Assignment 3: Make It Visible

CS 356 UT Austin · Fall 2026 · Student handout (draft)

When a network misbehaves, the first question is always the same: what happened? You cannot rewind a network to look, so the only answer is whatever it recorded at the time. Good monitoring turns an outage into a quick diagnosis; without it, you are left guessing. **The catch: you choose what to count, where, and how much to keep, before anything breaks.**

In A1 and A2 your program ran the network. In A3 we run it, and your program's job is to be the thing that was counting. The course network forwards every packet, links fail, queues fill, and your switches can only **watch**. Somewhere in that network we have hidden **eggs**: things that happen in the network, which you can catch only if you were measuring the right thing at the right place.

Watching is not free, and that is the whole game. A switch's controller sees only a sample of events, never everything. Every report it sends spends a byte budget and rides behind customer traffic, so on a busy link it waits, and in a full queue it is lost first. And it all ends in one small log, the only thing you will ever see. So every byte is a choice. Count in the data plane, or sample to the controller? Summarise at the switch, or ship raw events? Write on a timer, or only when something changes? Put your logging switch near the trouble, or in the middle where every report has a short trip? No answer fits every egg; finding the ones that fit is the assignment.

1 The hunt: truth runs and egg claims

You write two programs. `switch.wasm`, your instruments, starts from your A2 program and runs on every switch, deciding what to notice, count and report; it names **one logging switch**, which writes **your log** in a format you design. `detect.wasm`, your detector, reads only your log, after a run, and answers **true or false**. Figure 1 shows how the two kinds of attempt fit together.

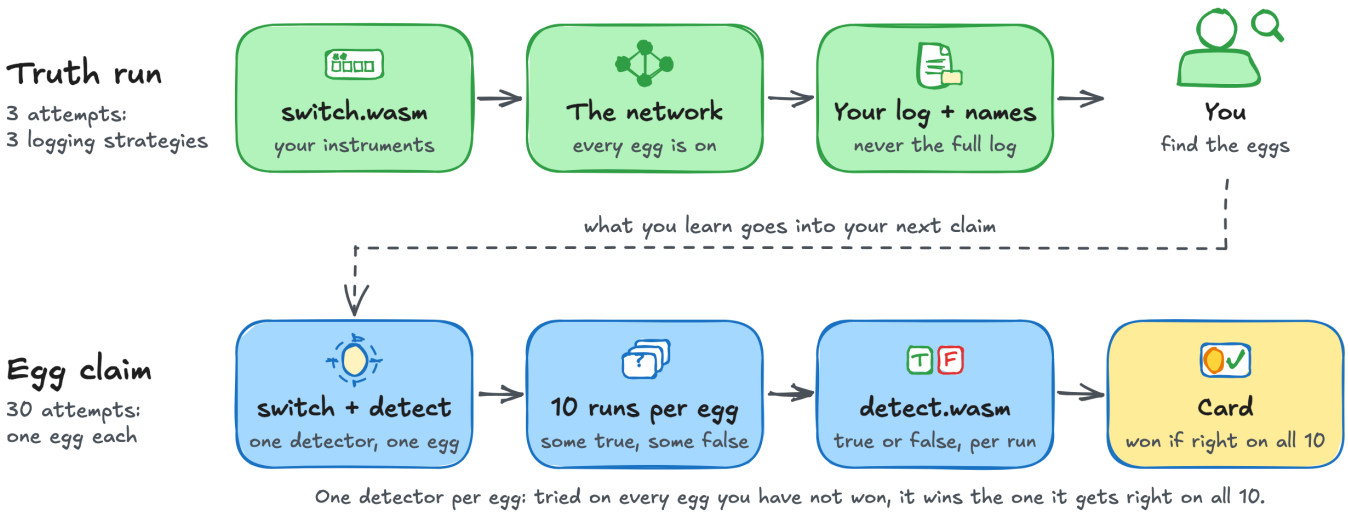


Figure 1: Truth runs and egg claims.

