**Top 15 Blockchain Networks In-Depth Review**

Below we provide detailed profiles for the **top 15 blockchain networks** identified above, following a structure similar to the provided Solana and Synergy Network documents. Each profile includes: **Overview** (purpose and key features), **Address & Key Formats** (how addresses and keys are structured, with examples), **Transaction & Token Details** (fee mechanics, token standards, etc.), and other relevant categories (e.g. **Consensus & Security**, **Cross-Chain Interoperability**). If a category from the reference docs (e.g. “Freeze Authority”) doesn’t apply, we substitute a comparable relevant detail.

**Solana**

**Overview:** Solana is a high-performance Layer-1 blockchain utilizing a hybrid Proof-of-History (PoH) + Proof-of-Stake consensus. It is designed for scalability, achieving block times of ~400ms and throughput of tens of thousands of TPS in practice. **Solana’s architecture** emphasizes parallel transaction processing (Sealevel runtime) and uses the Rust and C++ programming languages for smart contracts. In 2024, Solana saw massive growth in user activity (2.6M daily active addresses) driven by its vibrant ecosystem of DeFi, NFTs, and even meme coins​. The network’s low fees (a fraction of a cent) and high speed have attracted developers and users to applications like Serum (DEX), Raydium, and Phantom wallet. Solana’s native asset **SOL** is used for staking (securing the network via ~1,800 validators) and paying transaction fees.

**Address & Key Formats:** A Solana **wallet address is the public key** of an account (32 bytes), encoded in base58. There is no fixed prefix for Solana addresses (unlike Ethereum’s 0x); they can start with any alphanumeric character except 0/O and l/I (due to base58 alphabet)​. Typical Solana addresses are 44 characters (e.g. 8HJc8B4...XqNi7) though they can range 32–44 chars​. The address is derived from an ed25519 public key (Solana uses ed25519 keypairs – the private key is 64 bytes (32-byte secret + 32-byte public)). **Example:** a private key might be ed25519:4B9DmX... and its public key (and address) could be FsjE8hD3J4y... (base58). Solana **does not employ checksums** in addresses (relying on signature verification for correctness). There is no lowercase/uppercase distinction (all base58).

* *Associated accounts:* Solana uses a unique system for token accounts. Each SPL token held by a user resides in an **Associated Token Account (ATA)**, which is a derivation of the user’s main address and the token mint address. The ATA is also a 32-byte address (base58). For example, if SOL address = AlicePubKey, and token = USDC with mint EPjFW...Dt1v, Alice’s USDC account address is deterministically derived (via a program) – ensuring a one-to-one mapping.

**Transaction Structure:** Solana transactions are structured as a list of instructions to programs. A single transaction can contain multiple instructions (e.g. transfer tokens, invoke a contract) and supports **atomicity** (all or none execute). Transactions are identified by their signature (Solana uses the first signature as the “transaction ID”). A transaction ID in Solana is essentially a base58 string (64 chars) representing the hash of the signed transaction (often the same as the first signature). *Example:* 5YWoD5v...ftKGmZys9wqfNLXfpQvsXHAbf3Uf8ZvJc9 is a transaction hash. Solana’s **fee** for a basic transaction is 0.000005 SOL (5e-6 SOL) by default, and it uses a priority fee mechanism for congestion (users can add an extra tip in SOL).

**Native Token (SOL) Details:** SOL has a total supply of ~558 million, with inflationary rewards (initial inflation ~8% declining over time). SOL is used to stake with validators (Solana uses delegated PoS — users delegate SOL to validators and earn rewards ~6% APY). There is no explicit “mint authority” for SOL like an SPL token; SOL’s supply is managed by protocol governance (and inflation schedule). **Freeze authority:** Not applicable to SOL (it’s the native coin). However, for SPL tokens on Solana (analogous to ERC-20), there can be a freeze authority and mint authority on the token’s mint account (as given in the Solana Token Program).

**Token Standards:** Solana’s token program (SPL Token Program) defines the standard for fungible tokens, analogous to ERC-20. Each **SPL token** has:

* a **Mint Address** (identifies the token type),
* metadata via the Metaplex Token Metadata program (name, symbol, URI, etc.),
* authority roles: *Mint Authority* (who can mint new tokens) and *Freeze Authority* (who can freeze token accounts). For example, USDC’s mint EPjFW...Dt1v has its mint authority set to a specific upgradeable program (for freeze/mint control).

Solana also supports NFTs via the Metaplex standard (which are essentially SPL tokens with supply 1 and unique metadata).

**Smart Contracts (Programs):** Solana smart contracts are called **programs**, identified by program ID (also a 32-byte address). For instance, the SPL Token Program has the well-known id TokenkegQfeZ...623VQ5DA​. Programs are deployed via a special loader and are immutable (upgradeable only if using a proxy/upgrade authority pattern). Interactions with programs (invoking them) are included as instructions in transactions.

**Consensus & Security:** Solana’s consensus is a customized Tower BFT (Byzantine Fault Tolerance) that leverages the PoH clock. Validators take turns leadering (producing blocks) in a schedule, and PoH provides a cryptographic timestamp that orders events. Solana currently has a validator quorum that votes on blocks, requiring supermajority ~66% to finalize. Checkpoints (finalized blocks) on Solana occur typically within a few seconds (Solana’s confirmation times are often 1–2 seconds for finality under normal conditions). To secure the network, validators have to stake SOL – any malicious behavior can lead to slashing (though slashing is currently conservative/non-frequent on Solana). As of 2024, Solana’s Nakamoto coefficient (number of validators required to collude to slow the network) was around 30.

**Interoperability:** Solana supports cross-chain bridges (e.g. Wormhole, which connects Solana with Ethereum, BSC, Terra, etc.). Solana addresses can be mapped from other chains via these bridges (e.g. Wormhole creates wrapped assets on Solana with their own mint addresses corresponding to ETH or BSC tokens). Additionally, Solana is integrating with interoperability protocols and has its own concept of **Universal Address** mapping via the Address Lookup Table feature (which is more for transaction size reduction). From the Synergy perspective (UMA – Universal Meta-Addresses), Solana could participate by linking its addresses to a meta-address scheme, but natively, Solana addresses are unique to Solana.

**Developer & Program Info:** Key programs on Solana include:

* **System Program:** facilitates account creation and SOL transfers (id: 11111111111111111111111111111111).
* **SPL Token Program:** for fungible token management (id: TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA)​.
* **Token Associated Account Program:** to derive token accounts (id: ATokenGPvR8...47sX)​.
* **Metaplex Token Metadata Program:** for NFT metadata (id: metaqbxxUerd...518x1s)​.

Solana’s RPC API allows querying account data, transaction history, etc., similar to Ethereum’s JSON-RPC but with Solana-specific methods (e.g. getAccountInfo, getProgramAccounts). There are multiple explorers (Solana Beach, Solscan, Explorer.solana.com) to view transactions and accounts by their base58 addresses.

**NEAR Protocol**

**Overview:** NEAR is a Layer-1 blockchain that uses **sharding** to scale (its Nightshade sharding approach). It employs a Proof-of-Stake consensus (called Doomslug for block production and finality gadget) with a focus on developer-friendliness and user-friendly account names. NEAR’s differentiator is its **human-readable accounts** (e.g. alice.near) and robust runtime that allows WASM smart contracts (Rust and AssemblyScript are common). NEAR saw a surge in activity in 2024 (reaching ~3.3M daily addresses) attributed to growing DeFi and gaming dApps and maybe its move to phase1 sharding​

[cryptonews.net](https://www.cryptonews.net/news/blockchain/30298036/#:~:text=1,YoY)

. The native token **NEAR** is used for staking (validators earn ~4.5% APY) and transaction fees (NEAR has low fees, on the order of $0.001 for simple transfers).

**Account & Address Structure:** NEAR accounts can be one of two types​

[docs.near.org](https://docs.near.org/concepts/protocol/account-id#:~:text=NEAR%20accounts%20are%20identified%20by,takes%20one%20of%20two%20forms)

:

* **Named Accounts:** human-readable strings similar to domain names, e.g. example.near. These can have subaccounts like app.example.near. Named accounts are at least 2 characters, may include alphanumeric, - or . (for subaccounts), and are managed via NEAR’s namespace model (an account can create a subaccount if it owns the parent name).
* **Implicit Accounts:** 64-character hexadecimal addresses, which are the lowercase hex encoding of a 32-byte public key​

[docs.near.org](https://docs.near.org/concepts/protocol/account-id#:~:text=1,alice.near)

. These look like fd83b2dc2f104efcbf8d6d7c4a8d... and are used as default when a public key has no named account attached. For example, if someone only has a key and hasn’t registered a name, their account ID is the 64-char hex of their ed25519 pubkey. Any NEAR public key corresponds to an implicit address (if that address receives funds, the account is implicitly created).

**Key Management:** NEAR’s approach to keys is flexible – each account can have multiple key pairs associated with it, each with different permissions:

* **Full access keys:** can sign any transaction for the account.
* **Function call keys:** limited to calling specific contract methods with a transaction fee allowance (useful for giving dApps the ability to act on behalf of the user within limits).

For example, account alice.near might have one full key controlled by Alice’s wallet, and a function-call key given to a game dApp to allow it to perform a certain number of moves on behalf of Alice.

