ParallaxShift

Self-Hosted Architecture Guide

FastAPI + Next.js + Langflow + PostgreSQL

Executive Summary

This architecture separates concerns between a Python backend (FastAPI) for all business logic and a TypeScript frontend (Next.js) for UI. Langflow handles LLM orchestration. This approach keeps core logic in Python while enabling professional-grade UI customization.

Design Principles

- **Python for logic** — All business rules, auth, database queries, and Langflow integration in FastAPI
- **TypeScript for UI** — Next.js handles rendering only; no business logic in frontend
- **Langflow for prompts** — Visual prompt engineering, mode-specific flows, LLM API calls
- **Clear API contract** — FastAPI's OpenAPI spec serves as the interface between frontend and backend

Architecture Overview

```
┌─────────────────────────────────────────────────┐

│  NEXT.JS (UI Layer — TypeScript)                │

│  • React components, Tailwind styling           │

│  • Chat interface, mode selector, settings      │

│  • Consumes FastAPI endpoints — no business logic │

└─────────────────────────────────────────────────┘

                        ↓

┌─────────────────────────────────────────────────┐

│  FASTAPI (Backend — Python)                     │

│  • Auth, JWT sessions, user management           │

│  • Usage tracking, rate limits, tier enforcement │
│  • PostgreSQL queries via SQLAlchemy               │
│  • Calls Langflow, streams responses to frontend    │
└─────────────────────────────────────────────────────┘
                         ↓
┌─────────────────────────────────────────────────────┐
│  LANGFLOW (Orchestration Layer)                     │
│  • 7 mode-specific flows with system prompts        │
│  • Prompt chaining, RAG integration (future)        │
│  • REST API endpoints for FastAPI calls             │
└─────────────────────────────────────────────────────┘
                         ↓
┌─────────────────────────────────────────────────────┐
│  LLM PROVIDERS                                      │
│  • Claude API (Anthropic) — primary                 │
│  • Gemini Pro (Google) — fallback                   │
└─────────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────────┐
│  POSTGRESQL (Persistence Layer)                     │
│  • Users, conversations, messages, documents        │
│  • Accessed exclusively by FastAPI                  │
└─────────────────────────────────────────────────────┘
```

Logic Distribution

Logic Type	Layer	Details
UI Rendering	Next.js	Chat bubbles, mode selector, forms, settings, streaming display
Authentication	FastAPI	JWT generation/validation, password hashing, session management
Authorization	FastAPI	Tier checks, rate limits, feature flags
Data Access	FastAPI	SQLAlchemy models, all PostgreSQL queries
Usage Tracking	FastAPI	Token counting, cost calculation, billing integration
Prompt Logic	Langflow	System prompts, mode switching, prompt chains
LLM Calls	Langflow	Claude/Gemini API requests, response streaming

Technology Stack

Layer	Technology	Purpose
Frontend	Next.js 14 + React +	Server components, streaming UI, responsive design
Backend	FastAPI + Pydantic	Async API, auto-generated OpenAPI docs, validation
ORM	SQLAlchemy 2.0	Async database access, migrations via Alembic
Auth	python-jose + passlib	JWT tokens, bcrypt password hashing
Orchestration	Langflow	Visual LLM flows, prompt management
Database	PostgreSQL 16	Relational data, JSONB for flexible metadata
Hosting	Hetzner + Coolify	VPS + Docker orchestration, SSL, reverse proxy
LLM	Claude API / Gemini API	Direct API via Langflow — you absorb costs

API Contract

FastAPI endpoints consumed by Next.js. Auto-documented via OpenAPI/Swagger.

