ChatGPT

1. ARCHITECTURAL DOCUMENT SET

- **System Architecture Document (SAD)**
- **Purpose & Scope:** Describes the overall system context, high-level design, and how Elspeth fits into the enterprise ecosystem. It covers the core modules (CLI, core runtime, plugin layer, external integrations) and how data flows from ingestion to results [1].
- **Key Findings:** Elspeth is a **single-process orchestration framework** (invoked via CLI) that runs experiments on tabular data using LLMs. It follows a **modular plugin architecture** – the orchestrator dynamically wires together a DataSource, LLM client, and one or more result sinks per experiment [2] [3]. The design emphasizes **security by design** (e.g. enforcing data classification labels, artifact signing, and output sanitization) and **composability** (swappable plugins for I/O, LLM backends, metrics, etc.).

- **Critical Issues:** No critical design gaps in overall architecture. However, the presence of a legacy codebase (the `old/` directory) not used by the main system could confuse maintainers or pose an **accidental execution risk** if not clearly segregated or removed. Also, clarity is needed on deployment context – as a CLI tool, integration into enterprise workflows (e.g. pipeline automation or multi-user environments) requires additional controls (auth, logging, etc.) beyond what a standalone tool provides.

- **Technical Architecture Overview**

- **Purpose & Scope:** Details the specific technologies, frameworks, and languages used, and how the code is structured into packages and modules. It identifies major components and their interactions (following a C4 model at Container/Component level).
- **Key Findings:** The project is implemented in **Python 3.12** [4], with a well-defined package structure under `src/elspeth`. Key sub-packages include `core` (orchestration logic, security, controls), `plugins` (various plugin implementations), and `tools` (reporting utilities). The architecture is **monolithic** (all components run in-process), using threads for concurrency when needed [5] [6]. Major third-party libraries include **Pandas** for data handling, **Requests** for HTTP calls (to LLM APIs) [7], and **OpenPyXL** (optional) for Excel report generation [8]. Dependencies are managed via a `pyproject.toml` with extras for dev and visualization; the build uses `make` for bootstrapping and testing.

- **Critical Issues:** Overall code quality is high (extensive typing, linting, and an automated test suite with dozens of tests [9]), but a few technical concerns exist. There is **trace legacy naming** (e.g. references to `DMP` in env vars or comments) [10] which could confuse new developers. Also, partial refactoring is in progress – e.g. the plugin registry has new "nodes/transforms" structure emerging – which means both old and new patterns coexist. This could increase maintenance complexity until fully unified.

- **Data Architecture & Flow Diagrams**

- **Purpose & Scope:** Illustrates how data moves through the system – from input data ingestion, through processing (prompts, LLM calls, plugins) and into output artifacts. Includes data models (e.g. experiment result schema) and data storage considerations.

- **Key Findings: Data flow is batch-oriented**: a DataSource plugin loads input data into a Pandas DataFrame [11], which the Orchestrator passes to the ExperimentRunner. Each row is transformed into prompts and sent to the LLM, producing responses that are collected along with metrics [12] [13]. The runner accumulates results in memory and then feeds them into an Artifact Pipeline which invokes each configured ResultSink [14]. Output data artifacts include JSON/CSV results, Excel reports, charts, and signed bundles. The data model for results is basically a mapping with keys `results` (list of per-row outputs), optional `aggregates`, `metrics`, etc., as seen in the JSON outputs. Each result record includes the original input ("row"), the LLM response(s), metrics, and metadata such as security level [15]. **Data schemas** are enforced where possible (e.g. optional schema validation of DataFrame columns against plugin expectations [16] [17]).

- **Critical Issues:** The in-memory DataFrame approach could become a bottleneck for very large datasets (no streaming of input yet – a known roadmap item). This poses **scalability limits** and potential memory exhaustion for huge inputs. Also, while data classification tags are tracked in memory and in metadata, the system does not encrypt or segregate data based on classification – relying on procedural controls for now. For ATO, consider whether PROTECTED/SECRET data processed in-memory and output to local files meets data handling rules or if additional safeguards (like encryption at rest) are needed.

