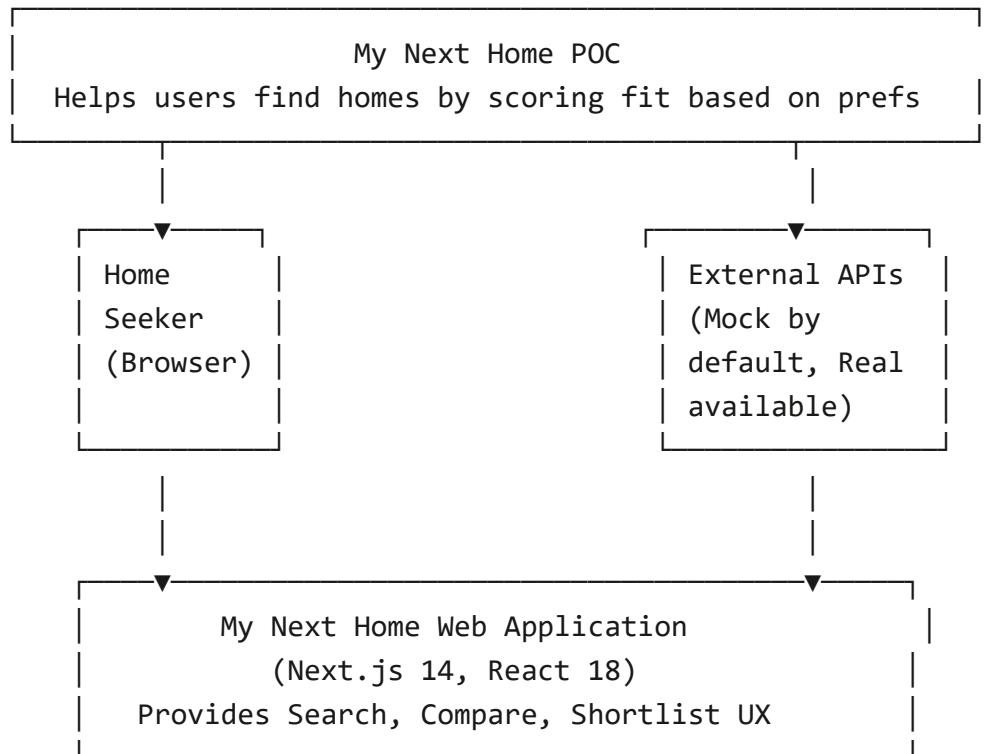


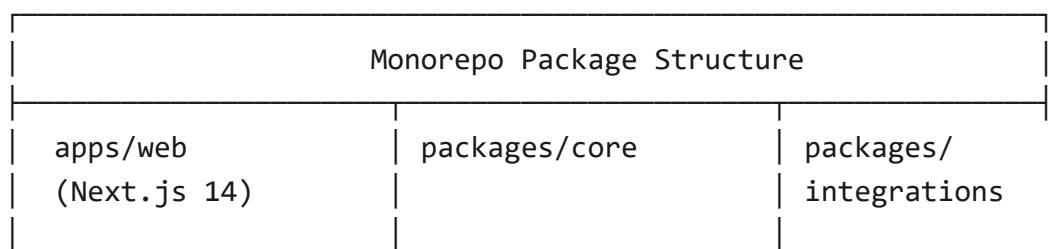


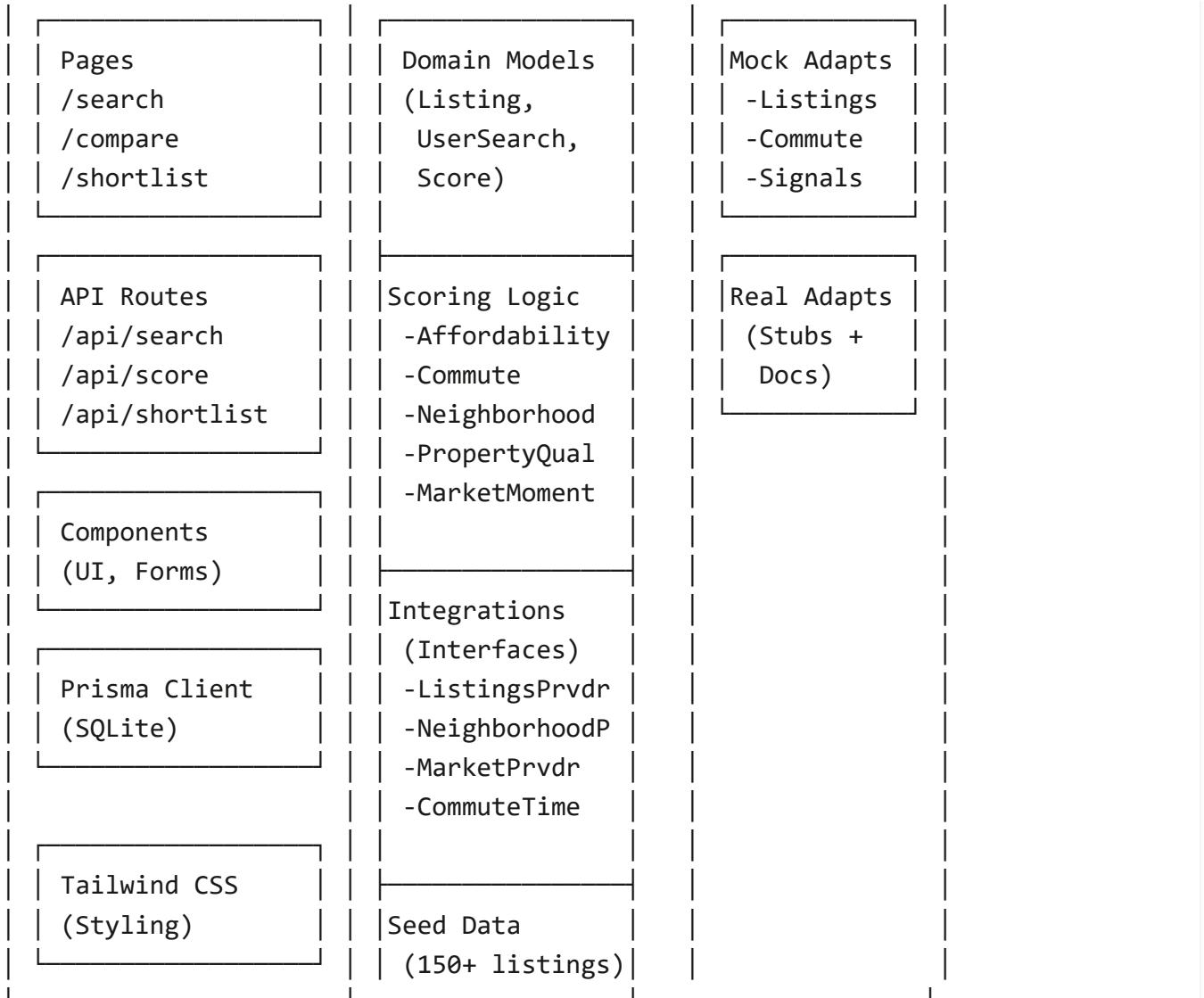
# Architecture - My Next Home POC

## C4 Context Diagram



## C4 Container Diagram



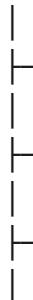


## Data Flow: Search

User fills search form



POST /api/search { location, budget, beds, baths, mustHaves }



→ Zod validation

→ getListingsProvider() [Mock by default]

→ Filter ALL\_LISTINGS in memory:

- Price range

```
- Beds/baths  
- Location (zip/city match)  
- Must-haves (feature match)  
  
→ Sort by price ascending  
  
→ Limit to 20 results  
  
▼  
Return { listings: Listing[] }  
|  
▼  
Browser renders listing cards with checkboxes
```

## Data Flow: Scoring (Compare)

```
User selects 2-4 homes and clicks "Compare"  
|  
▼  
POST /api/score { listingIds[], search }  
|  
|→ Zod validation  
  
|→ For each listing:  
| |  
| |→ Fetch from Listings Provider  
  
| |→ getNeighborhoodSignalsProvider()  
| | Returns: { schoolRating, safetyIndex, walkability }  
  
| |→ getMarketSignalsProvider()  
| | Returns: { daysOnMarket, yoyPriceChange%, inventoryLevel }  
  
| |→ getCommuteTimeProvider()  
| | Returns: estimated minutes  
  
| |→ scoreListings(listing, search, signals, market, commute)  
| | Computes: 5 subscores + overall + 3 reasons  
  
| |→ Store score breakdown
```

```
Return { listings: Listing[], scores: ScoreBreakdown[] }
```

Browser renders comparison table + detailed score cards

# Data Flow: Shortlist

User clicks "Shortlist" on a home

```
POST /api/shortlist { listingId, search }
```

→ Compute score (same as above)

→ Save to Prisma:

