

# Enterprise Customer Onboarding Agent — Solution Design Document

February 2026 — StackAdapt Case Study Submission

## 1. Architecture Overview

The solution implements an **AI-powered automation agent** that orchestrates enterprise customer onboarding from deal closure through SaaS provisioning. Built with **LangGraph** for state machine orchestration and **FastAPI** for the REST interface, the agent integrates with multiple enterprise systems, validates business rules, assesses risks using LLM intelligence, and takes autonomous actions including tenant provisioning and task management.

### 1.1 System Integrations

System	Objects & Key Fields
Salesforce CRM	Account (Id, Name, IsDeleted, BillingCountry, Industry, OwnerId), Opportunity (StageName, Amount, CloseDate, AccountId, ContractId), User (IsActive, Email, ProfileId). <a href="#">[API Ref]</a>
NetSuite ERP	Invoice (status, dueDate, amountRemaining, total, tranId, entity). <a href="#">[API Ref]</a>
CLM System	Contract status (DRAFT/SENT/SIGNED/EXECUTED), signatures, effective_date, expiry_date, key_terms. Mock simulating DocuSign CLM.
Provisioning	Tenant creation + 14-task onboarding checklist with dependencies, owners, and due dates.

**Field Selection Rationale:** `Account.IsDeleted` validates account exists; `Opportunity.StageName="Closed Won"` confirms deal closure; `Invoice.status` identifies payment blockers; `CLM.status` in {EXECUTED, SIGNED} required for provisioning. Minimizes API calls while capturing decision-critical data.

### 1.2 Data Flow

1. Webhook received with `account_id` and `correlation_id`
2. Sequential fetch: Salesforce → CLM → NetSuite
3. Invariant validation across 5 domains
4. LLM risk analysis (or rule-based fallback)
5. Decision: PROCEED → provision; BLOCK/ESCALATE → notify
6. Generate reports and complete

## 2. AI Agent Application

The agent leverages **OpenAI GPT-4o-mini** (configurable via `OPENAI_MODEL`) for intelligent analysis with a deterministic rule-based fallback, ensuring operation even without LLM connectivity.

### 2.1 LLM-Powered Intelligence

- **Risk Analysis:** Evaluates API errors, violations, warnings → risk levels (low/medium/high/critical) with business impact and resolution time estimates.
- **Summary Generation:** Human-readable reports for CS teams, translating technical states to actionable insights.
- **Action Recommendations:** Prioritized steps with owner assignment (CS, Finance, IT/DevOps, Legal, Sales Ops).
- **Error Interpretation:** Converts technical codes (e.g., `INVALID_SESSION_ID`) to plain English with resolution steps and responsible teams.

### 2.2 Autonomous Actions

Upon decision, the agent performs the following autonomous actions:

- (1) Auto-provisions the SaaS tenant with tier-appropriate configuration (Enterprise/Growth/Starter).
- (2) Creates a 14-task onboarding checklist with dependencies, ownership (system/cs\_team/customer), and due dates.
- (3) Sends Slack notifications to `#cs-onboarding-alerts` (blocked) or `#cs-onboarding` (success/escalate).
- (4) Sends customer-facing welcome emails with tenant ID and login URL.
- (5) Generates HTML/Markdown/JSON reports for auditing and traceability.

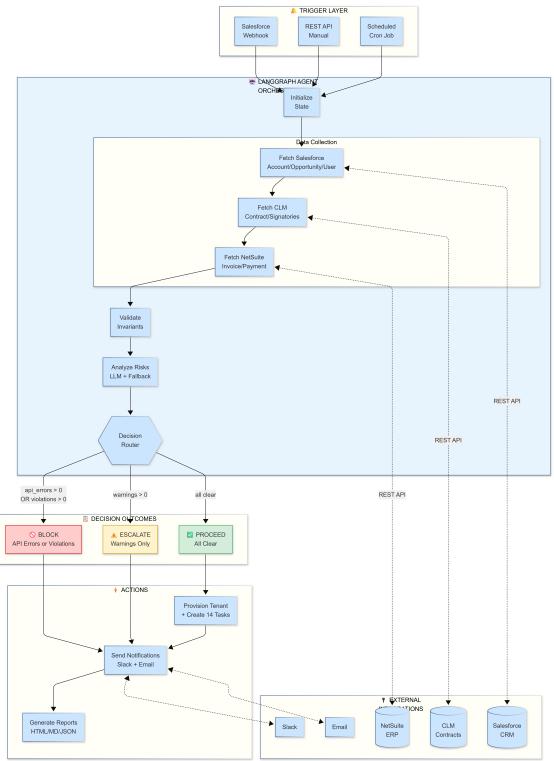


Figure 1: Architecture: triggers, LangGraph orchestration, integrations.

