

# PART 1: Hyperliquid & Infrastructure Research (30% weight)

## 1.1 Hyperliquid API Knowledge

Answer the following questions about Hyperliquid's API. You should reference the official documentation and demonstrate depth of understanding:

### Questions:

1. List and briefly explain the main REST API endpoints available for market data and order execution. What are the key differences between the public and authenticated endpoints?

Hyperliquid provides two main categories of REST API endpoints: **public** endpoints for market data and **authenticated** endpoints for order execution and account operations.

#### Public Endpoints – [/info](#)

Public endpoints allow users to query real-time and historical market information without authentication.

- These endpoints are used for building dashboards, market analytics, and basic data retrieval since they are **read-only** and **do not require a signature**.

#### Authenticated Endpoints – [/exchange](#)

Authenticated endpoints allow traders to perform state-changing operations, such as placing or canceling orders.

They require:

- A **signed request** using the trader's private key or approved API wallet.
- A **nonce** (unique timestamp) to prevent replay attack

### Key Differences Between Public and Authenticated Endpoints

Feature	Public ( <a href="#">/info</a> )	Authenticated ( <a href="#">/exchange</a> )
Purpose	Fetching market or user data	Executing trades or managing funds
Authentication	Not required	Required (signed with wallet key)
Effect on account	Read-only	Changes account state
Example Use	Get BTC order book	Place a BTC buy order

### References:

- Hyperliquid Docs: <https://docs.hyperliquid.xyz/api/info>

- <https://docs.hyperliquid.xyz/api/exchange>

## 2. Describe the WebSocket feed structure. What subscription channels are available, and how would you handle reconnection logic in a production system?

The Hyperliquid **WebSocket API** provides a real-time data stream for both market and user-specific updates.

Before subscribe into any channel, Developer have to connect to their API (Mainnet / Testnet). Once connected, Developer subscribe to one or more channels in **json format**, example: { "method": "subscribe", "subscription": { "type": "l2Book", "coin": "BTC" } }

### Example Subscription Channels

Category	Channel Type	Description
<b>Market Data</b>	<code>l2Book</code>	Live order book updates for a coin.
	<code>trades</code>	Stream of real-time executed trades.
	<code>candle</code>	Continuous OHLCV updates for charting.
	<code>allMids</code>	Mid-prices across all trading pairs.
<b>User Data</b>	<code>userFills</code>	Streams fill updates for a specific wallet.
	<code>orderUpdates</code>	Notifies of new, modified, or cancelled orders.
	<code>userFundings</code>	Updates on funding payments for perpetual positions.

### Reconnection Logic in Production

To maintain a stable connection in a production trading system:

1. **Detect disconnections** — monitor for `close` or `error` events on the WebSocket.
2. **Reconnect with exponential backoff** — wait 0.5s → 1s → 2s → 4s up to 30s between retries to avoid spamming.
3. **Re-subscribe automatically** — keep a list of all previous subscriptions and resend them after reconnecting.
4. **Send heartbeats** — send a `{"method": "ping"}` message every 20–30 seconds if no new data arrives. The server disconnects sockets idle for over 60 seconds.
5. **Recover data gaps** — after reconnection, request a fresh snapshot using the `/info` endpoint or a `post` message to ensure no missed trades or book updates.
6. **Respect rate limits** — maximum 100 WebSocket connections and 1000 subscriptions per IP; 2000 messages per minute globally.

In simple term: **connect** → **subscribe** → **stream**; if the wire breaks, **reconnect** → **re-subscribe** → **catch up** (so you don't miss data)

#### References:

- Hyperliquid WebSocket Docs: <https://docs.hyperliquid.xyz/api/websocket>

3. Explain the different order types supported by Hyperliquid (e.g., limit, market, stop). What are the use cases for each, and what are the key parameters required?

### 1. Limit Order

Executes only at a specific price or better. It remains in the order book until filled or cancelled.

**Use case:** Trader wants to buy BTC at \$60,000, not higher.

#### Example:

```
{
  "action": {
    "type": "order",
    "orders": [{
      "a": 1,      // asset ID for BTC
      "b": true,   // true = buy
      "p": "60000", // target price
      "s": "0.5",  // order size
      "tif": "Gtc"  // Good Till Cancelled
    }]
  },
  "nonce": 17300000000000,
  "signature": "0x..."
}
```

#### Key parameters:

**a** (asset ID), **b** (buy/sell), **p** (price), **s** (size), **tif** (time in force).

### 2. Market Order

Executes immediately at the best available price, using **tif:"ioc"** (Immediate-or-Cancel).

