This blueprint provides a comprehensive, production-oriented, and ethically-segregated framework for developing an MEV searcher tool for academic research, ensuring verifiable simulation and a clear distinction between ethical and unethical (for simulation only) strategies.

---

1. Project Structure and Skeleton Repository Layout

The following structure is standard for a TypeScript/Node.js-based MEV bot.

codeCode

```
/MEV-Searcher-Blueprint
|-- /src
|   |-- config.ts          # Environment variables, constants, RPC URLs
|   |-- types.ts           # Custom types (e.g., Opportunity, Bundle)
|   |-- coreBot.ts         # Main execution loop, mempool monitoring
|   |-- strategies.ts      # **STRATEGY SEPARATION: Legal/Illegal functions here**
|   |-- builderProvider.ts # Flashbots/Eden Provider initialization
|   |-- simulation.ts      # Mainnet-fork simulation logic
|   |-- submission.ts      # Bundle submission logic (Testnet/Mainnet)
|-- .env                   # Secure storage for PRIVATE_KEY, RPC_URLs, etc. (DO NOT COMMIT)
|-- package.json           # Project dependencies (ethers, flashbots, etc.)
|-- tsconfig.json          # TypeScript configuration
|-- hardhat.config.ts (or foundry.toml) # Local fork config
```

```
mev-searcher/
├── .env.example
├── package.json
├── src/
│   ├── index.js              # main entry: loop, detection, send bundle
│   ├── simulator.js          # code to call eth_callBundle / simulate
│   ├── strategy/
│   │   └── LegalStrategy.js   # detection + tx builder (non-exploitative)
│   │   └── UnboundStrategy.js   # detection + tx builder (exploitative)

│   ├── utils/
```

```
│   │   ├── wallets.js
│   │   └── rpc.js
├── hardhat.config.js          # optional: Hardhat fork setup
├── foundry/                   # optional: foundry config files
├── scripts/
│   ├── run_local_fork.sh
│   └── submit_test_bundle.js
├── docs/
│   └── runbook.md
└── README.md
```

2. Exact Commands to Install and Run

A. Environment Setup

codeBash

```bash
# 1. Initialize Node.js project
npm init -y

# 2. Install core dependencies
npm install ethers @flashbots/ethers-provider-bundle dotenv

# 3. Install TypeScript and setup (if using TS)
npm install -D typescript ts-node @types/node

# 4. Install Foundry (Recommended for fast local forking)
curl -L https://foundry.paradigm.xyz | bash
foundryup
```

B. Local Mainnet Forking (Foundry/Anvil)

Foundry's `anvil` is the fastest and most stable local Ethereum node for testing.

codeBash

```
# Start a local Mainnet fork on a high-availability public RPC URL
# (Replace YOUR_MAINNET_RPC_URL with a premium provider like Alchemy/Infura)
anvil --fork-url $YOUR_MAINNET_RPC_URL --chain-id 1337 --steps-tracing
```

- `--chain-id 1337`: Standard chain ID for local testing.
- `--steps-tracing`: Useful for debugging why transactions fail.

C. Running the Searcher

codeBash

```
# Run the main bot logic (assuming your entry file is src/coreBot.ts)
ts-node src/coreBot.ts
```

3. Core Code Snippets (Flashbots & Eden)

This uses the standard `ethers` and `@flashbots/ethers-provider-bundle` approach.

