Executive Summary

The **Dynamic VLM CAPTCHA System** is a three-tier web application designed to evaluate Vision-Language Model (VLM) capabilities through animated counting challenges. The system generates synthetic multi-frame sequences containing geometric shapes that move and interact, then poses questions requiring either single-frame spatial reasoning (static tasks) or temporal analysis across the animation sequence (dynamic tasks).

Core objectives:

- Benchmark VLM accuracy on controlled visual reasoning tasks
- Support multiple VLM providers (Gemini, OpenRouter, GPT-4V, local models)
- Provide both automated evaluation and interactive human testing
- Scale to hundreds of challenge generations per minute
- Maintain deterministic ground truth for answer validation

Key metrics:

- Challenge generation: <200ms per 6-frame sequence
- VLM inference: 1-3s latency (external API dependent)
- Target accuracy: >80% for static tasks, >60% for dynamic tasks
- Deployment: Docker Compose, single-command startup

System Architecture

High-Level Component Diagram

```
┌──────────────────────────────────────────────────────────────┐
│                       Frontend Layer                          │
│  React + TypeScript + HTML5 Canvas                           │
│  • Renders animated CAPTCHA sequences                        │
│  • Displays questions and collects answers                   │
│  • Optional client-side demo mode                            │
└──────────────────────────────────────────────────────────────┘
             │
             │ REST API (JSON)
             │ POST /captcha/new, /captcha/answer
             ▼
┌──────────────────────────────────────────────────────────────┐
│                       Backend API Layer                       │
│  Node.js + TypeScript + Express                              │
│  • Generates challenge metadata (shapes, trajectories)       │
│  • Orchestrates VLM evaluation requests                      │
│  • Validates answers against ground truth                    │
│  • Manages sessions and benchmarking                         │
└──────────────────────────────────────────────────────────────┘
             │
             │ HTTP POST (JSON + base64 frames)
             │ /vlm/predict
             ▼
┌──────────────────────────────────────────────────────────────┐
│                     VLM Inference Service                     │
│  Python + FastAPI                                            │
│  • Wraps Gemini, OpenRouter, GPT-4V SDKs                    │
│  • Handles multi-frame vision API calls                      │
│  • Implements provider fallback chain                        │
│  • Returns parsed integer predictions                        │
└──────────────────────────────────────────────────────────────┘
```

Technology Stack Rationale

Component	Technology	Justification
Frontend	React + TypeScript	Modern, component-based UI; strong typing for shape metadata; excellent Canvas API integration
Backend	Node.js + TypeScript	Shared language with frontend reduces duplication; async I/O perfect for orchestration; mature ecosystem
VLM Service	Python + FastAPI	Best SDK support for Gemini/OpenRouter/GPT-4V; existing PIL codebase reusable; optional GPU inference later
Rendering (Client)	HTML5 Canvas 2D	Browser-native, hardware-accelerated, no plugin required; sufficient for 192×192 shapes
Rendering (Server)	skia-canvas (Node)	Rust-backed, faster than node-canvas; deterministic output for ground truth; easy Docker build
Data Format	JSON + base64 images	Universal browser/server support; simpler than multipart for initial MVP; 33% overhead acceptable
Persistence	Redis (optional)	In-memory session store for benchmarking state; can be replaced with in-process Map for MVP
Deployment	Docker + Compose	Reproducible builds; isolated Python/Node environments; single-command startup

---

Component Architecture

1. Frontend (React + TypeScript)

Responsibilities:

- Display animated CAPTCHA (Canvas-rendered frames or GIF playback)
- Show task question and collect user/VLM input
- Send answers to backend for validation
- Optional: Render challenges locally for demo mode (using shared TS logic)

Key modules:

```
/frontend
├── src/
│   ├── components/
│   │   ├── CaptchaCanvas.tsx       # Canvas animation player
│   │   ├── ChallengeView.tsx       # Question display + answer input
│   │   └── BenchmarkDashboard.tsx  # Admin: view accuracy stats
│   ├── hooks/
│   │   └── useCaptcha.ts           # Fetch challenges from backend
│   ├── services/
│   │   └── apiClient.ts            # Axios wrapper for backend API
│   ├── types/
│   │   └── index.ts                # Imported from @shared/types
│   └── utils/
│       └── canvasRenderer.ts       # Client-side rendering (demo mode)
└── package.json
```

Design decisions:

- **Canvas vs GIF**: Canvas allows interactive controls (pause, step frame); GIF simpler but less flexible. **Choice: Canvas** for extensibility.
- **State management**: Simple useState for MVP; React Query for caching if scaling needed.
- **Styling**: Tailwind CSS for rapid prototyping; no heavy UI framework needed.