**Use case:** Trader wants to enter the market instantly.

#### Example:

```
{
  "action": {
```

```

    "type": "order",
    "orders": [{
      "a": 1,
      "b": true,
      "p": "999999", // far above current price to ensure instant fill
      "s": "0.5",
      "tif": "loc"    // Immediate or Cancel
    }]
  }
}

```

#### Key parameters:

`tif:"loc"` ensures immediate execution; if not filled instantly, the remainder cancels.

### 3. Stop / Trigger Order

Becomes active when the market hits a certain trigger price.

**Use case:** Trader wants to **sell 1 BTC if price falls below \$58,000** (stop-loss).

#### Example:

```

{
  "action": {
    "type": "order",
    "orders": [{
      "a": 1,
      "b": false,          // sell
      "p": "57900",        // limit price after trigger
      "s": "1",
      "tif": "Gtc",
      "triggerPx": "58000", // trigger level
      "triggerCondition": "Below" // activates when price goes below 58k
    }]
  }
}

```

#### Key parameters:

`triggerPx`, `triggerCondition` define the activation condition.

### 4. Post-Only Order (Add Liquidity Only)

Guarantees your order **adds liquidity** — cancels if it would fill immediately.

**Use case:** Trader wants to place a limit buy that *sits* on the book, not execute right away.

**Example:**

```
{
  "action": {
    "type": "order",
    "orders": [{
      "a": 1,
      "b": true,
      "p": "59950", // slightly below market to rest on the book
      "s": "1",
      "tif": "Alo" // Add-Liquidity-Only (Post-only)
    }]
  }
}
```

**Key parameter:**

`tif:"Alo"` — tells the system “cancel if this would execute immediately.”

---

## 5. Reduce-Only Order

Closes or reduces an existing position without opening a new one.

**Use case:** Trader is long 1 BTC and wants to close it at \$61,000 safely.

**Example:**

```
{
  "action": {
    "type": "order",
    "orders": [{
      "a": 1,
      "b": false, // sell
      "p": "61000",
      "s": "1",
      "tif": "Gtc",
      "r": true // reduce-only flag
    }]
  }
}
```

**Key parameter:**

`"r": true` → executes only if it reduces the trader's position. If no position exists, the order cancels automatically.

---

## 6. TWAP (Time-Weighted Average Price) Order

Splits a large trade into smaller parts executed over time to achieve an average price.

**Use case:** Trader wants to buy 10 BTC gradually (1 BTC every minute at \$60,000).

**Example:**

```
{
  "action": {
    "type": "twap",
    "coin": "BTC",
    "isBuy": true,
    "totalSz": "10", // total size
    "sliceSz": "1", // each slice = 1 BTC
    "intervalMs": 60000, // 60 seconds between slices
    "limitPx": "60000",
    "tif": "Gtc"
  },
  "nonce": 17300000000000,
  "signature": "0x..."
}
```

**Key parameters:**

`type:"twap"` , `totalSz` , `sliceSz` , `intervalMs` , `limitPx` , `tif` .

### Summary Table

Order Type	Description	Key Parameters	Example Intent
<b>Limit</b>	Executes at target price or better	<code>p</code> , <code>s</code> , <code>tif:"Gtc"</code>	Buy BTC at \$60k
<b>Market</b>	Executes instantly	<code>tif:"loc"</code>	Enter immediately
<b>Stop / Trigger</b>	Activates after trigger	<code>triggerPx</code> , <code>triggerCondition</code>	Sell if price < \$58k
<b>Post-Only</b>	Adds liquidity, cancels if would fill	<code>tif:"Alo"</code>	Maker order at \$59,950
<b>Reduce-Only</b>	Only closes existing position	<code>r:true</code>	Safely close long at \$61k
<b>TWAP</b>	Splits large orders over time	<code>sliceSz</code> , <code>intervalMs</code>	Buy 10 BTC in parts

**References:**

- [Hyperliquid Exchange API](#)
- [Hyperliquid Order Types](#)

4. **What are the rate limits for API requests? How would you design a system that respects these limits while maximizing data throughput?**

Hyperliquid enforces rate limits to ensure fair access and system stability. Each IP address has a total budget of approximately **1200 request weight per minute**. Every API call consumes a certain "weight" based on how heavy the request is

**Designing a system to respect limits:**

- Use **WebSockets** for real-time data instead of frequent REST polling.
- Implement a **request queue** or **token bucket** that tracks your remaining weight per minute.
- **Paginate** large data requests using time ranges (Example: 500 items at a time).
- **Batch multiple orders** into one API call to save weight.
- Add **backoff and retry logic** when receiving HTTP 429 errors.
- Cache frequently used data locally to avoid duplicate calls.

