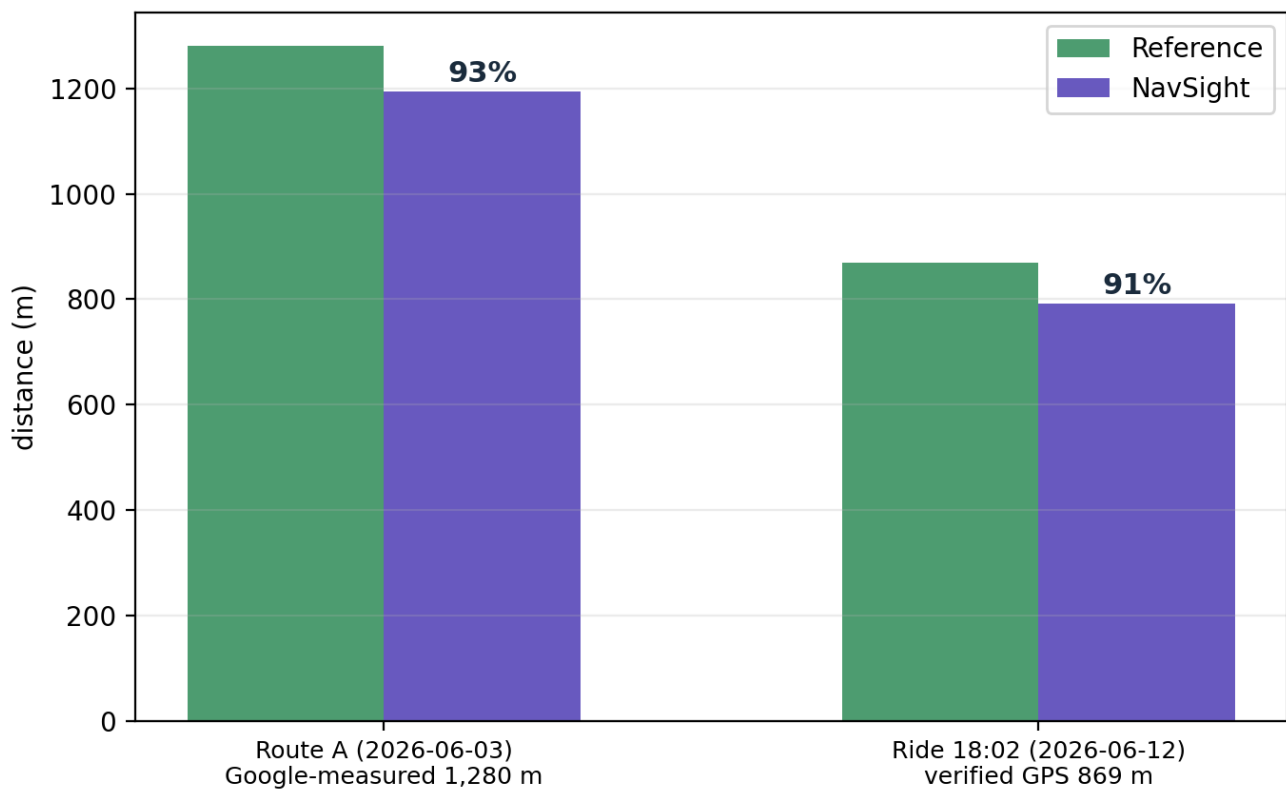


Figure 8.5: Travelled-distance accuracy on the two independently referenced routes: 93% against the map-measured route, 91% against health-verified GPS.

**Controlled A/B (offline replay).** Identical recorded input through two engine versions isolates the inertial bridge’s contribution in turns: **62 m → 77 m integrated distance (+24%)**. This is a controlled experiment, not a ride-to-ride comparison, the strongest form of evidence the platform produces.

**Reading the numbers together.** The three accuracy results measure different things and should not be conflated. The *distance ratio* ( $\rho = 0.91\text{--}0.93$ ) is the headline product metric: it answers “how far off is the trip length the rider sees?” and it improves on longer routes because per-frame errors are partly self-cancelling over distance. The *speed RMSE* (8.9 km/h) looks large only until it is read against its reference floor (§8.2): it is not the estimator’s error but the resolution limit of the GPS it is compared to, which is why the *bias* (−1.9 km/h) and the *median-speed agreement* (within 3.5%) are the more honest speed metrics; they survive the windowing that the noisy reference forces. And the *A/B delta* (+24%) is the only one of the three that isolates a single mechanism on identical input, which is why it, not any field ride, is cited when attributing the inertial bridge’s value. Reported separately, the three triangulate the same conclusion: the system is accurate where accuracy is measurable, and the places it is *not* cleanly measurable are the reference’s fault, not the estimator’s.

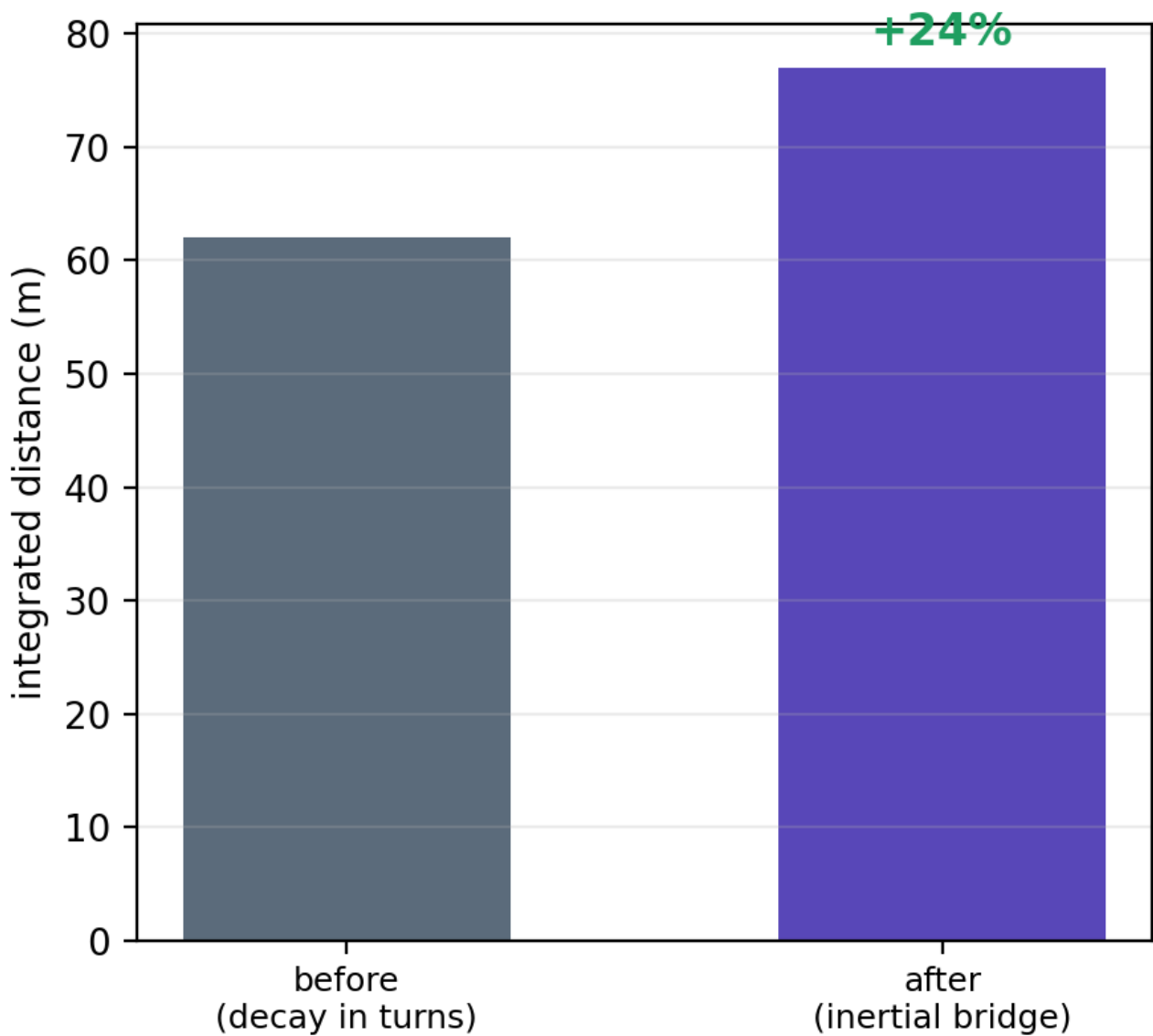


Figure 8.6: Offline A/B on identical recorded input; the inertial bridge adds +24% integrated distance through turns.

