AI Presence - Core Design [Updated Summary]

🏗️ Architecture

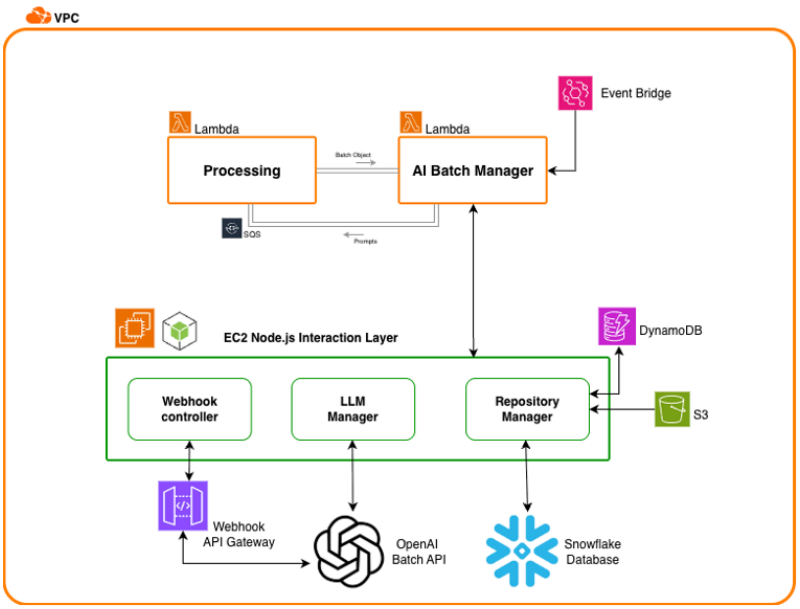
Description

A **serverless event-driven architecture**, processing Lambda which using a strategy patten, builds the relevant data pipeline on demand, coordinates closely with an EC2-hosted Node.js Communication Service, which provides orchestration, webhook control, LLM management, and repository handling. Communication between services remains asynchronous and decoupled**.**

Components

Component	Purpose
VPC	Isolates all Lambda functions and EC2 service within secure network boundaries
Processing Lambda	Dynamic Processing Lambda that orchestrates all phases of the data flow using an internal Strategy Pattern and dynamic pipeline configuration. Constructs the api request at the single level for the batch manager
AI Batch Manager	Handle prompt-triggered batch processing (initialize, process per provider/phase, and finalize). Continuously update job entities across the workflow.
EC2 Node.js Interaction Layer	Houses three core managers: Webhook controller (batch notification updates), LLM Manager interfaces with Model , Repository Manager manages data persistence to Snowflake, Presence Job on dynamoDB, config stored in S3.
DynamoDB	Provides persistent storage for storing data at prompt level to escort the lifecycle of a prompt (PresenceJob)
EventBridge	Triggers a scheduled daily job.
S3	
API Gateway	Exposes webhook endpoints for external system callbacks (OpenAI API Gateway)

Snowflake Database	External data warehouse for persistent storage of processed results.
OpenAI Batch API	External LLM service called by LLM Manager for AI/ML processing tasks.



Interaction layer

Docs:

AI Presence - Interaction Layer - API

S3 configurations object

```
{
  "open_ai": {
    "presence_batch": {
      "method" : "string",
      "system_prompt": "string",
      "model": "string",
      "max_res_tokens": number,
      "url": "string",
      "temperature": number
    },
    "analysis_batch":{
      "method" : "string",
      "system_prompt": "string",
      "model": "string",
      "max_res_tokens": number,
      "url": "string",
      "temperature": number,
      "output_schema" : {json_schema} *
    }
  }
}
```

json schema*

```
{
  "format": {
    "type": "json_schema",
    "name": "brand_mentions_by_sentiment",
    "schema": {
      "type": "object",
      "patternProperties": {
        "^[\\w\\s-]+$": {
          "type": "object",
          "properties": {
            "negative": { "type": "number" },
            "neutral": { "type": "number" },
            "positive": { "type": "number" }
          },
          "required": ["negative", "neutral", "positive"],
          "additionalProperties": false
        }
      }
    },
    "strict": true
  }
}

```

Processing Lambda

The lambda utilizes a strategy and builder design patterns in order to costumly adjust the required data pipeline required for each stage of the process.