---

2. Backend API (Node.js + TypeScript)

Responsibilities:

- Generate challenge sequences (shapes, trajectories, noise, metadata)
- Select task type (static vs dynamic) and generate question
- Render frames server-side for ground truth (prevents client tampering)

- Call VLM service for predictions (if `eval_mode=vlm`)
- Validate answers and return correctness
- Support batch benchmarking with aggregated stats

Key modules:

```
/backend
├── src/
│   ├── models/
│   │   ├── Shape.ts              # Abstract base + Circle, Square, etc.
│   │   ├── GameObject.ts         # Animated object with position, velocity
│   │   └── AnimationSequence.ts  # Timeline container with frame metadata
│   ├── tasks/
│   │   ├── TaskBase.ts           # ITask interface
│   │   ├── StaticTasks.ts        # CountShape, CountColorShape, etc.
│   │   └── DynamicTasks.ts       # CountCrossMidline, CountEnterBox
│   ├── services/
│   │   ├── ChallengeGenerator.ts # Orchestrates sequence generation
│   │   ├── CanvasRenderer.ts     # skia-canvas wrapper
│   │   ├── NoiseGenerator.ts     # Speckles, lines, blur
│   │   └── VLMEvaluator.ts       # Multi-provider fallback logic
│   ├── providers/
│   │   ├── VLMProvider.ts        # Interface
│   │   ├── OpenRouterProvider.ts # HTTP client for OpenRouter
│   │   ├── GeminiProvider.ts     # HTTP client for Gemini
│   │   ├── StubProvider.ts       # Weighted random fallback
│   │   └── OracleProvider.ts     # Ground-truth oracle (testing)
│   ├── api/
│   │   ├── routes.ts             # Express routes
│   │   └── middleware.ts         # CORS, error handling, rate limiting
│   ├── utils/
│   │   ├── seedrng.ts            # Seedable PRNG (seedrandom library)
│   │   ├── extractInt.ts         # Robust integer extraction from LLM text
│   │   └── bbox.ts               # Collision detection helpers
│   └── types/
│       └── index.ts              # Re-exports from @shared/types
└── package.json
```

Design decisions:

- **Why TypeScript OOP here, not C#/Java?**
  - **Shared types with frontend**: No need to duplicate Shape/Task schemas across languages
  - **No performance gain**: VLM API latency (1-3s) dominates; rendering is <200ms even in Node
  - **Simpler deployment**: Single runtime (Node), no JVM or .NET CoreCLR
  - **Ecosystem maturity**: npm has mature Canvas, HTTP, and serialization libraries
- **Why server-side rendering?**
  - **Ground truth authority**: Client Canvas can be manipulated; server metadata is authoritative
  - **Deterministic validation**: Same seed → identical frames → reproducible benchmarks
- **Why separate VLM service call?**
  - **Isolation**: Model API changes don't require backend redeploy
  - **Scalability**: VLM service can run on GPU instances independently
  - **Multi-model support**: Easy to add new providers without changing core logic

---

3. VLM Inference Service (Python + FastAPI)

Responsibilities:

- Accept multi-frame prediction requests (question + frames)
- Route to appropriate provider (Gemini, OpenRouter, GPT-4V, Stub)
- Handle retries, timeouts, and error fallbacks
- Parse LLM text responses into integers
- Return predictions to backend