A. Provider Initialization (`src/builderProvider.ts`)

codeTypeScript

```
import { providers, Wallet } from 'ethers';
import { FlashbotsBundleProvider } from '@flashbots/ethers-provider-bundle';

// Initialize a standard Mainnet provider
const MAINNET_RPC_URL = process.env.MAINNET_RPC_URL!;
const ETH_PROVIDER = new providers.JsonRpcProvider(MAINNET_RPC_URL);

// Separate signer for signing Flashbots requests (can be a burner wallet)
const FLASHBOTS_SIGNER = new Wallet(process.env.FLASHBOTS_KEY!);

// The actual wallet that executes the transaction (must have funds)
export const SEARCHER_WALLET = new Wallet(process.env.SEARCHER_PRIVATE_KEY!,
ETH_PROVIDER);

/**
 * Initializes the Flashbots Bundle Provider (for submission to Flashbots Relay).
 * @param chainId Use 1 for Mainnet, 11155111 for Sepolia Testnet.
 */
export async function getFlashbotsProvider(chainId: number): Promise<FlashbotsBundleProvider> {
    return FlashbotsBundleProvider.create(
        ETH_PROVIDER,
        FLASHBOTS_SIGNER,
        // Relay URL: Use the appropriate relay for the target chain
        chainId === 1 ? 'https://relay.flashbots.net' : 'https://relay-sepolia.flashbots.net'
    );
}
```

```typescript
/**
 * Eden Network uses a standard RPC structure for private transactions.
 * No special provider required, but you submit transactions via their RPC.
 */
export function getEdenProvider() {
    // You would use this RPC URL to submit a single, private transaction
    const EDEN_RPC_URL = 'https://mainnet.edennetwork.io/rpc';
    return new providers.JsonRpcProvider(EDEN_RPC_URL);
}
```

B. Bundle Creation, Simulation, and Submission (`src/submission.ts`)

codeTypeScript

```typescript
// Example: Creating two transactions for an arbitrage
const transaction1 = {
    // ... populated transaction data (to, data, gasLimit, maxFeePerGas, maxPriorityFeePerGas)
    nonce: await SEARCHER_WALLET.getTransactionCount()
};

const transaction2 = {
    // ... populated transaction data (the actual bribe to the builder)
    nonce: (await SEARCHER_WALLET.getTransactionCount()) + 1
};

const transactionBundle = [
    {
        signer: SEARCHER_WALLET,
        transaction: transaction1
    },
    {
        signer: SEARCHER_WALLET,
        transaction: transaction2
    }
];

/**
 * Simulates a bundle on the target network. Essential for profit verification.
 */
export async function simulateBundle(fbProvider: FlashbotsBundleProvider, bundle: any[]): Promise<any>
{
    const targetBlock = (await ETH_PROVIDER.getBlockNumber()) + 1;
    console.log(`Simulating bundle for block ${targetBlock}...`);

    // Use the official flashbots_simulate endpoint
    const simulationResult = await fbProvider.simulate(bundle, targetBlock);

    // CRITICAL: Check for error and profit before proceeding
```

```
  if ('error' in simulationResult) {
    console.error('Simulation Failed:', simulationResult.error.message);
    return null;
  }

  console.log(`Simulation Success. Profit: ${simulationResult.results[simulationResult.results.length -
1].coinbaseDiff.toString()}`);
  return simulationResult;
}

/**
 * Submits the bundle to the Relay.
 */
export async function submitBundle(fbProvider: FlashbotsBundleProvider, bundle: any[]) {
  const targetBlock = (await ETH_PROVIDER.getBlockNumber()) + 1;
  const signedBundle = await fbProvider.signBundle(bundle);

  // Send the bundle
  const submission = await fbProvider.sendBundle(signedBundle, targetBlock);

  console.log(`Bundle submitted. Hash: ${submission.bundleHash}`);

  // Monitor inclusion - crucial for production verification
  const status = await submission.wait();
  if (status === 0) {
    console.log('Bundle was NOT included in the target block.');
  } else {
    console.log(`Bundle INCLUDED in block ${status.blockNumber}!`);
  }
}
```

4. Mainnet-Fork Simulation Workflow

Tool	Purpose	Key Verification Command
Foundry (Anvil)	Fastest local execution environment. Allows full EVM state manipulation (e.g., funding a searcher wallet).	`eth_callBundle` or `flashbots_simulate` via the Flashbots Provider pointed to the local Anvil instance.
Hardhat	Debugging and advanced scripting (less favored for raw speed).	`hardhat_mine` to advance blocks after simulating. `tenderly.verify` for integration.

Tenderly	**CRITICAL for Production.** Provides powerful, shareable, and verifiable cloud-based simulation. Can trace state changes across multiple transactions.	Use the Tenderly SDK/API to submit the bundle for a **fork simulation** (allows peer review of the exact outcome).

Workflow:

1. Run `anvil --fork-url ...`.
2. Point your `ETH_PROVIDER` in `builderProvider.ts` to `http://127.0.0.1:8545`.
3. The bot detects an opportunity in the **live Mainnet mempool**.
4. The bot constructs the bundle and calls `simulateBundle` against the **local Anvil fork**.
5. If simulation is successful (positive profit, no errors), the bot switches the provider back to the live **Flashbots Testnet/Mainnet Relay** and calls `submitBundle`.

