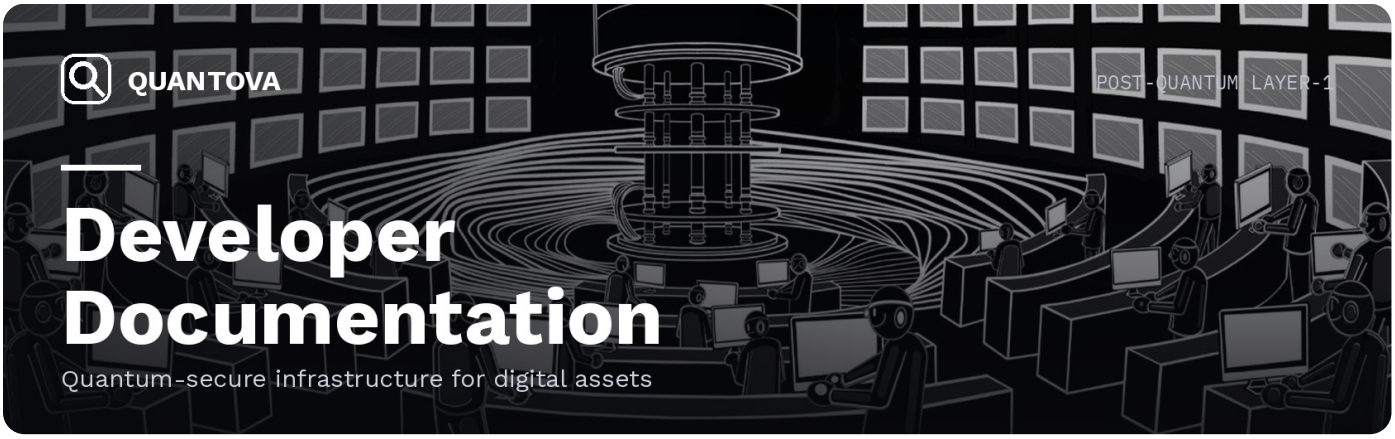


Developer Documentation

Quantum-secure infrastructure for digital assets

Welcome to the official developer documentation for **Quantova** — a quantum-secure Layer-1 blockchain that protects digital assets for governments, institutions, and retail users. Every account, transaction, contract, and bridge is secured end to end with NIST-standardized post-quantum cryptography. This reference takes you from your first signed transaction to running validators, deploying contracts on the QVM, and integrating the SDK.

START HERE — POPULAR PAGES

Quick Start SDK, account, first transaction	**Testnet & Faucet** Claim free TQTOV on Qtox.io	**Accounts & Keys** Post-quantum Q-addresses	**Smart Contracts** Q primitives on the QVM	**JSON-RPC** The q_ method reference

QUANTUM-SECURE BUILDING BLOCKS

CRYSTALS-Dilithium · **Falcon** · **SPHINCS+** · **SHA3-256 hashing** · **Q-address scheme** · **NPoS consensus** · **QVM · Solidity** · **QTE encrypted lane** · **Cross-chain bridges** · **QTOV** · **Qmask.io wallet** · **Qdock.io IDE**

GITHUB	DEVELOPER DISCORD	SDK & FAUCET
github.com/Quantova Source releases, runtime, SDK, and reference implementations.	**discord.gg/4Z6Zrx5s** Build help, protocol discussion, and announcements.	**npmjs.com/~quantova · Qtox.io** Install the @quantova SDK; claim free TQTOV testnet tokens.

Table of Contents

1. Read Me First

Quantova is a post-quantum Layer-1 blockchain. This document is the canonical developer reference: it explains the protocol, the account and consensus model, the economic system, and the complete toolchain — SDK, wallet, IDE, RPC, and nodes — with runnable code throughout.

If you read nothing else, read this page. It tells you what Quantova is, who the documentation is for, and the fastest path from here to a working integration.

1.1 What Quantova Is

Quantova is an independent Layer-1 blockchain whose cryptography is post-quantum from the signature layer up. Every account is secured by a NIST-standardized post-quantum signature scheme, every hash is SHA3-256, consensus avoids quantum-vulnerable randomness, and the smart-contract environment (the Quantova Virtual Machine, or QVM) runs Solidity compiled to QVM bytecode. The native asset is **QTOV**.

QTOV	~2.5 s	~3 s	18
NATIVE ASSET	BLOCK TIME	FINALITY	DECIMALS

The platform ships with first-party developer tooling: the **@quantova** SDK (JavaScript/TypeScript), `qweb3.js` for QVM contracts, the **Qmask.io** post-quantum wallet (browser web extension), and the **Qdock.io** IDE for compiling and deploying Solidity to the QVM. A JSON-RPC `q_` namespace plus REST developer endpoints make the chain reachable from any language.

1.2 Who This Is For

Audience	What you will find here
Application developers	Quick Start (3), the SDK cookbook (15) with JavaScript and Python, contract examples (14), and the JSON-RPC reference (16).
Smart-contract engineers	The QVM model, Q primitive precompiles, a full token contract, Quantova structures, and Qdock.io / Qmask.io deployment (14).
Validators & node operators	Consensus (6), staking & validator economics (10), node build/run and operator RPC (17), and chain specifications (19).
Cryptographers & researchers	Account derivation and the signature envelope (5), the no-VRF consensus rationale (6), and QTE encrypted execution (7).
Token holders & the community	Tokenomics (9), the issuance and fee-distribution model, QTOV voting weight, and the full governance structure (11).

1.3 How to Read This Document

The chapters are ordered so each builds on the last, but they are self-contained enough to jump between. A typical path by goal:

1. **Build an app fast:** Quick Start (3) → SDK & Code Examples (15) → JSON-RPC Reference (16).
2. **Ship a contract:** Accounts & Keys (5) → Smart Contracts on the QVM (14) → Deploy with Qdock.io & Qmask.io (14.5).
3. **Run a validator:** Consensus (6) → Staking & Validator Economics (10) → Running Nodes (17) → Chain Specifications (19).
4. **Understand the economy:** Economic Model & Fees (8) → QTOV Tokenomics (9) → Governance (11).

Conventions used throughout

Inline identifiers like `q_getBalance` are code. Endpoints are shown for mainnet (`wss://mainnet.quantova.io`); swap in your local node where noted. Addresses are shown in the canonical **Q** display form for humans and in `0x` 20-byte hex for the RPC wire. Economic figures are the network's current configured values; consensus-critical numbers are repeated in Chapter 19.

1.4 Glossary

QTOV
The native asset: gas, staking, governance, and bridge-settlement token (18 decimals).
QVM
Quantova Virtual Machine — the Solidity-compatible smart-contract execution environment.
Q primitives
Native post-quantum capabilities (hashing, signature verification, QNS, QTE) exposed to contracts as precompiles.
Qmask.io
The post-quantum wallet, delivered as a browser web extension; signs all transaction types.
Qdock.io
The Quantova IDE for writing, compiling, and deploying Solidity to the QVM (signs via Qmask.io).
QTE
Q Threshold Encryption — an encrypted execution lane that resists front-running.
QNS
Quantova Naming Service — on-chain human-readable names.
Locked QTOV
QTOV locked for a duration to gain a higher governance voting multiplier (9.7).
QGAS
The gas unit metering QVM execution; priced in QTOV.
Dilithium / Falcon / SPHINCS+
The three NIST post-quantum signature schemes Quantova accounts use.

2. Introduction

Public-key blockchains authenticate every transaction with elliptic-curve signatures — ECDSA on secp256k1, or Ed25519/sr25519. Shor's algorithm recovers the private key from a public key in polynomial time on a fault-tolerant quantum computer, which collapses that authentication entirely. Quantova removes the elliptic curve from the trust base and signs with NIST-standardized lattice and hash-based schemes instead, so accounts, consensus, contracts, and bridges do not depend on a problem a quantum adversary can solve.

2.1 Mission

The exposure is not purely prospective. A harvest-now, decrypt-later adversary can record on-chain public keys and signatures today and forge against them once hardware matures — and on a transparent ledger every spent account has already published its public key. Quantova is built so that the signature an account publishes carries no future liability. It serves three constituencies:

- **Governments and public-sector departments** holding sovereign and citizen assets under confidentiality and integrity requirements that outlast the arrival of quantum hardware.
- **Institutional and enterprise departments** that need auditable, standards-based custody and settlement they can defend to a regulator.
- **Retail users** who inherit the same post-quantum protection by default, with no configuration and no cryptographic decisions to get wrong.

Every account is secured by one of three NIST-standardized signature schemes — CRYSTALS-Dilithium (ML-DSA), Falcon (FN-DSA), and SPHINCS+ (SLH-DSA) — with SHA3-256 for hashing. A developer building on Quantova writes ordinary application code; the post-quantum security is a property of the protocol, not something the developer assembles.

2.2 Quantova Is Not a Fork

An independent Layer-1
Quantova is not a fork of any existing blockchain. It does not inherit another network's ledger, genesis, or security assumptions. It is engineered from first principles around post-quantum cryptography.

Quantova is built *with* the Substrate development framework — a toolkit for constructing blockchains — but a framework is not a chain. Using a framework to build an original network is categorically different from forking an existing chain's state and codebase. Quantova's consensus, account model, randomness source, virtual machine, and bridges are each designed for a quantum-secure setting. Where a classical primitive would reintroduce quantum exposure — elliptic-curve account keys, VRF-based randomness — it is removed and replaced, never retained for compatibility.

2.3 Design Principles

Principle	What it means on Quantova
Post-quantum by default	Every externally owned account is secured by Dilithium, Falcon, or SPHINCS+. There is no classical account type for users.
Algorithm agility	All three signature schemes share one address space; schemes can be added or retired through governance without changing the address format.
Quantum-safe hashing	SHA3-256 across storage, transaction, and state-trie hashing, with a defined upgrade path to SHA3-512 or larger.
No quantum-vulnerable randomness	Deterministic, round-robin slot leadership; randomness derived from post-quantum primitives, never a VRF.
Decentralized by construction	No superuser key. All network changes flow exclusively through on-chain governance.
Disinflationary issuance	A hardcoded, declining issuance schedule; fees are collected to the treasury and validators, never burned.
Trust-minimized interoperability	Bridges verify finalized source-chain state with light clients, Merkle/storage proofs, and zero-knowledge proofs inside the runtime.

2.4 The Quantum Threat & Quantova's Response

A sufficiently large quantum computer running Shor's algorithm recovers an elliptic-curve private key from its public key — fatal for any account whose public key has appeared on-chain. Grover's algorithm weakens hash functions, though far less severely (a quadratic, not exponential, speedup). Quantova's response is structural rather than cosmetic:

Classical exposure	Quantova's replacement
ECDSA / sr25519 account keys (Shor-breakable)	Dilithium / Falcon / SPHINCS+ signatures (NIST PQ standards)
VRF-based leader election (classical EC)	Deterministic round-robin slot leadership
Keccak/blake2 in some classical stacks	SHA3-256 everywhere, upgradeable to SHA3-512+
Hard forks for cryptographic upgrades	Forkless runtime upgrades via on-chain governance

Further reading: Post-quantum cryptography · Shor's algorithm · Grover's algorithm · NIST PQC standardization

2.5 Architecture at a Glance

Quantova separates concerns into protocol layers, each of which has its own chapter:

```
TEXT
+-------------------------------------------------------------+
|  Tooling     Qmask.io wallet  .  Qdock.io IDE  .  @quantova SDK |
+-------------------------------------------------------------+
|  Interfaces  JSON-RPC (q_ namespace)  .  REST  .  WebSocket     |
+-------------------------------------------------------------+
|  Execution   QVM (Solidity -> QVM bytecode)  .  Q primitives    |
+-------------------------------------------------------------+
|  Runtime     Staking . Governance . Bridges . QNS . QTE         |
+-------------------------------------------------------------+
|  Consensus   Nominated Proof-of-Stake (NPoS) . No VRF leader pick |
+-------------------------------------------------------------+
|  Crypto      Dilithium / Falcon / SPHINCS+  .  SHA3-256         |
+-------------------------------------------------------------+
```

Two execution planes share one account model: **runtime extrinsics** drive protocol modules (staking, governance, bridges, naming), and **QVM transactions** drive smart contracts through the `q_` namespace. Both are post-quantum signed and reach the same deterministic finality.

3. Quick Start

From an empty project to reading live chain state, creating a post-quantum account, sending QTOV, and deploying a contract. Every step is runnable.

The path below is the fastest route from nothing to a working integration. Each numbered step maps to a section in this chapter; follow them in order:

1. **Install** Node.js 18+, the `@quantova` SDK from npm, and the Qmask.io wallet (3.1).
2. **Connect** to a node over WebSocket and confirm the chain identity (3.2).
3. **Create** a post-quantum account — pick Dilithium, Falcon, or SPHINCS+ (3.3).
4. **Read & send:** query a balance with a `q_` call, then submit a signed transfer (3.4).
5. **Deploy** a Solidity contract to the QVM and call it (3.5).

> **What talks to what**
> **@quantova/api** opens the WebSocket session and reads runtime state. The `q_` **namespace** is JSON-RPC for QVM calls (balances, gas, contract reads). **qweb3.js** builds and signs contract transactions. **Qmask.io** holds keys and signs; **Qdock.io** compiles and deploys Solidity. You will use the SDK for steps 2–4 and Qdock.io/qweb3.js for step 5.