## 8.6 Requirement Coverage Matrix

Chapter 4 IDs → validation evidence:

Table 8.3: Requirement-coverage matrix, every Chapter 4 requirement traced to the test cases that validate it.

| Requirement                        | Validated by    | Status |
|------------------------------------|-----------------|--------|
| FR-1 Camera-based tracking         | T1, T9          | PASS   |
| FR-2 Inertial dead reckoning       | T7              | PASS   |
| FR-3 Sensor fusion (incl. heading) | T1, T2, T7, T11 | PASS   |

| Requirement                                     | Validated by                         | Status                                                        |
|-------------------------------------------------|--------------------------------------|---------------------------------------------------------------|
| FR-4 Path visualization & session control       | T10                                  | PASS                                                          |
| FR-5 Offline operation                          | T1 (jammed, no-network rides)        | PASS                                                          |
| FR-6 Initialization & calibration               | T8, T10                              | PASS                                                          |
| FR-7 Error handling & feedback                  | T7, T8, T10                          | PASS                                                          |
| FR-8 Session management                         | T10, T13 (telemetry replay)          | PASS                                                          |
| FR-9 Map-matched positioning                    | T4, T5, T6                           | PASS                                                          |
| NFR-1 Performance (incl. power-aware operation) | T9, T12                              | PASS (13× margin)                                             |
| NFR-2 Reliability & accuracy                    | T2–T4, T7                            | PASS (7–9% over 0.87–1.28 km vs the $\leq 10\%/200$ m target) |
| NFR-3 Compatibility                             | T10 (Android 15 / S21 Ultra)         | PASS on reference device                                      |
| NFR-4 Usability                                 | T10 (on-device UI validation)        | PASS                                                          |
| NFR-5 Maintainability                           | T13                                  | PASS                                                          |
| NFR-6 Privacy                                   | By construction (§7.7) + code review | PASS                                                          |

## 8.7 Development Evolution and Bug Resolution

The results above were not first-try outcomes. The entries below (each Issue → Resolution → Impact, several with the exact commit) are representative of the engineering process, and all were re-validated through the replay harness.

- **Heading regression, root-caused by replay bisection.** Mid-project, heading quality collapsed on validation walks. Rather than re-tune, the team bisected engine versions over identical replayed input; the regressor was an added yaw process-noise injection, reverted, and the lesson (“bisect on data, don’t tune on symptoms”) shaped all later debugging. The stable heading pipeline, gyro-primary with gated compass and gimbal-free rotation-vector extraction, landed as 4a9a212.
- **Teleport-to-start.** A shake detector meant to catch mount knocks could reset an in-progress ride, teleporting the display to the origin. Fixed in ff93962 by scoping the

detector out of active tracking. Impact: eliminated the most user-visible failure of the beta period.

- **Roundabout orbiting and re-entry.** The matcher could latch onto a roundabout ring and orbit (once for ~4 minutes at confidence 0.80), and after a legitimate exit could spuriously re-enter. Resolution: a maneuver state machine with an exit latch and re-entry hysteresis (7481ab5). Impact: T6 passes from the worst starting condition, a session beginning on the roundabout itself.
- **Standstill read as motion (IPM under-read family).** Root-caused on real frames: one-sided speed gating rectified tracker noise into phantom speed, and degraded flow under-measured cruise. Resolution: the two-sided vote / zero-witness taxonomy with a per-point  $\sigma$  floor and forward-backward flow checks (§7.1.4). Impact: T3's exact-o standstill and the honest cruise accuracy of T2.
- **One shared scale for every gait.** A single visual-scale factor could not serve walking, running, and riding (measured cold-start convergence  $0.68\times/0.53\times$  on walk/run). Resolution: cadence-classified per-gait K slots, learned and persisted per mode. Impact: consistent distance integration across motion modes.
- **Wrong fork on curves.** Per-vertex re-anchoring inside the matcher zeroed the accumulated turn signal, so gentle forks matched the wrong branch. Resolution: junction decisions from the gyro-relative heading offset against a pre-junction anchor (§7.1.6). Impact: correct branch selection through route A's forks.
- **Online matcher replaced.** The OSRM /match service (network-bound, rate-limited, coordinate-capped) was replaced by the on-device Viterbi LocalMatcher in 62b5027. Impact: matching became jamming-proof and testable offline.
- **Power trim.** Ride measurement showed loop-closure, landmark-mapping, windowed bundle adjustment, and SLAM-feature machinery consuming RAM and per-frame work with no user-facing consumer (their correcting role having been taken over by map matching). Disabled behind documented one-word flags in 7b333ac's power trim. Impact: ~50 MB RAM saved; T12.

## 8.8 Real-World Case Study: Route A Under Active Jamming

The longest validated route: a 1,280 m out-and-back urban path with a U-turn, ridden under active GNSS jamming, examined through three independent views.

1. **Ground truth.** The route measured on Google Maps' distance tool: **1.28 km**, a reference immune to jamming.
2. **NavSight's view.** The matched position over the full ride stays on the road for the entire route; after the U-turn, the return leg re-traces the same road, the graph-

constrained matcher behaving exactly as designed. Integrated distance: **1,195 m**, **93.4% of truth**.

3. **GPS's view.** The jammed GPS reference accumulated **1,705 m on the same 1,280 m route (+33%)** from fix scatter, and on other rides froze for 61 s while moving. The device's own navigation reference failed on the route where NavSight held to within 7%.

