İş Akışı

1. **Eldiven Tasarımı**
   Esnek eldiven kullanılacak.
2. **IMU Sensörlerinin Yerleşimi**
   Her bir IMU sensörü için 3D yazıcıda küçük ve hafif montaj yuvaları tasarlanacak. Bu yuvaların amacı sensörlerin parmaklara düzgün şekilde konumlandırılmasını sağlamaktır.

3. **Montaj ve Sabitleme**
   Ana 10-DoF sensör el sırtına, parmak IMU'ları ise ilgili kemiklerin üzerine ve sonrasında uygun şekilde dikilerek monte edilir.

Donanım Testleri

Montajdan önce her bir sensörü (özellikle I²C adreslerini kontrol ederek) ve I²C çoklayıcıları tek tek test edip çalıştıklarından emin olmak gerekir.

Veri Toplama Yazılımı

1. ESP-32 ile tüm sensörler I²C arabirimi üzerinden bağlanır.

2. Tüm sensörlerden (örneğin el sırtındaki 10-DoF sensör ve 5 parmak sensörü) gelen ham veriler toplanır ve işlenmeden temel bir kod üzerinden okunur.

Not: Ana sensörlerden ek olarak manyetometre ve barometre verileri de okunur.

Veri İşleme ve Sensör Füzyonu

Projenin en kritik adımıdır. Her bir IMU sensörü için ayrı ayrı sensör füzyon algoritmaları çalıştırılmalıdır.
Örnek algoritma: **Madgwick filtresi**.
Bu filtre sayesinde sensörlerden gelen verilerle stabil yönelim ölçümleri (quaternion veya Euler açıları) hesaplanır.

Kinematik Modelle ve Açı Hesabı

1. Parmaklardaki sensörlerden alınan yönelim verileri kullanılacak.
2. Bu yönelimler arasındaki fark hesaplanarak parmak eklem açıları çıkarılacak.
   Bu açıların matematiksel modeli sonraki adımlarda kullanılacaktır.

Entegrasyon ve Kalibrasyon

1. Sistemi birleştirmek.
2. Başlangıçta doğru pozisyonu tanıyabilmek için bir **kalibrasyon rutini** oluşturmak gerekir.
   (Örnek: Kullanıcıdan elini düz şekilde tutması istenip sensörler sıfırlanır.)

Veri Görselleştirme

1. ESP-32'den gelen işlenmiş veriler Wi-Fi üzerinden gönderilecek.
2. **Processing**, **Unity** veya **Python (vPython kütüphanesi)** ortamlarında bu veriler kullanılarak zamanla değişen 3B bir modelleme yapılabilir.

Malzeme Listesi

1. Ana Kontrolcü / Mikrodenetleyici → **ESP-32**
2. El üstü sensörü → **10-DoF IMU**
3. Parmak eklem sensörleri → **6-DoF IMU**
4. Haberleşme → **I²C çoklayıcı**
5. Eldiven
6. Sensör kutuları (3D yazıcı)
7. Kablo / İletişim hattı
8. Li-Po batarya

Sistemin Genel Mimarisi

1. Tanımlayıcı Katman (Interpreter)

Modelin görevi eldivenden gelen karmaşık anlık sensör verisini analiz edip "Bu hangi işaret?" sorusuna cevap vermektir.

2. Bağlamsal Katman (Contextual)

Bu sistem, tanımlanan işaretleri anlamlı bir yapıya oturtur.
Neo4J graf veritabanı bu noktada devreye girer.
Görevi:

"Bu işaretten sonra hangi işaret gelebilir?"
"Bu cümle dil bilgisi açısından mantıklı mı?"
Bu süreç tahmin ve doğrulama problemi olarak çalışır.

3. GraphQL API Katmanı

Tüm bu yapıyı dış dünyaya (mobil/web uygulama) açan arabirimdir.
FastAPI üzerinde çalışan GraphQL servisi, esnek ve sorgulanabilir bir arayüz sağlar.

Teknoloji Seçimleri

- **Donanım Programlama:** C++ (Arduino Framework, eldiven tarafı)
- **Veri İletimi:** MQTT veya WebSockets
- **Backend Servisleri:** Python
    - Web framework: **FastAPI (asenkron yapı)**
    - GraphQL entegrasyonu: **Strawberry** veya **Ariadne**
- **Yapay Zeka:** TensorFlow (Keras) veya PyTorch
- **Veritabanı:** Neo4J

Veri Akışı

Sensör Füzyonu Cihaz Üzerinde

ESP-32, 11 adet IMU'dan gelen ham verileri (ivmeölçer, jiroskop vb.) doğrudan göndermez.
Bunun yerine her bir sensör için cihaz üzerinde (**on-device**) sensör füzyonu yapılır.
Bu sayede her sensörün kararlı yönelim verileri (quaternion veya Euler açıları) elde edilir.

Veri Paketleme

ESP-32, belirli aralıklarla (örneğin her 20 milisaniyede bir) tüm sensörlerden gelen işlenmiş verileri tek bir JSON nesnesi içinde paketler.

JSON yapısı örneği:

```
{
  "timestamp": 1665495830.9,
  "hand_Imu": { "qW": 0.7, "qX": 0.1, "...": "...", "altitude": 452.3 },
  "finger_Imus": [
    { "id": 0, "qW": 0.8, "..." },
    { "id": 1, "qW": 0.75, "..." }
    // 9 sensör daha
  ]
}
```

Veri Yayınlama

Bu JSON paketi Wi-Fi üzerinden bir **MQTT broker'a** veya **WebSocket sunucusuna** gönderilir.