ShortlistedHome {

`id: string`

listingId: string

listingJson: List

scoreJson: Scor

GET /api/shortlist → Returns all saved items

Browser renders shortlist cards with delete button

`DELETE /api/shortlist/:id` → Delete item from DB

# Scoring Rules in Detail

## Affordability Score (0-100)

### Monthly Payment Estimate:

```
Principal = price * (1 - down%) [default 20%]
Monthly = Principal * [r(1+r)^n] / [(1+r)^n - 1]
where r = annual_rate/12, n = years*12
```

```
+ Property taxes (estimated or provided)
+ Insurance (estimated or provided)
+ HOA monthly fee
= Total Monthly Payment
```

### Score Calculation:

```
DTI_safe = (budgetMin + budgetMax)/2 / 360 * 0.25 [25% of monthly)
DTI_max = (budgetMin + budgetMax)/2 / 360 * 0.33 [33% DTI threshold]
```

```
if monthly > DTI_max:
    score = 0
else if monthly <= DTI_safe:
    score = 100
else:
    score = ((DTI_max - monthly) / (DTI_max - DTI_safe)) * 100
```

### Reasons:

- If >= 75: "Monthly payment fits comfortably in budget"
- If >= 50: "Monthly payment within acceptable range"
- Else: "Monthly payment may stretch budget"

## Commute Score (0-100)

```
If commuteMaxMinutes is null:
    score = 75 [neutral]

else:
    buffer = commuteMaxMinutes * 0.2 [20% is ideal]

    if actualMinutes <= commuteMaxMinutes - buffer:
        score = 100
    else if actualMinutes <= commuteMaxMinutes:
        score = 100 - ((actualMinutes - (max - buffer)) / buffer) * 25
    else:
        score = max(0, 75 - (actualMinutes - max) * 5)
```

#### Reasons:

- Well under target: "Commute is well under X minute target"
- Near target: "Commute near target of X minutes"
- Over target: "Commute exceeds X minute target by Y min"

## Neighborhood Score (0-100)

#### Weight by risk tolerance:

LOW: schools 40%, safety 40%, walkability 20%

MEDIUM: schools 40%, safety 30%, walkability 30%

HIGH: schools 40%, safety 20%, walkability 40%

```
Score = schoolRating/10*100 * schoolWeight  
      + safetyIndex * safetyWeight  
      + walkability * walkabilityWeight
```

#### Reasons:

- Strong: "Strong schools, safety, and walkability"
- Good: "Good neighborhood profile"
- Mixed: "Neighborhood signals are mixed"

## Property Quality Score (0-100)

```
baseline = 50  
  
if yearBuilt is recent (< 5 years): +20  
if yearBuilt is newer (5-20 years): +10  
if yearBuilt is old (> 50 years): -15  
  
if sqft > 2500: +10  
if sqft < 1200: -5  
  
features_bonus = min(feature_count * 5, 15)  
  
if propertyType == SINGLE_FAMILY: +5  
  
score = min(100, max(0, baseline + adjustments))
```

#### Reasons:

- Recent, large, features: "3bd/2ba, 2000sqft with good features"
- Average: "3bd/2ba with adequate size and features"
- Limited: "Property is functional but may have limited appeal"

## Market Momentum Score (0-100)

```
baseline = 50
```

#### Days on Market:

```
if <= 14 days: +10 [hot market]  
if > 60 days: +15 [buyer friendly]  
if > 30 days: +5
```

#### Inventory:

```
if HIGH: +15  
if LOW: -15
```

#### Price Change YoY:

```
if < -2%: +15 [buyer market]  
if < 2%: +5 [stable]  
if > 5%: -10 [seller market]
```

```
score = min(100, max(0, baseline + adjustments))
```

#### Reasons:

- High: "Buyer-friendly market conditions"
- Neutral: "Market conditions are neutral"
- Low: "Competitive seller market"

## Overall Score

```
overall = affordability * 0.25  
        + commute * 0.20  
        + neighborhood * 0.25  
        + propertyQuality * 0.20  
        + marketMomentum * 0.10
```

[Default weights sum to 1; can be overridden and re-normalized]

Top 3 Reasons: Select by highest subscores and include each score's reason

# Integration Architecture

## Interfaces (packages/core)

```
interface ListingsProvider {
  search(query: UserSearch): Promise<Listing[]>
  getById(id: string): Promise<Listing | null>
}

interface NeighborhoodSignalsProvider {
  getSignals(zip, city, state): Promise<NeighborhoodSignals>
  // { schoolRating: 1-10, safetyIndex: 1-100, walkability: 1-100 }
}

interface MarketSignalsProvider {
  getSignals(zip, city, state): Promise<MarketSignals>
  // { medianDaysOnMarket, yoyPriceChangePct, inventoryLevel }
}

interface CommuteTimeProvider {
  getEstimatedMinutes(fromLat, fromLng, toAddress): Promise<number>
}
```

## Mock Adapters (packages/integrations/mock)

- **MockListingsProvider:** Filters ALL\_LISTINGS in-memory by criteria
- **MockNeighborhoodSignalsProvider:** Deterministic generation from zip hash + city rules
- **MockMarketSignalsProvider:** Deterministic generation from zip hash
- **MockCommuteTimeProvider:** Simple distance-based estimate

All deterministic → reproducible test results.