- **Security Architecture Assessment**

- **Purpose & Scope:** Evaluates the security controls in the code: authentication, authorization, data protection (encryption, sanitization), secret management, auditing, and compliance with relevant security standards (ISM, ASD Essential Eight). Includes threat model considerations and how the architecture mitigates risks.
- **Key Findings:** Security is a strong focus of Elspeth's design. **Classification labels** are integrated into the core: each data source and LLM client can declare a security level, and the orchestrator computes the overall classification (e.g. OFFICIAL, PROTECTED) for the experiment [18]. These levels align with the Australian Government PSPF hierarchy (UNOFFICIAL up to SECRET) [19] and are propagated to outputs (each artifact can carry a `security_level` tag [20] [21]). The system enforces **content security** in multiple ways: it sanitizes spreadsheet formula inputs to prevent injection attacks [22], and includes LLM middleware for content filtering (e.g. `PromptShieldMiddleware` to block disallowed terms, configurable to abort or mask the prompt) [23]. An **audit logging middleware** records all LLM requests and responses (with toggle to include prompt content) to a secure log channel [24] [25], supporting traceability. All experiment outputs can be **digitally signed** using HMAC with a user-provided key: the SignedArtifact sink produces a JSON signature and manifest for results [26] [27], ensuring integrity and non-tampering of exported data. Secrets (like API keys for LLM services or signing keys) are not hard-coded; they are passed via config or environment variables (e.g. OpenAI API key via env var, signing key via `ELSPETH_SIGNING_KEY` env) [28] [29]. This aligns with Essential Eight control on secure config and secrets management. The code also implements request **rate limiting** and **cost tracking** to prevent misuse or cost overrun: RateLimiter plugins (fixed-window or adaptive) throttle LLM calls to configured rates [30] [31], and CostTracker plugins aggregate token usage costs per run [32] [33] (useful for monitoring and enforcing cost policies).

- **Critical Issues:** No major security holes were found in code. One area to monitor is **audit log sensitivity** – if `include_prompts` is enabled for audit logging, sensitive input data could be stored in logs. By default it's off, but governance should ensure logs are protected and that any personal or secret data in prompts is handled per policy. Another consideration is that while outputs are signed and classified, they are **not encrypted**; if an experiment produces sensitive

data (e.g. SECRET content), the burden is on operational controls to store those files securely. For ATO, it may be recommended to add an option to encrypt high-classification artifacts at rest (or integrate with an accredited storage service). Additionally, the use of third-party APIs (OpenAI/Azure) means data leaves the system; ensuring those integrations are approved services and that data sent to them is appropriately sanitized (the PromptShield and content safety middleware mitigate this) is crucial.

- **Integration Architecture & API Documentation**

- **Purpose & Scope:** Describes how Elspeth integrates with external systems and what APIs or extension points it provides. This includes external LLM services, data sources (files, blob storage), and how new plugins can be added by developers. It should also cover any APIs the system itself exposes (e.g. CLI commands, library calls) for integration into pipelines.
- **Key Findings:** Elspeth is **highly extensible via plugins**. External integration is done by implementing the plugin protocols – e.g., to use a new LLM service, one writes an `LLMClientProtocol` implementation and registers it in the plugin registry. The system provides built-in adapters for OpenAI's HTTP API [7], Azure OpenAI, and a mock LLM for offline use. Similarly, data ingestion can come from local CSV files, Azure blob storage, etc., via DataSource plugins (the sample uses CSV) and output sinks can target files, databases, or repositories. The integration with external infrastructure is intentionally abstracted: rather than direct coupling, the orchestrator only knows about the Protocol interfaces [11] [34], and plugins encapsulate the specifics of calling external APIs or services. For example, the OpenAI HTTP client plugin takes an `api_base` URL and API key, and simply posts to the `/v1/chat/completions` endpoint [35]. Likewise, the Azure OpenAI plugin uses Azure endpoints and keys. This design makes it easy to slot Elspeth into different environments – e.g., pointing to a government-authorized LLM endpoint vs. the public cloud – by configuration, not code changes. **Integration points** also include the **artifact pipeline**: a Repository sink is provided to push results to a Git repository or Azure DevOps (though in the current code, the Repository sink appears to operate in a dry-run mode unless `--live-outputs` is set, per CLI). For pipeline automation, the CLI can be invoked in batch jobs, and results (especially the JSON/Excel reports) can feed into downstream analytics or dashboards. There is no long-running server component; integration is typically via CLI execution or by importing Elspeth as a library in a Python script.
- **Critical Issues:** The integration design is flexible but **lacks strong enforcement of output destinations** in code – by default, many sinks (like Repository or blob outputs) run in a "dry" mode unless explicitly allowed, which is good for safety, but it relies on the operator to correctly enable and configure them. Documentation for plugin developers exists, but for ATO we must ensure that any external API usage (OpenAI, etc.) is vetted by compliance (e.g., no sensitive data is sent to unapproved cloud APIs). If certain integrations are not permitted (e.g. using public OpenAI API for PROTECTED data), those must be disabled or replaced with compliant alternatives. The architecture should document these decisions in the **Architectural Decisions Record (ADR)** – e.g., choice to allow/disable certain plugin classes in high-security deployments.

