Rust for Microservices

Background and Timeline

- Built Rust Proof of Concept for a service we were going to need.
- I became aware of the need ~mid March 2018 and began coding.
- Helped interview consultants to see if we wanted to outsource the work.
- Code complete date was supposed to be June 1st 2018.
- Times almost up end of April 2018.
- I was the only one in the code base until the beginning of May.
- POC is near code complete get go ahead to productize.
- Cleaned it up and out to production June 1st 2018.
- Delivered on June 1st.
- Heavy iteration by team over the summer with production users live in September.

The Team

- Go
- Scala
- Javascript
- PHP
- 1 new engineer with Rust experience started mid May.

Team Comments and Experience

- Lack of good IDE support slows you down while learning. (Compared to Scala)
- Everyone committed code fairly quickly.
- futures-await macro made debugging hard.

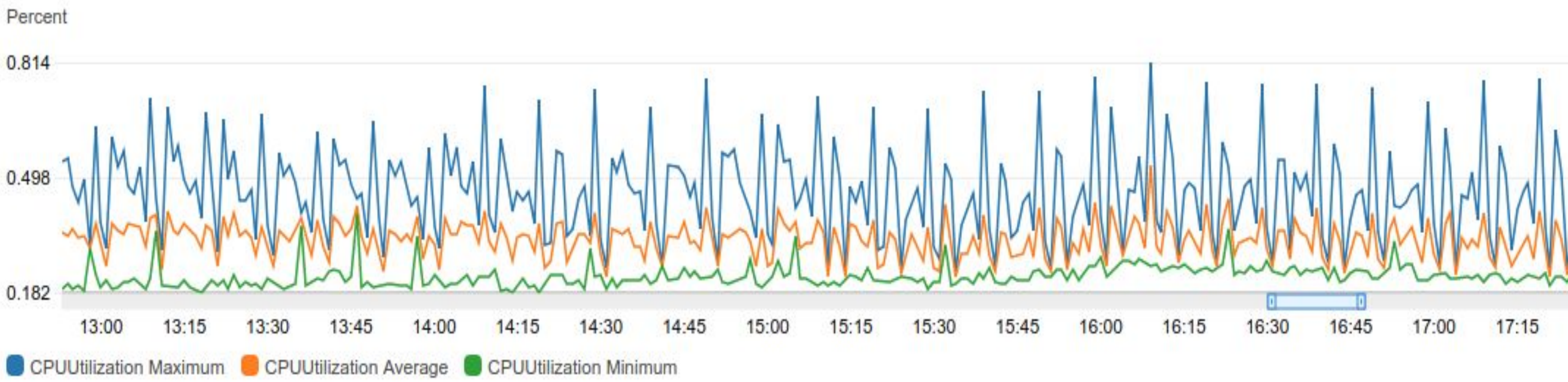
Async/Await

- Team was used to future combinators coming from Scala.
- Didn't want to keep putting them through that if at all possible.
- Chose to use the nightly compiler and futures-await
- The futures-await macros were great but would make compilation fail and hide the reason. (NOT GOOD)
- Recently moved to the built in async macro and the tokio compatibility layer. It has been great.

