There's a semantic issue with how transactions are processed which needed a bit of work, so we discussed it at some length.

A transaction is a piece of code, which executes (and may do reads), producing a result; then, this result is somehow processed into an actual state change. For example, it might produce something like the pseudocode value %{cms: [cm1, cm2], nfs: [nf1, nf2]}

, which is processed by the backend into "append cm1 and cm2 to the commitment set, nf1 and nf2 to the nullifier set".

The design was broadly conflating the production of the result with the state change the result is processed into; they're directly linked so this isn't a wrong semantic, but it was a confusing semantic slowing down our work on the system. In particular, a transaction's reads

are done after it's produced its VM result, this code being where reads are done, and its writes

are done after it's produced its "write result". After some discussion, we've agreed to split this up a bit.

A transaction that does no reads instantly (morally instantly) produces a VM result, blocking no one else's reads, but it's not "complete" until it's had its side effect. So this has been split up into a "result event" and a "completion event", the latter happening as the side effect is completed and the transaction process reaps itself. For example, the tx "doing no reads, unconditionally write 0xabcd to blob storage" instantly has a VM result, but it is still ordered, and things ordered after it can't write

until its completion event. After all, they might read hash-of-0xabcd in blob storage.

Transaction lifecycle notes:

- User casts a tx to mempool
- Mempool adds it to the state, sends it to Executor
- Executor starts the Worker with backend logic
- Worker may send result messages through the Broker
- Mempool receives this as it is unblocked. Checks that these are actual ids it has. Adds results to its state
- Consensus casts an ordered list of txs
- Mempool checks that all ids are in, casts execution to the Executor
- Executor call Ordering to order, waits to receive all completion events. If it receives all, sends an ExecutionEvent to the Broker
- Mempool is unblocked, so it receives results in the meantime
- Mempool is subscribed to a filter agent that sends both results and execution events so there arrive in order
- Hence Mempool gets an execution message after all results get written.
- Then add completion results and sub-results together, send Storage a message to commit