Project Title: Code reviewer using mulit-agent system

Synompsys
what are the sections of a code will the agent Review:

1. Architecture & Design (The "Big Picture")
    a. **Scalability:** Will this solution work if we have 10,000 users instead of 10? (e.g., Storing session data in local server memory vs. a shared Redis cache).
    b. **Reusability (YAGNI):** Did the developer write a generic "engine" for a simple one-off task? Or conversely, did they hardcode something that will definitely change next month?
    c. **Dependency Bloat:** Did the author import a massive 50MB library just to left-pad a string?

2. Functionality & Complexity
    a. **The "Happy Path" vs. Reality:** The code works when inputs are perfect, but does it crash on `null`, `undefined`, empty lists, or negative numbers?
    b. **Algorithmic Efficiency:** Are there nested loops (O(n2)) iterating over a list that could grow large?
    c. **Concurrency & Threading:** (If applicable) Are there race conditions? Is a shared resource being accessed by two threads at once without a lock?
    d. **Error Handling:** When an error occurs, does the app crash silently, or does it log a meaningful error message? Are exceptions being swallowed (`try { ... } catch (e) { // do nothing }`)?
    e. **Test Coverage:** Are there unit tests for the new logic? Do the tests actually fail if the logic is broken, or are they just "assertion-free" tests to pass coverage metrics?

3. Security
    a. **Injection Attacks:** Is user input concatenated directly into SQL queries (SQL Injection) or rendered directly to the DOM (XSS)?
    b. **Broken Access Control (IDOR):** Just because I'm logged in, can I access data belonging to User #55 if I simply change the ID in the URL?
    c. **Secrets Management:** Are there API keys, passwords, or AWS tokens hardcoded in the file?
    d. **Data Leakage:** Is sensitive data (PII like emails, passwords, credit cards) being printed to the console/logs for debugging?
    e. **Dependency Vulnerabilities:** Is the code using an old version of a library that has a known CVE (Common Vulnerabilities and Exposures)?

4. Maintainability & Clean Code
    a. **Naming: Do variables reveal intent?**
        - *Bad:* `let d = 5;`
        - *Good:* `let retry_delay_seconds = 5;`

b. **Function Size (SRP):** Does a single function do 5 different things (parse data, save to DB, email user, update UI)? It should be split up (Single Responsibility Principle).
c. **Magic Numbers:** Are there unexplained numbers in the logic?
    i. *Bad:* `if (status == 2) ...`
    ii. *Good:* `if (status == ORDER_SHIPPED) ...`
d. **Comments:** Do the comments explain *why*?
    i. *Useless:* `i++; // increment i`
    ii. *Useful:* `// We increment here to skip the header row in the CSV.`
e. **Dead Code:** Is there commented-out code left behind "just in case"? (Git is for history; delete it from the file).

5. Style (The Lowest Priority)
    a. **Consistency:** If the project uses `camelCase`, did this file switch to `snake_case`?
    b. **Formatting:** Are brackets on the same line or next line? Is indentation 2 spaces or 4 spaces?
    c. **File Structure:** Are imports sorted alphabetically? (Makes merge conflicts easier to resolve).
    d. **Grammar:** Are there typos in the user-facing text or variable names?

Input to the System:

1. **The Target Code (The Diff/File):**
    - *What it is:* The actual file or function being reviewed.
    - *Why:* This is the subject of the review.
    - Parent cl
2. **Dependency Manifest (`requirements.txt` / `package.json`):**
    - *Why:* This is **critical** for your "Dependency Bloat" and "Security" checks.
    - *Example:* If the user uploads a Python file using `pandas`, the AI needs to see `pandas==1.1.0` in the requirements file to know if it's an outdated, insecure version.
3. **Context/Readme (The "Intent"):**
    - *What it is:* A natural language summary of what the code *should* do.
    - *Why:* This allows the AI to check logic.
    - *Example:* "This function should take user input and save it to the `users` table." (Now the AI knows to check for SQL injection).
4. **Related/Reference Files (Optional Context):**

    - *What it is:* Interfaces, database schemas, or utility files that the Target Code calls.
    - *Why:* If the Target Code calls `db.save(user)`, the AI needs to see the `db` class to know if it handles connection pooling or transactions correctly.
5. **Test Files (Optional):**
    - *Why:* To verify if edge cases (nulls, empty lists) are covered.

System Configuration

System Output

This data structure is designed to be the "contract" between your AI Backend (Python/FastAPI) and your Frontend (Streamlit/React).

---

1. Root Level Keys

These keys provide high-level information about the entire review session.

Key	Type	Definition & Usage
review_id	String	**Unique Identifier.** A UUID (e.g., rev_550e8400) used to track this specific review in your database or logs. Essential for debugging.
timestamp	String	**Time of Completion.** formatted in ISO 8601 (e.g., 2025-10-25T14:30:00Z). Useful for sorting reviews historically.
summary	String	**Executive Summary.** A natural language paragraph generated by the "Judge" agent. It gives a quick overview of the code's health before the user dives into details.
praise	List[Str]	**Positive Reinforcement.** A list of strings highlighting what the developer did *well*. This balances the critique so the user doesn't feel attacked.

2. The `meta` Object

These keys are calculated metrics used for **UI Badges** and **Gamification**. They allow the user to understand the state of the code at a glance.

