ChatGPT

Building a Modular Super-Agent Platform With Interchangeable Components

Executive summary

A “lego-style” super-agent platform is best treated as two systems: a **control plane** (where users assemble agents from parts) and a **runtime/data plane** (where those assembled agents execute requests reliably, securely, and with low latency). The design goal is **interchangeability**: “brains” (LLMs), voice, avatar/video, integrations (calendar, GitHub), and code execution should be swappable without rewriting orchestration logic—accomplished by strict contracts, capability discovery, and versioned components. [1]

For 2026-era OpenAI integrations, build on the **Responses API** (tool calling + structured outputs + built-in tools), because OpenAI has announced the Assistants API deprecation with a sunset date of **August 26, 2026**. [2]

For real-time experiences, there are two broad modalities: - **Text-first orchestration + external media** (LLM → TTS → avatar/video), which pairs well with ElevenLabs streaming output and avatar vendors that accept an audio track and return a rendered video (often asynchronously, via polling or webhooks). [3]
- **Unified realtime speech sessions** using OpenAI’s Realtime API for low-latency multimodal interaction (audio/text in and audio/text out) and/or to simplify STT/TTS loops. [4]

Key architectural recommendations: - Start with a **modular monolith + plugin system** (fastest path for a solo/beginner Python build), and only extract microservices once you have clear scaling/ownership boundaries. Use Python packaging entry points (or a plugin manager) for component discovery and versioning. [5]
- Use an **event bus / work queue** for long-running work (avatar rendering, repository indexing, code execution jobs) so the interactive path stays under tight latency budgets. [6]
- Treat **auth as a first-class subsystem**: API keys for model/media vendors; OAuth 2.0 (with PKCE) for calendars and user-delegated integrations; refresh/revocation support; encrypted token storage; and scoped permissions. [7]
- Implement **state/memory** with clear tiers: short-term conversational state, episodic summaries, and long-term retrieval (vector DB) with explicit data retention and user controls. [8]
- Ship with **observability by default** (tracing + metrics + logs) and define metrics around latency, tool reliability, rate limits, and cost. [9]

Core principles for a modular super-agent

A rigorous modular design requires **contracts, not conventions**. The most common failure mode in “agent platforms” is that orchestration logic becomes tightly coupled to one LLM vendor, one tool schema format, or one media pipeline. The antidote is to define “component contracts” that are: - **Capability-based** (what a component can do) rather than name-based (which vendor it is). - **Versioned** (interface v1, v2…) and validated (schemas/tests). - **Observable** (every component call emits spans/metrics with correlation IDs). [10]

A practical decomposition that maps to your requirements:

- **Brain**: "generate" responses + optional tool calls (LLM via API). Tool calling is explicitly a multi-step loop: model suggests tool call → app executes tool → app returns tool output → model continues. [11]
- **Voice**: TTS (and optionally STT). ElevenLabs supports real-time streaming audio over HTTP for select endpoints and WebSockets for partial text streaming, including word alignment use cases. [12]
- **Avatar/Video**: "render" a video from an image/avatar + audio; typically asynchronous (job creation + polling/webhook). Vendors like D-ID [13] document video creation endpoints and webhook options. [14]
- **Integrations**: calendar/appointments and GitHub operations. Calendar APIs expose event creation endpoints and require OAuth. [15]
- **Code execution**: safe Python execution via OpenAI Code Interpreter tool (sandboxed) or your own sandbox. [16]
- **Memory/state**: conversation snapshots + vector retrieval. OpenAI's file search tool uses "vector stores" conceptually similar to external vector DBs. [17]

One 2025–2026 pattern that fits "lego-style" composition is **MCP (Model Context Protocol)**: a standardized way to expose tools/context from external servers, plus OpenAI "connectors" as MCP wrappers for popular services. This reduces bespoke integration work when you adopt MCP-compatible ecosystems. [18]

Architecture patterns and recommended option set

The core tradeoff is **speed of iteration vs operational complexity**. The following three options are the most defensible "platform shapes" for your use case:

Option	What it is	Strengths	Tradeoffs	When it's the right choice
Modular monolith (plugin-first)	One Python app (API + orchestration) with a plugin registry for brains/voice/avatar/integrations	Fastest to build and refactor; easiest local debugging; plugins via Python entry points are a mature pattern	Scaling is coarser; one deploy unit; needs discipline to avoid "spaghetti imports"	You are newer to coding and want an MVP quickly; early product discovery
Microservices (adapter services + orchestration core)	Separate services: Orchestrator, Brain Gateway, Media Gateway, Integrations, Memory	Independent scaling and isolation; teams can own services; clearer SLAs per subsystem	Higher complexity: networking, distributed tracing, service contracts	You expect high load, multiple developers, or strict isolation (e.g., code execution)

Option	What it is	Strengths	Tradeoffs	When it's the right choice
Event-driven orchestration (bus + workers)	Interactive requests stay thin; long work is jobs on a message bus; state transitions are evented	Smooth handling of slow tasks (video rendering); resilient retries; backpressure	Requires careful idempotency and state machines; more moving parts	Your platform will do lots of asynchronous actions (video, GitHub indexing, batch analysis)