**Address Examples:**

* Named: weather.near (could be a contract or user account).
* Implicit: 4f2ba3c1080e3be3f8e1d3f79d6c5e8b549cf1c552d8e8f2146c3feaac6e5e44 (this is 64 hex chars, and if you prepend “ed25519:” and base58 encode, you’d get the full key).

**Transaction Format:** NEAR transactions include: signer ID, public key, nonce, receiver ID, actions, and a recent block hash for validity. Actions can be things like: Transfer, FunctionCall, CreateAccount, DeployContract, Stake, etc. A transaction hash on NEAR is a base58-encoded 32-byte value (since NEAR uses ed25519 signatures, the transaction hash is typically the hash of the signed TX). NEAR has **human-readable receipts** as well, representing cross-contract calls that complete asynchronously. Finality on NEAR is typically 1-2 seconds (one block for near-instant finality due to Doomslug, and 2-3 blocks (~3s) for absolute finality).

Transaction fees on NEAR are paid in NEAR and are typically very low; NEAR uses a gas unit (1e12 gas roughly equals 0.001 NEAR fee, for example). NEAR’s fees are partly burned (NEAR has a fee-burning mechanism: 70% of fees are burned, 30% go to validators as reward), making NEAR somewhat deflationary if network usage is high.

**Native Token (NEAR) Info:** NEAR (the token) started with 1 billion supply at genesis, with an annual inflation of 5% (to validators) minus fee burns. NEAR’s circulating supply is slightly inflationary but high usage can reduce net inflation. NEAR tokens are needed to become a validator (validators must stake a certain threshold, which dynamically adjusts based on total stake and number of seats). Governance of protocol upgrades is currently mainly via the NEAR Foundation and core developers, but moves toward more open governance are in progress.

**Contract Model and Tokens:** NEAR’s runtime supports **WASM smart contracts**. A contract is associated with a NEAR account (the account name serves as the contract’s address). For example, the popular automated market maker REF Finance has contract accounts like app.ref-finance.near. To call a contract, one sends a transaction to its account ID with a FunctionCall action.

NEAR’s equivalent of ERC-20 is the **NEP-141 standard** (for fungible tokens). Under NEP-141:

* Tokens are smart contracts on NEAR accounts (often ending with “.token.near” or similar).
* They implement methods like ft\_transfer, ft\_balance\_of, and support an analog of allowance (ft\_transfer\_call for callbacks instead of approve).
* Example: The USDC token on NEAR (native via Circle) might reside at usdc.facts.testnet (on testnet) or a similar contract on mainnet. The contract manages balances mapping of account -> balance internally, and calls the NEAR runtime’s storage for maintaining state.

NEAR also has standards for NFTs (NEP-171) and other assets. Each again is a contract on an account (e.g., an NFT collection might be a contract account, with each token identified by an internal ID).

**Security & Consensus:** NEAR uses a BFT consensus (nightshade) where validators produce chunks for their shard and a block producer assembles them. NEAR’s sharding (as of 2024) might be in a phase where there’s one shard for execution and “chunk-only producers” for scaling out, with full sharding (multiple execution shards) on the roadmap. Finality is quick (one round of communication thanks to Doomslug – once a block is seen and a quorum of validators have signed approvals, it’s final). NEAR’s staking mechanism selects validators for each epoch (~12 hours) based on stake weighted. It has on-chain governance for certain parameters but not a coin-vote governance like some chains (the NEAR Foundation and core devs manage upgrades).

**Cross-Chain & Bridges:** NEAR was built with interoperability in mind. It has the **Rainbow Bridge** connecting it to Ethereum – which allows ERC-20 tokens to hop to NEAR as NEP-141 tokens (for instance, Ethereum’s DAI when bridged becomes an equivalent token contract on NEAR). NEAR’s sharded design also means it can host **Aurora**, an EVM-compatible environment that runs as a smart contract on NEAR (Aurora has its own ETH-like address space and uses a separate token AURORA for governance but uses NEAR for base security). Assets can move between NEAR and Aurora internally with connectors.

NEAR accounts can call contracts on Aurora via cross-contract call (Aurora is essentially one big contract on NEAR that interprets Ethereum transactions). Conversely, Aurora’s Rainbow Bridge connects to Ethereum L1 directly as well.

**Notable Projects & Ecosystem:** NEAR’s ecosystem includes Ref Finance (AMM DEX), MetaPool (staking pool for liquid NEAR), Sweat Economy (move-to-earn app issuing SWEAT token on NEAR), Mintbase (NFT marketplace), and Aurora (EVM) hosting things like Trisolaris DEX. Many of these have their tokens (REF, META, SWEAT) as NEP-141 contracts. NEAR also has a naming system integrated (every account is essentially a name, and there isn’t a separate DNS or ENS needed, except to map .near to DNS perhaps).

**Example Walk-through:** If Alice (alice.near) wants to send 10 NEAR to Bob (bob.near), she’d submit a transaction with: signer=alice.near, receiver=bob.near, action=Transfer{10 NEAR}. Bob’s account increases by 10 NEAR. If Alice wants to call a contract (say the fungible token contract at usdt.tether.near) to transfer 100 USDT to Bob, the transaction: signer=alice.near, receiver=usdt.tether.near, action=FunctionCall{method: 'ft\_transfer', params: { "receiver\_id": "bob.near", "amount": "100000000" }, deposit: 1 yoctoNEAR}. This invokes the contract which will reduce Alice’s internal balance and increase Bob’s. Bob will then have an entry in that contract’s internal ledger indicating his USDT balance.

**Developer Info:** NEAR provides a CLI and SDKs. NEAR’s accounts and contract model means that to deploy a contract, one must call deploy\_contract action on an account (which uses that account’s balance to pay for storage). Storage on NEAR costs NEAR tokens (state rent-like mechanism: a small amount of NEAR gets locked for storing data, and if data is freed, NEAR is returned). This means contract developers often require users to attach a deposit for certain calls that increase storage (for example, when a user registers an account with a token contract, they might need to attach 0.00125 NEAR as storage payment).

**TRON**

**Overview:** TRON is a Layer-1 blockchain aimed at high throughput and low-cost transactions, often associated with entertainment content and, more recently, as a popular network for stablecoins. TRON uses a **Delegated Proof-of-Stake (DPoS)** consensus with a fixed set of 27 Super Representatives producing blocks in a round-robin fashion every 3 seconds​

[wazirx.com](https://wazirx.com/blog/faqs/how-tron-works/#:~:text=How%20Tron%20Works%3F%20,track%20of%20the%20transaction)

​

[binance.com](https://www.binance.com/research/projects/tron#:~:text=%2A%20Delegated%20Proof,scalability%20than%20existing%20programmable%20networks)

. This yields a typical TPS in the hundreds and very fast block confirmation (1 block ~3s final). TRON’s ecosystem includes Tron-based tokens (TRC-20 and TRC-10 standards), a variety of dApps (mostly gambling, simple games, and DeFi like JustLend, SunSwap), and notably it became the chain carrying a large portion of **Tether (USDT)** transactions due to negligible fees. The native token **TRX (Tronix)** is used for voting for Super Representatives and as the reserve currency for bandwidth/energy.

**Accounts & Addresses:** TRON’s address format is inspired by Ethereum’s (since Tron’s VM is EVM-like), but uses a different encoding. **Tron addresses** are 21-byte (160-bit) hashes with a version prefix 0x41, and are typically represented in **Base58Check** with a **“T” prefix**​

[ccfound.com](https://ccfound.com/en/questions/8237/what-signs-does-the-tron-trx-wallet-address-consist-of#:~:text=ccFound%20ccfound,and%20unique%20for%20each%20wallet)

​

[help.revolut.com](https://help.revolut.com/help/wealth/cryptocurrencies/transferring-cryptocurrencies/depositing-cryptocurrencies/issues-with-crypto-deposits/which-addresses-can-i-revert-my-deposit-to/#:~:text=,Addresses%20start%20with%20%27S)

. For example, an address might be TNDzfERD...p7Ar (which corresponds to hex 41... address). All Tron addresses for externally owned accounts (EOAs) and smart contracts share this format and always begin with “T”. Internally, Tron’s VM deals with the hex form (e.g., 0x41 + 20 bytes), but users see the Base58 T-address. **Public-private keys:** Tron uses the same elliptic curve (secp256k1) as Bitcoin/Ethereum. A Tron address is derived by taking the Keccak-256 hash of the public key and using the last 20 bytes, then adding 0x41 in front (which yields a hex starting 41...). That, with a SHA-256 double hash for checksum, is encoded to give the T... string. *Example:* A private key (random 32 bytes) -> public key (65-byte uncompressed) -> Keccak-256 -> say hash ends in 0x27ae... -> address hex = 4127ae... -> Base58Check = T9z.... The addresses are case-insensitive (Base58) and checksummed by the encoding.

**Resource Model (Bandwidth & Energy):** Instead of gas, TRON uses a resource model:

* **Bandwidth points:** Every account gets some free bandwidth daily (for simple transfers). Additional bandwidth or Energy can be obtained by freezing TRX (locking TRX grants resources) or by burning TRX directly as fee. Bandwidth is consumed by transactions (1 byte of tx = 1 bandwidth point). If an account lacks bandwidth, it pays 0.1 TRX for 1 bandwidth point (or must freeze TRX to gain).
* **Energy:** Used for smart contract execution (computational resources). Energy is only obtained by freezing TRX for a period (or having someone sponsor the contract call). If not enough Energy, TRX will be burned to cover execution. This model means typical TRC-20 token transfers cost some Energy and bandwidth, but many dApp operators will require users to freeze some TRX to interact continually.