5. Strategy Separation and Documentation

This section provides the essential academic and legal separation needed for your paper. All strategies below utilize the same `createBundle`, `simulateBundle`, and `submitBundle` functions, but the *logic* within the detection/bundle construction phase differs.

A. Legal and Ethical Strategies (`src/strategies.ts`)

These are safe for Mainnet deployment as they do not victimize users.

Strategy	Definition	Academic/Legal Domain
1. Cross-DEX Arbitrage	Identifying a price difference between two independent DEX pools and profiting by buying on one and selling on the other. This benefits the market by reducing price inefficiency.	**Legal:** Benign market making. Does not rely on front-running a *specific* user transaction.

2. White-Hat Liquidation Rescue	Detecting a loan eligible for liquidation and constructing a bundle to repay the loan or withdraw collateral to a safe address *before* a malicious liquidator can seize the collateral.	**Ethical/Defensive:** Uses the exploit mechanism (private ordering) for a beneficial outcome. Focuses on **resilience** and **defense modeling**.
3. CEX-DEX Arbitrage	Identifying a price difference between a Centralized Exchange (CEX) and a Decentralized Exchange (DEX). Requires complex off-chain/on-chain coordination.	**Legal:** Benign market making. Does not front-run on-chain users.

B. Illegal/Unethical Strategies (For Simulation and Research Only)

These must **NEVER** be submitted to the Mainnet Relay and should have built-in checks to ensure they only run against a local fork or Testnet.

Strategy	Definition	Academic/Legal Domain
1. Sandwich Attack	Front-running a user's trade (TX A) to move the price, including the user's trade at a worse price (TX B), and then back-running to return the price (TX C) and capture the profit.	**Illegal/Unethical (Exploitative):** Relies on intentionally victimizing a specific user transaction. Research focus is on **Detection** and **Mitigation** (e.g., using private RPCs).
2. Front-Running Simple User Trades	Observing a user transaction (e.g., a simple token swap) and submitting a bundle that executes the same trade with a higher fee	**Illegal/Unethical (Exploitative):** Directly victimizes the user by stealing their favorable trade outcome. Research focus is on **Resilience Modeling** and **Fairness Mechanisms**.

to beat them to the best price.

---

6. Production Checklist & Cost Estimates

Category	Checklist Item	Cost Estimate (Monthly)
Infrastructure	**High-Speed RPC:** Premium Alchemy/Infura for low latency state access.	**$100 - $500** (Tier dependent on request volume)
Execution	**Cloud VM:** AWS EC2 (c6i.large or similar) or GCP Compute Engine for low-latency hosting.	**$50 - $150**
Monitoring	**Alerting:** Prometheus/Grafana or PagerDuty for "Bot Died" or "Profit Dropped Below Threshold" alerts.	**$0 - $50**
Security	**Firewall:** Restrict access to the VM to a specific IP range.	Included in Cloud VM cost
Gas/Transaction	**Mainnet Gas Fees:** Fees for transactions that *aren't* included in a bundle (e.g., funding, initial setup). **Priority Fees/Bribes:** Paid *only* on successful bundle inclusion.	**Variable:** Dependent on trade volume/success. Can be high.

7. Troubleshooting, Best Practice, and Next Steps

Issue	Root Cause & Troubleshooting	Best Practice in Deployment
Bundle Submission Failure	Incorrect nonce, insufficient gas limits, or transaction is invalid on a specific block.	**Always simulate first.** Ensure the **Searcher Wallet** has enough Ether for execution, not just the bundle profit.

Profitable Simulation, Failed Inclusion	Another searcher bid a higher priority fee (bribe).	Implement a **Dynamic Bidding** function. Constantly monitor the public Flashbots stats for the minimum required bribe and adjust your bundle's `maxPriorityFeePerGas` instantly.
Latency/Stale Data	RPC node is lagging the network state.	**Use a dedicated, low-latency, WebSockets-enabled RPC provider.** Run the bot in a region geographically close to the relay.
Private Key Compromise	Secrets exposed in the codebase or environment.	**NEVER hardcode keys.** Use a secure secrets manager (like AWS Secrets Manager) or a robust `.env` file with `.gitignore` enforced. Use a separate, low-balance **`FLASHBOTS_SIGNER`** key that is only for signing bundles, not holding funds.

