

1 Plan de développement — Caracat

Our work on Caracat went from early February to late April 2026, so about three months in total. The two of us divided things based on what made sense as the project progressed: Sheharyar took on the comparative analysis between Caracat and Caracal, built the architectural diagrams, and ran all the functional tests. Seyeong dug deeper into the static code analysis, traced the program flow through Caracat's internals, drew up her own flowcharts, and later wrote up the Analyse and Conception parts of this report.

The first couple of weeks were mostly reading. We went through the main research papers — Paris Traceroute, YARRP, and MDA — to understand why tools like Caracat exist in the first place and what problems they are trying to solve. We also looked at the Scamper paper and the Internet Topology Discovery survey by Donnet and Friedman, mostly to get a sense of how other tools in this space work and where Caracat sits relative to them.

After that we got into the actual code. We cloned both Caracat and Caracal and tried building them. Caracat was easy enough, just `cargo build` and it compiled. Caracal gave us more trouble — it depends on CMake and Conan, and getting libtins and the rest of the C++ dependencies working on macOS was not straightforward. We also ran `cargo doc` to generate Caracat's API documentation, but honestly, it was not very helpful. Most modules had barely any comments, and there was nothing explaining the overall design. The supervisor had already pointed this out in his own notes, so at least it confirmed we were not the only ones who found it hard to navigate.

The bulk of our time went into actually reading through the source code of both tools, module by module. We looked at the sender, receiver, parser, builder, checksum, timestamp, rate limiter, neighbour resolution — pretty much everything. What we were really trying to figure out was whether Caracat genuinely reimplements what Caracal does, or if there are gaps beyond what is obvious from the outside. We kept track of what we found as we went, and that became the basis for the analysis documents we wrote later. On her side, Seyeong spent a lot of time specifically tracing how data flows through the code and produced flowcharts showing the full sending and receiving pipelines step by step.

While we were still going through the code, we also started running tests to compare the two tools directly. We came up with five experiments — single probes at a fixed TTL, full traceroute runs, stateless Yarrp-mode probing, and so on — all against Google DNS. We ran the same thing on both Caracat and Caracal and compared what came back. The results matched, which was reassuring: same routers at every hop, same RTT values. That gave us more confidence that the core engine really is the same across both implementations.

For diagrams, we ended up producing two different sets. Sheharyar created six diagrams in Excalidraw covering the system architecture, module dependencies, the comparison between the two tools, and the key internal mechanisms. Seyeong made her own flowcharts

that focus more on the program flow — showing exactly what happens when a packet gets built and sent, or when a reply comes in and gets parsed. They ended up being complementary, which was not really planned but worked out well.

On April 1st we met with the developers who actually maintain Caracal and use it in production at Measurement Lab. We went through our analysis and they asked us about things like protocol support, how the checksum behaves for IPv6, whether prefix filtering could be added, and what it would take to get features they care about. Seyeong's flowcharts came in handy here because the developers wanted to see exactly how packets move through the system. The meeting was useful — it confirmed we had understood the code correctly, and it also gave us a clearer picture of what real users actually need from these tools.

The last few weeks were about writing. We put together a detailed comparison document, a shorter version of it, a breakdown of each tool's output format, and a technical document going deep into six areas of the codebase. All of these feed into this report and the most detailed ones are in the annexes.

2 Compte rendu — Caracat

2.1 Codebase Exploration

When we started, we did not know much about Caracat beyond the fact that it was supposed to be a Rust rewrite of Caracal. So the first thing we did was clone both repos and try to get them running. Caracat compiled without issues — just `cargo build` and we were done. Caracal was another story. It uses CMake with Conan for its C++ dependencies, and getting libtins and everything else to build on macOS took some back and forth. It was worth the effort though, because being able to run both tools side by side turned out to be essential for everything that came after.

We also tried generating documentation with `cargo doc --open`. The result was pretty bare. Most of the modules had little to no inline documentation, and there was nothing resembling an overview of how the system is structured. If we were having trouble figuring out the architecture just from reading the docs, then anyone else coming to this codebase for the first time would have the same problem. Professor Friedman had actually noted this early on and suggested that an architectural overview should be added to the README.

From there we just started reading code. Caracat is around 2,000 lines spread across roughly 20 files, so it was feasible to go through all of it. We started with the data models — the Probe and Reply structs — since those are what everything else revolves around. Then we moved on to the sender and receiver, and left the trickier parts like checksum and timestamp encoding for last, once we had enough context to make sense of them. We took notes along the way, and those notes eventually turned into the analysis documents we wrote up later.

2.2 Core Mechanisms

What makes Caracat interesting, technically, is how it pulls off high-speed probing without storing any state about individual probes. All the information the receiver needs to match a reply back to its probe is encoded directly in the outgoing packet. We spent quite a bit of time going through how exactly this works, so we break it down here.