3.1 Install the SDK & Wallet

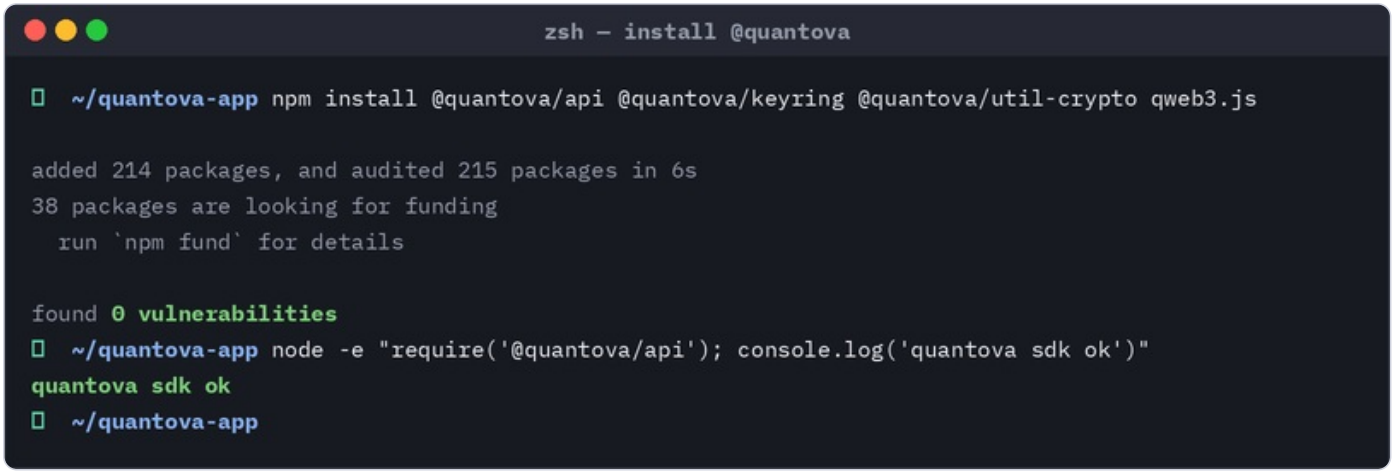
The post-quantum SDK is published under the @quantova scope on the public npm registry (`registry.npmjs.org`). The high-level client is `@quantova/api` ; `qweb3.js` is the QVM contract client. You need Node.js 18+ and npm 9+ on your path:

```
BASH
# 1. verify your toolchain (Node 18+ is required for the WASM PQ backends)
node --version        # v18.0.0 or newer
npm  --version        # 9.0.0  or newer

# 2. start a project if you do not have one
mkdir quantova-app && cd quantova-app
npm init -y

# 3. install the @quantova SDK from the public npm registry
npm install @quantova/api @quantova/keyring @quantova/util-crypto qweb3.js
```

The packages resolve from the `@quantova` registry scope; no private registry, token, or `.npmrc` entry is required. A clean install reports zero vulnerabilities and the SDK loads immediately:



Installing the @quantova SDK from the public npm registry, then loading it in Node to confirm the install.

For Python, the `q_` JSON-RPC namespace is plain HTTP, so a single HTTP client is enough — there is no separate native dependency to compile:

BASH

```bash
python3 -m venv .venv && source .venv/bin/activate
pip install requests          # only an HTTP client is needed for the q_ namespace
```

Interactive users sign with **Qmask.io**, the post-quantum wallet (browser web extension). Qmask.io derives and stores Dilithium / Falcon / SPHINCS+ keys, ships with Quantova's network configuration preloaded (RPC endpoint, chain id, QTOV as the native currency), and signs every transaction type: transfers, QVM calls, staking, governance, QNS, and QTE.

> **Two tools, two jobs**
> **Qmask.io** is the wallet and signer. **Qdock.io** is the IDE that compiles and deploys Solidity to the QVM; it signs deployments through Qmask.io. See 13.5.

3.2 Connect to a Node

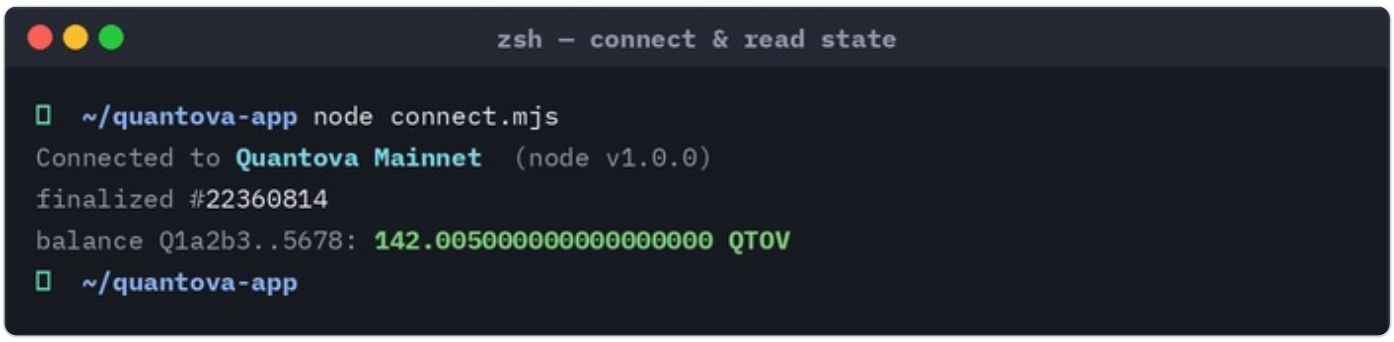
Open a WebSocket session to a node and read its chain and runtime identity. Save this as `connect.mjs` and run it with `node connect.mjs`:

JAVASCRIPT

```javascript
import { ApiPromise, WsProvider } from '@quantova/api';

const provider = new WsProvider('wss://mainnet.quantova.io'); // or ws://127.0.0.1:9944
const api = await ApiPromise.create({ provider });

const [chain, version] = await Promise.all([
  api.rpc.system.chain(),
  api.rpc.system.version(),
]);
console.log(`Connected to ${chain} (node ${version})`);
console.log('runtime:', api.runtimeVersion.specName.toString(),
            api.runtimeVersion.specVersion.toNumber());
```



Connecting to mainnet, reading the finalized head, and printing an account balance.

Environment	WebSocket	HTTP JSON-RPC
Mainnet	`wss://mainnet.quantova.io`	`https://mainnet.quantova.io`
Local dev	`ws://127.0.0.1:9944`	`http://127.0.0.1:9933`

3.3 Create a Quantum-Secure Account

There is **no default signature algorithm** — you choose Dilithium, Falcon, or SPHINCS+ explicitly. All three derive into the same Q-prefixed address space.

JAVASCRIPT

```
import { Keyring } from '@quantova/keyring';
import { cryptoWaitReady, mnemonicGenerate } from '@quantova/util-crypto';

await cryptoWaitReady();                    // initialise the PQ WASM backends

// pick a scheme explicitly: 'dilithium' | 'falcon' | 'sphincsp'
const keyring = new Keyring({ type: 'dilithium' });

// from a generated mnemonic
const phrase = mnemonicGenerate();          // back this up securely
const alice  = keyring.addFromUri(phrase, { name: 'Alice' }, 'dilithium');

console.log('scheme :', alice.type);         // dilithium
console.log('address:', alice.address);      // canonical Quantova address, begins with 'Q'
```

3.4 Read State & Send a Transaction

Read a balance from runtime storage, then have Alice send QTOV to Bob. Both addresses are Q-prefixed; the transaction is signed with Alice's post-quantum key.

JAVASCRIPT

```
// --- read: query runtime storage (an API call) ---
const { data: balance, nonce } = await api.query.system.account(alice.address);
console.log('free:', balance.free.toHuman(), '| nonce:', nonce.toNumber());

// --- write: Alice -> Bob, 1 QTOV (18 decimals) ---
const ONE_QTOV = 1_000_000_000_000_000_000n;
const tx = api.tx.balances.transferKeepAlive(BOB_Q_ADDRESS, ONE_QTOV);

const unsub = await tx.signAndSend(alice, ({ status, dispatchError }) => {
  if (status.isInBlock)    console.log('in block  ', status.asInBlock.toHex());
  if (status.isFinalized) { console.log('finalized ', status.asFinalized.toHex()); unsub(); }
  if (dispatchError)       console.error('failed:', dispatchError.toString());
});
```

3.5 Deploy Your First Contract

Compile Solidity to QVM bytecode (in Qdock.io or with `solc`) and deploy with a QVM transaction where `to = null`, signed by a post-quantum account.

JAVASCRIPT

```
import { QWeb3 } from 'qweb3.js';

const q = new QWeb3('https://mainnet.quantova.io');

const deployTx = await q.qvm.buildTransaction({
  to:   null,                       // null => contract creation
  data: '0x' + contractBytecode,    // from Qdock.io / solc output
  gas:  GAS_LIMIT,                  // QGAS limit (see q_estimateGas)
});

const receipt = await q.qvm.sendSigned(deployTx, account); // PQ-signed (Qmask.io / keyring)
console.log('contract:', receipt.contractAddress);          // shown in canonical Q form
```

Where to go next

Before sending real value, get free testnet tokens and rehearse on the public testnet (Chapter 4). Full contract examples and Quantova structures are in Chapter 14; the complete SDK cookbook (JavaScript and Python) is in Chapter 15; every RPC method is in Chapter 16.

4. Testnet & TQTOV Faucet

Before deploying against mainnet, build and test on the public Quantova testnet using **TQTOV**, the free testnet asset. TQTOV behaves like mainnet QTOV for development purposes — it pays test fees, funds test stake, and exercises contracts and bridges — but it is distributed at no cost from a faucet and carries no monetary value.

4.1 The Quantova Testnet

The testnet is a full Quantova network running the same post-quantum runtime, NPoS consensus, QVM, and `q_` RPC as mainnet, with its own genesis and validator set. It lets developers exercise the entire stack — accounts, transactions, staking, governance, QNS, bridges, and contracts — without spending anything of value. Point the SDK at the testnet endpoints exactly as you would mainnet:

Environment	WebSocket	HTTP JSON-RPC
Testnet	`wss://testnet.quantova.io`	`https://testnet.quantova.io`
Local dev	`ws://127.0.0.1:9944`	`http://127.0.0.1:9933`

Same code, different endpoint

Testnet accounts use the identical Q-address scheme and the same `@quantova` SDK. To move an integration from testnet to mainnet you change only the RPC endpoint — no code changes.

4.2 TQTOV: The Testnet Asset

TQTOV is the native gas and staking asset of the Quantova testnet, mirroring the role QTOV plays on mainnet. It has **no price and no monetary value**: it is not for sale, cannot be purchased, is not redeemable for QTOV or any other asset, and is intended solely for development and testing. Treat any offer to buy or sell TQTOV as fraudulent.

TQTOV	18	Free	$0.00
TESTNET ASSET	DECIMALS	DISTRIBUTED VIA FAUCET	NO VALUE, NOT TRADABLE

4.3 Claiming TQTOV from Qtox.io

The official testnet faucet is **Qtox.io**. The flow follows the same pattern as the faucets on other Layer-1 testnets — you supply an address and receive a capped amount of test tokens. Quantova's faucet works with any wallet that supports the Quantova Q-address scheme:

1. **Generate a Quantova address.** Create an account in the Qmask.io wallet (or any wallet supporting the Q-address scheme), or programmatically with the SDK (3.3). Copy the canonical address — it begins with **Q**.
2. **Open the faucet.** Go to Qtox.io and select the Quantova testnet.
3. **Paste your Q-address** into the address input field. The faucet validates that it is a well-formed Q-address before accepting it.
4. **Set the amount** of TQTOV you want to request, up to the per-request cap (4.4).
5. **Submit the request.** The faucet signs and broadcasts a testnet transfer; the TQTOV arrives in your wallet once the block is finalized (a few seconds).

You can confirm receipt by querying your balance over the testnet RPC, exactly as on mainnet:

BASH

```
curl -s https://testnet.quantova.io \
  -H 'content-type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"q_getBalance",
       "params":["Q0b22901ba019b7f48327a891d21a9af2547493","0x15536ee"]}'
# result is the TQTOV balance in base units (QUANTITY hex)
```

4.4 Claim Limits & Anti-Spam Rules

To keep the faucet available to everyone and prevent draining or spam, Qtox.io rate-limits claims per address and per source — the same approach mainstream testnet faucets use. The limits below are the default policy:

Limit	Policy
Maximum per request	100 TQTOV
Maximum per address	100 TQTOV every 24 hours
Cooldown between requests	One claim per address per 24 hours
Per-IP / per-session cap	Bounded per rolling 24-hour window to deter automated draining
Address validation	Only well-formed Q-addresses are accepted

If you need a larger allocation for load-testing or a public hackathon, request it through the official Quantova developer channels rather than scripting repeated faucet calls; automated abuse is rate-limited and may be temporarily blocked.

> **Why faucets are capped**
> A small per-request cap with a 24-hour cooldown is standard across Layer-1 testnets: it spreads a finite pool of valueless tokens across many developers, keeps the faucet responsive, and removes any incentive to hoard an asset that has no value in the first place.

4.5 What TQTOV Is Used For

TQTOV unlocks the whole Quantova development surface so you can validate an integration end to end before touching mainnet. Across the ecosystem it is used to:

- **Pay testnet transaction fees** — transfers, contract calls, and QVM deployments are all metered in TQTOV (the fee model in Chapter 8 applies on testnet).
- **Deploy and call contracts** — ship Solidity to the testnet QVM through Qdock.io and exercise it with real signed transactions.
- **Test staking and delegation** — stake TQTOV to a testnet validator, observe rewards and slashing behaviour (Chapters 9–10) safely.
- **Rehearse governance** — raise and vote on testnet referenda with TQTOV-denominated bonds and quorums, without risking value.
- **Exercise the bridges and QNS** — register testnet names and move test assets across the testnet bridge endpoints (Chapters 12–13).
- **Build and demo dApps** — fund test users and wallets in a demo so an application behaves exactly as it will on mainnet.

> **TQTOV has no value**
> TQTOV exists only to make testing realistic. It has no price, is not an investment, is not tradable, and is not convertible to QTOV or fiat. Anyone offering to buy, sell, or exchange TQTOV is acting fraudulently. Testnet state may be reset, and TQTOV balances may be cleared, at any time.

5. Accounts & Keys

Quantova uses a post-quantum account model. Every externally owned account is secured by a post-quantum signature scheme — there is no classical account type for users. Three schemes are supported, the holder chooses one, and all three share a single address space.

5.1 Signature Schemes

Scheme	NIST standard	Family	Characteristics
CRYSTALS-Dilithium	ML-DSA	Module-lattice	Balanced signature size and verification speed; the general-purpose default choice.
Falcon	FN-DSA	NTRU-lattice	**Smallest signatures**; used for protocol and validator keys where size matters.
SPHINCS+	SLH-DSA	Stateless hash-based	Most conservative security assumptions; larger signatures, ideal for high-assurance accounts.

There is no default scheme imposed by the protocol — the user explicitly chooses one at account creation, and the choice is bound to the address at derivation time. The SDK exposes all three as keyring types: `'dilithium'`, `'falcon'`, `'sphincsp'`.

Further reading: CRYSTALS-Dilithium (ML-DSA) · Falcon (FN-DSA) · SPHINCS+ (SLH-DSA) · Lattice-based cryptography · SHA-3

5.2 Unified Account Model

A single account identity is used across the entire protocol. The same account signs:

- Runtime, protocol-level transactions (transfers, staking, governance, naming).
- QVM contract calls and deployments.
- Encrypted (QTE) submissions and bridge operations.