Final Workflow Summary (Following Prompt Instructions)

1. **Local Repo Setup:** Create the skeleton, install `ethers` and `@flashbots/ethers-provider-bundle`.
2. **Mainnet Fork:** Use **Foundry/Anvil** to start a local Mainnet fork (`--fork-url ...`).
3. **Verification:** Implement `simulateBundle`. Point the Flashbots Provider to the local Anvil instance and use the **`flashbots_simulate`** or `eth_callBundle` API call to verify the trades are profitable and error-free *before* sending.
4. **Testnet Iteration:** Change the Flashbots Provider to target the **Sepolia Testnet Relay** (`https://relay-sepolia.flashbots.net`). Submit the signed bundle. Iterate until inclusion is reliable.
5. **Mainnet Deployment:** When confident, move to Mainnet by changing the relay URL to `https://relay.flashbots.net` for bundles, or use **Flashbots Protect RPC** / **Eden RPC** for single-transaction private submissions as a defense against front-running.
6. **Safety:** Enforce strict profit (e.g., minimum $10 gross profit) and gas sanity checks in code. **All unethical strategies must remain strictly confined to the local fork/Testnet simulation environment**

0) What you'll need (prereqs)

- Node.js (v18+) & npm or pnpm.
- Git, VS Code (or preferred editor).
- An Ethereum RPC provider (Alchemy/Infura/QuickNode) for fork/simulation and testnet/mainnet RPC.
- Testnet ETH (Sepolia) for test submissions.
- Secrets management (.env).
- Optional but recommended: Tenderly account (simulation), Foundry / Anvil (fast local forks), and a small VPS for running the bot 24/7.
  Official Flashbots docs and libraries you'll rely on: Flashbots Quick Start and `ethers-provider-flashbots-bundle`. [Flashbots Docs+1]

---

1) Project layout (starter skeleton)

Create a repo `mev-searcher/` with this structure:

```
mev-searcher/
├─ .env.example
├─ package.json
├─ src/
│  ├─ index.js              # main entry: loop, detection, send
bundle
│  ├─ simulator.js          # code to call eth_callBundle / simulate
│  ├─ strategy/
│  │  └─ exampleStrategy.js   # detection + tx builder
(non-exploitative)
│  ├─ utils/
│  │  ├─ wallets.js
│  │  └─ rpc.js
├─ hardhat.config.js        # optional: Hardhat fork setup
├─ foundry/                 # optional: foundry config files
```

```
├─ scripts/
│   ├─ run_local_fork.sh
│   └─ submit_test_bundle.js
├─ docs/
│   └─ runbook.md
└─ README.md
```

Create `.env.example`:

```
RPC_URL=https://sepolia.infura.io/v3/YOUR_KEY
MAINNET_RPC=https://eth-mainnet.alchemyapi.io/v2/YOUR_KEY
SEARCHER_KEY=0x....        # signing key for bundles (keep small
balance)
FLASHBOTS_RELAY=https://relay-sepolia.flashbots.net
TARGET_PROFIT_USD=10       # safety minimum
```

---

2) Install the minimal dependencies

```
mkdir mev-searcher && cd mev-searcher
npm init -y
npm install ethers dotenv @flashbots/ethers-provider-bundle
# Optional for simulation helpers:
npm i --save-dev hardhat @nomiclabs/hardhat-ethers
```

The Flashbots ethers bundle provider is the standard high-level lib for `eth_sendBundle` and `eth_callBundle`. GitHub

---

3) Minimal safe skeleton (index.js)

