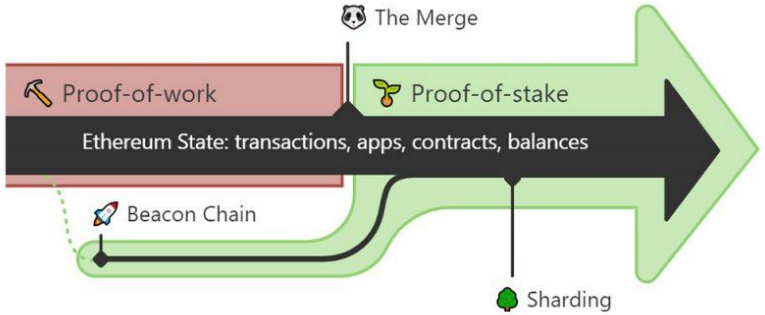
Debunking Ethereum

Daniel Feller, Venkatesh Kamat, Becky Blake, Xinhao Zhao, Kathy Lo

Consensus & Upgrades

From PoW to PoS – How The Merge Was Implemented

- The Merge joined two chains:
  - Execution Layer (eth1 / mainnet)
  - Consensus Layer (Beacon Chain / eth2)
- Activated at Terminal Total Difficulty (TTD) instead of block height
- Geth transitioned to PoS via MergeForkChoiceUpdate logic in consensus/engine/
- Mining APIs deprecated (e.g., eth_getWork, eth_submitWork)
- Validators became block proposers via Beacon Chain

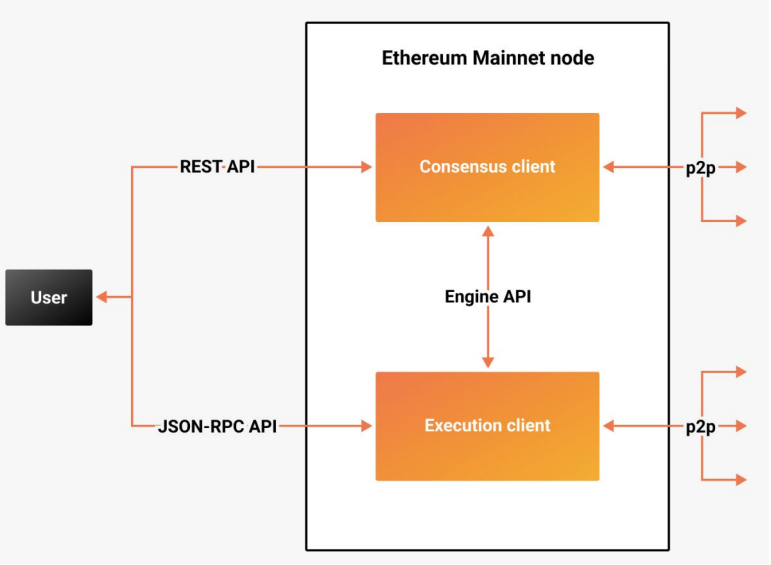


Execution Layer

After The Merge, the execution layer is responsible for:

- Managing Ethereum's state
- Processing transactions
- Executing smart contracts using the EVM

Implemented by clients like Geth(most widely used), Nethermind, and Besu



Geth Code Overview

🚀 Startup Flow – `cmd/geth` (https://github.com/ethereum/go-ethereum/tree/master)

1. **Entry Point: main()**
Located in cmd/geth/main.go, it initializes the CLI app using the urfave/cli package.
It sets up CLI flags and commands like geth, attach, console, etc.

2. **Command Execution: geth command**
The geth command calls geth.Run, which triggers the initialization logic for the Ethereum node.
Internally, it calls makeFullNode(ctx), defined in cmd/geth/config.go.

3. **Node Creation: makeFullNode()**
Creates a full Ethereum node by calling node.New().

