# Term Project

A Device Someone Needs: Two Sensors, One Raspberry Pi, and Software You Can Defend

*Real problems, kernel-adjacent mechanisms, quantitative evidence, and disciplined AI-assisted development*

Version 1.0 August 3, 2026

Version history appears in Section 20.

## Contents

**Nota Bene:** The term project is done in teams of exactly two. Its *shape* will feel familiar from the assignments: Claude Code is required, your process is graded from your transcripts, and the whole thing ends with a live demonstration and an individual defense in which each of you answers for the code with your name on it. What's different is the problem. Nobody hands you this one. You will choose a user, choose their problem, and build a working device that solves it, subject to the guardrails in Section 3. And one of those guardrails deserves top billing, because it inverts the assignments' arrangement: **Claude Code is required in your workflow and forbidden in your product.** Every line of intelligence that runs on the Pi, every decision, detection, and diagnosis, must be code you wrote and can defend. The agent helps you build the machine. It is not allowed to *be* the machine.

# 1 Overview

Every assignment in this course hands you a problem and asks for a program. The term project reverses the polarity. It hands you a program-shaped hole in the world and asks you to find it. You will design and build a working device (a Raspberry Pi, at least two sensors, and software of your own) that does something a specific person actually needs done. Not a demo of a sensor. Not a tutorial, reassembled. A device that could sit on a shelf in that person's life and earn its electricity.

The question that will follow you through this project, from the first memo to the final defense, is this: *who wants this, and why doesn't a phone app already solve it?* The second half of that question is quietly load-bearing. A phone runs its apps when someone is looking at it. Your device has to be awake at 3 a.m., unattended, on the day the interesting thing finally happens. Always-on, physically embedded, resource-constrained, unsupervised: that's exactly the territory where processes, schedulers, interrupts, storage, and IPC stop being lecture topics and start being the difference between a product and a demo. You don't need to bolt operating-systems content onto this project. Choose an honest problem and the operating-systems content will come to you. Uninvited, and usually at inconvenient hours. The guardrails in Section 3 exist only to make sure the problem you choose is honest.

The project is deliberately open. The five concepts in Section 4 are seeds for your ideation, not a menu to order from; the strongest projects most years are ideas we didn't think of. What is not open is the standard of evidence. Your system must survive 48 hours unattended and prove it with logs. Your claims about latency, throughput, and reliability must be measurements, not adjectives. And at the end, each of you, separately, must be able to stand in front of your own code and explain why it is the way it is. What is *not* expected is perfection, and Section 5 says so at length. The world your device ships into is imperfect. Your budget is finite, your sensors are cheap, and the project is graded accordingly: on what you did with what you had, and on the honesty of the accounting.

**Grading methodology and the demo you can't rehearse**

The term project accounts for 10% of your course grade and is graded out of 100 points, split the same way the assignments are: most of the points answer a question the artifacts can settle, and the rest answer a question only you can.

70 points attach to the work and its evidence: the problem memo (5), the design document (10), the checkpoint demonstration (10), the final system with its 48-hour soak test (30), and the quantitative evaluation report (15). These are graded from things we can run, read, and re-measure.

The remaining 30 points are *individual*, and no teammate can earn them for you. 10 points come from the analysis of *your* Claude Code transcripts, on the same five-level, repository-corroborated scale used in the assignments. 20 points come from the demo-day defense: questions about lines of your own subsystems, one question from across the ownership boundary, a defense of one episode from your `PROMPTLOG.md`, and a small live modification you make to your own code, alone, at the keyboard, while we watch. So two partners can submit one flawless repository and still leave demo day with very different grades. The defense was never grading the repository. It was grading each of you.

Why the split? Same reason as the assignments, scaled up. A device that works is a nice thing to have. But a pair of engineers who can each explain, unprompted, why their sampler can't drop a reading and what their watchdog does when a sensor gets unplugged? That's a nice thing to *become*, and it's the thing this course is actually for.

## 2 Learning Objectives

- Translate a real user's real problem into a system design, and defend every component of that design by the problem's demands rather than by habit.
- Build an embedded sensing product on Linux that exercises at least two below-the-application-layer mechanisms: device drivers, interrupt handling, real-time scheduling, custom storage, or multi-process architectures with fault isolation.
- Write the product's intelligence yourself (statistics, signal processing, state machines, control) and know precisely why a threshold wouldn't have sufficed.
- Engineer for unattended operation: supervision, recovery, log discipline, resource stability over days rather than seconds, and graceful degradation when hardware misbehaves.
- Evaluate a system quantitatively: design the experiments, take the measurements, and present latency, utilization, and reliability numbers you can stand behind under questioning.
- Collaborate as a two-person team with explicit ownership, mutual code review, and a development process (Claude Code included) that leaves a verifiable trail.

| Item | Detail |
|---|---|
| Team | Exactly two students; individual grade components as noted in Section 13 |
| Platform | Any Raspberry Pi with the 40-pin header (Pi 5, 4, 3B+, or Zero 2 W); see Section 17 |
| Sensors | At least two physically distinct sensors that genuinely cooperate (Section 3.1) |
| Language | Systems core in C (C17 or C23), `-Wall -Wextra -Werror` clean; the interface layer may use anything, but no graded mechanism may hide in it |
| Expected effort | Approximately 45–60 hours per student across the semester, milestoned in Section 6 |
| Tooling | Claude Code is required in your workflow (Section 10) and forbidden in your product (Section 3.3) |

# 3 The Guardrails: What "Non-Trivial" Means Here

An open-ended project needs a floor, or the path of least resistance becomes a Python script that polls a sensor and prints a number. Five guardrails define the floor. Everything above them is yours.

## 3.1 Two sensors that cooperate

Your device must incorporate at least two physically distinct sensors, and they must *cooperate*: fused into one decision, correlated against each other, or feeding a single shared pipeline whose behavior depends on both. Two independent demos stapled together at the README do not satisfy this requirement. That's one sensor short of the assignment, twice. Cooperation is also where the systems content hides. Two sensors almost never want the same sampling rate, the same interface, or the same latency budget, and reconciling those differences inside one program is the first real design problem this project hands you.