`prompt-object`

```
{
  "promptId" : "string",
  "jobId"?: string,
  "topic" : "string",
  "promptText" : "string",
  "batchId"?: "string",
  "method" : "string",
  "systemPrompt": "string",
  "model": "string",
  "llmProvider": "open_ai", // currently only one llm provider is supported
  "phase": "initial" | "follow_up"
  "maxResTokens": number,
  "url": "string",
  "temperature": number,
  "vectorStoreId": "string",
  "outputSchema"? : object {json_schema*}
}
```

`batch-object`

```
{
  "custom_id": "prompt_id",
  "method": "method",
  "url": "url",
  "body": completions-request | responses-request
}
```

`completions-request`

```
{
    "model": "{{ model }}",          // REQUIRED: model ID
    "temperature": {{temperture}},
    "max_tokens": number
    "messages": [
      {
        "role": "system",
        "content": "{{system_prompt}}."
      },
```

```
      {
        "role": "user",
        "content": "{{prompt_text}}!"
      }
    ]
}

```

`responses-request`

```
{
  "model": "{{ model }}",
  "temperature":  {{temperture}},
  "max_output_tokens": number,
  "text": {{ json_schema }},
  "store": false,
  "instructions": string {{brands-key}}
  "input": [
    {
      "type" : "message", // always "message"
      "role" : "system",
      "content": {{system_prompt}}
    },
    {
      "type" : "message", // always "message"
      "role" : "user",
      "content": {{prompt_text}}
    }
  ]
  }
```

🗄️ Data

💼 The PresenceJob Entity

💡 Overview

`PresenceJob` represents the **entire execution lifecycle** for a single prompt's daily AI Presence run.

🧮 DynamoDB Design

- **TTL:** 24 hours (automatic cleanup after each daily cycle).
- **Atomic updates:** Safe concurrent updates for webhooks and batch processing.
- **Update mechanism:**
  - Webhook events update job status atomically.
  - batches use separate fields to avoid overwriting.

```
{
  id: string;                          // Unique ID (UUID)
  topic: string;                       // Topic of analysis
  promptId: string;                    // Associated prompt identifier
  promptResponseId?: string             // Prompt response Id
  startTimestamp: string;              // ISO start time
  lastUpdateTimestamp?: string;        // Filled at each job update
  error?: string;                      // Any stage error
  llmProvider: "open_ai"               // currently a single provider
supported
  phase: "initial" | "follow_up"       // explicitly state job's phase

  batches: [{
    batchId: string;                  // Batch ID (OpenAI)
    batchInputFileId: string;
    batchOutputFileId?: string
    batchStatus: 'QUEUED' | 'RUNNING' | 'COMPLETED' | 'FAILED';
    tokenUsage?: {
      inputTokens: number;
      outputTokens: number;
      totalTokens: number;
    };
  },
  ...
  ]

```

```
}
```

❄️ Snowflake Database Design

💡 Overview

Our **Snowflake database** serves as the **source of truth** for historical visibility data, allowing us to query, aggregate, and visualize insights over time.

🧱 `PROMPT` Table

Column	Type	Description
`PROMPT_ID`	`VARCHAR` (PK)	Unique identifier for the prompt
`TOPIC`	`VARCHAR`	Topic prompt associated with
`PROMPT_TEXT`	`TEXT`	The full prompt text
`CREATED_AT`	`TIMESTAMP`	When prompt was created
`IS_ACTIVE`	`BOOLEAN`	Enables prompt activation/deactivation for rotation

🧱 `PROMPT_RESPONSE` Table

Column	Type	Description
`PROMPT_ID`	`VARCHAR`	Unique identifier for the prompt
`DATE`	`Date`	Date
`RESPONSE_ID`	`VARCHAR` (PK)	Unique identifier for the response
`RESPONSE_TEXT`	`TEXT`	The full response.

🧱 `PRESENCE_SENTIMENT_MENTIONS` Table

Column	Type	Description
`PROMPT_ID`	`VARCHAR` (FK → `PROMPT.PROMPT_ID`)	The prompt used in this analysis
`RESPONSE_ID`	`VARCHAR` (PK) (FK→ `PROMPT_RESPONSE.RESPONSE_ID`)	Unique identifier for the response
`MODEL`	`VARCHAR`	model (e.g gpt-4.1-nano)

`BRANDKEY`	`VARCHAR` (PK)	Brand name analyzed
`POSITIVE_MENTIONS`	`INTEGER`	Count of positive mentions
`NEUTRAL_MENTIONS`	`INTEGER`	Count of neutral mentions
`NEGATIVE_MENTIONS`	`INTEGER`	Count of negative mentions for brand
`DATE`	`DATE`	Analysis date (daily granularity)
`TOTAL_TOKENS`	`INTEGER`	Tokens consumed in LLM analysis