Truth runs (3 per team). Submit a `switch.wasm` and get the next run: A, then B, then C. You get back **your log**, written under a bigger cap of *[T]* KB, and a list of **which address range is which named customer**. You never see the full log, only what your instruments caught.

Every egg is true in all three runs; the runs differ in timing and load. Nothing your program does changes what happens in a run.

Egg claims (30 per team). A claim is both programs, and its detector is built for **one egg**. You do not say which: for every egg you have not won, the server runs the claim on that egg's 10 hidden **claim runs**, and you win the egg your detector is right on, **all 10** times. No logs come back, only a **card** listing the eggs you have won, each by number, category and name, and silent about the rest. A won egg stays won, and the runs never change, so an unchanged resubmission gains nothing. In the claim runs, customers' address ranges are **dealt afresh every run**: recognise Netflix by how its traffic looks, not by where it was in your truth runs.

2 Eggs

An egg is a **yes-or-no question about what happened in a run**, such as "Did the link between 4 and 5 go down?". It names what it is about (a link, a switch, a customer), and is never vague ("the network was slow"), never just a time, and never about your own program.

Finding them takes three skills. **Tell your customers apart**: once ranges are reshuffled, only their traffic says who is who. **Watch the physical network**: links and switches that go down, or stay up but work badly. **Read the traffic**: who sends what, to whom and when, including trouble that no failure announces.

The eggs are hidden. There are 20, and we do not say what they are. Each is true in all three truth runs, so the eggs are among what happens in all three, but not all of that is an egg, and you see only what your instruments catch. In its claim runs an egg is **true** in some and **false** in the rest, on the same network with every other egg on: only that egg changes (Figure 2). A false run is either **off**, where the thing never happens, or a **near miss**, where something close happens instead (the cut lands on another link; the link comes back). Each false run is built to fool one particular lazy detector, one that checks only part of the egg. For example, "a neighbour went silent" passes every truth run for the 4–5 cut, then says yes when 1–4 is cut instead, and wins nothing. "4 and 5 stopped hearing each other, both ways" wins. Always-true wins nothing either.

Three practice eggs. The kit has a **practice network** (not the real one) and the simulator, to run on your machine as often as you like. There you get the **full log** too, so it is where you check that your instruments see what really happened. Its three eggs are not hidden, one per skill:

- **The broken link** (Failure). Partway through, link 1–2 goes down and stays down. *Off:* no link goes down. *Near misses:* another link goes down; 1–2 goes down and comes back.
- **The long pause** (Customer). Partway through, Spotify stops sending for at least 10 s, then carries on. *Off:* Spotify never pauses. *Near misses:* Spotify pauses for only 2 s; another customer pauses for 10 s. Ranges are reshuffled, so find Spotify first.
- **The scanner** (Traffic). Partway through, one host sends a single small packet to each of at least 50 addresses in another customer's range, all within 5 s. *Off:* no host does. *Near misses:* one host sends many packets to just a few addresses; one host reaches 50 addresses, but spread over the whole run.

We solve the broken link for you: the kit has a switch program and a detector that win it, so you can read the whole chain, from a report to a true or false. The other two are yours. Practice eggs earn no points. Each comes with **10 practice claim runs** and their answers, and for all three the **practice checker** scores a claim on them as the server would: the only place to test the false side of a detector before it counts (Figure 3).

