A — Fit to Entropy V1 (short)

- Sidebets (social, on-demand markets spawned from Telegram) **is implementable** on Monad using the Entropy contract model (Factory → Instance → BalanceManager/Treasury).
- **Oracles:** Use Chainlink Any-API / Any-API + decentralized data feeds for outcome verification. Chainlink has tooling for Any-API and has been brought to Monad (testnet/mainnet availability). (Chainlink Documentation)
- **Wallet friction:** you can provide both wallet-required flows (MetaMask / WalletConnect) and near-walletless experience via Account-Abstraction + Paymaster (gasless sponsorship) or via custodial/third-party payment gateways. See MetaMask / Pimlico gasless guide & AA patterns. (MetaMask)
- **Telegram integration:** Bot spins up market + posts deep link to web page; users join via web wallet or gasless flow; bot receives on-chain events via webhook. This flow is proven and common in web3 bot patterns. (CoinsBench)

B — Architecture (diagram + components)

Below is a single coherent architecture. The **SidebetFactory** is a Solidity contract that creates per-market SidebetContracts. Each market resolves via oracle adapters (Chainlink Any-API or decentralized resolvers). Off-chain components include a Bot Service, Backend API / Indexer, Web UI, and optional Paymaster.

```
flowchart LR
  subgraph Users
    TG[Telegram User]
    Web[Web User (Next.js)]
    Wallet[Wallet (MetaMask / WalletConnect / AA)]
  end

  subgraph SocialLayer
    Bot[Telegram Bot (telegraf)]
  end

  subgraph Frontend
    Website[Next.js UI]
    ChatSvc[Chat Service (WebSocket/Firebase)]
  end
```

```
subgraph Backend
  API[Sidebets API (Node/Express)]
  Indexer[Event Listener / Indexer]
  Resolver[Resolution Orchestrator]
  Paymaster[Paymaster (optional gasless)]
end

subgraph Onchain
  Factory[SidebetFactory]
  SB[SidebetContract Instance]
  BalanceMgr[EntropyBalanceManager]
  Treasury[EntropyTreasury]
  Oracle[Chainlink AnyAPI / DataFeed]
end

TG -->|/sidebet 100 USDC ...| Bot
Bot -->|POST createMarket| API
API -->|tx createMarket()| Factory
Factory -->|emits MarketCreated| Indexer
Indexer --> API
API --> Website
Website --> Wallet
Wallet -->|joinMarket() tx| SB
Website --> ChatSvc
ChatSvc --> Users
Resolver -->|request data| Oracle
Oracle -->|callback| SB
SB -->|payouts| BalanceMgr
BalanceMgr --> Treasury
Paymaster -->|sponsors txs| Wallet
```

C — Implementation details (practical, copy-pasteable plan)

I. **Contracts — overview & skeletons**

1. **SidebetFactory (Solidity)**
    - Deploys new `Sidebet` instance with parameters (question text, deadline, stake token, min/max, dispute window, oracle type).

    - Stores list of created markets.
    - Uses `create2` for readable addresses (optional).
2. **Sidebet (per market)**
    - State: creator, token (ERC20), yesPool, noPool, deadline, resolved, outcome, resolutionWindow, oracleRequestId.
    - Methods:
        - `join(bool side, uint256 amount)` — transfers token to contract, updates balances (call BalanceManager).
        - `withdraw()` — withdraw after resolution.
        - `requestResolution()` — callable after deadline; triggers oracle request (Chainlink Any-API or decentralized aggregator).
        - `oracleCallback(bytes32 id, bool result)` — receives resolution; sets outcome, schedules payout.
        - `dispute()` — optional to open dispute window where human arbitrator / DNA protocol can intervene.
    - Security: ReentrancyGuard, Pausable, access control for factory / oracle callbacks.
3. **EntropyBalanceManager / Treasury**
    - Use your BalanceManager design (principal/yield/risk). For Sidebets, funds go to `risk` buckets while settled funds are moved to winners.
    - **Important**: SidebetService should not be allowed to mint tokens or change `principal` directly.
4. **OracleConsumer (Chainlink)**
    - Example pattern: Chainlink AnyAPI request + fulfill function that is `onlyOracle`.
    - For decentralized verification, require aggregate responses or use Chainlink Automation / Offchain Reporting.

