Hyperliquid Agent Wallets: Complete Developer Implementation Guide

Agent wallets are Hyperliquid's programmatic trading gateway

Hyperliquid agent wallets (also called API wallets) are specialized cryptographic wallets that serve as authorized signing proxies for master accounts, enabling automated trading without exposing master private keys. Hyperliquid Docs +3 Operating on Hyperliquid's high-performance Layer 1 blockchain that processes 200,000 orders per second with sub-second finality, Hyperliquid Docs CoinGecko these wallets form the foundation for algorithmic trading systems. Gem Wallet +2

The architecture employs a delegation-based system where master accounts approve agent wallets to sign transactions on their behalf. Unlike regular wallets, agent wallets cannot hold balances or be queried for account data - they exist solely for signing trading operations. Hyperliquid Docs +2 Each account can maintain 1 unnamed agent wallet plus 3 named ones, with additional capacity for subaccounts. Hyperliquid Docs Hyperliquid Docs The system uses EIP-712 structured data signing with a unique dual-chain ID approach: Chain ID 1337 for trading actions and 0x66eee for administrative operations. Chainstack DeepWiki

Creating and configuring agent wallets

Setting up agent wallets requires a funded Hyperliquid account with USDC deposited before API key generation. The process begins at `app.hyperliquid.xyz/API` where you connect your MetaMask or compatible wallet. Medium +2

Step-by-Step Setup Process

Generate the Agent Wallet:

1. Navigate to the API section and click "Generate" Tealstreet
2. Optionally name the wallet (recommended for organization) Hyperliquid Docs
3. Set validity period (maximum 180 days)
4. Immediately secure the private key - it cannot be recovered Medium Tealstreet

Authorize the Agent:

```python
```

```
# Python SDK authorization
from hyperliquid.exchange import Exchange
from hyperliquid.utils import constants

# Use master wallet to approve agent
exchange = Exchange(master_wallet_key, constants.MAINNET_API_URL)
result, agent_key = exchange.approve_agent("trading_bot_v1")
```

Configuration Structure:

```json
{
  "account_address": "0x[MASTER_ACCOUNT_ADDRESS]",
  "secret_key": "[AGENT_WALLET_PRIVATE_KEY]",
  "base_url": "https://api.hyperliquid.xyz"
}
```

Critical Configuration Requirements

Always specify the master account address when initializing agent wallets - this is the most common implementation error. GitHub +3 Agent wallets share nonces across subaccounts, so use separate agent wallets for parallel processes to avoid collisions. Hyperliquid Docs Never reuse deregistered agent wallet addresses as their nonce state may be pruned. Hyperliquid Docs gitbook

Permission systems enable granular trading control

Hyperliquid implements sophisticated permission delegation through two distinct signing mechanisms. Agent wallets receive trading permissions while administrative actions remain exclusively with the master account. TrySuper Hyperliquid Docs

Trading Permissions (L1 Actions)

Agent wallets can execute:

- Order placement, cancellation, and modification
- Batch operations for high-frequency strategies
- Leverage and margin adjustments
- Vault and subaccount transfers
- Dead man's switch activation

Administrative Restrictions

Only master accounts can:

- Approve or revoke agent wallets
- Withdraw funds externally
- Transfer USDC between accounts
- Modify builder fee settings

Security Implementation

```python
# Secure permission management
class SecureAgentManager:
    def __init__(self, exchange):
        self.exchange = exchange
        self.active_agents = {}

    def rotate_agent(self, old_name, new_name):
        """Rotate agent wallets for security"""
        # Generate new agent
        result, new_key = self.exchange.approve_agent(new_name)

        # Store securely
        self.active_agents[new_name] = {
            'key': new_key,
            'created': time.time(),
            'expires': time.time() + (180 * 24 * 3600)
        }

        # Revoke old agent
        if old_name in self.active_agents:
            del self.active_agents[old_name]

        return new_key
```

API integration powers automated trading

Hyperliquid provides REST and WebSocket APIs with comprehensive SDK support across multiple languages. Hyperliquid Docs GitHub The system uses EIP-712 signatures with a unique phantom agent construction for authentication. Chainstack +3