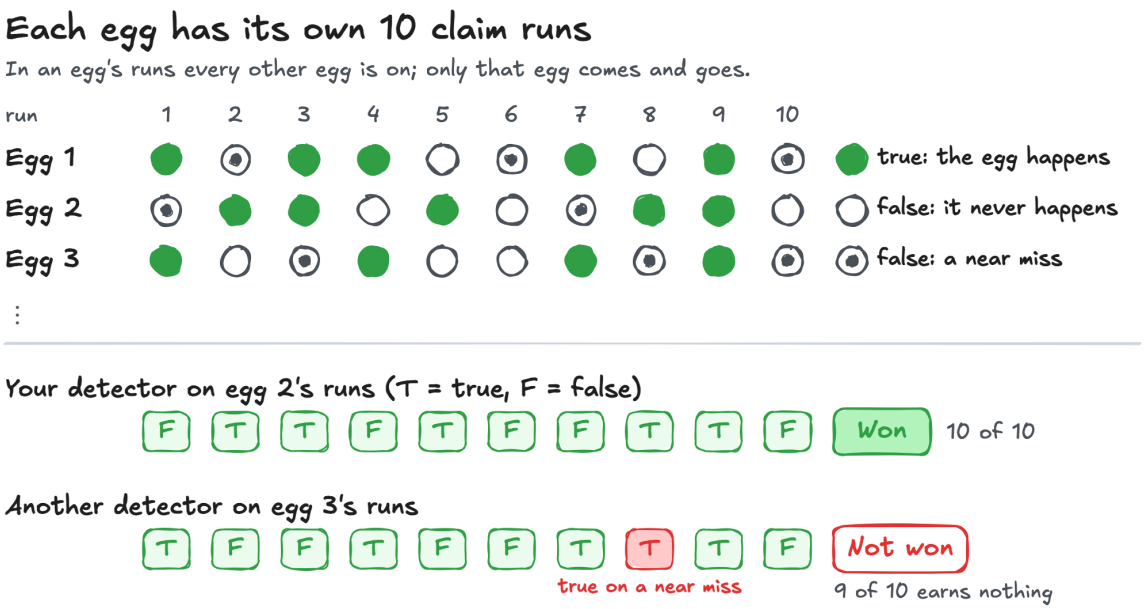


Figure 2: Each egg's claim runs, and two detectors.

3 Your log

Your log is the only thing your detector ever reads. Whatever an egg needs must get into it, arrive in time, and fit.

One switch writes it. You choose which: the first switch to ask keeps the job for the whole run. Only its writes count, so whatever other switches see must reach it as reports, which spend their switch's byte budget and wait behind customer traffic. Each controller sees at most 80 events a second, so what it reports is already a sample. Where you put the logging switch decides what arrives, and how late. You are not given the map, but switch ids and links are the same in every run, so your first truth run can find it.

Two caps, on purpose. How much your log holds depends on the run. In a truth run it holds *[T]* KB: you do not yet know what the eggs are, so you need room to look widely. In a claim it holds only *[S]* KB (the value is published with the submission server): by then you know what one egg needs, and the log should carry that and little else. Explore wide, then distil: that is the arc of the assignment, and the gap between the caps is where it bites. Logging that fit easily in a truth run will overflow in a claim unless you cut it down. A write that does not fit is lost whole, and a log that writes down every event fills in the first few seconds, so an egg that comes late never reaches your detector.

It holds only what you wrote. Nothing is added for you: if you did not write the time, or which switch saw it, you do not have it. The format is yours to design, and `detect.wasm` must read it back. Your switch program can instrument for several eggs at once, but they all share this one log. Summarise, sample, and write only what your detector will read. The kit's logging SDK has the details.

For ideas on fitting a lot into a little space, these are good starting points (optional, suggestions only):

- sFlow: how switches sample packets instead of keeping them all.
- Reservoir sampling: a fair sample from a stream when you do not know how long it is.

- Count–min sketch and Misra–Gries: counting many things, and finding the biggest, in fixed memory.
- Monitoring Distributed Systems (Google SRE book): what is worth measuring.