Key modules:

```
/vlm-service
├── app/
│   ├── main.py                # FastAPI app entry point
│   ├── models.py              # Pydantic request/response schemas
│   ├── providers/
│   │   ├── base.py            # VLMProvider abstract class
│   │   ├── gemini.py          # google-generativeai wrapper
│   │   ├── openrouter.py      # requests-based HTTP client
│   │   └── stub.py            # Weighted random for testing
│   └── utils.py               # extract_int, retry logic
├── requirements.txt
└── Dockerfile
```

Design decisions:

- **Why Python-only for this layer?**
  - **SDK availability**: Gemini, OpenRouter, OpenAI SDKs are Python-first
  - **Existing code reuse**: Current PIL-based implementation already in Python
  - **Optional GPU inference**: Later can add LLaVA, Qwen-VL with PyTorch
- **Why separate microservice, not embedded in backend?**
  - **Language isolation**: Backend stays pure TypeScript, no Python dependencies
  - **Independent scaling**: VLM calls are slow (1-3s); can deploy more VLM service replicas
  - **Testing flexibility**: Stub provider can run standalone without backend
- **API contract**: Simple JSON POST with base64 images (not multipart for MVP simplicity)

Data Models

Core TypeScript Interfaces

```
// @shared/types/index.ts

export type ShapeType =
  | "circle" | "square" | "triangle" | "rectangle"
  | "pentagon" | "hexagon" | "star" | "ellipse" | "diamond";

export type RGB = [number, number, number];

export interface Vec2 {
  x: number;
  y: number;
}

export interface BBox {
  x1: number;
  y1: number;
  x2: number;
  y2: number;
}

export interface GameObject {
  id: number;
  shape: ShapeType;
  color: RGB;
  position: Vec2;
  velocity: Vec2;
  size: number;
}

export interface FrameMetadata {
  frameIndex: number;
  objects: Array<{
    id: number;
    shape: ShapeType;
    color: RGB;
    bbox: BBox;
    center: Vec2;
```

```
  }>;
}

export interface AnimationMetadata {
  width: number;
  height: number;
  frames: number;
  fps: number;
  difficulty: "easy" | "medium" | "hard";
  seed: number;
  timeline: FrameMetadata[];
}

export interface Task {
  type: string; // e.g., "count_shape", "count_cross_midline"
  question: string;
  answer: number; // ground truth
}

export interface Challenge {
  id: string; // UUID
  animation: AnimationMetadata;
  task: Task;
  frames: string[]; // base64-encoded JPEG or URLs
  createdAt: string; // ISO timestamp
}

export interface ValidationResult {
  challengeId: string;
  prediction: number;
  correct: boolean;
  groundTruth: number;
}
```

Python Pydantic Models (VLM Service)

```
# /vlm-service/app/models.py

from pydantic import BaseModel
from typing import List

class PredictRequest(BaseModel):
    question: str
    frames: List[str]  # base64-encoded JPEG images
    model: str = "qwen/qwen2.5-vl-32b-instruct:free"
    temperature: float = 0.0
    max_tokens: int = 8

class PredictResponse(BaseModel):
    answer: int
    raw_text: str  # Original LLM response
    model_used: str
    latency_ms: int
```

API Contracts

- GET -> taking files / results, etc.
- POST -> sending safely results
- PUT -> edit
- DELETE -> delete

Backend API Endpoints

`POST /api/captcha/new`

Request:

```json
{
  "difficulty": "medium",
  "frames": 6,
  "taskType": "random", // or "static", "dynamic"
  "seed": 42 // optional, for reproducibility
}
```

Response:

```json
{
  "challenge": {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "animation": {
      "width": 192,
      "height": 192,
      "frames": 6,
      "fps": 12,
      "difficulty": "medium",
      "seed": 42,
      "timeline": [ /* FrameMetadata[] */ ]
    },
    "task": {
      "type": "count_cross_midline",
      "question": "Across the animation, how many circles crossed the vertical midline at least once?",
      "answer": 3
    },
    "frames": [
      "data:image/jpeg;base64,/9j/4AAQSkZJRg...",
      // ... 5 more base64 frames
    ],
    "createdAt": "2025-11-06T14:23:45Z"
  }
}
```