Key	Type	Definition & Usage
`final_verdict`	`Enum`	**The Decision.** Tells the UI what big icon to show. • `APPROVE`: Green Checkmark (Good to merge). • `REQUEST_CHANGES`: Red Cross (Must fix blocking issues). • `COMMENT_ONLY`: Yellow Warning (Suggestions only, no blockers).
`quality_score`	`Integer`	**The Grade (0-100).** An algorithmic score based on the number and severity of bugs found. *Formula Example:* Start at 100, subtract 10 for Critical, 5 for High, 1 for Low.

risk_level	Enum	**Security Status.** Driven exclusively by the "Security Hawk" agent. • CRITICAL: Immediate exploit found (e.g., SQL Injection). • HIGH: Major vulnerability. • LOW/SAFE: No obvious threats.

---

3. The comments List Object

This is the core of your application. Each object in this list represents a specific issue found by one of your agents.

Key	Type	Definition & Usage
id	String	**Comment ID.** Unique ID for the specific comment (e.g., c_1). Useful if you want to implement a "Dismiss" or "Resolve" button in the UI.
file_path	String	**Location.** The name of the file containing the error (e.g., src/utils.py).
line_start	Integer	**Start Line.** The line number where the issue begins. Used to highlight code in the editor.

line_end	Integer	**End Line.** The line number where the issue ends.
category	Enum	**Filter Tag.** Used to color-code the comment card. • SECURITY (Red) • PERFORMANCE (Orange) • MAINTAINABILITY (Blue) • ARCHITECTURE (Purple)
severity	Enum	**Priority.** Tells the developer what to fix first. • CRITICAL: Must fix now. • NITPICK: Subjective or minor (e.g., "add a space here").

title	String	**Headline.** A short, 5-7 word summary of the issue (e.g., "Possible SQL Injection").
body	String	**The Description.** A detailed explanation of the error.
suggestion	String	**The Fix.** A code snippet showing how to rewrite the code correctly. The UI can display this as a "Diff."
rationale	String	**The "Why".** Explains the Computer Science principle behind the error. (e.g., "Using `eval()` is dangerous because..."). This is the educational component.
references	List[URL]	**Further Reading.** Links to official docs or StackOverflow discussions supporting the agent's claim.
policy_violated	String	**The Rule.** Links back to the `org_policy.json` config. Tells the user *why* their code was rejected based on company rules (e.g., "Violated Rule: `no_nested_loops`").

Why these keys matter for your Competition:

1. **`policy_violated`**: Proves your system is **Enterprise Ready** (it enforces rules, it doesn't just guess).
2. **`rationale` & `references`**: Proves your system is **Educational** (it helps Junior Devs grow).
3. **`praise`**: Proves your system has **Human-Centric Design** (it isn't just a harsh robot).

Would you like me to generate a sample `org_policy.json` (Configuration File) next, so we can see how the "Input" connects to this "Output"?

4. Final JSON Example (With New Features)

JSON


```
{
  "review_id": "rev_882910",
  "timestamp": "2025-10-25T14:30:00Z",
  "meta": {
    "final_verdict": "REQUEST_CHANGES",
    "quality_score": 65,
    "risk_level": "HIGH"
  },
  "summary": "The submission implements the user login flow correctly but introduces a Critical Security Vulnerability via direct SQL injection. While the variable naming is excellent, the database interaction needs immediate refactoring.",
  "praise": [
    "Excellent use of descriptive variable names in `auth_controller.py`.",
    "Good separation of concerns by keeping HTML generation out of the database layer."
  ],
  "comments": [
    {
      "id": "c_1",
      "file_path": "src/db/queries.py",
      "line_start": 45,
      "line_end": 45,
      "category": "SECURITY",
      "severity": "CRITICAL",
      "title": "SQL Injection Vulnerability",
      "body": "User input is being directly concatenated into the SQL query string.",
      "suggestion": "cursor.execute('SELECT * FROM users WHERE email = %s', (email,))",
```

```
      "rationale": "Direct concatenation allows attackers to manipulate the query (e.g., 'OR 1=1') to bypass authentication.",

      "references": ["https://owasp.org/www-community/attacks/SQL_Injection"],

      "policy_violated": "security.banned_patterns: [sql_concatenation]"

    },

    {

      "id": "c_2",

      "file_path": "src/utils/parser.py",

      "line_start": 12,

      "line_end": 20,

      "category": "PERFORMANCE",

      "severity": "MEDIUM",

      "title": "Inefficient String Concatenation",

      "body": "Using `+=` in a loop to build a large string is O(n^2).",

      "suggestion": "results = []\nfor item in items:\n    results.append(item)\nreturn ''.join(results)",

      "rationale": "Python strings are immutable. Each `+=` creates a new string object, copying all previous characters.",

      "references": ["https://docs.python.org/3/faq/programming.html#what-is-the-most-efficient-way-to-concatenate-many-strings-together"],

      "policy_violated": "performance.efficiency_check"

    }

  ]

}
```

Multi Agents

Three specialized Agents:

Here is the fully updated profile for **Agent A**, incorporating the detailed tool definitions we finalized.

Agent A: The Security & Risk Auditor ("The Hawk")

> *"I don't care if it works. I care if it's safe."*

Role: Cyber Security Specialist.

Focus Areas:

- **Injection Attacks:** SQL Injection (SQLi), Cross-Site Scripting (XSS), Command Injection.
- **Broken Access Control:** Insecure Direct Object References (IDOR), missing authorization checks.
- **Hardcoded Secrets:** API Keys, Passwords, Tokens, AWS Credentials.
- **Supply Chain Attacks:** Vulnerable dependencies (e.g., using an old version of `flask` with known CVEs).