Ethereum Node Initialization in Geth

makeFullNode() Flow

- makeFullNode(ctx) is defined in cmd/geth/config.go
- Calls makeConfigNode() to load configurations:
  - Chain ID, DB, network settings
  - Returns a base node and config object
- Registers core services
  - Ethereum backend, P2P, RPC

```
func makeFullNode(ctx *cli.Context) *node.Node {
  stack, _ := makeConfigNode(ctx)
  // services like Ethereum backend are registered here
  return stack
}
```

```
// makeFullNode loads geth configuration and creates the Ethereum backend.
func makeFullNode(ctx *cli.Context) *node.Node {
	stack, cfg := makeConfigNode(ctx)
	if ctx.IsSet(utils.OverridePrague.Name) {
		v := ctx.Uint64(utils.OverridePrague.Name)
		cfg.Eth.OverridePrague = &v
	}
	if ctx.IsSet(utils.OverrideVerkle.Name) {
		v := ctx.Uint64(utils.OverrideVerkle.Name)
		cfg.Eth.OverrideVerkle = &v
	}

	// Start metrics export if enabled
	utils.SetupMetrics(&cfg.Metrics)

	backend, eth := utils.RegisterEthService(stack, &cfg.Eth)

	// Create gauge with geth system and build information
	if eth != nil { // The 'eth' backend may be nil in light mode
		var protos []string
		for _, p := range eth.Protocols() {
			protos = append(protos, fmt.Sprintf("%v/%d", p.Name, p.Version))
		}
		metrics.NewRegisteredGaugeInfo("geth/info", nil).Update(metrics.GaugeInfo
			"arch":      runtime.GOARCH,
			"os":        runtime.GOOS,
			"version":   cfg.Node.Version,
			"protocols": strings.Join(protos, ","),
		})
	}

	// Configure log filter RPC API.
	filterSystem := utils.RegisterFilterAPI(stack, backend, &cfg.Eth)

	// Configure GraphQL if requested.
	if ctx.IsSet(utils.GraphQLEnabledFlag.Name) {
		utils.RegisterGraphQLService(stack, backend, filterSystem, &cfg.Node)
	}
	// Add the Ethereum Stats daemon if requested.
	if cfg.Ethstats.URL != "" {
```

Service Registration

- Defined in `eth/backend.go`
- After makeConfigNode, Geth registers services using stack.Register()
- Registers eth.New() which bootstraps:
    - Blockchain
    - State DB
    - Transaction pool
    - Miner (or PoS proposer)

```
stack.Register(func(ctx *node.ServiceContext) (node.Service, error) {
   return eth.New(ctx, &cfg)
})
```

Geth Code Overview

🧱 Module Map – Core Directories

1. eth/

- Purpose: Implements the Ethereum protocol and consensus engine.
- Initializes blockchain, state database, transaction pool, etc.
- Registers the Ethereum service to the p2p network.

- Core files:
  backend.go: Ethereum service backend.
  handler.go: Protocol handler (ETH protocol).
  miner/: Mining logic.
  tracers/: EVM tracing.

```
type Ethereum struct {
    // core protocol objects
    config         *ethconfig.Config
    txPool         *txpool.TxPool
    localTxTracker *locals.TxTracker
    blockchain     *core.BlockChain

    handler *handler
    discmix *enode.FairMix
    dropper *dropper

    // DB interfaces
    chainDb ethdb.Database // Block chain database

    eventMux       *event.TypeMux
    engine         consensus.Engine
    accountManager *accounts.Manager

    filterMaps      *filtermaps.FilterMaps
    closeFilterMaps chan chan struct{}

    APIBackend *EthAPIBackend

    miner    *miner.Miner
    gasPrice *big.Int

    networkID     uint64
    netRPCService *ethapi.NetAPI

    p2pServer *p2p.Server

    lock sync.RWMutex // Protects the variadic fields (e.g. gas price and etherbase)

    shutdownTracker *shutdowncheck.ShutdownTracker // Tracks if and when the node has shutdown ungracefully
}
```

```
func New(stack *node.Node, config *ethconfig.Config) (*Ethereum, error) {
    // Ensure configuration values are compatible and sane
    if !config.SyncMode.IsValid() {
        return nil, fmt.Errorf("invalid sync mode %d", config.SyncMode)
    }
```

```
chainDb, err := stack.OpenDatabaseWithFreezer("chaindata", config.DatabaseCache, config.DatabaseHandles, config.DatabaseFreezer,
"eth/db/chaindata/", false)
if err != nil {
    return nil, err
}
```

```
chainConfig, _, err := core.LoadChainConfig(chainDb, config.Genesis)
if err != nil {
    return nil, err
}
```