`POST /api/captcha/answer`

Request:

```json
{
  "challengeId": "550e8400-e29b-41d4-a716-446655440000",
  "prediction": 3,
  "evalMode": "vlm" // or "oracle", "stub"
}
```

Response:

```json
{
  "result": {
    "challengeId": "550e8400-e29b-41d4-a716-446655440000",
    "prediction": 3,
    "correct": true,
    "groundTruth": 3,
    "latencyMs": 1847
  }
}
```

`POST /api/captcha/benchmark`

Request:

```json
{
  "nStatic": 10,
  "nDynamic": 10,
  "difficulty": "medium",
  "frames": 6,
  "evalMode": "vlm"
}
```

Response:


```json
{
  "summary": {
    "totalTrials": 20,
    "correct": 14,
    "accuracy": 70.0,
    "avgLatencyMs": 1923,
    "byTaskType": {
      "static": { "accuracy": 80.0, "count": 10 },
      "dynamic": { "accuracy": 60.0, "count": 10 }
    }
  },
  "details": [
    {
      "taskType": "count_shape",
      "question": "How many circles are in the image?",
      "answer": 2,
      "prediction": 2,
      "correct": true
    }
    // ... 19 more
  ]
}
```

VLM Service API

`POST /vlm/predict`

Request:


```json
{
  "question": "How many circles crossed the vertical midline?",
  "frames": [
    "data:image/jpeg;base64,/9j/4AAQSkZJRg...",
    // ... up to 6 frames
  ],
  "model": "qwen/qwen2.5-vl-32b-instruct:free",
  "temperature": 0.0,
  "max_tokens": 8
}
```


Response:


```json
{
  "answer": 3,
  "raw_text": "3",
  "model_used": "qwen/qwen2.5-vl-32b-instruct:free",
  "latency_ms": 1847
}
```


`GET /vlm/models`

Response:


```json
{
  "available": [
    "qwen/qwen2.5-vl-32b-instruct:free",
    "meta-llama/llama-3.2-11b-vision-instruct:free",
    "gemini-flash-lite-latest",
    "gpt-4o-mini"
  ]
}
```

Deployment Architecture

Docker Compose Configuration

```yaml
# docker-compose.yml
version: '3.8'

services:
  frontend:
    build: ./frontend
    ports:
      - "3000:3000"
    environment:
      - REACT_APP_API_URL=http://localhost:4000
    depends_on:
      - backend

  backend:
    build: ./backend
    ports:
      - "4000:4000"
    environment:
      - NODE_ENV=production
      - VLM_SERVICE_URL=http://vlm-service:5000
      - REDIS_URL=redis://redis:6379 # optional
    depends_on:
      - vlm-service

  vlm-service:
    build: ./vlm-service
    ports:
      - "5000:5000"
    environment:
      - OPENROUTER_API_KEY=${OPENROUTER_API_KEY}
      - GEMINI_API_KEY=${GEMINI_API_KEY}
      - LOG_LEVEL=INFO

  redis: # optional, for session store
    image: redis:7-alpine
    ports:
      - "6379:6379"
```

Containerization Strategy

Service	Base Image	Key Dependencies	Notes
Frontend	`node:20-alpine`	React, Vite, TypeScript	Static build served by nginx in prod
Backend	`node:20-alpine`	Express, skia-canvas, seedrandom	Native Rust deps for skia-canvas
VLM Service	`python:3.11-slim`	FastAPI, google-generativeai, requests	Keep image <500MB

Security Considerations

API Key Management

- **Never commit keys**: Use `.env` files (gitignored) or secret management (AWS Secrets Manager, Vault)
- **Environment variables**: `OPENROUTER_API_KEY`, `GEMINI_API_KEY` injected at runtime
- **VLM service isolation**: Keys only accessible to Python container, not exposed to frontend

Rate Limiting

- **Backend → VLM service**: Queue requests to stay under provider limits (Gemini 60 RPM, OpenRouter varies)
- **Frontend → Backend**: Express rate-limit middleware (e.g., 100 requests/minute per IP)
- **Benchmarking**: Throttle to avoid quota exhaustion (max 10 parallel VLM calls)

