Task 2: A lot of these are very subjective and opinionated. To be clear, these aren't reasons that I'd avoid the library or anything, but more so these are decisions I would've done differently.

- Overuse of builders. Can give impression that e.g. multiple serializers can be used. I know it's the Ktor style, but it does feel over-the-top. Fix is really simple, just use data classes. If you really want to continue using builders, there's also ways to make them safer (so that fields must be initialized). IIRC, there's a compiler plugin that adds checkers for that, or there's a trick that [I discovered recently](https://github.com/arrow-kt/arrow/pull/3837) using contracts.

- There's a lot of legacy support already for such a young library that's pre 1.0. I think it may be worth it to rip the bandaid and say "sorry, no backwards compatibility" to help simplify a lot of the code. Maybe deprecate for only one cycle? Users of a pre 1.0 library are usually expected to update their versions often and such. I fear the added complexity from backwards compatibility might hinder developing the protocol

- It might be worth it to focus only on Ktor users and their needs. Currently, it hasn't made the API complex, but I fear that it might in the future.

- Pick up serialization from ContentNegotiation, instead of having the user configure it (might be hard because ContentNegotiation is rather HTTP specific, with content types and all). Not sure exactly how to write this to be honest. It avoids some amount of duplication on the user's side.

- Sending exception stack traces seems like a really bad idea (security/privacy issues, maybe even potential for DoS attacks?). Maybe remove them by default? Coroutines has a CoroutineExceptionHandler that usually logs the stack traces, so using that seems reasonable. (#213)

- A lot of internal APIs are exposed due to Kotlin module limitations. There's no easy fix for this yet, but [this youtrack issue](https://youtrack.jetbrains.com/issue/KT-62688) suggests that friend modules will be properly supported in the future. It's not really a big deal since they're marked with an opt-in, but it's nicer if they never appear to the user.

- This isn't really a problem, but Arrow integrations could be nice! Specifically the `Raise` api might come in handy. The easiest way to support that is to use `Either` and `Either.bind` to marshall the data back and forth, but there might be a more clever implementation with Exceptions and handling `RaiseCancellationException` in a custom way.

- Very minor: KrpcCallMessage-related code has some duplication. I think instead KrpcCallMessage should have its data be `String` or `Binary`, but all the other fields stay the same (i.e. KrpcCallMessage should be a data class, while its data is some sealed hierarchy).

- I wonder why a custom cancellation implementation is necessary? It feels like the KrpcTransport should be the one in control of cancellation, so there's no need to include it in the protocol.
- There's a lot of OOP code (mostly abstract classes) going on, and I wonder how much of it is needed. If it's the apt abstraction, then it's absolutely fine! I just worry that it may become a maintenance burden when compared to a more composition-based approach

- Passing rpc services to other services should be supported somehow. The easiest way would be to pass the unique service id along (as long as we're sure that we're communicating to the server that owns that instance). If the service isn't local to the server we send it to, though, then it's tougher. Maybe we should set up a whole server? This is a generalisation from the bidirectional flow streaming that the library already has. I was surprised to see there's a github issue for this already: #341. This issue also has some interesting ideas, but I don't think it's the right approach: #403.

- The future is context parameters, and I think some types from the library can benefit from having contextual bridge methods (namely RpcClient and RpcServer). Another related issue is supporting them for services, but that's trivial to fix (#507)

- #485 reports that FqNames are passed over the transport. There's some amount of inefficiency with that, and it can likely be easily optimized. As to how to optimize it, maybe when doing code-gen, also make an array that has all the rpc service types, and simply pass around indices into that array, instead of fqnames. I'm not sure how to get that working with services defined in libraries, however, so maybe this isn't the best of ideas. Maybe the protocol can, whenever a new fqname is used, also send along an id number that it wants associated with that fqname, and then, if the other side agrees, the id gets used for future communication.

- Authentication seeems to be commonly requested, but I think great care should be taken here. Some people are asking for having different auth levels in the same service, but I think that's fundamentally wrong (IMHO), since it's confusing on the service consumer side (no type reason as to why this method is fine but this one needs auth). I think the apt way to do it is authentication per service or even per `RpcClient`. The only issue that remains, then, is expiry/reauthentication, which I have no idea on how to solve! Similarly, reconnection is brought up a lot (too many issues/slack discussions to link, but I'm sure you've seen them all), and I have no authority to speak on that, either.

- It would be nice if, once stable/post 1.0, the compiler plugin part is moved into the kotlin repo, just like what happened with serialization and compose. I fear that having to manage compiler versions and match them up to kotlinx.rpc versions might put off some users. Of course, this problem goes away once there's a stable compiler plugin API, but that seems far enough away to me.