Referenced from eth/backend.go

Geth Code Overview

- eth/backend.go

```
    engine, err := ethconfig.CreateConsensusEngine(chainConfig, chainDb)
    if err != nil {
        return nil, err
    }
    eth.txPool, err = txpool.New(config.TxPool.PriceLimit, eth.blockchain, []txpool.SubPool{legacyPool, blobPool})
    if err != nil {
        return nil, err
    }
eth.APIBackend = &EthAPIBackend{stack.Config().ExtRPCEnabled(), stack.Config().AllowUnprotectedTxs, eth, nil}
if eth.APIBackend.allowUnprotectedTxs {
    log.Info("Unprotected transactions allowed")
}
```

```
if eth.handler, err = newHandler(&handlerConfig{
    NodeID:         eth.p2pServer.Self().ID(),
    Database:       chainDb,
    Chain:          eth.blockchain,
    TxPool:         eth.txPool,
    Network:        networkID,
    Sync:           config.SyncMode,
    BloomCache:     uint64(cacheLimit),
    EventMux:       eth.eventMux,
    RequiredBlocks: config.RequiredBlocks,
}); err != nil {
    return nil, err
}
```

```
// Register the backend on the node
stack.RegisterAPIs(eth.APIs())
stack.RegisterProtocols(eth.Protocols())
stack.RegisterLifecycle(eth)
```

Referenced from eth/backend.go

Geth Code Overview

🧱 Module Map – Core Directories

2. p2p/

- Purpose: Peer-to-peer networking stack.
- Manages peer discovery (via devp2p), connections, and communication.
- Handles protocols like ETH, LES, and custom subprotocols.

- Core files:
  server.go: The main p2p server.
  peer.go: Peer connection and messaging.
  disc/: Node discovery (Kademlia DHT).

3. core/

- Purpose: Core Ethereum data structures and state transition logic.
- Defines the blockchain, block processing, EVM execution, and transaction handling.

- Core Files:

  /blockchain.go: Main blockchain structure.
  state/: The world state (Merkle Patricia Trie).
  vm/: EVM implementation.
  types/: Block, header, transaction types.

Contract Creation & Execution core/vm/evm.go

Contract Creation

create():
Creates a contract at a non-deterministic address.
Address = hash(sender address + nonce)

```
func (evm *EVM) Create(caller common.Address, code []byte, gas uint64, value *uint256.Int) (ret []byte, contractAddr co
    contractAddr = crypto.CreateAddress(caller, evm.StateDB.GetNonce(caller))
    return evm.create(caller, code, gas, value, contractAddr, CREATE)
}
```

create2():
Creates contract with deterministic address.
Address = hash(sender + salt + codeHash)
Introduced in EIP-1014 for off-chain contract address prediction.

```
func (evm *EVM) Create2(caller common.Address, code []byte, gas uint64, endowment *uint256.Int, salt *uint256.Int) (ret
    contractAddr = crypto.CreateAddress2(caller, salt.Bytes32(), crypto.Keccak256(code))
    return evm.create(caller, code, gas, endowment, contractAddr, CREATE2)
}
```

Execution

call()

- Executes contract code at a target address.
- Transfers ETH and passes input data.
- Invokes Run() in the EVM interpreter to simulate contract logic.

```
func (evm *EVM) Call(caller common.Address, addr common.Address, input []byte, gas uint64,
    value *uint256.Int) (ret []byte, leftOverGas uint64, err error) {
```

```
p, isPrecompile := evm.precompile(addr)
```

```
if isPrecompile {
    ret, gas, err = RunPrecompiledContract(p, input, gas, evm.Config.Tracer)
} else {
    // Initialise a new contract and set the code that is to be used by the EVM.
    code := evm.resolveCode(addr)
    if len(code) == 0 {
        ret, err = nil, nil // gas is unchanged
    } else {
        // The contract is a scoped environment for this execution context only.
        contract := NewContract(caller, addr, value, gas, evm.jumpDests)
        contract.IsSystemCall = isSystemCall(caller)
        contract.SetCallCode(evm.resolveCodeHash(addr), code)
        ret, err = evm.interpreter.Run(contract, input, readOnly: false)
        gas = contract.Gas
    }
}
```

State Management

Transactions submissions initially start in the /internal directory.

- **internal/ethapi/api.go** - this is the file that initially accepts and decodes the transaction request.

