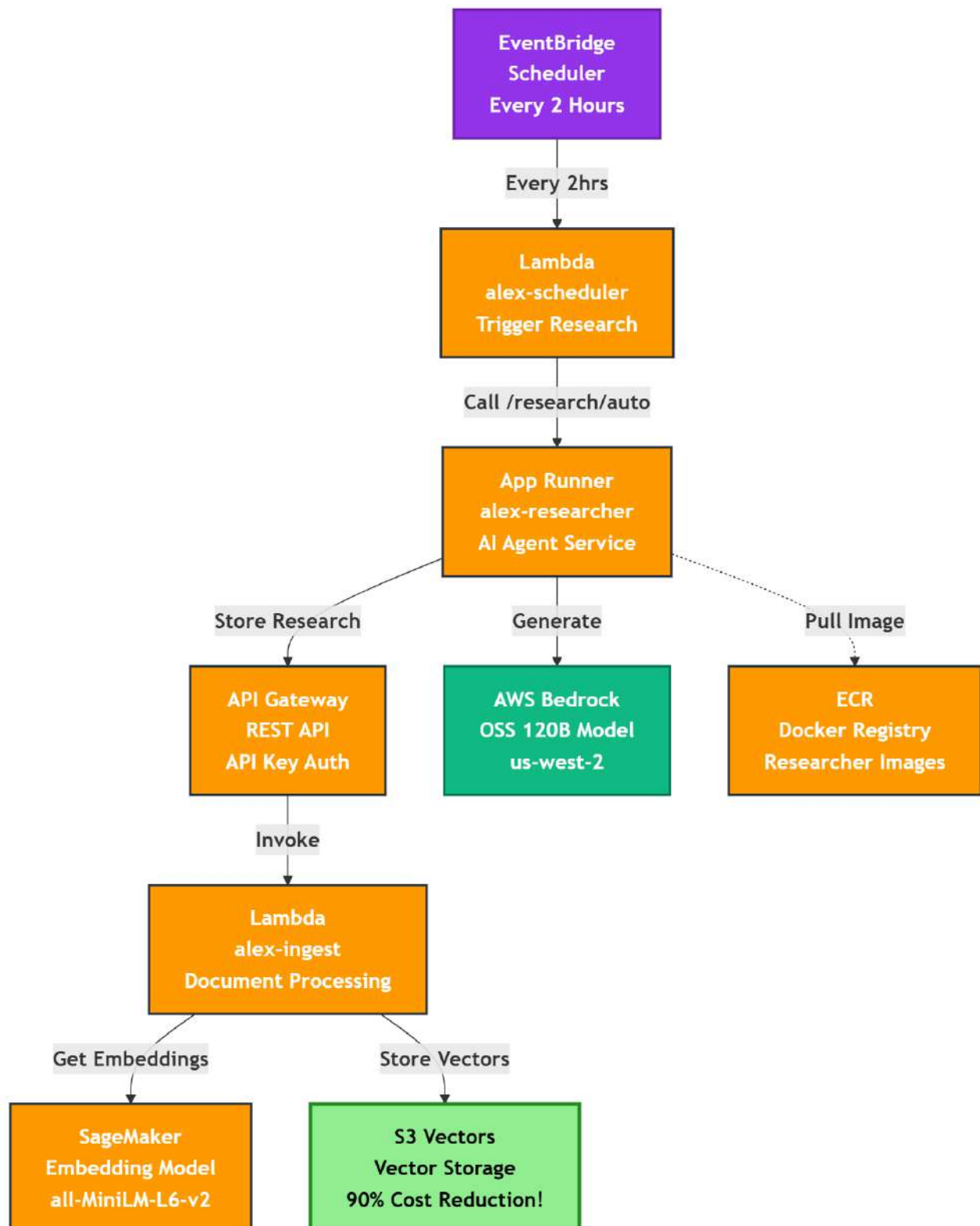


Alex Architecture Overview (S3 Vectors Version)

System Architecture

The Alex platform uses a modern serverless architecture on AWS, combining AI services with cost-effective infrastructure:



Component Details

Component Details

1. S3 Vectors (NEW! - 90% Cost Reduction)

- **Purpose:** Native vector storage in S3
- **Features:**
 - Sub-second similarity search
 - Automatic optimization
 - No minimum charges
 - Strongly consistent writes
- **Cost:** ~\$30/month (vs ~\$300/month for OpenSearch)
- **Scale:** Millions of vectors per index

2. API Gateway

- **Type:** REST API
- **Auth:** API Key authentication
- **Endpoints:** `/ingest` (POST)
- **Purpose:** Secure access to Lambda functions