**Transaction Features:** TRON’s block time is 3 seconds, and one block contains typically up to ~2000 transactions. **Transaction IDs** on Tron are a 32-byte hash (txid = keccak256 of the raw transaction). They’re usually represented as hex strings (not Base58) in block explorers. For example, a0b9f1...88d3 (64 hex chars). Tron transactions are signed via ECDSA (like Ethereum’s secp256k1) but Tron uses a different hashing (it signs the SHA256 double hash of the transaction protobuf). There is no notion of transaction nonce per account (Tron uses an incremental “latest timestamp” or “duplicate check” mechanism to prevent replay, and each transaction has a timestamp). Tron transactions can contain multiple operations as well (like multiple transfers, but commonly it’s one contract call or one transfer per tx).

**Smart Contracts & Tokens:** Tron has two token standards:

* **TRC-10:** Basic built-in token type that can be created by paying 1024 TRX. TRC-10 tokens are simpler (they don’t involve smart contracts; they are akin to “colored coin” accounts maintained by the protocol). They have numerical IDs. e.g., BitTorrent (BTT) was originally TRC-10 with ID 1002000.
* **TRC-20:** Equivalent to ERC-20 smart contract tokens running on Tron’s Tron Virtual Machine (TVM). TRC-20 allows full smart contract logic. Notably, USDT on Tron is a TRC-20 contract (address TR7NHqjeKQ...Lj6t as cited)​

[tron.network](https://tron.network/usdt#:~:text=Exchanges%20Supporting%20TRC20%20based%20USDT,smart%20contract%20address%3A%20TR7NHqjeKQxGTCi8q8ZY4pL8otSzgjLj6t)

, which implements the usual functions (balanceOf, transfer, etc.). Tron’s smart contracts are written in Solidity and are very close to Ethereum’s (some opcodes differ, and Tron has some system contracts for certain functions). TVM lacks gas fees per op—execution simply consumes Energy. A contract will run until it’s out of Energy. If a user hasn’t enough Energy, the execution fails or burns TRX for energy.

**Notable Tron Contracts:** JustSwap (now SunSwap) is a TRC-20 DEX, JustLend for lending, Sun.io for staking—these mirror Ethereum DeFi analogs. Tron’s contracts have addresses like EOAs (no special prefix; they also start with T and are distinguishable only by chain data indicating there’s code at that address). To find if an address is a contract, one queries a Tron node or explorer (for instance, TRONSCAN API has getContract).

**Consensus & Governance:** TRON’s 27 Super Representatives (SRs) are elected by TRX holders. TRX holders can **freeze** TRX to get “Tron Power” (vote power) and vote for SR candidates. The top 27 by votes become SRs and produce blocks in turns (each gets to produce one block every 3 seconds, effectively). SRs get block rewards (e.g. 32 TRX per block, which is distributed to voters as they choose – many SRs share rewards to attract votes)​

[trondao.medium.com](https://trondao.medium.com/tron-developer-guide-super-representatives-4d5b1d87ecb4#:~:text=DAO%20trondao,32%20TRX%20to%20Super%20Representatives)

. This DPoS means the network is fairly centralized in terms of block production (27 entities), but allows high throughput and low cost. TRON’s on-chain governance is limited (SRs essentially govern by choosing to run certain protocol updates; there isn’t an on-chain voting for parameter changes by tokenholders except through electing SRs). The total TRX supply is about 91 billion, with some deflationary burning (for example, a portion of fees or account creation costs get burned). TRON initially was inflationary with block rewards but in recent tokenomics updates, block rewards now come from a community reserve etc. (Adjustments over time have aimed to control inflation).

**Cross-Chain Interoperability:** TRON has its own official cross-chain bridge called **BitTorrent Chain (BTTC)** connecting Tron, Ethereum, and BSC, and others for asset transfers. TRON also introduced **USDD**, an algorithmic stablecoin (though its stability mechanism is questionable after Terra’s collapse) – USDD exists across Tron and other chains via bridging. Tron’s position in the crypto ecosystem is often as a cheap settlement layer for stablecoins; exchanges support TRC-20 USDT widely for withdrawals because of near-zero fees. Tron doesn’t natively interoperate with Ethereum (no shared address format or L2), but wrapped versions of assets exist (e.g., ETH on Tron as TRC-20).

**Address & Key Example:** A sample key:

* Private key (hex): 0xC3A... (64 hex chars).
* Corresponding Tron address: hex 41 + RIPEMD160(KECCAK(pubkey)) = e.g. 41A614... which in Base58 becomes TLa2f6VPqDgRE67v1736s7bJ8Ray5wYjU7 (this is actually Tron’s USDT contract address on Tron) – as you see, it starts with T and is fairly long (34 chars). A user wallet address example: TMwFHYXLJaRUPeW6421Mq... etc. Public keys themselves are not commonly shown to users, only addresses.

**Token Economics:** TRX has no maximum supply (initially 100 billion in 2017, current ~91.2 billion after burns). There is a natural burn of 0.1 TRX per transaction if free bandwidth is exceeded, and the new account creation fee (via sending TRX to a new address) costs 0.1 TRX burn. But block rewards add TRX. Tron’s token governance also includes **SR-partner rewards** (ranks 28–127 get smaller rewards). In mid-2021, the protocol started burning some TRX, making it slightly deflationary at times.

**Notable App:** One special system contract is the **TRON token issuance contract** for TRC-10, and **Proposal system** where SRs vote on proposals (like changing parameters). These are part of the Tron protocol governance.

**BNB Chain (Binance Smart Chain)**

**Overview:** BNB Chain (specifically the BNB Smart Chain, BSC) is a Layer-1 blockchain initially developed by Binance, optimized for high throughput of smart contracts with EVM compatibility. BSC gained prominence in 2021 as a low-fee alternative to Ethereum, leading to a rapid expansion of DeFi projects (PancakeSwap, Venus, etc.). It uses a unique consensus called **Proof-of-Staked Authority (PoSA)**, which is effectively a limited validator set (21 active validators at a time) voted in by BNB staking​

[coinmarketcap.com](https://coinmarketcap.com/view/bnb-chain-ecosystem/#:~:text=Security%20and%20decentralization%20efforts%20continue,the%20ecosystem%E2%80%99s%20commitment%20to%20decentralization)

. Block time is ~3 seconds, and capacity is similar to Ethereum (~100+ TPS in ideal conditions) but often utilized more fully due to lower fees. **BNB**, the chain’s native token, is used for transaction fees and staking. BNB Chain’s ecosystem now spans not just BSC but also the Beacon Chain (for simple token transfer & governance) and sidechains, but here we focus on the smart contract chain.

**Address Format:** BNB Smart Chain being EVM-compatible means it uses **Ethereum-style addresses**: 20-byte hex addresses with 0x prefix​

[support.metamask.io](https://support.metamask.io/start/learn/the-ethereum-address-format-and-why-it-matters-when-using-metamask/#:~:text=On%20Ethereum%20and%20other%20networks,They%27re%20also%20not%20case%20sensitive)

. In fact, BSC’s address space is identical to Ethereum’s (some addresses like the zero address, or certain contract addresses like PancakeSwap’s contract, all look like regular Ethereum addresses but operate on BSC). This allows easy porting – users can use MetaMask and the same account on Ethereum and BSC. *Example:* The BSC validator consensus contract is at 0x0000000000000000000000000000000000001000 on BSC (system reserved address), and a typical user address is 0xb8cD...37e4. There’s no built-in vanity or readability feature beyond Ethereum’s (you can use ENS on BSC if it’s pointed via some cross-chain solution, but generally not common).

**Consensus Details:** BSC’s PoSA involves validators taking turns producing blocks in a predetermined schedule. There are 21 active validators (chosen daily from a larger set by BNB staking weight). Every 24 hours an epoch changes who the active validators are based on who has the most delegated stake (somewhat similar to EOS’s DPoS or Tron’s SR, but using stake). Blocks are produced in a rotation; if one validator misses, others still continue (the algorithm is based on Tendermint-style consensus modified for PoSA). BSC sacrifices some decentralization (only ~21 nodes sign blocks) for performance. This consensus yields finality typically within a block (since it’s Tendermint-like BFT, finality is immediate once a block is signed by 2/3 validators). Reorgs are very rare.

**Native Token (BNB) and Economy:** BNB (originally Binance Coin) acts as gas on BSC. Transactions fees are on the order of 5 gwei gas price \* 21k = 0.000105 BNB (with BNB ~$300 that’s <$0.05 typically). BNB has a fixed supply (initial 200m, burning down to 100m). BSC burns a portion of gas fees (burns 10% of gas, and also BNB is burned quarterly from exchange profits). BNB is also staked/delegated to validators (through Binance Chain governance, users can delegate BNB to validators to share block rewards). The block reward on BSC is actually mostly the transaction fees (there’s no new BNB issued on BSC; BNB is not inflationary through block subsidies, unlike ETH’s minimal issuance—BNB is only deflationary via burns).

**Smart Contracts & Tokens:** BSC fully supports Ethereum’s VM and standards:

* **BEP-20** tokens are the BSC equivalent of ERC-20. In practice, they are the same interface as ERC-20. A BEP-20 token contract on BSC might have minor differences (often they add a method to get owner or some “getOwner” per BEP-20 spec, but not required).
* BEP-721 and BEP-1155 similarly mirror ERC-721/1155 for NFTs.