## 3.2 Below the application layer: the mechanism menu

Your system must include **at least two** of the following mechanisms, implemented by you, identified explicitly in your design document, and measured in your evaluation. This is the guardrail that keeps the project an operating-systems project rather than a wiring exercise.

| # | Mechanism | What "measured" means |
|---|---|---|
| A | A kernel module or character driver for one of your sensors, exposing it via `/dev` or sysfs | Correct concurrent access; behavior under load; a clean `dmesg` |
| B | Interrupt-driven input, with a head-to-head comparison against a polling design | Event latency distribution and CPU utilization, both ways |
| C | Real-time scheduling (`SCHED_FIFO`/`SCHED_RR`) for a latency-critical path | Jitter and deadline-miss rate vs. the default scheduler, under load |
| D | A custom storage layer: ring buffer, append-only log, or small FUSE filesystem, with an explicit crash-consistency story | Write batching and `fsync` discipline; recovery after a mid-write power cut |
| E | A multi-process architecture: sensors isolated in separate processes, a hub aggregating over IPC (shared memory, message queues, or Unix domain sockets), a supervisor that survives any child dying | Kill any child at any time; the system degrades, logs, recovers |
| F | A high-rate sampling pipeline with a no-drop guarantee: lock-free or single-producer/single-consumer ring between capture and consumption | Sustained rate with zero drops, proven by sequence accounting, under CPU contention |

Choose the two (or more) that your *problem* demands, not the two that look easiest. The design document (Section 8) requires you to justify each choice from the user's requirements, and the defense will test that justification. Figure 1 shows where these mechanisms live in a reference architecture.

### 3.3 The intelligence is yours

The running product may not call a hosted model of any kind (no Claude, no other LLM, no cloud inference API) and may not embed a pretrained model unless you petition for it in your design document and receive approval. All analysis, detection, diagnosis, and decision logic must be code you wrote: rolling statistics, spectral features, sensor-fusion rules, state machines, control loops. Libraries for mathematical primitives (an FFT, a linear solver) are fine, provided you can explain what the primitive computes and why your pipeline needs it.

Two rules of thumb bound the requirement, one from below and one from above. From below: *if a single threshold could do it, it isn't intelligence*, and your product must do something with its two sensors that an `if`-statement could not. From above: *if a model you didn't write does it, it isn't yours*, and this project is grading you, not a weights file. Related, and deliberate: the device

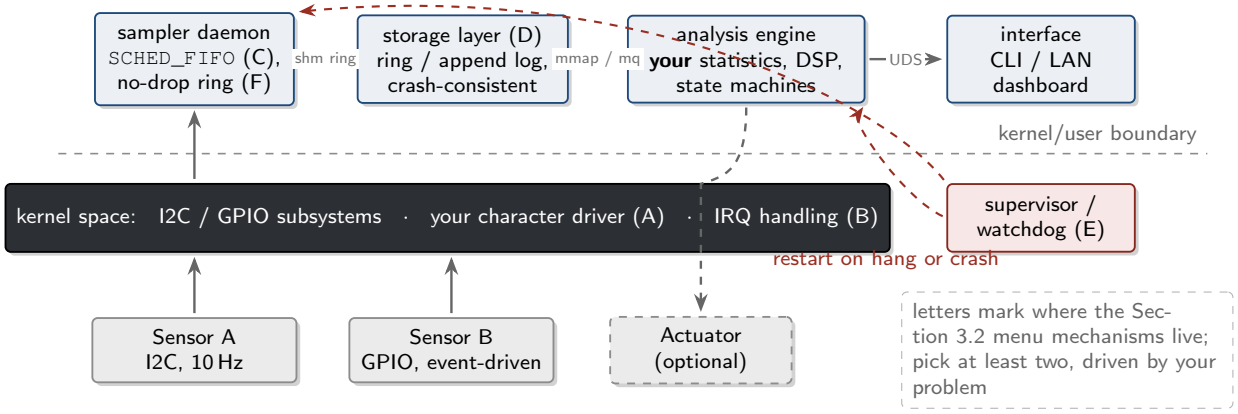


Figure 1: A reference architecture, not a required one. Sensors enter through the kernel; a sampler moves data into a storage layer without losing any; an analysis engine you wrote turns data into decisions; an interface lets a human see them; a supervisor keeps the whole thing alive at 3 a.m. Your product may collapse, split, or rearrange these boxes freely. The graded requirement is only that at least two menu mechanisms exist somewhere in it, below the application layer, written by you.

must function with the network cable unplugged. Serving a dashboard on the LAN is fine. But a product whose brain lives in the cloud is not an embedded product, and it is not this project.

### 3.4 Survivability: the 48-hour soak

In the week before demos, your final build must run unattended for at least 48 continuous hours, and the logs proving it are a graded deliverable. The logs must show: a timestamped startup; a heartbeat at least hourly recording process liveness, resident memory (RSS), and cumulative event counts; at least one injected fault during the run (physically unplug a sensor for ten minutes, or `kill -9` a sensor process) with the system's detection, degradation, and recovery visible in the log; and an orderly state at the end.

Understand what this guardrail is really doing. Surviving two days unattended is an operating-systems problem wearing a scheduling-your-week costume. It's the slow memory leak. The unhandled error on the path you never tested. The log that fills the SD card, the child process that dies silently at hour 31, and the watchdog you will be very glad you wrote. A system that dazzles for the five minutes someone is watching and decays the moment they leave has a name: *demo*. This project is asking for the other thing.

### 3.5 Someone wants this

And the last guardrail is the one that shapes all the others: your project must serve *a specific user with a specific problem*. This should be named in your problem memo (Section 7) and defensible at the walkthrough. "People who might want to monitor things" is not a user. Your roommate whose

sourdough starter keeps dying is a user. So is the shop downstairs whose compressor makes a new noise. Grandparents, greenhouses, 3D printers, practice rooms, bee hives: the world is oversupplied with small, real problems that a phone app can't solve because a phone is never *there* when it matters. Find one. Every design decision you'll be asked to defend gets easier to defend when the answer can begin with "because the user needs. . . "

# 4 Ideation

This is where the project actually begins, so it gets more room than any other section. I am including 5 worked seeds. But they are seeds, not a menu: none is required, all are claimable by any team, and the strongest proposals most years are ideas we didn't think of. Read the seeds less for their subjects than for their skeleton. Every one of them has the same bones if you will: an absent observer, an always-on device, intelligence a threshold couldn't provide, and a user you could name. Your job is to find a problem with the same skeleton and a different flesh.

## 4.1 Three ways to find your own

Audit a week. Carry a note for one week and write down every time you, or someone near/close-to you, utters one of two sentences: *"I wish I knew. . . "* and *"by the time I noticed. . . "*. These are sensor products wearing plain clothes. The first names a measurement that is absent. The second names a measurement that existed but had nobody watching it, which is the 3 a.m. test passing itself, unprompted, in ordinary conversation. A week of honest auditing typically yields a dozen candidates, of which two survive the guardrails. And that's exactly as many as you need.

Shrink an industry. Predictive maintenance, cold-chain monitoring, occupancy analytics, precision agriculture, structural health monitoring: entire companies exist to do each of these at scale, which means someone has already established, expensively, that the problem is worth solving. Take one. Shrink it to a single machine, a single room, a single shelf, and you inherit a validated problem sized for a semester. You also inherit something rarer: the discovery, which tends to stay with people, that the distance between a course project and a real industry is mostly a matter of scale, not of kind.

Instrument a devotion. Hobbies are problems folks have volunteered for: fermentation, aquariums, climbing training, beekeeping, film-developing chemistry, model rocketry. Devotees make the best users, for three reasons that matter to your grade. They already measure things by hand, so the value of automating the measurement is pre-established. They will actually run your device. And they will tell you (at length and for free) exactly what it gets wrong, which is the most expensive kind of feedback in all of engineering; that is now available here for the price of listening.

Whatever you find, run it through the guardrails of Section 3 as filters, and through Section 5 as a reality check. Ideally before the memo. Certainly before the design document.

## 4.2 Seed 1: "Is my elderly parent okay today?" (ambient wellness monitor)

The user is an adult child who lives an hour, or a continent, away from a parent who lives alone. The moment the product exists for is a quiet one: not the fall, but the morning that didn't follow

its rhythm, noticed hours before anyone would have been frightened enough to call. Every existing answer asks the *parent* to do something. Wear the pendant. Charge the watch. Answer the daily check-in. And the failure mode is that the people most at risk are precisely the people least likely to keep doing it. The design brief is then a device that demands nothing: no camera, no microphone, no button, no new habit. That restraint is not a limitation to apologize for. It's the product's spine, and a memo that can articulate why a motion sensor is more respectful than a camera is a memo that understands its user.

Engineering: a PIR motion sensor and a door or pressure sensor, learning a household's rhythm. The intelligence (all of it yours) is per-hour activity histograms over a rolling window of weeks, a deviation score that knows a Tuesday from a Sunday, and alert hysteresis. Because the real analytical problem here isn't detection. It's *restraint*. A monitor that cries wolf twice gets unplugged. An unplugged monitor is worse than none, because somebody far away believes it's on duty. Natural mechanisms: E, because a silently dead wellness monitor is this product's own nightmare scenario, which makes process isolation and supervision requirements rather than garnish; and D, because weeks of learned baseline must survive a power cut. A scoped-down version that still clears every bar: one room, one rhythm, two weeks of baseline, and an alert *log* you evaluate for false-alarm rate. No paging infrastructure required.

### 4.3 Seed 2: "Why does my sourdough / homebrew / greenhouse keep failing?" (environmental coach)

The user is a devotee (Section 4.1, third method) whose batches keep dying and who has already tried the obvious: a thermometer that's glanced at while awake. The insight the product is built on is that what kills the batch is almost never the moment anyone is looking. It's the 4 a.m. excursion. The slow weekend drift. The heater that quietly began short-cycling on Tuesday. And the user's real question is never "what is the temperature?"; it's "what is *wrong*?" The distance between those two questions, between a reading and a diagnosis, is exactly the distance your intelligence has to cover. No threshold covers it.

Engineering: temperature/humidity plus light or $CO_2$, sampled at rates that differ by an order of magnitude, flowing through a storage layer of your own design (D), often behind a no-drop pipeline (F). The intelligence is rolling statistics, rate-of-change detection, and, the satisfying part, *signature* detection. A short-cycling heater oscillates with a period and amplitude you can measure. A failing seal shows up as indoor humidity coupling to the outdoor weather. A dying yeast culture flattens a $CO_2$ curve that should be climbing. Encoding that physics into diagnostic rules requires actually learning the domain. Which means talking to your user, which is the job. Where it bites: the temptation to promise twelve diagnoses and deliver them all vaguely. Three diagnoses, each with a measured true- and false-positive rate against failure episodes you *provoked on purpose*, is the far stronger project. And Section 5 is about why provoking them on purpose is not a workaround but a method.

### 4.4 Seed 3: "Did something happen while I was gone?" (explainable security node)

Everyone with a door has asked the question. The commercial answer is four hours of footage nobody will ever watch. This product's differentiator is one word: *explainable*. It answers with a timeline (3:12 p.m., something person-sized, eleven seconds, moving away) that a human can read in five seconds and interrogate afterward. The user is anyone who wants that answer without becoming a security-camera hobbyist and, not incidentally, without pointing an internet-connected lens at their own life. Like Seed 1, the absence of a camera is a feature you can defend out loud.

Engineering: PIR motion plus ultrasonic ranging in an interrupt-driven pipeline. That's mechanism B, with the polling comparison built into your evaluation, and usually C, because ranging an object during a two-second event is a latency budget you have to actually meet rather than merely admire. The intelligence is fusion (motion says *something*, range says *how big and how far*), debouncing, event segmentation, and the design of the timeline itself. A log schema from which a human can reconstruct the truth is a genuine engineering artifact, and it's the thing your demo-day answer to "was that the cat?" will be read from. Speaking of the cat: it is this seed's honest adversary, and a magnetic reed switch on the door is the cheapest ground-truth generator you will ever wire. Every real entry gets a free label. Scoped-down version: three event classes (person-sized, small, unknown) classified honestly, with the confusion matrix to prove it, beats ten classes classified wishfully.

### 4.5 Seed 4: "Is this machine about to break?" (small-shop predictive maintenance)

There is an entire industry (vibration analysis, condition monitoring) built on the fact that machines announce their failures early, in frequencies, to anyone equipped to listen. Its instruments cost more than your car. Its customers are factories. The user here is everyone below that line: the shop with one compressor, the makerspace with one 3-D printer, the laundromat, the house with a furnace fan. The moment the product exists for is the new noise that started on a Tuesday when nobody was in the room. We tend to light up on discovering that the hobby-scale version of a real industry fits inside one semester and about $18 in parts. That discovery is worth having. And nothing says it has to expire with the course.