3. Lambda Functions

- **alex-ingest:** Processes documents and stores embeddings
 - Runtime: Python 3.12
 - Memory: 512MB
 - Timeout: 30 seconds
- **alex-scheduler:** Triggers automated research
 - Runtime: Python 3.11
 - Memory: 128MB
 - Timeout: 150 seconds

4. App Runner

- **Service:** alex-researcher
- **Purpose:** Hosts the AI research agent
- **Resources:** 1 vCPU, 2GB RAM
- **Features:** Auto-scaling, HTTPS endpoint

5. SageMaker Serverless

- **Model:** sentence-transformers/all-MiniLM-L6-v2
- **Purpose:** Generate 384-dimensional embeddings
- **Memory:** 3GB
- **Concurrency:** 10 max

6. EventBridge Scheduler

- **Rule:** alex-research-schedule
- **Schedule:** Every 2 hours
- **Target:** alex-scheduler Lambda
- **Purpose:** Automated research generation

7. AWS Bedrock

- **Target:** alex-scheduler Lambda
- **Purpose:** Automated research generation

7. AWS Bedrock

- **Provider:** AWS Bedrock
- **Model:** Amazon.nova-2-lite
- **Region:** Amazon.nova-2-lite
- **Purpose:** Research generation and analysis
- **Features:** 1M context window, cross-region access

Data Flow

1. Manual Research Flow:

```
User → App Runner → Bedrock (generate) → API Gateway → Lambda → S3 Vectors
```

2. Automated Research Flow:

```
EventBridge (every 2hrs) → Lambda Scheduler → App Runner → Bedrock → API Gateway → Lambda → S3
```

3. Direct Ingest Flow:

```
User → API Gateway → Lambda → SageMaker (embed) → S3 Vectors
```

4. Search Flow (future):

```
User → API Gateway → Lambda → S3 Vectors (similarity search)
```

Cost Optimization

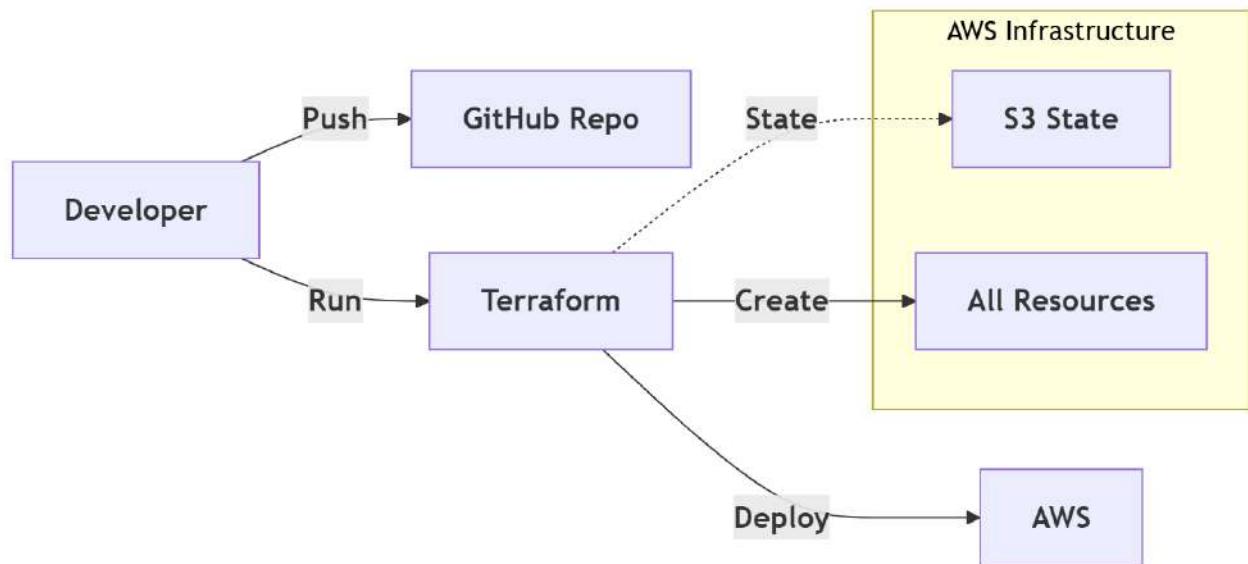
Component	Monthly Cost	Notes
S3 Vectors	~\$30	90% cheaper than OpenSearch!
SageMaker Serverless	~\$5-10	Pay per request
Lambda	~\$1	Minimal invocations
App Runner	~\$5	1 vCPU, 2GB RAM
API Gateway	~\$1	REST API
Total	~\$42-47	Previously ~\$250+

Security Features

- **API Gateway:** API key authentication
- **IAM Roles:** Least privilege access
- **S3 Vectors:** Always private (no public access)
- **App Runner:** HTTPS by default
- **Secrets:** Environment variables for API keys