There is no separation between a “runtime account” and a “QVM account”. This works because all three signature schemes derive into one shared address space: whichever scheme is chosen, the resulting address has the same Q-prefixed format and width.

5.3 The Q Address Format

The Q prefix is by design
Every Quantova address is written in canonical form with a leading **Q**. This is Quantova's native address brand — the format users, wallets, and explorers see. Qmask.io and any third-party wallet built on the SDK inherit it automatically.

When Alice sends to Bob, both parties identify each other by their Q-prefixed addresses:

Alice `Q1a2b3c4d5e6f78901234567890abcdef12345678`

Bob `Q9f3c7a1d4e8b29f6a3c5d1e7f8a9b2c4d6e8f1a3`

The Q prefix is anchored on-chain: address derivation fixes the first byte of every post-quantum account to `0x40` — the Quantova account marker that the canonical form renders as the leading **Q**. Because the marker is fixed at the protocol level, *every* valid account address begins with Q; an address that does not is not a Quantova account.

Representation	Where it is used	Shape
Canonical "Q" form	Wallets (Qmask.io), SDK, explorers, `q_` RPC, QVM Solidity, user-to-user	`Q` + address body
Hash / DATA	Block & transaction hashes, digests, calldata (not accounts)	`0x`-prefixed hex

Q for addresses, 0x for hashes
There is one address representation: the Q form. It is what humans, wallets, explorers, the SDK, the `q_` RPC, and QVM Solidity all use. The `0x` hex notation is reserved for values that are *not* accounts — block and transaction hashes, SHA3 digests, and calldata. A `0x` value is raw bytes; a Q value is an account. This separation is Quantova's own and is not inherited from Ethereum (see 16.1).

5.4 Address Derivation

Public keys for the three schemes have very different sizes, so the address is derived from a hash of the public key. The derivation is identical for all three schemes:

1. Compute `SHA3-256(publicKey)` → a fixed **32-byte digest**.
2. Take the **first 20 bytes** of the digest as the account body (H160 width).
3. Overwrite byte `0` with the fixed prefix `0x40` — the Q marker.
4. Render canonically with the leading **Q** — this Q-format string is the address used everywhere, including the `q_` RPC.

Why 20 bytes, not 32
The SHA3-256 *digest* is 32 bytes, but the *address* is the first 20 bytes (H160). Because the leading byte is the fixed Q marker, the address carries **152 bits** of derived entropy. Addresses are H160-width — they are not 32-byte addresses. See the FAQ for the full rationale for hashing the public key.

5.5 Signature Envelope (QSignature)

A submitted signature is wrapped in a self-describing envelope so the runtime knows which scheme to verify with:

```
TEXT
QSignature = variant_byte || length_prefixed_signature || public_key
```

Variant byte	Scheme
0	SPHINCS+
1	Falcon
2	Dilithium

Because the public key travels in the envelope and the address is a hash of that key, the public key is revealed only when an account first signs — not before. This preserves preimage resistance for accounts that have only ever received funds.

5.6 Working With Keys in Code

Generate a keypair, inspect its address, sign a message, and verify — all post-quantum:

JAVASCRIPT

```
import { Keyring } from '@quantova/keyring';
import { cryptoWaitReady, signatureVerify, stringToU8a } from '@quantova/util-crypto';

await cryptoWaitReady();

const keyring = new Keyring({ type: 'falcon' });          // smallest signatures
const acct = keyring.addFromUri('//Alice', { name: 'Alice' }, 'falcon');

const message = stringToU8a('approve transfer of 10 QTOV');
const signature = acct.sign(message);                     // Falcon signature

const { isValid } = signatureVerify(message, signature, acct.address);
console.log('address  :', acct.address);                  // Q...
console.log('scheme   :', acct.type);                     // falcon
console.log('verified :', isValid);                       // true
```

The keyring abstracts the envelope: `acct.sign()` returns a scheme-tagged signature, and `signatureVerify()` reads the variant byte to select the verifier automatically.

6. Consensus & Finality

Quantova secures the network with **Nominated Proof-of-Stake (NPoS)**: a bounded active validator set is elected from validators and their nominators' combined stake (Chapter 10). On top of NPoS, Quantova separates block production from finality and removes the quantum-vulnerable randomness that classical Substrate consensus depends on.

Three properties define the consensus layer: stake-weighted validator selection (**NPoS**), **deterministic** slot leadership with no VRF, and **deterministic, provable finality** with post-quantum (Falcon) authority keys. Production can briefly fork at the head; finality then commits a single canonical chain.

Further reading: Proof of stake (overview) · Nominated Proof-of-Stake (NPoS) · Verifiable random function

6.1 No VRF: Deterministic Slot Leadership

Classical BABE selects block authors with a Verifiable Random Function (VRF). A VRF relies on classical cryptography, and **there is no widely accepted post-quantum VRF** in the literature today. Rather than ship a primitive with an unproven quantum-security story, Quantova removes VRF-based leader election entirely.

- **Slot assignment is round-robin, deterministic, and predictable.** The eligible author for each slot is fixed by the active validator order — there is no per-slot random draw.
- **Randomness needed elsewhere is sourced from Quantova's post-quantum primitives**, removing the quantum vulnerability a VRF would reintroduce in the randomness path.

> **Migration note for Substrate developers**
> The base stack used classical EC keys for validator identity and a VRF for slot leadership. Quantova replaces validator authority keys with post-quantum primitives (Falcon) and replaces the VRF with deterministic slot allocation. Post-quantum VRF research is tracked as future work, not part of the current implementation.

6.2 Block Production

Block production is slot-based. In each slot the scheduled validator authors a block, which is gossiped and imported. Because leadership is deterministic, peers independently verify that the author was the legitimate validator for that slot without checking a randomized eligibility proof. Target block time is **~2.5 seconds**.

6.3 Finality

Finality is deterministic and provable. Block production can briefly fork at the head; the validator set then votes to finalize a single canonical chain, reaching finality in **~3 seconds**. Once finalized, a block is irreversible unless a supermajority of validators violates protocol assumptions. **Validator authority and signing keys at the finality layer are post-quantum (Falcon).**

Finality yields two properties: users gain irreversibility once a transaction's block is finalized, and bridges anchor outbound proofs to finalized blocks before a withdrawal can be claimed on a remote chain.

6.4 Validator Set & Timing

There are **no separate validator authority sets** for production and finality — the same elected set handles both. Validators are chosen through Nominated Proof-of-Stake (Chapter 10). All time-based behaviour is block-derived, so it scales automatically if block time changes.

Block time
~2.5 seconds (target)
Finality
~3 seconds (deterministic, provable finality)
Epoch length
24 hours
Active validator cap
200 (target at maturity)
Consensus
Nominated Proof-of-Stake (NPoS)

7. Transactions

A transaction is a signed instruction that modifies state. All transactions are validated, included in blocks, executed deterministically inside the block state-transition function, and committed to the SHA3-256 state root.

7.1 Two Transaction Planes

Plane	Targets	RPC surface
Runtime extrinsics	Protocol modules: staking, governance, bridges, naming, validator ops	`author_*` / WebSocket
QVM transactions	Smart-contract execution in the Quantova Virtual Machine	`q_*` namespace

Both planes share the same account model, post-quantum signing, nonce enforcement, fee validation, deterministic execution, and finality.

7.2 Lifecycle

1. **Construction** — build the payload (call/calldata, value, nonce, fees; chain id and gas limit for QVM).
2. **Signing** — signed with the account's post-quantum key via Qmask.io or the SDK keyring, wrapped in the QSignature envelope.
3. **Submission** — `q_sendRawTransaction` for QVM; `author_submitExtrinsic` for runtime calls.
4. **Validation** — signature, nonce, gas/weight and fee checks; valid transactions enter the mempool.
5. **Execution & finality** — included, executed deterministically, then finalized and irreversible.

7.3 Constructing & Signing in Code

A runtime extrinsic with an explicit nonce and mortality (validity window), signed and submitted:

JAVASCRIPT

```javascript
// build a batch: transfer + remark, signed once
const calls = [
  api.tx.balances.transferKeepAlive(BOB_Q_ADDRESS, 5_000_000_000_000_000_000n),
  api.tx.system.remarkWithEvent('hello quantova'),
];

const nonce = (await api.rpc.system.accountNextIndex(alice.address)).toNumber();

const hash = await api.tx.utility.batchAll(calls).signAndSend(alice, {
  nonce,
  era: 64,            // mortal transaction: valid for ~64 blocks
});
console.log('submitted batch:', hash.toHex());
```

7.4 Encrypted Lane (Q Threshold Encryption)

QTE is an encrypted execution lane that protects sensitive transactions from front-running:

- **What is encrypted:** the *full signed post-quantum transaction*, not just calldata. The user encrypts the entire signed transaction with the chain's current public encryption key.
- **Key rotation:** the chain's encryption key rotates every epoch with the validator set, so a captured key is useful for only one epoch.
- **When revealed:** the ciphertext is submitted on-chain, execution order is fixed, and the transaction is decrypted only after the block is finalized.
- **Why it resists front-running:** plaintext does not exist until after ordering and finality, so it cannot be reordered around or MEV-extracted. Decrypting early would require a majority of validators — an eclipse or ≥67% attack.

What QTE does not hide

QTE hides transaction contents, but not that an encrypted submission occurred, the ciphertext length, or the targeted round. Execution is round-based and sequential, keeping the system deterministic.

Contracts submit ciphertext through the QTE precompile, reserved in the Q address space at `Qbabb37461b793748e239608abf906a6fc115aa`; see 14.2 for the precompile set and the `QPrimitives` library.

8. Economic Model & Fees

Quantova prices transactions with an EIP-1559-style fee model: an algorithmic base fee plus a user-set priority fee that incentivizes validator inclusion. Fees are not burned — the base fee is collected to the Quantova treasury and the priority fee is paid to validators. Fees are predictable and low, targeting **$0.05–$0.10** per transaction.

8.1 The Fee Model

TEXT

```
Total Transaction Fee = Base Fee + Priority Fee
```

BASE FEE MECHANICS

- **Algorithmic:** set by the previous block's gas usage relative to target.
- **Block target utilization:** 50%.
- **Maximum change per block:** ±12.5%.
- **Denomination:** QTOV, in fractional units (18 decimals).
- **Disposition:** collected to the Quantova treasury — **no QTOV is burned**.

Quantova does not use a fee-burning mechanism. Collected fees are revenue for the organization: the base fee is routed to the **Quantova treasury** (operated by Quantova Inc for the organization and ecosystem), and the priority fee — the user's tip — is paid to the **validator** that includes the transaction. Every unit of fee is retained and redistributed; none is destroyed.

Component	Rate	Destination
Base fee	Algorithmic	100% to the Quantova treasury (organization)
Priority fee	User-set tip	100% to the including validator

> **Fees are revenue, not burned**
> Quantova collects transaction fees as organizational revenue. The base fee funds the Quantova treasury; the priority fee compensates validators for inclusion. There is no burn, so fees do not reduce supply — they circulate from users to the treasury and validators.

8.2 Transaction Fees

Transaction fees target a low, predictable band of **$0.05–$0.10** per transaction, denominated in USD-equivalent QTOV. Simple operations sit at the low end and computation-heavy ones at the high end:

Operation	Estimated fee (USD)
Native QTOV transfer	~$0.05
Token transfer (QRC-20)	~$0.06
Token swap (DEX)	~$0.08
NFT mint	~$0.10
Smart-contract deployment	~$0.10 – $0.50

Estimate fees programmatically before submitting. The `q_` namespace exposes gas and fee-history methods (16.3); REST developer endpoints provide suggestions and simulation (16.4):

```bash
GET  /v1/fees/estimate    # suggested base fee + priority fee
POST /v1/fees/simulate    # simulate total fee for a proposed transaction
```

8.3 Contract Deployment Fees

Deploying a contract costs QGAS for constructor execution plus a storage cost for the deployed bytecode. Deployment is a QVM transaction with `to = null`; total cost scales with code size and constructor work and typically falls in the **$0.10–$0.50** range. Estimate precisely with `q_estimateGas` against the deployment payload, then multiply by the current `q_gasPrice`.

```javascript
const gas   = await q.rpc('q_estimateGas', [{ from: FROM, to: null, data: '0x'+bytecode }]);
const price = await q.rpc('q_gasPrice', []);
console.log('est. gas:', parseInt(gas,16), '| gas price:', parseInt(price,16));
```

8.4 Bridge Fees

Cross-chain operations are priced in QTOV and cover protocol revenue, validator attestation, and destination-chain gas. As with transaction fees, bridge fees are collected as revenue — none is burned:

```text
Total Bridge Fee = Protocol Fee + Validator Attestation Fee + Destination Gas
```

Component	Rate	Destination
Protocol fee	0.05–0.10% of bridged amount	Quantova treasury (organization)
Validator attestation fee	Flat fee in QTOV	100% to attesting validators
Destination gas	Variable, paid in QTOV (swapped at destination)	Destination chain

ASSET-TIER PROTOCOL FEE

Asset type	Protocol fee
Native QTOV	0.05% (reduced tier)
Major stablecoins (USDC, USDT, DAI)	0.10%
Major assets (ETH, WBTC, BNB, SOL)	0.10%
Long-tail / wrapped assets	0.15%

Bridges are permissionless: there are no protocol-level minimums, maximums, or per-address caps. Settlement timing and the security model are detailed in Chapter 12.

8.5 Governance Bonds

Governance has no transaction “fee” beyond the standard weight fee, but proposals require QTOV bonds to prevent spam. The bond is returned if a proposal reaches the engagement threshold and forfeited to the **Quantova treasury** if it fails to. Full bond and referendum tables are in Chapter 11.

9. QTOV Tokenomics

QTOV is the native asset of Quantova. It uses an uncapped, disinflationary issuance model: new issuance funds network security and declines on a hardcoded schedule. Transaction, bridge, and governance fees are collected as revenue to the Quantova treasury and to validators — Quantova does not burn QTOV.