Plugin systems in Python are well supported by **entry points** (package metadata advertising components) plus tooling that helps load/manage plugins. [19]

For the event bus layer, choose based on your needed semantics: - Kafka topics/partitions and consumer groups are designed for high-throughput log/event workloads. [20]
- RabbitMQ exchanges/queues/bindings are a standard model for message routing patterns in AMQP. [21]
- Redis Streams provide persistent, consumer-group based stream consumption via commands like XREADGROUP. [22]
- NATS implements pub/sub and request-reply patterns (conceptually useful for "tool calls" between services). [23]

A pragmatic starter choice for a beginner Python build: **Redis Streams or RabbitMQ** for job queues (simple ops), then Kafka only when you have proven throughput needs. [24]

Component interfaces and data flows

Interchangeable component contracts

Define internal contracts as *vendor-neutral* interfaces. A minimal contract set:

- **BrainProvider**
- `generate(input, state, tools) -> (assistant_output, tool_calls, usage)`
- **ToolProvider**
- `invoke(tool_name, args, auth_context) -> tool_result`
- **VoiceProvider**
- `tts(text, voice_config) -> audio_stream|audio_asset`
- `stt(audio_stream) -> transcript` (optional)
- **AvatarProvider**
- `render(audio_asset, avatar_config) -> video_job_handle`
- `get_status(job_handle) -> (status, video_asset?)`
- **MemoryStore**
- `append_turn(conversation_id, turn)`
- `summarize_if_needed(conversation_id)`
- `retrieve(query, filters) -> passages`
- **CodeExecutionProvider**
- `run_python(code, files, limits) -> outputs` (sandboxed)

OpenAI's tool calling flow is explicitly multi-step (request with tools → tool call → execute → return tool output → continue), so your BrainProvider should yield structured tool calls, not "hidden" side effects. [11]

Minimal viable architecture diagram

```
flowchart TB
  subgraph Client
    UI["Web/Mobile UI\n(chat + voice + avatar)"]
  end

  subgraph ControlPlane["Control plane"]
    Builder["Agent Builder UI\n(Lego assembly)"]
    Registry["Component Registry\n(versions + capabilities)"]
    BlueprintDB["Blueprint Store\n(agent configs)"]
  end

  subgraph Runtime["Runtime / data plane"]
    APIGW["API Gateway / FastAPI"]
    Orchestrator["Orchestrator\n(state machine + routing)"]
    Queue["Job Queue / Event Bus"]
    Workers["Workers\n(video, indexing, code jobs)"]
    Memory["Memory Layer\n(short + long term)"]
  end

  subgraph Providers["External providers"]
    LLMs["Brains (LLMs via API)"]
    TTS["Voice (ElevenLabs)"]
    Avatar["Avatar/Video API\n(VidID-like)"]
    Cal["Calendar APIs"]
    GH["GitHub APIs"]
  end

  subgraph Stores
    SQL["Relational DB\n(users, conversations)"]
    VDB["Vector DB\n(embeddings)"]
    OBJ["Object storage\n(audio/video/files)"]
  end

  UI --> APIGW
  Builder --> Registry --> BlueprintDB
  APIGW --> Orchestrator --> Memory
  Orchestrator --> LLMs
  Orchestrator --> TTS
  Orchestrator --> Queue --> Workers --> Avatar
  Orchestrator --> Cal
  Orchestrator --> GH
  Memory --> SQL
  Memory --> VDB
  Workers --> OBJ
```

This architecture is aligned with OpenAI's recommended direction (Responses API + tools) and supports asynchronous work for media generation. [25]

Voice + avatar turn flow

For a voice/avatar agent, you want a pipeline that supports: - low-latency partial text/audio streaming (for responsiveness), and - asynchronous video rendering (because avatar generation is often slower than TTS). [26]

```
sequenceDiagram
  autonumber
  participant U as User
  participant UI as Client UI
  participant API as Orchestrator API
  participant B as Brain (LLM)
  participant T as TTS (ElevenLabs)
  participant V as Avatar/Video API
  participant Q as Queue/Worker

  U->>UI: Speak / type
  UI->>API: input (audio or text)
  API->>B: prompt + state + tool schemas
  B-->>API: text response + optional tool calls
  API->>T: stream speech from text
  T-->>UI: audio stream (play immediately)
  API->>Q: enqueue avatar render job (audio asset + avatar config)
  Q->>V: create video
  V-->>Q: job_id / pending
  Q-->>API: job complete (video URL/asset) via webhook/poll
  API-->>UI: video ready (sync lip + audio)
```

ElevenLabs supports streaming speech (HTTP streaming endpoint) and also has a WebSocket TTS API for partial input and alignment use cases. [27]
Avatar APIs commonly provide "create video" endpoints and webhooks; D-ID [13] documents "Create a Video" and webhook parameters. [28]

Orchestration of multiple brains

A "multi-brain" platform benefits from explicit router patterns:

- **Routing**: pick one brain based on task type, latency SLA, budget, or tool needs.
- **Fallback**: if the primary brain times out or hits rate limits, fail over.

- **Ensemble + judge**: generate N candidate answers and have a designated "judge" brain select/merge.
- **Specialized subagents**: delegate to specialist brains for coding, retrieval, or policy checks.