2.2.1 Checksum Validation

Caracat needs a way to tell whether an incoming ICMP reply actually corresponds to a probe it sent, as opposed to some other traffic on the same machine. It does this with a checksum computed from four values — the instance ID (a random number picked at startup), the destination address, the source port, and the TTL:

```
checksum = NOT(instance_id + dst_addr + src_port + ttl) MOD 65535
```

This gets stuffed into the IPv4 Identification field of the outgoing packet. Later, when a

router along the path returns an ICMP Time Exceeded message, it includes a copy of our original IP header. The receiver pulls the ID field out of that quoted header, recomputes the checksum, and checks if they match. If not, the reply gets thrown away.

One thing we noticed is that this does not really work for IPv6. There is no Identification field in the IPv6 header, so there is nowhere to put the checksum. Right now, the code just accepts ICMPv6 replies without doing the full validation. The developers know about this but have not fixed it yet.

2.2.2 Timestamp Encoding

To compute round-trip time, you need to know when each probe was sent. The problem is that there are only 16 bits available to store this — Caracat uses the ICMP checksum field for ICMP probes and the UDP checksum for UDP probes. A full timestamp clearly does not fit, so the code uses modular arithmetic:

$$\text{encoded} = \text{timestamp} \bmod 65535$$

On the receive side, the current time gets the same modulo applied, and the difference gives the RTT. This works as long as the reply comes back within about 6.5 seconds, which covers basically any real internet path.

Something else we picked up on: the send timestamp is captured in user space, not kernel space. So there is a small gap — usually 1 to 50 microseconds — between when the timestamp is recorded and when the packet actually hits the wire. The receive timestamp comes from pcap at the kernel level and is more accurate. At the millisecond scale this difference does not really matter, but it is the kind of thing worth knowing if someone ever tries to use these tools for very precise timing measurements.

2.2.3 ICMP Demultiplexing

ICMP does not have port numbers. With TCP or UDP you can look at the port to figure out which application a packet belongs to, but ICMP gives you nothing like that. So how does Caracat figure out which reply goes with which probe?

It comes down to how ICMP Time Exceeded messages work. When a router drops a packet because the TTL hit zero, the RFC says it has to include the original IP header plus at least the first 8 bytes of the transport header in the error message. That quoted packet is basically a snapshot of what we sent.

Caracat's parser goes through this in layers: first the outer IP header to get the router's address, then the ICMP header to check that it is actually a Time Exceeded, then the inner IP header from the quoted packet to pull out our destination and the checksum from the ID field, and finally the inner ICMP or UDP header to get the source port and the encoded timestamp. After all that, it has everything it needs to match the reply to the

right probe and compute the RTT. And none of this required storing anything when the probe was originally sent — that is the whole point of the stateless design.

2.2.4 Paris Traceroute and Flow Consistency

Load balancers are everywhere on the modern internet. They distribute traffic across multiple equal-cost paths using a hash of the flow identifier — basically the combination of source IP, destination IP, protocol, source port, and destination port. If you change any of those between probes, you might end up on a different path each time, and your traceroute will show connections between routers that are not actually neighbours.

Caracat handles this by keeping the source port fixed across all TTL values for a given destination. Since the source port feeds into the load balancer’s hash, same port means same path. For ICMP probes the port is encoded in the ICMP Identifier field, and for UDP it is just the actual source port.

We saw this in practice during our tests. With a fixed source port, both tools consistently showed the same path. When we changed the port between runs, hops 3 and 4 sometimes came back with slightly different IPs — different interfaces on the same routers, reached because the load balancer picked a different path. Exactly what the theory predicts.

2.3 Caracat versus Caracal

A big part of what we were trying to answer was: is Caracat actually a faithful rewrite of Caracal? And what does it mean that Measurement Lab considers it “not production ready”?

At the engine level, they are the same. We went through both codebases and mapped the files against each other. Sender, receiver, parser, builder, checksum, timestamp — they all use the same algorithms, the same formulas. The data structures line up too. Both have a Probe with five fields (destination, source port, destination port, TTL, protocol) and a Reply with the same attributes (capture timestamp, reply source, ICMP type/code, quoted probe fields, RTT, MPLS labels).

Where they differ is the interface. Caracal is a full command-line tool. You pipe in CSV probe specs on stdin, it does its thing, and you get CSV results on stdout. It has a Prober that orchestrates the sending and receiving, prefix filtering via longest-prefix matching so you can exclude or include IP ranges, and statistics tracking that tells you how many packets were sent, failed, or filtered. This is what makes it fit into the shell-based pipelines that Measurement Lab runs in production.

Caracat does not have any of that. It is a library. You import its modules into your Rust code and call functions directly. There is no main binary, no CSV parsing, no prefix filtering, no stats. The repository has two example programs — `traceroute.rs` and `yarrp.rs` —

but those are just demos, not production tools. This is not an oversight; it is the design intent. Caracat was built to be used as a dependency by other Rust programs, which can call its probing functions directly without the overhead of spawning a subprocess and parsing text.

We estimated that bridging this gap would take around 500 lines of code: maybe 80 for CSV I/O, 150 for a prober orchestrator, 100 for prefix filtering, 50 for statistics, plus a small main binary. That is not a lot. The hard part — building packets correctly, parsing nested ICMP responses, computing checksums, encoding timestamps — that is all done and working.