**Reference:** [Hyperliquid Rate Limits Documentation](#)

## 1.2 HIP-3 Understanding

**Task:** Read about HIP-3 and answer the following:

1. **Explain HIP-3 in your own words (2-3 paragraphs). What problem does it solve?**

HIP-3 (Hyperliquid Improvement Proposal 3) brought in a **Vault and Delegation System**. This system lets normal users hand over their trading funds to skilled managers through on-chain vaults. Instead of every person making trades on their own, professional traders can run strategies that others can join. All the trades, balances, and profit-sharing are visible on the blockchain, so everything is transparent.

Before HIP-3, people could only trade by themselves. This was hard for beginners because they had to manage their own risk. With HIP-3, users can now delegate their funds in a **non-custodial way**. This means their money stays safe in smart contracts while managers make trades for them. Managers get more capital to use, and users get the benefit of expert trading without giving up ownership of their assets.

In conclusion HIP-3 solves the problem of trust and difficulty in trading. It creates a clear and safe system for users to follow pro traders. This makes trading on Hyperliquid more open, secure, and fair for both experts and everyday users.

2. **What are the implications of HIP-3 for developers building applications on top of Hyperliquid? How does it affect system design decisions?**

HIP-3 brings a new feature to Hyperliquid called **vaults** and **delegation**. This means users can let professional traders manage their funds through special on-chain accounts called vaults. Because of this, developers now have to build apps that can show things like vault

performance, user deposits, withdrawals, and how profits are shared — all in a clear and honest way.

Before HIP-3, apps only needed to support single traders. Now, developers must design for a **social trading system**, where many users can join a vault and follow a manager's strategy. This includes building dashboards that show live updates, track user shares, and display how fees or profits are divided.

To summarize, HIP-3 makes developers focus more on **trust and transparency**. Systems must clearly show what's happening on-chain so users can see where their money goes. It's not just about trading anymore — it's about building tools that are easy to understand, fair to everyone, and keep users' funds safe.

### 3. Are there any potential challenges or limitations you foresee when working with HIP-3 in production?




HIP-3 is great but it does bring issues to it. One of the issues is how to **track vault funds correctly**. Vaults handle deposits, withdrawals, profits, and fees, so developers must make sure all the numbers are accurate. Even small mistakes in profit sharing or fee calculation could make users lose trust in the system.

Another challenge is **risk control**. When users let vault managers trade for them, they also take on the risk of the manager's decisions. If a manager makes bad trades or uses too much leverage, users could lose money. This means systems must show everything clearly and have ways to warn or limit risky behavior.

Finally, **security and scaling** are also concerns. Vaults create more on-chain data and need stronger systems to handle it. The smart contracts running these vaults must be checked carefully so hackers can't steal funds. In some countries, there might also be **legal rules** about managing other people's money.

In short, the main challenges of HIP-3 are **fair tracking, risk management, security, and legal safety**. Developers need to build systems that solve these problems while keeping everything clear and easy for users to understand.

### Reference

1. Official documentation explaining vault logic, manager responsibilities, and slashing risks.  
 [Hyperliquid Gitbook – HIP-3](#)
2. Highlights risks like oracle attacks, governance slashing, and liquidity fragmentation.  
 [CCN Article](#)
3. Talks about the opportunities of HIP-3 but also the “price of freedom,” including regulatory and governance risks.  
 [CoinRank Article](#)



4. Points out infrastructure scaling challenges and compliance concerns as the protocol grows.

 [PanewsLab Article](#)

### 1.3 Subgraph Provider Comparison

**Research Task:** Compare the following subgraph/indexing providers for building a high-performance, low-latency DeFAI system:

- The Graph
- Goldsky
- Ponder
- Envio

**Requirements:**

1. Create a decision matrix comparing these providers across relevant dimensions (e.g., latency, cost, ease of integration, scalability, reliability)
2. Which would you choose for a system that requires:
  - Real-time data ingestion (<500ms lag)
  - High query throughput (1000+ queries/min)
  - Cost efficiency
  - Reliable uptime for production trading
3. Document your decision-making process. What trade-offs are you making?