CORS Configuration

```
// backend/src/api/middleware.ts
import cors from 'cors';

const corsOptions = {
  origin: process.env.FRONTEND_URL || 'http://localhost:3000',
  credentials: true,
  methods: ['GET', 'POST'],
  allowedHeaders: ['Content-Type', 'Authorization']
};

app.use(cors(corsOptions));
```

Input Validation

- **Backend**: Validate `difficulty`, `frames`, `taskType` enums
- **VLM service**: Pydantic models reject malformed requests
- **Integer extraction**: Regex `[-+]?\d+` prevents injection attacks

---

Implementation Roadmap

Phase 1: MVP Backend (Week 1-2)

Deliverables:

- [ ] TypeScript models: `Shape`, `GameObject`, `AnimationSequence`, `Task`
- [ ] Static tasks: `CountShape`, `CountColorShape`, `CountLeftHalf`, `CountInsideBox`, `CountOverlappingPairs`
- [ ] Dynamic tasks: `CountCrossMidline`, `CountEnterCentralBox`
- [ ] `ChallengeGenerator` with deterministic seeding (seedrandom)
- [ ] Server-side Canvas rendering (skia-canvas)
- [ ] Express routes: `/api/captcha/new`, `/api/captcha/answer`
- [ ] Oracle provider (ground-truth testing)
- [ ] Unit tests: task correctness, collision detection, bbox calculations

Success criteria:

- Generate 100 challenges in <20s
- Oracle mode: 100% accuracy on all task types
- Deterministic: same seed → identical frames

Phase 2: VLM Service Integration (Week 2-3)

Deliverables:

- [ ] Python FastAPI app structure
- [ ] Pydantic models: `PredictRequest`, `PredictResponse`
- [ ] OpenRouter provider with retry logic
- [ ] Gemini provider with retry logic
- [ ] Stub provider (weighted random for testing)
- [ ] Multi-provider fallback chain in `VLMEvaluator`
- [ ] Backend calls VLM service via HTTP POST
- [ ] `/api/captcha/benchmark` endpoint
- [ ] Integration tests: mock VLM responses

Success criteria:

- Benchmark 20 challenges in <60s (including VLM latency)
- Fallback works: if OpenRouter fails, tries Gemini, then Stub
- Robust parsing: extracts integers from "The answer is 5" and "5 shapes"

Phase 3: Frontend UI (Week 3-4)

Deliverables:

- [ ] React app with Vite build
- [ ] `CaptchaCanvas` component (Canvas animation player)
- [ ] `ChallengeView` component (question + answer input)
- [ ] `useCaptcha` hook (fetch from backend)
- [ ] Submit answer and display validation result
- [ ] Optional: Client-side demo mode (render locally with shared TS logic)
- [ ] Tailwind CSS styling
- [ ] Responsive design (mobile-friendly)

Success criteria:

- Smooth 12 FPS animation playback
- <500ms latency from answer submit to validation response
- Works on Chrome, Firefox, Safari (desktop + mobile)

Phase 4: Deployment & Benchmarking (Week 4-5)

Deliverables:

- [ ] Docker Compose setup (frontend, backend, vlm-service, redis)
- [ ] Environment variable configuration (`.env.example`)
- [ ] CI/CD pipeline (GitHub Actions or GitLab CI)
- [ ] Monitoring: Prometheus + Grafana for latency/accuracy metrics
- [ ] Benchmarking script: run 1000 challenges, export CSV
- [ ] Documentation: `README.md` , `API.md` , `CONTRIBUTING.md`

Success criteria:

- Single-command startup: `docker-compose up`
- Benchmark 1000 challenges in <2 hours
- Accuracy reports: static >80%, dynamic >60% (Qwen-VL baseline)

Phase 5: Advanced Features (Optional, Week 6+)

- [ ] WebSocket support for real-time eval feedback
- [ ] User accounts + leaderboard (PostgreSQL)
- [ ] Custom task creation UI (admin panel)
- [ ] GPU-based local inference (LLaVA, Qwen-VL via Ollama)
- [ ] A/B testing framework (compare model versions)
- [ ] Export challenges to dataset format (JSON, HuggingFace)

Trade-offs & Alternatives

Why NOT C#/Java for Backend Logic?