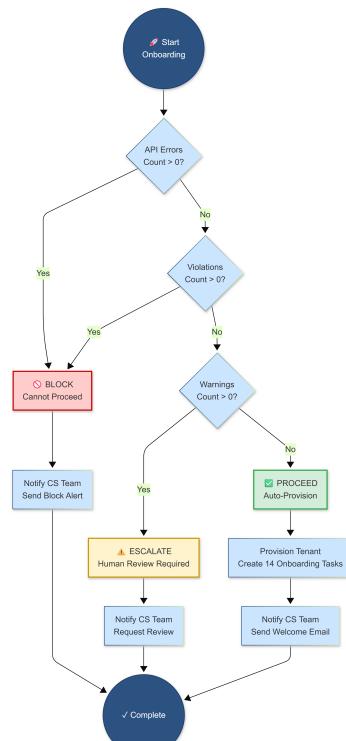


Figure 2: Decision: API Errors/Violations → BLOCK, Warnings → ESCALATE, Clear → PROCEED.

## 2.3 Onboarding Task Management

The provisioning system creates a 14-task onboarding checklist with the following structure:

### Task Categories:

- **Automated (4)**: Tenant creation, API credentials, welcome email, training materials
- **CS Team (5)**: Schedule/conduct kickoff, SSO config, custom reports, 30-day check-in
- **Customer (4)**: Verify login, complete tour, invite team, create first campaign
- **Milestone (1)**: Onboarding complete marker

Tasks include dependencies (e.g., “Complete Tour” depends on “Verify Login”), due dates calculated relative to provisioning, and ownership assignment enabling filtered views per team.

## 2.4 Business Rule Validation (Invariants)

The agent enforces a two-tier validation system across five domains. **Violations** are blocking (cause BLOCK decision); **Warnings** are non-blocking (cause ESCALATE for human review).

Domain	Blocking Violations	Non-blocking Warnings
Account	Missing Id/Name, IsDeleted=true	Missing BillingCountry, Industry, OwnerId
Opportunity	Stage ≠ “Closed Won”, Missing Id/AccountId	Missing Amount, CloseDate, ContractId
Contract (CLM)	Status ∉ {EXECUTED, SIGNED}, Missing contract_id	No effective/expiry date, pending signatures
Invoice	Voided, Cancelled, Missing invoice_id	Overdue, Pending, <50% paid, missing due_date
User	Inactive, Missing Id/Email/ProfileId	Missing Title, Department, ManagerId

## 3. Orchestration & Event-Driven Flows

Trigger	Source	Endpoint
Webhook	Salesforce Flow	POST /webhook/onboarding
Manual	CS Team	POST /demo/run/{account_id}
Batch	Scheduler	POST /demo/run-all

**Error Simulation:** /demo/enable-random-errors injects failures (401 auth, 400 validation, 429 rate limit, 500 server) at configurable rates for chaos testing without modifying business logic.

**Report Generation:** Each run produces three output files:

- run\_report.\*.md – Full Markdown report with all details
- email\_\*.html – Professional HTML email (blocked, escalation, success, or welcome)
- audit\_\*.json – Machine-readable audit log



Figure 3: State machine: full lifecycle from webhook ingestion through decision routing to completion.

## 3.1 API Error Classification & Handling

Each integration returns structured error payloads with `error_type`, `error_code`, `http_status`, `message`, `resolution`, and `owner`. Errors are classified and handled as follows:

Error Type	HTTP	Example Code	Owner	Resolution
Authentication	401	INVALID_SESSION_ID	IT/DevOps	Re-authenticate, refresh token
Authorization	403	INSUFFICIENT_ACCESS	System Admin	Check profile/permission sets
Validation	400	REQUIRED_FIELD_MISSING	Data Admin	Fix field values in source system
Rate Limit	429	REQUEST_LIMIT_EXCEEDED	Integration Admin	Implement backoff, increase quota
Server Error	500	SERVER_ERROR	Support Team	Check status page, retry later

## 4. Trade-offs, Assumptions & Considerations

### 4.1 Key Trade-offs

Decision	Trade-off
API Errors → BLOCK	Integrity over speed
Sequential Data Fetching	Clearer failure tracking but has higher latency
Rule-based Fallback	Less smart but reliable
14 Fixed Tasks	Predictable but rigid
LLM Advisory Only	Auditable but less adaptive