Minimal Sidebet Solidity skeleton (abridged):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

interface IChainlinkOracle {
  function requestBool(bytes calldata params) external returns (bytes32);
}

contract Sidebet is ReentrancyGuard {
  IERC20 public token;
  address public factory;
  uint256 public deadline;
```

```
bool public resolved;
bool public result; // true == YES
mapping(address => uint256) public yesBalance;
mapping(address => uint256) public noBalance;
uint256 public yesTotal;
uint256 public noTotal;
bytes32 public oracleRequestId;

modifier onlyFactory() { require(msg.sender == factory); _; }

constructor(address _token, uint256 _deadline, address _factory) {
  token = IERC20(_token);
  deadline = _deadline;
  factory = _factory;
}

function join(bool side, uint256 amount) external nonReentrant {
  require(block.timestamp < deadline, "market closed");
  token.transferFrom(msg.sender, address(this), amount);
  if (side) { yesBalance[msg.sender] += amount; yesTotal += amount; }
  else { noBalance[msg.sender] += amount; noTotal += amount; }
  // also call BalanceManager if integrated with user ledger
}

function requestResolution(bytes calldata oracleParams) external {
  require(block.timestamp >= deadline, "too early");
  require(!resolved, "already");
  // call Oracle via Factory or OracleAdapter
  // oracleRequestId = oracle.requestBool(oracleParams);
}

// called by oracle node
function fulfill(bytes32 requestId, bool _result) external {
  require(!resolved, "already");
  // verify sender is oracle
  resolved = true;
  result = _result;
  // pay winners or move to BalanceManager
}

function withdraw() external nonReentrant {
  require(resolved, "not resolved");
  uint256 payout;
  if (result) { payout = yesBalance[msg.sender] * (yesTotal + noTotal) / yesTotal; }
```

```
    else { payout = noBalance[msg.sender] * (yesTotal + noTotal) / noTotal; }
    // zero balances before transfer
    if (payout > 0) {
      // move funds to BalanceManager or transfer
      token.transfer(msg.sender, payout);
    }
  }
}
```

Notes: This is a minimal skeleton. Production code must handle fee cuts, edge cases (zero pool), and rounding.

II. **Oracle integration & verification**

Options (ranked by decentralization / reliability):

1. **Chainlink Any-API / Any-API + Off-Chain Reporting** (recommended)
   - Advantage: Chainlink provides decentralized request/response; Any-API lets nodes call a web API you define to validate an event (e.g., check Twitter, news, or a trusted aggregator). Chainlink is available on Monad. (Chainlink Documentation)
   - Implementation:
     - The Sidebet triggers `requestResolution(params)` which emits an oracle request.
     - Chainlink node picks job, calls external REST endpoint (your verifier service), and calls `fulfill`.
     - Your verifier service should implement multi-source checks (Twitter API, web scraping, NFT/timestamped evidence) and return canonical boolean.
2. **Multi-oracle aggregation (safer)**
   - Hit 3+ oracle providers (Chainlink, Tellor, API3) and require >2/3 consensus. Tellor & API3 provide guides for integration. (docs.tellor.io)
3. **Human arbitrator fallback**
   - If data is contested, send to human panel (Kleros style) with dispute bond.

Practical job: For “Trump tweets tomorrow”:

- Your verifier service queries:
  - Official Twitter API (X) via enterprise endpoints or recent API (may require paid keys).
  - Cross-checks via archived sources (e.g., news outlets) and screenshots with signatures.
- Chainlink node calls your verifier endpoint; verifier returns `{ "result": true }`.

Important: Twitter/X API access is rate-limited and monetized; build fallback multi-source and dispute options.

Sources: Chainlink Any-API docs. (Chainlink Documentation)

III. **Telegram Bot + UX**

1. **Bot responsibilities**
    - Parse `/sidebet` commands.
    - Call Sidebets API to create market (sends create tx or instructs user to fund).
    - Post market card with deep link to website: `https://entropy.example.com/market/<addr>`
    - Provide quick join buttons:
        - If user has wallet connected via WalletConnect in Telegram WebView: deep link opens the web page with `?ref=tg&user=telegramId`.
        - If gasless is enabled: bot displays a one-click payment link that opens a hosted payment page (custodial) or WebAuth flow.
2. **Bot implementation (node)** — use `telegraf`:

```
const { Telegraf } = require('telegraf');
const bot = new Telegraf(process.env.BOT_TOKEN);

