ChatGPT

Lead Ingestion & Outreach Platform: Phases 4–6 Architecture

Building on the existing linear pipeline architecture [1] [2], Phases 4–6 introduce new agents and an event-driven outreach layer while preserving backward compatibility. Each new agent has well-defined input/output contracts (keys) added to the existing lead record schema. We summarize each agent's role, inputs/outputs, and schema changes, then describe the event flow for outreach. The architecture uses an in-memory **MessageBus** (pub/sub) and **StateStore** abstraction now, with the ability to swap in Kafka/Redis/Postgres later. All changes are designed to be optional extensions so the original pipeline can still run unchanged.

Phase 4 – Lead Qualification & Enrichment

Objective: Compute quality metrics and enrich leads with missing contact details before outreach. This builds on the routed leads (from Phase 1–3) and adds two agents:

- **EnrichmentAggregatorAgent:** Gathers additional lead data (e.g. contact info, social profiles) from external APIs to fill gaps. It consumes `routed_leads` (each with fields like `dedup_key`, address, website, etc.) and outputs `enriched_leads`. New fields (under an `enrichment.` namespace) are added, such as verified `email`, `phone`, `industry_category`, etc. This agent ensures downstream processes have as complete information as possible.

- **LeadScoringAgent:** Calculates lead-quality scores and flags. It consumes `enriched_leads` and outputs `scored_leads`. It computes metrics such as **quality.completeness_score** (fraction of key contact fields present), **quality.confidence_score** (e.g. geocoding accuracy for location), **quality.category_score** (relevance of business category), and **quality.size_score** (estimated business size). These scores are added under a `quality.` namespace. For example, `quality.completeness_score` is a 0–1 float, where 1.0 means all expected fields are filled. All input keys (including routing info and dedup keys) are preserved. A sample contract summary is:

Agent	Input Key(s)	Output Key(s)
EnrichmentAggregatorAgent	`routed_leads` (list of lead objects)	`enriched_leads` (adds enrichment.* fields)
LeadScoringAgent	`enriched_leads`	`scored_leads` (adds quality.* fields)

New Output Schema Fields (Phase 4): The following keys are appended to each lead record by these agents:

Field	Description
`enrichment.email`	Verified contact email (if found)
`enrichment.phone`	Verified phone number (if found)
`enrichment.other_ids`	Other identifiers (e.g. social profiles)
`quality.completeness_score`	Fraction (0–1) of required fields present
`quality.confidence_score`	Data confidence (e.g. geocode or data accuracy)
`quality.category_score`	Relevance of business category (0–1)
`quality.size_score`	Estimated business size/importance (0–1)

These fields augment the existing lead schema. The agents must validate and preserve all prior fields (e.g. `dedup_key`, address, website) and only add new keys. This follows the pipeline contract style already used in Phases 0–3 [1].

Agent Responsibilities:

- **EnrichmentAggregatorAgent:**
- Fetches data from enrichment services (e.g. third-party APIs, web scrapers) for each lead.
- Merges results to fill missing fields like email, phone, and company details.
- Updates the lead record with new keys (e.g. `enrichment.email`).
- Passes through original fields unchanged (including dedup and routing fields) for downstream consistency.

- **LeadScoringAgent:**

- Computes scoring heuristics per lead (e.g. completeness of contact info, confidence of location, category match).
- Writes the scores into `quality.*` fields (e.g. `quality.completeness_score`, `quality.confidence_score`).
- May also flag low-scoring leads or assign a `lead.qualified` boolean.
- Outputs `scored_leads` with all original and enrichment fields preserved.

These agents plug into the pipeline after routing. For example, the output of `LeadRouterAgent` feeds `EnrichmentAggregatorAgent`, whose output feeds `LeadScoringAgent`, continuing the deterministic chain. All new scoring and enrichment logic is implemented with fail-fast contract checks (as before [3]) and unit tests to ensure no unexpected schema drift.

Phase 5 – Landing Page Generator

Objective: Automatically create SEO-optimized landing pages for TARGET leads. Phase 5 adds a **LandingPageGeneratorAgent** plus templating components that use lead data to produce deployable web pages.

- **LandingPageGeneratorAgent:** Consumes `scored_leads` (from Phase 4) and selects leads with route=TARGET. For each target lead, it generates a static landing page using content templates. It injects business-specific data (e.g. name, logo, description), SEO metadata, and call-to-action links. This agent outputs `landing_pages`, a structure linking each lead's `dedup_key` to a generated HTML file or page URL. It also produces `landing_stats` (counts of pages created).