OpenAI publishes guidance distinguishing reasoning models and notes that tool-capable models can combine tools. [29]

```
flowchart LR
  In["User request"] --> Router["Brain Router\n(policy + cost + SLA)"]
  Router -->|simple/cheap| BrainA["Brain A\nfast model"]
  Router -->|hard reasoning| BrainB["Brain B\nreasoning model"]
  Router -->|coding| BrainC["Brain C\ncode-focused"]
  BrainA --> Out
  BrainB --> Out
  BrainC --> Out
  subgraph Ensemble
    BrainA --> Judge["Judge/Verifier brain"]
    BrainB --> Judge
    BrainC --> Judge
    Judge --> Out["Final answer + tool calls"]
  end
```

Important implementation detail: do **not** couple the router to vendor names; couple it to **capabilities** and measured performance (p50/p95 latency, tool success rate, cost). OpenAI rate limits are measured across dimensions like RPM/TPM and can be hit in different ways, so rate-limit aware routing is practical. [30]

Notes on "VidID-like API"

"VidID" is ambiguous in public references (in some contexts it refers to rights management rather than avatar generation). [31]
When the avatar/video API is unspecified, implement a **generic render-job pattern** (create → status → result/webhook) and keep adapter code isolated behind an AvatarProvider interface. [28]

Authentication, authorization, and integration security

Authentication patterns by provider type

API key providers (LLMs, TTS, many avatar APIs)
- OpenAI API: uses API keys via Bearer authentication; keys must be protected and not exposed client-side. [32]
- ElevenLabs API: uses an `xi-api-key` header; keys can be scoped and quota-limited. [33]
- Avatar/video vendors: commonly use API keys or Basic auth and support webhooks for async completion. [28]

OAuth 2.0 integrations (calendar, user-delegated GitHub access)
OAuth 2.0 is defined by IETF [34] standards (RFC 6749). PKCE (RFC 7636) mitigates authorization code

interception for public clients. Bearer token usage over TLS is defined in RFC 6750. Token revocation is defined in RFC 7009. [35]

Calendar integrations

For calendar creation: - Google [36] Calendar API exposes `events.insert` ("Creates an event") and publishes a Python Quickstart emphasizing OAuth client IDs for end-user access. [37]
- Microsoft [38] Graph exposes "Create event" endpoints and distinguishes delegated vs application permissions; refresh tokens are issued when `offline_access` is requested. [39]

Design choice: treat "calendar" as an integration plugin with a uniform interface (create event, update, free/busy), and let the auth layer produce a normalized `AuthContext` (provider, access token, refresh token, scopes/permissions, expiry).

GitHub integration

Prefer GitHub Apps over personal access tokens for production, because Apps support scoped permissions and token rotation patterns: - Installation access tokens for GitHub Apps are minted via REST calls and expire after one hour. [40]
- User access tokens for GitHub Apps can be configured to expire (GitHub docs note a typical 8-hour expiry and a refresh token lasting months). [41]
- GitHub documents multiple authentication options for the REST API (PATs, GitHub App tokens, workflow tokens). [42]

This matters for a "super agent" because actions like repo cloning, indexing, PR creation, and comment posting should be **explicitly authorized**, auditable, and revocable.

Security checklist for a super-agent platform

This is a practical checklist oriented to your component mix and threat model (prompt injection + API abuse + data leakage + tool misuse):

- **Secrets and tokens**
- Never ship vendor API keys to browsers; use a backend token exchange or signed/temporary URLs where supported (ElevenLabs documents signed URLs for client-side agent connections in their agent product context). [43]

- Encrypt refresh tokens at rest; rotate keys; maintain token revocation/reauthorization flows (RFC 7009). [44]

- **OAuth hardening**

- Use Authorization Code + PKCE for native/mobile/SPA flows (RFC 7636). [45]
- Use TLS everywhere for bearer tokens (RFC 6750). [46]

- Request only needed scopes/permissions; for Microsoft identity platform, request `offline_access` only when you truly need refresh tokens. [47]

- **API security controls (your own platform APIs)**

- Implement object-level authorization checks consistently (OWASP API Security Top 10 highlights Broken Object Level Authorization as a primary risk). [48]

- Add endpoint rate limiting and abuse monitoring; OpenAI's own rate limit dimensions (RPM/TPM/etc.) illustrate why multi-dimensional quotas matter. [49]

- **LLM security**

- Treat prompt injection and insecure tool usage as top risks; OWASP [50] LLM Top 10 includes Prompt Injection and Insecure Output Handling among core concerns. [51]

- Put **human-approval gates** or policy gates on high-impact tools (calendar booking, GitHub write actions, payments if ever added). OpenAI's connectors/MCP guide explicitly supports restricting tool calls via approval. [52]

- **Code execution**

- If you allow arbitrary code execution, isolate it. OpenAI's Code Interpreter is described as a sandboxed Python environment. If you build your own sandbox, enforce no-network by default, CPU/memory/time limits, and strict file system boundaries. [16]

Privacy and compliance considerations for voice/video + user data

Voice and video are high-sensitivity data (biometric-adjacent in many contexts). A defensible baseline:

- **Data minimization & purpose limitation**: GDPR's principles include lawfulness/fairness/transparency and "data minimization" / "purpose limitation" concepts in Article 5. [53]
- **User rights**: California's CCPA guidance lists rights including deletion requests and opt-out of sale/sharing (high level); implement deletion workflows and retention controls accordingly. [54]
- **LLM vendor data controls**: OpenAI documents that API data is not used to train models by default (absent opt-in), and provides "data controls" guidance. [55]

For AI risk governance, align internal controls to NIST [56] AI RMF guidance (risk management practices for AI systems). [57]

State, memory, and data stores

State and memory model

A modular agent platform typically needs four tiers of state:

1. **Request state (ephemeral)**
   Correlation IDs, per-request tool execution logs, streaming buffers.