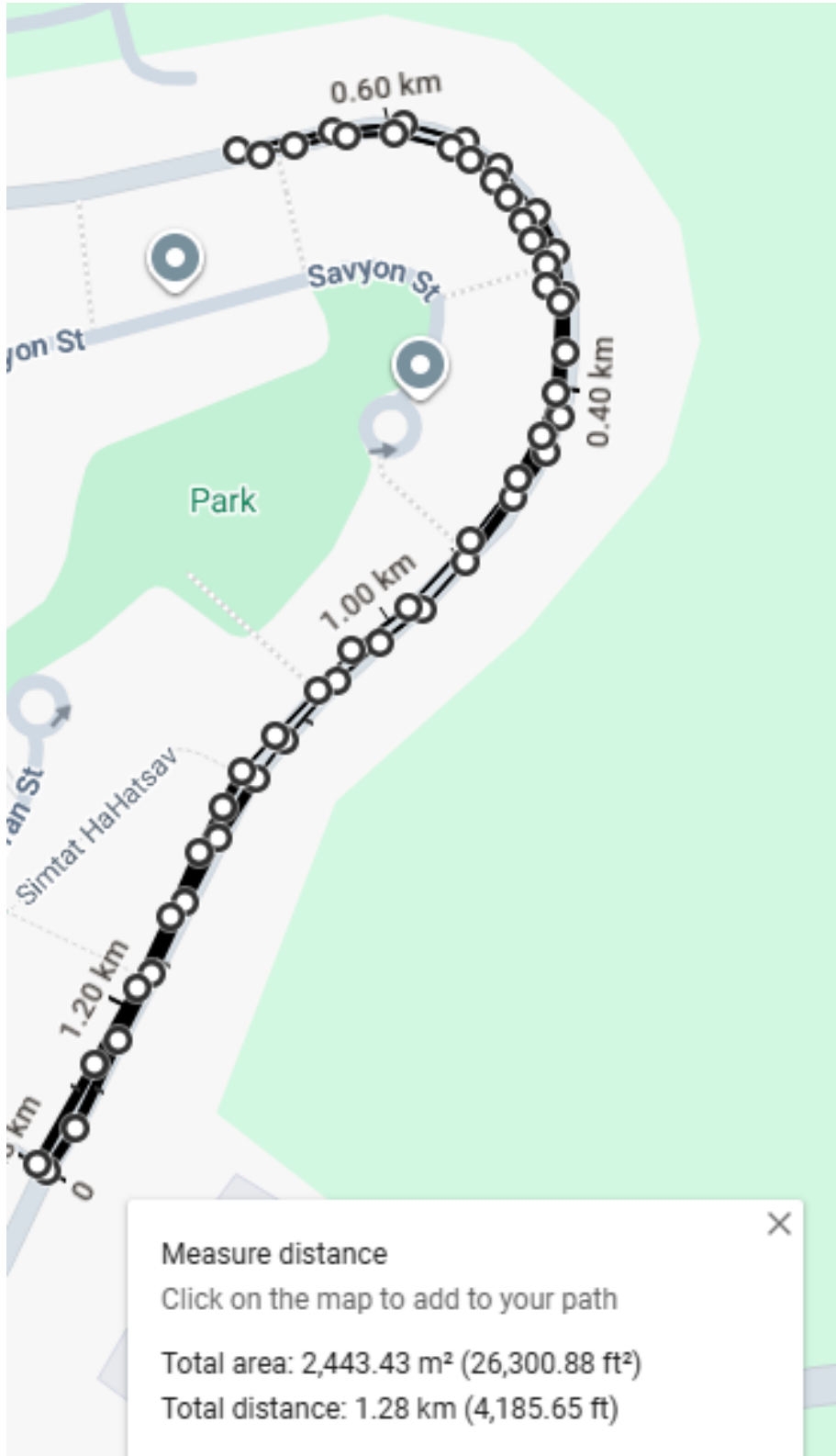


Figure 8.7: Route A ground truth. The route measured with the Google Maps distance tool (1.28 km), a jamming-immune reference.

- the U-turn re-traces the same road back

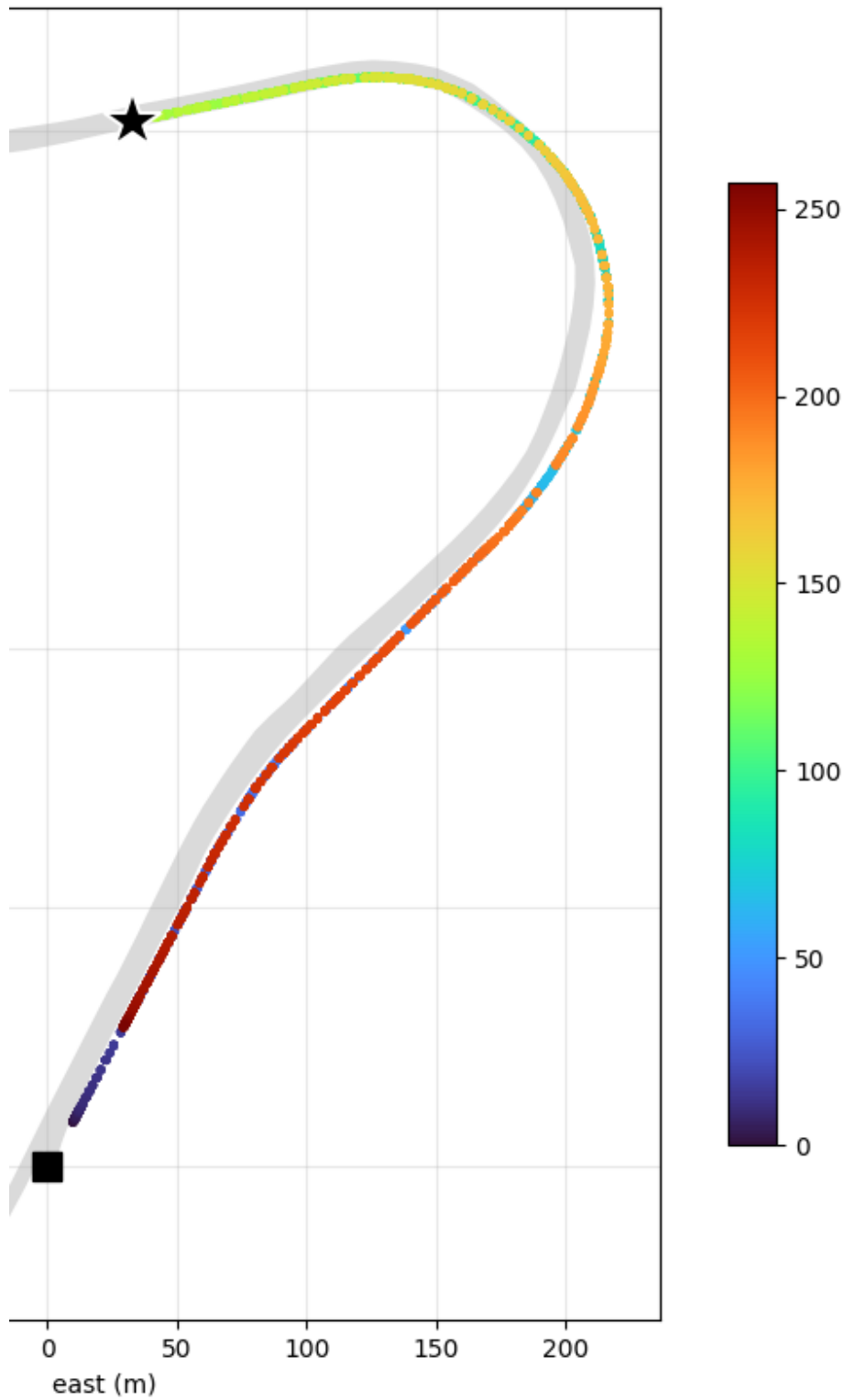


Figure 8.8: NavSight's matched position over the full Route A ride, colored by time. The track stays on the road throughout, and the return leg after the U-turn re-traces the same road.