Input:

The "Security Payload" (Filtered for relevance):

- **Target Code (Logic Only):**
  - *Included Extensions:* `.py`, `.js`, `.ts`, `.java`, `.go`, `.php`, `.rb`, `.sql`.
  - *Configuration Files:* `.yaml`, `.json`, `.xml`, `.env.example`, `Dockerfile`.
  - *Excluded:* CSS, Images, Documentation, and Test files (unless specifically scanning for secrets in tests).
- **Dependency Manifests:** `requirements.txt`, `package.json`, `pom.xml`, `go.mod`.
- **Context:**
  - `README.md` (To understand if the app is public-facing/critical).
  - `org_policy.json` (To know the "Vulnerability Tolerance" and banned functions).

Tools:

1. **`cve_lookup(library_name, version)`**
   - **Purpose:** Queries a vulnerability database to determine if a specific package version has known security defects (Supply Chain).
   - **Input:** `library_name` (str), `version` (str).
   - **Output:** JSON containing status (`VULNERABLE`/`SAFE`) and a list of specific CVE IDs with severity scores.
     1. {

```
  "status": "VULNERABLE",  // or "SAFE"
  "cves": [
    {
      "id": "CVE-2018-10001",
      "severity": "HIGH",
      "description": "Remote Code Execution in Flask < 1.0",
      "fix_version": "1.0.1"
    }
  ]
}
```

2. **scan_secrets(file_content)**
   - **Purpose:** Scans text content using high-entropy regex patterns to detect accidental commits of sensitive credentials (API keys, AWS tokens).
   - **Input:** `file_content` (str).
   - **Output:** JSON listing found secrets with line numbers and secret type (redacted for safety).

```
{
  "found_secrets": true,
  "matches": [
    {
      "line_number": 42,
      "type": "AWS_ACCESS_KEY",
      "snippet": "AKIA**********" // Redacted for safety
    },
    {
      "line_number": 105,
      "type": "Generic_API_Key",
      "snippet": "key = 'sk_live_******'"
    }
  ]
}
```

3. **analyze_ast_patterns(code_content)**
   - **Purpose:** Parses code into an Abstract Syntax Tree (AST) to find structural vulnerabilities regex misses, such as SQL injection via concatenation or use of `eval()`.
   - **Input:** `code_content` (str).
   - **Output:** JSON listing dangerous patterns found, including the vulnerability type (e.g., `SQL_INJECTION`) and severity.

```
{
  "risk_found": true,
  "patterns": [
    {
      "line": 15,
      "type": "SQL_INJECTION",
      "details": "Detected string concatenation ('+') inside 'cursor.execute()'.",
      "severity": "CRITICAL"
    },
```

```
{
  "line": 88,
  "type": "DANGEROUS_FUNCTION",
  "details": "Usage of 'eval()' detected.",
  "severity": "HIGH"
 }
]
}
```

4. **audit_route_permissions(code_content)**
    - **Purpose:** Identifies web API endpoints and analyzes decorators to determine if they are publicly accessible or protected by authentication (IDOR check).
    - **Input:** code_content (str).
    - **Output:** JSON list of routes found, including HTTP method, decorators present, and an assessed risk_level (HIGH if auth is missing).

```
{
 "routes_found": [
  {
   "path": "/api/users/delete",
   "method": "POST",
   "line": 20,
   "decorators": ["@app.route"],
   "auth_check": "MISSING", // The agent uses this field to scream "IDOR!"
   "risk_level": "HIGH"
  },
  {
   "path": "/api/dashboard",
   "method": "GET",
   "line": 55,
   "decorators": ["@app.route", "@login_required"],
   "auth_check": "PRESENT",
   "risk_level": "SAFE"
  }
 ]
}
```

ReAct (Reasoning + Acting) Strategy:

- **System Prompt:** Operate in a **Thought-Action-Observation** loop. Do not guess; verify.
- **Logic:**
    1. **Scan Manifests:** If a requirements.txt is present -> Call cve_lookup.
    2. **Scan Secrets:** If high-entropy strings are found -> Call scan_secrets.
    3. **Scan Logic:** If database calls or routes are detected -> Call analyze_ast_patterns and audit_route_permissions.
    4. **Report:** Only report confirmed risks with Evidence (the tool output).

Here is the detailed design for **Agent B**, structured exactly like Agent A.

Agent B: The Architect & Performance Engineer ("The Speed Demon")

> *"I don't care if it's pretty. I care if it scales."*

Role: Principal Systems Architect / Site Reliability Engineer (SRE).

Focus Areas:

- **Algorithmic Efficiency:** Detecting Time Complexity issues ($O(n^2)$, $O(n^3)$), redundant computations, and inefficient data structures (e.g., using a List for lookups instead of a Set).
- **Database Hygiene:** Identifying "N+1" query problems (queries inside loops), missing transactions, and fetching excessive data (`SELECT *` without pagination).
- **Resource Management:** Detecting memory leaks, unclosed file handles/connections, and blocking I/O operations in async contexts.
- **Concurrency & Safety:** Race conditions, lack of thread safety, and improper use of locks.