This skeleton connects to a relay and shows how to **simulate** and **send** a bundle (no strategy logic included). Drop into `src/index.js`:

```
require('dotenv').config();
```

```
const { ethers } = require('ethers');
const { FlashbotsBundleProvider } =
require('@flashbots/ethers-provider-bundle');

const RPC = process.env.RPC_URL;
const FLASHBOTS_RELAY = process.env.FLASHBOTS_RELAY ||
"https://relay-sepolia.flashbots.net";
const SEARCHER_KEY = process.env.SEARCHER_KEY;

async function main(){
  const provider = new ethers.providers.JsonRpcProvider(RPC);
  const authSigner = new ethers.Wallet(SEARCHER_KEY, provider); //
identity for relay
  const flashbotsProvider = await
FlashbotsBundleProvider.create(provider, authSigner, FLASHBOTS_RELAY);

  const blockNumber = await provider.getBlockNumber();
  console.log("current block:", blockNumber);

  // Example: an empty bundle (demo). signedTxs must be raw signed
transactions when real.
  const signedTxs = []; // e.g., ['0x...signedtx1', '0x...signedtx2']
  const targetBlock = blockNumber + 1;

  // simulate before sending
  try {
    const sim = await flashbotsProvider.simulate(signedTxs,
targetBlock);
    console.log("simulation:", sim);
  } catch(e){
    console.error("simulate error:", e);
  }

  // send if simulation OK (in real code: check sim.success + profit)
  if (signedTxs.length){
    const res = await flashbotsProvider.sendRawBundle(signedTxs,
targetBlock);
    console.log("sendRawBundle result:", res);
```

```
  } else {
    console.log("No signed txs to send — skeleton complete.");
  }
}

main().catch(console.error);
```

Notes:

- Use Sepolia relay `https://relay-sepolia.flashbots.net` to test safely. Flashbots operates a Sepolia testnet relay for searchers. [Flashbots Docs]
- Real bundles will contain `signedTxs` built from your strategy. Don't put strategy code here yet.

---

4) Mainnet fork simulation (Hardhat or Foundry) — exact steps

Why: pool marginal prices are useless for large trades. You must **simulate against on-chain pool depth** (mainnet state) to get realistic slippage and reverts. Use Hardhat or Foundry/Anvil to fork the chain at a recent block and run `eth_callBundle` or local simulation.

Hardhat approach (quick)

1. Install Hardhat:

```
npm i --save-dev hardhat @nomiclabs/hardhat-ethers
npx hardhat
```

2. `hardhat.config.js` — fork config:

```
module.exports = {
  solidity: "0.8.19",
```

```
  networks: {
    hardhat: {
      forking: {
        url: process.env.MAINNET_RPC, // Alchemy/Infura
        blockNumber: +process.env.FORK_BLOCK || undefined
      }
    }
  }
};
```

3. Start node fork:

```
npx hardhat node --fork ${MAINNET_RPC} --fork-block-number
${FORK_BLOCK}
```

4. Point your `RPC_URL` in `.env` to `http://127.0.0.1:8545` and run simulation code that calls `flashbotsProvider.simulate(...)` or use `provider.send('eth_call', ...)` on the fork. Hardhat allows impersonation of accounts for testing.

Foundry / Anvil approach (fast)

- Foundry/Anvil is faster for repeated simulations. Guide: quicknode Foundry guide (forking with Anvil). QuickNode

Tenderly (hosted, easiest for visual debugging)

- You can run transaction simulations and time-travel forks on Tenderly's dashboard — great for visual trace and debugging reverts before sending to testnet. Use Tenderly simulator API to run complex scenarios. Tenderly+1

---

5) Example strategy scaffold (safe, non-exploitative)

Create `src/strategy/exampleStrategy.js` with a detection stub and tx builder:

```
const { ethers } = require('ethers');

async function detectOpportunity(provider){
  // DO NOT implement front-running or oracle manipulation.
  // Safe example: detect disparity between two DEX quoted amounts for
a tiny size.
  // Return null if no op; otherwise return an object with txs and
expectedProfitUSD.
  // Implement using provider.call and on-chain reserves.
  return null; // placeholder
}

async function buildBundle(signer, opportunity){
  // Build raw txs: sign them with signer and return array of raw
signed txs
  // Example tx construction:
  // const tx = await signer.populateTransaction({...});
  // const signed = await signer.signTransaction(tx);
  // return [signed];
  return [];
}

module.exports = { detectOpportunity, buildBundle };
```

Important: keep this module limited to *detection + safe checks* — no instructions on harmful operations.