1,050,000,000	18	5.0% → 1.5%	Treasury
GENESIS SUPPLY (QTOV)	DECIMALS	ISSUANCE (YR 1 → TERMINAL)	FEE DESTINATION

9.1 Core Parameters

Native asset
QTOV
Token standard
Quantova native
Decimals
18
Genesis supply
1,050,000,000 QTOV
Supply model
Uncapped, disinflationary issuance (no burn)
Transaction fee
$0.05–$0.10 (collected to treasury & validators)
Block time
~2.5 seconds
Finality
~3 seconds (deterministic, provable finality)

9.2 Token Functions

Function	Role
Gas token	Required to pay transaction fees on all Quantova transactions.
Staking asset	Required for validator participation and delegation.
Governance token	Voting power on protocol parameters, treasury actions, and ecosystem decisions.
Bridge settlement asset	Native settlement asset for all cross-chain operations.
Ecosystem collateral	Base collateral for Quantova-native DeFi protocols.
Treasury revenue asset	Fees collected across the network accrue to the Quantova treasury in QTOV.

9.3 Genesis Allocation

The 1,050,000,000 QTOV genesis supply is allocated across categories supporting network security, ecosystem growth, and treasury durability. The distribution is weighted toward broad participation: a larger public sale and a dedicated institutional round give external investors and institutions a meaningful share, in line with how recent Layer-1 networks structure their raises. All allocations are enforced by audited on-chain vesting contracts with publicly verifiable addresses.

Allocation	%	Tokens (QTOV)
Community & Ecosystem	30.00%	315,000,000
Foundation / Treasury	20.00%	210,000,000
Validator Bootstrap Rewards	14.00%	147,000,000
Team & Core Contributors	11.00%	115,500,000
Private & Institutional Round	10.00%	105,000,000
Strategic Reserve	8.00%	84,000,000
Public Sale	7.00%	73,500,000

WHAT EACH CATEGORY FUNDS

- **Community & Ecosystem:** grants, airdrops, developer incentives, and liquidity programs.
- **Foundation / Treasury:** sustained Foundation operations, partnerships, audits, regulatory engagement, and ecosystem development.
- **Validator Bootstrap Rewards:** pre-funded staking rewards distributed to validators over the network's first four years.
- **Team & Core Contributors:** founders, core engineers, and contributors under multi-year vesting.
- **Private & Institutional Round:** allocations for institutional investors, funds, and strategic partners, under multi-year vesting to align long-horizon backers with the network.
- **Strategic Reserve:** held by the Foundation for future strategic partnerships, exchange and market-making programs, and contingencies, with varied vesting terms.
- **Public Sale:** available to the public at the Token Generation Event (TGE) — a larger tranche than typical genesis sales, widening initial distribution and circulating float.

9.4 Vesting Schedule

Every allocation is released through on-chain vesting contracts. TGE is the release at the Token Generation Event; cliff is the delay before linear release begins; linear is the streaming period.

Allocation	TGE release	Cliff	Linear	Total
Community & Ecosystem	5%	None	60 mo	60 mo
Foundation / Treasury	5%	12 mo	60 mo	72 mo
Validator Bootstrap	0%	None	48 mo	48 mo
Team & Core Contributors	0%	24 mo	36 mo	60 mo
Private & Institutional Round	5%	12 mo	24 mo	36 mo
Strategic Reserve	0–10%	12–24 mo	18–30 mo	30–48 mo
Public Sale	25%	None	6 mo	6 mo

9.5 Emissions & Inflation

QTOV emissions follow a hardcoded disinflationary curve. New issuance is paid exclusively to validators and delegators as staking rewards. The schedule is encoded into protocol consensus and **cannot be raised by governance** — governance can only lower it.

Year	Inflation rate	Year	Inflation rate
Year 1	5.0%	Year 6	2.2%
Year 2	4.2%	Year 7	1.9%
Year 3	3.6%	Year 8	1.6%
Year 4	3.1%	Year 9+	1.5% (terminal floor)
Year 5	2.6%		

ISSUANCE RULES

- The schedule is hardcoded into protocol consensus and applied automatically at each epoch.
- Governance can lower the inflation rate; it cannot raise it.
- 100% of new issuance is distributed to validators and delegators based on active stake.
- The terminal inflation floor is 1.5% from Year 9 onward.

9.6 Fee Distribution & Treasury

Quantova does not burn QTOV. Fees collected across the network are revenue: they are routed to the **Quantova treasury** (operated by Quantova Inc for the organization and ecosystem) and to **validators**. This funds ongoing development, security, and operations while rewarding the validators that produce blocks.

Source	Distribution
Transaction base fee	100% to the Quantova treasury.
Transaction priority fee	100% to the including validator.
Bridge protocol fee	100% to the Quantova treasury.
Bridge attestation fee	100% to attesting validators.
Slashed stake & forfeited bonds	100% to the Quantova treasury (see 10.5, 10.6).

> **No burn — fees are collected as revenue**
> QTOV is never destroyed by the protocol. Transaction and bridge fees flow to the Quantova treasury and to validators, and stake or bonds forfeited by bad actors are routed to the treasury. Supply changes only through the disinflationary issuance schedule in 8.5.

9.7 QTOV Voting Weight & Locking

Governance voting power is denominated in QTOV. Holders can vote with liquid QTOV, or lock QTOV for a chosen duration to receive a higher voting multiplier — aligning influence with the duration of committed stake. Locking affects voting weight only; the asset is always QTOV.

Lock duration	Voting multiplier
Liquid QTOV (no lock)	1.0×
1-month lock	1.0×
12-month lock	1.5×
24-month lock	2.5×

Locked QTOV is the basis for on-chain referendum quorums (Chapter 11). Higher-impact referenda require proposers to lock a larger QTOV stake for the governance cycle, so the holders who can move the network are those committing the most stake.

10. Staking & Validator Economics

Validators secure Quantova by staking QTOV and proposing blocks. Validator economics combine four revenue streams and are subject to slashing for misbehavior. The minimum self-stake is set high on purpose: a meaningful capital requirement raises the cost of spinning up many nodes to attempt a Sybil or stake-concentration attack, the same reasoning behind Ethereum's fixed validator deposit. The minimum is a protocol parameter that governance can adjust later, but only through a high-threshold referendum (10.4).

10.1 Roles

- **Validators** bond stake, register session keys, author blocks, and vote in finality.
- **Delegators / nominators** bond stake and back validators; their stake contributes to validator exposure and earns a share of rewards net of commission.

10.2 Validator Requirements

Minimum self-stake
$50,000 USD-equivalent in QTOV
Active validator cap
200 (target at maturity)
Uptime requirement
95% per epoch
Epoch length
24 hours
Unbonding period
14 days
Redelegation cooldown
7 days

> **Why the minimum is high**
> The $50,000-equivalent self-stake is a deliberate Sybil-resistance measure. Requiring substantial capital per validator makes it expensive to register many nodes in an attempt to concentrate stake or mount a coordinated attack, and it pairs with the 100%-slash penalties in 10.5 so that any such attempt risks the full deposit. The figure is a protocol parameter: governance can raise or lower it later, but only by passing a high-threshold critical referendum with a large QTOV stake behind it (10.4), never by an operator or admin decision.

10.3 Revenue Streams

Source	Calculation
Block issuance rewards	Proportional to active stake, paid from the emissions schedule (9.5).
Priority-fee share	40% of priority fees from proposed blocks.
MEV auction share	40% of MEV auction proceeds.
Bridge attestation fees	20% of bridge protocol fees.

COMMISSION CONSTRAINTS

Minimum commission
0%
Maximum commission
25%
Max change per epoch
1%
Max daily change
5%

10.4 Reward Formula & APR

```text
Validator Gross Reward (per epoch) =
    (Validator Stake / Total Stake) x Epoch Issuance
  + Priority Fees from Proposed Blocks x 0.40
  + MEV Auction Wins x 0.40
  + Bridge Attestations x Attestation Fee

Delegator Reward =
    Gross Reward x (Delegator Stake / Validator Total Stake) x (1 - Commission)

Validator Net Reward = Gross Reward - Total Delegator Reward
```

ESTIMATED YEAR-1 APR

Profile	Gross APR
Pure issuance (no fee revenue)	~5.0%
Average network usage	~6.5% – 8.0%
High-performance + MEV	~9.0% – 11.5%

10.5 Slashing

Slashing is the protocol's response to validator misbehavior. It is automatic, enforced in consensus, and deliberately severe: Quantova treats any attempt to threaten finality, the bridges, or the integrity of the chain as a total-loss offense. Safety faults — double-signing, bridge manipulation, and equivocation — forfeit the validator's **entire stake**. Even repeated liveness failures escalate quickly. Slashed stake is routed in full to the **Quantova treasury**, so value taken from a bad actor is returned to the organization and ecosystem rather than destroyed or paid to any individual. The economics are set so that attacking the chain is never rational: the expected loss always exceeds any possible gain.

Offense	What it is	Penalty	Destination
Downtime (minor)	Missing >5% of assigned blocks in an epoch (a liveness fault).	1% of stake + removal for the epoch	100% to Quantova treasury
Repeated / prolonged downtime	Persistent unavailability across multiple epochs.	10% of stake + removal	100% to Quantova treasury
Double-signing	Signing two blocks at the same height (a safety fault).	100% of stake + permanent ban	100% to Quantova treasury
Bridge attestation fraud / manipulation	Attesting to a cross-chain event that did not occur, or manipulating a bridge.	100% of stake + permanent ban	100% to Quantova treasury
Equivocation / long-range attack	Coordinated attempt to rewrite finalized history.	100% of stake + permanent ban	100% to Quantova treasury
Coordinated / cartel attack	Multiple validators colluding to break safety.	100% of stake of every participant + permanent ban	100% to Quantova treasury

Delegators are slashed in proportion to the stake they delegated to an offending validator, which aligns nominator incentives with the conduct of the validators they back: choosing a reliable, honest operator is itself a security decision. Any validator slashed for a safety fault is **permanently removed** from the active set and barred from re-registering with the same identity; a validator slashed for downtime must re-bond and re-qualify to return.

> **Zero tolerance for attacks on the chain**
> Quantova does not subsidize misbehavior, and it does not give attackers a second chance. Any safety fault — double-signing, equivocation, or bridge manipulation — forfeits 100% of stake and permanently bans the validator, with the entire amount routed to the Quantova treasury. Spam and frivolous governance also forfeit their QTOV bonds. The incentive is unambiguous: honest participation earns issuance and fees; any attempt to endanger the chain loses everything at stake.

10.6 Election

- **Sequential Phragmén** spreads delegator stake across validators for balanced backing and decentralization.
- **Election Provider Multi-Phase (EPM)** runs the solution pipeline: an unsigned off-chain phase, a deposit-rate-limited signed phase, and an on-chain fallback with tighter bounds.
- **Snapshot bounds** cap voters and targets to keep elections tractable and DoS-resistant.

Staking and nomination are runtime extrinsics: bond with `staking.bond`, declare intent with `staking.validate` or `staking.nominate`, and you are considered at the next era boundary.

11. Governance

Quantova governance adapts Ethereum's structured improvement-proposal model for a post-quantum chain with active bridge and security operations. It separates social-layer protocol evolution from bounded on-chain referenda, each with calibrated thresholds.

11.1 No Superuser: Governance Only

No sudo, no superuser, no single point of control
Quantova has **no** `pallet_sudo` **and no superuser or emergency keys**. There is no unilateral execution path in the runtime, and **no single entity — including the founding team — can change the network.** Even high-impact actions like minting or freezing run through bonded, supermajority on-chain referenda, never an admin key.

All Root-origin execution authority derives exclusively from on-chain governance. Pure token-weighted governance has known failure modes — whale capture, low turnout, flash-loan attacks — so Quantova combines an open improvement process (QIPs) with bounded QTOV-weighted referenda for decisions where direct token-holder input is appropriate, including the emergency response an active L1 with bridges requires.

11.2 Three Categories of Governance

Category	Mechanism	Voter pool
Protocol improvements	QIP process with reference implementation	Open community / validator consensus
Treasury & ecosystem	On-chain referendum with QTOV-weighted voting	All QTOV holders
Critical network operations	High-bond on-chain referendum	High-stake QTOV holders

Protocol improvements (QIPs) are drafted, discussed openly, implemented in reference clients, and adopted through validator consensus — the social layer. Token-weighted voting is reserved for specific on-chain decisions.

11.3 Standard Referendum

For routine treasury and ecosystem decisions:

Discussion period
7 days minimum
Proposal submission
On-chain, requires a QTOV bond
Voting period
7 days
Execution delay
48 hours, time-locked
Passing threshold
Simple majority (>50%)
Quorum requirement
4% of circulating QTOV

11.4 Critical Referendum

High-impact operations change the supply, the protocol itself, or a security-critical parameter: minting new QTOV, upgrading the runtime, and adjusting safety parameters such as the validator minimum self-stake (10.2) or the slashing rates (10.5). Each is a referendum class with thresholds scaled to its impact. The pattern mirrors how mature governance chains such as Polkadot and Ethereum gate consequential changes: the native asset is bonded and locked for the governance cycle to raise a referendum, and the larger the change, the larger the stake and supermajority required. There is no admin shortcut around these paths.

> **Changing the validator minimum is a high-stake vote**
> The $50,000 validator minimum self-stake is a security parameter, so it cannot be changed by an operator or a routine proposal. Raising or lowering it requires a critical referendum with a large bonded QTOV stake and a supermajority — the same class of vote used for minting and runtime upgrades below. This keeps a Sybil-resistance control from being weakened cheaply or quietly.

MINTING REFERENDUM (NEW QTOV INTO CIRCULATION)

A moderate-cost referendum — the QTOV equivalent of a routine on-chain parameter vote on other L1s. It can only ever *lower* the hardcoded emission schedule or release pre-allocated treasury QTOV; it cannot raise issuance above the schedule in 8.5.

Parameter	Value
Proposal bond	500,000 QTOV
Minimum proposer stake	5,000,000 QTOV, locked for the governance cycle (~90 days)
Discussion period	30 days minimum
Voting period	14 days
Execution delay	30 days
Passing threshold	80% supermajority
Quorum requirement	35% of circulating QTOV

EMERGENCY PROTOCOL-UPGRADE REFERENDUM

The maximum-cost path, used to change the runtime itself — the only mechanism that can alter consensus, slashing, or the cryptographic primitives. It carries the highest bond and locked-stake requirement on the network.