Input:

The "Architecture Payload" (Filtered for structural analysis):

- **Target Code (Logic & Data):**
  - *Included Extensions:* `.py`, `.js`, `.ts`, `.java`, `.go`, `.cpp`, `.sql`.
  - *Configuration Files:* `docker-compose.yml` (to see infrastructure limits), `k8s.yaml`.
  - *Excluded:* Tests (performance of tests matters less), CSS, Documentation, HTML templates.
- **Dependency Manifests:** `requirements.txt` (To check for heavy/slow libraries like pandas vs polars).
- **Context:**
  - `org_policy.json`: **Crucial.** Checks the `scale` parameter (e.g., "High_Traffic" vs "Prototype"). If "High_Traffic", even $O(n \log n)$ might be scrutinized.
  - `README.md`: To understand the system architecture (Monolith vs Microservices).

Tools:

1. **`calculate_cognitive_complexity(code_content)`**
   - **Purpose:** Parses the AST to measure nesting depth (loops inside loops inside conditionals).
   - **Input:** `code_content` (str).

    - **Output:** JSON containing a `complexity_score` (int) and a list of specific blocks where nesting exceeds the threshold (indicating potential $O(n^2)$).
2. **`scan_database_patterns(code_content)`**
    - **Purpose:** Detects dangerous database interaction patterns, specifically the "N+1 Problem" (database calls inside iteration blocks).
    - **Input:** `code_content` (str).
    - **Output:** JSON listing risky lines where a DB execution method (e.g., `.execute`, `.query`, `.save`) appears inside a `for` or `while` loop.
3. **`detect_blocking_ops(code_content)`**
    - **Purpose:** Identifies synchronous blocking operations (like `time.sleep`, heavy file I/O, or synchronous HTTP calls) used within asynchronous functions (`async def`).
    - **Input:** `code_content` (str).
    - **Output:** JSON listing blocking calls that will freeze the event loop.
4. **`analyze_memory_usage(code_content)`**
    - **Purpose:** Heuristic check for memory-hogging patterns, such as reading entire files into memory (`.read()`) instead of streaming, or creating massive global lists.
    - **Input:** `code_content` (str).
    - **Output:** JSON warnings for variables or patterns that risk Out-Of-Memory (OOM) errors.

ReAct (Reasoning + Acting) Strategy:

- **System Prompt:** You are a Performance Engineer. Assume the code is running in a high-latency, resource-constrained environment.
- **Logic:**
    1. **Scan Complexity:** Call `calculate_cognitive_complexity`. If score > 15, flag as "Hard to Scale/Maintain."
    2. **Scan Loops:** Call `scan_database_patterns`. If a DB call is found in a loop -> SCREAM "N+1 Query Detected."
    3. **Context Check:** Check `org_policy.json`. If `scale == "Prototype"`, ignore minor inefficiencies. If `scale == "Enterprise"`, flag everything.
    4. **Report:** Focus on **Latency** (Time) and **Throughput** (Scale).

Here is the detailed design for **Agent C**, following the established structure.

Agent C: The Tech Lead ("The Clean Code Purist")

> *"I don't care how fast it is. I care if humans can read it."*

Role: Lead Developer / Code Maintainer.

Focus Areas:

- **Readability:** Variable and function naming (Intent-revealing names vs. `x`, `data`, `temp`).
- **Maintainability:** Single Responsibility Principle (SRP) violations (God Objects/Functions), high argument counts.
- **Code Rot:** Magic numbers (unexplained constants), dead code, "TODO" comments left in production.
- **DRY (Don't Repeat Yourself):** Copy-pasted logic blocks that should be refactored into a shared function.
- **Style Compliance:** Enforcing the organization's specific style guide (e.g., Google Style, PEP8) regarding formatting and docstrings.

Input:

The "Maintainability Payload" (Filtered for text and structure):

- **Target Code (All Text):**
  - *Included Extensions:* All source code (`.py`, `.js`, etc.).
  - *Excluded:* Minified files (e.g., `bundle.min.js`), auto-generated code, binary files.
- **Configuration Files:** `.eslintrc`, `pylintrc` (to understand existing rules).
- **Context:**
  - `org_policy.json`: **Crucial.** Defines `naming_convention` (camelCase vs snake_case), `max_function_lines`, and `require_docstrings`.
  - `README.md`: To check if the code actually matches the documentation.

Tools:

1. **`analyze_naming_conventions(code_content, language)`**
   - **Purpose:** Scans for variables/functions that violate the configured casing (e.g., detecting `camelCase` in Python) or are non-descriptive (e.g., single-letter variables like `x`, `i` outside of loops).
   - **Input:** `code_content` (str), `language` (str).
   - **Output:** JSON listing violations with line numbers and suggested renames (e.g., change `d` to `days_elapsed`).
2. **`measure_function_metrics(code_content)`**
   - **Purpose:** Calculates "Code Smell" metrics: Function Length (LOC) and Argument Count.
   - **Input:** `code_content` (str).
   - **Output:** JSON highlighting functions that are too long (>50 lines) or take too many arguments (>3), indicating a need for refactoring (SRP violation).
3. **`detect_code_duplication(code_content)`**
   - **Purpose:** Uses a hashing algorithm (like Rabin-Karp) to find blocks of code that are identical or nearly identical across the file(s).
   - **Input:** `code_content` (str).
   - **Output:** JSON listing duplicate blocks with line ranges, suggesting extraction to a utility function.
4. **`check_docstring_coverage(code_content)`**

    - **Purpose:** Verifies if public classes and functions have proper documentation strings (JSDoc/Docstrings) matching the `org_policy.json` requirement.
    - **Input:** `code_content` (str).
    - **Output:** JSON listing undocumented public interfaces.