2. **Session state (short-term memory)**
   The last N turns, plus a compacted summary when the context grows too large. OpenAI's platform materials emphasize "run and scale" topics like streaming and conversation state management. [58]

3. **Episodic memory (medium-term)**
   Summaries of completed tasks, user preferences, and outcomes, usually stored in relational DB and sometimes embedded for retrieval.

4. **Long-term retrieval memory**
   Vector search over documents, chats, repos, and "agent logs." OpenAI's file search tool uses vector stores conceptually and supports retrieval over uploaded files. [59]

Recommended data store mix

For your component mix (voice/video assets + agent configs + retrieval + integrations), a common and effective tri-store pattern is:

- **Relational DB** (core truth): users, agent blueprints, permissions, integration grants, audit logs.
- **Vector DB** (retrieval): embeddings + metadata filters.
- **Object storage** (blobs): audio, rendered video, large files, optional dataset artifacts.

A practical store decision table:

Need	Recommended store	Why	Notes / tradeoffs
Agent blueprints, users, jobs, audit logs	PostgreSQL	Strong consistency, relational modeling, mature ecosystem	Also supports vector search via pgvector extension
Retrieval embeddings + metadata filters	pgvector / Qdrant / Weaviate / Milvus	Designed for similarity search + filtering	Choose based on ops + scale + query patterns
Audio/video assets	S3/GCS/Azure Blob style object storage	Cheap storage, good for large media	Use signed URLs for controlled access

Evidence points from vector DB docs: - pgvector supports approximate indexes HNSW and IVFFlat, with explicit speed/recall/memory tradeoffs (HNSW usually better query performance but more memory and slower build). [60]
- Qdrant emphasizes combining vector indexes with payload indexes for effective filtering. [61]
- Weaviate documents hybrid search (vector + BM25) fusion. [62]
- Milvus documents a decoupled architecture with query/data/streaming nodes to scale search and ingestion. [63]

Cost and latency tradeoffs

A concise way to think about cost/latency is “interactive path” vs “batch path”:

- **LLM calls (brains)**:
  Latency depends on model size and whether you stream output. OpenAI supports SSE streaming for responses. [64]
  Cost typically scales with tokens; OpenAI model pages show per-token pricing and note tool-specific fees for some tools. [65]
  Operationally, you must design around rate limits (RPM/TPM/etc.). [49]

- **TTS (voice)**:
  Streaming TTS endpoints can deliver audio as it is generated (lower perceived latency), which is valuable for conversational experiences. [66]

- **Avatar/video rendering**:
  Typically slower and should be placed on an async job queue; expect polling/webhooks. [28]

- **Vector retrieval**:
  pgvector ANN indexes explicitly trade recall for speed; HNSW vs IVFFlat are meaningful tuning levers. [60]

- **Object storage**:
  Often the cheapest way to store audio/video; design your API to return references (URLs/keys), not raw blobs. ElevenLabs’ cookbook explicitly demonstrates uploading generated audio to S3 and sharing via signed URL (a common distribution pattern). [67]

Deployment, CI/CD, testing, and monitoring

Deployment options

A “super-agent” platform is typically deployed in one of these modes:

- **Containers (recommended for MVP → scale)**: Docker Compose locally; then a container platform (managed Kubernetes or managed container services) for production. This fits multi-service growth and worker pools.
- **Serverless**: good for webhook handlers and small integration endpoints, but streaming (WebSockets) and long-running tasks can complicate serverless constraints.
- **Hybrid**: orchestrator as a long-running service; workers scale separately; serverless for low-duty integration endpoints.

For realtime voice interactions, you’ll often need WebSockets (or WebRTC). OpenAI Realtime supports WebSockets and recommends WebRTC for browser/mobile clients; WebSocket auth uses an API key header, and ephemeral tokens are referenced for browser scenarios. [68]

CI/CD

A minimal CI/CD stack for a Python platform:

- GitHub Actions for lint/test/build, and container image pushes.
- Automated secret scanning and dependency checks.
- Optional: deployment triggers and migrations.