Parameter	Value
Proposal bond	1,000,000 QTOV
Minimum proposer stake	10,000,000 QTOV, locked for the governance cycle (~90 days)
Discussion period	14 days (reducible to 48h for an actively exploited vulnerability)
Voting period	14 days
Execution delay	30 days (reducible under an active exploit)
Passing threshold	80% supermajority
Quorum requirement	40% of circulating QTOV

11.5 Freeze & Asset Recovery

An active Layer-1 with cross-chain bridges needs a way to respond to a live theft or a bridge compromise without handing any party a unilateral capability. Quantova provides this as a high-threshold referendum, not an operator power. The scope is narrow and recorded on-chain: freeze assets that are demonstrably the proceeds of theft or an exploit so they cannot be bridged out, return them through the same process, and lift the freeze by a follow-up vote. The chain itself keeps producing and finalizing blocks throughout — a freeze targets specific assets, it does not stop the network.

ADDRESS FREEZE / ASSET-RECOVERY REFERENDUM

Used to freeze stolen or exploited funds before they leave the chain. Following the pattern of other top-tier Layer-1 emergency votes, the stake to propose is **high** but its lock is **short** — held only through the referendum and a brief execution window, then released — so the safeguard is costly to abuse yet does not tie up large stake for long. A freeze touches the property of a specific account, so it requires a broad supermajority; no single actor can act.

Parameter	Value
Proposal bond	250,000 QTOV
Minimum proposer stake	2,500,000 QTOV (high stake, short lock)
Stake lock	Held through the referendum + execution window only (~14–21 days), then released
Discussion period	Up to 21 days (reducible to 24h for an active, evidenced theft)
Voting period	14 days (reducible to 24h under an active exploit)
Execution delay	14 days (reducible to 0 for an evidenced, in-progress theft)
Passing threshold	75% supermajority
Quorum requirement	30% of circulating QTOV

A decentralized, accountable safeguard — not a backdoor
A freeze executes only through a bonded supermajority referendum with on-chain evidence and a public record; no key holder, validator, or Quantova Inc can invoke it alone. It exists to protect holders from theft, applies to the narrowest set of assets that resolves the incident, and is reversible by the same process. This keeps emergency response auditable and consistent with a decentralized, rule-governed protocol rather than discretionary control.

11.6 Bonds, Participation & Anti-Spam

To raise any on-chain referendum a proposer bonds QTOV. The bond is returned when a proposal reaches the engagement threshold and **forfeited to the Quantova treasury** if it does not — so spam and frivolous proposals fund the organization they waste, rather than costing nothing. This is the same native-asset-at-stake model used by other referendum-governed chains: participation has a price, and that price scales with impact.

Action	QTOV at stake to propose	Voter pool
Ecosystem grant proposal	1,000 QTOV bond	QTOV holders
Parameter adjustment (QIP)	10,000 QTOV bond	QTOV holders / validators
Treasury action (under $500K equiv.)	5,000 QTOV bond	QTOV holders
Treasury action (over $500K equiv.)	25,000 QTOV bond	QTOV holders
Mint QTOV (lower emission / release treasury)	500,000 QTOV bond + 5,000,000 QTOV locked ~90 days	QTOV holders
Freeze / recover stolen assets	250,000 QTOV bond + 2,500,000 QTOV locked (short)	high-stake QTOV holders
Emergency runtime upgrade (major change)	1,000,000 QTOV bond + 10,000,000 QTOV locked ~90 days	high-stake QTOV holders

Voting weight is denominated in QTOV (9.7): vote with liquid QTOV for 1.0× weight, or lock it for up to 2.5×. Routine votes need a simple majority and a 4% quorum; consequential votes need a 75–80% supermajority and a 30–40% quorum. The mint and emergency-upgrade paths lock the proposer's stake for the governance cycle, while the freeze path keeps the stake high but the lock short, so urgent asset recovery is not encumbered.

No spam, no bad actors
Every privileged action costs QTOV at stake, and QTOV forfeited by spammers, failed proposers, or slashed validators is routed in full to the Quantova treasury (10.5). The economic message is explicit: the network does not subsidize spam or bad actors — it bills them, and the proceeds accrue to the Quantova treasury.

12. Cross-Chain Bridges

Quantova bridges to external ecosystems on one principle: do not trust off-chain actors with correctness. Off-chain relayers provide liveness; all safety-enforcing verification happens inside the Quantova runtime consensus. Bridges are permissionless, with no protocol-level caps.

12.1 Architecture: A Pallet, Not a Separate Layer

The bridge is a **runtime pallet** — like the naming service — paired with an **off-chain relayer that uses zero-knowledge technology** to move data between chains. There is no separate "relay layer" or sidechain, and verification is not performed outside the chain.

> **The QVM is not involved**
> The bridge runs on the **native Quantova runtime, not the QVM**. The QVM is the smart-contract virtual machine and has no role in bridging. Light-client checks, Merkle/storage-proof validation, and zero-knowledge verification are performed by the bridge pallet inside consensus — this security-critical work must live in the runtime.

12.2 Deposits (foreign → Quantova)

1. **Observe** the deposit on the source chain (a user locks tokens in the bridge contract).
2. **Prove source-chain finality** — on BSC via Parlia-style finality attestations; on Ethereum via beacon-chain sync-committee updates establishing finalized headers.
3. **Prove inclusion in state** with EIP-1186 Merkle-Patricia proofs anchored to the finalized header's `state_root` (`accountProof` + `storageProof`).
4. **Credit on Quantova with strict ordering** — enforce nonce ordering (and hash-chaining where applicable) so each deposit is processed exactly once; reject replays.

All of the above runs inside the Quantova runtime — not on the QVM, and not delegated to an off-chain actor.

12.3 Withdrawals (Quantova → foreign)

1. A user initiates a withdrawal on Quantova; the outward transfer is recorded in on-chain state.
2. A **zero-knowledge proof** is generated attesting to both *finality* (the recording block is finalized, not merely produced) and the *post-quantum state transitions* that approve the withdrawal (recipient, amount, nonce).
3. The **destination-chain bridge contract verifies the proof on-chain**, then enforces nonce ordering / replay protection (and batching invariants) before releasing or minting.

12.4 TRON: Trusted-Relayer Integration

> **TRON is multisig-based**
> TRON does not expose the account/state-root commitments needed for trust-minimized Merkle-proof verification, so its path currently uses a **centralized, multisig-based trusted relayer**. Quantova therefore does not claim "no custodial intermediaries" as a blanket property: the Ethereum and BSC paths are trust-minimized via on-chain verification; the TRON path relies on the trusted multisig relayer for correctness.

12.5 Fees & Settlement Timing

Bridge fee composition and the asset-tier table are in 7.4. Estimated settlement times:

Direction	Estimated time
Outbound to EVM L1	3–8 minutes
Outbound to non-EVM L1	8–15 minutes
Inbound from EVM L1	5–12 minutes
Inbound from non-EVM L1	10–20 minutes

BRIDGE DEVELOPER ENDPOINTS

```bash
POST /v1/bridge/quote        # fee breakdown for a proposed bridge
POST /v1/bridge/initiate     # initiate an outbound bridge transaction
GET  /v1/bridge/status/:tx   # returns bridge progress
POST /v1/bridge/claim        # claim inbound bridged assets
```

Bridges are secured by validator attestation with slashing for fraudulent attestations (10.5), post-quantum signature verification on Quantova-side proofs, and ongoing third-party audits. There is no centralized authority over bridge operations on the trust-minimized paths.

13. Quantova Naming Service (QNS)

QNS is Quantova's on-chain naming registry, implemented as a runtime pallet. It maps human-readable names to ownership and resolver records and provides reverse lookup so an account can present a primary name.

- **Commit-reveal registration** — two steps (commit, then reveal) so the plaintext label is not exposed until the claim is finalized, resisting label sniping.
- **Name normalization** — IDNA / UTS-46 to lowercase ASCII (including Punycode) before hashing, mitigating Unicode homograph attacks.
- **Resolver records** — a `Domains` registry stores ownership, permissions, pricing, expiry, and resolver records; `Roots` holds governance-managed root metadata.
- **Reverse resolution** — a `ReverseMap` maps an account to a single primary name (one-to-one).

Domains are keyed by an EIP-137-style namehash (H256); subdomains derive deterministically via `child_hash(parent, label)`. Front-running resistance comes from commit-reveal binding the commitment to `(who, label, parent, salt)` plus an expiry window. Registration and renewal fees are paid in QTOV. Contracts can resolve names through the QNS precompile (14.2).

JAVASCRIPT

```javascript
// resolve a QNS name to an address, and reverse-resolve an address to its primary name
const addr = await api.query.qns.resolve('alice.q');          // -> Q...
const name = await api.query.qns.reverse(ALICE_Q_ADDRESS);    // -> 'alice.q'
console.log(addr.toString(), name.toString());
```

14. Smart Contracts on the QVM

The Quantova Virtual Machine (QVM) is the smart-contract execution environment. Contracts are written in Solidity, compiled to QVM bytecode, and deployed as QVM transactions. The accounts that own and call contracts are post-quantum, so contract authorization inherits quantum security.

14.1 The QVM Execution Model

- Contracts execute deterministically as part of the block state-transition function; storage commits to the SHA3-256 state root.
- Execution is metered in **QGAS**, priced in QTOV (Chapter 8).
- Contract addresses are the same 20-byte H160 width as accounts and use the same Q-format scheme — in `q_` methods, in QVM Solidity, and on explorers.
- Deploy with a QVM transaction where `to = null`; call with `q_sendRawTransaction` / read with `q_call`.

14.2 Q Primitives: Native Precompiles

Quantova exposes native runtime capabilities to Solidity through **QVM precompiles** — reserved system contracts callable with `staticcall`. Like every account on Quantova, a precompile is identified by a **Q-format address**, and the QVM Solidity toolchain accepts Q-format address literals directly. They let contracts use post-quantum primitives without reimplementing them.

Q primitive	Purpose	Precompile address
SHA3-256	Canonical Quantova hashing	`Qa25f20eb7833f473574f1eadda691bf6c4133b`
Dilithium verify	Verify an ML-DSA signature on-chain	`Q0710f6ad4607006bcebc926f9887c34816d97a`
Falcon verify	Verify an FN-DSA signature on-chain	`Q2765d43b5cc993762880321502b2385da7e1bb`
SPHINCS+ verify	Verify an SLH-DSA signature on-chain	`Qc148806bf7cead0b754f2cd3345ca9c3d60f48`
QTE submit	Submit a ciphertext into the encrypted lane	`Qbabb37461b793748e239608abf906a6fc115aa`
QNS resolve	Resolve a name to an address from a contract	`Q59dabf31c0b9496c9015a98b0b0c72249e9f0f`

Why addresses are Q and hashes are 0x

Quantova uses two distinct notations, and they are **not** the same as Ethereum's. **Addresses use the Q scheme:** every account, contract, and precompile is identified by a Q-prefixed address derived from a public key with the `0x40` Q marker (Chapter 5) — this is Quantova's native account format. **Hashes use `0x` hex:** a hash (a SHA3-256 digest, a block hash, a transaction hash) is raw 32-byte data, not an account, and `0x` is simply the universal hexadecimal way to write a byte array. The `0x` on a hash is notation for bytes — it does not make the value an address, and it is not inherited from Ethereum. So in code you will see Q-format *addresses* and `0x` *hashes* side by side, by design.

The addresses below are reserved demonstration values; confirm the exact reserved precompile addresses against your target runtime metadata. A reusable Solidity library wraps the precompiles:

SOLIDITY

```solidity
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.24;

/// @title QPrimitives - access Quantova post-quantum precompiles from Solidity
/// @dev Precompiles are reserved system contracts addressed by Q-format addresses
///      (Quantova's native account scheme). The QVM toolchain accepts Q literals.
///      Hashes (bytes32) are written in 0x hex: a hash is raw bytes, not an account.
library QPrimitives {
    address internal constant SHA3_256         = Qa25f20eb7833f473574f1eadda691bf6c4133b;
    address internal constant DILITHIUM_VERIFY = Q0710f6ad4607006bcebc926f9887c34816d97a;
    address internal constant FALCON_VERIFY    = Q2765d43b5cc993762880321502b2385da7e1bb;
    address internal constant SPHINCS_VERIFY   = Qc148806bf7cead0b754f2cd3345ca9c3d60f48;
    address internal constant QTE_SUBMIT       = Qbabb37461b793748e239608abf906a6fc115aa;

    /// @notice SHA3-256 (the chain's canonical hash) of arbitrary bytes.
    /// @return out a bytes32 digest, written in 0x hex (raw bytes, not an address).
    function sha3_256(bytes memory data) internal view returns (bytes32 out) {
        (bool ok, bytes memory r) = SHA3_256.staticcall(data);
        require(ok && r.length == 32, "QP: sha3 failed");
        out = abi.decode(r, (bytes32));
    }
    // ... pqVerify continues below
}
```

And the signature-verification helper in the same library — it calls whichever post-quantum verifier precompile you pass in:

SOLIDITY

```
library QPrimitives {                       // (continued)
    /// @notice Verify a post-quantum signature with the chosen verifier.
    function pqVerify(
        address verifier, bytes memory publicKey,
        bytes memory message, bytes memory signature
    ) internal view returns (bool valid) {
        (bool ok, bytes memory r) =
            verifier.staticcall(abi.encode(publicKey, message, signature));
        require(ok, "QP: verify call failed");
        valid = abi.decode(r, (bool));
    }
}
```

Calling a Q primitive: step by step

Each primitive is invoked with `staticcall` to its Q-format address and an ABI-encoded argument. The pattern is the same for all of them:

1. **Import the library:** `import {QPrimitives} from "./QPrimitives.sol";` in your contract.
2. **Pick the primitive** by its constant (e.g. `QPrimitives.FALCON_VERIFY`) — these are the Q-format addresses listed in 13.2.
3. **ABI-encode the inputs** the primitive expects (for a verifier: public key, message, signature).
4. **staticcall** the address — it is read-only, so it cannot change state and is cheap.
5. **Check the success flag** and `abi.decode` the returned bytes into the expected type.