### 4.2 Assumptions

- Salesforce is source of truth for accounts
- CLM status reflects actual signatures
- Invoice payment status is current
- All invoices are single-currency (CAD) with no partial payments, credit memos, or discounts
- One invoice per account per onboarding run
- One Opportunity per Account for onboarding (no multi-deal scenarios)
- Contract has a single version with no amendments
- CS monitors Slack channels for alerts
- Single onboarding per account at a time
- Customer email available in CLM signatories
- Account ID is consistent across all systems

### 4.3 Limitations

- **Mock integrations:** Salesforce, NetSuite, and CLM use simulated data; not connected to real APIs
- **In-memory state:** Provisioning and task data lost on restart; no persistent database
- **Single instance:** No horizontal scaling or load balancing for high-volume webhooks
- **No approval UI:** ESCALATE decisions notify Slack but lack a UI for human approve/reject workflow
- **No webhook auth:** Inbound webhooks not verified via HMAC signature or shared secret

## 4.4 Testing & Demonstration

The system includes 5 pre-configured demo scenarios validating all decision paths:

Account ID	Scenario	Description	Decision
ACME-001	Happy Path	All checks pass, contract executed, invoice paid	PROCEED
BETA-002	Opportunity Blocked	Opportunity in “Negotiation” stage	BLOCK
GAMMA-003	Overdue Invoice	Invoice 25+ days overdue, needs finance review	ESCALATE
DELETED-004	Deleted Account	Account marked <code>IsDeleted=true</code>	BLOCK
MISSING-999	Account Not Found	Account does not exist in any system	BLOCK

## 5. Multi-Agent Architecture via Model Context Protocol (MCP) Server Connections — *Proposed Extension*

*Note: This section describes a proposed architectural extension. The current implementation uses direct function calls within a single agent.*

The architecture is designed to support multi-agent collaboration through the **Model Context Protocol (MCP)**, enabling specialized agents to interact through standardized, governed tool interfaces rather than direct API access. This design cleanly separates decision logic from execution concerns while allowing agents to remain focused on their respective domains.

Agent	Responsibility & MCP Tools
Coordinator	Orchestrates end-to-end workflow ( <code>salesforce.*</code> , <code>provision.*</code> )
Contract Agent	Monitors contract execution and signatures ( <code>clm.get.contract</code> )
Finance Agent	Evaluates billing and payment state ( <code>netsuite.get.invoice</code> )
Task Monitor	Detects overdue onboarding tasks ( <code>tasks.get.overdue</code> )

**Benefits:** Independent versioning per integration • Clear domain ownership • Centralized access control and audit logging • Scalable agent complexity • Cross-agent context sharing.

**Concurrency & API Execution Strategy:** MCP servers would act as controlled execution layers for external APIs, enabling concurrent and batch-aware request handling. Where supported, batch or composite API calls are leveraged to reduce network overhead and latency (e.g., Salesforce composite reads). If batch execution fails or yields partial results, the system safely falls back to bounded parallel single-entity requests.

Concurrency limits, rate limiting, and circuit breakers would be enforced at the MCP layer to prevent downstream system overload and cascading failures. This approach improves onboarding throughput while preserving idempotency, error isolation, and predictable failure handling across agents.

## 6. Production Considerations

*For additional production enhancements, see the `production_roadmap` directory in the repository.*

### Cloud Deployment

**Frontend:** A role-based operations interface (e.g., React, ASP.NET, or Node.js-backed UI) providing real-time visibility into onboarding runs, decisions, and system health. Enables Customer Success and Operations teams to review onboarding outcomes, inspect violations or API errors, manually intervene when required, and track provisioning progress and onboarding task completion.

**AWS Infrastructure:** The onboarding agent executes as containerized services on ECS/Fargate, enabling horizontal scaling based on inbound webhook volume. LLM inference is handled via **AWS Bedrock** rather than direct OpenAI calls, allowing multi-model flexibility (Claude, Llama, Titan), private networking with traffic contained within the VPC, IAM-based authentication, and enterprise-grade security and compliance. Bedrock enables cost controls and model switching without application changes.

**Docker + Kubernetes:** Containerization ensures consistent environments across development, staging, and production. Kubernetes orchestration (or ECS equivalents) provides auto-scaling for traffic spikes (e.g., batch onboarding runs), self-healing services, rolling updates, and zero-downtime deployments as agent logic evolves.

### Agentic Execution & API Governance

**Deterministic Decision Boundaries:** The onboarding agent follows a hybrid execution model in which deterministic business rules define all state transitions and final decisions (`PROCEED`, `ESCALATE`, `BLOCK`). LLM components are strictly advisory and are limited to risk interpretation, summarization, and action recommendations. LLM outputs cannot directly mutate agent state or override invariant checks, ensuring predictable, testable, and auditable outcomes.