Consideration	TypeScript	C#/Java	Decision
Type safety	Strong (with strict mode)	Stronger (compile-time)	TS sufficient for this domain
Shared types FE/BE	Native	Requires codegen/duplication	TS wins
Deployment	Single runtime (Node)	JVM/.NET CoreCLR	TS simpler
Performance	Fast enough (<200ms render)	Slightly faster	VLM latency dominates anyway
Ecosystem	Mature (npm, Canvas)	Mature (NuGet, System.Drawing)	Both good, TS more FE-friendly

Consideration	TypeScript	C#/Java	Decision
Team skillset	JS/TS common	C#/Java enterprise-focused	TS easier to hire for

Verdict: TypeScript backend is **optimal** unless you need enterprise Java infrastructure (Spring Boot, Kubernetes Operators) or Windows-only hosting.

Why NOT Monolithic Python App?

Consideration	Monolith (Python)	Microservices (TS + Python)	Decision
Simplicity	Single codebase	Two codebases	Monolith simpler
Type safety	Weak (mypy optional)	Strong (TS strict mode)	Microservices safer
VLM SDK support	Native	Isolated in Python service	Both work
Scaling	Scale entire app	Scale VLM service independently	Microservices flexible
Deployment	One container	Three containers (FE/BE/VLM)	Monolith easier
Team separation	Backend + ML in one team	Backend team + ML team	Microservices better for large teams

Verdict: **Microservices** chosen for **flexibility** (isolate VLM changes) and **type safety** (TS OOP for shapes/tasks). For solo dev or MVP, monolithic Python would also work.

Why Base64 in JSON, Not Multipart/Form-Data?

Consideration	Base64 JSON	Multipart	Decision
Simplicity	Single JSON payload	Complex boundary parsing	Base64 simpler
Size	+33% overhead (6 frames × 15KB → 120KB)	90KB raw	Base64 acceptable for MVP
Browser support	Native	Requires FormData API	Both work
Debugging	Easy (copy/paste JSON)	Harder (binary inspection)	Base64 easier
Performance	Negligible for <1MB	Slightly better	VLM latency >> encoding time

Verdict: **Base64 JSON** for MVP; switch to multipart if payload size becomes issue (e.g., 1080p frames).

Conclusion

This architecture provides a **clean separation of concerns**:

- **Frontend (React/TS)**: User interaction and display
- **Backend (Node/TS)**: Challenge generation, orchestration, validation
- **VLM Service (Python)**: Model inference with provider abstraction

Key benefits:

1. **Single language for application logic** (TypeScript) reduces duplication and simplifies debugging
2. **Isolated VLM layer** allows model provider changes without backend redeploy
3. **Deterministic ground truth** via server-side rendering prevents cheating
4. **Scalable architecture** supports independent scaling of VLM inference
5. **Clear API contracts** enable parallel frontend/backend development

Next steps:

- Review and approve architecture
- Set up monorepo structure (`/frontend` , `/backend` , `/vlm-service` , `/shared`)
- Begin Phase 1: Backend MVP implementation

Appendix: Folder Structure

```
/dynamic-vlm-captcha
├── frontend/                    # React + TypeScript + Vite
│   ├── src/
│   │   ├── components/
│   │   ├── hooks/
│   │   ├── services/
│   │   ├── types/              # Re-exports from @shared/types
│   │   └── utils/
│   ├── public/
│   ├── package.json
│   └── vite.config.ts
├── backend/                     # Node.js + TypeScript + Express
│   ├── src/
│   │   ├── models/
│   │   ├── tasks/
│   │   ├── services/
│   │   ├── providers/
│   │   ├── api/
│   │   ├── utils/
│   │   └── types/              # Re-exports from @shared/types
│   ├── tests/
│   ├── package.json
│   └── tsconfig.json
├── vlm-service/                 # Python + FastAPI
│   ├── app/
│   │   ├── main.py
│   │   ├── models.py
│   │   ├── providers/
│   │   └── utils.py
│   ├── tests/
│   ├── requirements.txt
│   └── Dockerfile
├── shared/                      # Shared TypeScript types
│   ├── types/
│   │   └── index.ts
│   ├── package.json
│   └── tsconfig.json
├── docs/
│   ├── ARCHITECTURE.md         # This document
│   ├── API.md                  # Detailed OpenAPI spec
│   └── CONTRIBUTING.md
├── docker-compose.yml
├── .env.example
├── .gitignore
└── README.md
```