If you use GitHub Apps for automation, a GitHub-maintained action exists to create installation access tokens (useful inside CI/CD). [69]

Testing strategy

Design tests around your component contracts:

- **Contract tests (adapter compliance)**: ensure BrainProvider/VoiceProvider/AvatarProvider implement required behaviors and error semantics consistently.
- **Tool call loop tests**: validate that your orchestrator correctly handles model tool calls and re-invocation loops (OpenAI's tool calling flow is explicitly multi-step). [70]
- **OAuth token lifecycle tests**: simulate expiry + refresh for GitHub/Microsoft/Google flows; verify encrypted storage and revocation. [71]
- **Streaming tests**: SSE streaming (LLM) and audio streaming (TTS) backpressure, reconnection, partial results.
- **Security tests**: prompt injection regression cases; SSRF protections for any URL-fetching tools; authorization bypass tests mapped to OWASP API risks. [72]

Monitoring & metrics

Use a three-signal approach (logs/metrics/traces) with correlation IDs: - OpenTelemetry defines standard approaches for logs and metrics correlation, and positions itself as vendor-neutral observability. [73]
- Prometheus guidance recommends instrumenting broadly; it is a common baseline metrics backend. [74]
- Sentry provides distributed tracing and performance monitoring options for application debugging. [75]

Suggested "platform health" metrics:

Category	Metrics	Why it matters
Latency	p50/p95/p99 end-to-end; per-component (LLM, TTS, avatar, calendar, GitHub)	Keeps UX responsive; isolates bottlenecks
Reliability	success rate per provider; retry counts; timeout rates	Identifies flaky vendors/adapters
Rate limiting	OpenAI RPM/TPM related errors; vendor quota errors	Prevents cascading failures and cost spikes [49]
Queue health	queue depth; job age; worker utilization	Ensures async media jobs complete

Category	Metrics	Why it matters
Safety	tool call approvals denied; policy violations flagged	Tracks risk controls effectiveness
Cost	tokens in/out per brain; media credits consumed; retrieval volume	Enables budget-aware routing

Roadmaps and starter code

Learning roadmap for a Python beginner

This is a staged learning path where each stage yields a concrete artifact you can reuse in the platform.

Stage	Focus	Outcome artifact
Python foundations	types, dataclasses, exceptions, file I/O	reusable domain models for “AgentState”, “ToolCall”
HTTP + APIs	REST basics, JSON, auth headers	small client wrappers for OpenAI + ElevenLabs
Async Python	`asyncio`, async HTTP, streaming	SSE consumer + audio streaming playback buffer
Web frameworks	FastAPI routes + WebSockets	runtime API service for the orchestrator [76]
OAuth 2.0	authorization code + PKCE, refresh tokens	working Google/Microsoft calendar connector [77]
Data stores	Postgres basics + vector search	conversation store + retrieval layer [78]
Docker	containerize API + worker	reproducible dev environment
Git/GitHub	branching, PRs, Actions	CI running unit + contract tests
Testing	pytest, mocks, integration tests	stable adapter contracts and tool loops
Observability	OpenTelemetry + metrics	dashboards for latency + error rates [79]

Implementation milestones with estimated effort

Effort is given as ranges because the variance is mostly driven by prior OAuth experience and how polished your UI must be.

Milestone	Deliverable	Novice effort	Intermediate effort	Advanced effort
Skeleton platform	FastAPI app, config loading, basic plugin registry	1–2 weeks	3–5 days	1–2 days

Milestone	Deliverable	Novice effort	Intermediate effort	Advanced effort
Brain adapters	At least 2 “brains” behind one interface + router	1–2 weeks	4–7 days	2–4 days
Tool calling loop	Tool schema, execution loop, retries, audit	2–3 weeks	1–2 weeks	4–7 days [80]
Calendar integration	Google + Microsoft event creation	2–4 weeks	1–2 weeks	4–7 days [81]
GitHub integration	GitHub App auth + read/write ops	2–4 weeks	1–2 weeks	4–7 days [82]
Memory layer	Postgres + vector DB + retrieval policies	2–4 weeks	1–2 weeks	4–7 days [83]
Voice pipeline	ElevenLabs streaming TTS + optional STT	1–2 weeks	4–7 days	2–4 days [84]
Avatar pipeline	Generic render job adapter + worker queue	2–4 weeks	1–2 weeks	4–7 days [28]
Deployment + observability	Docker, CI/CD, OTel metrics/traces	2–4 weeks	1–2 weeks	4–7 days [85]

Milestone timeline Gantt

```
gantt
  title Modular Super-Agent Platform Roadmap
  dateFormat  YYYY-MM-DD
  axisFormat  %b %d
  section Foundation
  Skeleton platform              :a1, 2026-02-17, 10d
  Brain adapters + router        :a2, after a1, 12d
  Tool calling loop              :a3, after a2, 15d
  section Integrations
  Calendar integrations          :b1, after a3, 15d
  GitHub integration             :b2, after a3, 15d
  section Memory + Media
  Memory layer + vector store    :c1, after a3, 15d
  Voice pipeline (TTS/STT)       :c2, after c1, 10d
  Avatar/video pipeline          :c3, after c2, 15d
  section Productionization
  Deployment + CI/CD + Observability :d1, after c3, 15d
```

Recommended libraries, SDKs, and repos

This table focuses on "foundation pieces" that align with your requirements (Python-first, modularity, and agent/workflow support).