Because of this compatibility, many projects simply deployed their Ethereum contracts to BSC. E.g., the same code of Uniswap became PancakeSwap with minimal modifications. Addresses of deployed contracts on BSC are different from Ethereum’s (unless someone deliberately used CREATE2 with same salt and initial address to clone exactly).

**Address & Key Management:** Users manage BSC accounts with the same tools as Ethereum. For instance, MetaMask with BSC network selected will show the same 0x address and keys. Hardware wallets etc. all interoperable. This drastically lowered the entry barrier for users migrating from Ethereum.

**Transaction Mechanics:** BSC uses essentially the same transaction format as Ethereum (RLP encoded, with fields: nonce, gasPrice, gasLimit, to, value, data, v,r,s). The only difference historically is chainId (BSC’s chainId is 56 for mainnet). This prevents replay of transactions between Ethereum and BSC. Gas limit per block on BSC is higher than Ethereum’s (e.g., 30 million gas vs. Ethereum’s 15 million at one point, BSC often raised it, sometimes 60M gas per block). This contributed to BSC’s higher throughput but also meant BSC validators run heavier loads (which they handle since validators are relatively few and presumably well-provisioned).

**Notable Contracts:**

* **PancakeSwap (CAKE)** – at one point BSC’s largest DEX, its contracts (factory, router, etc.) have addresses on BSC like any Ethereum contract (factory v2 at 0xBCf...FE etc.).
* **Cross-chain Bridge** – Binance provides **Binance Bridge** and now a more decentralized bridge for BSC <-> other chains. Bridged tokens from Ethereum appear as BEP-20 tokens on BSC (e.g., ETH on BSC is represented by Binance-Peg ETH, contract 0x2170Ed...0B907).
* **System contracts:** BSC has a few special addresses (like zero address for burns, address 0x0000...1001 for governance voting contract, etc.). Normal users don’t interact with these directly.

**Staking and Governance:** BSC staking is done on the parallel Beacon Chain (Binance Chain) or via the Binance Smart Chain governance dApp – essentially, users delegate BNB to validators. Validators periodically distribute rewards. Governance proposals (like parameter changes or adding validators) are voted by validators (and maybe via an on-chain proposal system on Binance Chain where BNB holders vote). This governance is somewhat centralized (the largest BNB holders often being Binance or exchanges who can decide validators).

**Interoperability:** BNB Chain is expanding to a multi-chain ecosystem:

* **Beacon Chain (BC)**: originally for simple token transfers and trading (like a DEX chain).
* **BSC (Smart Chain)**: for EVM smart contracts.
* **Bridges:** BNB Chain’s bridges connect to Ethereum, and also to side chains or Layer-2s.
* **Binance Cross-Chain (BCC) communications:** Allows BEP-2 assets on Beacon Chain to move to BEP-20 on BSC and vice versa.
* There’s also **zkBNB** (under development) as a zkRollup attached to BNB Chain, and **OpBNB** (an optimistic rollup launching on BNB Chain). These Layer-2 solutions use BNB as well.

**Address Example Recap:**

* User address: 0xfA3A8...88b0 on BSC (could be same as on ETH).
* Contract address: CAKE token 0x0E09F...aBB98 on BSC (that’s PancakeSwap’s CAKE).
* Both look just like Ethereum addresses (0x followed by 40 hex). They can be checksummed with uppercase for readability.

**Synergy with Synergy Network’s concerns:** If relating to Synergy Network’s UMA (Universal Meta-Address), BSC’s adoption of Ethereum format addresses means that an Ethereum meta-address covers BSC too. A cross-chain address scheme could treat BSC and Ethereum addresses interchangeably with a chain identifier. BNB Chain’s approach wasn’t to differentiate addresses by prefix (like TRON uses T, or Cosmos uses cosmos1) but by requiring context of network.

**Sui**

**Overview:** Sui is a new Layer-1 blockchain (Mainnet launched in May 2023) developed by Mysten Labs (ex-Meta engineers). It focuses on **Move programming language** and object-centric data model to achieve high throughput and fast finality. Sui uses a consensus called Narwhal & Bullshark (an asynchronous DAG-based mempool with a leader-based BFT consensus) for transactions involving shared objects, and bypasses consensus for simple transfers of owned objects (which allows parallelization). In practice, Sui can finalize simple transfers in under 1 second and handle TPS in the thousands (in benchmarks). Sui’s design is somewhat similar to Aptos (both use Move), but Sui emphasizes an object model where many operations don’t require global consensus if they don’t touch shared state. **SUI** is the native token for gas, with a supply of 10 billion (a large portion locked or in future emissions). Sui’s early use cases revolve around gaming (where the object model shines) and DeFi primitives being built from scratch in Move.

**Account Address Format:** Sui accounts (addresses) are based on 256-bit public keys (Ed25519 or Secp256k1 supported). A Sui address is typically represented as a 64-character hexadecimal string (representing 32 bytes). Often prefixed with 0x in documentation. For example: 0x3ae9d3c3b45d2e4f9c737... (64 hex chars after 0x). Sui addresses do not have checksum or case variations (usually lowercase). In Sui’s CLI and explorer, addresses may be abbreviated (first 10 chars + last 10). Keys in Sui follow the format: an Ed25519 public key is 32 bytes, and the address is actually the **SHA3-256 hash of the public key’s bytes** truncated or used fully (there was some slight nuance in Sui’s address derivation, but essentially address = first 32 bytes of SHA3(pubkey) which often equals pubkey if using certain key types like ED25519 where pubkey is 32 bytes, possibly they just use pubkey bytes themselves). Anyway, conceptually, Sui addresses = account identifiers similar to Ethereum (0x plus 64 hex).

**Object IDs:** Another important address-like concept in Sui is the **object ID**. Every resource (coin, NFT, etc.) on Sui is an object with a unique 32-byte ID (same format as addresses). For instance, each SUI token coin is an object with its own ID (though fungible SUI coins can be merged/split). Smart contracts are also objects (called Move modules, published to an immutable address which is an object ID representing that code). So Sui blurs the line: an “address” might refer to a user account or a Move package or a specific object.

**Transaction & Execution:** Sui transactions are different from Ethereum’s. A transaction might be a Move call, which specifies objects being passed as inputs. Sui classifies objects into **owned (single-owner)** or **shared**. If only owned objects are touched, the transaction can be processed individually in parallel. If any shared object is touched (like a global state, e.g., a DEX pool object that is globally accessible), that transaction goes through the full consensus ordering (slightly slower). Sui’s consensus ensures finality typically < 2 seconds for shared object tx. Simpler owned-object transfers can finalize even faster (often under 1 second, sometimes almost instantly from user perspective).

Transaction IDs in Sui are referred to as **digests** (it’s basically the transaction’s BCS-serialized data hash). In explorer they appear as hex strings (64 chars). Example TX digest: 2W5Zp...kJ8Q (they sometimes show as base64 or base58 in APIs, but explorer shows a shortened base58 or something). For simplicity: yes, a transaction ID can be considered a 32-byte hash, shown in base16 or base58.

**Gas and Fees:** Gas on Sui is paid in SUI tokens. Each transaction specifies a gas\_object (which is a coin object owned by the signer to pay fees) and a gas\_budget (max fee willing to pay). Sui’s gas price is set by protocol governance and validators (initially 1 SUI per gas unit, but effectively you can think of it as a fixed rate for now). Most transactions cost a tiny fraction of a SUI (e.g. 0.0001 SUI) because SUI has 10^9 smallest units. Sui’s fee model is simpler: not auction-based like Ethereum, more like fixed price and you just need to budget enough. Unused gas is refunded.

**Move Contracts and Tokens:** Sui uses Move, a Rust-like safe memory language initially from Facebook’s Libra. Move programs compiled to bytecode are published to Sui as **Move packages**. Each package has an address (object ID). Within a package, there are modules that define struct types and procedures. Sui’s object model means, for example, a **fungible token** is typically implemented as a struct type (with fields like balance) and the module offers methods to transfer or split these coin objects. Sui’s standard library provides a **Coin<T>** generic type for fungible tokens, where T is a type that represents the coin’s type (like an empty struct acting as a marker for “SUI” or “USDC”). SUI coin itself is native but also represented as Coin<SUI>. Other custom tokens can be created permissionlessly by publishing a Move contract that mints them, or by using the managed Coin type.

Because objects have distinct IDs, owning 100 SUI could mean you have several coin objects (like one with 60 SUI, one with 40 SUI). Transactions can merge them or split them as needed. This is a different model from Ethereum’s account balance approach, but Sui provides convenience so that in practice a wallet shows one balance and hides multiple coin objects under the hood.

**Consensus & Validators:** Sui uses delegated Proof-of-Stake. A set of validators run the network (there were 100 validators at launch, planned to scale). Each validator has a certain stake of SUI (self-staked + delegated). Epochs (24h or 48h) determine validator set and their voting power proportionate to stake. Gas fees are distributed to validators and their delegators as rewards, and there’s also an epoch staking reward in SUI (from an inflation schedule of SUI, although initial inflation might be low or zero, I’d have to confirm specifics; likely SUI has some epoch rewards, possibly from storage fund).