**Deliverable:** A markdown document with your research findings, decision matrix, and final recommendation with justification.

#### Decision Matrix - Subgraph/Indexing providers

Provider	Latency (Data Freshness)	Cost Model	Ease of Integration	Scalability (Query Throughput)	Reliability (Uptime & Support)
<b>The Graph</b>	Medium (seconds to minutes) – decentralized nodes mean syncing can lag behind real-time	Pay-per-query on decentralized network; free on hosted service (deprecated soon)	Mature tooling (GraphQL, Subgraph Studio), lots of docs, but steeper learning curve	High (global decentralized infra, many queries supported)	Good (widely used, but reliant on indexer quality; uptime depends on network)

Provider	Latency (Data Freshness)	Cost Model	Ease of Integration	Scalability (Query Throughput)	Reliability (Uptime & Support)
<b>Goldsky</b>	Low (sub-second to <1s) – optimized infra for real-time dashboards	Subscription or usage-based pricing; generally higher than The Graph but predictable	Easy integration (GraphQL, APIs, managed infra) – “The Graph without the headaches”	Very High (handles 1000s of queries/min easily)	Excellent (managed infra, SLAs, enterprise-grade support)
<b>Ponder</b>	Low to Medium (depends on your infra; typically ~1s to few seconds)	Free (open-source) but infra costs on you (hosting DB, scaling servers)	Very dev-friendly (TypeScript/Node.js, schema-first), flexible mappings	Medium to High (scales with your infra; no managed scaling)	Depends on your infra – can be reliable if well maintained, but no managed support
<b>Envio</b>	Very Low (sub-500ms latency, designed for real-time DeFi apps)	Early stage: pricing still evolving; infra costs may be higher than The Graph but lower than full DIY	Modern, simple setup (focus on real-time use cases), but smaller ecosystem than The Graph	High (built for trading/data-heavy apps, parallel processing)	Strong focus on reliability, but newer ecosystem means fewer guarantees compared to Goldsky

## Which is best?

### Evaluation Against Requirements

#### 1. The Graph

- ❌ Latency: Too slow for sub-500ms (usually seconds to minutes).
- ✅ Throughput: Can scale via decentralized indexers.
- ✅ Cost: Pay-per-query can be cheap if usage is low, but gets costly at scale.
- ⚠️ Reliability: Decentralized network = variable uptime depending on indexers.  
→ **Not suitable for real-time trading use cases.**

#### 2. Goldsky

- ✅ Latency: Sub-second, good for dashboards but may not always hit <500ms.
- ✅ Throughput: Designed for large query volumes.
- ⚠️ Cost: Subscription-based, generally more expensive than DIY.
- ✅ Reliability: Strong uptime, SLAs, enterprise support.

→ **Great balance of performance and reliability, but slightly higher cost.**

### 3. Ponder

- ▲ Latency: Depends on your infra (usually 1–2s), unlikely to guarantee <500ms without heavy tuning.
- ▲ Throughput: Scales with your servers, but you must manage infra.
- ✅ Cost: Open-source = free software, but infra costs add up at high throughput.
- ▲ Reliability: Only as good as your infra and monitoring.

→ **Flexible and cheap for devs, but not strong enough for mission-critical trading.**

### 4. Envio

- ✅ Latency: Designed for **sub-500ms real-time indexing** → best fit for DeFi/DeFAI.
- ✅ Throughput: Parallelized architecture handles 1000+ queries/min.
- ▲ Cost: Newer provider, pricing not fully standardized; may be higher than The Graph but lower than enterprise Goldsky.
- ▲ Reliability: Early ecosystem, less proven than Goldsky, but marketed for production-grade apps.

→ **Strong candidate for production trading requiring ultra-low latency.**

## Recommendation

For a **DeFAI system that requires <500ms latency, high throughput, and reliable uptime**, I would choose **Envio** as the primary provider. It is explicitly designed for **real-time DeFi indexing**, making it the best fit for latency-sensitive applications like trading bots or execution dashboards.

However, I would also consider a **hybrid approach**:

- **Envio** for real-time trading feeds (sub-500ms ingestion).
- **Goldsky** as a backup / secondary indexer for historical data and stability (since it has stronger SLAs and ecosystem support).

## Why?

hybrid increases cost, but for a trading system where downtime or data lag could cause losses, the trade-off is worth it. To control costs, the hybrid setup can split responsibilities, Envio for real-time low-latency feeds, and Goldsky for historical data and redundancy.