Category	Option	Strengths	Tradeoffs / fit notes
OpenAI SDK	`openai-python` repo	Official client; supports Responses API and streaming SSE patterns	You still need your own orchestration layer [86]
ElevenLabs SDK	`elevenlabs-python` repo	Official client; supports streaming TTS endpoints	Still need your own media pipeline + storage conventions [87]
Agent orchestration	LangGraph	Graph/stateful agent orchestration; "long-running, stateful agents" positioning	Adds a framework layer; evaluate complexity carefully [88]
Multi-agent framework	AutoGen / Microsoft Agent Framework	Event-driven multi-agent patterns; successor framework emphasizes state/telemetry	Larger abstraction surface; may be more than MVP needs [89]
Retrieval framework	LlamaIndex	Data framework for LLM apps; large integration ecosystem	You must still define memory governance policies [90]
Plugin system	Python entry points + Stevedore or Pluggy	Mature plugin discovery/management patterns	Requires packaging discipline and version management [91]
Observability	OpenTelemetry + Prometheus + Sentry	Standard signals; flexible backends; performance + debugging	Requires up-front instrumentation effort [92]
Vector DB	pgvector / Qdrant / Weaviate / Milvus	Strong retrieval options; tradeoffs by scale and ops	Choose based on hosting + filtering + hybrid needs [93]

Python snippets for core flows

The snippets below are intentionally "adapter-oriented": they show the core flow you'll reuse while swapping vendors.

Calling an LLM brain via OpenAI Responses API (streaming)

```
from openai import OpenAI
```

```
client = OpenAI()

stream = client.responses.create(
    model="gpt-4.1",
    input="Summarize the key design goals for a modular agent platform in 5 
bullets.",
    stream=True,
)

for event in stream:
    # In production: route events to your UI via SSE/WebSocket
    print(event)
```

86

Brain registry + “switching brains” routing pattern

```
from dataclasses import dataclass
from typing import Protocol, Any, Dict, Optional

class Brain(Protocol):
    name: str
    capabilities: set[str]  # e.g., {"tool_calling", "reasoning", "fast"}

    def generate(self, text: str, context: Dict[str, Any]) -> str: ...

@dataclass
class BrainConfig:
    preferred: str
    fallback: str

class BrainRouter:
    def __init__(self, brains: Dict[str, Brain], cfg: BrainConfig):
        self.brains = brains
        self.cfg = cfg

    def pick(self, task_hint: Optional[str] = None) -> Brain:
        # Example heuristic: route "code" to a code-focused brain if available.
        if task_hint == "code":
            for b in self.brains.values():
                if "code" in b.capabilities:
                    return b
        return self.brains.get(self.cfg.preferred, 
self.brains[self.cfg.fallback])

    def run(self, text: str, context: Dict[str, Any], task_hint: Optional[str]
= None) -> str:
        brain = self.pick(task_hint)
        try:
            return brain.generate(text, context)
        except Exception:
            # Basic fallback strategy; add retry budgets + error classification.
            return self.brains[self.cfg.fallback].generate(text, context)
```

This pattern is motivated by the operational realities of rate limits and reliability variance across providers/ models. [30]

ElevenLabs streaming TTS over HTTP

```
import os
import requests

ELEVEN_API_KEY = os.environ["ELEVENLABS_API_KEY"]
VOICE_ID = "YOUR_VOICE_ID"

url = f"https://api.elevenlabs.io/v1/text-to-speech/{VOICE_ID}/stream"
headers = {
    "xi-api-key": ELEVEN_API_KEY,
    "Content-Type": "application/json",
}
payload = {
    "text": "Hello. This audio should begin playing while it is still being
generated.",
    "model_id": "eleven_multilingual_v2",
    "output_format": "mp3_44100_128",
}

with requests.post(url, headers=headers, json=payload, stream=True, timeout=60)
as r:
    r.raise_for_status()
    with open("out.mp3", "wb") as f:
        for chunk in r.iter_content(chunk_size=8192):
            if chunk:
                f.write(chunk)
```

[94]

Avatar/video render job pattern (VidID-like), using D-ID as an example adapter

```
import os
import time
import requests
from requests.auth import HTTPBasicAuth

DID_API_KEY = os.environ["DID_API_KEY"]

create_url = "https://api.d-id.com/talks"
get_url_tpl = "https://api.d-id.com/talks/{talk_id}"

payload = {
    "source_url": "https://example.com/your-avatar-image.jpg",
    "script": {
        "type": "audio",
        "audio_url": "https://example.com/your-audio.mp3"
    },
    # Many providers support webhooks; if available, prefer webhook over
polling.
    # "webhook": "https://your-domain.com/webhooks/avatar"
}

resp = requests.post(create_url, auth=HTTPBasicAuth(DID_API_KEY, ""),
json=payload, timeout=30)
resp.raise_for_status()
talk_id = resp.json()["id"]

# Poll (MVP). Production: exponential backoff + webhook + idempotency keys.
for _ in range(60):
    status = requests.get(get_url_tpl.format(talk_id=talk_id),
auth=HTTPBasicAuth(DID_API_KEY, ""), timeout=30).json()
    if status.get("result_url"):
        print("Video:", status["result_url"])
        break
    time.sleep(2)
```