- **Infrastructure & Deployment Architecture**

- **Purpose & Scope:** Describes how and where the system is deployed: network diagrams, infrastructure components (servers, containers, etc.), and runtime environment configuration. Even if it's a CLI tool, for ATO we consider how it will run in production (e.g. on a secure server, in a CI pipeline, or an analyst's workstation) and what infrastructure is needed (database, message queues, etc. – in this case none beyond ephemeral files and optional blob storage).

- **Key Findings:** Elspeth is delivered as a **Python package/CLI** and does not require persistent infrastructure like databases or web servers. Deployment can be as simple as running it on a hardened VM or container in the target environment. The only external services it interacts with are those configured via plugins (e.g., Azure Blob Storage for data, LLM APIs, etc.). Thus, the infrastructure architecture is lightweight: essentially, an **operator's machine or a batch execution environment** with network access to the needed APIs. From an environment hardening perspective, the documentation emphasizes running Elspeth in a controlled environment (the docs mention guidance for deployment hardening [36]). Key considerations include securing the environment variables (for keys), ensuring the host has the required Python version, and possibly containerizing the app for consistency. The system creates output files in specified directories; these should reside on approved storage (e.g., an encrypted volume or secure file store). For concurrency, Elspeth uses threads internally, so scaling is vertical (within one machine); if higher throughput is needed, one could run multiple instances in parallel on separate data partitions.

- **Critical Issues:** Because Elspeth runs with the privileges of whoever invokes it, **operator workstation security** is crucial. If run on a standard user's machine, that machine becomes part of the accredited boundary. There is no built-in multi-user or network service component to secure, which simplifies ATO concerns, but it shifts focus to endpoint security (ensuring the system is run in an environment meeting ASD Essential Eight baseline: patched OS, application control in place – e.g., only this approved code runs – and so on). We should also explicitly define the deployment model: for example, if it's to be used in an internal research environment vs. in production automation, as this affects how tightly it must be locked down. Currently, no container image is provided, but creating a Docker image with only necessary components could ease deployment and ensure consistency across environments.

- **Component Dependency Analysis**

- **Purpose & Scope:** Catalogs the internal and external dependencies between components. Which modules depend on which, and what external libraries are used. This is important for understanding blast radius of changes and supply chain risk (third-party libraries).
- **Key Findings:** Internally, the code is logically decoupled through interfaces and registry lookups. For instance, the Orchestrator depends on the presence of plugins but not their concrete classes – it uses the plugin registry to create instances by name [37]. The core package has low-level dependencies (e.g., `core.security` and `core.types` are used widely for classification constants, `core.validation` for config checks). Most cross-module interactions are via clearly defined data structures (dataclasses like `OrchestratorConfig` [38], `ExperimentContext`, and plugin Protocols [11] [34]). External library dependencies include: **PyYAML** (for config parsing) [39] [40], **Pandas** (for DataFrame operations) [41], **Requests** (HTTP calls) [42], **OpenPyXL** (Excel output) [8], and possibly Jinja2 (the prompt templating engine, referred to as `PromptEngine` – likely using Jinja under the hood). The project pins or references versions for these in its configuration to avoid compatibility issues. The test suite and dev tooling add further dependencies (pytest, linting tools, etc., for development only). There are no database engines or web frameworks in use – simplifying dependency scope.