Engineering: an accelerometer (the MPU-6050's hardware FIFO is your friend) and a temperature sensor. Mechanism F is nearly mandatory, because a dropped sample corrupts the spectrum; this is the seed where the systems work and the analysis work hold each other honest most tightly. C is close behind, and A is there for the ambitious. The intelligence is feature extraction (FFT band energies, RMS trends) compared against a healthy baseline you recorded yourself. Where it bites, instructively: you can't wait a semester for a real bearing to fail. So you *induce* faults. Tape a washer to a fan blade for imbalance. Loosen a mounting bolt. Run the printer with a dry rod. Record labeled captures of each. That's not cheating; it's how the real industry builds its test sets, and Section 5 makes it doctrine. The scoped-down version is also simply the honest one: "detects the three faults I can induce on this machine, with these error rates" survives a defense. "Predicts failure" does not.

### 4.6 Seed 5: "Ask the room" (facilities query hub)

The user is whoever answers for a room that other people complain about: the practice-space manager, the lab steward, the roommate whose name is on the lease. The complaints arrive in the past tense. "It was freezing in here yesterday morning." "It gets stuffy every afternoon." And no live dashboard on earth answers a past-tense question. The product is a memory for a building: several sensors, a history, and an interface that can answer "mean $CO_2$ by hour, weekdays, last month" in less time than the argument takes.

Engineering: three or more sensors (air quality, sound level, temperature) as isolated processes feeding a hub over IPC. That's mechanism E, with real supervision, since a facilities record with a three-day hole has lost the argument it exists to win. Everything persists through a storage layer (D) whose retention, rollup, and `fsync` policies you must actually decide rather than inherit. The intelligence is the query engine: time-range selection, aggregation, alignment of multi-rate series, and cross-sensor correlation (afternoon $CO_2$ tracking the sound level is an occupancy story told by two sensors that never met). Where it bites: interface ambition. Resist the pretty web app. A crisp CLI with five query verbs and exact semantics is more defensible, and far more extensible at the live-modification station on demo day, than a dashboard with none. This is the seed for teams who suspect they might enjoy building databases, because it is one, in miniature, standing on an IPC layer they also built.

## 5 Building in an Imperfect World

Somewhere between the memo and the checkpoint, every team hits the same wall. It's worth telling you now that the wall is scheduled. The sensor your design really wants is a $400 thermal imager and your budget is $60. The $4 accelerometer you bought instead has a noise floor sitting exactly where the interesting signal lives. The $CO_2$ sensor needs three minutes to warm up and drifts with the weather. And the data you need at the scale you want (a year of failing compressors, a winter of greenhouse nights, $10^4$ labeled door events) does not exist, can't be bought, and won't be gathered in one semester by one device on one shelf. None of this means you chose the wrong project! It means the project has become *real*. Engineering has been called the art of the possible. The phrase is not a consolation; it's the job description. The constraints are not in the way of the work. They are the medium the work is done in.

The unproductive response is to stare at what can't be done. The productive one is to ask, repeatedly and in roughly this order, what can:

- Substitute. Downgrade the sensor; keep the problem. A PIR and a door switch see more of a household's rhythm than you'd guess a thermal camera was needed for. A microphone's RMS level is a serviceable occupancy proxy. And two cheap sensors that corroborate each other routinely outperform one expensive one standing alone. Notice that "corroborate each other" is Section 3.1 arriving early, wearing a budget. Record in the design document what you wanted, what you used, and what the substitution costs in capability.
- Make your own ground truth. You can't wait for the bearing to fail, so you provoke the conditions and label the recordings: the taped washer, the loosened bolt, the greenhouse vent propped open

on purpose, a one-week paper logbook of real door events. Ten labeled recordings you made are worth more than ten thousand unlabeled samples you found, because you know what yours mean. And at the defense, you will be asked how you know.

- Replay. Build a harness that feeds recorded sensor streams through your *real* pipeline at real or accelerated speed. Now a month of "operation" fits in an afternoon, rare events can be rehearsed on demand, and regressions are caught by re-running last week. The harness is not a workaround to be embarrassed about. It's a legitimate systems artifact (real products are tested exactly this way), and building one that faithfully preserves inter-sample timing is engineering in its own right. One rule, absolute: replayed or synthesized data is labeled as such everywhere it appears. In logs. In the report. In the demo. The 48-hour soak and the live demonstration run on live sensors; everything else may run on honest replay.
- Narrow the claim. Shrink the sentence until it is true. "Detects the three faults I can induce on this compressor, with these error rates" is a claim that survives cross-examination; "predicts failure" is not. A narrow claim, met and measured, outscores a broad claim that's gestured at. On this rubric, and on every rubric that matters after this course.

What this means for your grade deserves to be said without hedging: **the term project does not expect a perfect project.** It expects an honest one that works. We grade your reasoning about constraints, never the absence of constraints, because the absence of constraints is not a condition that occurs in nature. The design document has a home for this reasoning (Section 8), and the evaluation report's limitations section is where it pays off. The team that writes "we wanted X, we could afford Y, and here is precisely what Y cost us" has demonstrated the exact judgment this course exists to build, and will be graded like it. The team that quietly papers over the gap has converted an ordinary constraint (the most ordinary thing in engineering) into an integrity problem. And that is the worst trade available anywhere in this course. We live in an imperfect world. Every system you will ever ship will ship into it. So you may as well learn to do it well, and without apology, now.

## 6 Milestones

The project runs from Week 3 through demo week. Milestones are due at the end of the listed week; each is graded, and each exists to make a specific late-semester disaster impossible.

| Milestone | Deliverable | Week | Est. team hours |
|---|---|---|---|
| M0 | Team formed; hardware ordered; repo and `CLAUDE.md` initialized | 4 | 2–3 |
| M1 | Problem memo (Section 7); each partner's `.jsonl` transcripts to date copied out | 5 | 3–4 |
| M2 | Design document (Section 8); both sensors electrically alive; transcripts copied out | 7 | 8–10 |
| M3 | Checkpoint demo: both sensors flowing through the real pipeline into real storage; at least one menu mechanism working; transcripts copied out | 10 | 25–30 |
| M4 | Feature freeze; 48-hour soak begins; transcripts copied out | 14 | 30–40 |
| M5 | Final submission: system, soak logs, evaluation report, process artifacts, complete transcripts | 15 | 10–14 |
| M6 | Demo day: live demonstration, fault injection, individual defenses | 15–16 | n/a |

Transcript submission is not a one-time, end-of-semester task. At every milestone from M1 through M5, each partner copies out their raw `.jsonl` Claude Code session files (Section 10) covering the work done since the previous milestone. Building this habit in Week 5 is the point: it costs a minute per milestone, and it is the only way to guarantee that the transcripts graded at M5 actually cover the whole semester rather than whatever the local 30-day retention window happened to spare.