14

Google Calendar event creation (OAuth required)

```
from googleapiclient.discovery import build
from google.oauth2.credentials import Credentials

# You obtain + refresh these via Google's OAuth flow (store refresh tokens
securely).
creds = Credentials(token="ACCESS_TOKEN")

service = build("calendar", "v3", credentials=creds)
```

```
event = {
    "summary": "Demo: Agent-scheduled meeting",
    "start": {"dateTime": "2026-02-20T10:00:00-08:00"},
    "end": {"dateTime": "2026-02-20T10:30:00-08:00"},
}

created = service.events().insert(calendarId="primary", body=event).execute()
print(created["id"])
```

95

GitHub App installation token (server-to-server)

```
import os
import time
import jwt  # PyJWT
import requests

APP_ID = os.environ["GITHUB_APP_ID"]                  # integer-like
PRIVATE_KEY_PEM = os.environ["GITHUB_APP_PRIVATE_KEY_PEM"]
INSTALLATION_ID = os.environ["GITHUB_INSTALLATION_ID"]

now = int(time.time())
payload = {
    "iat": now - 60,
    "exp": now + 9 * 60,
    "iss": APP_ID,
}
app_jwt = jwt.encode(payload, PRIVATE_KEY_PEM, algorithm="RS256")

headers = {
    "Authorization": f"Bearer {app_jwt}",
    "Accept": "application/vnd.github+json",
}

token_url = f"https://api.github.com/app/installations/{INSTALLATION_ID}/
access_tokens"
r = requests.post(token_url, headers=headers, timeout=30)
r.raise_for_status()
installation_token = r.json()["token"]

print("Installation token acquired (expires ~1 hour).")
```

96

Q1:

If I wanted this “super agent” to support 50,000 users without my servers becoming a modern art exhibit titled *Flames*, which components should become microservices first—and why?

Q2:

What’s the most defensible “memory policy” for voice/video agents so users feel personalization, but auditors feel calm—like, *tea and spreadsheets* calm?

Q3:

How do I design multi-brain routing so it’s not just “pick the expensive model when I panic,” but a measurable, testable policy with real SLAs and budget caps?

---

1 Responses | OpenAI API Reference
https://platform.openai.com/docs/api-reference/responses?utm_source=chatgpt.com

2 Deprecations | OpenAI API
https://developers.openai.com/api/docs/deprecations/?utm_source=chatgpt.com

3 27 84 94 Stream speech
https://elevenlabs.io/docs/api-reference/text-to-speech/stream?utm_source=chatgpt.com

4 Realtime API
https://developers.openai.com/api/docs/guides/realtime/?utm_source=chatgpt.com

5 19 38 91 Entry points specification
https://packaging.python.org/specifications/entry-points/?utm_source=chatgpt.com

6 20 Introduction | Apache Kafka
https://kafka.apache.org/documentation/?utm_source=chatgpt.com

7 35 RFC 6749 - The OAuth 2.0 Authorization Framework
https://datatracker.ietf.org/doc/html/rfc6749?utm_source=chatgpt.com

8 13 17 36 59 File search | OpenAI API
https://developers.openai.com/api/docs/guides/tools-file-search/?utm_source=chatgpt.com

9 10 73 79 85 92 Documentation
https://opentelemetry.io/docs/?utm_source=chatgpt.com

11 50 70 80 Function calling | OpenAI API
https://developers.openai.com/api/docs/guides/function-calling/?utm_source=chatgpt.com

12 26 66 Streaming | ElevenLabs Documentation
https://elevenlabs.io/docs/api-reference/streaming?utm_source=chatgpt.com

14 28 Create a Video
https://docs.d-id.com/reference/createtalk?utm_source=chatgpt.com

15 37 81 Events: insert | Google Calendar
https://developers.google.com/workspace/calendar/api/v3/reference/events/insert?utm_source=chatgpt.com

[16] Code Interpreter | OpenAI API
https://developers.openai.com/api/docs/guides/tools-code-interpreter/?utm_source=chatgpt.com

[18] [52] Connectors and MCP servers | OpenAI API
https://developers.openai.com/api/docs/guides/tools-connectors-mcp/?utm_source=chatgpt.com

[21] Exchanges
https://www.rabbitmq.com/docs/exchanges?utm_source=chatgpt.com

[22] XREADGROUP | Docs
https://redis.io/docs/latest/commands/xreadgroup/?utm_source=chatgpt.com

[23] Publish-Subscribe - NATS Docs
https://docs.nats.io/nats-concepts/core-nats/pubsub?utm_source=chatgpt.com

[24] Redis Streams - Introduction
https://redis.io/docs/latest/develop/data-types/streams/?utm_source=chatgpt.com

[25] Migrate to the Responses API
https://developers.openai.com/api/docs/guides/migrate-to-responses/?utm_source=chatgpt.com

[29] Introducing OpenAI o3 and o4-mini
https://openai.com/index/introducing-o3-and-o4-mini/?utm_source=chatgpt.com

[30] [49] Rate limits | OpenAI API
https://developers.openai.com/api/docs/guides/rate-limits/?utm_source=chatgpt.com

[31] Rapid Shorts AI vs. VidID Comparison
https://sourceforge.net/software/compare/Rapid-Shorts-AI-vs-VidID/?utm_source=chatgpt.com