Authentication Endpoints

```
POST /api/auth/register      → Create user, return JWT

POST /api/auth/login         → Validate credentials, return JWT

POST /api/auth/refresh       → Refresh expired JWT

GET  /api/auth/me            → Get current user profile
```

Chat Endpoints

```
GET  /api/conversations               → List user's conversations
```

```
POST /api/conversations                → Create new conversation
GET  /api/conversations/{id}           → Get conversation with messages
POST /api/conversations/{id}/chat      → Send message, stream response
PATCH /api/conversations/{id}/mode     → Change conversation mode
```

Usage & Billing Endpoints

```
GET  /api/usage                   → Token usage stats for current period
GET  /api/billing/subscription    → Current subscription status
POST /api/billing/webhook         → Stripe webhook handler
```

Streaming Implementation

FastAPI (Python)

```
@app.post("/api/conversations/{conv_id}/chat")
async def chat(conv_id: UUID, request: ChatRequest, user: User = 
Depends(get_current_user)):
    # 1. Check rate limits
    if not await check_rate_limit(user):
        raise HTTPException(429, "Rate limit exceeded")

    # 2. Get conversation, verify ownership
    conv = await get_conversation(conv_id, user.id)

    # 3. Save user message to DB
    await save_message(conv_id, "user", request.content)

    # 4. Stream from Langflow
    async def generate():
        full_response = ""
        async for chunk in langflow_stream(request.content, conv.current_mode):
            full_response += chunk
            yield f"data: {json.dumps({'chunk': chunk})}\n\n"
        # 5. Save assistant response, update usage
        await save_message(conv_id, "assistant", full_response)
        await update_usage(user.id, count_tokens(full_response))

    return StreamingResponse(generate(), media_type="text/event-stream")
```

Next.js (TypeScript)

```
// app/chat/[id]/page.tsx
async function sendMessage(content: string) {
  const response = await fetch(`/api/conversations/${convId}/chat`, {
    method: 'POST',
    headers: { 'Authorization': `Bearer ${token}` },
    body: JSON.stringify({ content })
  });

  const reader = response.body?.getReader();
  const decoder = new TextDecoder();

  while (true) {
    const { done, value } = await reader.read();
    if (done) break;
    const chunk = JSON.parse(decoder.decode(value).replace('data: ', ''));
    setStreamingMessage(prev => prev + chunk.chunk);
  }
}
```

Database Schema

```sql
-- Users

CREATE TABLE users (

  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

  email VARCHAR(255) UNIQUE NOT NULL,

  password_hash VARCHAR(255) NOT NULL,

  tier VARCHAR(20) DEFAULT 'free',  -- free/pro/enterprise

  stripe_customer_id VARCHAR(255),

  tokens_used_this_period INTEGER DEFAULT 0,

  period_reset_at TIMESTAMP,

  created_at TIMESTAMP DEFAULT NOW()

);



-- Conversations

CREATE TABLE conversations (

  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

  user_id UUID REFERENCES users(id) ON DELETE CASCADE,

  title VARCHAR(255),

  current_mode VARCHAR(20) DEFAULT 'balanced',

  created_at TIMESTAMP DEFAULT NOW(),

  updated_at TIMESTAMP DEFAULT NOW()
);



-- Messages

CREATE TABLE messages (

  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

  conversation_id UUID REFERENCES conversations(id) ON DELETE CASCADE,

  role VARCHAR(20) NOT NULL,  -- user/assistant/system

  content TEXT NOT NULL,

  mode_used VARCHAR(20),

  tokens_used INTEGER,

  created_at TIMESTAMP DEFAULT NOW()

);



-- Indexes

CREATE INDEX idx_conversations_user_id ON conversations(user_id);

CREATE INDEX idx_messages_conversation_id ON messages(conversation_id);
```

Cost Estimate

Component	Monthly Cost	Notes
Hetzner CPX21	€7.55 (~$8)	3 vCPU, 4GB RAM
Coolify	$0	Self-hosted
Claude API (Sonnet 4.5)	$15–100+	$3/M in, $15/M out
Domain + SSL	~$1/mo	Let's Encrypt free
Estimated Total	**$25–120/mo**	API is the variable

Implementation Roadmap

Time estimates assume AI-assisted development (Claude Opus 4.5 or GitHub Copilot). Multiply by 2-3x for unassisted development.