---

6) Full test flow (end-to-end)

1. **Local iteration (no real ETH):**
    - Use Hardhat/Foundry fork of mainnet block N.
    - Run the strategy against live state, simulate the bundle (`eth_callBundle`/`simulate`) until success.
2. **Sepolia submission:**
    - Switch `.env.RPC_URL` to Sepolia.
    - Use `FLASHBOTS_RELAY=https://relay-sepolia.flashbots.net`.
    - Submit signed bundles (low value) and verify inclusion or rejection responses. Flashbots has a Sepolia test relay for searchers. [Flashbots Docs]
3. **Tenderly checks (optional):**
    - Upload the tx trace to Tenderly to inspect gas consumption, reverts, and state changes. Tenderly helps detect hidden failure reasons. [Tenderly]
4. **Mainnet pilot (micro):**
    - Move RPC to mainnet, but start with *very* small nominal bundle sizes (penny packets) and strict profit thresholds.
    - Use Flashbots Protect RPC or Eden RPC to reduce exposure while testing. Flashbots Protect allows private RPC submission and some protection against frontrunning. [Flashbots Docs+1]

---

7) Submitting bundles: example sequence (simulate → send)

1. `simulate = await flashbotsProvider.simulate(signedTxs, targetBlock)`

  - Check `simulate.firstRevert`, `simulate.revert`, gasUsed, and expected state changes. (If revert — fix.) [GitHub]

2. If simulate OK and `expectedProfitUSD >= TARGET_PROFIT_USD`, then:

   - `res = await flashbotsProvider.sendRawBundle(signedTxs, targetBlock)`

3. Interpret `res`:

   - If res indicates acceptance, wait for inclusion.
   - If rejected, log and analyze failure cause. Flashbots docs provide guidance on interpreting relay responses. [Flashbots Docs]

---

8) Monitoring, logging & observability

- **Log**: bundle id, target block, signed txs hash, sim result, estimated profit, gas used, relay response, timestamp.
- **Dashboards**: use Grafana + Prometheus or a simple Kibana/ELK pipeline for logs. Track metrics: success rate, avg profit per bundle, failed gas spent per day.
- **Alerting**: Slack/Discord/Telegram alerts for repeated reverts, mispriced bundles, or sudden drop in success rate.
- **Cost accounting**: maintain precise accounting of gas used by failed bundles — this is where beginners burn cash.

---

9) Security & keys

- Use **separate** keys: one for signing bundles (low balance), separate hotkey(s) for testing. Never store secrets in plaintext; use a secrets manager (AWS Secrets Manager, HashiCorp Vault) in production.

- Add **rate limits** & timeouts to prevent runaway spamming.
- Enforce a **daily loss cap**: if failed gas > X USD, stop the bot until manual review.

---

10) Costs & small budget estimate (monthly)

- RPC (Alchemy/QuickNode) with archive/forking: $50–200
- VPS (Hetzner/AWS) for 24/7 runner: $20–200
- Tenderly (paid simulations) or Foundry infra: $0–200
- Optional private RPC / Eden / bloXroute: $50–500
- Total minimal: **~$150–$500 / month** for a competent solo searcher testing & running small pilot bots. (Scale up for colocation, multi-node, or many strategies.)

---

11) Checklist before going live on mainnet

- All simulations succeed on a recent mainnet fork (Hardhat or Anvil).
- Tenderly simulation shows no hidden reverts and gas usage is acceptable. Tenderly
- Sepolia relay submissions succeed repeatedly (testnet acceptance). Flashbots Docs
- Monitoring, alerting and kill-switch in place.
- Security review of signing keys & environment variables.
- Profitability threshold & daily loss cap configured.

---

12) Troubleshooting common failures

- **Bundle simulation always reverts** — inspect revert reason in simulation trace; frequently caused by incorrect calldata or insufficient approval allowances. Tenderly / Hardhat traces are invaluable for this. Tenderly
- **Bundle accepted by relay but not included** — competition; increase tip or adjust target block; analyze timings. Builders may have selected a more profitable competing bundle. Flashbots Docs
- **High failed-gas burn** — reduce retry aggressiveness and implement stricter simulation checks.

---

13) Next-level improvements (when ready)

- Automate simulation using Foundry + scripting for continuous checks. Foundry/Anvil is optimized for quick forks and iterations. QuickNode
- Use private RPC / block distribution networks (Eden, bloXroute) for lower latency & private routing. Eden Docs+1
- Join searcher communities (Flashbots Discord) to learn strategies and best practices. Flashbots docs and community are essential starting points. Flashbots Docs+1

1.