Solana Backend Integration Roadmap

To connect the **AutoSave Protocol** frontend to the blockchain, you need to implement three distinct layers: the Program (on-chain), the Provider (connection), and the Instruction calls (frontend logic).

1. The On-Chain Program (Rust/Anchor)

Your Solana program would define the **Personal Derived Address (PDA)** logic. In Rust, a simplified version of your "Auto-Savings" instruction might look like this:

```
#[derive(Accounts)]
pub struct SaveFunds<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    #[account(
        init_if_needed,
        payer = user,
        space = 8 + 8, // Discriminator + amount (u64)
        seeds = [b"vault", user.key().as_ref()],
        bump
    )]
    pub vault_pda: Account<'info, VaultAccount>,
    pub system_program: Program<'info, System>,
}
```

2. Wallet Integration

You would replace the "Connect Wallet" button in auto_savings.jsx with the @solana/wallet-adapter-react library. This handles the handshake with browser wallets like Phantom or Solflare.

3. Connecting the Frontend (The "Bridge")

In the React app, you would swap the setTimeout simulations for actual RPC calls. We use the @coral-xyz/anchor library to talk to the program.

Example Transformation:

Before (Current Simulation):

```
const handleTransaction = () => {
```

```
  setIsProcessing(true);
  setTimeout(() => {
    setMockStats(...);
    setIsProcessing(false);
  }, 1800);
};
```

After (Real Blockchain Call):

```
const handleTransaction = async () => {
  const provider = new AnchorProvider(connection, wallet, {});
  const program = new Program(IDL, PROGRAM_ID, provider);

  try {
    setIsProcessing(true);
    const tx = await program.methods
      .deposit(new BN(amountInput * LAMPORTS_PER_SOL))
      .accounts({
        user: wallet.publicKey,
        vaultPda: vaultPdaAddress,
      })
      .rpc();

    // Wait for confirmation
    await connection.confirmTransaction(tx);
    // Refresh balances from the chain...
  } catch (err) {
    console.error("Transaction failed", err);
  } finally {
    setIsProcessing(false);
  }
};
```

4. Real-Time Analytics

Instead of mockStats, you would use:

- **Helius or Alchemy Webhooks**: To listen for every transfer the user makes and update the UI.
- **Birdeye API or Pyth Network**: To replace the Gemini price fetch with high-frequency, institutional-grade price feeds.
- **Dune Analytics**: To pull the "Global Protocol Analytics" (Total Value Saved) directly from

the program's total TVL.

5. Deployment Steps

1. **Devnet**: Deploy the Rust program to Solana Devnet (solana program deploy).
2. **IDL Generation**: Generate the JSON interface (IDL) for the frontend.
3. **Environment Variables**: Update the frontend with the PROGRAM_ID and RPC Endpoint (e.g., QuickNode or Helius).