- **Critical Issues:** From a **supply-chain security** perspective, each third-party package should be kept updated to patch vulnerabilities (this aligns with Essential Eight patching requirements). For example, if a vulnerability in `requests` or `openpyxl` arises, the team must promptly update those. A **dependency analysis document** exists (and should be referenced in the ATO) enumerating these libraries and any known risks (e.g., ensure no banned licenses or known CVEs in current versions). Internally, one minor issue is that certain utility modules (like

`elspeth.core.processing` or `tools.reporting`) serve cross-cutting purposes and if changed, could affect multiple plugins – these should be carefully managed but currently pose no immediate risk.

- **Technology Stack Assessment**

- **Purpose & Scope:** Provides rationale for the chosen tech stack and evaluates its suitability and compliance. Confirms that the programming language, frameworks, and tools meet enterprise standards and that the system can be maintained long-term.
- **Key Findings:** The technology stack is **Python-based**, which is common and acceptable in the enterprise (provided that runtime is kept updated for security). Python 3.12 is used, which as of 2025 is a modern version with ongoing support. The decision to build a CLI tool with plugins (versus a web service or GUI) is deliberate for simplicity and security – it minimizes the attack surface (no open ports or complex web frameworks to secure) and aligns with a "tools for analysts" paradigm. The stack leverages **proven libraries** (Pandas, Requests, etc.) and avoids overly exotic tech, which bodes well for maintainability. The use of dataclasses, type hints, and extensive tests indicates a mature engineering approach. The architecture is also cloud-agnostic: aside from needing a Python environment, it does not lock in any proprietary platform. This flexibility allows it to run on Windows, Linux, or in cloud VM/container equally (though most testing likely on Linux).

- **Critical Issues:** Ensure that the Python environment is configured securely in production (e.g., use of a virtual env or container to avoid dependency conflicts, and no unnecessary packages installed). One Essential Eight consideration is **application control** – if this is to be run on a secure workstation, that workstation's allowed-software list must include this tool and its Python interpreter. The stack itself is fine, but operations should document the installation procedure (the `make bootstrap` provides a script for that). There's also a note that openpyxl is an optional dependency loaded at runtime [8]; if Excel reports are needed, that must be pre-installed, otherwise the system will throw runtime errors. This should be accounted for in deployment (perhaps making those extras part of the standard install).

- **Architectural Decisions Record (ADR)**

- **Purpose & Scope:** Summarizes key design decisions and the reasoning behind them. For ATO, this provides context to reviewers on why certain approaches were taken (e.g., why a plugin architecture, why Python, how security was baked in, etc.), showing that due diligence was done.
- **Key Findings:** While we don't have a separate ADR file in the repository, the code and notes reveal several critical decisions:
    - **Monolith + Plugins vs Microservices:** Chose an in-process plugin model for simplicity, performance, and easier compliance (no network communication between components to secure beyond LLM API calls). This avoids the complexity of securing multiple services – a conscious decision likely to meet tight experimentation feedback loops and reduce dev overhead.
    - **Stateless CLI vs Persistent Service:** Decided to implement as a CLI tool, meaning each run is ephemeral. This simplifies state management and cleanup (each run starts fresh) and avoids server-side persistence of sensitive data – aligning with principle of least data retention. The trade-off is lack of multi-user concurrency, which was deemed acceptable for the intended use (experimentation by individuals or batch jobs).
    - **Use of Pandas DataFrames:** To leverage its robust CSV handling and familiarity within data science teams, at the cost of memory usage. Future ADRs may consider migrating to

streaming for large data, but initially the team prioritized ease of use and existing Pandas ecosystem tools.
    - **Security-by-Design features:** Decisions like including security level tagging, HMAC signing of outputs, and sanitization by default were made early (as evidenced by these features being thoroughly implemented in v1). This indicates a proactive stance on compliance – an ADR could note that meeting PSPF requirements and audit traceability was a driving requirement from the start.
    - **External API usage:** There was an implicit decision to allow use of external LLM services (OpenAI/Azure). It was mitigated by providing thorough logging and content controls, but it's a notable decision given sensitive contexts. Documentation (or an ADR) should capture this and likely restrict usage to approved endpoints in high-security deployments.