Veri Alımı ve İşaret Tanımlama (Python Backend + AI)

Veri Dinleme

Python backend tarafında bir MQTT (veya WebSocket) istemcisi sürekli olarak ilgili **topic**'i dinler ve gelen veri paketlerini alır.

Pencereleme (Windowing)

Gelen anlık veri akışı, işaret tanıma modelinin anlayabileceği sabit boyutlu "pencerelere" bölünür.
Örneğin her 2 saniyelik (100 örnek noktası) veri biriktirilerek bir pencere oluşturulur.

Yapay Zeka Modeli (LSTM / Transformer)

Bu görev için sıralı veri işlemede başarılı olan **LSTM (Long Short-Term Memory)** veya daha modern bir **Transformer** tabanlı sinir ağı modeli kullanılır.

Model Girdisi:
Bir pencereye ait (örneğin 100 zaman adımı × N özellik) boyutunda tensör.
Burada *N*, tüm sensörlerden gelen toplam özellik sayısıdır.

Model Çıktısı:
Sistemde tanımlı tüm işaretler için bir olasılık dağılımı vektörü.
Örneğin:
`[0.02, 0.95, 0.01, ...] → %95 "Merhaba", %2 "Ben"`

İşaret Tespiti:
En yüksek olasılığa sahip olan işaret "tespit edilen işaret" olarak belirlenir ve bir sonraki katmana gönderilir.

Anlamsal Analiz ve Cümle Tahmini (Neo4J Graph)

Graf Veritabanı Tasarımı

Neo4J'de başarı, doğru modelleme ile sağlanır.

Düğümler (Nodes)

- **Sign:** Her bir tekil işareti temsil eder.
  Özellikler: `{ name: "Merhaba", id: "1", lang: "hello" }`

- **Category:** İşaretin dilbilgisel kategorisini temsil eder.
  Özellikler: `{ name: "İsim" }` veya `{ name: "Fiil" }`

Kenarlar (Edges)

- **NEXT_SIGN:** Bir işaretten sonra gelme olasılığı olan diğer işareti bağlar.
  Özellikler: `weight` (ağırlık) veya `probability` (olasılık).

Cümle Tahmini

AI modeli "Ben" işaretini tespit ettiğinde backend, Neo4J'ye şu **Cypher** sorgusunu gönderir:

```
// "Ben" işaretinden sonra gelme olasılığı en yüksek 5 işareti getir.
MATCH (current:Sign { name: "Ben" })-[:NEXT_SIGN]->(next:Sign)
RETURN next.name, r.probability
ORDER BY r.probability DESC
LIMIT 5
```

Bu sorgunun sonucu Neo4J'den örneğin:

```
["gidiyorum", "geldim", "ismim", "mutluyum"]
```

şeklinde dönebilir.
Bu liste kullanıcıya "otomatik tamamlama" önerisi olarak sunulabilir.

API Katmanı (GraphQL)

Bu katman tüm bu karmaşık yapıyı soyutlayarak FastAPI üzerinde çalışan bir **GraphQL sunucusu** olarak dış dünyaya açar.

GraphQL Şeması

```
type Sign {
  name: String!
  probability: Float!
}

type Query {
  # Belirli bir işaretten sonraki olası işareti tahmin et
  predictNextSign(currentSign: String!): [Sign]
}

type Mutation {
  # Anlık sensör verisi penceresini al, hangi işaret olduğunu tanı
  recognizeSign(sensorWindow: [[Float!]!]!): Sign
}
```

Resolver Fonksiyonları

- **predictNextSign resolver'i:**
  Gelen `currentSign` argümanını alır, Neo4J'ye Cypher sorgusu gönderir ve sonucu döndürür.

- **recognizeSign resolver'i:**
  Gelen sensör verisi penceresini alır, LSTM/Transformer modeline verir ve en olası işareti döndürür.

Yapay Zeka Eğitim Süreci (Offline İşlemler)

Sistemin çalışması için öncelikle AI modellerinin eğitilmesi ve Neo4J veritabanının doldurulması gerekir.

Veri Seti Oluşturma

Tanımlanacak her işaret (örneğin 100 farklı işaret) için, farklı kişiler tarafından onlarca kez tekrarlanan sensör verileri kaydedilmelidir.

Her kayıt:

- 1 pencere (örneğin 2 saniyelik veri)

- 1 etiket (örneğin `"label": "Merhaba"`) içerir.

Bu yapı **denetimli öğrenme (supervised learning)** için gereklidir.

İşaret Tanıma Modelinin Eğitimi

Toplanan bu büyük veri seti, **Keras** veya **PyTorch** kullanılarak tasarlanan LSTM/Transformer modeliyle eğitilir.
Model, sensör verisi desenleri ile işaret etiketleri arasındaki ilişkiyi öğrenir.

Neo4J Grafının Doldurulması

Bu aşama model eğitiminden farklıdır.
Türk İşaret Dili'nin dilbilgisi kuralları ve kelime kullanım sıklıkları analiz edilir.

Büyük metin veya işaret dili çeviri veri setleri incelenir, hangi kelimeden sonra hangisinin geldiği istatistiksel olarak çıkarılır.
Bu istatistikler, Neo4J'deki `Sign` nodeları arasındaki `NEXT_SIGN` ilişkilerinin `probability` özelliklerini oluşturmak için kullanılır.
Bu işlem Python script'iyle otomatik olarak yapılabilir.