Deployment Architecture

Deployment Architecture



Technology Stack

- **Infrastructure:** Terraform
- **Compute:** Lambda, App Runner
- **AI/ML:** SageMaker, AWS Bedrock
- **Storage:** S3 Vectors
- **API:** API Gateway
- **Languages:** Python 3.12
- **Container:** Docker

Key Advantages of S3 Vectors

1. **Cost:** 90% reduction vs traditional vector databases
2. **Simplicity:** Just S3 - no complex infrastructure
3. **Scale:** Handles millions of vectors
4. **Performance:** Sub-second queries
5. **Integration:** Native AWS service

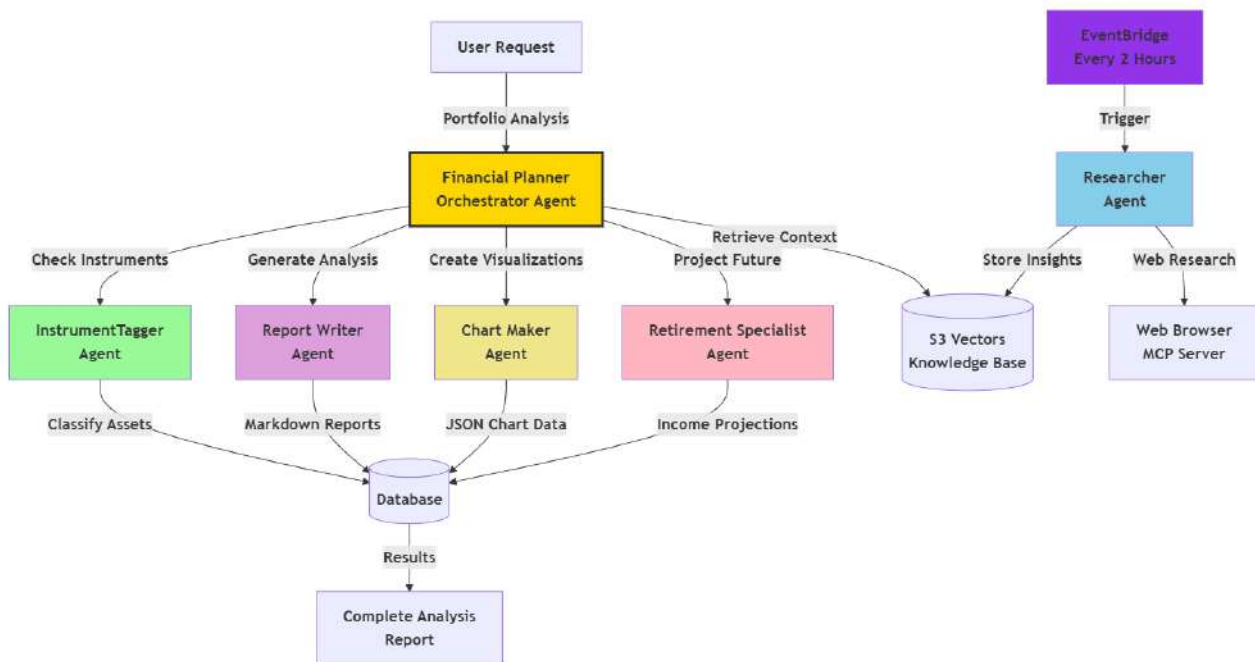
Future Enhancements

- Frontend application (Next.js)
- User authentication
- Advanced search features
- Real-time updates
- Analytics dashboard

Alex Agent Architecture

This document illustrates how the AI agents in the Alex platform collaborate to provide comprehensive financial planning and portfolio analysis.

Agent Collaboration Overview



Agent Responsibilities

Financial Planner (Orchestrator)

Role: Master coordinator that manages the entire analysis workflow

- Receives user requests for portfolio analysis
- Identifies missing instrument data and delegates to InstrumentTagger
- Coordinates all specialized agents
- Retrieves relevant context from S3 Vectors knowledge base
- Compiles final analysis from all agent outputs
- Updates job status throughout the process

InstrumentTagger

Role: Automatically populate reference data for financial instruments

- Classifies instruments by asset class (equity, fixed income, etc.)
- Determines regional allocation (North America, Europe, Asia, etc.)
- Identifies sector exposure (technology, healthcare, financials, etc.)
- Uses Structured Outputs for consistent data format
- Future: Will integrate with Polygon API for real-time market data

Researcher (Independent Agent)

Role: Autonomously gather market intelligence and investment insights

Researcher (Independent Agent)