Core Endpoints

- **Mainnet:** `https://api.hyperliquid.xyz` Hyperliquid Docs
- **WebSocket:** `wss://api.hyperliquid.xyz/ws` Hyperliquid Docs
- **Exchange Actions**: `POST /exchange`

- **Market Data**: `POST /info`

Authentication Architecture

```javascript
// TypeScript authentication implementation
const sdk = new Hyperliquid({
  privateKey: process.env.AGENT_PRIVATE_KEY,  // Agent wallet key
  walletAddress: process.env.MASTER_ADDRESS,  // Critical: Master account
  testnet: false,
  enableWs: true
});

// Place order with proper authentication
const orderResult = await sdk.exchange.placeOrder({
  coin: 'BTC-PERP',
  is_buy: true,
  sz: '0.001',
  limit_px: '45000',
  order_type: { limit: { tif: 'Gtc' } }
});
```

Nonce Management System

Hyperliquid stores the 100 highest nonces per address with time-bounded validation. Nonces must be unique, larger than the smallest stored nonce, and within (T-2 days, T+1 day) of current timestamp. Hyperliquid Docs

```python
# Robust nonce manager
class NonceManager:
    def __init__(self):
        self._lock = threading.Lock()
        self._nonce = int(time.time() * 1000)

    def get_next_nonce(self):
        with self._lock:
            self._nonce += 1
            return self._nonce
```

Security best practices prevent common vulnerabilities

Implementing robust security requires multi-layered protection strategies focusing on key management, access control, and continuous monitoring.

Key Management Architecture

```python
# Production key management
import os
from cryptography.fernet import Fernet

class SecureKeyManager:
    def __init__(self):
        self.cipher = Fernet(os.getenv('ENCRYPTION_KEY'))

    def store_agent_key(self, name, private_key):
        encrypted = self.cipher.encrypt(private_key.encode())
        # Store encrypted key in secure database

    def retrieve_agent_key(self, name):
        # Retrieve from secure storage
        encrypted_key = self.get_from_storage(name)
        return self.cipher.decrypt(encrypted_key).decode()
```

Multi-Wallet Strategy

Implement separation of concerns:

- **Storage Wallet**: Hardware wallet, never connected to dApps
- **Trading Wallet**: Hot wallet for daily operations
- **Agent Wallets**: Separate keys per trading process Gitbook

Rate Limit Management

```javascript
```

```
// Rate limiter with exponential backoff
class RateLimiter {
  constructor() {
    this.requestCount = 0;
    this.lastReset = Date.now();
    this.maxRequests = 1200; // Per minute
  }

  async executeWithLimit(operation) {
    if (this.requestCount >= this.maxRequests) {
      const waitTime = 60000 - (Date.now() - this.lastReset);
      await this.sleep(waitTime);
      this.reset();
    }

    this.requestCount++;
    return await operation();
  }
}
```

Code examples demonstrate practical implementation

Complete Trading Bot Implementation

```
python
```

```python
# Production-ready trading bot
from hyperliquid.info import Info
from hyperliquid.exchange import Exchange
from hyperliquid.utils import constants
import asyncio
import logging

class HyperliquidTradingBot:
    def __init__(self, config):
        self.info = Info(constants.MAINNET_API_URL, skip_ws=True)
        self.exchange = Exchange(
            config['agent_key'],
            constants.MAINNET_API_URL,
            account_address=config['master_address']
        )
        self.position_limits = config['position_limits']

    async def execute_strategy(self):
        """Main trading logic"""
        while True:
            try:
                # Get market data
                all_mids = self.info.all_mids()
                user_state = self.info.user_state(self.exchange.account_address)

                # Analyze positions
                positions = user_state.get('assetPositions', [])

                # Execute trading logic
                for asset in self.position_limits.keys():
                    signal = self.calculate_signal(asset, all_mids)
                    if signal:
                        await self.place_order(asset, signal)

                await asyncio.sleep(1)  # Rate limit compliance

            except Exception as e:
                logging.error(f"Strategy error: {e}")
                await asyncio.sleep(10)

    def calculate_signal(self, asset, prices):
        # Implement your strategy logic
        pass

    async def place_order(self, asset, signal):
        """Place order with error handling"""
        try:
            result = self.exchange.order(
                coin=asset,
                is_buy=signal['side'] == 'buy',
                sz=signal['size'],
                px=signal['price'],
                order_type={"limit": {"tif": "Gtc"}},
                reduce_only=False
            )
            logging.info(f"Order placed: {result}")
            return result
        except Exception as e:
            logging.error(f"Order failed: {e}")
            return None
```