What each primitive takes and returns:

Primitive	Input (ABI-encoded)	Returns
`SHA3_256`	raw `bytes` to hash	`bytes32` digest
`DILITHIUM_VERIFY`	`(publicKey, message, signature)`	`bool` valid
`FALCON_VERIFY`	`(publicKey, message, signature)`	`bool` valid
`SPHINCS_VERIFY`	`(publicKey, message, signature)`	`bool` valid
`QTE_SUBMIT`	`(ciphertext, recipientTag)`	`bytes32` submission id
`QNS_RESOLVE`	`(bytes32 nameHash)`	`address` (Q-format) owner

A minimal contract that hashes data with the library — the hashing helper:

SOLIDITY

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.24;

import {QPrimitives} from "./QPrimitives.sol";

contract UsesPrimitives {
    using QPrimitives for bytes;

    /// Hash arbitrary data with the chain's canonical SHA3-256 precompile.
    function hash(bytes calldata data) external view returns (bytes32) {
        return QPrimitives.sha3_256(data);          // -> bytes32 digest (0x hex)
    }
    // ... verification below
}
```

... and verifying a Falcon signature on-chain via the FALCON_VERIFY precompile:

SOLIDITY

```
contract UsesPrimitives {                       // (continued)
    function checkFalcon(
        bytes calldata publicKey,
        bytes calldata message,
        bytes calldata signature
    ) external view returns (bool ok) {
        ok = QPrimitives.pqVerify(
            QPrimitives.FALCON_VERIFY,              // Q-format precompile address
            publicKey, message, signature
        );
    }
}
```

Addresses are Q, the digest is 0x

In the code above the precompile address is a Q-format value (`QPrimitives.FALCON_VERIFY`), while the returned digest is a `bytes32` written in `0x` hex. That is the rule throughout: **accounts and contracts are Q; hashes and raw bytes are 0x** (14.2, 15.1).

14.3 Token Contract Example

A standard fungible token on the QVM with an optional *quantum permit*: a holder can authorize a spender by signing an approval off-chain with their post-quantum key, verified through the Falcon precompile — a gasless approval secured by a NIST PQ signature. Part 1: storage, transfers, and approvals.

SOLIDITY

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.24;
import {QPrimitives} from "./QPrimitives.sol";

/// @title QToken - fungible token with a post-quantum permit
contract QToken {
    string  public name     = "Example Quantova Token";
    string  public symbol   = "EXQ";
    uint8   public constant decimals = 18;
    uint256 public totalSupply;
    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256)) public allowance;
    mapping(address => uint256) public permitNonce;          // replay protection
    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    constructor(uint256 initialSupply) {
        totalSupply = initialSupply;
        balanceOf[msg.sender] = initialSupply;
        emit Transfer(address(0), msg.sender, initialSupply);
    }
    // ... transfer, approve, and the PQ permit follow
}
```

The basic transfer and approval entry points:

SOLIDITY

```
contract QToken {                               // (continued)
    function transfer(address to, uint256 value) external returns (bool) {
        _move(msg.sender, to, value);
        return true;
    }

    function approve(address spender, uint256 value) external returns (bool) {
        allowance[msg.sender][spender] = value;
        emit Approval(msg.sender, spender, value);
        return true;
    }
    // ... transferFrom + permitFalcon + _move continue in Part 2
}
```

Part 2: delegated transfer, the post-quantum permit, and the internal transfer helper. The owner signs `sha3_256(owner, spender, value, nonce, chainid)` off-chain with their Falcon key; the contract verifies it through the Falcon precompile.

SOLIDITY

```
contract QToken {                               // (continued)
    function transferFrom(address from, address to, uint256 value) external returns (bool) {
        uint256 a = allowance[from][msg.sender];
        require(a >= value, "QToken: allowance");
        if (a != type(uint256).max) allowance[from][msg.sender] = a - value;
        _move(from, to, value);
        return true;
    }

    /// @notice Approve via an off-chain Falcon (post-quantum) signature.
    function permitFalcon(
        address owner, address spender, uint256 value,
        bytes calldata ownerPublicKey, bytes calldata signature
    ) external {
        bytes32 digest = QPrimitives.sha3_256(
            abi.encode(owner, spender, value, permitNonce[owner], block.chainid)
        );
        require(
            QPrimitives.pqVerify(
                QPrimitives.FALCON_VERIFY, ownerPublicKey, abi.encodePacked(digest), signature),
            "QToken: bad PQ signature"
        );
        permitNonce[owner]++;
        allowance[owner][spender] = value;
        emit Approval(owner, spender, value);
    }
    // ... internal _move helper below
}
```

And the internal balance-moving helper shared by all transfer paths:

SOLIDITY

```
contract QToken {                               // (continued)
    function _move(address from, address to, uint256 value) internal {
        require(balanceOf[from] >= value, "QToken: balance");
        balanceOf[from] -= value;
        balanceOf[to]   += value;
        emit Transfer(from, to, value);
    }
}
```

14.4 Quantova Variables & Structures

QVM Solidity uses the standard value model with Quantova-specific conventions:

- **Addresses** are 20-byte H160 (Solidity `address`) — the same width as the Q-prefixed account address; the Q form is a display convention.
- **Post-quantum keys and signatures** are variable-length, so they are passed and stored as `bytes` — never fixed-size types.
- **Hashes** use the SHA3-256 precompile; store digests as `bytes32`.
- **Value** is QTOV in base units (18 decimals), held in `uint256`.

A registry demonstrating structs, mappings, dynamic arrays, enums, events, and a PQ check. Part 1 — storage layout:

SOLIDITY

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.24;

import {QPrimitives} from "./QPrimitives.sol";

contract QRegistry {
    enum Scheme { SPHINCS, FALCON, DILITHIUM }   // matches QSignature variants 0,1,2

    struct Account {
        address owner;          // 20-byte H160 (canonical Q-address holder)
        Scheme  scheme;         // which PQ scheme secures this entry
        bytes   publicKey;      // variable-length PQ public key -> bytes
        uint256 registeredAt;   // block.number
        bool    active;
    }

    mapping(address => Account) public accounts;   // address  -> record
    mapping(bytes32 => address) public byKeyHash;  // sha3(pk) -> owner
    address[] public members;                      // dynamic array

    event Registered(address indexed owner, Scheme scheme, bytes32 keyHash);
    // ... functions in Part 2
}
```

Part 2 — registration with an on-chain post-quantum signature check, plus helpers:

SOLIDITY

```
contract QRegistry {                          // (continued)
    function register(Scheme scheme, bytes calldata publicKey, bytes calldata signature) external {
        bytes32 keyHash  = QPrimitives.sha3_256(publicKey);
        address verifier = _verifierFor(scheme);
        bytes memory challenge = abi.encodePacked(msg.sender, keyHash, block.chainid);

        require(QPrimitives.pqVerify(verifier, publicKey, challenge, signature), "bad sig");
        require(!accounts[msg.sender].active, "exists");

        accounts[msg.sender] = Account(msg.sender, scheme, publicKey, block.number, true);
        byKeyHash[keyHash] = msg.sender;
        members.push(msg.sender);
        emit Registered(msg.sender, scheme, keyHash);
    }

    function memberCount() external view returns (uint256) { return members.length; }

    function _verifierFor(Scheme s) internal pure returns (address) {
        if (s == Scheme.FALCON)    return QPrimitives.FALCON_VERIFY;
        if (s == Scheme.DILITHIUM) return QPrimitives.DILITHIUM_VERIFY;
        return QPrimitives.SPHINCS_VERIFY;
    }
}
```

14.5 Deploy with Qdock.io & Qmask.io

Qdock.io is the Quantova IDE for writing, compiling, and deploying Solidity to the QVM. It signs deployments through **Qmask.io**, the post-quantum wallet web extension, so the deploying account is a real Q-prefixed post-quantum account.

1. Open **Qdock.io** and create or paste your Solidity sources (e.g. `QToken.sol`, `QPrimitives.sol`).
2. Compile to QVM bytecode in the IDE and review the QGAS estimate.
3. Connect **Qmask.io** and select the deploying Q-account (Dilithium / Falcon / SPHINCS+).
4. Deploy: Qdock.io builds the `to = null` QVM transaction; Qmask.io signs it with your post-quantum key and submits via `q_sendRawTransaction`.
5. Qdock.io shows the deployed contract address (canonical Q form) and lets you call methods interactively.

Programmatic deployment with `qweb3.js` (CI or scripts):

JAVASCRIPT

```javascript
import { QWeb3 } from 'qweb3.js';
const q = new QWeb3('https://mainnet.quantova.io');

const deployTx = await q.qvm.buildTransaction({ to: null, data: '0x'+bytecode, gas: GAS_LIMIT });
const receipt  = await q.qvm.sendSigned(deployTx, account);   // PQ-signed via Qmask.io / keyring
console.log('contract address:', receipt.contractAddress);

// call a view method through q_call
const out = await q.qvm.call({
  to:   receipt.contractAddress,
  data: q.abi.encode('balanceOf', [account.address]),
});
console.log('balance:', q.abi.decode('uint256', out).toString());
```

15. Developer SDK & Code Examples

The Quantova post-quantum SDK is published under the @quantova npm scope. The QVM `q_` namespace is plain JSON-RPC, so any language can talk to the chain. This chapter is a working cookbook in JavaScript, Python, and Solidity.

15.1 Packages

Package	Purpose
`@quantova/api`	High-level client: chain queries, transactions, RPC. Exposes `ApiPromise`, `WsProvider`.
`@quantova/keyring`	Account/key management for all signature schemes.
`@quantova/util`	Encoding utilities: hex ↔ bytes, BN/bigint, SCALE helpers.
`@quantova/util-crypto`	Crypto & hashing: SHA3 plus Dilithium / Falcon / SPHINCS+ primitives.
`@quantova/falcon-wasm`	Quantum-signature WASM bindings used by `util-crypto`.
`@quantova/types`	Runtime type definitions.
`@quantova/rpc-provider`	Transport providers (`WsProvider`, HTTP).
`qweb3.js`	QVM contract client: transaction building, ABI encoding, QGAS, PQ signing.

BASH

```bash
npm install @quantova/api @quantova/keyring @quantova/util @quantova/util-crypto qweb3.js
```

15.2 JavaScript: API Calls (runtime state & transactions)

Use `@quantova/api` for runtime storage queries, transactions, and subscriptions. Storage-map queries return arrays you can iterate. First, reading state:

JAVASCRIPT

```javascript
import { ApiPromise, WsProvider } from '@quantova/api';
const api = await ApiPromise.create({ provider: new WsProvider('wss://mainnet.quantova.io') });

// --- single storage item (a variable) ---
const { data, nonce } = await api.query.system.account(ALICE_Q_ADDRESS);
console.log('free', data.free.toString(), 'nonce', nonce.toNumber());

// --- storage map -> array of [key, value] entries ---
const entries = await api.query.staking.validators.entries();
const validators = entries.map(([key, prefs]) => ({
  address: key.args[0].toString(),       // canonical Q-address
  commission: prefs.commission.toString(),
}));
console.log('validator count:', validators.length);
```

... then sending a transaction and subscribing to finalized heads:

JAVASCRIPT

```javascript
// --- transaction: Alice sends QTOV to Bob ---
const tx = api.tx.balances.transferKeepAlive(BOB_Q_ADDRESS, 1_000_000_000_000_000_000n);
const unsub = await tx.signAndSend(alice, ({ status }) => {
  if (status.isFinalized) { console.log('finalized', status.asFinalized.toHex()); unsub(); }
});

// --- subscribe to finalized heads ---
await api.rpc.chain.subscribeFinalizedHeads((h) => console.log('finalized #', h.number.toNumber()));
```

15.3 JavaScript: RPC Calls (q_ namespace)

The QVM `q_` methods are plain JSON-RPC. Account addresses are passed as canonical **Q-format strings**; `params` is a JSON array in the documented order. Balances and block numbers come back as `QUANTITY` hex you convert with `parseInt(x, 16)` or `BigInt(x)`.

JAVASCRIPT

```javascript
const RPC = 'https://mainnet.quantova.io';

async function q(method, params = []) {
  const res = await fetch(RPC, {
    method: 'POST',
    headers: { 'content-type': 'application/json' },
    body: JSON.stringify({ jsonrpc: '2.0', id: 1, method, params }),
  });
  const { result, error } = await res.json();
  if (error) throw new Error(error.message);
  return result;
}

const ADDRESS = 'Q0b22901ba019b7f48327a891d21a9af2547493';   // Bob, canonical Q-format
const height  = await q('q_blockNumber');                    // "0x15536ee"
const balance = await q('q_getBalance', [ADDRESS, height]);  // params is an array
const fees    = await q('q_feeHistory', [4, height, [25, 50, 75]]);
console.log({ height, balance, baseFees: fees.baseFeePerGas });
```

15.4 JavaScript: Primitives, Variables & Arrays

`@quantova/util-crypto` exposes the post-quantum primitives and SHA3 hashing; `@quantova/util` handles variable encoding and arrays. First, hashing and signing:

JAVASCRIPT

```javascript
import { cryptoWaitReady, sha3AsU8a, pqGenerate, pqSign, pqVerify } from '@quantova/util-crypto';
import { u8aToHex, hexToU8a, stringToU8a } from '@quantova/util';

await cryptoWaitReady();

// --- hashing primitive (SHA3-256) ---
const digest = sha3AsU8a('hello quantova', 256);          // Uint8Array(32)
console.log('digest', u8aToHex(digest));

// --- key generation + sign + verify for a chosen scheme ---
const { publicKey, secretKey } = pqGenerate('falcon');    // 'dilithium'|'falcon'|'sphincsp'
const msg = stringToU8a('transfer 10 QTOV to Bob');
const sig = pqSign(msg, secretKey, 'falcon');
console.log('valid?', pqVerify(msg, sig, publicKey, 'falcon'));
```

Then variable encoding and batching across an array of Q-addresses:

JAVASCRIPT

```javascript
// --- variables: hex <-> bytes ---
const bytes = hexToU8a('0x405abc');
const back  = u8aToHex(bytes);

// --- arrays: batch over many addresses ---
const addresses = [ALICE_Q_ADDRESS, BOB_Q_ADDRESS, CAROL_Q_ADDRESS];
const balances = await Promise.all(addresses.map(async (a) => {
  const { data } = await api.query.system.account(a);
  return { address: a, free: data.free.toBigInt() };
}));
balances.forEach((b) => console.log(b.address, b.free.toString()));
```

15.5 Python: RPC, Data & Batching

The `q_` namespace is JSON-RPC over HTTP, so Python needs only `requests`. A small client with a batch (array) request:

PYTHON

```python
import requests

class Quantova:
    def __init__(self, url="https://mainnet.quantova.io"):
        self.url = url

    def call(self, method, params=None):
        payload = {"jsonrpc": "2.0", "id": 1, "method": method, "params": params or []}
        r = requests.post(self.url, json=payload, timeout=10)
        r.raise_for_status()
        out = r.json()
        if "error" in out:
            raise RuntimeError(out["error"])
        return out["result"]

    def batch(self, calls):
        # calls: list of (method, params) -> a single JSON-RPC batch request
        payload = [
            {"jsonrpc": "2.0", "id": i, "method": m, "params": p or []}
            for i, (m, p) in enumerate(calls)
        ]
        return requests.post(self.url, json=payload, timeout=10).json()
```

Using the client — reading a balance and batching across several Q-addresses:

PYTHON

```python
q = Quantova()
height  = int(q.call("q_blockNumber"), 16)                  # hex QUANTITY -> int
ADDRESS = "Q0b22901ba019b7f48327a891d21a9af2547493"       # Bob (canonical Q-format)
balance = int(q.call("q_getBalance", [ADDRESS, hex(height)]), 16)

# array of accounts -> list of balances
wallets  = [ALICE_Q, BOB_Q, CAROL_Q]    # canonical Q-format address strings
balances = [int(q.call("q_getBalance", [w, hex(height)]), 16) for w in wallets]

fees = q.call("q_feeHistory", [4, hex(height), [25, 50, 75]])
print("height:", height, "| balance:", balance)
print("base fees:", fees["baseFeePerGas"])
```

15.6 Solidity: Variables & Structures (quick reference)

A compact reference for QVM contract data types — declarations first. Full contracts are in 14.3–14.4; deployment via Qdock.io / Qmask.io is in 14.5.

SOLIDITY

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity ^0.8.24;

import {QPrimitives} from "./QPrimitives.sol";

contract QTypes {
    address public owner;          // 20-byte H160 (a Q-address holder)
    uint256 public amount;         // QTOV base units (18 decimals)
    bytes32 public stateHash;      // SHA3-256 digest
    bool    public active;

    bytes public pubKey;           // variable-length PQ public key
    bytes public signature;        // variable-length PQ signature

    enum Scheme { SPHINCS, FALCON, DILITHIUM }   // = QSignature variants 0,1,2
    Scheme public scheme;

    struct Entry { address who; uint256 value; bytes32 keyHash; }
    mapping(address => Entry) public entries;                       // keyed map
    mapping(address => mapping(address => uint256)) public allow;   // nested map
    address[] public list;                                          // dynamic array
    uint256[3] public buckets;                                      // fixed array

    event Stored(address indexed who, bytes32 keyHash, uint256 value);
    // ... behavior below
}
```

... and the behavior, using a Q primitive to hash a key:

SOLIDITY

```
contract QTypes {                              // (continued)
    function store(uint256 value) external {
        bytes32 kh = QPrimitives.sha3_256(pubKey);                  // Q primitive
        entries[msg.sender] = Entry(msg.sender, value, kh);
        list.push(msg.sender);
        emit Stored(msg.sender, kh, value);
    }

    function count() external view returns (uint256) { return list.length; }
}
```

16. JSON-RPC Reference

Quantova exposes two RPC surfaces. Runtime/chain state uses the Substrate namespaces (`chain_*`, `state_*`, `system_*`, `author_*`) via `@quantova/api`. QVM execution uses the Quantova `q_` **namespace** — the branded entry point for all virtual-machine interaction.

16.1 Address Format & Encoding

Quantova account addresses are 20 bytes (H160), derived as the first 20 bytes of `SHA3-256(publicKey)` with the fixed `0x40` Q marker as byte 0 (Chapter 5). Their canonical form is **Q-prefixed**, and that canonical form is what the `q_` namespace uses on the wire: account-address parameters and results are passed as **Q-format strings**, the same identifier users see in Qmask.io and explorers. This mirrors string-address chains such as TRON (`T…`) and keeps one address representation across wallet, explorer, and RPC.

Field type	Representation	Example
Account / contract address	Canonical **Q**-format string	`Qa0018bcd09ed8fc81b323331950a89541d2416`
Hash (block, transaction, digest)	`0x` 32-byte SHA3 hex (never Q)	`0x9f3c…1a3`
Quantity (balance, gas, block no.)	`0x` hex integer	`0x7b2f1c8a0d9e4000`

Q for addresses, 0x for hashes and quantities

Account, contract, and precompile addresses are always written in the Q scheme — in the wallet, the explorer, the `q_` RPC, and in QVM Solidity (the toolchain accepts Q-format address literals). The `0x` hex notation is reserved for values that are not accounts: hashes (32-byte SHA3 digests, block and transaction hashes) and numeric quantities. A `0x` value is raw bytes or a number — it is never an address, and the convention is standard hexadecimal, not inherited from Ethereum.

16.2 Data Types

Type	Encoding
`ADDRESS`	Canonical **Q**-format account/contract address string (e.g. `Qa0018bcd…2416`).
`QUANTITY`	Integer, hex-encoded, `0x`-prefixed, **no leading zeroes** (e.g. `0x5208`). Used for balances, gas, block numbers.
`DATA`	Arbitrary byte array, hex-encoded, `0x`-prefixed (calldata, bytecode, signatures).
Block hash	A **32-byte SHA3 hash** (`0x`-prefixed). **Not** Q-prefixed — a hash is not an account.
Transaction hash	A **32-byte SHA3 hash** (`0x`-prefixed). **Not** Q-prefixed.

Block parameter

Methods that take a block reference accept a block number (`QUANTITY`) or a 32-byte block hash. Named tags (`"latest"`, `"earliest"`, `"pending"`, `"safe"`, `"finalized"`) are **not implemented** in the current version — do not pass them.

Only *account and contract addresses* carry the Q prefix. Balances, gas, and block numbers are `QUANTITY` hex; block and transaction hashes are 32-byte SHA3 values with no Q prefix — a hash identifies data, not an account.

16.3 q_ Methods

Every method below is a POST to the HTTP endpoint with a JSON body `{jsonrpc, id, method, params}`. `params` is a JSON array in the documented order. Account addresses are passed as **Q-format strings**; numeric results (balances, gas) come back as `QUANTITY` hex you convert to a number. The most-used call is `q_getBalance` — here it is end to end with `curl`:

```
zsh — q_getBalance via JSON-RPC

~/quantova-app curl -s https://mainnet.quantova.io \
  -H 'content-type: application/json' \
  -d '{"jsonrpc":"2.0","id":1,"method":"q_getBalance",
      "params":["Q0b22901ba019b7f48327a891d21a9af2547493","0x15536ee"]}'

{"jsonrpc":"2.0","id":1,"result":"0x7b2f1c8a0d9e4000"}
# result is a QUANTITY (hex base units): 0x7b2f1c8a0d9e4000 / 1e18 = 8.876 QTOV
~/quantova-app
```

Reading an account balance with q_getBalance. params = [Q-address, block]; the result is QUANTITY hex.

How to read it. The request sends two params: the Q-format account address and a block reference (a block-number `QUANTITY` — named tags like `"latest"` are not implemented, see 16.2). The response `result` is a `QUANTITY` in base units: `0x7b2f1c8a0d9e4000` = `8 876 000 000 000 000 000` ÷ `10^18` = **8.876 QTOV**. Convert with `parseInt(result, 16)` (JavaScript) or `int(result, 16)` (Python). The same request from JavaScript and Python is in 15.3 and 14.5.

State & account methods

Method	Params	What it does
`q_getBalance`	address, block	Returns the QTOV balance of an account in base units (QUANTITY). Use this to show balances and to check funds before sending.
`q_getTransactionCount`	address, block	Returns the account nonce — the number of transactions sent. Use it to set the `nonce` of the next transaction.
`q_getCode`	address, block	Returns the deployed QVM bytecode at an address, or `0x` if the address is not a contract.
`q_getStorageAt`	address, slot, block	Returns the raw 32-byte value in a contract storage slot. For reading internal state an ABI getter does not expose.
`q_call`	call object, block	Executes a read-only contract call and returns its return data, without creating a transaction or spending QGAS.
`q_estimateGas`	call object	Returns the QGAS a call or deployment would consume, so you can set a safe `gas` limit.

Block methods

Method	Params	What it does
`q_blockNumber`	—	Returns the height of the most recent block (QUANTITY). The starting point for most reads.
`q_getBlockByNumber`	number, fullTx	Returns a block by height, with either transaction hashes or full transaction objects.
`q_getBlockByHash`	hash, fullTx	Returns a block by its 32-byte SHA3 hash.
`q_getBlockTransactionCountByHash`	hash	Returns how many transactions a block contains.

Transaction methods

Method	Params	What it does
`q_sendRawTransaction`	signed tx (DATA)	Submits an already-signed, serialized QVM transaction. The standard path: sign locally with a post-quantum key, then broadcast. Returns the 32-byte transaction hash.
`q_sendTransaction`	tx object	Submits a transaction from an account the node holds (operator/dev use). Returns the transaction hash.
`q_getTransactionReceipt`	tx hash	Returns the receipt — status, QGAS used, logs, and (for deployments) the new contract address. Poll this to confirm a transaction.

Fee & gas methods

Method	Params	What it does
`q_gasPrice`	—	Returns the current gas price (QUANTITY) for building a transaction.
`q_maxPriorityFeePerGas`	—	Returns a suggested priority fee (tip) for timely inclusion (8.1).
`q_feeHistory`	blockCount, newest, percentiles	Returns recent base fees, priority-fee percentiles, and gas-used ratios — the inputs to a fee-estimation model.

JSON

```
// q_feeHistory(4, "0x1554ec6", [25, 50, 75])
{
  "jsonrpc": "2.0", "id": 1,
  "result": {
    "oldestBlock": "0x1554ec2",
    "reward": [["0x4b571c0","0x2f23c46c"], ["0x396c1b9","0x4d579d50"]],
    "baseFeePerGas": ["0x3af6c9f1","0x3b19496d","0x36647614","0x302c838b"],
    "gasUsedRatio": [0.5091416944, 0.1814587222, 0.0426904106]
  }
}
```

Node & network methods

Method	Params	What it does
`q_chainId`	—	Returns the chain id, used in transaction signing to prevent cross-chain replay.
`q_accounts`	—	Returns the accounts the client controls, as Q-format address strings (empty array if none).
`q_syncing`	—	Returns a sync-progress object while catching up, or `false` when fully synced.
`q_listening`	—	Returns whether the node is listening for peers.
`q_net_version`	—	Returns the network id.
`q_web3_clientVersion`	—	Returns the node client version string.

The q_call / q_estimateGas call object

Field	Type	Required	Description
`from`	ADDRESS	Yes	Q-format address the call is sent from.
`to`	ADDRESS	Yes*	Q-format QVM contract address. `null` for a deployment estimate.
`gas`	QUANTITY	No	Execution QGAS limit.
`gasPrice`	QUANTITY	No	Gas price for the execution context.
`value`	QUANTITY	No	QTOV value transferred with the call.
`input`	DATA	No	Method selector + QVM-ABI-encoded parameters.

Fee model placement
The QVM fee model is congestion-responsive and documented in Chapter 8 — it is not interleaved between method definitions. All QUANTITY fee values follow the encoding rules in 15.2.

16.4 REST Developer Endpoints

Alongside JSON-RPC, Quantova exposes REST helpers for fee estimation and bridging used by wallets and dApps:

```bash
GET  /v1/fees/estimate        # suggested base fee + priority fee
POST /v1/fees/simulate        # simulate total fee for a proposed tx

POST /v1/bridge/quote         # fee breakdown for a proposed bridge
POST /v1/bridge/initiate      # initiate an outbound bridge transaction
GET  /v1/bridge/status/:tx    # bridge progress
POST /v1/bridge/claim         # claim inbound bridged assets
```

17. Running Nodes & Operator RPC

How to build the Quantova node from source, run a full or archive node, and configure the operator RPC surface safely. Validators and infrastructure providers should read this alongside the staking requirements (Chapter 10) and the security notes below.

17.1 Build From Source

BASH

```bash
# 1. Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env

# 2. Build dependencies
sudo apt-get update && sudo apt-get install -y --no-install-recommends \
  clang cmake pkg-config libssl-dev protobuf-compiler libprotobuf-dev \
  libclang-dev build-essential git

# 3. Clone + submodules
git clone https://github.com/quantova/quantova.git && cd quantova
git submodule update --init --force --recursive

# 4. Build
cargo build --release --bin quantova-node
```

17.2 Run a Node

LOCAL DEV NODE

BASH

```bash
./target/release/quantova-node --dev --ws-external
# point the SDK at ws://127.0.0.1:9944
```

FULL NODE / PUBLIC RPC PROVIDER

BASH

```bash
./target/release/quantova-node \
  --base-path ~/.quantova/rpc --chain mainnet --name "RPC-Node-001" \
  --port 30333 --rpc-port 9933 --ws-port 9944 \
  --ws-external --rpc-cors all \
  --rpc-max-connections 1000 --prometheus-port 9615
# add --pruning archive for full historical state (indexers, oracles)
```

VALIDATOR NODE

BASH

```bash
./target/release/quantova-node \
  --base-path ~/.quantova/validator --chain mainnet --validator \
  --name "My-Validator" --port 30333 --rpc-port 9933 --ws-port 9944 \
  --bootnodes "/ip4/BOOTNODE_IP/tcp/30333/p2p/BOOTNODE_PEER_ID" \
  --prometheus-port 9615
```

Validator checklist: generate/rotate session keys (`author_rotateKeys`) and register them on-chain; keep session keys separate from funded accounts; keep P2P reachable; monitor sync, production, and finality lag; bond a stash, set preferences, then submit `staking.validate` and wait for election at an era boundary.

17.3 Operator RPC Configuration

Flag	Purpose
`--ws-external` / `--rpc-external`	Bind RPC/WS to all interfaces (place behind a proxy/firewall).
`--rpc-cors all`	Allow browser origins (lock down in production).
`--rpc-methods=Safe|Unsafe`	**Keep Safe in production.** Unsafe only on trusted networks.
`--rpc-max-connections`	Cap concurrent clients.
`--pruning archive`	Retain full historical state (indexer/oracle workloads).
`--prometheus-port`	Expose metrics (peers, import queue, height/finality lag).

Treat RPC as an edge surface
P2P ports must stay reachable for sync and consensus. RPC ports are an application interface — authenticate, rate-limit expensive methods, disable unnecessary modules, and never expose unsafe methods or dev signers publicly.

18. Oracle Integration Guide

Everything an oracle, data provider, or indexer needs to read Quantova safely and at scale. Always read **finalized** state — finality is deterministic, so finalized data will not be reorged.