Q&A 1 - Deployment

- Docker container?
  - Yes, you can simply setup a Dockerfile for both services (VLM in Python and everything else in TS/Api)
- Website to be reached?
  - GitHub Pages (con: make repo public)

Does it make sense to put **everything** on Vercel?

Short answer:
✅ **Frontend + backend API in Node** → YES, Vercel is perfect
❌ **Python VLM service** → NO, Vercel does not support Python server runtimes

So a *full stack* on Vercel is only possible **if you drop Python completely** and move all VLM calls into TypeScript (direct calls to OpenRouter / Gemini APIs).
If you **keep Python**, then deployment must be **split**.

If we keep the Python VLM service — where should it live?

You need a host that supports:

Requirement	Why	GitHub Pages	Vercel	Render	Railway	Fly.io	EC2
Python runtime	Needed for FastAPI + model clients	❌	❌	✅	✅	✅	✅
Long-running service	VLM calls may take 3–10s	❌	⚠️ timeout	✅	✅	✅	✅
Ability to call OpenRouter / Gemini APIs	External HTTP	n/a	✅	✅	✅	✅	✅
Optional GPU / offline models	future-proofing	❌	❌	❌	❌	✅	✅

Best options right now

Host	Why it fits
Render.com (free web service)	Dead simple, supports Python, auto HTTPS, no timeout issues
Railway.app	Also easy, runs Docker, free tier (limits apply)
Fly.io	Best if you *later* want GPU or run local models
AWS EC2 / Lightsail	Full control, but more ops work

For a start, **Render is the smoothest**:

✅ Deploy Python FastAPI with 1 click
✅ Free tier okay for low traffic
✅ Can scale if needed
✅ Works well with Vercel frontend/backend

What the deployment split looks like

```
[Vercel] Next.js frontend  ──────────────┐
                                         │
[Vercel] Node API (captcha logic)  ─────→│──→  [Render] Python VLM service
                                         │
                                         └──→  OpenRouter / Gemini / GPT-4V
```

So the flow for a real user or benchmark is:

1. Browser requests `/captcha/new` → Vercel API responds with frames + question
2. Browser or backend sends frames to Python VLM service
3. Python service calls Gemini/OpenRouter, parses integer, returns result
4. Vercel API checks ground-truth and returns correctness

Clean separation of concerns.

When does it make sense to move **everything** into Vercel?

Only if:

✅ You drop Python entirely
✅ VLM API calls are rewritten in TypeScript
✅ You only ever use remote models (OpenRouter, Gemini, OpenAI)
✅ You do NOT plan to run local models (e.g., LLaVA, Qwen-VL, etc.)

Then you get single-host deployment:

```
/frontend  (Next.js + Canvas)
 /api/vlm  (Vercel serverless, TS, calls OpenRouter API)
 /api/captcha (TS)
```

That's technically nice, but note:

Trade-off	Impact
No Python = no local inference	You can't benchmark llama-vision etc.
Debugging VLM logic in TS is worse	Python has better tooling/logging libs
Real-time frame preprocessing (numpy/PIL) harder in TS	Requires sharp or wasm tooling

So: **if research + future local models matter → keep Python**.
If you want a **clean SaaS product** → full TS on Vercel is fine.

Recommended choice based on your goals

Your Goal	Best Deployment
Showcase project online	Vercel + Render (split)
Build long-term research tool	Render/Fly for Python + Vercel for UI
Enterprise-grade internal tool	AWS ECS or Azure App Service
Fully serverless, no Python ever	100% Vercel with TS-only VLM calls

---