Transactions primarily execute through the core directory.

- **core/blockchain.go** - manages the blocks and coordination of the state updates.
- **core/state_processor.go** - this applies the transaction to the state, manages the gas consumption, and updates the related balances and accounts.
- **core/vm/evm.go & core/vm/interpreter.go** - these files execute created smart contracts in an EVM environment.
- **core/state/statedb.go** - manages the state updates resulting from any transactions.

State changes are stored in Ethereum's trie data structure.

- **trie/trie.go & trie/database.go** - it maintains and retrieves state data using a Merkle Patricia Trie.

State Management - State Transition

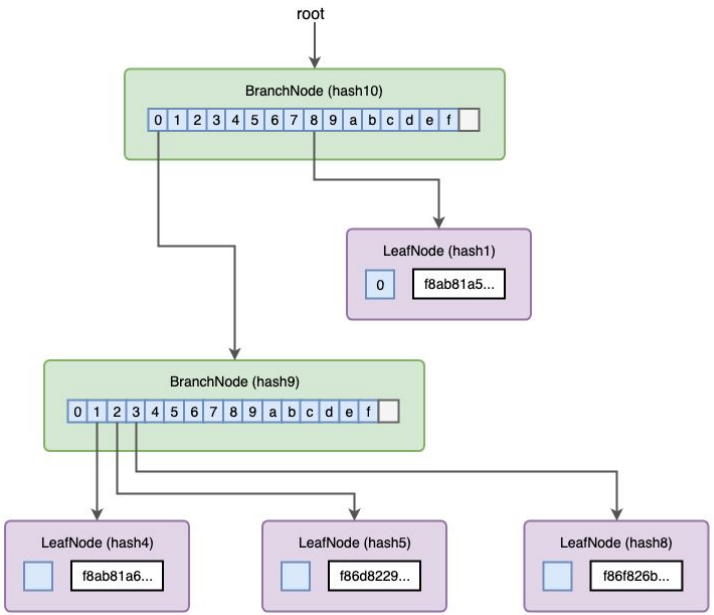
- Verifies funds
- Calculate gas consumption
- Move Eth/ run smart contract
- Update changes

```go
// Processes a transaction and updates state
func (st *StateTransition) ProcessTransaction() error {
    // 1. Check sender's balance and nonce
    if err := st.checkSender(); err != nil {
        return err
    }

    // 2. Deduct gas payment
    if err := st.payGas(); err != nil {
        return err
    }

    // 3. Execute transaction
    if st.isContractCreation() {
        // Create new contract
        return st.createContract()
    } else {
        // Execute contract call
        return st.executeCall()
    }
}
```

State Management - Database Operation




```go
// Manages the entire Ethereum state
type StateDB struct {
    accounts map[common.Address]*stateObject  // Account cache
    storage  map[common.Hash]common.Hash      // Storage cache
    logs     []*types.Log                     // Transaction logs
}

// Get account state
func (db *StateDB) GetAccount(addr common.Address) *stateObject {
    // 1. Check cache
    if account := db.accounts[addr]; account != nil {
        return account
    }

    // 2. Load from database
    account := db.loadAccount(addr)

    // 3. Cache and return
    db.accounts[addr] = account
    return account
}
```

State Management - Storage Management

- Key-value pairs
- Multiple layers of caching
- Ensure consistency

```go
// Manages contract storage
func (account *stateObject) GetStorage(key common.Hash) common.Hash {
    // 1. Check cache
    if value, exists := account.storage[key]; exists {
        return value
    }

    // 2. Load from database
    value := account.loadFromDatabase(key)

    // 3. Cache and return
    account.storage[key] = value
    return value
}
```