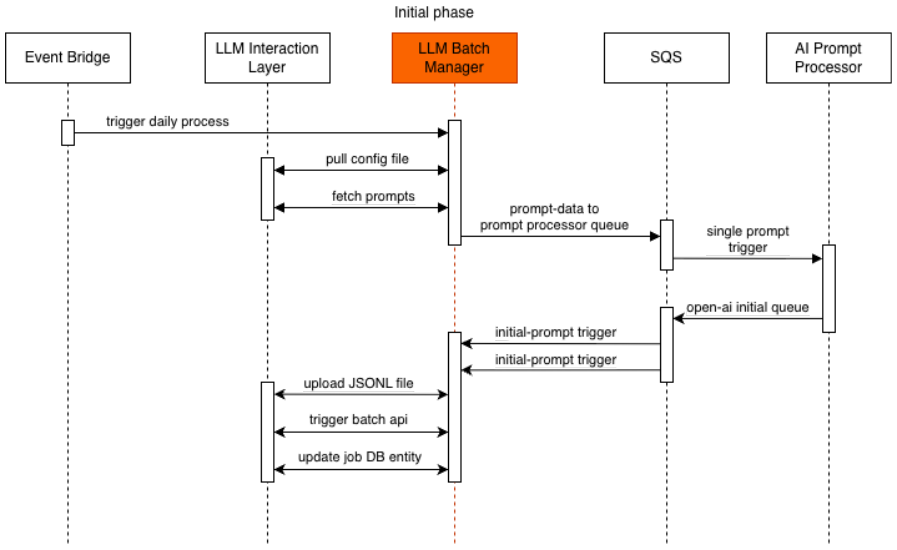
🪓 AI Batch Manager

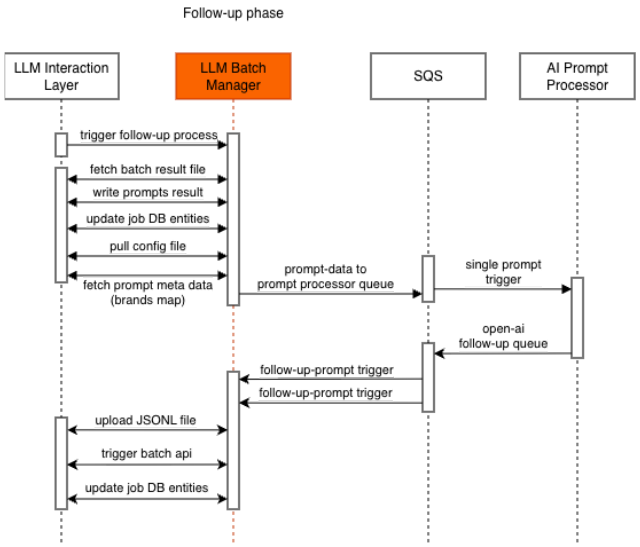
Responsibilities Overview

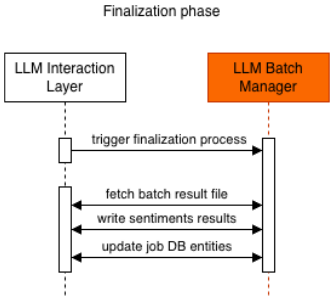
1. **Prompt triggering -** Create config (per prompt) → trigger (using SQS) the AIPromptProcessor lambda
   a. per initial/follow-up phase
2. **Batch initialization -** Consume SQS messages created by the AIPromptProcessor lambda → initialize batch process
   a. per LLM provider → per initial/follow-up phase
3. **Batch Finalization**
4. Throughout the flow update job entities as needed

Flows

* Mind the following schemas neglects [interaction layer ↔ ai prompt processor] processes, for feature focusing & simplicity







Lambda ↔ SQS Limitations

AWS Lambda has a built-in & non-configurable mechanism to Manage load balancing and functions invocations. This means that multiple “AI batch manager” functions can be triggered for a stream of SQS messages fired around at the same time.

Implications

ANY size of prompt count WILL be split and handled by multiple “AI Batch Manager” functions, leading to multiple batches.

Short (?) term solution

Limiting the “Maximum concurrency” of SQS triggers to “2” (2 being the minimum allowed by AWS).

Meaning:
10 prompts → 2 batches (each one with ~5)
1000 prompts → 2 batches (each one with ~500)

AI Batch Manager ↔ SQS (↔ AI Prompt Processor) - I/O specification

Refer to AI Presence - Core Design [Updated Summary] | Processing Lambda

EventBridge → AI Batch Manager - integration

Daily trigger JSON input (from eventBridge)

```
{
  eventSource: "event_bridge"
  phase: "initial"
}
```

AI Batch Manager ↔ Interaction Layer - I/O specification

> This section represents only the events for lambda invocation by ai-presence-interaction-layer. for full integration docs refer to AI Presence - Interaction Layer - API | Update Job by ID

Trigger AI batch manager (used by Interaction layer at Webhook invocations)

Usage notes: manual lambda invocation using *aws-sdk*

Invocation event:

```
{
  eventSource: "manual_invocation", // this exact string value
  phase: "follow_up" | "final", // these exact string values
  llmProvider: "open_ai" // this exact string value
  responseId: string // e.g. open-ai's output-file-id
  batchId: string // batchId
}
```

Response (200): empty