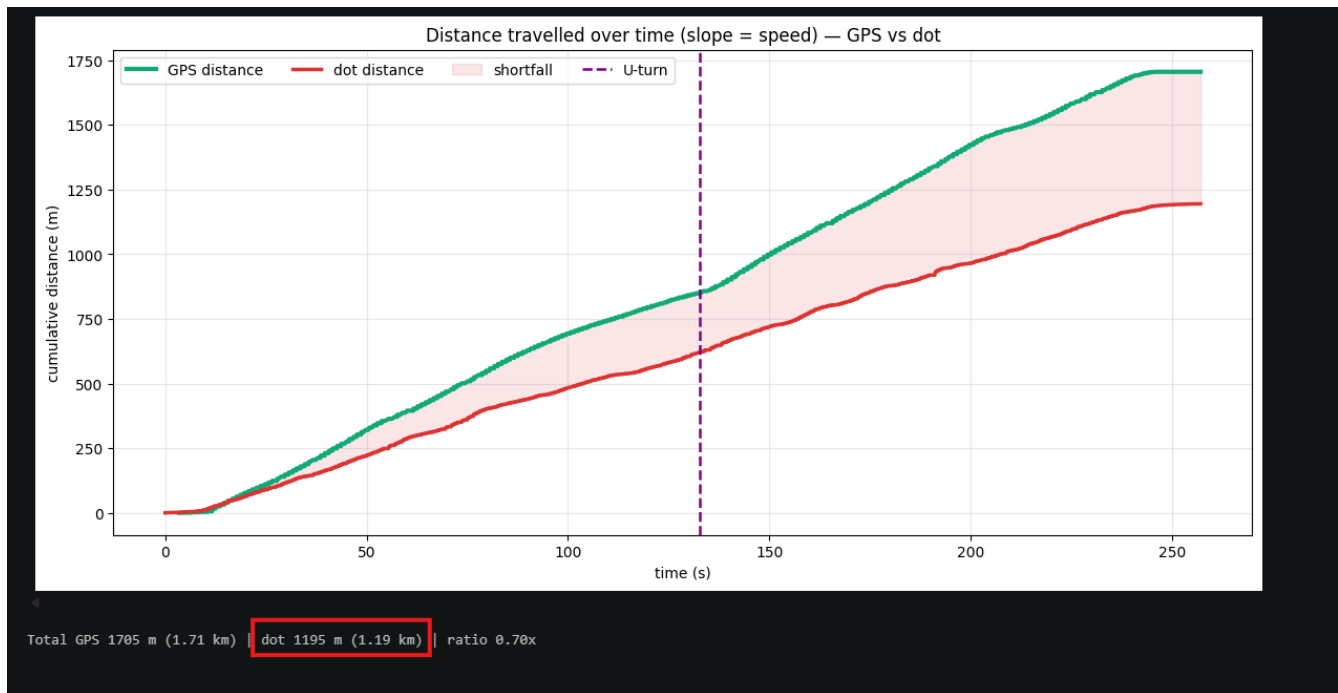


Figure 8.9: Cumulative travelled distance on Route A. The jammed GPS reference accumulates 1,705 m on a route that truly measures 1,280 m, while NavSight integrates 1,195 m.

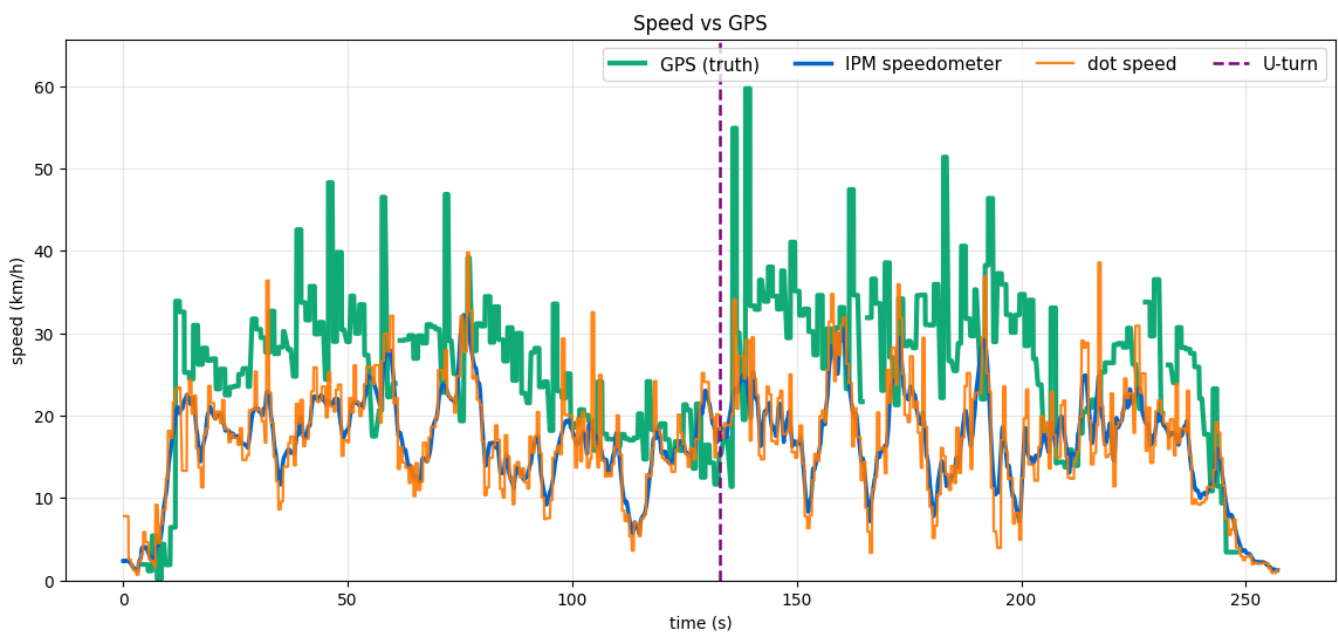


Figure 8.10: Speed channels on Route A. The ground-plane (IPM) speedometer and the raw dot speed track each other closely, while the jammed GPS trace rides visibly above both, consistent with the +33% distance inflation it accumulates over the ride.

This is the project’s thesis in one experiment: in the operating environment it was built for, NavSight is not merely a fallback for GNSS; it is more accurate than GNSS.

## 8.9 Validation Conclusion

The validated build meets the SDD scope and both measurable constraints on real hardware in the intended GPS-denied environment: **91–93% travelled-distance accuracy** against two

independent reference types, speed agreement at the conditioned reference's own resolution limit, a clean **0 km/h** standstill, a processing pipeline **13×** **under** the real-time budget, and **13/13 test cases passing**. Known limitations are stated plainly: the measurable speed agreement is bounded by the consumer-GNSS reference floor; the inertial bridge's full trust window is  $\leq 6$  s by design; the on-road guarantee for the displayed position is structural (the rail ball), with the live map→VIO feedback deliberately bounded, a straight-road heading nudge applied only short of perpendicular ( $|\Delta| < 90^\circ$ ) and  $a \leq 0.05$  m-per-frame cross-track correction to the raw trail, while the stronger matcher-supervised re-acquisition ships disabled pending on-device validation. Every numeric figure in this chapter is regenerated from the archived telemetry by the repository's analysis scripts; the live-session counter readings quoted at T7 are preserved as recorded in the signed V&V report.

# 9 Summary, Evaluation, Conclusions and Future Work

## 9.1 Summary

NavSight set out to answer a single question: can a consumer smartphone navigate a city when the satellites are gone? The delivered system answers yes. It is a native Android application with a C++ visual-inertial core: KLT visual tracking fused with preintegrated IMU data in an error-state EKF, heading from a gyro-primary Madgwick AHRS, metric speed from optical flow over the road plane at a calibrated camera height with a bounded inertial bridge, and an on-device hidden-Markov map matcher that keeps the displayed position on a locally held OpenStreetMap road graph. Every navigation-critical function runs offline, on the device, with no satellite signal and no network round-trip. It was validated on real urban rides in Haifa under active GNSS jamming (the environment it was built for, not a laboratory proxy for it) where it delivered 91–93% travelled-distance accuracy while the jammed GPS reference itself erred by +33% on the same route.

The project also delivered something less visible but equally real: a *method* for building and trusting such a system. The deterministic replay harness, the ~180 in-engine event counters, and the CI regression scoring together turned a notoriously hard-to-validate class of software, real-time sensor fusion, into something that could be measured, bisected, and defended with archived evidence rather than anecdote.