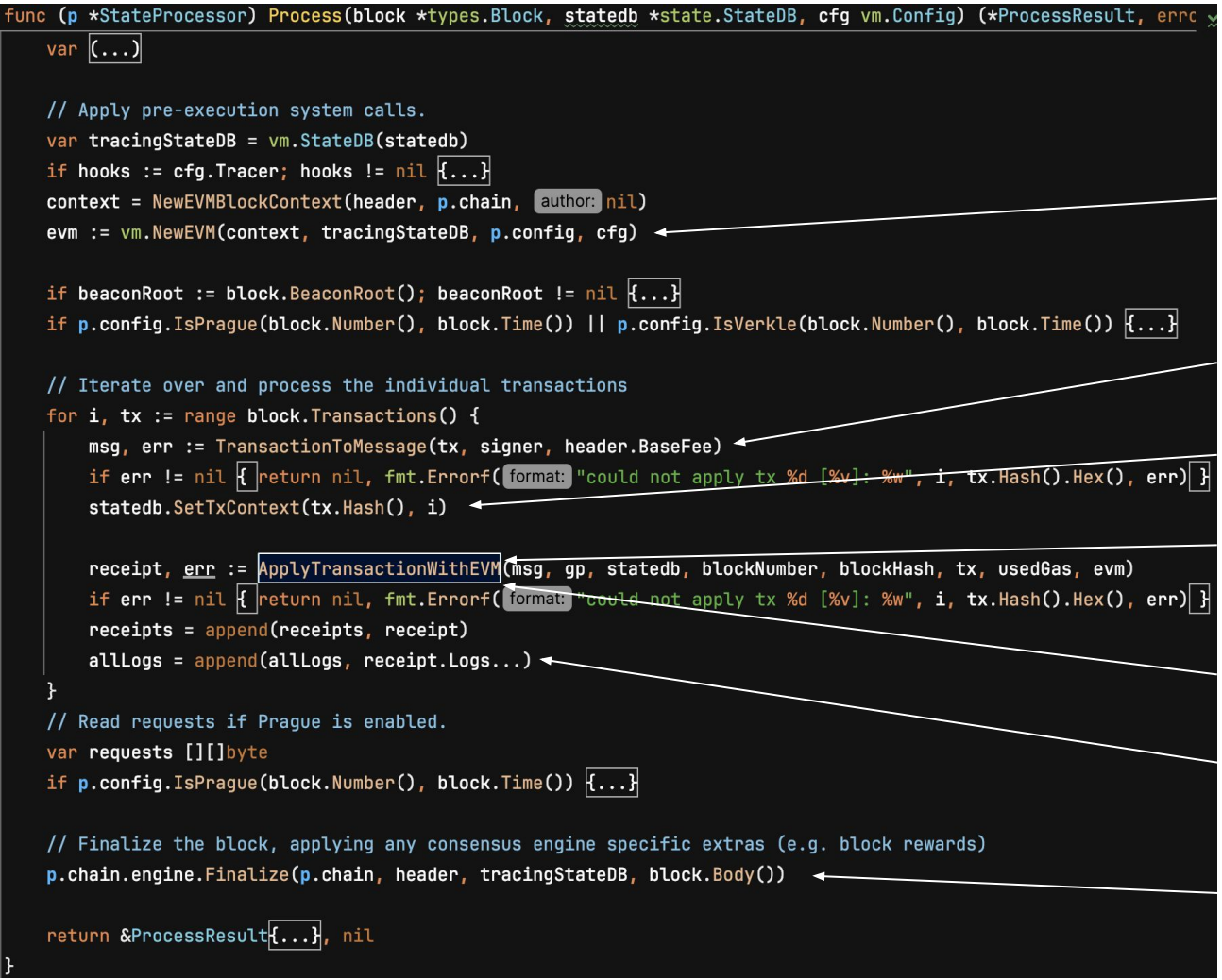
```go
// Update storage
func (account *stateObject) SetStorage(key, value common.Hash) {
    // 1. Mark as modified
    account.dirty = true

    // 2. Update storage
    account.storage[key] = value

    // 3. Record change
    account.db.recordChange(account.address, key, value)
}
```

Transaction Flow

1. User signs a transaction using their private key (e.g., via MetaMask)
2. Transaction sent to an Ethereum node via JSON-RPC(`eth_sendTransaction` or `eth_sendRawTransaction`)
3. The execution client (e.g., Geth) receives the transaction(mempool) and:
    - Verifies the signature, nonce, and balance for gas (in core/tx_pool/ and core/state/)
    - If valid - > add to the mempool as pending
4. Node broadcast the transaction
    - Other nodes receive the transaction and add it to their mempools
5. Validator Proposer Selection(PoS)
    - Beacon chain(consensus client) selects a validator at each 12s slot
6. Validator Proposes the Block
    - Consensus client packages the payload into a Beacon Block and broadcasts it across the network for validation
7. Transaction execution
    - Geth's EVM executes the transaction and updates state(core/state_processor.go)
8. Block is broadcasted, verified, and finalized
    - Transaction now part of Ethereum blockchain