- **Critical Issues:** Ensure any **major changes in the refactoring roadmap** also have recorded decisions. For instance, a planned migration to a "data flow orchestration model" is mentioned in docs, altering internal structure – the rationale and expected benefits (e.g., streaming) should be captured for future reference. Lack of a formal ADR file means institutional knowledge could be lost; it's recommended to maintain an ADR log going forward for transparency.

- **Risk Assessment (Architecture Perspective)**

- **Purpose & Scope:** Identifies potential risks inherent in the architecture and how they have been mitigated or will be addressed. This covers security risks, technical debt, performance, and operational risks.
- **Key Findings:** The architecture addresses many risks proactively (e.g., output sanitization mitigates injection attacks; classification tagging mitigates mishandling of sensitive data; comprehensive testing mitigates regressions). Nonetheless, some **risks remain**: e.g., **Legacy code debt** – the included `old/` code poses risk if inadvertently used or if it diverges and introduces confusion; **Scalability risk** – if usage grows (bigger datasets or many concurrent runs), the current design might not scale horizontally without modification; **Dependency risks** – reliance on external services for critical functions (if OpenAI API changes or experiences outage, experiments fail; this is partially mitigated by the retry logic and the ability to switch to a mock LLM for testing). From a security viewpoint, threat modeling would highlight the **LLM invocation as a potential attack surface** (an attacker could attempt prompt injections or malicious inputs to trick the system). The architecture's countermeasures (PromptShield, content safety checks) help here, but their effectiveness depends on configuration. Operationally, lack of persistent state means lower risk of long-term data leakage, but also means any failure in a run could lose data unless outputs were written or checkpointing was enabled (the system does support checkpointing partial results to resume [43] [44], reducing risk of data loss on interruption).

- **Critical Issues:** No showstopper risks from architecture alone, but a few medium risks (as mentioned) that need active management. These will be detailed in the Risk Register section. Notably, **misconfiguration** is a concern: because of the flexibility, a user could disable safety features (e.g. turn off sanitization or content filters) – procedures/policies must ensure default secure configurations are used in production. Additionally, any use of **AI services brings compliance risk** – the architecture must be deployed in a way that only approved models/datasets are used, or the tool could be misused (this is more of an operational policy risk than an architectural flaw).

- **Scalability & Performance Architecture**

- **Purpose & Scope:** Examines how the system handles load and how it can scale. Identifies any performance bottlenecks in code and strategies for horizontal or vertical scaling.
- **Key Findings:** Elspeth is optimized for experimentation convenience over raw performance, but it includes features to improve throughput: e.g., **multi-threaded LLM calls** – the ExperimentRunner will parallelize requests if enabled in config and if the input size is above a threshold [45], using Python threads to overlap I/O-bound LLM API calls. This can significantly speed up processing of large input sets (subject to API rate limits). The main bottleneck is likely the LLM API latency itself; the code has minimal overhead around it (prompt rendering and Pandas operations are relatively fast vs network calls). Memory could become a bottleneck with very large DataFrames – all input is loaded at once and results are accumulated in dictionaries before writing out. There is a checkpoint mechanism to alleviate long runs by saving progress [43], but it doesn't reduce memory use, only helps with resumability. Horizontal scaling (distributing different experiments or parts of data to multiple machines) is not directly supported by the tool itself, but could be achieved externally (e.g., splitting input and running multiple instances of Elspeth in parallel, then combining results). For typical usage (experiments on tens of thousands of rows or fewer), this architecture is sufficient. The overhead of plugin architecture is low – registry lookups and Python function calls – not a concern unless at extreme scale.