The disasters, in order: a team that never quite forms; a project nobody wants; a design that was never written down and so was never argued about; an integration that begins the night before it is graded; a soak test that can't start because the system can't yet run for an hour; a demo that meets its first unplugged sensor in front of an audience; and Week-5 transcripts that no longer exist by Week 15 because nobody copied them out along the way.

# 7 The Problem Memo (M1)

One page. Its job is to survive being read by a skeptical stranger. Six headings, all required:

1. The user. A person or place, named or nameable, whom you could actually put this device in front of.
2. The problem. Observable and costly: what goes wrong, how often, what it costs in money, worry, or ruined sourdough.

3. Why a device. The 3 a.m. test: why must something be physically present and always awake? And why doesn't a phone app already solve this?
4. The sensors. Which two (or more) and how they *cooperate* rather than coexist.
5. The mechanisms. Your first guess at which two menu items (Section 3.2) the problem demands, in one sentence each. This guess may change by M2. Guessing is still required.
6. The risk. Stated plainly this is the single thing most likely to sink this project. Section 5 is, among other things, a catalog of the usual suspects. Teams that can name their risk in Week 5 rarely meet it in Week 14.

A template is provided in Appendix C. Memos that fail the phone-app test or name no real user will be returned for revision. The five points attach to the revised memo, but the revision costs you a week you'll want back.

## 8 The Design Document (M2)

3-5 pages and the most leveraged hours of the whole project. Because every hour of arguing here saves five of rewriting later. Required contents:

- Architecture. A diagram in the spirit of Figure 1: processes, threads, kernel components, data flows, and rates on every arrow.
- Mechanism mapping. For each chosen menu item: which component implements it, and a justification *from the user's requirements* ("the vibration analysis is meaningless above 2 ms of sampling jitter, hence `SCHED_FIFO`").
- Failure-mode table. For each component: how it can fail, how the failure is detected, what the system does about it, and what the log will show. The soak test (Section 3.4) will grade this table's honesty.
- Storage and data. What is stored, at what rate, in what format, with what retention, and what happens to it when the power dies mid-write.
- Constraints and substitutions. What the ideal build would use, what you're actually using, and what the gap costs: sensors downgraded, data synthesized or replayed, claims narrowed (Section 5). A design document with nothing to report here has usually not met its hardware yet.
- Evaluation plan. The measurements you will take (Section 9), each with its method and its target. Numbers you commit to now are twice as credible when you hit them later. And instructive either way.
- Ownership map. Which partner owns which subsystems (Section 10.1). Ownership means first authorship and answerability at the defense, not exclusivity.

- AI-use plan. What you'll use Claude Code for, what you won't, and how you'll keep the Section 3.3 boundary visible in your repository (e.g., the product source tree contains no network client but the LAN interface).

# 9 The Evaluation Report (M5)

Adjectives are not measurements. Your evaluation report (3–4 pages, submitted with the final system) must contain, at minimum:

- Latency or timing, wherever your design claims it matters: distributions (not just means) for event-to-detection or sample-to-storage, measured under both idle and loaded CPU.
- Resource footprint: CPU utilization and RSS for each process, at steady state and over the full soak (the soak heartbeats give you this series for free; plot it).
- One domain metric: detection accuracy against ground truth you constructed, sustained sample rate with drop accounting, or query latency over the stored history. Pick the one your product lives or dies by.
- Fault-injection results: for each failure mode in your design-doc table, what you injected, what the system did, and the log excerpt proving it.
- The mechanism comparisons your menu choices require (e.g., interrupt vs. polling for item B, RT vs. default scheduling for item C), presented as experiments with method, data, and conclusion.
- Limitations, stated plainly, closing the loop on the constraints and substitutions declared in your design document (Section 5). An honest limitations section is worth more at the defense than a suspiciously perfect results section, and we notice which one we are reading.

Claude Code can generate plausible code. But it can't fake your understanding of your own measurements. Which is why the defense draws so heavily on this report.

# 10 Required Use of Claude Code

The nine-step workflow from Assignment 0, Section 10 applies to this project *verbatim*: persistent context in `CLAUDE.md`, explore before writing, plan mode first, tests leading, evidence over assertion, context hygiene, adversarial review, git discipline, and automation. Your transcripts will be read against it. This section covers only what changes when the codebase is shared, the target is hardware, and the model is banned from the payroll.

## 10.1 What changes in a team of two

- One `CLAUDE.md`, two authors. The file is team law: build and deploy commands, the hard constraints (no runtime model calls; no network dependence; every daemon supervisable), style rules, and the ownership map. Both partners maintain it. The agent reads it in every session, so a constraint that lives only in one partner's head protects nothing. A sample is in Appendix B.

- Sessions are personal. Each partner works in their own clone and their own Claude Code sessions, and submits their own transcripts. Your 10 process points are computed from *your* `.jsonl` files, corroborated against *your* commits.
- Your partner is a reviewer the agent can't replace. Every milestone's diffs get two reviews: a fresh-context agent review (Assignment 0, Step 7) and a human one by the partner who did not write the code. Each partner's `PROMPTLOG.md` must include at least one episode of reviewing the other's work: what you questioned, what the answer was, what changed. A team whose two halves never disagreed about anything all semester has not been reviewing.
- Commits carry names. At least 40 meaningful commits across the project, with both partners well represented. A history in which one partner authored under a quarter of the substantive commits will be probed hard at the individual defense. Which, remember, is 20 individual points.

### 10.2 What changes when the target is hardware

The agent can't see your breadboard and this cuts both ways. It will confidently reason about a sensor it has never met, and it will be wrong in ways that only an instrument can catch. So the evidence discipline of Assignment 0, Step 5 gets a hardware clause: close the loop with *measurements*. Paste `dmesg` verbatim. Paste the timing capture. Paste `/proc/interrupts` before and after. The agent reads a kernel oops better than a vague description of one. And so will you, by the end of this project,.

> *Our DHT22 reads fail checksum roughly 30% of the time, but only when the analysis process is busy. Here is the pulse-width capture from the logic analyzer for one failed read: [paste]. Our read path bit-bangs from userspace via libgpiod. Identify where the timing budget breaks under scheduling delay, and propose two designs that survive it, with tradeoffs, before writing any code.*

### 10.3 What changes because the model is banned from the product

Nothing about how you *develop*; everything about what you *ship*. Use the agent freely to plan the supervisor, explain `epoll`, draft the FFT feature extractor, and generate labeled test fixtures for your detector. But the shipped tree must honor Section 3.3, and the boundary must be legible: a grader reading your repository should be able to confirm in one minute that the product's only network activity is its own LAN interface. And if you're ever tempted to resolve a hard analysis problem by "just asking a model" at runtime, notice that the temptation *is* the assignment. The hard analysis problem is the part you are here to learn to solve.

### 10.4 Prompt patterns: good vs. useless

| Useless | Effective |
|---|---|
| "make the sensor work" | "`i2cdetect` sees the BME280 at 0x76 but our reads return 0x80 in every field. Here is our init sequence and the datasheet's required mode transition [paste both]. Diff them and identify the missing step; do not rewrite the module." |
| "fix the crash" | "`dmesg` oops attached [paste]. The fault is in our driver's read path, which takes `lock` at line 84. Walk the call path from IRQ context and identify what we are doing there that IRQ context forbids." |
| "make it faster" | "`perf report`: 38% of sampler CPU in `memcpy`. Plan a zero-copy handoff from the capture ring to the consumer given single-producer/single-consumer and a 16 KiB budget. Risks and invariants first; no code." |
| "write the daemon" | "Plan the supervisor: restart policy with backoff, fd hygiene across `fork/exec`, and how we avoid a restart storm when a sensor is physically unplugged rather than crashed. List what you'd log at each decision." |
| "why is the soak failing" | "RSS grew 40 MB over 12 hours; hourly heartbeats attached. Rank hypotheses by likelihood given that event counts are flat but query counts are not, then propose the cheapest instrumentation to discriminate between the top two." |

**Transcript submission.** Identical to Assignment 0, per partner: the raw `.jsonl` session files from `~/.claude/projects/`, copied in as-is. Not summarized. Not exported. Not retyped. Work in a dedicated project directory per partner so the copy is one `cp`. Transcripts are purged locally after 30 days by default, and this project spans more than 30 days, so copy them out at every milestone or raise `cleanupPeriodDays` now. Losing Week-5 transcripts to the default in Week-15 is a foreseeable loss; foreseeable losses are not excused.

## 11 Personalized Verification: Demo Day

Every team demos live; every student defends individually. The session runs roughly thirty minutes per team, in 3 movements.

The demonstration. Your device, running your soaked build, doing its job. At a moment of our choosing, we will inject a fault, most often by unplugging one of your sensors mid-demo. This is announced now so that nobody can call it unfair later. The graded behaviors are graceful degradation, honest reporting through your own interface, and the log line proving the system noticed. A device that freezes, lies, or crashes when hardware misbehaves has failed its user at exactly the moment it existed for.

The individual defense. Separately, each partner answers for their own subsystems: lines of your code ("what wakes this thread?", "what happens to this fd when the child dies?", "why can't this ring overwrite an unread sample?"), one judgment call from your `PROMPTLOG.md`, and one question from across the ownership boundary. Because owning your half deeply does not excuse understanding the other half not at all.

The live modification. Each partner, alone at the keyboard, makes one small previously-unseen change to their own subsystem, starting from the submitted source: a new field in the heartbeat, a changed threshold semantic, a new flag on the query CLI. Small for an author. Revealing for anyone else.

**Why this exists.** For the same reason it exists in the assignments, with the stakes raised. A repository, a report, and a beautiful demo video can all be produced without understanding. Thirty minutes of your own code behaving unexpectedly in front of you cannot. The defense is not an audit bolted onto the project; it's the project's final measurement, and the quantity it measures is you.

## 12 Deliverables (M5)

Submit, via Canvas, a tar file containing:

- **Source and build:** the full repository, with a `Makefile` (or equivalent) and a `README.md` that takes a TA from a clean Pi to your running system without your help. If the TA can't boot it, the TA can't grade it.
- **Documents:** `PROBLEM.md` (the memo, as revised), `DESIGN.md` (as it ended, not as it began, with a short changelog of what M2's version got wrong), and `EVALUATION.md`.
- **Soak evidence:** the raw, unedited logs of the 48-hour run, including the injected fault.
- `CLAUDE.md`, checked in and evolving across the milestones.
- **Per partner:** `PROMPTLOG.md` (6–10 annotated episodes, including one revised plan, one rejected diff, one tool-output debugging loop with real hardware evidence, and one review of your partner's work), `REFLECTION.md` (one page: where the agent was most and least reliable against real hardware; one bug it introduced that you caught, and how), and the raw `.jsonl` transcripts.
- **Git history:** $\geq 40$ meaningful commits spanning the milestones, both partners represented. A pair of "final submission" commits is an automatic process-grade of zero. For both partners.

## 13 Grading

| Component | Method | Points | Scope |
|---|---|---|---|
| Problem memo (M1) | Passes the phone-app test; user and problem concrete; risk named | 5 | team |
| Design document (M2) | Mechanisms justified from requirements; failure-mode table; evaluation plan; ownership map | 10 | team |
| Checkpoint demo (M3) | Both sensors through the real pipeline into real storage; one menu mechanism live | 10 | team |
| Final system (M4/M5) | All five guardrails met; runs from the README; survives our fault injection; 48-hour soak evidenced | 30 | team |
| Evaluation report (M5) | Quantitative, reproducible, includes the required mechanism comparisons, honest about limits | 15 | team |
| Process | Your transcripts on the five-level repository-corroborated scale of Assignment 0 (level 5 = 10 pts, each level down −2), read alongside your `PROMPTLOG.md` and commits | 10 | individual |
| Demo-day defense | Subsystem fluency, cross-boundary question, `PROMPTLOG` defense, live modification | 20 | individual |
| **Total** | | **100** | |
| The Reach | Deployment with the real user (Section 14): attempted for the craft, not for points | 0 | optional |

Two clarifications. First, the memory-bug discipline of the assignments carries over: the systems core must be clean under `-Wall -Wextra -Werror`, and ASan/valgrind findings in components where they apply cap that component's contribution to the final-system score at 50%, exactly as in Assignment 0. A device that leaks for 48 hours and happens not to die has not passed the soak. It has outrun it. Second, the transcript scale is applied per partner, to that partner's own sessions. Working in one shared login defeats this and will cost *both* partners.