ReAct (Reasoning + Acting) Strategy:

- **System Prompt:** You are the author of "Clean Code." You prioritize long-term maintenance over short-term hacks.
- **Logic:**
    1. **Check Policy:** Read `org_policy.json`. Is `strict_mode` on?
    2. **Scan Metrics:** Call `measure_function_metrics`. If a function takes 7 arguments -> Reject it ("Too complex").
    3. **Scan Names:** Call `analyze_naming_conventions`. If you see `var data`, ask "What data?".
    4. **Scan Duplication:** Call `detect_code_duplication`. If code is copied 3 times -> Suggest a refactor.
    5. **Report:** Focus on **Clarity** and **Future Cost**.

You have correctly identified the dual nature of the "Central Node." However, in a production system, these two roles are fundamentally different:

1. **The Router (Input Handler):** This should generally be **deterministic Python code**, not an AI. (It is faster and cheaper to use code to say "Send `.py` files to Agent A" than to pay an LLM to decide that).
2. **The Synthesizer (The Judge):** This **MUST be an AI Agent** (an LLM). Its job is nuanced: it has to understand context, tone, and resolve conflicts between the experts.

Central Node

Part 1: The Router (The Orchestration Logic)

This is the "Traffic Controller." It runs before the agents.

Role: The Dispatcher.

Mechanism: Pure Code (Python/LangGraph Logic), not an LLM.

Logic Definition:

1. **Ingest:** Receives the raw `user_submission` (Zip file or Repo URL) and `org_policy.json`.
2. **Filter & route (The "Split"):**

    - **Stream A (Security):** Payload = { Manifests, Config, Logic_Files, SQL_Files } -> Send to **Agent A**.
    - **Stream B (Architecture):** Payload = { Config, Logic_Files, Infra_Files (Docker/K8s) } -> Send to **Agent B**.
    - **Stream C (Maintainability):** Payload = { Config, Logic_Files (All text sources) } -> Send to **Agent C**.
3. **Parallel Execution:** Initiates the asynchronous calls to A, B, and C.

Part 2: The Synthesizer Agent ("The Judge")

This is the "Staff Engineer" who overrides the juniors.

Role: Senior Staff Engineer / Engineering Manager.

Goal: Provide a single, cohesive voice. The user should not feel like they are being yelled at by three different robots.

Input:

- **The 3 JSON Reports:** The outputs from Agent A, Agent B, and Agent C.
- **The Context:** org_policy.json (This contains the "Tie-Breaker" rules).
- **The Code Summary:** (Optional) A brief summary of what the code does.

Core Responsibilities:

1. **Deduplication:** If Agent A flags a "Database Error" on line 10, and Agent B flags "Inefficient Query" on line 10, the Judge merges them into one comment.
2. **Conflict Resolution:**
    - *Scenario:* Agent B (Speed) says "Use a bitwise shift." Agent C (Purist) says "Use readable math."
    - *Action:* The Judge looks at org_policy. If scale == "Startup", it side with Agent C (Readability). If scale == "HFT", it sides with Agent B (Speed).
3. **Tone Calibration:** Ensures the feedback is constructive (adding the praise section) rather than just a list of failures.
4. **Scoring:** Calculates the final quality_score (0-100) based on the severity of the surviving comments.

The "Prompt" Strategy:

This agent doesn't need "Tools." It needs a very strong **System Prompt**.

📝 Draft System Prompt for The Judge

Plaintext

```
You are the "Senior Staff Engineer" and Lead Reviewer.
You have received code review feedback from three specialized junior agents:
1. Security Agent (The Hawk) - Focused on risks.
2. Performance Agent (The Speed Demon) - Focused on efficiency.
```

3. Maintainability Agent (The Purist) - Focused on style and clarity.

Your Task:
Synthesize these three inputs into a SINGLE, professional Code Review JSON output.

Rules for Synthesis:
1. SECURITY TRUMPS ALL: If the Security Agent flags a "CRITICAL" issue, that is the most important thing. You must REQUEST_CHANGES.
2. RESOLVE CONFLICTS:
   - If Speed and Style disagree (e.g., Speed wants complex code, Style wants simple code), look at the 'org_policy'.
   - Default: Favor Readability unless 'org_policy.scale' is 'High_Performance'.
3. DEDUPLICATE: Do not list the same error twice. Combine them.
4. BE HUMAN: Generate a 'summary' that encourages the developer. Find 1-2 things to 'praise'.
5. CALCULATE SCORE: Start at 100. Deduct 10 for CRITICAL, 5 for HIGH, 2 for MEDIUM.

Input Data:
[JSON Data from Agents A, B, C]

Output Format:
Strict JSON matching the { review_id, summary, comments[], praise[], meta{} } schema.

The Conflict Resolution Matrix (Logic Table)

To make the Judge robust, we define a "Hierarchy of Needs" for it to follow.