WebSocket Real-Time Integration

```javascript
```

```javascript
// WebSocket-based high-frequency trading
const { Hyperliquid } = require('hyperliquid');

class WebSocketTradingSystem {
    constructor(config) {
        this.sdk = new Hyperliquid({
            enableWs: true,
            privateKey: config.agentKey,
            walletAddress: config.masterAddress
        });
        this.orderBook = {};
        this.positions = {};
    }

    async initialize() {
        await this.sdk.connect();

        // Subscribe to real-time data
        this.sdk.subscriptions.subscribeToAllMids(data => {
            this.processPriceUpdate(data);
        });

        this.sdk.subscriptions.subscribeToL2Book('BTC-PERP', data => {
            this.orderBook['BTC-PERP'] = data;
            this.checkArbitrageOpportunity('BTC-PERP');
        });

        this.sdk.subscriptions.subscribeToUserFills(
            this.sdk.walletAddress,
            fill => this.processFill(fill)
        );
    }

    async checkArbitrageOpportunity(asset) {
        const book = this.orderBook[asset];
        if (!book) return;

        const spread = book.asks[0][0] - book.bids[0][0];
        if (spread > this.minSpread) {
            await this.executeArbitrage(asset, book);
        }
    }

    async executeArbitrage(asset, book) {
        // Place simultaneous buy and sell orders
        const orders = [
        { coin: asset, is_buy: true, sz: '0.01', limit_px: book.bids[0][0] },
        { coin: asset, is_buy: false, sz: '0.01', limit_px: book.asks[0][0] }
      ];

      await this.sdk.exchange.placeOrder({ orders });
    }
  }
```

Common use cases span diverse trading strategies

Market Making Implementation

Market makers provide liquidity by maintaining buy and sell orders around fair value. Agent wallets enable continuous order management without manual intervention.

```
python
```

```
# Market making strategy
class MarketMaker:
    def __init__(self, exchange, config):
        self.exchange = exchange
        self.spread_bps = config['spread_bps']
        self.order_size = config['order_size']
        self.max_position = config['max_position']

    def update_orders(self, fair_price, current_position):
        # Cancel existing orders
        self.exchange.cancel(coin='BTC')

        # Calculate order prices
        bid_price = fair_price * (1 - self.spread_bps/10000)
        ask_price = fair_price * (1 + self.spread_bps/10000)

        # Place new orders
        orders = []
        if current_position < self.max_position:
            orders.append({
                'a': 0, 'b': True, 'p': str(bid_price),
                's': str(self.order_size), 'r': False,
                't': {'limit': {'tif': 'Gtc'}}
            })
        if current_position > -self.max_position:
            orders.append({
                'a': 0, 'b': False, 'p': str(ask_price),
                's': str(self.order_size), 'r': False,
                't': {'limit': {'tif': 'Gtc'}}
            })

        if orders:
            self.exchange.order(orders=orders)
```

Copy Trading Systems

Agent wallets enable automated replication of successful traders' strategies through on-chain monitoring and instant execution. TrySuper

Vault Management

Vault leaders use agent wallets to manage community funds with sophisticated multi-asset strategies and 10% profit sharing. Blocmates Hummingbot

Rate limits require strategic optimization

Hyperliquid implements dual rate limiting systems that developers must carefully manage for optimal performance. gitbook Hyperliquid Docs

IP-Based Limits

- **REST Requests**: 1,200 weight/minute
- **WebSocket**: 100 connections, 1,000 subscriptions GitHub
- **Weight Calculation**: 1 + floor(batch_length/40) for orders gitbook Hyperliquid Docs

Address-Based Limits

- **Volume-Based**: 1 request per 1 USDC traded
- **Initial Buffer**: 10,000 requests
- **Enhanced Cancel**: min(limit + 100,000, limit × 2) gitbook Hyperliquid Docs