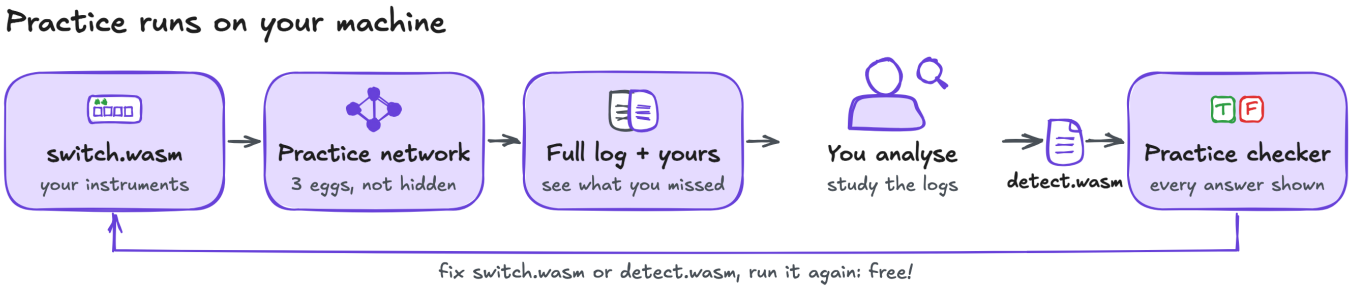


Figure 3: The practice loop.

4 Making your attempts count

You have two budgets, three truth runs and thirty claims. Spend them in that order: learn in the truth runs, then claim.

Truth runs: learn. They are the only time you see the real network, and there is no fourth. Plan each one before you spend it: what you want to learn, and what you will log to learn it. Debug your logging on the practice network first; a truth run spent finding a bug in your own code is wasted. Your first run is also where you find the map and decide where your logging switch belongs. Truth runs give you a bigger log ($\lfloor T \rfloor$ KB) to explore with; keep every one, because they are all your evidence.

Claims: commit. Each claim is a bet on one egg, the egg its detector was built for. The runs never change, and the card says only what you won, never why you missed. So test the false side before you submit: would your rule say yes if the cut were on another link, at another time, for another customer? Try it on the practice runs and against your truth logs. Then cut your logging down to what this one detector reads: the wide logging that served a truth run will not fit in a claim's log.

5 What you get

Everything from A2: the simulator, the Rust SDK, and your own A2 switch, which is your starting point. It still runs on every switch, but in A3 its forwarding is ignored; what it notices, reports and logs is what counts. New in this kit:

- **The practice network** and its **three practice eggs**, each with 10 practice claim runs and their answers. It is a different, smaller network from the one you are graded on, and you can run it as often as you like.
- **The full log, locally.** On the practice network, every run gives you the full log and your log side by side, so you can check what your instruments caught against what really happened. `simlog_dump` prints a full log as text, and the **replay viewer** draws it.
- **A solved practice egg**: a switch program and a detector that win the broken link, as source, to start from.

- **The practice checker**, which runs your two programs on a practice egg's 10 runs and prints a card like the server's, with every run's answer shown. It also checks that both programs load, so a claim it accepts will not come back invalid.
- **The logging SDK**: how to name your logging switch, write to your log, send reports under the budget, and read your log back in `detect.wasm`.

6 Grading

The 20 eggs are worth **200 points** in all. Each egg is **easy**, **medium** or **hard**, and the harder it is, the more points it is worth. Your points are those of every egg you have won by the deadline; a won egg stays won.

150 points is 100% on A3. Below that, your grade is your points out of 150.

A4 bonus. Your A3 points also earn a bonus in A4, at one of three levels:

Level 1	100 points or more
Level 2	150 points or more
Level 3	200 points: every egg

What each level earns comes with the A4 handout.

7 Using AI

Use it however you like. AI knows the textbook ways to measure a network, and will write you a logging switch or a detector in a minute. What it has never seen is this network, or what your truth runs showed. Deciding what to watch, and what your detector should believe, is yours, and the card will not tell you when you got it wrong. Nor can AI sit the quiz, where you explain what your instruments measured and why your detector answers as it does.

8 Logistics

Server opens	Saturday, October 24, end of day, for truth runs and claims.
Due	Monday, November 2.
Teams	2 students.
Submitting	How to request a truth run and submit a claim, and the claim log cap, are published in `SUBMITTING.md` when the submission server opens.