Sui does not currently shard (all validators execute all transactions, though the parallel execution model internally is there). They may scale through more complex sharding or simply rely on parallelism.

**Notable Ecosystem and Use Cases:** Being new, Sui’s ecosystem is small but growing. A highlight was a wave of Sui NFT mints and a few DeFi prototypes:

* **Cetus DEX** (an AMM on Sui, with its token CETUS)​

[coinmarketcap.com](https://coinmarketcap.com/view/sui-ecosystem/#:~:text=CoinMarketCap%20coinmarketcap,07)

,

* **Suiswap** (another DEX),
* **MovEX** (orderbook),
* Various NFT collections and marketplaces (e.g. BlueMove).
* **Sui Name Service (SNS)** offering human-readable .sui names mapped to addresses (these are NFT objects representing the name).

Sui’s object-centric approach suits games: for example, a game can represent characters and items as objects that users own and trade without involving global state each time (unless interacting in a shared environment).

**Interoperability:** Sui can connect to other chains via bridges; given Move’s novelty, dedicated bridges (Wormhole added support for Sui in 2023, bridging assets like USDC from Ethereum which then appear as objects in Sui). Sui itself does not yet have direct interoperability with Aptos or others, aside from via third-party bridges. Sui addresses share format with Aptos (both are 32-byte hex), but they are separate namespaces (an address on Sui doesn’t correspond to same user on Aptos, since the key derivation might differ and they’re different networks).

**Security:** Move’s security is a plus – assets cannot be duplicated or lost inadvertently because Move enforces linear types. The global storage is tracked such that if a resource is not in some account’s possession, it must be somewhere or was destroyed explicitly. Sui has a feature called **object locks** which ensures that when a transaction is executing, the input objects are locked to that transaction until it finalizes (preventing double-spend). If the transaction fails or doesn’t go through, those objects are unlocked.

**Address Example:**

* 0xbff3d3ec71d24f1b8b8e3d6b2fc2d1a5ca88f6e4f0c1c4e2c9b9b7a9c8d4e3f5 – a hypothetical Sui address (64 hex chars). In Sui explorer, they might show it as bff3d3ec...d4e3f5.
* Sui’s **Framework Package** (similar to Move stdlib) has an object ID (which is often 0x2 or something short representing that core package).
* The **SUI coin type** is denoted by object type UID 0x2::sui::SUI (in code, but this 0x2 is the address of the Sui framework, and sui::SUI struct inside it identifies the coin type).
* If Bob has 50 SUI, he might have an object with type 0x2::coin::Coin<0x2::sui::SUI> and id 0xabc...123. That object’s owner is Bob’s address.

**Aptos**

**Overview:** Aptos is another Layer-1 blockchain that emerged from Facebook’s Diem project, launched mainnet in Oct 2022. It shares some lineage with Sui (both use Move language) but has distinct architecture: Aptos aims for **high throughput via parallel execution** (Block-STM), **fast finality** (1-2 seconds), and a focus on user experience. Aptos uses a BFT consensus called AptosBFT (derived from HotStuff). In 2023, Aptos reached significant activity (e.g., up to 743k daily addresses​

[binance.com](https://www.binance.com/en/square/post/17124019107850#:~:text=)

, partly from airdrop farming and new apps). **APT** is the native token (1 billion initial supply, 8% annual staking rewards). Aptos positions itself as a safe and scalable base for DeFi and NFTs, with funding and community growing.

**Account Address & Authentication:** Aptos addresses, like Sui’s, are 32-byte values usually shown as hex strings (0x prefixed). **Aptos account addresses** can be derived from public keys but interestingly Aptos decouples the concept of an account from a public key slightly via an Authentication Key. By default, an Aptos account address = the 32-byte SHA3-256 of the public key (for single key accounts), which ends up usually just being the last 32 bytes of the pubkey (if pubkey is 32 bytes Ed25519, its hash is itself if no collisions – slight nuance: I think they actually do use the hash, so it’s possible the address differs from raw pubkey bytes but effectively similar). Aptos also allows **account key rotation** and **multisig**: the account’s authentication key can be a combination of multiple pubkeys (allowing multisig accounts without changing address). This means an account address is actually initially derived from one key, but the account can later rotate its public key. The address remains the same (because the address is not simply the pubkey bytes, it’s locked in at creation).

**Resource Model:** Like NEAR and Sui, Aptos has accounts that hold resources (Move’s model). Each Aptos account has a collection of Move resources (e.g., Coin store for each token type, etc.). Aptos’s Move modules can define resource types that get stored under accounts. For example, Aptos’s coin standard (APT and any other fungible token) uses a resource CoinStore<T> under the account to record balances of type T. This is different from Sui’s object-per-coin; Aptos uses a more traditional ledger style (account holds a balance resource with a number). This makes Aptos operate more like Ethereum (one account with a balance field) in terms of usage, even though it’s Move under the hood.

**Transactions & Throughput:** Aptos introduced **parallel execution** using Block-STM, meaning it executes transactions in parallel and uses conflict detection to commit those that don’t conflict. This requires the ability to know which resources each transaction will touch (which Move helps with to some degree). In testing, Aptos demonstrated very high TPS in ideal conditions (over 10k), though on mainnet practical TPS is lower (but still high relative to many L1s). The typical block/round time is ~1 second, and finality around 1-2 seconds after 2 rounds of consensus. Aptos’s transaction fee is paid in APT and uses a gas model (with gas units and a gas price). Currently, fees are low because gas price is minimal (not much congestion).

**Consensus:** AptosBFTv4 (in 2023) is their consensus algorithm. It’s a proof-of-stake BFT with ~100 validators at launch (could grow). Validators are chosen based on staked APT (anyone can delegate to validators to increase their stake weight). The consensus is similar to DiemBFT/HotStuff (3-phase commit). There’s no slashing currently on Aptos, but they plan to introduce slashing for misbehavior.

**Native Token (APT):** Aptos’s APT started with a supply of 1e9, with a significant portion allocated to community, foundation, investors, core team, unlocked gradually. Aptos has set 7% annual inflation for staking rewards (which can be adjusted via governance). APT has 8 decimal places (like 1 APT = 10^8 Octas in their units).

**Move and Contracts:** Aptos Move is similar to Sui Move in language basics but different on how it handles global storage. Aptos allows publishing Move modules to an account (like how Ethereum stores contract code at an address). For instance, if Bob publishes a Move module, that module lives at Bob’s account address (0xBob...). Aptos accounts can have a **ModuleId** which is just <account\_address>::<module\_name>. To call a function in that module, one sends a transaction script that invokes it. Aptos recently added an ability to schedule **transaction scripts** similarly to Ethereum transactions (script calls are packaged as entry functions in modules as well, IIRC at launch Aptos allowed arbitrary script calls if authorized by signers).

**Token Standard:** Aptos’s fungible token standard is basically an implementation of coins as Move resources. They provide an official module 0x1::coin that can be used to register a new coin type easily (with supply management). Many top Aptos tokens (like USDT on Aptos, USDC on Aptos, etc.) are issued via the official coin standard, meaning each user has a 0xAccount::coin::CoinStore<MyToken> resource for balance. To transfer, one calls 0x1::coin::transfer<MyToken>(to, amount) which moves the resource internally. There is no separate contract address per token like an ERC-20 – the coin type itself (struct defined in a Move module) is the “identifier”. However, for familiarity, Aptos governance gave each coin type a **unique 32-byte address ID** (the account that published it, plus struct tag), and explorers often show a “token address” which is basically a combination of publisher address and coin type name.

For example, Aptos’s own APT is the native coin (not just a Move resource but special in protocol). For a token like USDC (launched via LayerZero’s bridge), there’s a coin type presumably published at some address (say 0xAf...:USDC). The explorer might label that with a token address (maybe the publisher’s address).

**Governance:** Aptos has on-chain governance for protocol upgrades and parameters, controlled by APT token holders (though early on foundation likely had large influence). They also allow the community to propose changes via governance proposals (voting done by staked APT weight).

**Interoperability:** Aptos has multiple bridges: e.g. LayerZero, Wormhole, Celer, etc., bridging assets from Ethereum and other chains. Aptos and Sui, despite both using Move, are not directly interoperable (addresses and state are separate). But conceptually, if Synergy’s UMA was mapping addresses, an interesting note: Aptos addresses and Sui addresses are the same format (64-hex), so a UMA scheme could unify them with a prefix to indicate chain.

**Key & Address Example:** If an Aptos private key = Ed25519 (32-byte secret), public key is 32-byte, Aptos account address = sha3(pubkey) (32-byte). Example pubkey: 0x77f2...89ae -> address might be 0xccb45...ffe3. If the user rotates key, the address stays, and the new pubkey’s hash must match the originally recorded authentication\_key (which for single-key accounts is pubkey hash + 0x00 or 0x01 suffix to allow different scheme of address derivation if they choose; there is concept of auth key = pubkey hash + suffix for multi-sig). But for most users, just think address = initial pubkey hash.

**Parallel Execution Impact:** On Aptos, if two tx don’t conflict (e.g., Alice sends Bob APT, and Carol swaps on a DEX), they can execute simultaneously on different threads, making use of multi-core validators. If two tx do conflict (both try to swap on same DEX pool), one will be rolled back in STM and either re-executed or delayed behind the other. This is internal – from outside, it just appears as high throughput and similar final ordering as if sequential, but faster throughput.

**Notable Ecosystem Projects:**

* **Pontem** (DeFi hub, AMM Liquidswap, etc., with token PONT),
* **Anime Swap**, **Tsunami** (other DEXes),
* **Aux Exchange** (orderbook Dex),
* **Argo** (liquid staking APT -> stAPT),
* **Tortuga** (another staking),
* **Topaz** (NFT marketplace). Many of these have Move modules deployed at certain addresses. E.g., Tortuga’s TAPT token is a coin type at an address they published.

**Celo**

**Overview:** Celo is a mobile-first blockchain that’s EVM-compatible and PoS-based. It launched in 2020 with the goal of enabling crypto payments via phones, even allowing phone numbers to be mapped to addresses (through a lightweight identity protocol). Celo’s consensus is **Proof-of-Stake** using IBFT (Istanbul BFT) with a fixed validator set (~100 validators elected). Block time ~5 seconds. It features **platform-native stablecoins** (cUSD, cEUR, cREAL) which are stabilized via a reserve and algorithmic adjustments (Mento mechanism)​

[docs.celo.org](https://docs.celo.org/learn/platform-native-stablecoins-summary#:~:text=Platform,more%20easily%20on%20your)

. The native asset **CELO** is used for staking and governance. Celo emphasizes user experience: for example, users can receive payments to their phone number (which is hashed and mapped to their wallet address via a secure enclave on-chain, allowing invites etc.), and transaction fees can be paid in stablecoins (not just CELO) – a unique feature.

**Address Format:** Celo’s addresses are the same format as Ethereum (0x + 40 hex). *Example:* a Celo account: 0xeec8...933c. They did not invent a new format; being EVM, they kept compatibility. Phone number mapping doesn’t change addresses, it’s an upper-layer feature via a separate contract (the **Valora wallet** or others can hash a phone number and register with a mapping smart contract that associates it with a Celo address, allowing someone to find that mapping and send funds).

**Gas and Currency:** Uniquely, Celo allows paying gas fees not only in the native CELO but also in its stablecoins cUSD, cEUR (this is implemented by having the validators agree to accept those and the protocol translates that to equivalent fees). This means a user holding only cUSD can still pay for transactions, improving usability for non-technical users who might not acquire CELO. Gas prices are still in terms of “fraction of stablecoin” under the hood (or they set a gas price in an oracle).

**Consensus & Staking:** Celo uses a Byzantine Fault Tolerant consensus (IBFT variant) with 100 validators. Validators are elected by CELO holders (via a locked gold mechanism – CELO must be locked to vote for validators). The elected validators form a set for an epoch ( day) and produce blocks in a round-robin fashion (IBFT requires a supermajority to sign each block). Finality is immediate when a block is signed (no chain reorgs since it’s BFT, not longest-chain). CELO staking yield is around 5% (adjustable by on-chain governance).

**Stablecoins (cUSD, cEUR, cREAL):** These are Celo-native stable tokens. They are implemented as ERC-20 contracts but with special stabilization logic off-chain/on-chain:

* The **Mento** mechanism: Celo has a reserve (initially including BTC, ETH, and CELO itself) and uses an on-chain exchange where CELO can be exchanged for cUSD or cEUR at a price set by a moving peg (algorithm adjusts supply by open market operations). If demand for cUSD rises, users can send CELO to the reserve and mint cUSD (which expands supply), and vice versa to contract supply.
* These stablecoins are fully on-chain and the reserve is overcollateralized (Celo Foundation manages reserve assets).
* Each stablecoin is an ERC-20 token contract. cUSD’s contract is at a certain address (like 0x765DE816845861e75A25fCA122bb6898B8B1282a which was given as example earlier).
* The Celo platform uses these as first-class citizens; even governance proposals can ask something like adjusting cUSD stability parameters.

**Smart Contracts & Compatibility:** Celo is EVM, so any Ethereum contract can deploy. It has a DeFi ecosystem: Ubeswap (DEX like Uniswap), Moola (lending like Compound), and impact-focused projects (e.g. Grassroot Economics UBI tokens). Many Ethereum dApps adapted to Celo given easy porting.

**Identity and UX:** The **Celo Ledger** has an Identity contract that maps phone number hashes to addresses with attestations. How it works: Alice installs Valora wallet, inputs phone number. The app generates a cryptographic hash of her number and sends it to Celo’s Identity service. The network then requires Alice to verify control of that number via SMS codes (sent by 3 of Celo’s validators). Once she enters the codes, those validators attest on-chain that the hash corresponds to her address. Now if Bob has Alice’s phone number and uses Valora, it can find through the contract a matching address hash with attestations, so it suggests Alice’s wallet address to send payment. The actual phone number isn’t on-chain, only blinding of it, so it tries to preserve privacy to an extent (though one can attempt to brute-force small number spaces). This system lowers friction for onboarding.

**Governance:** Celo’s governance is on-chain with CELO (locked) used to vote. Proposals can range from adding a new stable asset, adjusting reserve target allocations, to technical upgrades. The governance process is somewhat similar to Compound’s (proposal, referendum, quorum, execute).

**Interoperability:** Celo can interoperate via bridges. Celo has a Bridge to Ethereum (Optics, now called Celo Bridge, which had an incident, I recall something with Optics hack). cUSD and cEUR exist as tokens on Ethereum too (as ERC-20 via bridge). Conversely, Ethereum assets like USDC, USDT have bridged versions on Celo (though ironically Celo’s own stable wants to reduce need for USDC, bridging still exists for compatibility). Also, Chainlink oracles are integrated to help with price feeds.

**Address Example Recap:** Celo addresses being 0x type means any example Ethereum address works. For instance, the Celo reserve smart contract might be at a certain 0x... address. A user’s CELO account might look identical to an Ethereum one. This synergy means Synergy’s meta-address concept could treat a Celo address just like Ethereum but perhaps with a prefix to denote chain (since 0xabc... could exist on many chains).

**Token Transfers and Phone number mapping example:** If Alice wants to send cUSD to Bob but only has his phone number:

* Alice enters Bob’s number, her wallet queries the identity registry. It finds mapping to an address (if Bob has set it up). It shows “Sending to 0x1234... (Bob)”.
* She sends 10 cUSD; the transaction is a standard ERC-20 transfer on Celo (to 0x1234...).
* If Bob hasn’t set up, the wallet might allow Alice to send an invite: basically escrow some funds to a temporary address and generate a code that Bob can use to claim by setting up a wallet (Valora has that flow: you can send funds to a phone number not yet on Celo, and the funds will be claimable when that user joins and verifies the number).

**Stable Value Fees:** Because you can pay fees in cUSD or cEUR, a user holding stable doesn’t worry about fluctuating gas costs. The network updates an on-chain variable for gas price for each currency (so that validators still ultimately earn in CELO equivalent or are comfortable with the value). It’s an interesting alternative to Ethereum’s pure ETH gas model.

**The Open Network (TON)**

**Overview:** TON was originally “Telegram Open Network,” designed by the Telegram team, later abandoned due to regulatory pressure and picked up by open-source community as “The Open Network.” It’s a PoS blockchain intended to integrate with the Telegram messenger (recently there’s been official support with Telegram enabling username auctions on TON etc.). TON uses a novel **Infinite sharding** architecture with **workchains** and **shardchains** (somewhat like ETH2 original plan or multi-chain structure), though currently only one workchain (base chain) is active. Consensus is based on a variant of BFT with validators rotating in rounds, and it uses catchain and vertical blockchains (it’s quite complex tech from Nikolai Durov’s design). In simpler terms, it’s a fast chain (under a minute finality, typically a few seconds for block generation). **Toncoin (TON)** is the native token, used for gas and staking. TON has gained popularity for Telegram integrated features, such as **TON DNS** (human-readable @ usernames for wallets), **TON Payments** (plans for off-chain payment channels), and a vibrant TON ecosystem including a DEX, NFT platforms, etc.

**Address Formats:** TON addresses are notably different from Ethereum. They are usually represented in **base64** with a special prefix, or as a human-readable @ handle:

* A full TON address binary is 256 bits, typically written in a format like EQDn7\_JK... (which is a base64url string starting with EQ or Ef depending on address type). This is called the **user-friendly address** format, which includes a checksum and network ID.
* TON addresses can also be represented as raw hex of 256 bits plus a workchain ID. For instance, -1:abcd1234... format (where -1 or 0 is the workchain, -1 for masterchain, 0 for base workchain).
* Usually, normal accounts are in workchain 0, so an address might be 0:abcd... (and often explorers show a base64 that encodes that).

For example, a TON wallet address might be given as EQC\_hVy...skH (characters like \_ and - can appear in the base64). That is the standard format used in wallets and Telegram’s UI.

**Public keys and addresses:** In TON, an account’s address can be derived from a public key and an optional smart contract. For a simple wallet, the address is basically a hash of the wallet’s initial code and public key. Many TON wallets are actually smart contracts (wallet contracts), not just public key accounts. So your TON address encapsulates the wallet code (which might enforce basic send functionality and multi-sig rules) plus your public key.

**Workchains and Shards:** TON was designed to have up to 2^32 workchains (like parallel blockchains) and each workchain can shard into up to 2^60 shards. Currently only Workchain 0 (general purpose one) is in use (and the masterchain, which is workchain -1 for coordination). Sharding in workchain 0 happens when load is high (so far, the network might not have needed much sharding, or at times there are a few shards). When active, an address includes a **shard prefix** implicitly in its high bits. But users don’t have to worry about that; routing transactions to addresses finds the right shard automatically.

**Gas and Fees:** TON uses a gas and coin system similar to Ethereum’s in concept. Transaction fees are paid in TON (also called ever-scale sometimes, but in TON they just say “nanograms” as smallest unit, 1 TON = 1e9 nanoTON). Gas is measured and there's a gas price (set by validators or protocol). However, because TON is designed to handle millions of tx per second in future, its fee mechanism is complex and involves per-shard compute budgets. In practice, current TON fees are extremely low (like 0.1 TON or much less for a basic transfer, which is a few cents).

**Smart Contracts:** TON uses a bespoke smart contract language called **Fift** (and more recently high-level languages like **FunC** and **TON Solidity** which compile to TON Virtual Machine bytecode). Smart contracts in TON are very flexible (you can even run custom code on message receipt to decide fees). They persist data in a cell-tree structure (TLB scheme). For developers, FunC is commonly used now (especially for building things like Jettons or NFT contracts).

**Token Standard (Jettons):** TON’s equivalent of ERC-20 tokens are called **Jettons**​

[docs.ton.org](https://docs.ton.org/v3/guidelines/dapps/asset-processing/jettons#:~:text=Best%20Practices%20on%20Jettons%20Processing%E2%80%8B,20%20tokens%20on%20Ethereum)

. A Jetton is essentially a pair of smart contracts: a **Jetton Minter** (holds metadata, total supply, mint/burn logic) and multiple **Jetton Wallet** contracts (each user who holds the token deploys or is assigned a jetton wallet contract that tracks their balance). When someone transfers jettons to another, their jetton wallet contract sends a message to the recipient’s jetton wallet contract (creating it if it didn’t exist) with the new balance. It’s a bit more involved than just balance in storage, but it allows each user to have a contract that they fully control which contains their tokens (and can add custom receive logic). This is partly because TON doesn’t do dynamic storage per address elegantly; they prefer separate contract instances.

Jetton addresses: The minter has a normal address (for example, USDT’s minter on TON has an address, which is known – and as mentioned, TON optimized the USDT contract code to reduce fees​

[docs.ton.org](https://docs.ton.org/v3/documentation/dapps/assets/usdt#:~:text=USDT%20Processing%20,Faster%20and%20scalable%E2%80%8B)

). Each user’s jetton wallet also has an address (derived from the minter and user’s own address, usually).

**TON NFT Standard:** Similar approach: each NFT is a contract (or sometimes a collection contract and individual item contracts). TON has TON DNS and TON Site as special cases of NFTs where owning the NFT = owning a domain or site content.

**Consensus & Validators:** TON’s current validators are community-run (some by FreeTON initial supporters). They rotate every election (~hourly) and must stake TON to be in. The total validators count is in the hundreds. TON uses a variant of BFT where a small subset of validators validate each shardchain and masterchain. The masterchain (like Polkadot’s relay chain) contains references to all shard blocks. Finality is a bit intricate, involving a catch-up process. But typically, one can consider final in a couple of blocks (maybe ~block every 5 seconds, final in < minute on average conditions).

**Integration with Telegram:** Recently, Telegram launched the **TON DNS auction** for @ usernames (e.g., @alice could be auctioned as a TON domain NFT and then used as a username in Telegram). Payment for these auctions happens in TON through a bot. This spiked TON usage and price in late 2022. They may integrate TON wallet features natively in Telegram in future (currently a separate Wallet bot exists).

**Special Features:**

* **Atomic swaps** and mini-exchange built-in proposals were in original whitepaper.
* **TON Payments**: micropayment channels for instant off-chain transactions (not sure if launched yet).
* TON has extremely efficient serialization (the TL-BC) making data small on chain, and they have “vertical blockchain” concept where each validator keeps its own local series of votes that are hashed in masterchain for accountability.

**User-friendly aspects:** Because addresses are long and weird, TON users often use **TON DNS** short names (like alice.ton) pointing to addresses. Or they share QR codes/links. The wallet experience is improving with apps like Tonkeeper, Tonhub.

**Example Address and Transfer:**

* Alice’s wallet contract address: EQCxh...
* Bob’s address: EQB8...
* If Alice sends 5 TON to Bob, her wallet contract will create an external message with 5 TON and send to Bob’s address. The network routes it to the shard containing Bob’s address. Bob’s account (if just a basic wallet or even if not yet initialized) will receive the funds (if it doesn’t exist, TON accounts are virtual—they “exist” when they have a balance or code, so Bob’s address becomes active once funds arrive).
* If Bob’s address corresponds to a smart contract with custom on\_receive, that code executes (for a wallet, typically just adds to balance and can emit an event).

**Security model:** The multi-shard aspect means the network can scale but is also complex to secure. If a validator misbehaves on a shardchain, the masterchain should detect discrepancy and slash it. Slashing exists in TON for misbehavior or equivocation.

**Polygon (PoS Chain)**

*(See* ***Top Network #10*** *above for core info; here we add more structure matching Solana/Synergy)*

**Overview:** Polygon PoS is a sidechain to Ethereum that provides faster and cheaper transactions using a PoS checkpointing mechanism. It’s built on a three-layer architecture: Ethereum mainnet (for final checkpoints and staking contracts), the Bor sidechain (produces blocks ~every 2 seconds), and Tendermint-based Heimdall (aggregates and submits checkpoints to Ethereum). This network became very popular for DeFi and NFTs that needed low fees, effectively acting as an “Ethereum-compatible network” (often called commit chain). Native token **MATIC** is used for staking and gas. Polygon aims to be an entire suite of scaling solutions (PoS, zkRollups, etc.), but the PoS chain is its flagship with widespread use (OpenSea integration, Aave, etc.).

**Address & Key Format:** Polygon PoS uses Ethereum addresses and keys. Users can use the same public/private key on Polygon as on Ethereum. **Example:** If a user’s Ethereum address is 0xABCD...1234, their Polygon address is identically 0xABCD...1234. This makes user experience seamless (no need to manage separate key pairs).

**Consensus & Security:** Polygon’s validator set (currently 100+ validators) stakes MATIC on Ethereum’s staking contract to become validators on the sidechain. They run Heimdall (a Tendermint consensus layer) which elects proposers for Bor (block production layer). **Bor blocks** – produced every ~2 seconds – are grouped into a span, and at the end of each span (like every 5 minutes or so), a checkpoint (containing Merkle roots of sidechain blocks) is submitted to Ethereum by validators. These checkpoints finalize the sidechain blocks on Ethereum. If sidechain blocks conflict or a malicious fork is made, only the branch whose checkpoints get submitted and validated on Ethereum is considered valid (others could be slashed if they attempted a bad checkpoint). This gives Polygon PoS stronger security than an isolated sidechain, though not as strong as a true rollup (since user funds still rely on the validator honesty between checkpoints).

**Native Token (MATIC) and Gas:** MATIC is used in two ways:

* **Staking:** Validators stake MATIC (and delegators can delegate to them) to earn a share of transaction fees and some rewards.
* **Gas fees:** All transactions on Polygon PoS require MATIC for gas (very small amounts; e.g., transferring tokens might cost 0.001 MATIC, where MATIC is ~$1, so negligible).

MATIC has a fixed max supply (10 billion, with a portion released as staking rewards until ~2025). Gas on Polygon works like Ethereum’s EIP-1559 now (they implemented base fee burn on Polygon). Indeed, MATIC fees have a base fee that gets burned, making MATIC slightly deflationary when usage is high.

**Tokens and Smart Contracts:** Polygon supports **ERC-20, ERC-721, etc.** Many Ethereum projects deployed clones on Polygon (sometimes with same addresses for contracts via deterministic deployment). **Bridged assets:** A lot of tokens on Polygon are bridged from Ethereum. For example, USDC on Polygon is a bridged token managed by a Polygon-Ethereum Plasma Bridge contract; its contract address on Polygon is 0x2791Bca... and it’s pegged 1:1 to USDC on Ethereum held in a bridge. Polygon’s canonical bridge (PoS bridge) allows arbitrary token bridging (users deposit tokens to an Ethereum contract, which credits them on Polygon via a predicate contract logic, and vice versa for withdrawals, using the checkpoints as finality proof).

Additionally, there are **Polygon-native tokens** (like QuickSwap’s QUICK, which originally only lived on Polygon, or GENSOKISHI’s MV). They follow ERC standards and have contract addresses on Polygon.

**Cross-chain Interactions:** To move assets from Polygon to Ethereum, users initiate a withdrawal, which locks/burns tokens on Polygon and generates a proof, after the checkpoint they can use that proof on Ethereum to release the original tokens. This process takes a checkpoint interval (typically ~7-8 minutes plus some buffer).

**Polygon’s Future (POL token mention):** Polygon PoS is planned to integrate into Polygon 2.0 which will use a new token (POL) replacing MATIC, and possibly transform the architecture to a validium or something akin to more secure. For now it’s the sidechain as described.

**Key Management & UI:** Because it’s EVM, users typically use MetaMask or similar wallets. Interacting with Polygon often means just switching network in MetaMask – all the addresses and keys remain same. This ease contributed to adoption (no need to manage new seed phrase or wallet, just add RPC URL for Polygon).