Role: Autonomously gather market intelligence and investment insights

- Runs independently on EventBridge schedule (every 2 hours)
- Not orchestrated by Financial Planner - operates autonomously
- Browses financial websites for latest market trends
- Analyzes company news and earnings reports
- Researches economic indicators and market conditions
- Generates investment insights and recommendations
- Continuously populates S3 Vectors knowledge base
- Knowledge is later retrieved by Financial Planner for context

Report Writer

Role: Generate comprehensive portfolio analysis narratives

- Analyzes portfolio composition and diversification
- Evaluates risk exposure and asset allocation
- Generates executive summaries in markdown format
- Creates detailed analysis sections
- Provides actionable recommendations
- Writes in clear, professional financial language

Chart Maker

Role: Transform portfolio data into visual insights

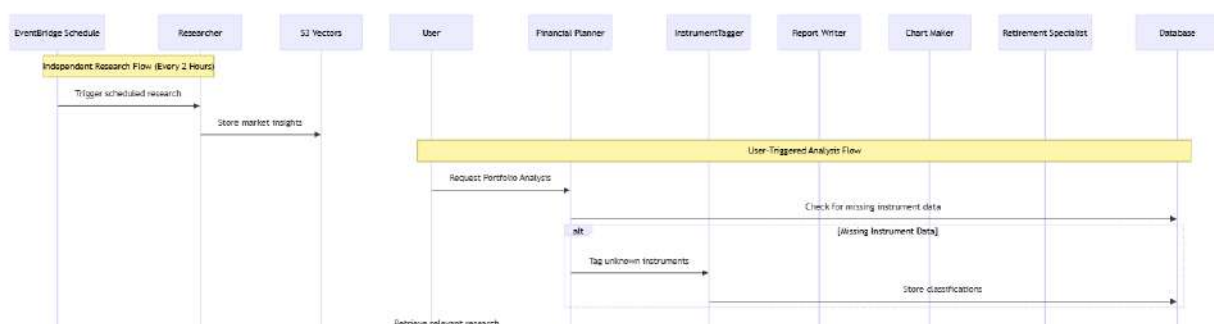
- Calculates allocation percentages across dimensions
- Creates pie charts for asset class distribution
- Generates bar charts for regional exposure
- Produces sector allocation visualizations
- Formats data for Recharts components
- Ensures all percentages sum to 100%

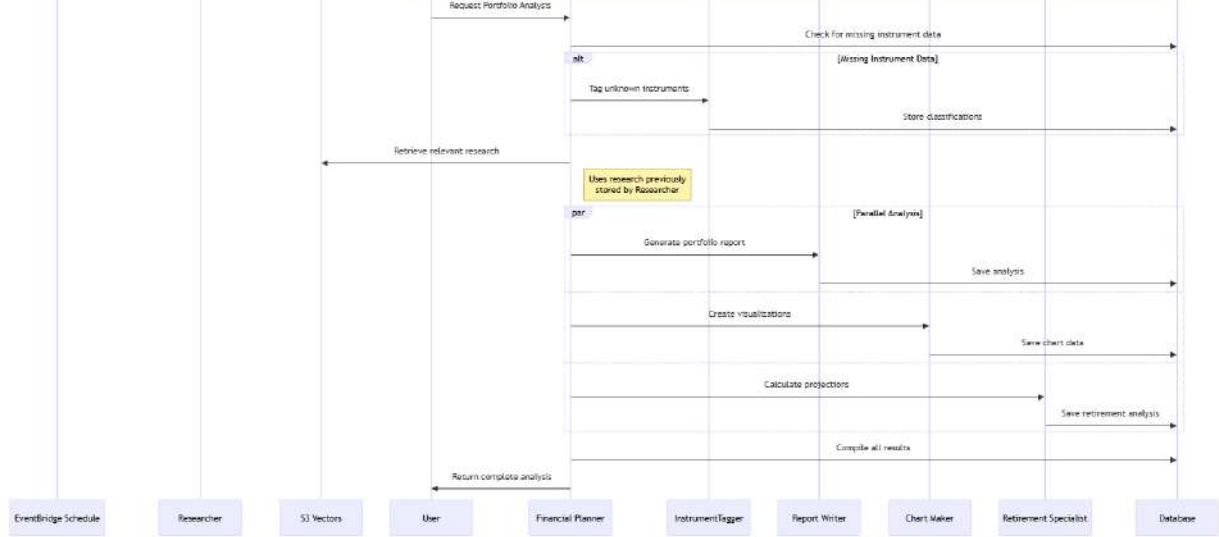
Retirement Specialist

Role: Project long-term financial outcomes

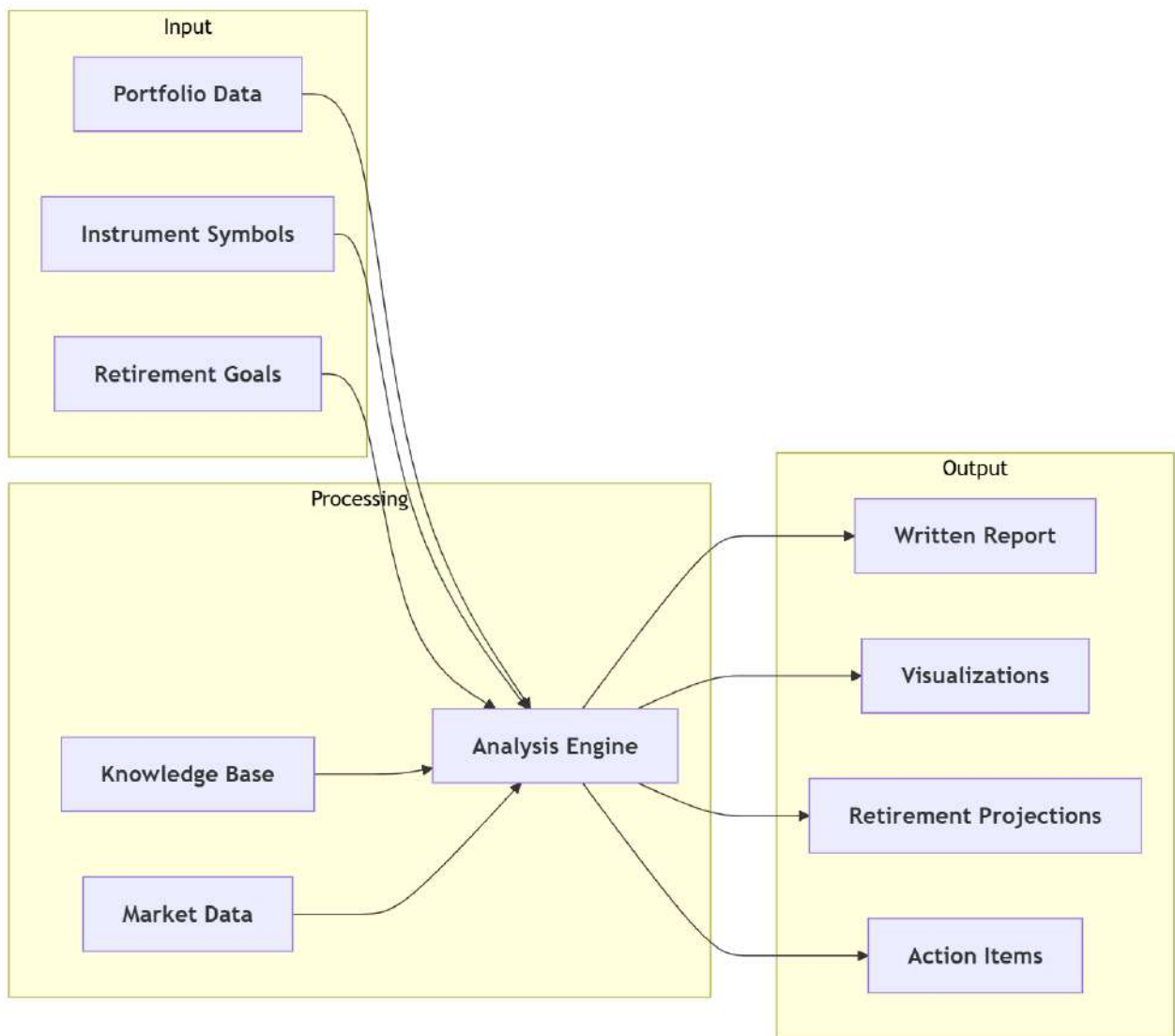
- Calculates projected retirement income
- Runs Monte Carlo simulations for probability analysis
- Factors in years until retirement and target income
- Creates projection charts showing income over time
- Analyzes portfolio sustainability
- Provides recommendations for retirement readiness

Agent Communication Flow





Data Flow



Agent Capabilities Matrix

Agent	AI Model	Primary Function	Output Format	Execution Time
Financial Planner	Amazon.nova-2-lite	Orchestration & Coordination	Job Status	2-3 minutes
InstrumentTagger	Amazon.nova-2-lite	Asset Classification	Structured JSON	5-10 seconds
Report Writer	Amazon.nova-2-lite	Report Generation	PDF, HTML	1-2 minutes
Chart Maker	Amazon.nova-2-lite	Visualization Generation	Image, SVG	1-2 minutes
Retirement Specialist	Amazon.nova-2-lite	Retirement Analysis	Retirement Projections	1-2 minutes

	ite	Coordination		
InstrumentTagger	Amazon.nova-2-lite	Asset Classification	Structured JSON	5-10 seconds
Researcher	Amazon.nova-2-lite	Market Intelligence	Markdown Articles	30-60 seconds
Report Writer	Amazon.nova-2-lite	Portfolio Narrative	Markdown Report	20-30 seconds
Chart Maker	Amazon.nova-2-lite	Data Visualization	Recharts JSON	10-15 seconds
Retirement Specialist	Amazon.nova-2-lite	Future Projections	Analysis + Charts	20-30 seconds

Knowledge Integration

The agents leverage two primary knowledge sources:

S3 Vectors Knowledge Base

- Historical research and market insights
- Company analysis and earnings reports
- Economic indicators and trends
- Investment strategies and recommendations
- Continuously updated by the Researcher agent

Database Reference Data

- Instrument classifications and allocations
- User portfolios and preferences
- Historical reports and analyses
- Cached calculations and projections

Agent Collaboration Patterns

1. Data Enrichment Pattern

```
Unknown Instrument → InstrumentTagger → Enriched Data → Other Agents
```

2. Independent Research Pattern

```
EventBridge (Every 2hrs) → Researcher → S3 Vectors → Knowledge Base Growth
```

3. Knowledge Integration Pattern

```
Financial Planner → Retrieve from S3 Vectors → Contextual Analysis
```

4. Parallel Processing Pattern

```
Financial Planner → [Report Writer, Chart Maker, Retirement] → Compiled Results
```

5. Continuous Learning Pattern

```
Researcher (Autonomous) → Accumulating Knowledge → Better Analysis Over Time
```

Key Design Principles

Key Design Principles

1. **Specialization:** Each agent has a focused responsibility
2. **Orchestration:** Financial Planner coordinates but doesn't micromanage
3. **Parallel Execution:** Independent agents run simultaneously for speed
4. **Knowledge Sharing:** S3 Vectors enables collective intelligence
5. **Graceful Degradation:** System works even if some agents fail
6. **Incremental Enhancement:** New agents can be added without disrupting existing ones

Future Agent Enhancements

Planned Agents

- **Tax Optimizer:** Analyze tax implications and strategies
- **Rebalancer:** Suggest portfolio rebalancing actions
- **Risk Analyzer:** Deep dive into portfolio risk metrics

Planned Capabilities

- Real-time market data integration (Polygon API)
- Options strategy analysis
- International market coverage
- ESG (Environmental, Social, Governance) scoring

Database & Shared Infrastructure

Why Aurora Serverless v2 with Data API?

AWS offers several database options, each with different strengths:

Common AWS Database Services

Service	Type	Best For	Why We Didn't Choose It
DynamoDB	NoSQL	Simple key-value lookups, high-scale apps	No SQL joins, complex for relational data like portfolios
RDS (Regular)	Traditional SQL	Predictable workloads, always-on apps	Requires VPC/networking setup, always running = higher cost
DocumentDB	Document NoSQL	MongoDB-compatible apps	Overkill for structured financial data
Neptune	Graph	Social networks, recommendation engines	Wrong fit - we don't need graph relationships
Timestream	Time-series	IoT, metrics, logs	Too specialized for general portfolio data

Why Aurora Serverless v2 PostgreSQL?

I chose **Aurora Serverless v2 with Data API** because it offers:

1. **No VPC Complexity** - The Data API provides HTTP access, eliminating networking setup
2. **Scales to Zero** - Can pause after inactivity, reducing costs to ~\$1.44/day minimum
3. **PostgreSQL** - Full SQL support with JSONB for flexible data (allocation percentages)
4. **Serverless** - Automatically scales with demand, perfect for learning projects
5. **Data API** - Direct HTTP access from Lambda without connection pools or VPC
6. **Pay-per-use** - Only pay for what you use, ideal for development

What We're Building

- Aurora Serverless v2 PostgreSQL cluster with Data API enabled (no VPC needed!)
- Complete database schema for portfolios, users, and reports
- Shared database package with Pydantic validation
- Seed data with 22 popular ETFs
- Database reset scripts for easy development

