Originally set to be a talk at Devconnect, I'm now tossing my random thoughts onto this blog, thanks to being unable to attend. We stumbled upon a pretty neat question during a chat in Prysm's internal channels: “What would you have done differently if you were to start Prysm from scratch?” This question caught my attention, opening up a whole world of thoughts. In this post, I'm going to share my reflections on how I'd go about creating an Ethereum consensus layer client if I could turn back time to early 2019, with a metaphorical crystal ball in hand . I’ll dive into that and my overall take on Ethereum consensus layer client develop

Ethereum thrives on decentralization, and client diversity is a crucial part of that DNA. The key differences between client implementations boil down to the programming language and the team behind the code. There's also variety in the tech stacks and design decisions, making client development an incredibly diverse field. But this leads to a question: How many different client implementations do we really need? While we might not require as many as 100, having 5 robust consensus layer clients currently leading the charge shows the strength of our ecosystem.

Now, the exciting part is considering what other specialized clients might benefit. Imagine a client-focused more on research and simulation. For instance, if we wanted to add a new field to the beacon block or tweak the state transition function, how quickly could we set up a local devnet? How accessible would it be for someone unfamiliar with the client codebase to implement such changes? A research-oriented client might be precisely what we need in today's landscape to help push forward Ethereum.

One GitHub repository that truly deserves more recognition is the

Ethereum Consensus Spec. It's a compact yet powerful collection of about 17,000 lines of Python and Markdown, outlining everything from the consensus state transition function and fork choice rule to P2P network guidelines, honest validator behavior, and hashing algorithms. This isn't just code; it's the backbone of Ethereum's consensus, securing a

$58 billion in staked ether, not to mention the

immense value tied up in DeFi, NFTs, and Layer 2 solutions relying on Ethereum for security.

What's really cool is that all of this is open source. There’s no hidden, proprietary magic here. The beauty of the spec lies in its minimalism and a certain degree of intentional ambiguity. It sets the rules, but how clients meet these rules involves a range of design decisions. It's a feature, not a bug, Why? Because it creates herd immunity. The diversity in implementation enhances the network's resilience against systemic attacks and failures. The more varied the implementations, the tougher it becomes to break the system.

While the spec is designed for readability, prioritizing clear understanding, client code is all about optimizing performance and ensuring safety. But don't get me wrong, creating specs is no walk in the park. A bug in the spec can be far more damaging because it might spread across multiple client implementations, potentially leading to a consensus split and finalizing the wrong outcome. That said, changes to the spec are usually easier to manage compared to client code. The role of a client implementer extends beyond just following the spec. They have to deal with various designs, including syncing, caching, persistent storage, instrumentation, and providing a smooth experience for stakes and node operators. So, while the spec provides the framework, the client implementers are the ones who bring it to life.

If I could rewrite a consensus client from the ground up, my first guiding principle would be minimalism. I’d focus on being conservative with new features, preferring to leverage proven solutions. The goal would be to build only what's essential, minimizing maintenance costs and steering clear of 'nice-to-have' features. This approach centers on the core functionality, remembering that every line of code matters and has long-term implications. In client development, there are typically two philosophies. The first prioritizes correctness and rock-solid stability, aiming for absolute compliance and reliability. The second leans towards experimentation, pushing the limits of performance, speed, and high throughput, even if it means occasionally deviating from the protocol occasionally. The question here is: Can you successfully marry these two approaches? Is it more feasible to start focusing on performance and then shift towards security, or the other way around? Adopting a minimalist approach gives you the flexibility to navigate both paths without committing too early to either one.

The second principle I'd embrace is designing with failure modes in mind. It's important to accept that failures will happen, and in many cases, they're part of the process. Distinguishing between hard and soft failures is crucial. From a programming standpoint, this involves deciding when to log an error versus when to return an error and terminate a process. Take the block proposal as a concrete example. In Ethereum, liveness slightly favors safety. So, as a block proposer, if you encounter issues while packing attestations or packing transaction payload, should you abandon the block proposal? The spec provides some guidance: a block without attestations can still pass the state transition function, but one without transaction payloads likely won't because of withdrawals. When I first started writing code in 2019, I didn’t fully grasp this. Now, with hindsight, I see the immense value in maximizing liveness whenever possible.