Create new EVM instance

Decode transaction to a Message struct

Assign ID to transaction in the block in StateDB

Reset EVM to current transaction context & stateDB(inside ApplyTransactionWithEVM)

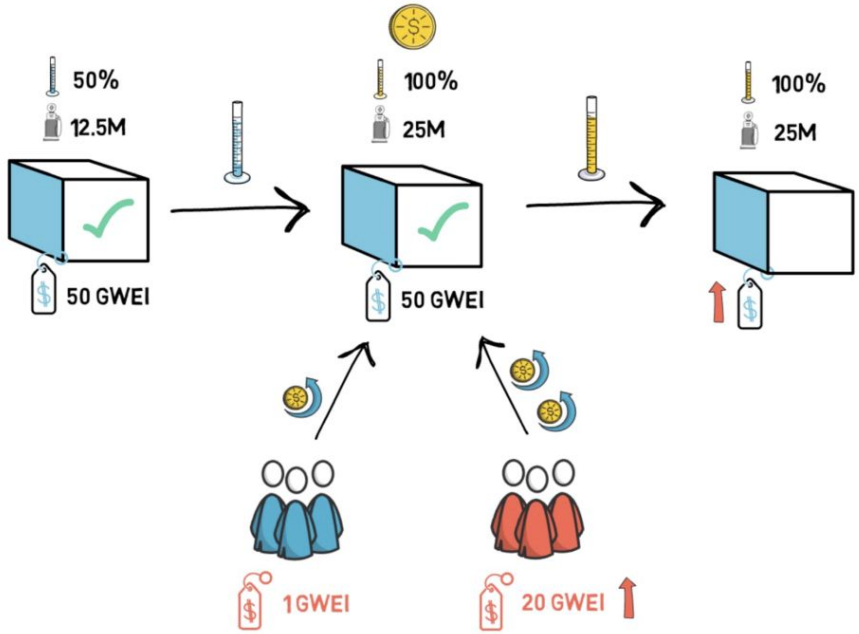
Apply the message to current state (inside ApplyTransactionWithEVM)

Prepare transaction receipt and append it together with logs

Final step: finalize block, apply consensus extras

EIP-1559 – New Gas Mechanism

- Introduced:
  - Dynamic base fee
  - Increased gas limit per block
  - Tip
  - Base fees are burned
- Base fee adjusts per block based on gas usage
  - Aims to achieve equilibrium of 50% network utilization
- Located in core/tx_pool, core/fee.go, and params modules
- Gas refunds calculated in ApplyTransaction() in core/state_transition.go



Executing A Transaction After EIP-1559

- User specifies different arguments in a TransactOpts struct which is used to create a transaction
  - `From` and `Signer` are required
- Tipping mechanism introduced in EIP-1599:
  - Used to increase speed of transaction by incentivizing validators to choose your transaction

```go
// TransactOpts is the collection of authorization data required to create a
// valid Ethereum transaction.
type TransactOpts struct {
    From   common.Address // Ethereum account to send the transaction from
    Nonce  *big.Int       // Nonce to use for the transaction execution (nil = use pending state)
    Signer SignerFn       // Method to use for signing the transaction (mandatory)

    Value      *big.Int         // Funds to transfer along the transaction (nil = 0 = no funds)
    GasPrice   *big.Int         // Gas price to use for the transaction execution (nil = gas price oracle)
    GasFeeCap  *big.Int         // Gas fee cap to use for the 1559 transaction execution (nil = gas price oracle)
    GasTipCap  *big.Int         // Gas priority fee cap to use for the 1559 transaction execution (nil = gas price oracle)
    GasLimit   uint64           // Gas limit to set for the transaction execution (0 = estimate)
    AccessList types.AccessList // Access list to set for the transaction execution (nil = no access list)

    Context context.Context // Network context to support cancellation and timeouts (nil = no timeout)

    NoSend bool // Do all transact steps but do not send the transaction
}
```

```go
// Create the transaction
var (
    rawTx *types.Transaction
    err   error
)
if opts.GasPrice != nil {
    rawTx, err = c.createLegacyTx(opts, contract, input)
} else if opts.GasFeeCap != nil && opts.GasTipCap != nil {
    rawTx, err = c.createDynamicTx(opts, contract, input, nil)
} else {
```

Executing A Transaction After EIP-1559 (cont.)