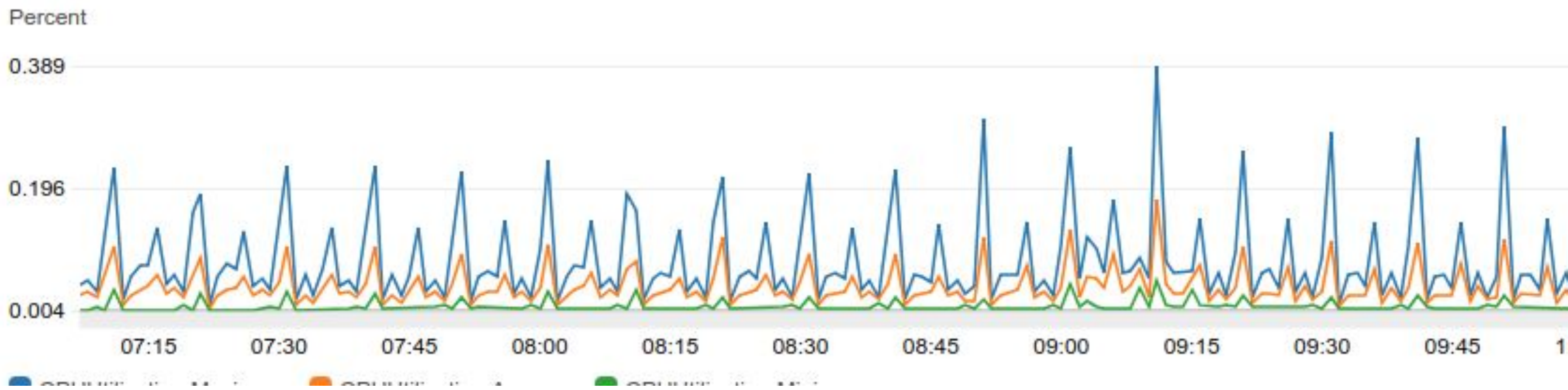
Actix Web

- Good choice for the team coming from Play 2 which is built on Akka.
- Most ergonomic async framework I have found.
- Supported fancy stuff like websockets (don't need them but just in case).
- Has been very stable even though it is very young.
- Only issue thus far was around trust-dns which had a bug that would cause a crash once every 24 hours or so. Maintainers of both libraries were very supportive and the problem has been fixed.
- Performance is way, way more than enough for our needs.

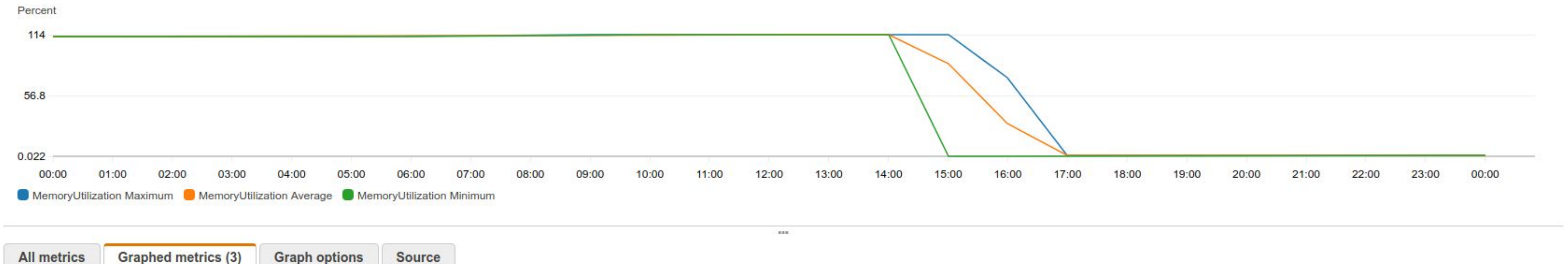
CPU Performance



Rust CPU Performance



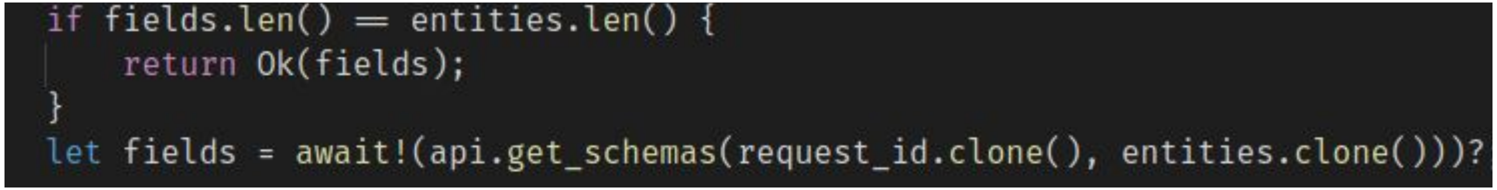
Memory Use (graph shows % of 256 MB)



- Scala is at ~114% or ~292MB
- Rust ~1.17% ~3MB

Why Rust?

- Error Handling
  - try! eg ? make it easy to decide once how certain types of errors are handled and forget about it.



```
if fields.len() == entities.len() {
    return Ok(fields);
}
let fields = await!(api.get_schemas(request_id.clone(), entities.clone()))?
```

- If it compiles it runs.
- Explicit and correct by design.
- Despite possibly noisy syntax logic is clear and obvious.
- Resource usage is very predictable.

Errors to date

- Last I checked we have had 6 legitimate bugs.
- Of those bugs all were related to errors around mapping data from the wrong fields or not pulling a field we needed from the underlying system.
- This service has never woken me up.
- None of the bugs I mentioned were reported by the customer.

- Last service rewrite was done in the span of 3 days when I had down time. Semantics between Scala and Rust were so close my tests caught a potential issue in both code bases. (Easy rewrite is a credit to how clean the Scala code base was.)

My Complaints

- I wish I could use stable and have async/await today. (I’m impatient)
- Compile times make me look bad.

What's Next?

- More Rust services. (If my team doesn't run me away with torches and pitchforks)
- Yesterday and today I built a frontend for the latest service I threw together for Rust and I built it in Rust with WASM. Not too painful using seed (seed-rs.org). Also not production ready. Maybe one day. Data structure reuse was nice.