[32] [56] API Overview | OpenAI API Reference
https://developers.openai.com/api/reference/overview/?utm_source=chatgpt.com

[33] API Authentication
https://elevenlabs.io/docs/api-reference/authentication?utm_source=chatgpt.com

[34] [44] RFC 7009 - OAuth 2.0 Token Revocation
https://datatracker.ietf.org/doc/html/rfc7009?utm_source=chatgpt.com

[39] Create event - Microsoft Graph v1.0
https://learn.microsoft.com/en-us/graph/api/calendar-post-events?view=graph-rest-1.0&utm_source=chatgpt.com

[40] [82] REST API endpoints for GitHub Apps
https://docs.github.com/en/rest/apps/apps?utm_source=chatgpt.com

[41] [71] Refreshing user access tokens
https://docs.github.com/en/apps/creating-github-apps/authenticating-with-a-github-app/refreshing-user-access-tokens?
utm_source=chatgpt.com

[42] Authenticating to the REST API
https://docs.github.com/en/rest/authentication/authenticating-to-the-rest-api?utm_source=chatgpt.com

[43] Agent authentication
https://elevenlabs.io/docs/eleven-agents/customization/authentication?utm_source=chatgpt.com

[45] [77] RFC 7636 - Proof Key for Code Exchange by OAuth Public ...
https://datatracker.ietf.org/doc/html/rfc7636?utm_source=chatgpt.com

46 RFC 6750 - The OAuth 2.0 Authorization Framework
https://datatracker.ietf.org/doc/html/rfc6750?utm_source=chatgpt.com

47 Microsoft identity platform and OAuth 2.0 authorization ...
https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-auth-code-flow?utm_source=chatgpt.com

48 72 OWASP Top 10 API Security Risks – 2023
https://owasp.org/API-Security/editions/2023/en/0x11-t10/?utm_source=chatgpt.com

51 OWASP Top 10 for Large Language Model Applications
https://owasp.org/www-project-top-10-for-large-language-model-applications/?utm_source=chatgpt.com

53 Art. 5 GDPR – Principles relating to processing of personal ...
https://gdpr-info.eu/art-5-gdpr/?utm_source=chatgpt.com

54 California Consumer Privacy Act (CCPA) - Attorney General
https://oag.ca.gov/privacy/ccpa?utm_source=chatgpt.com

55 Data controls in the OpenAI platform
https://developers.openai.com/api/docs/guides/your-data/?utm_source=chatgpt.com

57 Artificial Intelligence Risk Management Framework (AI RMF 1.0)
https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.100-1.pdf?utm_source=chatgpt.com

58 Developer quickstart | OpenAI API
https://developers.openai.com/api/docs/quickstart/?utm_source=chatgpt.com

60 78 83 93 pgvector/pgvector: Open-source vector similarity search for ...
https://github.com/pgvector/pgvector?utm_source=chatgpt.com

61 Indexing
https://qdrant.tech/documentation/concepts/indexing/?utm_source=chatgpt.com

62 Hybrid search | Weaviate Documentation
https://docs.weaviate.io/weaviate/search/hybrid?utm_source=chatgpt.com

63 Milvus Architecture Overview
https://milvus.io/docs/architecture_overview.md?utm_source=chatgpt.com

64 Streaming API responses
https://developers.openai.com/api/docs/guides/streaming-responses/?utm_source=chatgpt.com

65 o4-mini Model | OpenAI API
https://developers.openai.com/api/docs/models/o4-mini?utm_source=chatgpt.com

67 Streaming text to speech
https://elevenlabs.io/docs/eleven-api/guides/cookbooks/text-to-speech/streaming?utm_source=chatgpt.com

68 Realtime API with WebSocket
https://developers.openai.com/api/docs/guides/realtime-websocket/?utm_source=chatgpt.com

69 actions/create-github-app-token
https://github.com/actions/create-github-app-token?utm_source=chatgpt.com

74 Instrumentation
https://prometheus.io/docs/practices/instrumentation/?utm_source=chatgpt.com

75 Performance Monitoring
https://docs.sentry.io/product/sentry-basics/performance-monitoring/?utm_source=chatgpt.com

76 WebSockets
https://fastapi.tiangolo.com/advanced/websockets/?utm_source=chatgpt.com

86 The official Python library for the OpenAI API
https://github.com/openai/openai-python?utm_source=chatgpt.com

87 The official Python SDK for the ElevenLabs API.
https://github.com/elevenlabs/elevenlabs-python?utm_source=chatgpt.com

88 langchain-ai/langgraph: Build resilient language agents as ...
https://github.com/langchain-ai/langgraph?utm_source=chatgpt.com

89 microsoft/autogen: A programming framework for agentic AI
https://github.com/microsoft/autogen?utm_source=chatgpt.com

90 run-llama/llama_index: LlamaIndex is the leading ...
https://github.com/run-llama/llama_index?utm_source=chatgpt.com

95 Python quickstart | Google Calendar
https://developers.google.com/workspace/calendar/api/quickstart/python?utm_source=chatgpt.com

96 Generating an installation access token for a GitHub App
https://docs.github.com/en/apps/creating-github-apps/authenticating-with-a-github-app/generating-an-installation-access-token-for-a-github-app?utm_source=chatgpt.com