Templating Components:
We use a set of HTML/CSS templates with placeholders. The agent fills in fields like business name, category, and keywords, and writes out HTML (or markdown) content. For example: - **Templates:** Base layouts per industry (e.g. "real estate", "auto repair"), with placeholder tags `{{business.name}}`, `{{business.tagline}}`, etc.
- **Data Injection:** The agent merges each lead's data (including enrichment results) into the template. For example, a template's `<title>` and meta description are filled with the lead's business name and keywords for SEO.
- **Brand Customization:** If an external brand directory or assets exist, logos and brand colors can be pulled in (future extension).

Contract & Schema:
The agent's input is `scored_leads`. Its output is `landing_pages` – a map of lead IDs to page content. For clarity:

Agent	Input Key(s)	Output Key(s)
LandingPageGeneratorAgent	`scored_leads`	`landing_pages` (maps `dedup_key` → page content/URL), plus `landing_stats`

No existing lead fields are removed; rather, the agent adds HTML assets. A summary: the agent creates static pages in a format suitable for one-click deployment (e.g. as Markdown or HTML files), so they can later be hosted on a web server or S3 bucket. This respects the Phase 5 goal of "static site generation" and preview pipelines [4] .

Phase 6 – Outreach Automation

Objective: Orchestrate asynchronous outreach (email, SMS/WhatsApp, CRM integration) using an event-driven design. Instead of a simple linear flow, leads now drive events that cascade through multiple agents. We introduce a **MessageBus** (pub/sub) and a **StateStore** for lead status.

Overview & Events

We define these core events:

- `lead.contacted` – Published when a lead is first contacted (e.g. an email or message is sent). Payload includes `lead_id`, contact channel, timestamp.
- `lead.responded` – Published when a response from the lead is received (e.g. email reply). Payload includes `lead_id`, response content, timestamp.
- **(Optional)** `lead.bounced` – If an email bounces or fails.
- **(Optional)** `lead.unsubscribed` – If a lead opts out.

In an event-driven pattern, these events are published to the MessageBus so any interested agent can react [5]. For example, the **OutreachOrchestratorAgent** will subscribe to these events and update lead states or trigger follow-ups. Other agents (EmailAgent, WhatsAppAgent, CRMAdapter) may also consume these events. This decouples components: senders do not call receivers directly [5].

Agents & Workflow

- **OutreachOrchestratorAgent:** Core coordinator for outreach workflows. It subscribes to the message bus and maintains state in the StateStore for each lead's campaign (e.g. "new", "emailed", "responded", "closed"). Workflow steps (in simplified form):

- **Kickoff:** On a new `scored_lead` (with route=TARGET), orchestrator schedules initial outreach (e.g. email or WhatsApp) by publishing an internal command or calling EmailAgent/WhatsAppAgent.

- **Contacted:** When EmailAgent or WhatsAppAgent publishes `lead.contacted`, the orchestrator updates the lead state to "contacted" in the StateStore.

- **Await Response:** The orchestrator monitors for `lead.responded`. If received within a timeout window, it updates state to "responded" and may trigger a "thank you" or CRM update. If no response after retries, it may escalate (e.g. send a follow-up email or mark as "no response").

- **Follow-Up:** The orchestrator can enforce a retry policy (limited attempts) by republishing outreach commands. Duplicate events or late messages are ignored since we make processing idempotent.

- **Completion:** Once a lead responds or is finalized, the orchestrator may publish an event (e.g. `lead.engaged`) or call a CRMExportAgent to sync with a CRM.

This agent effectively mediates the outreach conversation, ensuring agents collaborate. It embodies the "mediator" topology: it keeps state and controls flow [6], unlike a simple broadcast. By centralizing logic here, we avoid hardcoding email agent → next agent links, and can modify campaigns without touching lower-level agents [7] [8].

- **EmailAgent:** Sends personalized email campaigns. Subscribes to orchestrator commands (or a specific topic like `outreach.email.send`). When invoked, it sends an email via SMTP or an email API, using templates that can include the landing page link. Upon successful send, it publishes `lead.contacted`. If delivery fails, it can publish `lead.bounced`.

- **WhatsAppAgent:** Similar to EmailAgent but for WhatsApp or SMS. It sends messages via an SMS gateway or WhatsApp API. On send, also emits `lead.contacted`.

- **CRMAdapterAgent:** Pushes qualified leads into a CRM system (e.g. Salesforce). It may run in response to events like `lead.responded` or on a schedule. It consumes lead data (via message or pull from StateStore) and invokes a CRM API, then records success in `crm_stats`. This agent is triggered indirectly by orchestrator state changes.

These agents all interact via the MessageBus. For example, when EmailAgent sends an email, it **publishes** `lead.contacted`. The orchestrator (and possibly others) **subscribes** to this event. This publish/subscribe decouples producer and consumer [5].