bot.command('sidebet', async (ctx) => {
  const args = parseArgs(ctx.message.text);
  // call backend to create pending market
  const { txLink, marketUrl } = await apiCreateMarket(args, ctx.from.id);
  await ctx.replyWithMarkdown(`Market created: [Open](${marketUrl})\nJoin: ${txLink}`);
});
```

Sources & patterns: coinsbench example and bot-to-dapp bridging. (CoinsBench)

3. **Deep linking & web view**
    - Telegram supports opening links in browser or WebView.
    - The webpage must accept query parameters and render a sign/tx flow.
    - Use `walletconnect`/`viem`/`wagmi` for wallet connection.

IV. **Walletless / Gasless UX options**

You have three realistic approaches:

1. **Account Abstraction + Paymaster (gasless)** — recommended for best UX
    - Users create a smart-contract wallet (social login via Web3Auth) and sign intents; Paymaster (sponsored by your Paymaster contract) pays gas. MetaMask docs and Pimlico show examples. (MetaMask)
    - Implementation steps:
        - Integrate Web3Auth / Magic for initial key bootstrap.
        - Deploy paymaster; fund it with MON/ETH for gas.
        - User creates SCW, then uses SCW to pay for market joins.
2. **Custodial micro-wallet / off-ramp** (fastest)
    - Use a payment gateway (OnchainPay or similar) to accept credit card / stablecoin in Telegram and credit an off-chain balance to a user account. When they want onchain withdrawal they KYC. Faster onboarding but centralized and KYC/AML burdens. (onchainpay.io)
3. **Lightweight custodial relay**
    - Bot collects signature & relays transactions via your relayer that pays gas; user later withdraws via onchain signed intent.

Tradeoffs:

- **AA Paymaster**: good decentralization, more infra.
- **Custodial**: fastest, regulatory friction.
  Choose per product risk appetite.

V. **Chat on the market page**

- Use a simple chat service:
    - **Option A:** Firebase Realtime DB / Firestore (fast, cheap) — good for MVP.
    - **Option B:** WebSocket server (Node + Redis) if you want fine control and moderation.
- Link chat room to market address. Only users who have placed a bet can be allowed to chat (enforce by reading wallet signature or token gating).
- Example: On join, store `userAddress` + `marketId` in your DB and issue a signed session token to the web chat.

VI. **Indexing / Event listening**

- Run an **Indexer** service:
    - Subscribes to `MarketCreated`, `BetPlaced`, `Resolved`, `Payout` events.
    - Populates DB for UI pages, bot updates, and notifications.
- Use provider (Alchemy / RPC) and a small DB (Postgres).
- This service also notifies the Telegram bot to post resolution updates.

VII. **Dispute & finality model**

- **On resolution:** mark `resolved = true` once oracle callback confirmed and no dispute raised within `disputeWindow` (e.g., 24 hours).
- **Dispute paths:**
  - Allow creator / disputers to post bond and open arbitration.
  - If computation or oracle is unreliable, human arbitration as fallback.

VIII. **Security & anti-abuse**

- **Market definitions must be precise** (time bounds, interpretation rules). Avoid ambiguous text.
- **Anti-sybil onboarding**: rate limit the bot and markets.
- **Front-running**: prevent frontrunning by disallowing owner calls to change outcome, and by using oracle callbacks instead of trusting on-chain data like `blockhash`.
- **Oracle liveness**: set timeout path to let market be resolved via human arbitration if oracle fails.
- **Escrow & solvency**: require market creator to post initial liquidity or use treasury collateralization model and display solvency metrics on UI.

D — Required integrations & verification (sources)

- **Chainlink Any-API** (use for fetching/verifying off-chain boolean results). Docs: Chainlink Any-API. (Chainlink Documentation)
- **Chainlink on Monad:** Chainlink Data Feeds and Any-API support exist on Monad testnet/mainnet; Chainlink Data Feeds were added to Monad testnet and then mainnet. Use Chainlink nodes or work with Chainlink to create jobs. (dev.chain.link)
- **Integration alternatives:** Tellor / API3 provide guides for adding support to new EVM chains. Useful if you want multi-oracle redundancy. (docs.tellor.io)
- **Telegram bot → DApp bridging patterns**: examples and guides (bot posts link with query params; web page handles tx and notifies bot via webhook). (CoinsBench)
- **Gasless / AA patterns**: MetaMask + Pimlico guide; Base docs on paymaster patterns show examples for implementing sponsored txs. (MetaMask)
- **Payment gateways / custodial** (optional): OnchainPay example for accepting payments if you want fiat/credit card integration. (onchainpay.io)

E — Concrete next steps & deliverables (what I can give now)

If you want, I will immediately (pick one):

1. Build **complete Solidity contracts** for `SidebetFactory + Sidebet` + Chainlink adapter (audit-ready skeleton).
2. Produce **Node.js bot + API** reference code (Telegraf + Express) and full webhook flows.
3. Implement **Next.js market page** (connect wallet, join, chat embed, view events).
4. Create **AA Paymaster + Web3Auth integration** example (gasless join flow).

Tell me which of the 4 you want first and I'll produce code + deployment steps. (No waiting — I'll implement in this reply.)

F — Must-fixs for your Entropy doc (summary)

- Replace "staking-backed yield" with **treasury-backed yield or external liquid staking integration plan** (staking direct to validators from dApp not possible today).
- Treat **commit-reveal as primary randomness** unless you onboard a verified on-chain VRF provider (Chainlink VRF is an option; Chainlink docs explain VRF patterns). (Chainlink Documentation)

If you want me to **start implementing code now**, say which of the Deliverables (1–4) to produce first. I'll generate the contract code / bot code / frontend skeleton + deploy/test instructions in the next message.