The third principle, though perhaps not surprising, is crucial: Be conservative about caching. As the saying goes in computer science, 'There are only two hard things: cache invalidation and naming things.' Caches might be easy to implement, but they're notoriously difficult to debug and even harder to remove later. Common pitfalls include deadlocks, off-by-one errors, and memory bloat – all issues we'd rather not see in a production-level client. Reflecting on the necessities, how many caches do we truly need for a production-level consensus layer client? Surprisingly few, I'd argue. Take the beacon state, for example, which is one of the more resource-intensive. The only relevant states are those needed for processing blocks, which narrows down the caching requirements to states advanced to the next slot, epoch boundaries, and for replay/syncing purposes. A client could efficiently operate with just 2-3 types of caches, each with a limited number of entries. Some clients are even experimenting with innovative caching strategies, like using a base state with a forward diff. So, before adding a cache to your code, it’s crucial to scrutinize each decision carefully

In the next sections, I'll touch on some minor principles that are widely known, so I’ll keep it brief. First, choose data structures that are friendly to garbage collection, minimize heap allocations, and aim to reuse objects whenever possible. Contiguous memory data structures are highly recommended, provided they're well-supported in your programming language. Leverage your language's library ecosystem and prioritize type safety, as languages with strong type systems help catch errors during compilation. Memory management is another area with interesting choices – controlling memory or manually relying on garbage collection. Reflecting on

Prysm's journey since 2019, one significant development in Golang is the introduction of generics. Generics enable more flexible and reusable code without compromising Go's simplicity, but they come with potential drawbacks like runtime overhead and increased memory usage. It's crucial to profile and benchmark your code before usage thoroughly.

Testing strategies for a production-grade client deserve more attention than they typically get. Internally, we rely on unit and end-to-end tests. Externally, there are several types of tests for specific purposes: consensus spec tests ensure compliance, hive tests check for interoperability, and kurtosis tests assess resilience and stress tolerance. Reflecting on my experience, more emphasis should be placed on end-to-end tests. It's more beneficial to test the entire system holistically than to cover specific lines of code. Another beautiful yet underrated aspect is the shareability of integration tests across different client implementations. However, a significant gap in current testing strategies is API testing. Many of today's issues are related to the beacon-API, making the development of a robust test framework increasingly vital. Key areas to focus on are cross-client compliance and performance. These tests don't need to run with every pull request; they can be more effectively utilized as part of a separate suite for pre-release regression detection.

When developing a client, it's easy to lose sight of our actual customers - the individuals and groups for whom we're building these tools. Our typical audience extends beyond the general blockchain community to more specific groups like professional stakers, Layer 2 solutions, and DeFi applications. Understanding the specific needs of these users allows us to explore and balance interesting trade-offs. Earlier, I discussed the necessity for a client tailored to researchers and prototypers, which naturally involves its own set of trade-offs. For such a client, the emphasis might shift from raw performance to factors like expressibility and robust data collection capabilities. The current challenge lies in the sheer complexity of required changes, often spanning multiple APIs and layers, including consensus and execution. This complexity means that, language proficiency aside, only a few domain experts are truly equipped to navigate these changes effectively. For simulation purposes, a research-oriented client could serve as a vital bridge between researchers and implementors.

Wrapping up, some final thoughts: Working with open source is not just rewarding; it's exhilarating. There's something incredibly fulfilling about contributing to a project that has a tangible impact on the real world. Yes, it demands a significant amount of dedication and effort, but the learning curve is steep and incredibly satisfying. For me, there's no place I'd rather be. However, amidst all the hard work and breakthroughs, it's essential to take a moment to breathe and enjoy the journey. After all, the joy of the journey is as important as the destination.