- Transaction is signed
- Scheduled for execution unless user sets NoSend = true

```
if opts.Signer == nil {
    return nil, errors.New("no signer to authorize the transaction with")
}
signedTx, err := opts.Signer(opts.From, rawTx)
if err != nil {
    return nil, err
}
if opts.NoSend {
    return signedTx, nil
}
if err := c.transactor.SendTransaction(ensureContext(opts.Context), signedTx); err != nil {
    return nil, err
}
return signedTx, nil
```

Deployment & Interaction

We can access and interact with the Ethereum blockchain through an execution client. Ethereum's most notable client is Geth. There are three components to accessing the blockchain:

- You need an execution client that runs the transactions and initiates contracts, such as Geth.
- As of the transition to PoS, you need a consensus client to validate transactions and allows the network to agree on the existing block's legitimacy.
- You also need an account management tool to be able to manipulate the blockchain, such as Clef.

As an alternate way to access Geth is using Web3, which is a Python library for communicating the the Ethereum blockchain without the need for directly using Geth or even a consensus client locally.

- With Web3, we can connect to a public node that runs both Geth and a consensus client, so we can worry about transactions and account creation.

Proto-Danksharding (EIP-4844)

Scaling with Blobs

- Introduces a new transaction type: blob-carrying transactions
  - Blobs = large chunks of data used by Layer 2 rollups
- Not stored in the main EVM state; only available temporarily for data availability

- core/types/tx_blob.go: Defines BlobTx structure
- eth/eth.go: Integration of blob tx support
- params/config.go: Enables EIP-4844 under Cancun fork
- eth/downloader/, blob/, internal/ethapi/: support logic

Verkle Trees

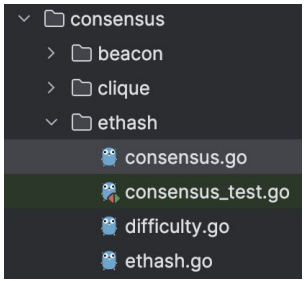
Replacement for Merkle Patricia Trie

Uses vector commitments (e.g., KZG) for compact proofs

Reduces proof size from ~100s of KB to < 1KB

In client code: state database rewrites in progress (core/state, trie/ modules)

Consensus Layer




```go
func (ethash *Ethash) Finalize(chain consensus.ChainHeaderReader, header *types.Header,
    state vm.StateDB, body *types.Body) {
    // Accumulate any block and uncle rewards
    accumulateRewards(chain.Config(), state, header, body.Uncles)
}
```

Block Creation

```go
func (api *ConsensusAPI) getPayload(payloadID engine.PayloadID, full bool) (*engine.ExecutionPayl
    log.Trace( msg: "Engine API request received", ctx...: "method", "GetPayload", "id", payloadID)
    data := api.localBlocks.get(payloadID, full)
    if data == nil { return nil, engine.UnknownPayload }
    return data, nil
}
```

```go
type ExecutionPayloadEnvelope struct { 17 usages
    ExecutionPayload *ExecutableData  `json:"exec
    BlockValue       *big.Int         `json:"bloc
    BlobsBundle      *BlobsBundleV1   `json:"blob
    Requests         [][]byte         `json:"exec
    Override         bool             `json:"shou
    Witness          *hexutil.Bytes   `json:"witn
}
```

```go
type ExecutableData struct { 31 usages  Mariu
    ParentHash       common.Hash
    FeeRecipient     common.Address
    StateRoot        common.Hash
    ReceiptsRoot     common.Hash
    LogsBloom        []byte
    Random           common.Hash
    Number           uint64
    GasLimit         uint64
    GasUsed          uint64
    Timestamp        uint64
    ExtraData        []byte
    BaseFeePerGas    *big.Int
    BlockHash        common.Hash
    Transactions     [][]byte
    Withdrawals      []*types.Withdrawal
    BlobGasUsed      *uint64
    ExcessBlobGas    *uint64
    ExecutionWitness *types.ExecutionWitness
}
```

Thank you