**Optimized Data Fetching:** To minimize latency when fetching from Salesforce, CLM, and NetSuite, the agent employs a tiered optimization strategy: (1) Batch API requests where supported (e.g., Salesforce Composite API) to retrieve Account, Opportunity, User, and Contract in a single round-trip; (2) Concurrent multithreaded fetches for systems that don't support batching, with configurable thread pools per integration; (3) Bounded retry logic with exponential backoff (max 3 retries, 1s/2s/4s delays) for transient failures before escalating to permanent failure state.

**Idempotency & Correlation-Based Execution:** Each onboarding run is keyed by a `correlation_id` propagated across webhook ingestion, integrations, logs, reports, and notifications. This enables safe replays, deduplication of duplicate webhook events, and re-entrant execution without risk of duplicate provisioning, notifications, or side effects.

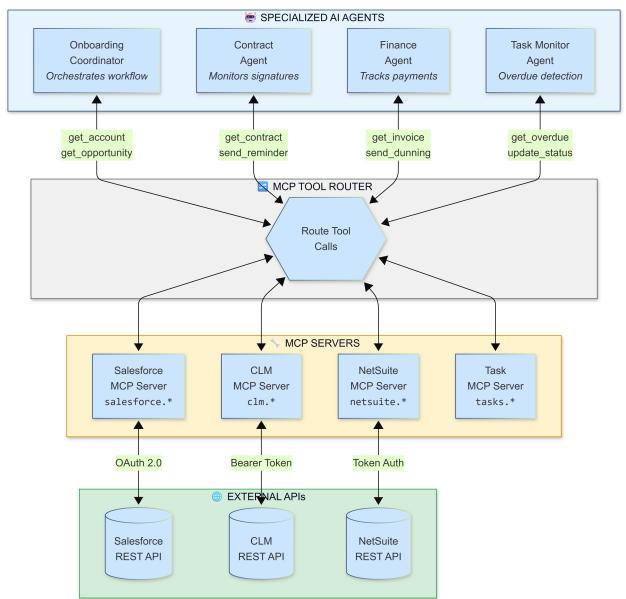
**API Error Classification & Retry Strategy:** External API failures are classified into retryable (e.g., transient 5xx, rate limiting) and non-retryable (e.g., validation, authorization) categories. Retryable errors follow bounded exponential backoff, while non-retryable errors immediately halt onboarding and surface actionable diagnostics. Circuit breakers prevent cascading failures across dependent systems.

**Rate Limiting & Backpressure:** Inbound webhook traffic is protected via per-source rate limits and queue-based ingestion to absorb traffic spikes from batch updates or retries. Concurrency limits are enforced per integration to prevent overwhelming Salesforce, CLM, or NetSuite APIs during peak load.

**Agent Versioning & Safe Rollouts:** Every onboarding run records the agent version, rule set, and prompt version used. This enables full auditability, regression analysis, and controlled rollouts (e.g., canary deployments) of agent logic and prompt updates without impacting in-flight executions.

**LLM Data Governance & Security:** All data sent to LLMs is explicitly field-filtered and sanitized to prevent leakage of sensitive information. PII is masked prior to inference, and external system credentials are never exposed beyond their integration boundaries. Trust boundaries between systems are strictly enforced at the API layer.

**Multi-Agent Interoperability:** As described in Section 5, wrapping integrations behind MCP servers cleanly separates agent logic from external system access, enabling independent versioning, access control, and audit logging. If these MCP servers evolve into autonomous agents, the A2A protocol provides a standardized layer for agent-to-agent collaboration without exposing internal state or tools across boundaries.



**Figure 4:** MCP: Specialized agents collaborate through shared execution servers wrapping external APIs.

### CI/CD & Observability

**CI/CD** (GitHub Actions / CodePipeline): Automated pipelines validate agent logic on every pull request, including unit tests for invariant checks, risk analysis, and decision routing. Deployments are reproducible and auditable, with fast feedback loops, reduced manual error, and the ability to roll back safely in case of regressions impacting onboarding outcomes.

**LangSmith Enterprise:** Provides deep observability into LLM-powered components, including prompt inputs/outputs, token usage, latency, and cost tracking. Supports prompt versioning, regression detection, and A/B testing of risk analysis and summary generation logic without affecting deterministic rule-based execution.

**Monitoring & Alerting:** Prometheus metrics capture agent throughput, API error rates, and decision distributions. Distributed tracing via DataDog or Jaeger enables end-to-end visibility across webhook ingestion, integrations, LLM calls, and provisioning. PagerDuty alerts notify on-call teams of critical failures such as sustained API outages or elevated BLOCK rates.