**Notable DApps:** QuickSwap (Uniswap fork), SushiSwap deployed, Aave deployed on Polygon, Curve, and various NFT games (Sandbox moved land NFTs to Polygon, Reddit issued Community Points as ERC-20 on Polygon, Instagram and NFL NFTs launched on Polygon). This made Polygon busy in daily transactions (it often did 3-4 million tx/day in 2021).

**Governance:** Initially Polygon’s PoS chain didn’t have on-chain governance; upgrades and validator changes were coordinated by the team and validators. They are moving to more formal governance with the introduction of Polygon Improvement Proposals (PIPs) and community votes (most recently, the decision to upgrade to zkEVM-based tech or token changes was done via off-chain voting of token holders).

**Synergy cross-chain addressing:** A user’s Polygon address being same as Ethereum’s means a cross-chain UMA could unify them under one meta-address (with chain context). But Synergy’s approach might differentiate if needed by prefix (like sYnE for Ethereum, sYnP for Polygon, etc.), unless a more clever universal scheme is used.

**Bitcoin**

*(See* ***Top Network #11*** *above for details, here structured with headings.)*

**Overview:** Bitcoin is the first decentralized blockchain (launched 2009) focusing on secure, censorship-resistant store-of-value and transfer. It uses **Proof-of-Work (SHA-256)** mining for consensus, producing a block approximately every 10 minutes. Bitcoin’s design prioritizes security and decentralization over throughput (it handles ~7 transactions per second on-chain). The supply of **BTC** is capped at 21 million (currently ~19.5M mined), and a halving event every 4 years reduces the mining block reward, making BTC deflationary in issuance. Bitcoin’s simple scripting system allows basic smart contracts (multisig, timelocks) but not complex dApps. However, additional layers (Lightning, sidechains) extend its functionality. Over the years, Bitcoin remains the most valuable crypto network by market cap and is often seen as “digital gold.”

**Address & Key Formats:** Bitcoin uses UTXO (unspent transaction output) model, and addresses represent scripts to which value can be locked. The main address formats in use​

[cointelegraph.com](https://cointelegraph.com/learn/articles/types-of-bitcoin-addresses#:~:text=A%20Bitcoin%20address%20is%20an,throughout%20the%20decentralized%20network)

:

* **P2PKH (Legacy):** Addresses starting with 1. These are derived from a public key hashed with RIPEMD160(SHA256(pubkey)). Example: 1BoatSLRHtKNngkdXEeobR76b53LETtpyT is a P2PKH address. When spending, you provide a signature and the public key, and script verifies hash matches address and signature matches pubkey.
* **P2SH:** Addresses starting with 3. These represent a hashed redeem script (could be multisig or other script). For example, 3QJmV3... might correspond to a 2-of-3 multisig script hash. When spending, you provide the redeem script and the satisfying data (sigs).
* **Bech32 (SegWit):** Modern addresses starting with bc1 (for mainnet)​

[help.revolut.com](https://help.revolut.com/help/wealth/cryptocurrencies/transferring-cryptocurrencies/depositing-cryptocurrencies/issues-with-crypto-deposits/which-addresses-can-i-revert-my-deposit-to/#:~:text=Bitcoin%20)

. SegWit v0 addresses (P2WPKH) start with bc1q and are 42 characters long. Example: bc1qw508d6qejxtdg4y5r3zarvaryvg6kdajk0mssh. These encode either a witness public key hash or script hash directly in Bech32 (case-insensitive). Taproot (SegWit v1, for Schnorr/Taproot outputs) addresses also start with bc1p. Example: bc1p5cyxnuxmeuft... (longer).

The **private key** in Bitcoin is a 32-byte number. Typically encoded in WIF (Wallet Import Format) starting with 5 or K/L if compressed pubkey. The public key is a 33-byte (compressed) or 65-byte (uncompressed) point on secp256k1 curve. Standard wallets use compressed pubkeys now (leading to addresses as above). A Bitcoin wallet typically has a **seed phrase (BIP39)** from which it derives many keys (BIP32 HD wallet). So one wallet = many addresses.

**Transaction Structure:** A Bitcoin transaction consumes UTXOs and creates new UTXOs. Each input references a previous output and provides an unlocking script (witness for segwit). Each output has a locking script (scriptPubKey) which corresponds to an address. The sum of inputs minus outputs is the fee (paid to miner). Transactions are identified by a TXID (double SHA256 hash of the serialized tx data). SegWit introduced a separate TxID vs WITNESS-TxID concept, but commonly the TXID (without witness data) is used as transaction identifier. TXIDs are 32-byte and shown as hex in block explorers (e.g. e2e0…b3af). Confirmations measure how many blocks have been mined on top of a given transaction’s block.

**Block and Consensus:** Miners assemble transactions into a block (max ~4MB weight, ~1-2MB actual size). They compute a SHA-256 proof-of-work, finding a hash below a target. Current difficulty makes this extremely hard, requiring enormous hash power (~10^20 hashes per block). About every 10 minutes, one miner finds a block and broadcasts it. Nodes validate it (checking all tx are valid, UTXOs exist, signatures correct, block reward correct, and PoW meets difficulty). If valid, it’s added to their chain. Longest chain rule (technically most accumulated work) determines consensus. This means finality is probabilistic; after 6 confirmations (~1 hour), probability of reorg is negligible for practical purposes.

**Security and Incentives:** Block reward = 6.25 BTC (as of 2023) + fees. This halves to 3.125 BTC in 2024 halving. Eventually block reward will be tiny and fees must sustain miners. Bitcoin’s economic security relies on the huge energy expended by miners, making attacks extremely expensive. It’s often said with >$10 billion annualized mining cost, reversing transactions or double-spending is infeasible beyond trivial amounts, as an attacker would need comparable expenditure.

**Scripts and Smart Contracts:** Bitcoin script is a stack-based, stateless script limited purposely to avoid loops. It has opcodes for math, logic, crypto. Standard scripts are:

* Pay-to-PubKeyHash (as mentioned),
* Pay-to-ScriptHash (encapsulating custom scripts),
* Pay-to-Witness-PubKeyHash (segwit v0),
* Pay-to-Witness-ScriptHash (segwit v0 for e.g. multi-sig in segwit),
* Taproot (v1 segwit) which allows complex tree of scripts or single-key spending using Schnorr signatures.

Bitcoin can do things like timelocks (CheckLockTimeVerify), hashlocks (for atomic swaps across chains e.g. HTLCs for Lightning), multisignature (n-of-m via OP\_CHECKMULTISIG). However, it cannot do arbitrary loops or stateful contracts like Ethereum – each UTXO is independent. This limits DeFi or complex dApps on L1. Instead, people use L2 or sidechains (e.g., Lightning for fast payments, RSK sidechain for EVM smart contracts in BTC realm).

**Lightning Network (L2):** A network of payment channels built on Bitcoin (Lock funds in a multi-sig 2-of-2 output with a peer, then do off-chain signed updates, eventually settle net). Lightning greatly increases throughput for small payments and is widely used for fast transactions (like El Salvador’s Chivo uses Lightning for daily BTC payments).

**Notable Upgrades & Forks:** Bitcoin’s development is conservative. SegWit in 2017 improved capacity and malleability. Taproot in 2021 improved privacy and allowed more flexible smart contracts (via Musig2 and script trees). There’s emerging use of **ordinal inscriptions** (writing data like NFTs in witness field, enabling BRC-20 tokens etc.), though this is a user innovation not originally intended, causing debate about block space usage.

**Governance:** No on-chain governance. Changes to Bitcoin come through the BIP process and widespread agreement among node operators and miners. It’s often said Bitcoin governance is “rough consensus” + users ultimately decide which software to run (e.g., UASF in SegWit activation).

**Interoperability:** Bitcoin doesn’t natively bridge, but many projects create wrapped BTC on other chains (WBTC on Ethereum, etc.) so BTC liquidity is represented elsewhere. Conversely, sidechains like Liquid or RSK allow moving BTC via federations or pegs. But those require trust in federations or extra validators. Bitcoin’s own mainnet stays fairly isolated by design.

**Privacy:** Base Bitcoin is pseudonymous. There’s active development on privacy (Taproot helps a bit when combined with Lightning or coinjoin techniques). CoinJoins (mixing multiple users’ UTXOs in one tx) are used via wallets like Wasabi or Samurai to improve fungibility. Future possible upgrades: Schnorr + Taproot already in, maybe future cross-input signature aggregation or even drivechains (sidechain mechanism) or covenants (BIP119 like features) under discussion.

**Example Use:** If Alice wants to pay Bob 0.01 BTC:

* Alice’s wallet finds one of her UTXOs with at least 0.0101 BTC (assuming ~0.0001 fee).
* Creates a tx: Input (Alice’s UTXO, with her signature) -> Output (0.01 BTC to Bob’s address) + Output (change back to Alice, 0.000? BTC).
* She signs it and broadcasts. Miners include it, Bob sees 1 confirmation ~10 min later. For safety, Bob might wait 6 conf (~1 hour) for finality if it’s a large amount.

**Synergy Perspective:** Bitcoin addresses are unique (starting 1,3,bc1, etc.), so a Synergy UMA could incorporate those by prefix or recognition. The Synergy doc mentioned mapping Synergy addresses to external like Bitcoin. Likely it would maintain a directory linking a Synergy UMA to a specific Bitcoin address (via FROST threshold sig maybe). As such, understanding these formats is key to compatibility.