Phase 1: Infrastructure (3-4 hours)

1. Provision Hetzner CPX21 VPS (Ubuntu 22.04)
2. Install Coolify, configure DNS
3. Deploy PostgreSQL via Coolify
4. Deploy Langflow via Coolify

Phase 2: FastAPI Backend (8-12 hours)

1. Project setup: FastAPI, SQLAlchemy, Alembic
2. Database models and migrations
3. Auth endpoints (register, login, JWT)
4. Conversation/message CRUD endpoints
5. Langflow integration with streaming
6. Usage tracking middleware

Phase 3: Langflow Setup (3-4 hours)

1. Create base flow with Claude API
2. Build 7 mode-specific flow variants
3. Configure API endpoints per mode
4. Test streaming responses

Phase 4: Next.js Frontend (12-16 hours)

1. Project setup: Next.js 14, Tailwind, shadcn/ui
2. Auth screens (login, register, password reset)
3. Chat interface with streaming display
4. Mode selector component
5. Conversation list/history sidebar
6. Settings and usage dashboard

Phase 5: Monetization & Polish (4-6 hours)

1. Stripe integration skeleton
2. Tier-based rate limiting
3. Usage alerts and upgrade prompts
4. Deploy frontend and backend via Coolify

Total Estimated Effort: 30-42 hours (AI-assisted) / 60-120 hours (unassisted)

Migration Guide for AI Assistants

Context for Claude Opus 4.5 or GitHub Copilot to migrate from Poe Canvas App.

Source: Poe Canvas App (index.html)

- Embedded system prompts for 7 modes
- HTML/CSS/JS in single file
- Poe.sendUserMessage() API
- No persistence, no auth

Migration Steps

1. **Extract system prompts** → Create Langflow flows for each mode
2. **Create FastAPI backend** → Auth, DB models, Langflow client, streaming endpoints
3. **Build Next.js frontend** → Convert HTML/JS to React components
4. **Replace Poe API** → Call FastAPI endpoints instead
5. **Add persistence** → SQLAlchemy models, Alembic migrations

File Structure Target

```
parallaxshift/

├── backend/                    # FastAPI

│   ├── app/

│   │   ├── main.py             # FastAPI app

│   │   ├── auth/               # JWT, password hashing

│   │   ├── models/             # SQLAlchemy models

│   │   ├── routers/            # API endpoints

│   │   ├── services/           # Langflow client, usage tracking

│   │   └── schemas/            # Pydantic models

│   ├── alembic/                # DB migrations

│   └── requirements.txt

├── frontend/                   # Next.js

│   ├── app/

│   │   ├── (auth)/             # Login, register pages

│   │   ├── chat/[id]/          # Chat interface

│   │   └── settings/           # User settings
│   ├── components/          # React components
│   └── package.json
└── docker-compose.yml       # Local dev
```

Critical Analysis

Strengths

1. **Clean separation:** Python devs work on backend, TypeScript devs on frontend. Clear API contract.
2. **Python-centric logic:** All business rules in one language. Easier to test, debug, and maintain.
3. **UI flexibility:** Next.js + Tailwind enables any design. No "Streamlit look."
4. **Auto-generated API docs:** FastAPI's OpenAPI spec documents the contract automatically.
5. **Production-ready auth:** JWT + bcrypt is battle-tested. Easy to add OAuth later.

Weaknesses

1. **Two codebases:** Must maintain Python and TypeScript projects. More moving parts.
2. **Network latency:** Every UI action requires API call. Adds ~50-100ms per request.
3. **Deployment complexity:** Three services (FastAPI, Next.js, Langflow) vs one monolith.
4. **Langflow as dependency:** Adds complexity. Consider eliminating if prompt iteration slows.

Alternatives to Consider

1. **Skip Langflow:** Store prompts in DB or config. Simpler but loses visual editing.
2. **Use Supabase:** Replace PostgreSQL + auth code with Supabase. Adds $25/mo, reduces code.
3. **Monorepo:** Put backend and frontend in single repo with shared types. Easier deploys.

Recommendation

This architecture balances your goals well: Python for logic, TypeScript for UI polish, and Langflow for prompt iteration. The main trade-off is complexity — three services instead of one.

Proceed if: You want professional UI, expect to iterate heavily on prompts, and are comfortable managing multiple services.

Simplify if: Prompt iteration slows down (drop Langflow) or you want faster initial deployment (use Streamlit first, migrate later).

Document generated: January 1, 2025