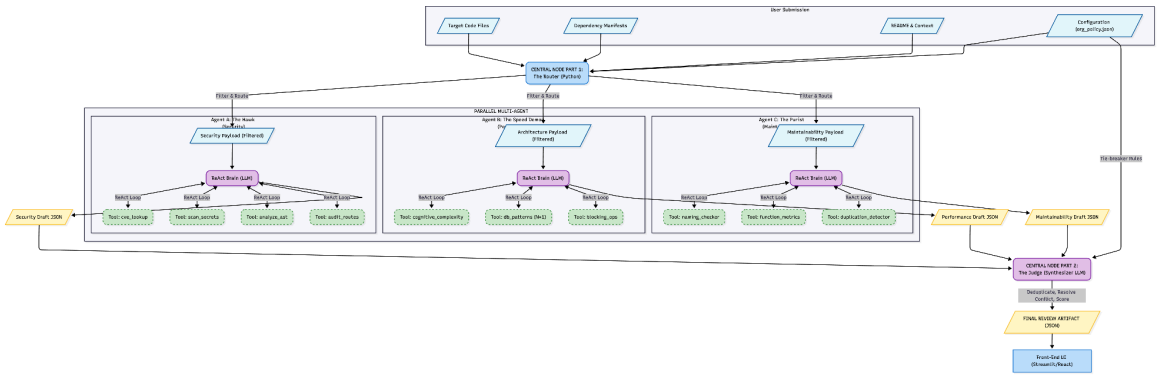
Priority	Agent	Rule
1 (Highest)	**Agent A (Security)**	Security flaws are non-negotiable. If Agent A screams, the PR is rejected.
2	**Config Policy**	If the user explicitly set `enforce_typing = True`, then Agent C's typing complaints overwrite Agent B's speed complaints.
3	**Agent B (Performance)**	*Only* if `scale` is "High". Otherwise, this is lower priority than readability.

4 (Lowest)	**Agent C (Style)**	"Nitpicks" (indentation, naming) are dropped if they conflict with significant performance gains in a High-Scale system.

Visualizing the Flow

1. **Router:** Splits the Zip file.
2. **Agents:** Run in parallel.
3. **Judge:**
    - Receives: [Security: 2 Critical], [Speed: 1 Warn], [Style: 5 Nitpicks].
    - Logic: "Security is Critical -> Verdict: REQUEST_CHANGES. The Style nitpicks are annoying, I'll filter out the minor ones to keep the review focused."
    - Output: Final JSON.

System flow Diagram



It is a Fan-in Fan-out Multi Agent system with agents working in parallel

File Structure

---

1. Project File Structure

Plaintext

```
ai-code-reviewer/
├── .env                    # Environment variables (API Keys, DB Config)
├── .gitignore
├── README.md
├── requirements.txt
├── docker-compose.yml
```

```
│
├── config/                # Central Configuration
│   ├── __init__.py
│   ├── settings.py        # Loads .env, defines app constants
│   └── org_policy.json    # The "Ruleset" for the agents
│
├── src/
│   ├── __init__.py
│   ├── main.py            # Orchestrator Entry point
│   │
│   ├── schemas/           # [NEW] Global Pydantic Models
│   │   ├── __init__.py
│   │   ├── common.py      # Shared objects (e.g., File, Project, Config)
│   │   └── responses.py   # Standard API response wrappers
│   │
│   ├── core/              # Core Logic
│   │   ├── __init__.py
│   │   ├── orchestrator.py
│   │   ├── state.py       # LangGraph State definitions
│   │   └── llm.py         # Mistral Client wrapper
│   │
│   ├── agents/            # Specialized Modules
│   │   ├── __init__.py
│   │   ├── base_agent.py  # Abstract Base Class
│   │   │
│   │   ├── security/      # Agent A
│   │   │   ├── __init__.py
│   │   │   ├── agent.py
│   │   │   ├── tools.py
│   │   │   ├── schemas.py  # [NEW] Agent-Specific Models (e.g., CVEResult)
│   │   │   └── api.py
│   │   │
│   │   ├── performance/   # Agent B
│   │   │   ├── __init__.py
│   │   │   ├── agent.py
│   │   │   ├── tools.py
│   │   │   ├── schemas.py  # [NEW] (e.g., ComplexityScore)
│   │   │   └── api.py
│   │   │
│   │   ├── maintainability/# Agent C
│   │   │   ├── __init__.py
│   │   │   ├── agent.py
│   │   │   ├── tools.py
│   │   │   ├── schemas.py  # [NEW] (e.g., LintViolation)
│   │   │   └── api.py
│   │   │
│   │   └── judge/         # The Synthesizer
│   │       ├── __init__.py
│   │       ├── agent.py
│   │       ├── schemas.py  # [NEW] (e.g., FinalVerdict)
```

```
│   │       └── api.py
│   │
│   ├── utils/              # Shared Helpers
│   │   ├── __init__.py
│   │   ├── logger.py       # [CRITICAL] Standardized Logger
│   │   └── file_parser.py
│   │
│   └── api/                # Main API Gateway
│       ├── __init__.py
│       ├── main.py
│       └── routes.py
│
├── frontend/
│   ├── app.py
│   └── components/
│
└── tests/
    ├── agents/
    └── integration/
```

Coding Standards & Protocols

Adopting a **Modular Sprint-Based** approach with strict **OOP principles** is exactly how you build complex systems like this without creating "Spaghetti Code."

Here is your **Project Blueprint**, designed for modularity, testability, and speed.

To ensure we can plug-and-play modules, we need strict interfaces.

1. General Style Guide

- **Standard: PEP 8** (The Python Standard).
- **Linter:** We will use `ruff` or `black` to enforce this automatically.
- **Docstrings: Google Style** docstrings. Every public function must have one.