18.1 Endpoints

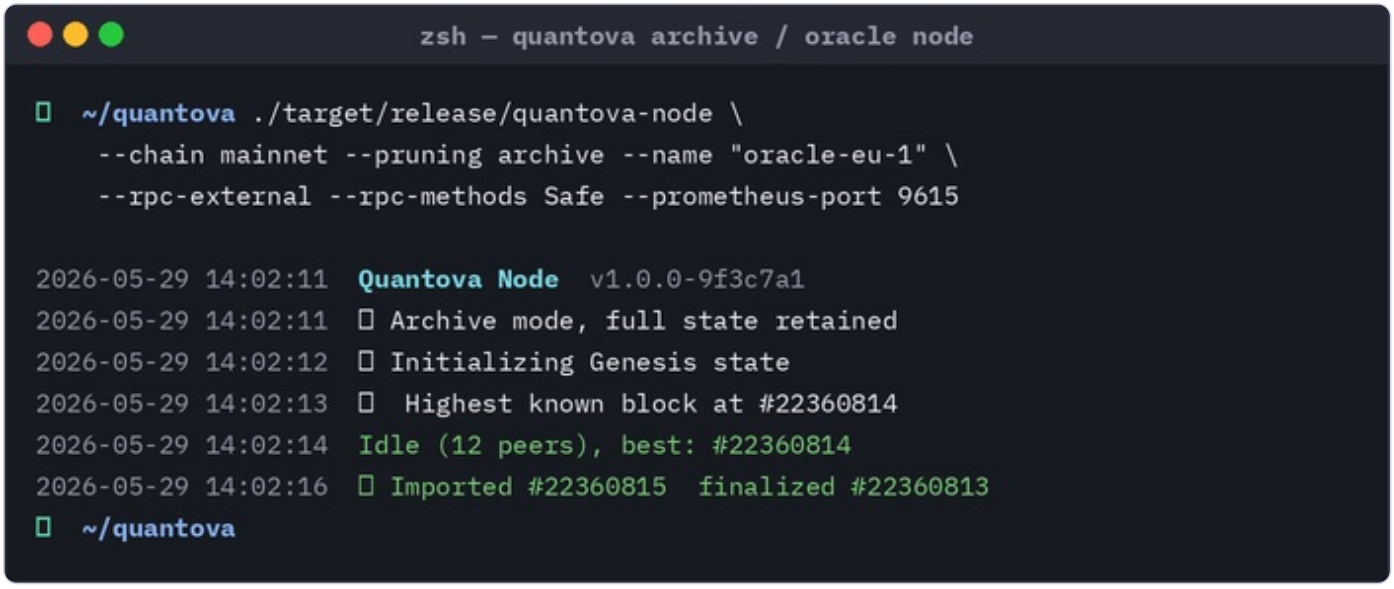
Purpose	Endpoint
WebSocket (subscriptions, runtime + QVM)	`wss://mainnet.quantova.io`
HTTP JSON-RPC (`q_` namespace)	`https://mainnet.quantova.io`
Self-hosted (recommended for production)	Run a full/archive node and read locally

Run your own archive node
For production data feeds, operate a dedicated archive node (`--pruning archive`) rather than depending on public endpoints. It removes rate limits, gives full history for backfills, and isolates your latency and availability.

Start the archive node the oracle will read from. The node prints its version, confirms archive mode, syncs to the head, and then imports and finalizes blocks — point your oracle at `http://127.0.0.1:9933` once it reports a finalized height:

BASH

```
./target/release/quantova-node \
  --chain mainnet --pruning archive --name "oracle-eu-1" \
  --rpc-external --rpc-methods Safe --prometheus-port 9615
```



Bringing up a Quantova archive node for an oracle feed. Wait for an 'Imported … finalized' line before reading.

18.2 Reading Finalized Data

Subscribe to finalized heads and read state at the finalized hash — never act on the latest (possibly forked) head:

JAVASCRIPT

```
import { ApiPromise, WsProvider } from '@quantova/api';
const api = await ApiPromise.create({ provider: new WsProvider('wss://mainnet.quantova.io') });

const unsub = await api.rpc.chain.subscribeFinalizedHeads(async (header) => {
  const at = await api.at(header.hash);                 // state at the finalized block
  const { data } = await at.query.system.account(TARGET_Q_ADDRESS);
  console.log('finalized #', header.number.toNumber(), 'balance', data.free.toString());
});
```

For QVM contract data, read at a finalized block number with `q_call` / `q_getStorageAt`, passing the finalized block number from `q_blockNumber`.

18.3 Methods Reference for Data Providers

Need	Method(s)
Finalized block height	`q_blockNumber`; `chain.subscribeFinalizedHeads`
Block contents & timestamp	`q_getBlockByNumber`, `q_getBlockByHash`
Account / contract balance	`q_getBalance`; runtime `system.account`
Read contract state (view call)	`q_call`
Raw contract storage slot	`q_getStorageAt`
Confirm a transaction	`q_getTransactionReceipt`
Gas / fee context	`q_gasPrice`, `q_feeHistory`, `q_maxPriorityFeePerGas`
Runtime storage (any pallet)	`api.query.<pallet>.<item>`
Chain / network identity	`q_chainId`, `q_net_version`, `system.chain`

18.4 Best Practices

- Act only on finalized state; deterministic finality means no reorg risk once finalized.
- Prefer subscriptions over polling for heads; poll `q_feeHistory` on a schedule for gas models.
- Pin reads to an explicit finalized block number so a feed is reproducible and auditable.
- Pass account addresses as canonical Q-format strings to `q_` calls; the SDK handles any internal encoding.
- Run redundant nodes across regions; alert on finality-lag and height-lag, not just liveness.

19. Chain Specifications

The canonical quick reference for Quantova's network-level parameters — the authoritative values governing identity, consensus, execution, staking, and economics. Consensus-critical economic values are restated here for convenience; the full treatment is in Chapters 7–10.

Network identity & token

Native asset
QTOV
Token standard
Quantova native
Decimals
18
Genesis supply
1,050,000,000 QTOV
Address format
Q-prefixed, 20-byte H160 (0x40 marker)
Address derivation
SHA3-256(publicKey)[0..20], byte0 = 0x40

Consensus & block configuration

Consensus
Nominated Proof-of-Stake (NPoS)
Block production
Deterministic round-robin (no VRF)
Block time
~2.5 seconds
Finality
~3 seconds (deterministic, provable finality)
Epoch length
24 hours
Active validator cap
200 (target at maturity)

Staking & slashing

Minimum self-stake
$50,000 USD-equivalent in QTOV (governance-adjustable)
Uptime requirement
95% per epoch
Unbonding period
14 days
Redelegation cooldown
7 days
Commission range
0% – 25% (max change 1%/epoch, 5%/day)
Slashing
1% minor downtime · 100% + permanent ban for double-signing, bridge manipulation, or equivocation; all forfeited stake to Quantova treasury

Fees & economics

Fee model
EIP-1559 style (base fee + priority tip); no burn
Base fee
Algorithmic; 50% target utilization; ±12.5%/block; 100% to Quantova treasury
Priority fee
100% to the including validator
Average transaction fee
$0.05 - $0.10 USD
Inflation
5.0% (Yr 1) declining to 1.5% terminal floor (Yr 9+)
Issuance recipients
100% to validators & delegators by active stake
Fee destination
Base & bridge protocol fees to treasury; priority & attestation fees to validators; no burn

Bridge configuration

Protocol fee
0.05% QTOV / 0.10% major / 0.15% long-tail
Protocol-fee destination
Quantova treasury (no burn)
Caps
None (permissionless)
Settlement
3–20 minutes by direction

20. FAQ

Do Quantova addresses really start with Q?

Yes — by design. The Q prefix is Quantova's native address brand. Every post-quantum account fixes its first byte to `0x40`, which the canonical form renders as a leading Q, so every valid address begins with Q. When Alice sends to Bob, both are Q-addresses, and the Q-format string is used everywhere — wallet, explorer, `q_` RPC, and QVM Solidity. The `0x` notation is reserved for hashes and quantities, not addresses (see 5.3 and 15.1).

Why hash the public key — and why is the digest 32 bytes?

Quantova hashes each account's post-quantum public key with SHA3-256 and derives the address from the 32-byte digest. Hashing the key gives four properties:

- **Fixed-size addresses.** Dilithium, Falcon, and SPHINCS+ public keys range from ~1 KB to many KB. Hashing collapses any of them into a uniform 32-byte digest, so the address format is identical regardless of scheme.
- **Smaller on-chain state.** You never store kilobyte-scale public keys as addresses; the key is revealed only when an account first signs.
- **Algorithm agility.** The address layer stays fixed as an account migrates schemes or as new PQ standards mature.
- **Preimage resistance.** An address should not reveal the public key; a hash provides that.

So why is the address 20 bytes, not 32?

The SHA3-256 *digest* is 32 bytes, but the *address* is the first 20 bytes of it (H160), with byte 0 fixed to the `0x40` Q marker. The address therefore carries 152 bits of derived entropy. Hashing happens at 32 bytes; the address is the 20-byte slice.

Which signature scheme should I use?

There is no default — choose explicitly. Falcon for the smallest signatures (also used for validator/authority keys); Dilithium for a balanced lattice-based option; SPHINCS+ for hash-based security with the most conservative assumptions at the cost of larger signatures.

How do I write and deploy a contract?

Write Solidity using the `QPrimitives` library for post-quantum operations (14.2–14.4), then deploy through **Qdock.io** (the IDE) signing with **Qmask.io** (the wallet), or programmatically with `qweb3.js` (14.5).

How does QTOV supply change — is anything burned?

Supply changes only through issuance, and **nothing is burned**. Issuance is disinflationary — 5.0% in Year 1 declining to a 1.5% terminal floor — and is paid entirely to validators and delegators as staking rewards. Transaction and bridge fees are not destroyed: the base fee and bridge protocol fee are collected to the Quantova treasury, and the priority and attestation fees go to validators. QTOV forfeited by slashing or failed governance bonds is also routed to the treasury. Fees and penalties therefore circulate as revenue rather than reducing supply.

Does Quantova use a VRF for block production?

No. There is no widely accepted post-quantum VRF, so Quantova removed VRF-based leader election. Slot assignment is deterministic round-robin, and protocol randomness is sourced from post-quantum primitives.

Does the runtime have a superuser or admin key?

No. There is no `pallet_sudo`, no superuser, and no emergency key. All Root-origin authority derives exclusively from on-chain governance via referenda, and no single entity can change the network. Even minting and address-freeze actions require bonded supermajority referenda, not an admin key.

Is Quantova a fork of another chain?

No. Quantova is an independent Layer-1 engineered from first principles around post-quantum cryptography. It is built with the Substrate framework as a toolkit, which is categorically different from forking an existing chain's ledger and codebase.

Licensing, Intellectual Property & Copyright

This section is the authoritative licensing overview for the Quantova technology stack. It covers the software, specifications, and documentation developed and maintained by Quantova Inc., and applies to all associated materials unless explicitly stated otherwise.

Copyright & Ownership

 Quantova Inc. is a company registered in Singapore and is the legal owner and steward of:

- The Quantova protocol
- The QRC-20 network standards
- The Quantova Virtual Machine (QVM)
- The Provenance and Quantization Registry (PQR)
- Associated research, specifications, and reference implementations

Primary License: Business Source License (BUSL-1.1)

Unless otherwise stated, all Quantova protocol code — including but not limited to consensus and finality logic, runtime and state-transition code, the Quantova Virtual Machine (QVM), networking and node implementation, and protocol-level cryptographic integrations — is licensed under the **Business Source License, version 1.1 (BUSL-1.1)**.

The full license text is included in the repository at `/LICENSE-BUSL-1.1`. The authoritative BUSL-1.1 text may also be obtained from mariadb.com/bsl11.

Validator & Node-Operator Clarification

Running a validator node or full node on the canonical Quantova network is explicitly permitted under BUSL-1.1 and does **not** constitute restricted "Production Use." Participation in staking, consensus, block production, and transaction processing on the Quantova network does not create additional licensing obligations. A formal clarification for validators, node operators, exchanges, custodians, and infrastructure providers is provided at `/docs/validators/licensing.md`.

Restricted Use Under BUSL-1.1

BUSL-1.1 restricts the use of Quantova consensus, runtime, and QVM code to launch, operate, or market a competing blockchain network or distributed ledger that is not the canonical Quantova network. This restriction applies to, but is not limited to:

- Independent or derivative mainnets
- Forks marketed as separate networks
- Networks intended to replace or compete with Quantova

Forking, modifying, or analyzing the code for testing, auditing, research, or contribution purposes is permitted.

Canonical Network Definition

For licensing and authorization purposes, the canonical Quantova network is defined by all of the following:

- Official signed source releases published by Quantova Inc.
- A unique post-quantum genesis hash
- Signed runtime and protocol artifacts
- Post-quantum cryptographic signatures, including CRYSTALS-Dilithium and Falcon

Any network deployment that does not match these identifiers is not the Quantova network and is not authorized under the Quantova BUSL license. Protocol identity and licensing intent are cryptographically anchored at genesis via an immutable on-chain commitment.

Documentation, Third-Party Software & Disclaimers

Documentation, research materials, and protocol specifications are licensed under BUSL-1.1 unless explicitly stated otherwise within the relevant file or directory, and are provided for informational and technical reference, reflecting protocol behavior enforced by code.

This repository may include or depend on third-party open-source software; such components remain subject to their original licenses, with notices provided in `/THIRD_PARTY_LICENSES.md` or within dependency manifests.

Trademarks. Quantova™, the Quantova logo, Quantova Network™, QTOV™, QVM™, Qmask.io™, Qdock.io™, QNS™, and the Q-address brand are trademarks of Quantova Inc. The trademarks are not licensed under BUSL-1.1 or any third-party software license and remain the property of Quantova Inc.

No legal, financial, or regulatory advice

Nothing in this repository, including the documentation and specifications, constitutes legal, financial, or regulatory advice. Operators, validators, and users are responsible for ensuring compliance with applicable laws and regulations in their respective jurisdictions. Licensing, institutional, and regulatory inquiries should be directed through official Quantova channels published by Quantova Inc.

Document control

Quantova Developer Documentation · Version 1.0 · © 2026 Quantova Inc. (Singapore) · Protocol code and documentation licensed under BUSL-1.1 · Quantova™ and QTOV™ are trademarks of Quantova Inc.