## Real Adapters (packages/integrations/real)

Skeleton implementations with TODO comments:

- **RealListingsProvider**: RESO Web API / MLS partnership
- **RealNeighborhoodSignalsProvider**: GreatSchools, crime data APIs
- **RealMarketSignalsProvider**: Redfin, FHFA, local MLS stats
- **RealCommuteTimeProvider**: Google Maps / Mapbox

See [docs/05-deployment.md](#) for full integration guides.

## Database Schema

### ShortlistedHome (Prisma)

```
model ShortlistedHome {  
    id          String    @id @default(cuid())  
    listingId   String  
    listingJson  Json      // Full Listing object (denormalized)  
    scoreJson    Json      // Full ScoreBreakdown (denormalized)  
    createdAt    DateTime @default(now())  
    updatedAt    DateTime @updatedAt  
  
    @@index([listingId])  
}
```

### Production Normalized Schema (Reference)

```
Listing  
  id: UUID  
  addressMasked: string  
  price: decimal  
  beds: int  
  baths: decimal  
  sqft: int  
  [... other fields]
```

```
NeighborhoodSignal
```

```
zip: string
city: string
state: string
schoolRating: float
safetyIndex: float
walkability: float
lastUpdated: timestamp
TTL: 5 days (cron refresh)
```

```
MarketSignal
zip: string
city: string
state: string
daysOnMarket: int
yoYPriceChange: float
inventoryLevel: enum
lastUpdated: timestamp
TTL: 7 days
```

```
ShortlistedHome (normalized)
id: UUID
userId: UUID (if adding auth)
listingId: UUID (FK)
createdAt: timestamp
```

```
Score
id: UUID
listingId: UUID
userId: UUID
searchContext: JSON (what params generated this score)
affordabilityScore: int
commuteScore: int
[... other scores]
overallScore: int
reasons: text[]
createdAt: timestamp
```

## Deployment Strategy

### POC (Vercel + SQLite)

Browser → Vercel Edge Functions → Next.js API Routes → Prisma → SQLite

Works for < 1000 users, single region.

## Production (Vercel + PostgreSQL + Caching)

```
Browser → Vercel Edge
  ↓
  Next.js API
  ↓
  Redis Cache (1hr TTL on search results, 5day on signals)
  ↓
  PostgreSQL (Supabase/Neon) + Read Replica
  ↓
  Cron Job: Refresh neighborhood/market signals nightly
  Cron Job: Refresh popular search results cache
```

## Error Handling

All API responses follow consistent schema:

### Success (200)

```
{ "data": { ... } }
```

### Error (400/404/500)

```
{
  "error": {
    "code": "VALIDATION_ERROR|NOT_FOUND|INTERNAL_ERROR",
    "message": "Human-readable error message",
    "details": [ ... ] // Optional, for validation errors
  }
}
```

# Security Considerations

- Zod validation on all POST bodies
- Address masking (privacy by default)
- No authentication (POC; can add OAuth)
- No secrets in code (use .env)
- CORS headers (if API exposed publicly later)
- Rate limiting (TODO for production)
- SQL injection protection (Prisma uses prepared statements)

# Performance Targets

Operation	Target	Notes
Search (20 results)	< 500ms	In-memory filtering
Score 4 homes	< 1s	Parallel signal fetches
Page load (3G)	< 2s	Includes Tailwind CSS, React hydration
List shortlist	< 100ms	Direct DB query
Add to shortlist	< 2s	Includes scoring

# Testing Strategy

See [docs/04-testing.md](#) for detailed test plan, but briefly:

- **Unit:** Vitest, 70%+ coverage
  - Mortgage math
  - Scoring rules

- Weight normalization
- Reason generation

- **Integration:** API route tests

- /api/search returns filtered results
- /api/score returns deterministic scores

- **E2E:** Playwright

- Search → compare → shortlist
- Delete from shortlist
- Page navigation

- **Manual:** Visual inspection

- UI responsiveness (mobile/desktop)
- Error messages
- Data accuracy