Event Flow Diagram (conceptual):

```
[LeadScoringAgent] --> (scored_leads) --> [OutreachOrchestratorAgent]
           Orchestrator Agent   <---- MessageBus (lead.contacted,
lead.responded, etc) ---->   [EmailAgent]
                                          \----> [WhatsAppAgent]
                                          \----> [CRMAdapterAgent]
```

(Arrows via MessageBus.)

1. **New Lead:** Scored lead enters orchestrator.
2. **Contact:** Orchestrator tells EmailAgent/WhatsAppAgent to send messages.
3. **On Send:** Email/WhatsAppAgent emit `lead.contacted`.
4. **On Response:** External events (e.g. mail API webhook) cause `lead.responded` to be published.
5. **State Update:** Orchestrator updates state (via StateStore) and may end flow or schedule next step.

This event-driven design is in line with best practices: it allows each agent to focus on one task and react to events [7] [9]. It also ensures components remain decoupled and scalable. For instance, new channels (e.g. a LinkedInOutreachAgent) can be added by subscribing to the same events without changing the orchestrator.

MessageBus and StateStore Abstractions

To support this, we introduce two abstractions:

- **MessageBus:** An interface for pub/sub messaging. Currently implemented as an in-memory channel (e.g. NodeJS EventEmitter or simple queues), it allows publishing events (topics like `lead.contacted`, `outreach.email.send`, etc.) and subscribing. In Phase 7 we plan to swap in a durable broker (e.g. Kafka, Redis Streams, RabbitMQ). By coding to the MessageBus interface, agents can publish without knowing the underlying system [10].

- **StateStore:** A simple key-value store for workflow state. E.g. it tracks `lead_id → {status, attempt_count, last_contacted}`. Initially this can be an in-memory map. In Phase 7 we will support plugging in Redis or a database for persistence. This follows the event-carried state transfer pattern: components subscribe to events and update their own local state (the StateStore) to maintain a consistent view [9].

By using these abstractions, we ensure all messaging is idempotent and reliable. For example, the Publisher/Subscriber pattern notes that **message ordering is not guaranteed** and duplicates can occur [11] [12], so agents must handle these cases. Our agents, like in earlier phases, will treat processing idempotently (e.g., only marking "contacted" once) and rely on the pipeline's dedup keys and test suite for consistency [3] [12].

Contract Summary (Phase 6)

Event / Command	Producer	Consumer(s)	Description
`lead.contacted`	EmailAgent, WhatsAppAgent	OutreachOrchestratorAgent, Logging, Analytics	Lead was contacted via outreach.
`lead.responded`	InboundService (email webhook)	OutreachOrchestratorAgent	Lead replied to outreach.
`lead.bounced`	EmailAgent	OutreachOrchestratorAgent	Email failed to deliver.
`outreach.email.send`	OutreachOrchestratorAgent	EmailAgent	Command to send outreach email.
`outreach.whatsapp.send`	OutreachOrchestratorAgent	WhatsAppAgent	Command to send WhatsApp message.

These event schemas carry minimal payload (primarily `lead_id`, timestamps, and relevant content). Consumers retrieve any needed lead details from the StateStore or underlying lead datastore. This keeps events lightweight and focused [5].

Backward Compatibility and Migration

Crucially, all new agents and events are optional extensions. If the message bus is turned off, or outreach agents are disabled, the system still runs the old linear pipeline (MapsSearch → Normalize → Validate → Route → Format → Export) as before. The new outputs (`scored_leads`, `landing_pages`) simply won't be used until enabled. This ensures we meet existing milestone gates (Phases 0–3) and can incrementally test Phases 4–6. In Phase 7, we will swap in production-grade infra (Kafka, Redis, persistent DB) for

MessageBus and StateStore, but by then the contract-driven design and event patterns will already be in place [10] [9].

References: The design follows the phased roadmap [2] [4] [13] and core principles (explicit contracts, idempotency) from the existing spec [3]. Event-driven best practices (decoupling, pub/sub) and the orchestrator pattern inform Phase 6 design [5] [7] [8].

---

[1] [3] lead_ingestion_outreach_platform_architecture_spec.md
file://file_00000000670c71fa9a6f62f10abf8588

[2] [4] [13] milestone_based_roadmap_timeline (1).md
file://file_00000000e11871fa8524b4d5b13ac930

[5] [6] Event-Driven Architecture Style - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven

[7] [8] How to build a multi-agent orchestrator using Flink and Kafka
https://www.confluent.io/blog/multi-agent-orchestrator-using-flink-and-kafka/

[9] The Ultimate Guide to Event-Driven Architecture Patterns | Solace
https://solace.com/event-driven-architecture-patterns/

[10] [11] [12] Publisher-Subscriber pattern - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber