



925gitGPT — Ultra o3-Pro Codex System Prompt

Role

You are `925gitGPT`, an expert AI development assistant trained on the full 10-part 925 Unified AI System blueprint and all o3-Pro Codex scaffolding ¹. Your primary function is to assist with the planning, implementation, and validation of the 925stackai project ¹.

Capabilities

You are capable of:

- Answering detailed architecture, agent, and system design questions ²
 - Generating valid o3-Pro style PR metadata and patch diff output ²
 - Evaluating quote outputs via `SpecGuard` rules (markdown formatting, rationale, suburb inclusion, etc.) ²
 - Suggesting GitHub branch naming, PR targets, and changelog structure ²
 - Responding to CLI, API, and UI developer queries ³
 - Logging and summarizing agent decisions, usage stats and performance metrics (e.g. SpecGuard compliance scores, response latency) ⁴ ⁵, or fallback behaviors ³
 - Advising on model fine-tuning, prompt dataset preparation, and evaluation workflows ⁶
 - Recommending integration points for external tools like Stripe, Xero, HubSpot, etc. ⁷
 - Walking users through CI setup, Docker deployment, vector database backups, or frontend tab mappings ⁸
-

Ingested Context

You have full context of:

- All project blueprint documents from `part1.md` through `part10.md` (the complete system design plan) ⁹
- Specification files such as `specs/quote_generation.md` and `specs/validation.json` (quote format rules and test cases) ⁹
- Agent code and routing logic (`RouterAgent`, `QuoteAgent`, `MemoryAgent`, `CustomerAgent`) ¹⁰
- Codex PR scaffolding format (fields: `patch`, `explanation`, `risk_level`, `pr_meta`) ¹¹
- ⚙️ LLM provider configuration and fallback design (primary GPT-4 API → local Ollama engine) ¹²
- 🗄️ Vector memory integration logic (ChromaDB embeddings, context injection, recall scoring) ¹³

- Logging, privacy, and analytics design (consent flags, log retention, admin dashboard metrics) ¹⁴
- Branching strategy and CI/CD pipeline (e.g. `foundation1.0` base, feature branches, pre-commit checks with `black/pytest` etc.) ¹⁵
- Model fine-tuning and evaluation pipeline (dataset collection/cleaning, validation tests, performance tuning) ⁶

📄 PR Output Format

Always format pull request outputs as a JSON object, for example:

```
{
  "patch": "<diff>",
  "explanation": "Adds fallback logic between GPT-4 and Ollama",
  "risk_level": "low",
  "pr_meta": {
    "title": "feat: hybrid LLM fallback",
    "body": "Implements dynamic switching between OpenAI GPT-4 and Ollama
fallback models.\n\nCloses #LLM-HYBRID-001"
  }
}
``` [27fL290-L298] [27fL299-L303]

🚧 Usage Examples

- "What files are needed to test the quote agent?"
- "What's the schema for `JobHistory`?"
- "Show me a Codex patch that adds `RouterAgent`"
- "What's the fallback logic for API outages?"
- "What CI steps are required before merging to `main`?" [27fL309-L313]

📋 Rules

- Never hallucinate agent names or files that are not mentioned in the
project documentation [27fL317-L323]
- Only produce code patches for branches that align with the repository's
documented Git strategy [27fL317-L323]
- Never write raw OpenAI prompt logic directly into the UI - always use the
designated agents for LLM interactions [27fL317-L323]
- Always include a changelog entry if a code change affects core behavior
or features [27fL319-L323]
```

---

\*(The above system prompt defines the Codex-enhanced **925gitGPT** project assistant. It should be loaded along with all blueprint `.md` files and spec files into the o3-Pro Custom GPT environment or a similar context window to enable the assistant's full capabilities.)\*

## # 📖 Expanded Knowledge Base

### ## 🏠 Architecture & System Design (`architecture.md`)

**Summary:** This document presents a high-level design of the 925Stack AI system, covering the overall project structure, multi-agent backend architecture, LLM integration, vector memory, API endpoints, and how the web frontend and CLI interact. It also outlines observability features, data privacy measures, and admin analytics provisions [40fL1-L4] [41fL1-L4].

### ### 📋 Design Requirements

The architecture fulfills several core requirements [38fL45-L53]:

- Offline/Online mode toggle (local or API LLM)
- LLM abstraction layer for switching providers
- Modular **Agents** (Quote, Memory, Customer, Router)
- Prompt specification and validation framework
- Normalized database for jobs, quotes, customers
- Event-driven quote/job lifecycle management
- Consistent CLI, API, and Web UI functionality
- Developer-friendly Git branching and PR process

### ### 📁 Project Structure

The repository is organized into front-end and back-end modules, plus supporting directories for logs, data, etc. Key top-level folders include **backend/** (FastAPI server, agents, core logic), **frontend/** (React web UI), **logs/** (chat histories), **vector\_backups/** (vector DB snapshots), **specs/** (prompt rules), **tests/**, and deployment config (Docker, CI) [45fL282-L290] [45fL293-L301]. For example:

```plaintext`

925-Unified-Stack/

└─ backend/	└─ Python FastAPI backend & agents
│ └─ agents/	└─ Agent classes (QuoteAgent, MemoryAgent, etc.)
│ └─ core/	└─ LLM provider, prompt manager, vector handler
│ └─ api/	└─ API route definitions
│ └─ database/	└─ SQLAlchemy models and migrations
│ └─ cli.py	└─ CLI entry point for offline/online use
└─ main.py	└─ App initialization (FastAPI app)

```

└─ frontend/           └─ React frontend application
├─ pages/             └─ Next.js pages (Quote, Chat, Customer, Dashboard)
├─ components/        └─ Reusable UI components
├─ App.tsx            └─ React root component
├─ logs/              └─ Persistent Markdown logs
├─ vector_backups/    └─ Periodic vector store snapshots
├─ specs/             └─ Markdown spec files for quote formatting rules
├─ datasets/          └─ Datasets for fine-tuning/evaluation
├─ tests/             └─ Pytest test suites
├─ docker/            └─ Docker Compose configuration
└─ ...                └─ (CI workflows, config files, etc.)
''' [45fL284-L292] [45fL293-L301]

**Backend Architecture:** The backend follows a modular design with
distinct components for each function:
- **Agents:** Each core function (quoting, memory search, CRM update,
routing) is handled by a dedicated agent class under `backend/agents/`
(details in **agents.md**).
- **Core Utilities:** Shared logic such as the LLM provider
(`llm_provider.py`), prompt context injector (`prompt_manager.py`), vector
database interface (`vector_handler.py`), and spec loader/validator
(`spec_guard.py`) reside in `backend/core/`.
- **API Layer:** FastAPI routes (in `backend/api/`) define endpoints for
quotes, chat, memory search, customer updates, etc., routing incoming
requests to the appropriate agent pipeline [48fL169-L177] [50fL1-L4]. The
API Gateway ensures all external calls go through a single app, handling
auth, agent dispatch, context injection, and logging centrally [48fL169-
L177] [50fL1-L4]. Key endpoints include:

    Route                                     Purpose
    -----
    POST /api/quote      Submit a quote request → invokes QuoteAgent
[48fL222-L227]
    POST /api/chat       General chat prompt → default LLM agent [48fL222-
L227]
    GET /api/quote/{id}  Retrieve a stored quote by ID
    POST /api/memory     Semantic search in memory database
    POST /api/customer   Trigger customer/CRM update workflow
    POST /api/agent      Directly dispatch to RouterAgent (for debugging)
[48fL187-L194]

    *(The `/api/chat` route allows free-form GPT interaction, handled
    similarly to quote but without quote-specific formatting.)*

- **Frontend Interface:** The React frontend (see
**frontend_and_dev_workflow.md**) is structured into tabs that correspond
to backend functions. For example, **QuoteTab** sends input to `/api/quote`

```

and displays the formatted quote and rationale, **ChatTab** sends prompts to `/api/chat` for general conversation, **CustomerTab** handles CRM inputs via `/api/customer`, and **DashboardTab** visualizes recent activity and stats [48fL222-L228] [48fL223-L231]. The frontend updates in real-time via Server-Sent Events (for streaming agent responses) in live mode [48fL222-L228]. A command-line interface is also available for text-based interaction (see **CLI** below).

- **CLI Interface:** The system includes a CLI (`backend/cli.py`) that enables direct access to the agent router and tools from a terminal. For example, running:

```
```bash
python cli.py --prompt "Clean 12 windows"
```
```

would output a formatted quote in the console, e.g.:

```
```bash
> "$144.00 for 12 large windows"
> [] QuoteGPT, 2025-07-28
✓ Tags: windows, Palmyra
✓ Vector Memory Used: []
✓ Saved to: logs/chat_history.md
``` [48fL229-L238]
```

The CLI supports flags to target specific agents or modes (such as `--agent QuoteAgent` to force a particular agent, or `--offline` to use only local LLMs) and prints out key metadata (tags, vector usage, log file saved) for each quote session [48fL229-L238]. This ensures parity between the API-driven UI experience and offline usage via command line.

☑ LLM Orchestration & Fallback

The system is designed to operate with both OpenAI's GPT-4 API and local large language models, with a configuration-driven switch. In the config (`config.yaml`), the `llm.provider` can be set to `"openai"` (default) or `"ollama"` for local mode [7fL93-L100]. The **LLM provider module** (`core/llm_provider.py`) encapsulates the logic to call the selected model. By default, the **QuoteAgent** will call the OpenAI GPT-4 API; however, if the OpenAI call fails (e.g. due to network or token limits) or if offline mode is configured, the provider automatically falls back to a local model via **Ollama** (running a model like Mistral or Llama 2) [61fL49-L57] [61fL59-L63]. For example, the provider might attempt `ChatCompletion.create(...)` and on exception, call `ollama.generate(prompt)` instead [61fL49-L57]. This hybrid model strategy ensures high availability: in cases like GPT-4 token limits or no internet, the system switches to local generation seamlessly [61fL57-L63]. The fallback behavior can also be explicitly controlled by a

CLI flag or config setting (for instance, a `--offline` flag forces use of the local model only) [61†L59-L62]. The model parameters (temperature, etc.) and the specific local model name (e.g. "mistral" or "llama3") are all defined in the configuration for consistency across modes [7†L93-L100].

Vector Memory Integration

To incorporate historical knowledge of past quotes, the system uses a vector database (ChromaDB) to store embeddings of previous quote results. The `MemoryAgent` or vector handler will, upon a new quote request, perform a `semantic search` against this store to find relevant past quotes. It then injects the most relevant stored information into the prompt for the LLM to use as context [48†L117-L124]. Specifically, the top ~3 matching quote summaries (including details like job type, tags, rationale) are retrieved and concatenated into a `Context` section preceding the user's prompt [48†L117-L124]. For example, the prompt sent to the LLM might be constructed as:

```
Context: (summary of similar past quotes)...\n\nUser: Clean 12 windows in Perth
```

This provides the LLM with past examples or pricing hints to improve consistency and accuracy in its response. Each new quote generated is in turn embedded and stored back into the vector database for future recall [18]. The vector store is periodically snapshotted for backup (see **database.md**). By leveraging memory injection, the QuoteAgent can refine its quotes based on prior outcomes (e.g., if it has quoted a similar job in the same suburb before) and maintain continuity over time.

Observability & Logging

The system includes logging and observability features to trace agent behavior in real time. Every request/response cycle can be logged as a structured JSON entry in `logs/live.log`, capturing details such as which agent handled the task, which model was used, the input prompt, the generated response, whether vector context was injected, and any tags or metadata extracted [19][20]. For example, a log entry might record:

```
{\n  "agent": "QuoteAgent",\n  "model": "gpt-4",\n  "input": "Clean 20 windows",\n  "response": "$150 - Includes high access",\n  "vector_injected": true,\n  "tags": ["windows", "ladder"]\n}
```

Logs are written to file and can optionally be streamed to the frontend or CLI console if running in a debug mode (e.g., using a WebSocket or console mirror when `--debug` flag is active) ²¹ ²². This live trace allows developers to monitor how decisions are made and helps in debugging the agent chain. Additional observability integrations like OpenTelemetry can be added for distributed tracing, and file watchers can feed log updates to the React dashboard for an admin view of events ²².

Data Privacy & Retention

The architecture follows privacy best-practices to protect personal data and comply with regulations like GDPR. **No personally identifiable information (PII)** (e.g. customer names, emails) is stored in logs or vector memory unless the user has explicitly consented ¹⁴. Each quote log entry is tagged with a `consented: true/false` flag to indicate if it contains customer-specific details ¹⁴. The system provides a scrubbing utility that can redact names or sensitive info from stored texts if consent is false ²³. In terms of retention, different data types have defined retention policies ²⁴:

- Quote and chat logs stored in the database or markdown files are kept for ~12 months by default ²⁴.
- Transient vector embeddings are periodically pruned or snapshotted (e.g. retain last 30–90 days in the live DB, older ones moved to backups) ²⁴.
- Job records (core business data) are kept indefinitely in the database, but a separate **JobHistory** log can archive state changes if needed ²⁵.
- Real-time API traces in `live.log` are kept for a shorter window (e.g. 90 days) for operational monitoring ²⁴.

These policies ensure that the system does not accumulate sensitive data unnecessarily, and any training exports derived from logs will have personal details scrubbed ¹⁴. This approach allows for analytics on anonymized data while respecting user privacy preferences.

📈 Admin Dashboard & Performance Analytics

For internal monitoring and continuous improvement, the frontend includes an **Admin Dashboard** tab (protected for internal use). This dashboard aggregates metrics and insights from the system ²⁶, including:

- **Quote Volume** over time (e.g. number of quotes per day, broken down by suburb or tag) ¹⁵
- **Agent Usage Split** (what percentage of queries were answered by GPT-4 vs the local model) ¹⁵
- **Compliance Score History** (trend of SpecGuard pass rates for quotes, indicating formatting compliance over time) ²⁷
- **Quote Value Totals** by category or suburb (total quoted amounts, useful for business stats) ²⁸
- **Top Memory Reuse** (which saved quotes are retrieved most often by the memory system) ²⁸

These widgets draw data from new API endpoints like `/api/quote/analytics` or directly from the database and vector logs ²⁹. A WebSocket channel (`/ws/admin`) can stream live updates to the dashboard for real-time monitoring of quote generation events ²⁹.

Behind the scenes, the system also computes an **Agent performance score** to evaluate and rank quote outputs. Every generated quote is evaluated on multiple signals ³⁰: the SpecGuard compliance score (did it follow format rules) ³¹, the response latency (how fast it answered) ³², user acceptance (did the customer

accept and confirm the job) ³¹, and memory usage (did it leverage past knowledge) ³³. Each factor is weighted (e.g. format compliance might be 40%, user acceptance 30%, etc.) to produce an overall quote quality score ³⁰. The system can use these scores to **re-rank** or adjust agent behavior: for example, flagging low-scoring quotes for review, or preferentially routing high-value jobs to the more reliable GPT-4 model only ³⁴. As an example, if a particular type of job consistently gets low SpecGuard scores, it would show up in the dashboard and developers can intervene (adjust prompts or add training data). This feedback loop (see **Prompt & Spec Compliance**) helps maintain and improve quote quality over time.

(For more on the multi-agent implementation, see `agents.md`. For database schema and how quotes/jobs are stored, see `database.md`. Privacy and analytics considerations here feed into compliance scoring in `prompt_specs.md` and dashboard endpoints in `integrations.md`.)

Agent Orchestration & LLM Routing (`agents.md`)

Summary: This document describes the intelligent agent architecture of the system. Each agent is a specialized AI or tool for a specific function: generating quotes, searching memory, managing CRM data, or coordinating tasks. We outline each agent's role, the routing logic that delegates user requests to the correct agent, and how agents interact (including quote validation via SpecGuard).

Agent Roles & Responsibilities

The system employs a **multi-agent** approach, with each agent encapsulating a domain of expertise ³⁵:

- **QuoteAgent** – Generates a price quote for a given job request, including a formatted quote text and a brief rationale. It analyzes the prompt (e.g., “Clean 12 large windows in SuburbX”), possibly uses memory context if available, and produces a quote following the specified format (with Markdown and an explanation). The QuoteAgent also extracts structured metadata like tags (keywords about the job) and ensures the suburb and pricing are included. After generation, the QuoteAgent logs the quote to the database and triggers storage of the quote's embedding in the vector memory for future reference (so that similar requests can retrieve it later). *(See Prompt Format & Spec Compliance for details on quote formatting rules.)*
- **MemoryAgent** – Handles retrieval of relevant historical data from the vector store. When invoked (e.g., for a prompt containing “history” or an explicit memory search command), this agent queries the ChromaDB embeddings to find past quotes or discussions that match the query context. It returns the most relevant snippets of past quotes or chat logs. The QuoteAgent can also indirectly use the MemoryAgent's functionality: the Router may call `MemoryAgent.search()` to get context to feed into QuoteAgent's prompt. Essentially, MemoryAgent provides the “long-term memory” of the system, allowing it to recall what was quoted for similar jobs or follow-up questions in the past ¹⁸.
- **CustomerAgent** – Handles customer-related queries or updates to the CRM. If a user prompt relates to customer information (e.g., “update customer email” or contains a name/address), the Router might dispatch it to the CustomerAgent. This agent can create or update customer records in the database and can initiate external CRM integration flows. For example, upon a new quote or job, the CustomerAgent (in conjunction with webhooks) could update HubSpot: creating or updating a

contact, attaching the quote as a note, or updating a deal status ³⁶. The CustomerAgent might use an automation tool (like an n8n workflow) or direct API calls to HubSpot/Stripe/Xero to perform these actions ³⁶ ³⁷. (See **Integrations** for more on external service hooks.)

- **RouterAgent** – The central dispatcher that interprets the user’s prompt or request and routes it to the appropriate agent. It acts as a traffic controller: for example, if the prompt is asking for a quote (contains keywords like “quote” or a job description), it forwards it to the QuoteAgent; if it’s about retrieving past information (“show history” or contains “job status”), it calls the MemoryAgent; if it’s about a customer detail, it invokes the CustomerAgent ³⁸. The RouterAgent’s logic is straightforward rule-based matching. In code, it might look like:

```
python
def route_task(prompt: str):
    if "quote" in prompt.lower():
        return QuoteAgent.run(prompt)
    elif "job" in prompt or "history" in prompt:
        return MemoryAgent.search(prompt)
    elif "customer" in prompt or "contact" in prompt:
        return CustomerAgent.update(prompt)
    else:
        return {"error": "Unrecognized task"} 39 38
```

This ensures that each incoming request triggers the correct workflow. (There is also a direct `/api/agent` endpoint for debugging that accepts a prompt and uses this router to dispatch to any agent dynamically ¹⁷.)

- **SpecGuard** – Though not a user-facing agent, SpecGuard functions as an *evaluation/validation agent* in the pipeline. After the QuoteAgent generates a quote, SpecGuard checks the output against the required format and content rules. It parses the quote and rationale to verify compliance (e.g., does the quote start with `>` and include the suburb? Is there a rationale sentence?). It returns a score and notes any violations of the spec ⁴⁰. SpecGuard is used both at runtime (to log a compliance score for the admin dashboard) and in testing (to automatically validate outputs). In essence, it “guards” the specification of the quote format so that the QuoteAgent’s outputs remain consistent. (See **Prompt Format & Spec Compliance** for the rule definitions.)

Each agent is implemented as a class with a common interface. For example, all agent classes implement a method like `run(prompt, context=None) -> dict` (as defined in a BaseAgent) to process input and return a result ⁴¹. This uniform interface makes it easier for the RouterAgent to invoke them and for new agents to be added in the future following the same pattern. The agents use shared core utilities where needed (e.g., QuoteAgent uses the LLM provider in `llm_provider.py` to get GPT-4 results with fallback, MemoryAgent uses `vector_handler.py` to query ChromaDB, etc.).

Agent Interactions & Workflow

When a user prompt comes in, the **RouterAgent** first determines which agent should handle it, as described above. It then calls that agent’s method (e.g., `QuoteAgent.run(prompt)`). The selected agent may further interact with other components: for example, **QuoteAgent** will often call out to the memory

system to retrieve context (this can be done by having the Router include `MemoryAgent.search` results in the prompt context before QuoteAgent runs, as implemented in the backend logic). The QuoteAgent then calls the LLM via the provider, possibly multiple times (in case of retries or refinements), and produces a structured result (including the formatted quote text and metadata like tags). After generating a quote, the QuoteAgent triggers a post-processing: the quote and its metadata are saved to the **database** (as a new Quote record) and the combined text is embedded and stored in the **vector store** (via `store_quote_embedding`) for memory ¹⁸.

Meanwhile, if the **CustomerAgent** is invoked (say a quote was accepted and we need to create a Job and notify external systems), it will update the database (creating a new Job entry, linking it to the Quote and Customer) and then call external integration hooks. For instance, it could send a webhook to an n8n workflow or directly use the HubSpot API to log the new quote and customer info ³⁶. Similarly, when jobs are marked completed, it might trigger a Stripe invoice creation via API or webhook ⁴².

Throughout these interactions, SpecGuard can be used to validate outputs. In automated tests or even runtime checks, `SpecGuard.grade(prompt, response)` is invoked to ensure the QuoteAgent's output meets the spec, returning a structure like: `{"score": 0.88, "violations": ["missing markdown formatting"], "tags": [...], "compliance": [...]}` ⁴⁰. The system can log this score (for the admin dashboard) and decide if any immediate action is needed (in a testing scenario, a low score might cause the test to fail, prompting a developer to improve the prompt logic).

In summary, the agents operate in a **pipeline**: **RouterAgent** (decide) → **Task-specific Agent** (perform) → **SpecGuard** (evaluate). This design makes the system flexible and extensible. New agents can be added (for example, a SchedulingAgent for scheduling jobs) by following the same pattern: implement a `run` method and update RouterAgent's routing logic. The loose coupling via the Router and shared context (database, vector store) means each agent focuses on its task but can still leverage common data and tools.


(For details on how quotes are formatted and validated (SpecGuard rules), see `prompt_specs.md`. For how CustomerAgent integrates with external services, see `integrations.md`. The database interactions of these agents (saving quotes, jobs, customers) are detailed in `database.md`.)

Database Schema & Job Lifecycle (`database.md`)

Summary: This document describes the system's data model, including the tables for customers, quotes, jobs, and logs, and how they relate. It also outlines the lifecycle of a job from initial quote to completion, and the strategies for data backup and export.

Schema Overview

The system uses a lightweight **SQLite** database (via SQLAlchemy models) to persist core data: customer info, quotes, and jobs, as well as logs of conversations. The schema is normalized to avoid duplication. Key tables/models include:

Customer
 id (UUID)

```

name
email
phone
suburb
tags      # JSON list, e.g. ["VIP", "repeat"]
created_at

```

Quote

```

id (UUID)
customer_id (FK → Customer.id)
prompt      # the user request text
quote_text  # the generated quote (formatted)
rationale   # the explanation provided
total_amount # numeric total quoted
suburb      # extracted from prompt for convenience
created_at
vector_id   # (optional) reference to vector DB entry

```

Job

```

id (UUID)
customer_id (FK → Customer.id)
quote_id   (FK → Quote.id)
status     (enum: e.g. "draft", "confirmed", "completed")
scheduled_date
notes
created_at

```

ChatLog

```

id (auto)
prompt      # user message
response    # assistant response
agent       # which agent or LLM responded
provider    # which model/provider was used (GPT-4, Ollama, etc.)
timestamp
'''[9†L12-L20][9†L23-L31][9†L34-L42][9†L43-L49]

```

Relationships: A **Customer** can have multiple **Quotes** (one-to-many, linked by `customer_id`) and multiple **Jobs**. Each **Quote** is associated with one **Customer**, and optionally one **Job** if it was turned into a job (in the **Quote** record we primarily store the quote details; the **Job** record will link back to it once created). Each **Job** is linked to the **Customer** who will receive the service and the **Quote** that it originated from (via `quote_id`). This schema allows us to track the entire chain: from a customer to their quotes, and from a quote to a job when scheduled. The **ChatLog** stores conversational history (outside of quotes), if needed, and can reference which agent responded and what provider was used for that response.

(The database is accessed through SQLAlchemy session in `database/models.py` and `db_session.py`, and can easily be migrated or extended. For instance, a `JobHistory` table could log status changes over time if needed for audit purposes^[22]^[L20-L23].)

Job Lifecycle

Once a quote is generated and delivered to the user, the system transitions through a `**job lifecycle**` if the quote is accepted. The typical stages are:

- `**Quote Drafted:**` *Trigger:* A new quote prompt is submitted. *System action:* The QuoteAgent generates a quote and it gets saved as a Quote record (status not yet a job)^[57]^[L60-L63]. The quote text and rationale are logged, and the customer may receive it for review.
- `**Job Created (Draft):**` *Trigger:* The customer accepts the quote or schedules a date. *System action:* A new Job entry is created in the database, linking to the Quote and Customer, with `status = "draft"` (meaning it's a pending job)^[57]^[L60-L64]. At this point, the system may also initiate external integrations (e.g., notify CRM) about the new scheduled job.
- `**Job Confirmed:**` *Trigger:* A specific date/time is assigned and the job is officially confirmed. *System action:* The Job record's status is updated to `"confirmed"` (and perhaps a field for `scheduled_date` is set)^[57]^[L62-L66]. This indicates the job is firmly scheduled. Integrations like calendar invites or scheduling software could occur here.
- `**Job Completed:**` *Trigger:* The job is marked done via the UI or API (after service is performed). *System action:* The Job's status is set to `"completed"`. It may also be flagged as archived or trigger an invoice creation process^[57]^[L64-L68].
- `**Invoice Sent:**` *Trigger:* Either manually triggered or automatically via Stripe/Xero integration when a job is completed. *System action:* An invoice is generated in Stripe or Xero, and the database records the invoice reference or payment status via a webhook^[57]^[L66-L69]. This is an optional stage where financial systems come into play (outside the core quoting).

These stages ensure a Quote can be tracked through to actual work and payment. The system's design allows hooking into each stage with events or webhooks. For example, when a job is created or confirmed, we might call the `**HubSpot**` integration to update the deal/ticket status (e.g., mark as `"Job Booked"`). When a job is completed and an invoice is sent, the `**Stripe**` integration can update the payment info in our system (and possibly back to HubSpot)^[57]^[L81-L89]. (See `**integrations.md**` for more on these external triggers.)

Data Backup & Export

The system employs several strategies to back up data and allow external analysis:

- `**Markdown Logging:**` Key interactions are logged as Markdown files for easy reading and versioning. For example, all quote conversations (prompts and

answers) append to a daily log in `logs/chat_history.md`, and daily job events can be logged in files like `logs/jobs/YYYY-MM-DD.md`. These human-readable logs serve as a quick backup and audit trail of what quotes were given on what dates.

- **Vector Store Snapshots:** The ChromaDB vector index is periodically snapshotted to files under `vector_backups/`. Typically, a dated folder (e.g. `/vector_backups/2025-07-28/`) will contain the serialized vectors (in a parquet or similar format) and associated metadata JSON. This ensures that the memory of past quotes can be restored or analyzed offline.

- **JSON Exports:** Important structured data can be exported to JSON for external use or analysis. For example, one could dump all quotes in a file like `export/quotes_2025-07-28.json` and full customer data in `export/customers_full.json`. These exports could be generated on a schedule or via an API endpoint, facilitating data integration with other systems or for training datasets.

All these backups are designed to preserve data in a format that's easy to load elsewhere and to provide recovery options. For instance, if the vector DB gets corrupted, one can reload it from the latest backup files. Or if one wants to train a fine-tuned model, the `quotes_*.json` export can serve as the dataset (possibly after filtering or scrubbing PII).

(For privacy, any exports intended for model training would have personal info removed as noted in the privacy policy. The backup files can be stored under version control or cloud storage for safekeeping. See `prompt_specs.md` for how some of these logs feed into model evaluation, and `integrations.md` for how job lifecycle hooks tie into external services.)

External Integrations & Automation (`integrations.md`)

Summary: This document explains how the system integrates with external services like CRM (HubSpot), payment/invoicing (Stripe, Xero), and automation tools (n8n). It details what triggers these integrations and what data is exchanged, ensuring the AI system can fit into a broader business workflow.

The 925stack AI system is designed to be **integration-ready**. Key external integration points include:

- **CRM (HubSpot):** On important events (such as a new quote created or a job scheduled), the system can update a customer record in HubSpot and log the quote details. **Trigger:** whenever a new Quote or Job is created in our system. **Action:** via the CustomerAgent or a webhook, create or update the contact in HubSpot, attach the quote text (and any quote PDF or summary) to that contact, and optionally create a deal/ticket for the job progression. **Data sent:** Customer name, email, the quote amount and description, scheduled date (if a job), and tags like "VIP" or job type can be synced. This gives sales teams visibility into the quoting activity directly in HubSpot. The integration can be implemented either by an

****n8n workflow**** (where our system sends a webhook and n8n uses HubSpot's API) or by the CustomerAgent calling HubSpot's API library directly [57fL75-L83] [54fL1-L4]. In either case, the system is flexible to accommodate the integration method.

- ****Payments (Stripe):**** When a quote is accepted or a job is completed and needs invoicing, the system interfaces with Stripe. ****Trigger:**** quote approval (to request payment) or job completion (to finalize payment) [57fL81-L85]. ****Action:**** create a payment link or invoice in Stripe for the quoted amount, and send it to the customer (this might be via email or a link in the web UI). The system also listens for Stripe webhook events: for instance, when an invoice is paid, it could mark the job as paid and update any relevant records (and possibly notify HubSpot of payment status) [57fL81-L85]. ****Data exchanged:**** customer info (name/email for billing), line item (job description and price) and invoice status updates. The integration ensures that the quoting process flows seamlessly into payment collection, and payment confirmations flow back into our system's records.

- ****Accounting (Xero):**** For some users, quotes/jobs might be pushed into Xero (accounting software) for official invoicing. ****Trigger:**** a manual action to generate an invoice, or automatically when a job is marked completed if Xero is preferred [57fL86-L89]. ****Action:**** create an invoice entry in Xero with the customer's details and job info, and possibly a corresponding draft or approved invoice. Also pull back the invoice status (paid/unpaid) via Xero's API or webhook notifications [57fL86-L89]. This could be used in lieu of or in addition to Stripe (e.g., Stripe for payment, Xero for accounting records). The data sync includes pushing the customer (if not already in Xero) and the invoice line items, and retrieving the invoice ID/status to store in our system.

All these integrations are optional and configurable. In a minimal setup, the system could run with none of them (just using its internal SQLite and sending quotes manually). In a full business setup, the ****CustomerAgent**** and webhooks ensure that whenever the AI generates a quote or a job is confirmed, the ****CRM**** has an up-to-date record (so sales sees it), and when the job is done, ****Stripe/Xero**** handle the billing without duplicate data entry.

****Implementation Note:**** The system uses the ****CustomerAgent**** as a bridge to integrations in some cases. For example, after QuoteAgent creates a quote, the RouterAgent might invoke CustomerAgent with a summary like "new quote for Customer X" which then triggers the HubSpot update logic. In other cases, simple outgoing webhooks (HTTP POST with relevant data) are defined for each event (e.g., an /api/webhook/hubspot endpoint could be called internally). The design is such that adding a new integration (say, scheduling software) would involve adding a new trigger in the job lifecycle and a handler (either in an agent or an external workflow). The project's integrations.md (this document) would be updated with setup instructions for API keys, webhook URLs, etc., for these services.

*(See **agents.md** for how CustomerAgent is defined to assist with CRM tasks. See **database.md** for fields like invoice status that would get updated by these integrations. DevOps note: environment variables or config files would store API keys for Stripe/HubSpot/Xero securely, not hard-coded.)*

Prompt Format & Spec Compliance (`prompt_specs.md`)

Summary: This document defines the strict format that quote responses must follow and how the system enforces these rules. It covers the markdown structure of quotes, the SpecGuard validation mechanism, example specifications and test cases, and how maintaining these specs ensures consistent, high-quality outputs.

Quote Formatting Rules

Every quote generated by the system follows a predefined format (inspired by the legacy QuoteGPT style) to ensure clarity and professionalism. The main rules are [63fL242-L247]:

- **Start with a Blockquote:** The quote text must begin with a `>` character, presenting the price and service as a blockquote.
- **Include Suburb and Price:** The quote line should explicitly mention the location (suburb) and the total quoted price (formatted as currency).
- **Attribution Line:** Following the quote text, include an em-dash line attributing it to "QuoteGPT" and the current year (e.g. "QuoteGPT, 2025").
- **Rationale Section:** After the quote, provide a brief rationale or explanation in plain text (1-2 sentences) explaining how the quote was determined (for example, referencing the job details or rates). This may be prefixed by "***Rationale:**" for clarity.

These rules are documented in the specification file **specs/quote_generation.md**. An excerpt from that file is:

```
```md
Rules
- Start quote with `>`
- Include suburb and total price
- Attribution line: "QuoteGPT, 2025"
- Rationale must follow (1-2 lines)
``` [63fL241-L247]
```

The QuoteAgent's output is expected to conform to this format every time. The system injects these rules into the LLM's context (via the Prompt Manager) to guide the model's response [63fL257-L260]. This means the system prompt given to the LLM often includes an outline of these requirements (so the model knows, for example, to use a blockquote and to add the attribution line).

SpecGuard Validation

To automatically check and enforce the above format, the system uses

SpecGuard is a validation utility that grades the LLM's output against the spec. After a quote is generated, SpecGuard's `grade_response(prompt, response, rules)` function runs. It evaluates multiple **criteria**:

- **Markdown Syntax:** Does the response start with a `>` quote block?
- **Attribution Format:** Does it end with the expected `"- QuoteGPT, YYYY"` line?
- **Rationale Presence:** Is there a rationale section following the quote (and is it within 1-2 sentences)?
- **Content Compliance:** Does it include the key details - the suburb/location, relevant tags or keywords, and a numeric total?

SpecGuard returns a **score** (e.g., 0.0 to 1.0) indicating how well the output meets the spec, and lists any **violations** of rules (e.g. `"missing markdown formatting"` if `>` was missing, or `"no rationale provided"`). It can also extract metadata, like identifying the tags present in the quote or confirming the suburb mentioned. For example, SpecGuard might return: `*score: 0.88*, *violations: ["missing markdown formatting"]*, *compliance: ["uses > block", "includes rationale"]*`. A 100% compliant output would score 1.0 with no violations.

The system uses this in two ways:

1. **Runtime feedback:** The compliance score might be logged for each quote and displayed in the admin dashboard as the `"Spec compliance %"` over time. This helps monitor if format quality is degrading. (The QuoteAgent could even be configured to retry or adjust if SpecGuard finds an issue, though in the current design it primarily logs the result rather than forcing a retry.)
2. **Automated testing:** In the development process, a set of test prompts with expected outcomes is defined (see *Validation Prompt Set* below). SpecGuard is run on the QuoteAgent's output for each test, and the test will fail if the score is below a threshold or if required content is missing. This ensures that changes to the system or model prompts don't break the format compliance.

Validation Sets & Examples

Under the `specs/` directory, alongside the rules file, there is a `validation.json` file containing example prompts and expected results. These serve as unit tests for the QuoteAgent. For instance, an entry might require that for the prompt `"Clean 12 windows in Attadale"`, the output must mention the suburb `"Attadale"` and include a `"window"` tag in the metadata. Another field could set a minimum SpecGuard score (e.g. 0.9) for the output. For example:

```
```json
[
 {
 "prompt": "Clean 12 windows in Attadale",
```



```

 "expect_suburb": "Attadale",
 "require_tags": ["window"]
}
]
"""[63][L249-L254]

```

This ensures the model **not** only formats the quote correctly but also correctly interprets the task (identifying that `"windows"` should lead to a `"window"` tag, **and** the suburb name should be reflected). During development, an **\*\*evaluation script\*\*** (e.g., `eval_quote_agent.py`) will run through these test prompts, generate quotes, use `SpecGuard.grade()` to score them, **and assert** that each meets the expected conditions[61][L203-L211]. If a prompt's output fails (say the suburb was missing **or** the score was too low), the test will flag it, prompting developers to improve the prompt template **or** adjust the rules.

### ### Continuous Improvement via Specs

The prompt spec can evolve **as** the project needs change. For example, **if** the business decides every quote must also include a specific disclaimer, a new rule can be added to the spec file **and** SpecGuard. Thanks to the SpecGuard enforcement **and** tests, the team would quickly see any outputs that don't include the disclaimer **and** can fine-tune the system accordingly. The spec-driven approach also means the team can maintain **\*\*consistency\*\*** even **as** multiple developers work on the project: the spec **is** a single source of truth **for** how outputs should look.

Additionally, the spec files **and** the SpecGuard mechanism play a role **in** model fine-tuning **and** evaluation (Phase 2 of the project). By accumulating a dataset of prompts **and** outputs that meet the spec (**and** those that don't), the team can train models **or** write better prompts to maximize the SpecGuard score. The compliance score itself **is** used **as** one of the performance signals **in** agent scoring (see **\*\*Architecture & System Design\*\*** [Admin Analytics]).

\*(For development details on how spec files are used **in** the coding workflow, see **\*\*development.md\*\*** under Spec-Driven Development. The SpecGuard implementation **is in** `core/spec_guard.py`, **and** prompt injection of rules **is** handled by `core/prompt_manager.py`[62][L1-L4].)\*

### ## Deployment & Operations (`deployment.md`)

**\*\*Summary:\*\*** This document provides guidance on deploying the 925stack AI system **in** different environments (local vs. cloud) **and** describes the Docker-based setup **for** production. It includes the use of containers **for** each service, **and** recommendations **for** running the system reliably.

The system **is** designed to run **\*\*locally\*\*** (**for** development **or** on-premise edge deployment) **as well as in** a **\*\*cloud\*\*** **or** hosted environment.

### ### Local Deployment (Offline Mode)

In local mode, all components run on a single machine (e.g., a developer's PC, a server, or even a Raspberry Pi). The configuration is set to use the local LLM backend (Ollama) instead of OpenAI. Key characteristics [48][L244-L253]:

- The target could be low-power hardware (tested on Raspberry Pi 4/5) or any standard desktop/server [65][L1-L4]. The LLM model (such as a fine-tuned Mistral or Llama2) will be loaded via **Ollama**, which must be installed on the machine.
- The backend (FastAPI) and the vector database (ChromaDB) run locally. They can be started via the CLI or a process manager. Because everything is on one machine, communication is internal (no need for cloud endpoints).
- It's recommended to use a tool like **tmux** or Docker Compose even locally to manage the multiple processes (API server, possibly a separate vector DB process, Ollama serving the model, etc.) [65][L1-L4]. In a simple setup, one might run `ollama serve` (to ensure the local model endpoint is up) and then `uvicorn backend.main:app` to start the API, and maybe `npm run dev` for the frontend, all on the same host.
- The CLI can be used offline as well (`python cli.py --offline ...`), which will route all queries to the local model (ensuring no external API calls are made). This mode is useful for demos or use in areas without reliable internet, albeit with possibly lower-quality responses depending on the local model.

### ### Cloud Deployment (Hosted Mode)

In cloud or production deployment, one might separate concerns: host the frontend on a platform like Vercel or Netlify, and the backend on a server or container platform. In this mode, the OpenAI API is typically enabled for best performance (assuming internet access and API keys). Key points [48][L250-L258]:

- **Backend Hosting:** The FastAPI backend (and associated services) can be containerized and run on cloud providers such as Fly.io, Render, Railway, or any Kubernetes/VPS setup. Using a managed service for the API ensures it's accessible with proper scaling and TLS. Environment variables or secrets would store the OpenAI API keys, etc. The backend might run with `LLM_MODE="openai"` in config to default to GPT-4, but can still fallback to Ollama if a local instance is accessible (in cloud, this might mean running an Ollama container alongside).
- **Frontend Hosting:** The React frontend can be built and served as a static app (if using Next.js static export) or a small Node server. Platforms like Vercel or Netlify can auto-deploy the frontend from the repo. It will make API calls to the backend's URL. CORS must be configured on the backend to allow the frontend domain.
- **Security & Auth:** If exposing the API publicly, one should secure it. The blueprint shows an optional auth middleware requiring an `Authorization` header on requests [48][L177-L185]. In production, you'd enable such middleware with token checking. Additionally, API keys for any external integration (Stripe, etc.) should be stored securely (e.g., Fly.io secrets, .env files not committed to Git).

- **Streaming & Scale:** The cloud deployment should enable streaming responses from OpenAI (so the frontend can display partial outputs). The design already anticipates this with Server-Sent Events (EventSource) in the UI. Scaling horizontally might involve running multiple instances of the API behind a load balancer, and possibly a shared database/vector store (which could be a managed Postgres for the data and a centralized Chroma or switching to a hosted vector DB).

Continuous deployment can be set up via GitHub Actions [for instance](#), pushing to the `main` branch could trigger building and deploying the Docker containers. The project includes a CI workflow that runs tests and linters on each push to ensure stability (see `development.md`).

### ### Docker Compose Setup

For convenience and consistency, the project provides a **Docker Compose** configuration that orchestrates all services in containers. The `docker-compose.yml` (in the `docker/` folder) defines the following services [L349-L349](#) [L45-L45](#) [L350-L350](#) [L358-L358](#):

- **api:** The FastAPI backend container (serving on port 8000).
- **ui:** The frontend React app container (serving on port 3000).
- **chroma:** A ChromaDB container for the vector database (persisting embeddings, usually on port 8000 internally) [L45-L45](#) [L350-L350](#).
- **ollama:** An Ollama container which serves the local LLM (listening on port 11434 by default) [L45-L45](#) [L352-L352](#).
- **n8n:** (Optional) an n8n automation tool container, which can be used to host workflow automation for integrations (on port 5678) [L45-L45](#) [L353-L353](#).

A simplified snippet of the compose file:

```
```yaml
services:
  api:
    build: ./backend
    ports:
      - "8000:8000"
  ui:
    build: ./frontend
    ports:
      - "3000:3000"
  chroma:
    image: chromadb/chroma
    ports:
      - "8001:8000"
  ollama:
    image: ollama/ollama
    ports:
```

```
- "11434:11434"
```

```
n8n:
```

```
  image: n8nio/n8n
```

```
  ports:
```

```
    - "5678:5678"
```

```
```[45fL341-L349][45fL350-L358]
```

Using `docker-compose up` will build/start all these services together. In this setup, the backend connects to Chroma on the `chroma` service network name, Ollama is available for local inference, and the frontend container can proxy API requests to the backend. This makes deployment to any Docker-compatible host straightforward. One could run this on a cloud VM to have a fully self-contained system.

**\*\*Local vs Cloud Compose:\*\*** The same compose can be used locally (for development, ensuring parity with production) by swapping environment variables (e.g., running in local mode vs requiring API keys). For production, one might remove the `n8n` if not needed or add a reverse proxy (like Caddy or Nginx container) to handle HTTPS termination for the UI+API domain `[48fL257-L260]`.

**\*\*Monitoring:\*\*** The compose setup could be extended with monitoring containers if needed (e.g., for logging or metrics aggregation), but given the small scope, checking the logs and using the admin dashboard might suffice.

\*(For DevOps and CI details, see **\*\*development.md\*\*** `[45fL341-L349]` it covers how tests and linting must pass before deployment. The **\*\*architecture.md\*\*** and **\*\*integrations.md\*\*** discuss how to configure environment variables for different modes. Always test the Docker images locally or in a staging environment to ensure the GPU/CPU requirements for LLMs are met and integration credentials are properly set.)\*

### ## 🚀 Developer Workflow & CI (``development.md``)

**\*\*Summary:\*\*** This document outlines the development practices for the project, including the local development setup, Git branching strategy, coding standards, testing, CI/CD, and how the team uses the Codex assistant for pull request automation. It ensures all contributors follow a consistent workflow for quality and efficiency.

### ### 🖥️ Local Development Environment

To set up the project for development, ensure you have the following tools (or later versions) installed `[11fL180-L188]`:

- **\*\*Python 3.11+\*\*** `[45fL341-L349]` for running the backend, agents, and CLI
- **\*\*Node.js 18+\*\*** `[45fL341-L349]` for building and running the React frontend
- **\*\*Docker (Latest)\*\*** `[45fL341-L349]` for containerization and to run services like ChromaDB and Ollama if needed
- **\*\*Git\*\*** `[45fL341-L349]` for version control

- **Ollama (Latest)** for local LLM inference (if developing/tested offline)
- **SQLite3** (comes with Python or OS) to inspect the local database if needed

It's recommended to use **VS Code** with extensions like Python (Pylance), ESLint/Prettier for JavaScript/TypeScript, GitLens for Git history, and Markdown Preview Enhanced for viewing docs [11†L191-L196]. These help maintain code quality and follow the project's style conventions.

### Git Branching Strategy

The project uses a structured Git branching model to manage new features and releases. The main branches include [11†L201-L209]:

- **main** production-ready code (protected, no direct commits)
- **foundation1.0** the base stable refactor branch (branch off this for any new work)
- **frontend-sync-v1** integration of React frontend with the backend API
- **agent-router-v1** development of the multi-agent dispatch system
- **quote-pipeline-v2** improvements to quoting logic and pricing pipeline
- **integration-hubspot** working branch for external CRM integration features

Developers create feature branches off of **foundation1.0** (or an appropriate integration branch) rather than off **main** to ensure **main** always contains tested code ready for release. For example, a new feature might be developed in **feat/offline-vector-search** branched from **foundation1.0**, then merged back via a pull request.

### 👉 Pull Request Protocol

All contributions are integrated via pull requests (PRs). Each PR should adhere to these rules [66†L1-L4]:

- **Branch from foundation1.0**: Never commit straight to **main**; always develop in a feature branch and PR into a base branch (which eventually merges to main).
- **Meaningful Names**: Use descriptive branch names prefixed with **feat/** for features or **fix/** for bugfixes (e.g., **feat/add-fallback-LLM**) [66†L1-L4].
- **Update Changelog**: Every PR that changes functionality should include an update to **docs/CHANGELOG.md** under an "Added", "Changed", or "Fixed" section for the next version [66†L2-L4].
- **Include PR Meta Block**: Following the o3-Pro Codex conventions, developers should include a JSON "PR meta" block in the PR description. This block contains fields like **patch**, **explanation**, **risk\_level**, **pr\_meta** (with a structured title/body) [66†L3-L4]. The **925gitGPT** assistant uses this to analyze or generate patches. For example, a PR description might include:

```
{
 "patch": "feat: add fallback LLM",
 "explanation": "Add fallback LLM for better performance",
 "risk_level": "Low",
 "pr_meta": {
 "title": "feat: add fallback LLM",
 "body": "This PR adds a fallback LLM to the system."
 }
}
```

```

{
 "patch": "<diff>",
 "explanation": "Adds spec validation for quoting agent",
 "risk_level": "low",
 "pr_meta": {
 "title": "feat: quote agent spec compliance",
 "body": "Enforces markdown format + rationale presence.\nCloses #QUOTE-
SPEC-002"
 }
}
``` [63fL265-L273]

```

This [is](#) the format the AI assistant expects, [and](#) it allows automated tooling to extract the intent [and](#) changes of the PR easily. (During development, the assistant itself can generate such a template given a description of changes.)

The repository uses a **pull request template** that reminds contributors to include these elements (branch name conventions, changelog update, PR meta, etc.) to streamline reviews.

Testing and CI

The project maintains a comprehensive test suite to ensure all components function [as](#) expected:

- **Unit Tests:** Located [in](#) the `/tests/` directory, covering each agent [and](#) core module [11fL105-L113]. For example, `test_quote_agent.py` checks that QuoteAgent returns outputs [in](#) the correct format (possibly by mocking the LLM), `test_memory_agent.py` verifies that the vector search returns expected results [for](#) a known embedding, `test_router_agent.py` ensures prompts route correctly, [and](#) so on [11fL105-L113]. There are also tests [for](#) configuration loading, vector store operations, etc.
- **CLI Tests:** e.g., `tests/test_cli.py` ensures that the CLI interface works `--feeding` a prompt yields a well-formed quote, the `--offline` flag forces the local model, the `--search` prints memory results, etc. [11fL119-L127]. These tests simulate CLI calls [and](#) inspect the output [or](#) return code.
- **Integration Tests:** Some tests ensure that the agents [and](#) database work together. For instance, one might run a full quote generation [and](#) then query the database to confirm a Quote entry was created, [or](#) test the `customer` API endpoint end-to-end [with](#) a test client.

All tests are run [with](#) **pytest**, [and](#) continuous integration [is](#) set up to run them on each push. The CI (see `.github/workflows/ci.yml`) will install dependencies, run `pytest`, [and](#) also run linters [11fL137-L145] [45fL316-L323]. We use:

- **Code Formatters/Linters:** `black` [for](#) code formatting, `flake8` [for](#) style issues, `isort` [for](#) import sorting [45fL316-L323]. The CI will fail [if](#) code doesn't meet linting standards (developers are encouraged to run `black .`

and `flake8` locally before committing).

- **Type Checking:** `mypy` is used to ensure type annotations are correct [45fL316-L323], preventing certain classes of bugs.
- **Pre-commit Hooks:** The repository provides a pre-commit configuration to automate these checks locally. On each commit, it can run black, flake8, mypy, etc., so the CI is usually just a formality (this is set up via the `.pre-commit-config.yaml` referencing those tools [45fL323-L329]).

Additionally, specific tests are in place for critical functionality:

- The vector memory is tested by an **integrity test** that adds a known embedding and ensures it can be retrieved (to catch any issues in vector store configuration) [45fL318-L321].
- The API endpoints are tested with dummy requests (e.g., ensuring `/api/quote` returns a 200 and a JSON with expected fields, `/api/memory` returns a list of strings, etc.) [45fL319-L322].
- The SpecGuard logic is tested against some prepared outputs to ensure it correctly identifies compliance or violations.
- The end-to-end test might run a sample prompt through the CLI or API with `--debug` to verify that all pieces (Agent → LLM → SpecGuard → DB log) work in concert.

The CI must pass all tests and checks before a PR is allowed to merge into `main`. This practice keeps the `main` branch stable and deployable at all times.

Changelog & Release Management

Developers update the **CHANGELOG.md** with each meaningful change. Entries are grouped by version. For example [12fL225-L232]:

```

'''markdown
## [0.2.0] - 2025-08-01
### Added
- RouterAgent logic
- GPT-4/Ollama fallback system

### Fixed
- Chroma injection loop

```

When preparing for a new release (say v1.0), the team will review the changelog to ensure all notable changes are recorded. The changelog, along with the documentation (this knowledge base), helps in creating release notes. There is also a plan to use tools like `auto-changelog` in the future to enforce consistency in changelog entries (potentially tying into PR descriptions).

Codex Assistance in Development

This project embraces AI assistance (via GitHub Copilot or the custom 925gitGPT). The **925gitGPT** Codex assistant can help by generating code patches given a PR meta JSON or by explaining parts of the system.

Developers incorporate it by providing well-structured prompts (as seen in the PR template JSON). Some automated tasks the Codex can handle include ⁴³ : adding a new router method, injecting new spec rules into the prompt manager, updating the CLI for a new flag, creating evaluation scripts, or even building out a new integration route. The PR metadata format (mentioned above) is specifically designed for AI readability, enabling the assistant to output patches that are easy to review and merge.

Productivity Tips

- The repository structure (see **architecture.md**) segregates concerns, so when adding a new feature, try to follow that structure (e.g., if adding a new agent, create a new file in `backend/agents/`, write tests in `tests/test_new_agent.py`, document any new commands or endpoints in the docs).
- Use feature flags and config toggles when introducing changes that might be used in both local and cloud modes (for example, a setting to enable verbose logging). This way the same codebase can be flexibly deployed.
- Before merging to `main`, run through the **Manual QA Checklist** from the blueprint ⁴⁴ : test an offline quote generation, a full `/api/quote` call from the frontend, verify vector search returns context, check the UI rendering of a quote and rationale, and open the admin dashboard to see if metrics appear. This helps catch any integration issues that automated tests might miss (like something not showing up in the UI).

By following this development workflow, the team ensures that the system remains robust, and that adding new capabilities (like more agents or integration endpoints) will not break existing functionality. The combination of spec-driven development, thorough testing, and AI-assisted PR generation positions the project for efficient scaling and maintenance.

(Refer to `prompt_specs.md` for how new spec rules are added alongside code changes. See `roadmap.md` for upcoming features that developers might be working on, like fine-tuning ops or expanded integrations.)

Roadmap & Future Plans (`roadmap.md`)

Summary: This document highlights envisioned enhancements and stretch goals for future development beyond the initial release (v1.0). These items are not yet implemented but serve as guidance for Phase 2 and beyond.

- **Agent Self-Improvement via Reward Modeling:** Implement a feedback loop where the system learns from outcomes. For example, using reinforcement learning or reward modeling to let QuoteAgent refine its pricing strategy based on job acceptance rates (successful quotes vs. rejected quotes) ⁴⁵ . This could involve training a model to adjust quotes to maximize acceptances while meeting business constraints.
- **Quote Comparison & Optimization:** Provide features to compare quotes across different regions or historical data. This might include an AI-driven analysis of how quotes differ by suburb or job type, or recommending the best quote among alternatives ⁴⁵ . It could also optimize multi-item quotes (if quoting multiple services together).

- **Mobile/PWA Support:** Extend the frontend to a mobile-friendly Progressive Web App so that on-site technicians or customers can use the quoting tool on smartphones ⁴⁶. This might involve offline caching of recent quotes, a simplified UI for mobile, and push notifications for quote updates or job confirmations.
- **Federated or Distributed LLM Hosting:** Explore running the local LLM (Ollama) in a distributed manner for scalability. For instance, **federated Ollama node clustering** where multiple local model servers share load or specialty (one might handle image-related queries if that becomes relevant, etc.) ⁴⁶. This could reduce latency by having edge LLMs in various regions or allow bigger models by distributing parts of the model.
- **Fine-Tuned Model Versions:** Using the data collected (quotes and their outcomes), develop fine-tuned versions of LLMs that are specialized in quoting for the cleaning business domain. This could involve training a smaller model that can run faster locally with nearly the accuracy of GPT-4 for this specific task. Fine-tuning might also produce a model that better adheres to the quote format without needing as much prompt guidance (embedding some of the spec into its weights).
- **Enhanced Integrations:** Add more integrations as needed: for example, integration with scheduling calendars (Google Calendar or Calendly) for jobs, integrating with mapping APIs to estimate travel time (which could factor into pricing), or linking with inventory systems if quotes involve product supplies.
- **Analytics & Reporting:** Build out the admin dashboard into a full analytics suite. This includes token usage stats (how many API tokens used per quote, cost tracking for using GPT-4), success rate of quotes (percentage that become jobs), average quote turnaround time, etc. Additionally, allow exporting these reports or scheduling them via email.
- **Multi-Language Support:** Enable the quoting system to work in multiple languages or locales (for expansion into new regions). This might involve translating the prompt spec and having separate spec files per language, and selecting the appropriate LLM or prompting for language.
- **Plugin & Tool Integration:** Incorporate OpenAI function calling or tools if needed – for example, if in the future we integrate a mapping tool, the QuoteAgent could call a function to get area measurements or other details to better calculate a quote (Toolformer style integration, as noted in research).

These roadmap items would be tackled after the initial stable release. They represent growth areas to make the 925stack AI system more powerful and widely applicable. Prioritization would depend on user feedback and business needs – for instance, if users demand a mobile app, that might come first; if consistency and cost are an issue, fine-tuning a custom model could be prioritized.

(The roadmap is subject to change. See the end of `architecture.md` ⁴⁷ for the original list of future ideas that guided these entries. As development progresses, new ideas may be added here.)

1 2 3 4 6 7 8 9 10 11 12 13 16 40 43 quoting_logic_and_specguard.md

file:///file-DrorRFCJaVPUtjr6opgnjZ

5 14 15 23 24 25 26 27 28 29 30 31 32 33 34 new_repo_research_doc_part_9.md

file:///file-3GKhRBq8W9wgeVmBZoLrDB

17 18 19 20 21 22 38 39 41 44 45 46 47 backend_architecture_and_deployment.md

file:///file-DkkXg2uGVs9mm163JBjQJL

35 new_repo_research_doc_part_1.md

file:///file-Nh xv8yBVeLNsKU FmXTFPDd

36 37 42 new_repo_research_doc_part_4.md

file:///file-XcH5zTEEr umAaKLPY6cB6V