## 14 The Reach: Put It in Front of the User

This carries no points. Do it anyway. Install the device where the problem actually lives (the parent's kitchen, the shop floor, the greenhouse) for at least a week, and add one page to your reflection about what the user did with it, what surprised you, and what you'd change now that a

stranger has depended on your code. Nobody is checking whether you do this. Which is exactly what makes doing it mean something. Software that has been used by someone who does not care how it works is a different kind of artifact from software that has only ever been demonstrated, and building the first kind, even once, changes how you build everything after. And if you finish the week wondering whether the thing on that shelf could become something more than a course project? That is not an accident, and my office hours are open.

## 15 Whereto, from Here?

Some of you will finish this project, look at the device sitting in its food-container enclosure and think: *hang on, would somebody pay for this?* Hold onto that thought. It may be worth more than the grade.

Startups are rarely founded on a dazzling idea. They are founded on a problem somebody already has and is already solving *badly*; usually with a spreadsheet and a bad habit. Finding one of those is the hard part, and you were made to do it in Week 5, in front of a skeptical reader. Your problem memo is structurally the opening of a pitch: who the user is, what the problem costs them, why the obvious answer fails, and what you propose instead. The same questions get asked later in nicer rooms with worse coffee.

The next thing you have is rarer than you'd guess. An enormous quantity of software talent can wrap a web API and call it a company. Much less of it can make a physical thing stay alive, unattended, in a stranger's basement, through a power blip and a sensor that quietly fell off in March. That is a real barrier to entry, and you now have 48 hours of logs saying you cleared it.

You also have evidence. When a customer or an investor asks the only question that matters ("does it work?"), most people answer with adjectives. You can answer with a measured false-positive rate and a limitations section you wrote when nobody was forcing you to be honest. That habit will outlive every technology named in this handout.

Now the part I owe you. The distance from a working prototype to a shippable product is long: sourcing, enclosures, certification if you emit anything interesting, unit economics, a support inbox somebody has to read on a Sunday. No device survives contact with 50 strangers unchanged. Assume version two exists before version one leaves the bench.

But if you are ever going to try this, the cheapest moment of your life is now! You have no payroll and no mortgage. Plus there's a campus stuffed with resources you are already paying for: the institute for entrepreneurship, pitch competitions, prototyping space, an IP office, faculty who will read your business plan for the price of asking nicely. You also have a co-founder (in your teammate) whose code you have already reviewed for fifteen weeks, which is more due diligence than most founding teams ever manage on each other.

And if the answer turns out to be no? That is a perfectly good answer. You will still have built a real thing for a real person, which was the point the entire time. My door is open either way. Bring the food container.

## 16 Suggested Schedule

- **Weeks 3–4:** Form the team. Ideate widely (Section 4.1), then apply the guardrails as filters and Section 5 as the reality check. Order hardware immediately. Shipping time is the most common silent schedule-killer. M0.
- **Week 5:** Problem memo (M1). Get both sensors electrically alive on a breadboard while the memo is under review.
- **Weeks 6–7:** Design document (M2). Argue about the architecture now, in ink, while changing it is free.
- **Weeks 8–10:** Build the pipeline: capture, storage, supervision. Checkpoint demo (M3). And start soak-style overnight runs *now*, at small scale; every leak found in Week 9 is a crisis avoided in Week 14.
- **Weeks 11–13:** The analysis engine, the interface, the mechanism-comparison experiments, and the evaluation measurements.
- **Week 14:** Feature freeze (M4). Begin the graded 48-hour soak with margin to run it twice. You will want to run it twice.
- **Weeks 15–16:** Final submission (M5), demo day (M6), and, if you are wise, The Reach.

## 17 Hardware Notes

- Boards. Any Raspberry Pi with the 40-pin header works. A Pi 4 or 5 is the comfortable choice; a Zero 2 W is fine for the deployed device but slow to compile on, so develop on something larger or cross-compile.
- GPIO is 3.3 V and unforgiving. The Pi has no 5 V-tolerant inputs and no analog inputs at all. The HC-SR04's echo pin needs a voltage divider; analog sensors need an ADC (an MCP3008 over SPI is the classic answer, and SPI is a fine thing to have met). Wire with the power off. A Pi killed by a hot-plugged jumper in Week 13 is a schedule event, not a hardware event.
- Power and storage are design inputs. Brownouts corrupt SD cards, and SD cards wear. Your storage layer's `fsync` discipline, write batching, and recovery story (Section 3.2, item D) are not academic. The soak test runs on this exact hardware.
- Budget. Plan on roughly $50–80 per team including the Pi if you own nothing; individual sensors run $3–15.
- Appendix A lists common sensors with their interfaces and, more usefully, the operating-systems lesson each one naturally teaches.

## 18 Late Submission

The late submission policy for CS370 is available from the course website at `https://www.cs.colostate.edu/~cs370`. Milestone dates for the term project are firm in a way assignment dates are not, because each milestone protects the ones after it.

## 19 The Spirit of This Project

You are required to submit your own work, and you are required to use Claude Code to help you produce it, and, new this time, you are required to keep the model out of the product itself. These three requirements are one requirement wearing three hats: the work must be *yours*, where yours has never meant unaided and has always meant understood, defended, and extendable by you.

There is no version of this project worth having that you didn't build. A repository assembled by wholesale generation will be a stranger to you at the defense, where thirty minutes with your own code costs more than the generation saved. A product that quietly phones a model has outsourced exactly the part of the semester you were supposed to keep. And a soak log that was tidied after the fact is the one artifact in this course we are best equipped to notice, because real systems are never that tidy.

But hold this next paragraph together with the previous one, because they are not in tension: understand what is *not* being demanded of you. The device you ship in Week 15 will be smaller than the one you imagined in Week 5. A sensor will have been substituted, a feature cut, a claim narrowed. The enclosure will be a food container. One graph in your evaluation will end in a shrug. Good. That shrinkage, honestly accounted for, is not the project failing. It's the project working, because it's the exact shape of every real system you will ever ship. Perfection is not on the rubric and never was. What is on the rubric is a device that runs, numbers that are real, limitations that are named, and two people who can each explain the distance between what they wanted to build and what they built. The teams that struggle here are almost never the ones facing the most constraints. They are the ones who spend the semester mourning the project they couldn't do instead of building the one they could. Don't join them. When the ideal sensor turns out not to exist at your budget, or the data you need refuses to exist at your scale, the assignment at that moment is not to document an obstacle. It is to turn to what is actually on the bench and ask what *can* be done with it. Which is, not incidentally, the question the entire history of good engineering consists of asking.

It's also worth saying plainly what this project is a rehearsal for, because it is not really the Raspberry Pi. Every project you ship for the rest of your career will have a stakeholder who cares about outcomes and not mechanisms, a budget that forecloses the elegant option, a teammate whose code you must both trust and verify, a tool that is confidently wrong at the worst possible moments, a deadline that does not move, and a world that declines to be ideal about any of it. This project is that, at one-tenth scale, with gentler consequences and better office hours. Learning to do good work *inside* those conditions, not despite them, and never while waiting for them to lift, is the transferable skill. The drivers and ring buffers are the vehicle. The judgment is the cargo.

So do it the way it is designed to be done. Find a person with a problem. Build them a machine that is awake when they are not. Use the agent as the collaborator it is (plan first, keep

the diffs small, demand evidence, merge nothing you can't explain) and write the thinking parts yourself. Accept the constraints as the terms of the craft, and let the limitations section tell the truth without flinching. At the end you'll have a device on a shelf, doing a job, imperfectly and honestly, for someone who is glad it exists. Most courses can't offer you that. This one is trying to.

## 20 Version History

This section will reflect the change history for the assignment. It will list the version number, the date it was released, and the changes that were made to the preceding version. Changes to the first public release are made to clarify the assignment; the spirit or the crux of the assignment will not change.

| Version | Date | Comments |
|---|---|---|
| 1.0 | 8/3/2026 | First public release of the assignment. |

# A Sensor Starter List

Quick flag: anything marked **(!)** has 5V lurking in it somewhere, so treat those interfaces with a little extra respect.

Here's the thing about the Pi's GPIO pins: they only speak 3.3V logic, and there's zero over-voltage protection built in. Feed one a 5V signal directly and it can just... die. Silently. No warning light, no smoke, sometimes not even a clue until you notice nothing's responding anymore. And sometimes it doesn't stop at the pin, it takes the whole SoC down with it. That's exactly why the HC-SR04 needs a voltage divider on its echo line. The trigger pin is fine with 5V, but the echo pin is not, and if you wire them the same way because *"eh, they're both just signal pins,"* congratulations, you've found the single most popular way to lose a Pi in Week 13.

But power-rail mistakes are even less forgiving. Swap VCC and GND, or let a 5V rail touch the 3.3V rail while you're poking around with a multimeter, and your board usually doesn't warn you first. It just stops working. No gradual decline, no second chances.

So here's the move: build anything 5V-adjacent on a breadboard first. Check your voltages with a meter before the Pi ever gets plugged in. And always, always wire things up with the board powered off.

None of this is meant to be a hard boundary, think of it more as a floor. Weird, exotic sensors are still welcome at the table, as long as the guardrails hold.

| Sensor | Interface | What it naturally teaches |
|---|---|---|
| BME280 (temp / humidity / pressure) | I2C | Multi-rate polled sampling; an approachable first target for a character driver (menu A) |
| DHT22 (temp / humidity) | bit-banged single-wire | Microsecond timing windows from userspace, and therefore a lesson in why kernels exist |
| HC-SR501 PIR motion | GPIO edge | Interrupts, debouncing, and the polling comparison (menu B) |
| HC-SR04 ultrasonic **(!)** | GPIO trigger/echo | Echo-timing jitter as a measurable scheduling artifact (menu C); voltage dividers |
| MPU-6050 IMU | I2C (400 kHz), FIFO | High-rate sampling without drops; ring buffers; FIFO-overflow accounting (menu F) |
| Photoresistor + MCP3008 | SPI ADC | SPI, and sampling an analog world on a board with no ADC |
| INMP441 microphone | I2S | Real streaming rates; DMA; buffer discipline (menu F) |
| Reed switch / door contact | GPIO | Events, state machines, and the cheapest ground truth you will ever get |
| MH-Z19 $CO_2$ | UART | Serial protocols, framing, checksums, and vendors' creative datasheets |
| Load cell + HX711 | bit-banged serial | Timing plus calibration: where systems code meets the physical world's noise |

## B Sample Team `CLAUDE.md`

```
# <project-name>: team rules

## Commands
- Build: `make`        Cross/deploy: `make deploy PI=pi@<host>`
- Tests: `make test`   Sanitizers: `make asan`   Valgrind: `make memcheck`
- Soak sanity: `make soakcheck` (1-hour miniature of the 48h run)
- A change is DONE only when build, test, and asan pass. Show output.

## Hard constraints
- The PRODUCT makes no network calls except serving its own LAN
  interface. No LLM APIs, no cloud inference, no pretrained models.
  The intelligence in analysis/ is ours. This is a graded boundary.
- Every daemon must be supervisable: clean exit codes, no orphaned
  fds across restart, heartbeat within 60s of start.
- Every allocation checked; every syscall's error path handled and
  logged. The 48-hour soak is the test suite of last resort.
- NEVER weaken, skip, or delete a test to make the suite pass.

## Ownership
- alice/: capture pipeline, kernel-facing code (src/capture, src/drv)
- bob/:   storage layer, analysis engine (src/store, src/analysis)
- Shared: supervisor, interface. The agent edits outside the current
  session owner's area only when told explicitly whose session this is.

## Style
- Systems core: C17, -Wall -Wextra -Werror, no VLAs.
  goto-cleanup for multi-resource functions.
- Python permitted only under tools/ and ui/. No graded mechanism
  may live there.
- Smallest diff that passes. Do not refactor unrelated code.

## Workflow
- Multi-file or algorithmic change: plan first, wait for approval.
- Hardware bugs: paste real evidence (dmesg, timing capture,
  /proc/interrupts). No fixes proposed from a verbal description.
- Commit only from a green state; message format "M<n>: <what>".
```

# C Problem Memo Template

```
# Problem memo -- <team name> (<partner 1>, <partner 2>)

## The user
A person or place, named or nameable. Who will this sit next to?

## The problem
What goes wrong, how often, and what it costs (money, worry,
ruined batches, missed warnings). Observable, not hypothetical.

## Why a device
The 3 a.m. test: why must something be physically present and
always awake? Why doesn't a phone app already solve this?

## The sensors
Which two (or more), and how they COOPERATE  (fused, correlated,
or one pipeline) rather than merely coexist.

## The mechanisms
First guess at two items from the Section 3.2 menu, one sentence
of justification each. Allowed to change by the design doc.

## The risk
The single thing most likely to sink this project. Name it now;
it is cheaper to meet in Week 5 than in Week 14.
```