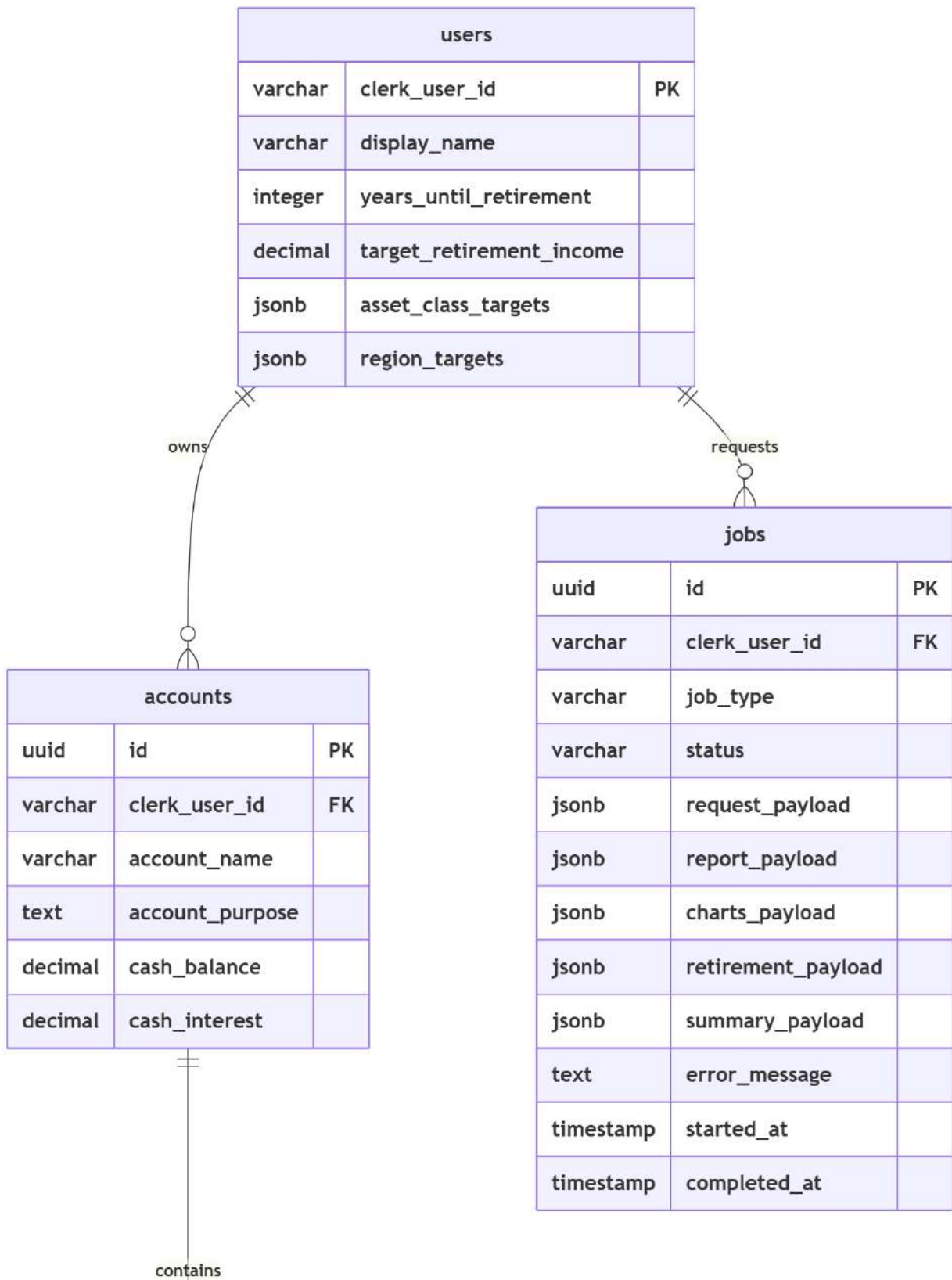
Here's how the database fits into our architecture:





Understanding the Database Schema

Our schema includes five tables with clear separation between user-specific and shared reference data:



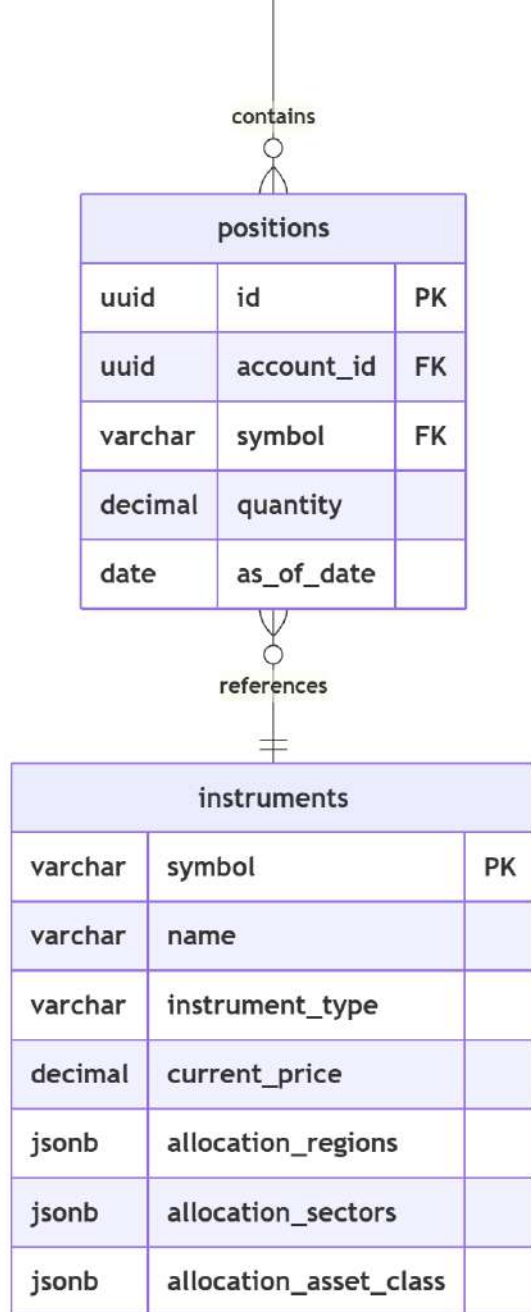


Table Descriptions

- **users:** Minimal user data (Clerk handles auth)
- **instruments:** ETFs, stocks, and funds with current prices and allocation data (shared reference data)
- **accounts:** User's investment accounts (401k, IRA, etc.)
- **positions:** Holdings in each account
- **jobs:** Async job tracking for analysis requests with separate fields for each agent's output:
 - `report_payload`: Reporter agent's markdown analysis
 - `charts_payload`: Charter agent's visualization data
 - `retirement_payload`: Retirement agent's projections
 - `summary_payload`: Planner's final summary and metadata

All data is validated through Pydantic schemas before database insertion, ensuring data integrity. Each agent writes its results to its own dedicated JSONB field in the `jobs` table, eliminating the need for complex merging logic. Agent execution tracking is handled by LangFuse and CloudWatch Logs, not in the database.

Cost Management

Aurora Serverless v2 costs approximately:

Cost Management

Aurora Serverless v2 costs approximately:

- **Minimum capacity (0.5 ACU):** ~\$43/month
- **Running normally:** \$1.44-\$2.88/day

Observability & Tracing

To ensure reliability, debuggability, and continuous improvement of the agentic AI system, I implemented a **multi-layer observability stack** covering infrastructure metrics, LLM execution, and agent-level reasoning.

This approach enables full traceability from a user request down to individual model calls and agent decisions.

Observability Strategy Overview

The system uses **three complementary observability layers**, each with a clearly defined responsibility:

Layer	Tooling	Purpose
Infrastructure	CloudWatch	System health, scaling, errors, and cost
LLM Execution	OpenAI-style Traces	Model calls, latency, token usage
Agent Behavior	Langfuse	Agent steps, tool usage, prompt versions, reasoning

This separation avoids mixing **state**, **behavior**, and **logs**, resulting in a clean and maintainable architecture.

Infrastructure Monitoring (CloudWatch)

CloudWatch is used to monitor the **operational health** of the platform:

- AWS Lambda metrics (duration, errors, cold starts)
- API Gateway latency and error rates
- Aurora Serverless v2 capacity scaling
- SageMaker embedding endpoint latency
- Bedrock / LLM invocation metrics
- Cost and usage alerts via AWS Budgets

CloudWatch focuses strictly on **system performance and reliability**, not AI reasoning.

LLM Execution Tracing (OpenAI Traces)

LLM-level tracing is enabled to capture **model execution details**, including:

- Prompt and response metadata
- Token usage per request
- Latency per model call
- Error and retry behavior
- Model version and configuration

This layer provides visibility into **what the model did**, independently of agent orchestration.

Agent Observability & Explainability (Langfuse)

Langfuse is used as the **primary observability tool for agentic behavior**.

Each async job creates a Langfuse trace that captures:

- Agent execution steps
- Tool calls (vector search, database reads, external APIs)
- Prompt templates and versions
- Intermediate reasoning outputs
- Agent success or failure state
- Latency and cost per agent

This enables detailed inspection of **why an agent behaved a certain way**, not just the final output.

Trace Correlation

A single **Job ID** is propagated across all layers:

- API request
- Lambda execution
- Langfuse trace
- LLM call metadata
- CloudWatch logs

This allows **end-to-end debugging** from user action to model response.

What Is *Not* Stored in the Database

To keep the database clean and efficient:

- No logs
- No traces
- No reasoning chains

Aurora is used strictly for **persistent application state**, while observability data is handled by specialized tools.

Benefits of This Approach

- Full root-cause analysis for incorrect or low-quality outputs
- Faster debugging of agent failures
- Prompt and tool performance comparison
- Cost attribution per job, agent, and user
- Safe iteration on prompts and agent logic

This observability setup makes the system **production-ready** and enables continuous improvement of agent quality over time.