## 9.2 Evaluation and Conclusions

**Against the original targets.** Table 9.1 sets each proposal/SRS-era criterion against the delivered result. The honest reading is that the navigation-quality targets were met or exceeded, and the platform-breadth targets (iOS, formal power budgets) were deliberately traded away for depth on the one platform that was validated.

Table 9.1: Original targets versus delivered results.

| Target (proposal / SRS)             | Achieved                                                                               |
|-------------------------------------|----------------------------------------------------------------------------------------|
| Real-time: < 200 ms per frame       | 15.4 ms median, 78 ms worst, 13× margin                                                |
| Drift $\leq 10\%$ over 200 m        | 7–9% over 0.87–1.28 km routes, longer routes, better ratio                             |
| Offline, GNSS-independent operation | Positioning fully offline; search offline via bundled index; road data cached per area |
| Micromobility feasibility           | Validated on e-scooter rides under real jamming                                        |

| Target (proposal / SRS)                         | Achieved                                                                                       |
|-------------------------------------------------|------------------------------------------------------------------------------------------------|
| Cross-platform (iOS)                            | Dropped early, deliberate scope revision (\$3.1)                                               |
| CPU / RAM / battery budgets (proposal-era)      | Superseded by the V&V's measured set; formal battery profiling remains open                    |
| $\geq 5$ users, 15+ crash-free rides (proposal) | Multi-day validation, no crashes across validation days (T10); a formal user study was not run |

**Engineering conclusions worth carrying forward.** Five findings are the real intellectual output of the project; the things a successor team should inherit:

1. **The reference is part of the problem.** In a jammed environment, GPS cannot even grade the system. Map-measured routes and per-ride GPS health screening had to be built into the methodology before any accuracy number meant anything: the +33% GPS inflation on Route A is not a footnote, it is the reason the whole validation strategy exists.
2. **A road graph is a better drift anchor than a landmark map, for road vehicles.** The project implemented SLAM-style landmark mapping, loop closure, and windowed bundle adjustment, then measured them on rides: their corrections were redundant once map matching shipped (a one-way commute rarely revisits a place for loop closure to recognize), while their RAM and per-frame cost were real. They were disabled behind documented flags, an evidence-driven removal, not a guess.
3. **Two-sided evidence beats one-sided gating.** The vote / zero-witness taxonomy, requiring positive evidence for motion *and* for stillness, is what turned a noisy standstill into an exact 0 km/h. A one-sided magnitude gate would have rectified a quarter of the tracker's noise into phantom speed.
4. **Determinism is the debugging superpower.** The replay harness converted every regression hunt from guesswork into bisection, and every improvement claim into a controlled A/B on identical input. The single highest-leverage engineering decision in the project was making the C++ core byte-identical on device and desktop.
5. **Honest bounds are a feature.** The  $\leq 6$  s bridge trust, the perpendicular-backstopped heading nudge, and the  $\leq 0.05$  m-per-frame cap on the map→VIO position correction each trade coverage for the guarantee that the system never lies big, and the same conservatism kept the aggressive matcher-supervised re-acquisition disabled by default until it can be validated on device, leaning instead on the structural on-road constraint. A navigation display that admits uncertainty is safer than one that hides it, and, in a safety-relevant product, that is a design principle, not a limitation.

## 9.3 Future Work

The work below is ordered roughly by ratio of value to effort.

- **Formal power profiling.** Battery drain per 30-minute ride across devices, completing the proposal-era criterion the V&V deliberately set aside. This is measurement, not new engineering, and it closes an open acceptance criterion.
- **Wider offline coverage.** Pre-packing road regions into the APK (the asset pipeline already implements the NSSG/NSGR segment and graph formats, §7.1.7) to remove the first-visit connectivity need entirely, turning “offline where you’ve been” into “offline everywhere shipped.”
- **Multi-device calibration.** Automated per-device camera-intrinsics and mount-height calibration to move beyond the single validated reference device and make the app usable on arbitrary hardware without a manual chessboard step.
- **Exposure control.** The camera’s auto-exposure is the next measured lever on visual-dropout frequency; taking manual control of exposure in the ground-plane band should reduce the number of frames the inertial bridge has to cover.
- **Learned inertial velocity.** An RNIN-style network as an additional speed channel for pedestrian modes, where ground-plane geometry is weakest and the learned-inertial literature is strongest (§2.2), the natural complement to the geometric estimator, not a replacement for it.
- **Re-evaluating the shelved subsystems.** Loop closure, SLAM landmarks, and windowed bundle adjustment remain in the tree behind one-word compile-time flags. In GNSS-denied environments *without* a road network (open terrain, large indoor spaces) where map matching cannot anchor drift, they become the natural candidates to revisit; the disable was a decision for the road-vehicle case, not a permanent verdict.
- **Depth-model upgrade path.** Re-benchmark compact metric-depth models against the ~100 ms budget as mobile NNAPI transformer support matures (§3.4’s explicit re-open conditions); a metric depth model that fits the budget would simplify the scale-calibration pipeline.

The through-line of this list is that NavSight is a proof, not a product: it demonstrates GNSS-independent smartphone navigation is feasible and quantifies how well, and each item above widens the envelope in which that proof holds.

## 9.4 Closing Statement

The project began with a question that, at the outset, had no guaranteed answer: whether a phone alone (no satellites, no cloud, no special hardware) could tell a rider where they are while moving through a city that was actively denying them GPS. The answer this book documents is yes, with numbers: 91–93% of the true distance, an honest speed at the limit of what the reference can measure, a clean zero at rest, and thirteen times the real-time headroom the budget demanded, all on a phone anyone can buy, in the exact conditions the system was built for. More than any single metric, the result the team is proudest of is the one that is hardest to plot: on a route where the world's most-used positioning system overstated the distance by a third and froze for a minute at a time, NavSight kept the rider on the road. That is the whole point of the project, and it works.

## 10 References

- [1] D. Scaramuzza and F. Fraundorfer, “Visual Odometry [Tutorial], Part I: The First 30 Years and Fundamentals,” *IEEE Robotics & Automation Magazine*, vol. 18, no. 4, pp. 80–92, 2011.
- [2] A. I. Mourikis and S. I. Roumeliotis, “A Multi-State Constraint Kalman Filter for Vision-Aided Inertial Navigation,” *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3565–3572, 2007.
- [3] S. O. H. Madgwick, A. J. L. Harrison, and R. Vaidyanathan, “Estimation of IMU and MARG Orientation Using a Gradient Descent Algorithm,” *Proc. IEEE International Conference on Rehabilitation Robotics (ICORR)*, 2011.
- [4] P. Newson and J. Krumm, “Hidden Markov Map Matching Through Noise and Sparseness,” *Proc. 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 336–343, 2009.
- [5] R. Ranftl, K. Lasinger, D. Hafner, K. Schindler, and V. Koltun, “Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-Shot Cross-Dataset Transfer,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 3, 2022.
- [6] P. Geneva, K. Eickenhoff, W. Lee, Y. Yang, and G. Huang, “OpenVINS: A Research Platform for Visual-Inertial Estimation,” *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2020.
- [7] T. Qin, P. Li, and S. Shen, “VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator,” *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 1004–1020, 2018.
- [8] G. Zhou and Y. Liu, “Visual-Inertial Navigation Systems: A Review,” *Sensors*, 2020.
- [9] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, “ORB-SLAM: A Versatile and Accurate Monocular SLAM System,” *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.
- [10] J. Shi and C. Tomasi, “Good Features to Track,” *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 593–600, 1994.
- [11] J.-Y. Bouguet, “Pyramidal Implementation of the Affine Lucas Kanade Feature Tracker: Description of the Algorithm,” Intel Corporation, Technical Report, 2001.
- [12] C. Forster, L. Carlone, F. Dellaert, and D. Scaramuzza, “On-Manifold Preintegration for Real-Time Visual-Inertial Odometry,” *IEEE Transactions on Robotics*, vol. 33, no. 1, pp. 1–21, 2017.
- [13] OpenCV Documentation. <https://docs.opencv.org/>

- [14] OpenStreetMap and the Overpass API. <https://www.openstreetmap.org/>
- [15] TensorFlow Lite Documentation. <https://www.tensorflow.org/lite>
- [16] Android Developers Documentation (CameraX, NDK, Jetpack Compose). <https://developer.android.com/>
- [17] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.
- [18] D. Nistér, "An Efficient Solution to the Five-Point Relative Pose Problem," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 6, pp. 756–770, 2004.
- [19] K. Zuiderveld, "Contrast Limited Adaptive Histogram Equalization," in *Graphics Gems IV*, Academic Press, pp. 474–485, 1994.
- [20] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
- [21] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 47–57, 1984.

# Appendix A: Background Material

Concept primers for readers new to the underlying theory. Each is a summary sufficient to read Chapters 2 and 7; the references give the full treatments.

**Rotations and quaternions.** Orientation in 3-D is represented by unit quaternions (or equivalently rotation matrices in  $SO(3)$ ); unlike Euler angles, they have no gimbal singularity, no orientation at which a degree of freedom collapses. Rotations compose by quaternion multiplication, and small rotations are well approximated by a 3-vector (the rotation's axis scaled by its angle), which is the representation error-state filters exploit: the filter estimates a small rotation *error*  $\delta\theta$  around a nominal quaternion, which stays well-conditioned because the error is always near zero.

**The error-state EKF.** A Kalman filter maintains a state estimate and its covariance, alternating *prediction* (propagate through a motion model, growing uncertainty) with *update* (correct against a measurement, shrinking it). The extended form linearizes nonlinear models at the current estimate. The *error-state* variant goes one step further: the nominal state is propagated directly with the raw IMU, and the filter estimates only the small error between nominal and truth. The linearization is therefore always around zero error, the regime where linearization is most accurate, which is what makes EKF-based VIO stable in practice. NavSight's error state comprises rotation, gyro bias, velocity, accel bias, and position (15 DOF), plus camera–IMU time-offset and camera-mount rotation calibration states (§7.1.3).

**IMU preintegration.** Between two camera frames the IMU produces dozens of samples. Naively, changing the estimate of the starting pose would force re-integrating all of them. Preintegration [12] compounds the samples into a single relative-motion constraint ( $\Delta R$ ,  $\Delta v$ ,  $\Delta p$ ) defined on the  $SO(3)$  manifold, *independent of the absolute start state*, so the filter consumes one measurement per frame interval and can re-linearize freely. This is the standard bridge between high-rate inertial data and frame-rate vision.

**Epipolar geometry and KLT.** Two views of the same rigid scene are related by the essential matrix  $E$ : corresponding rays satisfy  $p'^T E p = 0$ . Given  $\geq 5$  correspondences,  $E$  can be estimated and decomposed into the camera's relative rotation and translation *direction*, but not the translation's magnitude, which is the monocular scale ambiguity (§2.1). Correspondences come from tracking: Shi-Tomasi selects corners whose local gradient matrix is strong in both directions [10], and pyramidal Lucas–Kanade [11] finds each corner's displacement by iterative alignment, coarse-to-fine over an image pyramid so large motions converge. RANSAC makes the E-fit robust: repeatedly fit on 5 random correspondences, keep the hypothesis most correspondences agree with, and discard the rest as outliers.

**The Kalman filter, in one paragraph.** A Kalman filter [17] is the optimal recursive estimator for a linear system with Gaussian noise. It carries a state estimate and a covariance (its own uncertainty), and alternates two steps: *predict*, which pushes the estimate through a motion model and *grows* the covariance by the process noise; and *update*, which folds in a measurement, shrinking the covariance in proportion to how much more it trusts the measurement than the prediction (the Kalman gain). Everything else in VIO (the extended form, the error-state form, MSCKF, ZUPT) is a variation on which quantities go in the state and how each measurement's residual and Jacobian are formed. The reason the filter is the right tool for a phone is the same reason it was invented for spacecraft: it is recursive, so it needs only the current state and covariance, never the whole history.

**Zero-velocity updates (ZUPT).** When a device is known to be stationary, that knowledge is itself a measurement: velocity = 0 exactly, with near-zero noise. Applying it collapses velocity uncertainty and, more valuably, makes the gyro and accelerometer biases directly observable, because when true velocity and acceleration are zero, whatever the sensors report is their bias. A few seconds of standstill therefore recalibrate the very error terms that would otherwise drive drift. The hard part is detecting stationarity reliably: a phone at a red light vibrates, and a phone moving slowly can look still to any single sensor. NavSight gates on IMU statistics and optical flow jointly (§6.2.2), so a false “stationary” requires both the inertial and the visual channels to agree, which they rarely do by accident.

**Complementary vs. Kalman fusion.** The simplest sensor fusion, a complementary filter, blends two sources with fixed weights (trust the gyro at high frequency, the accelerometer at low). It is cheap and stable, and it is what the SRS conservatively specified. Its limitation is that the weights are fixed: it cannot notice that *this* measurement is an outlier, because it has no notion of uncertainty. A Kalman filter's covariance gives it exactly that notion, which is why NavSight upgraded to one, in urban vision, the ability to reject a bad measurement matters more than the ability to blend good ones.

**Hidden Markov map matching.** Model the true road segment as a hidden state observed through noisy positions: *emissions* score how close a measurement lies to a candidate road (Gaussian in perpendicular distance), *transitions* score whether consecutive candidates are route-consistent (comparing straight-line and along-road distances), and the Viterbi algorithm recovers the most probable road *sequence* by dynamic programming [4]. Because the decision is joint over the whole window, a momentary excursion toward a parallel street is outvoted by history; the property that makes HMM matching robust where nearest-road snapping oscillates.

**Monocular depth networks.** MiDaS [5] predicts relative inverse depth from one image, trained across many datasets with a scale-and-shift-invariant loss for zero-shot robustness. Its

output is defined only up to an affine transform, which is why NavSight fits an affine mapping in disparity space before using it to weight optical flow (§7.1.5).

**Inverse Perspective Mapping (IPM).** If the ground is a plane and the camera's height above it and orientation are known, every image pixel that sees the ground has a *closed-form* depth: the ray from the camera through that pixel intersects the plane at exactly one point. This is the one place in a monocular system where metric scale is free (no second camera, no learned prior) at the cost of two assumptions (planar road, calibrated height) that a road vehicle mostly satisfies (§7.1.4).

## Appendix B: Test Data and Environment

**Reference device.** Samsung Galaxy S21 Ultra (SM-G998B), Exynos 2100, Mali-G78 GPU, Android 15 (API 35).

**Toolchain.** Android Studio + NDK/CMake (compileSdk 34, minSdk 24); OpenCV 4.5.3 (Android SDK, vendored); TensorFlow Lite (MiDaS v2.1 small, 256×256); MSYS2/MinGW GCC for the desktop replay build; Python 3 with NumPy/Matplotlib for asset building, analysis, and report generation.

