The native token and gas

Native token

A native token is the blockchain's built-in currency, hardcoded into the protocol itself. On Solana, that's SOL. It exists at the base layer — you don't need a smart contract to create or transfer it. Every validator, every node, every piece of the network infrastructure understands SOL natively. It's used for paying transaction fees, staking to secure the network, and as the fundamental unit of account on the chain.

How it differs from a custom token

Custom tokens (like USDC on Solana, or any SPL token) are created by deploying a program (smart contract) on top of the network. They only exist because a program defines their rules — supply, decimals, transfer logic, etc. The key differences: the network itself doesn't inherently "know" about custom tokens the way it knows about SOL. Transferring SOL is a system-level operation, while transferring a custom token requires invoking the SPL Token Program. Custom tokens also can't be used to pay for transaction fees — you always need SOL for that.

Gas

Every blockchain is essentially a shared computer. When you submit a transaction, you're asking every validator to execute some computation — verify signatures, update account states, run program logic. Without a cost mechanism, anyone could spam the network with infinite transactions and grind it to a halt. Gas is the pricing mechanism that prevents this. It makes every computation cost something, so there's always an economic disincentive to abuse the network.

Gas fee on Solana

1. Base Fee (fixed and predictable) Every signature on a transaction costs a flat 5,000 lamports (0.000005 SOL). Most transactions have one signature, so the base cost is 5,000 lamports. This doesn't change with network congestion — it's static and protocol-defined.

2. Compute Units (Solana's version of gas units) Every transaction gets a compute unit budget — the default is 200,000 compute units, and the maximum is 1.4 million. Each instruction your transaction executes (math operations, account reads/writes, cross-program invocations) consumes compute units. If your transaction exceeds its budget, it fails. You can request a specific compute unit limit using the `SetComputeUnitLimit` instruction, which is good practice because it tells the scheduler exactly how heavy your transaction is.

3. Priority Fee (optional, market-driven) This is where things get interesting. You can attach a priority fee priced in **micro-lamports per compute unit**. So if your transaction uses 200,000 compute units and you set a priority fee of 1 micro-lamport per compute unit, you pay an additional 200,000 micro-lamports (2 lamports). During high-demand periods — say a popular NFT mint or a token launch — you raise this to get your transaction processed faster. This is the closest Solana gets to Ethereum's gas price bidding, but it's scoped much more narrowly.

4. Where the fees go 50% of the total fee (base + priority) is burned, permanently removing that SOL from circulation. The other 50% goes to the validator that processed the transaction. The burn mechanism creates a mild deflationary pressure on SOL supply over time.

5. Why Solana fees stay low The fundamental reason is throughput. Ethereum processes roughly 15–30 transactions

per second, so block space is scarce and fees spike. Solana targets thousands of transactions per second, so there's far more supply of block space relative to demand.