2. Naming Conventions

Element	Case Style	Example
Classes	`PascalCase`	`SecurityAgent`, `CodeReviewRequest`
Functions/Methods	`snake_case`	`calculate_complexity()`, `scan_cve()`

Variables	snake_case	user_input, db_connection
Constants	UPPER_CASE	MAX_RETRIES, DEFAULT_MODEL
Private Members	_snake_case	_connect_to_db(), _internal_state
Modules/Files	snake_case	agent_manager.py, utils.py

3. Typing & Objects (Strict Portability)

We will use **Python Type Hints** everywhere. No Any unless absolutely necessary.

The Golden Rule: All data passed *between* agents or *to* the API must be **Pydantic Models**, not Dictionaries.

- *Bad (Dictionary hell):*
- Python

```
def analyze(code):
    return {"score": 10, "issues": []}
```

-
-
- *Good (Strict Contract):*
- Python

```
from pydantic import BaseModel

class ReviewResult(BaseModel):
    score: int
    issues: list[Issue]

def analyze(code: str) -> ReviewResult:
    ...
```

-
-

4. Function Signatures

Every function signature must include type hints for arguments and return values.

```python
# Correct Signature
def scan_secrets(content: str, strict_mode: bool = False) -> List[SecretFound]:
    """
    Scans the provided content for high-entropy strings.

    Args:
        content (str): The raw code content.
        strict_mode (bool): If True, uses aggressive regex.

    Returns:
        List[SecretFound]: A list of detected secrets.
    """
    pass
```

5. Object-Oriented Structure (The BaseAgent)

To implement the "Modular/Portable" requirement, we define an Interface that all agents must inherit from.

```python
# src/agents/base_agent.py
from abc import ABC, abstractmethod
from src.core.state import AgentState

class BaseAgent(ABC):
    def __init__(self, model_client):
        self.llm = model_client

    @abstractmethod
    def run(self, state: AgentState) -> AgentState:
        """
        Core logic for the agent.
```

```
    Takes the current state, performs analysis, and returns updated state.
    """
    pass

@abstractmethod
def get_tools(self) -> list:
    """Returns the list of tools available to this agent."""
    pass
```

6. Dependency Management (Imports)

- **Absolute Imports:** Always use absolute pathing to avoid "Module Not Found" errors when moving files.
    - *Yes:* `from src.agents.security.tools import cve_lookup`
    - *No:* `from ..tools import cve_lookup`

7. Logging Protocol

We use a centralized logger from `src.utils.logger`. **Do NOT** use Python's default `print()`.

- **Format:** `[TIMESTAMP] [LEVEL] [MODULE]: Message {metadata}`
- **Levels:**
    - `DEBUG`: Variable states, raw LLM inputs/outputs (Development only).
    - `INFO`: High-level flow events (e.g., "Agent A started", "File received").
    - `WARNING`: Handled issues (e.g., "Tool execution failed, retrying").
    - `ERROR`: Unhandled exceptions or critical failures blocking a workflow.
- **Implementation:**
- Python

```
from src.utils.logger import get_logger
logger = get_logger(__name__)

logger.info("Starting security scan", extra={"file": filename})
```

-
-

8. Exception Handling Protocol

Errors must be caught at the **lowest possible level** (e.g., inside the Tool) and bubbled up as meaningful data, not raw crashes.

- **Rule 1:** Never use bare `except:` clauses. Always catch specific exceptions.
- **Rule 2:** In Agents, tool failures should **not** crash the agent. They should return a "ToolFailure" state so the LLM knows what happened.

- **Rule 3:** The API layer must catch all bubbling exceptions and return a `500 Internal Server Error` with a standardized JSON error response.
- Python

```
try:
    result = run_tool()
except ToolTimeoutError as e:
    logger.warning(f"Tool timed out: {e}")
    return ToolResult(success=False, error="Timeout")
except Exception as e:
    logger.exception("Unexpected error in tool execution")
    raise e  # Bubble up if critical
```

-
-

9. Constants & Configuration

Hardcoding values (magic numbers/strings) is strictly forbidden.

- **Environment Variables:** Secrets (API Keys, DB URLs) go in `.env`.
- **Project Constants:** Non-secret settings (Timeouts, Default Model Name) go in `config/settings.py`.
- **Agent Constants:** Agent-specific prompts or thresholds go in the agent's `__init__.py` or a dedicated `constants.py` inside the agent folder.
- **Naming:** All constants must be `UPPER_CASE_WITH_UNDERSCORES`.

D. Documentation Standards

Every module, class, and public function must have a Google-style docstring.

- **Format:**
- Python

```
def calculate_risk_score(vuln_count: int, severity: str) -> float:
    """
    Calculates the normalized risk score based on vulnerability data.

    Args:
        vuln_count (int): Total number of CVEs found.
        severity (str): Highest severity level (LOW, MEDIUM, HIGH).

    Returns:
        float: A score between 0.0 and 100.0.

    Raises:
        ValueError: If severity is not a valid enum member.
    """
    ...
```

-

Project Phase Roadmap

Each phase typically corresponds to one or more sprints.