So “not production ready” depends on your perspective. If you need a CLI tool that slots into shell pipelines, then yes, Caracat cannot replace Caracal today. But if you are writing a Rust application that needs to send and receive probes directly through library calls, Caracat already provides everything needed.

2.4 Functional Testing and Validation

Reading code only tells you so much. We wanted to actually verify that the two tools produce the same output when given the same input. So we set up a series of tests, all run on macOS with Apple Silicon over a residential broadband connection.

We ran five experiments:

1. **Single probe at TTL=5.** We sent one ICMP probe to 8.8.8.8 with both tools. Both got a reply from 173.194.123.136, a Google edge router, ICMP type 11 (Time Exceeded), RTT of 5.7ms.
2. **Full traceroute (TTL 1–15).** Both tools found the same path: local gateway (10.1.0.1), ISP (146.70.194.1), two M247 Paris transit routers (83.97.21.x, 89.44.212.x), Google edge, a silent hop, then the destination. The small IP differences at hops 3–4 across runs are from load balancing — different interfaces on the same routers.
3. **Yarrp stateless mode.** We used Caracat’s yarrp example to test its stateless probing path. It captured replies, decoded the quoted fields correctly, and reported consistent RTTs.
4. **UDP probes.** Both tools correctly identified the ICMP Destination Unreachable (type 3, code 3) response you get when a UDP probe reaches a host with no listener on that port.
5. **IPv6.** We traced to Google’s IPv6 DNS (2001:4860:4860::8888). Both tools built valid ICMPv6 packets, parsed the Time Exceeded replies, and found the full path.

Everything matched. The only differences were cosmetic — Caracal writes CSV, Caracat’s examples print human-readable text. The underlying data is the same.

One thing we noticed across all runs: hop 6 never responded. This is not unusual. Some routers are configured to not send ICMP Time Exceeded messages, either to save processing or as a security policy. Both tools handled it fine, just showing a gap.

Test	Caracal	Caracat	Match
Single probe (TTL=5)	173.194.123.136	173.194.123.136	
Full path (hops)	7	7	
RTT	5.7ms	5.7ms	
UDP handling	Type=3, Code=3	Type=3, Code=3	
IPv6	Working	Working	

Table 1: Summary of functional test results comparing Caracal and Caracat.

2.5 Limitations and Improvement Suggestions

We found a few things that are either missing or could be done better in the current version of Caracat.

Protocol support. Right now it handles ICMP, ICMPv6, and UDP. TCP and DCCP are not there — there are TODO comments in the code for them. Sending a TCP SYN probe would not be too difficult to implement, the header construction is fairly standard. The hard part is on the receive side. When a SYN reaches the destination, you get back a SYN-ACK, which is a transport-layer response — not an ICMP error with a quoted packet inside. So the whole mechanism Caracat uses to recover probe information from quoted headers simply does not apply. Solving this properly would require rethinking how validation works for TCP, which is more of a design question than a coding task.

ICMPv6 validation gap. As we discussed earlier, the checksum validation depends on storing a value in the IPv4 ID field, which has no equivalent in IPv6. ICMPv6 replies are currently accepted on trust. For small-scale use this is fine, but for large campaigns on shared machines, there is a real risk of incorrectly matching replies from other processes. Some alternative encoding scheme for IPv6 would be needed to close this gap.

pcap coupling. Professor Friedman pointed out early in the project that Caracat calls pcap directly rather than going through a Rust trait. This makes unit testing difficult — you cannot test the probing logic without an actual network. If pcap were behind a trait, you could swap in a mock implementation for tests. It is a standard pattern in well-tested systems code and would make a real difference in test coverage.

Output format. Both Caracat and Caracal use CSV. The supervisor raised NDJSON (Newline-Delimited JSON) as a possible improvement. NDJSON is self-describing — each line carries field names, so you do not need to know the column order in advance. It also streams naturally, one line at a time, which matters when you are producing millions

of measurement results. Adding new fields later would not break existing parsers either, unlike with CSV.

Documentation. The docs on docs.rs are incomplete and there is no architectural overview anywhere. Fixing the inline documentation and writing a proper README that explains the overall design would make a big difference for anyone trying to contribute or just understand how Caracat works.

2.6 Deliverables

Over the course of this project, we produced the following for the Caracat side:

- A detailed comparative analysis of Caracat and Caracal (around 2,500 words), covering the architecture, core mechanisms, testing results, and a file-by-file mapping between the two codebases.
- A condensed version of the same analysis for quick reference.
- An output format breakdown showing exactly what each tool produces, field by field.
- A technical deep-dive covering six areas: protocol extensibility, Paris Traceroute implementation, probe self-identification, filtering, timestamp precision, and payload handling.
- Six Excalidraw diagrams: system architecture, probe sending flow, reply receiving flow, module dependencies, Caracat-vs-Caracal comparison, and key mechanisms.
- Program flow diagrams by Seyeong showing the packet processing pipeline for sending and receiving.
- A five-experiment test suite with documented results.
- A presentation to the Caracal development team with a Q&A session on production requirements.

The detailed documents are included in the annexes and referenced throughout this report.