**Datasets.** Recorded ride/walk sessions under `tests/sims/`, per-frame JSON (pose, IMU, GPS, speed channels, event counters) plus recorded camera frames for replay. The key validation sets:

| Dataset                        | Content                                                             | Used for                              |
|--------------------------------|---------------------------------------------------------------------|---------------------------------------|
| val_2026_06_12/ride2_1802.json | ~99 s urban ride, health-verified GPS (93 fixes, median 4 m)        | Figures 8.2–8.5; T2, T3, T9           |
| val_2026_06_03b/               | Route A: 1,280 m out-and-back with U-turn, jammed GPS; probe frames | Figures 8.7–8.10; T4; Figure 7.5      |
| val_2026_06_04/                | Hostile-lighting ride + performance logcat                          | T7 (36 fallbacks); Figure 8.1 timings |

**Map data.** Israel-wide place index built by `scripts/build_haifa_assets.py` (16.6 MB, ~254k places); road regions fetched from Overpass at ride time and disk-cached (§5.2).

# Appendix C: Validation Procedures

1. **Recording a session.** Enable recording in the debug panel; ride. The app writes per-frame telemetry JSON and (optionally) timestamped PNG frames for replay.
2. **Replaying offline.** Build the harness under MSYS2 (tests/cpp/build\_mingw); run it against a recorded session directory. The unmodified engine re-processes every frame deterministically and emits a scored CSV of the trajectory.
3. **A/B comparison.** Run the same recording through two engine builds; diff the scored outputs. Differences are attributable solely to the code change; this is the procedure behind Figure 8.6, and behind every fix in §8.7.
4. **Regression suites.** `gradlew test` (74 Kotlin unit tests) and the C++ unit suite (GoogleTest); CI executes replay scoring on fixtures for every push (`.github/workflows/replay.yml`).
5. **Regenerating the validation figures.** `python scripts/make_validation_report_pdf.py` rebuilds the full V&V report; `python scripts/export_vv_figures.py` exports this book's Chapter 8 plots (Figures 8.1–8.6) from the same telemetry.

## Appendix D: Calibration Procedure

1. **Hold-still calibration (every session).** With the phone mounted, hold still for a few seconds: gravity is estimated (world alignment), gyro/accel biases settle, and the health chip reports readiness. Results persist in a versioned calibration store and reload on the next session.
2. **Camera mount height.** Set once per mounting rig in the calibration sheet; this is the metric-scale anchor for the ground-plane speed estimator. A GPS auto-height assist (Doppler-vs-position cross-checked) can propose the value when healthy GPS happens to be available.
3. **Per-gait scale.** The cadence classifier learns per-mode scale factors (walk / run / vehicle) during travel; they persist across sessions and continue to refine.
4. **Camera intrinsics.** The reference device's intrinsics ship with the app (`camera_calib.json`); other devices require a one-time OpenCV chessboard calibration producing `fx`, `fy`, `cx`, `cy`.

# Appendix E: JNI Interface Reference

The complete surface of `NativeBridge` (Kotlin object, backed by `native-lib.cpp`). All getters and setters snapshot the shared engine state under a mutex; the hot path returns in microseconds except `processCameraFrameDirect`, which runs the frame pipeline.

## Lifecycle

| Method                  | Purpose                                                 |
|-------------------------|---------------------------------------------------------|
| <code>startVIO()</code> | (Re)create the engine; re-apply persisted camera height |
| <code>stopVIO()</code>  | Destroy the engine; reset pose                          |
| <code>resetVIO()</code> | Zero the pose and re-initialize gravity alignment       |

## Hot path

| Method                                                                                              | Purpose                                                  |
|-----------------------------------------------------------------------------------------------------|----------------------------------------------------------|
| <code>processCameraFrameDirect(y, uv, w, h, yStride, uvStride, uvPix, ts, rsSkewNs): VioData</code> | Zero-copy frame ingest → full pipeline → per-frame state |
| <code>processAccelerometer(ts, x, y, z) / processGyroscope(ts, x, y, z)</code>                      | Timestamped IMU samples into the native queues           |

## Configuration and calibration

| Method                                                                | Purpose                                               |
|-----------------------------------------------------------------------|-------------------------------------------------------|
| <code>setIntrinsics(fx, fy, cx, cy)</code>                            | Camera intrinsics                                     |
| <code>setCameraHeight(m) / setUserHeight(m)</code>                    | Mount height (IPM scale anchor) / rider stride height |
| <code>setAnalyzerRotation(deg)</code>                                 | Portrait-region hint for the ground-plane band        |
| <code>setInitialHeading(azRad) / seedMadgwickYaw(yawRad)</code>       | One-shot heading seeds                                |
| <code>setMagnetometerHeading(yawRad)</code>                           | Gated compass input (startup alignment)               |
| <code>setScale(scale) / setMidasScaleK(k) / getMidasScaleK()</code>   | Scale corrections and persistence                     |
| <code>loadStoredCalibration(rot, gb, ab) / getCalibration(...)</code> | Push/pull the persisted IMU calibration               |
| <code>nativeLoadCalibration(path)</code>                              | Parse the intrinsics file                             |
| <code>setDepthMap(float[], w, h)</code>                               | Push the 1 Hz MiDaS depth frame                       |

| Method                                | Purpose                                         |
|---------------------------------------|-------------------------------------------------|
| nativeSetSessionAnchor(lat, lng, tNs) | One bootstrap geographic anchor for the session |
| nativeSetExtrinsicsRotation(R_bc)     | Body→camera rotation seed                       |

### Map feedback (both legs live and bounded, §7.1.6)

| Method                                                         | Purpose                                                                                                                       |
|----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| setRoadHeadingHint(deg)                                        | Bounded road-tangent heading nudge (§7.1.2)                                                                                   |
| setMapPositionCorrection(dE, dN) /<br>setMapPositionEnabled(b) | Gated cross-track position correction ( $\leq 0.05$ m/frame, raw trail only); its kill switch (the leg is enabled by default) |

### Diagnostics and telemetry

| Method                                                       | Purpose                                                                |
|--------------------------------------------------------------|------------------------------------------------------------------------|
| getInitStatus() / clearInitTimeout()                         | Initializer state machine                                              |
| getPositionCovariance(float[3])                              | $\sigma_{xx}$ , $\sigma_{xz}$ , $\sigma_{zz}$ for the uncertainty ring |
| getFusedSpeedMps() /<br>getGroundFlowSpeedMps()              | The two speed channels (§7.1.4–§7.1.5)                                 |
| getLastTrackedPointAges/InlierFlags(...)                     | Camera-overlay feature annotations                                     |
| getGroundPlaneSnapshot / IPM overlay<br>getters              | Ground-plane band diagnostics                                          |
| getCurrentCameraPose(float[16])                              | Overlay camera pose                                                    |
| nativeGetEventCountersJson() /<br>nativeResetEventCounters() | The -180-counter telemetry block                                       |

A handful of additional getters serve legacy debug overlays for the compile-time disabled subsystems; they return empty snapshots in the shipped configuration.

## Appendix F: Offline OSM Asset Format

NavSight's offline map data uses three packed little-endian binary formats plus a JSON manifest, all under the asset directory `osm/haifa`, each stamped with a four-byte magic and a schema version (`SCHEMA_VERSION = 1`) that the readers verify before parsing. Defining a compact fixed-layout format, rather than shipping raw OSM XML or GeoJSON, is what lets the ~254,000-place index fit in 16.6 MB and load in well under a second.

### Common header (all three blobs).

| Offset | Field         | Type    | Notes                                                                                 |
|--------|---------------|---------|---------------------------------------------------------------------------------------|
| 0      | magic         | 4 bytes | NSSG / NSGR / NSGC                                                                    |
| 4      | schemaVersion | int32   | must equal SCHEMA_VERSION                                                             |
| 8      | recordCount   | int32   | number of primary records (NSGR carries two counts here: vertexCount, then edgeCount) |

**NSSG, road segments (`haifa_segments.bin`).** Consumed by the segment R-tree [21] for nearest-road queries. Each record: `segId` (int64), `wayId` (int64), `classCode` (int32, road class), endpoint coordinates `aLat`, `aLng`, `bLat`, `bLng` (float64), and `nameIdx` (int32 into the trailing string pool). A length-prefixed UTF-8 string pool follows the record array.