Optimization Strategies

python

```python
# Batch order optimization
def optimize_batch_requests(orders, max_batch_size=40):
    """Optimize order batching for rate limits"""
    batches = []

    # Separate by order type for priority
    alo_orders = [o for o in orders if o.get('alo')]
    regular_orders = [o for o in orders if not o.get('alo')]

    # Batch ALO orders first (higher priority)
    for i in range(0, len(alo_orders), max_batch_size):
        batches.append(alo_orders[i:i+max_batch_size])

    # Batch regular orders
    for i in range(0, len(regular_orders), max_batch_size):
        batches.append(regular_orders[i:i+max_batch_size])

    return batches

# Execute with timing
async def execute_batches(exchange, batches):
    for batch in batches:
        await exchange.order(orders=batch)
        await asyncio.sleep(0.1)  # Recommended spacing
```

Error handling ensures robust operation

Comprehensive Error Management

```python
class ErrorHandler:
    def __init__(self):
        self.error_counts = {}
        self.max_retries = 3

    def handle_exchange_error(self, error, context):
        error_msg = str(error)

        if "insufficient margin" in error_msg:
            return self.handle_insufficient_margin(context)
        elif "rate limit" in error_msg:
            return self.handle_rate_limit(context)
        elif "invalid signature" in error_msg:
            return self.handle_auth_error(context)
        elif "order not found" in error_msg:
            return self.handle_order_not_found(context)
        else:
            return self.handle_unknown_error(error, context)

    def handle_rate_limit(self, context):
        # Exponential backoff
        wait_time = 2 ** self.error_counts.get('rate_limit', 0)
        time.sleep(min(wait_time, 60))
        self.error_counts['rate_limit'] = self.error_counts.get('rate_limit', 0) + 1
        return {'retry': True, 'wait': wait_time}
```

Common Error Patterns

- **Price Tick Size**: Ensure prices divisible by asset tick size
- **Minimum Order Value**: Orders must exceed $10 USD
- **Nonce Conflicts**: Use atomic counters and separate wallets
- **WebSocket Disconnects**: Implement automatic reconnection TradingOnramp

Recent updates enhance system capabilities

2024 Major Releases

HYPE Staking Launch (December 30, 2024): Native staking with 2.26% APR enables governance participation and fee discounts up to 40% for Diamond tier stakers. Bitget +2

HyperEVM Compatibility: Full EVM support enables smart contract deployments and DeFi composability while maintaining 200k orders/second performance. Telegram Hyperliquid Docs

Multi-Sig Transactions: Native L1 primitive for enhanced security allows complex signing schemes for institutional traders. Telegram

Critical Changes for Developers

- **Never reuse deregistered agent addresses** - nonce state is pruned Hyperliquid Docs gitbook
- **Separate wallets for parallel processes** - avoid nonce collisions Hyperliquid Docs
- **180-day maximum validity** - implement rotation schedules
- **Batch every 0.1 seconds** - optimal for rate limits Hyperliquid Docs

Performance Metrics

- **Daily Volume**: $15B ATH (15x growth from 2023) medium
- **Order Throughput**: 200,000 orders/second Gem Wallet CoinGecko
- **Block Finality**: <1 second Hyperliquid Docs +2
- **Uptime**: 99.95% over 90 days Statuspage

Roadmap Items

- General ERC20 support with native transfers Telegram
- Enhanced multi-sig UI beyond current proof-of-concept Telegram
- Expanded validator network for further decentralization
- MetaMask integration for perpetual trading Yahoo Finance +3

Conclusion

Hyperliquid agent wallets provide a robust foundation for building sophisticated trading systems on one of crypto's highest-performance exchanges. Hyperliquid The unique architecture combining stateless signing wallets, dual-chain authentication, and sophisticated nonce management enables strategies from high-frequency market making to complex vault management. TrySuper +2 With zero gas fees, sub-second finality, and comprehensive SDK support, Hyperliquid Docs developers can implement production-ready trading systems that rival centralized exchange capabilities while maintaining self-custody and on-chain transparency. Coin Bureau +4

Success requires careful attention to configuration details - particularly specifying master addresses correctly, managing nonces properly, and implementing robust error handling. Hyperliquid Docs The platform's continued evolution with HyperEVM compatibility and enhanced multi-sig support positions it as a leading infrastructure for the next generation of decentralized finance applications. Hyperliquid Docs Telegram