Phase	Title	Goal	Key Deliverables
Phase 0	**Infrastructure Setup**	Establish the foundation so agents can be built in parallel later.	• Project Repo & Directory Structure • Docker/Env Setup • Base Classes (`BaseAgent`, `BaseTool`) • Logging & Config Modules • Hello World API + Streamlit connection

Phase 1	**The Security Agent**	Build & Test "The Hawk" (Agent A) in isolation.	• `cve_lookup` `scan_secrets`,`analyze_ast_patterns`,`audit_route_permissions` tools • Security ReAct Loop (LangGraph) • Unit Tests for Security Tools • UI Tab: "Security Audit"
Phase 2	**The Performance Agent**	Build & Test "The Speed Demon" (Agent B).	• `calculate_cognitive_complexity`,`scan_database_patterns`,`detect_blocking_ops`, tools • Performance ReAct Loop • AST Parsing Logic

			• UI Tab: "Performance Review"
Phase 3	**The Maintainability Agent**	Build & Test "The Purist" (Agent C).	• linting & naming tools • Maintainability ReAct Loop • UI Tab: "Code Quality"
Phase 4	**The Judge & Integration**	Connect all agents and build the Synthesis layer.	• The Judge Agent (Synthesizer) • Orchestrator Logic (**Fan-Out/Fan-In**) • Conflict Resolution Logic • Final "Summary" UI View

Tech Stack

This is the "Engine Room" view of your project. By defining the exact libraries for every single tool, you remove ambiguity and can start coding immediately.

Here is your updated **Tech Stack**, organized **Bottom-Up** from the database to the pixels on the screen.

The "War Room" Tech Stack (Bottom-Up)

Layer	Technology	Specific Tool / Library	Role & Purpose
5. User Interface (Presentation)	**Web Framework**	**Streamlit**	The Dashboard. Renders the split-screen comparison, handles file uploads, and visualizes the "War Room" status with real-time spinners.
4. API Gateway (Communication)	**Backend Framework**	**Flask**	The "Headless" Engine. Exposes `POST /review` endpoints. Decouples the UI from logic, making the system modular and testable.
3. Orchestration (Workflow)	**State Machine**	**LangGraph**	The "Traffic Controller." Manages the Fan-Out/Fan-In logic, ensuring Agents A, B, and C run in parallel and the Judge waits for synthesis.
2. Intelligence (The Brains)	**LLM Provider**	**Mistral AI (`codestral-latest`)**	The Reasoning Engine. Specifically the `codestral` model, fine-tuned for code analysis. Used by agents to generate fixes and explain "Why".

2.1 Agent A Tools (Security)	**Static Analysis**	**bandit**, **safety**	**bandit**: Scans AST for security flaws (e.g., eval, exec). **safety**: Scans requirements.txt for known CVEs.
	Regex / AST	**re**, **ast**	**re**: Custom high-entropy regex for scan_secrets. **ast**: Custom parsers to audit API route decorators (audit_route_permissions).
2.1 Agent B Tools (Performance)	**Code Metrics**	**radon**	**radon**: Calculates Cyclomatic Complexity scores for calculate_cognitive_complexity.
	Pattern Matching	**ast (Built-in)**	**ast**: Custom logic to detect "N+1" queries (scan_database_patterns) and blocking calls in async functions.
2.1 Agent C Tools (Maintainability)	**Linting / Style**	**pylint**, **pydocstyle**	**pylint**: Checks naming conventions (analyze_naming_conventions). **pydocstyle**: Verifies docstring coverage (check_docstring_coverage).

	Hashing	`hashlib`	`hashlib`: Implements Rabin-Karp hashing for `detect_code_duplication`.
1. Foundation (Infrastructure)	**Language**	**Python 3.10+**	The core programming language.
	Validation	**Pydantic**	Enforces strict data contracts (Input/Output Schemas) between Agents and API.
	Environment	**Docker**	Containerizes the API to ensure "It runs on my machine" applies to the judges too.

Tool Implementation Map (The "How-To")

To save you research time, here is exactly how to map your defined **Agent Tools** to these Python libraries.

Agent A: The Security Hawk

- **`cve_lookup`** → **Library:** `safety`
  - *Implementation:* Run `safety check -r requirements.txt --json` programmatically.
- **`scan_secrets`** → **Library:** `re` (Regex)
  - *Implementation:* Use pre-defined patterns (e.g., `(?i)api_key\s*=\s*['"][a-zA-Z0-9]{32,}['"]`).
- **`analyze_ast_patterns`** → **Library:** `bandit`
  - *Implementation:* Use `bandit` as a library (bypassing CLI) to scan specific file strings for issues like `B601` (shell injection).
- **`audit_route_permissions`** → **Library:** `ast`
  - *Implementation:* Walk the AST, find `FunctionDef` nodes with `@app.route` decorators, and check if `@login_required` is missing.

Agent B: The Speed Demon

- **`calculate_cognitive_complexity`** → **Library:** `radon`

    - *Implementation:* radon.complexity.cc_visit(code_string). Returns a score (1-10 is simple, 10+ is complex).
- **scan_database_patterns** → **Library:** ast
    - *Implementation:* Walk AST. If inside a For node → check for Call nodes matching execute, query, or commit.
- **detect_blocking_ops** → **Library:** ast
    - *Implementation:* If inside AsyncFunctionDef → check for Call nodes matching time.sleep or requests.get.

Agent C: The Clean Code Purist

- **analyze_naming_conventions** → **Library:** pylint
    - *Implementation:* Run pylint --disable=all --enable=C0103 (naming convention code) on the file.
- **measure_function_metrics** → **Library:** radon
    - *Implementation:* radon.raw.analyze(code_string) gives you LOC (Lines of Code) and SLOC.
- **detect_code_duplication** → **Library:** hashlib
    - *Implementation:* Split code into 6-line chunks, hash them, and look for collisions. (Faster than running a heavy tool like cpd).