**NSGR, routing graph (`haifa_graph.bin`).** Consumed by the on-device Dijkstra router. A vertex array of {`lat`, `lng`} (float64) followed by an edge array of {`fromIdx`, `toIdx`} (int32), `lenM` (float64, edge length in metres), `classCode` (int32), and `oneway` (int8 flag).

**NSGC, geocode index (`haifa_geocode.bin`).** Consumed by the offline destination search. A place array of {`placeId` (int64), `lat`, `lng` (float64), `nameIdx` (int32)}, a UTF-8 string pool, and an **inverted token index** mapping each lowercased search token to the list of place indices whose name contains it; the structure that makes “pharmacy” or a street name resolve without a network call.

**`haifa_assets.json`, manifest.** A small JSON sidecar carrying `schema_version`, the geographic bbox, entity counts (`segment_count`, `vertex_count`, `edge_count`, `place_count`), and the upstream OSM extract date, read by the region manager to validate coverage before use.

In the validated build only the NSGC geocode blob is bundled (search is needed everywhere the rider might type); the NSSG and NSGR road formats are produced by the same pipeline and used

for the on-demand, disk-cached road regions, ready for pre-packing whole regions into the APK as future work (§9.3).

## תקציר

מערכות הניווט המודרניות נשענות כמעט לחלוטין על מערכות לווין (GNSS), שם כולל לצבירי לווין אחדים: GPS האמריקאי, GLONASS הרוסי, Galileo האירופי ו-BeiDou הסיני. מערכות אלה עלולות להיכשל דווקא היכן שהן נדרשות ביותר: בקניונים עירוניים צפופים, במנהרות ותחת שיבוש רדיו. עבור פרויקט זה הבעיה לא הייתה תיאורטית. הפיתוח והאימות בוצעו בחיפה בתקופה של שיבוש GNSS אזורי ומתמשך. במהלך אותה תקופה נצפה מקלט ה-GPS קופא למשך יותר מדקה תוך כדי נסיעה, ומנפח ב-33% מסלול שאורכו המדוד 1,280 מטר.

NavSight היא מערכת ניווט לרוכב עירוני, כלומר רוכב קורקינט חשמלי או אופניים, אשר נכתבה ומיועדת לריצה על חומרת טלפון סלולרי בלבד. תפקידה לשמור את מיקום הרוכב על המפה ללא תלות בלוויינים. המערכת היא אפליקציית Android נטיבית (Kotlin / Jetpack Compose), הבנויה סביב ליבת חישוב בשפת ++C הנגישה דרך JNI.

הליבה משלבת שני מקורות מידע. המצלמה האחורית עוקבת אחר נקודות ציון בתמונה בשיטת KLT Kanade-Lucas-Tomasi, ומודדת כיצד העולם נע ביחס למכשיר. חיישני התנועה, מד התאוצה והג'ירוסקופ, עוברים אינטגרציה מוקדמת (preintegration) בין פריים לפריים. מסנן Kalman מורחב, בגישת error-state, ממזג את שני המקורות להערכה אחת של מיקום, מהירות וכיוון.

הכיוון מחושב על-ידי מסנן Madgwick הנשען בעיקר על הג'ירוסקופ. המצפן קובע את הכיוון ההתחלתי בעת תחילת הנסיעה, ולאחר מכן תורם תיקון עדין בלבד, אשר נדחה אוטומטית בנוכחות הפרעה מגנטית.

המהירות המטרית נמדדת מזרימה אופטית (optical flow) על מישור הכביש, בהסתמך על גובה מצלמה מכויל. בזמן נתק חזותי, למשל בחשכה או בטשטוש תנועה, המערכת גושרת על הפער באמצעות חיישני התנועה, לחלון זמן חסום ומוגדר מראש.

התאמת מפה (map matching), המבוססת על מודל Markov חבוי ומפוענחת באלגוריתם Viterbi, מצמידה את המיקום המוצג לרשת הכבישים של OpenStreetMap השמורה מקומית במכשיר. חיפוש יעדים מתבצע מול אינדקס ארצי הארוז בתוך האפליקציה. גאומטריית הכבישים סביב הרוכב נשמרת במטמון המכשיר, ולכן אזורים שכבר בוקרו פועלים ללא כל קישוריות. חישוב המיקום מתבצע כולו על המכשיר עצמו, ומסתמך על שלושה מקורות בלבד: המצלמה, חיישני התנועה ונתוני המפה השמורים מקומית.

המערכת אומתה ברכיבות עירוניות אמיתיות תחת שיבוש GNSS פעיל, על מכשיר Samsung Galaxy S21 Ultra. דיוק מרחק הנסיעה עמד על 91% עד 93% מול שני מקורות ייחוס בלתי תלויים: מסלול שנמדד במפה, שהוא חסין שיבוש, ומקלט GPS שעבר סינון תקינות. חציון מהירות הנסיעה היה בטווח 3.5% מחציון מהירות הייחוס. בעצירה מלאה הוצגה קריאה של 0 קמ"ש בדיוק. זמן העיבוד החציוני עמד על 15.4 אלפיות שנייה לפריים, פי 13 מתחת לתקציב הזמן אמת של 200 אלפיות שנייה. במסלול שבו ה-GPS המשובש הפריז במרחק ב-33%, NavSight נשארה בטווח 7% מהמרחק שנמדד במפה. כל 13 מקרי הבדיקה שהורצו עברו בהצלחה בגרסה המאומתת.

**מילות מפתח:** ראייה ממוחשבת, אודומטריה חזותית-אינרציאלית (VIO), ניווט ללא GNSS, התאמת מפה, Android.

## הכרת תודה

ברצוננו להודות למנחים שלנו, מר עמית דונסקי וד"ר דביר רוס, על ההנחיה, הביקורת המקצועית והתמיכה לאורך השנה, ולסגל המחלקה להנדסת תוכנה בשנקר על הידע והכלים שאפשרו את הפרויקט הזה.

שנקר, המכללה להנדסה

המחלקה להנדסת תוכנה

# NavSight

מעבר ל-GPS: ניווט מדויק בסביבות נטולות-GPS

ספר פרויקט גמר לתואר ראשון בהנדסת תוכנה

## מחברים

רועי בן חרוש · 315676163

תמיר סבוה · 206029084

מוראד זובידאת · 208156828

## מנחים

מר עמית דונסקי · ד"ר דביר רוס

יולי 2026

|                         |                                                         |
|-------------------------|---------------------------------------------------------|
| Title                   | Graduation Project Book - Final Version for E-Signature |
| File name               | NavSight_Project_Book_FINAL.pdf                         |
| Document ID             | aecfb11b0524d7ae39f58bb3d1cd83f93a942823                |
| Audit trail date format | DD / MM / YYYY                                          |
| Status                  | ● Signed                                                |

## Document History



**22 / 08 / 2026**  
21:52:19 UTC+3

Sent for signature to Dvir Ross ( [REDACTED] )  
from [REDACTED]  
IP: [REDACTED]



**23 / 08 / 2026**  
22:23:44 UTC+3

Viewed by Dvir Ross ( [REDACTED] )  
IP: [REDACTED]



**23 / 08 / 2026**  
22:25:21 UTC+3

Signed by Dvir Ross ( [REDACTED] )  
IP: [REDACTED]



COMPLETED

**23 / 08 / 2026**  
22:25:21 UTC+3

The document has been completed.