Document Nodes have primary inputs and secondary inputs
Proto Nodes have only a single primary input, and all secondary inputs are fields of the struct upon which the node implements

Currently caching uses CacheNodes that are manually included in the definition of certain specific nodes, like CreateCanvas that lets us reuse (cache) the canvas DOM element. In the future, during a graph rewriting step (specifically where is TBD), we can make it insert more CacheNodes.

Levels of abstraction

Explicit composition (input resolution step):
Happens at execution time, serves as the actual sequence of Rust functions that get called.
The Rust call stack gets built from the network output, traversing backwards up the graph flow, building a footprint.
Here, the compose nodes account for most of the nodes being called, as they delegate to the lambdas that form their wrapped operation.
The compiler automatically inserts the ComposeNodes (where no manual composition is enabled).

Type inference -> Type checking and mapping from stable node id to type

Proto graph with stable node IDs:

Topologically sorted proto graph:

“Input resolution” Proto graph with explicit composition

https://youtu.be/Q5yibBN3U1k?t=3341

- **A single eval()-able `Box<dyn Node>` (a struct that implements the Node trait)**
  This node can be called with .eval() with editor_api as its input, and it wraps the entire program internally.

  ↑
  Linking step: based on the generics, we look up the implementation from the node registry that fits the type signature for each node. If the node registry doesn’t contain it (the T -> T identity node might not have something arbitrary like (f32, f32, f32) -> (f32, f32, f32) for example), then we need to fetch it from the compilation server. Also we instantiate the node implementations with their arguments. We have the borrow tree in this step, which holds all the allocated and constructed nodes (from looking it up in the node registry and then storing it in the borrow tree), because we need to reference it where it can be stored. We need to link different nodes into a final complex form. We

need to allocate the nodes that live somewhere, this lets us store them persistently as the borrow tree is the owner of the data.
↑

- **Typed “simple” proto graph**
  While we haven’t modified the “simple” proto graph from the previous step, we now also have a separate data structure called TypingContext that can be used in conjunction with the proto graph to look up the type information.

  ↑
  Type inference (we start from the leaves of the graph, and look up what the type for each of the leaves is, and traverse upwards and always determine the type of the next node based on its inputs— does type checking and substitutes type arguments, where they exist, with concrete types. The result is that we’ve assigned concrete types to all nodes.)
  ↑

- **“Simple” proto graph without primary inputs (compose nodes substituted for primary inputs)**
  Still a valid proto graph, but now an even simpler grammar could be used to describe this form of proto graph.

  ↑
  Input resolution (the inputs are turned into compose nodes, this is where the compose node insertion happens)
  ↑

- **“Complex” proto graph with primary inputs:**
  Consists only of proto nodes, but they have primary inputs (represented as `CallArg::Node(NodeId)`) and no compose nodes yet.

  ↑
  Document graph -> proto graph conversion
  + Split document network with multiple outputs into one proto network per output
  ↑

- **Flattened document graph**
  We substitute nested document node implementations with their contents, so all nodes are document nodes that have an implementation that is a proto node.

  ↑
  Document graph flattening
  ↑

- **Nested document graph without Extract node implementations**

We substitute Extract node placeholder implementations with a ValueNode that produces a DocumentNode (graph source code representation usable at runtime/render-time). Also this step would do Inject in the future.

↑
Extract node resolution
↑

- **Nested document graph**
  This is what the user sees in the UI