- **Critical Issues: Performance under extreme load** is not fully proven. If an experiment tries to, say, run on millions of records or do complex baseline comparisons with multiple LLM calls per row, the current approach may become slow or memory-heavy. There is also potential Python GIL contention if heavy CPU-bound operations were added in plugins – currently most work is I/O-bound (network calls, file I/O) so threads are effective. If usage grows, an architectural re-evaluation might be needed (the team is considering a dataflow model which likely addresses streaming). For now, no immediate action is needed for ATO as long as usage parameters are understood. However, it is worth noting in ATO documentation what the **tested capacity** of the system is (e.g., has it been benchmarked on X records or Y concurrent calls?) so the Authority knows its limits.

- **Disaster Recovery & Business Continuity Design**

- **Purpose & Scope:** Although typically DR/BCP applies to services, we interpret it here as how the system ensures resilience and how one would recover from failures or continue operations if something goes wrong. For a tool like Elspeth, this includes how runs can be resumed, how outputs are safely stored, and how to backup any critical data (configurations or results).
- **Key Findings:** Elspeth runs are stateless and idempotent given the same inputs. **Recovery from failures** is facilitated by the checkpoint feature: if enabled, each processed record ID is appended to a checkpoint file, so if a run crashes or is stopped, it can skip already processed rows on rerun [43] [46]. This is important for long-running suites so you don't lose all progress on an interruption – a notable design choice for continuity. Configuration files (YAML suite definitions) represent the "experiments to run" and can be version-controlled (they even provide a repository sink to archive results/config together). For outputs, since they are written to the file system (or blob storage), standard backup processes apply – the tool doesn't itself duplicate or backup outputs, that's left to the environment. However, by producing signed manifests and detailed reports, it ensures that if outputs are moved or archived, their integrity and provenance are verifiable. In terms of **business continuity**, if Elspeth became unavailable (e.g., bug or environment issue), teams could revert to manual experimentation or an older version due to its self-contained nature. Also, because it's a client-side tool, an outage of external LLM services is a bigger continuity risk – the team should plan fallback (the design allows plugging in a different

LLM or using the mock for non-production, but if in production with a specific API, continuity depends on that API's availability).

- **Critical Issues:** Being a tool rather than a service, DR in the classical sense (e.g., multi-site failover) isn't applicable, but from an ATO perspective, we should ensure that **experiment configurations and results are backed up** as needed. One risk: if an experiment is running and the machine is lost (hardware failure), any in-memory results are gone – the checkpoint mitigates this partially. It might be worth recommending more granular checkpointing or periodic flush of partial results to disk for very critical long experiments. Another consideration is personnel continuity: only a small team may know how to operate/maintain this tool; documentation and knowledge transfer are key so that capability isn't lost if key personnel are unavailable. This falls under BCP – ensuring more than one person can run and interpret Elspeth's outputs in the future.

Each of these documents will be prepared with the above findings, including relevant diagrams (e.g. C4 context and data flow from the mermaid diagrams), and will highlight how Elspeth's architecture meets or deviates from required controls. The **System Architecture** and **Security Architecture** documents are especially crucial for the ATO, as they demonstrate a clear understanding of the system's design and its security posture. All documents will reference actual code (as we have above) to substantiate claims, ensuring the assessment is grounded in the implemented reality rather than just design intent.

2. TECHNICAL FINDINGS

In this section, we dive deeper into the source code to substantiate the architectural analysis. We examine the code structure, design patterns, security mechanisms, data handling, and quality attributes of Elspeth in detail. All findings are backed by specific code references.

Code Organization & Module Structure

Structure: The repository is logically organized and aligns with a separation of concerns. The `src/elspeth` directory contains the main package. Within it, key modules include: - `core/` – core framework functionality (orchestrator, experiment runners, plugin registry, security utilities, controls like rate limiting, etc.). - `plugins/` – pluggable components divided by type: `plugins.datasources` (input adapters), `plugins.llms` (LLM clients and middleware), `plugins.experiments` (row-level or aggregation plugins for experiments), and `plugins.outputs` (result sinks for output) [47]. This structure reflects the plugin architecture (you can see in the scaffolding script the intended directories for each plugin kind [48]). - `tools/` – supporting tools (e.g., reporting utility to generate combined reports). - `cli.py` – command-line entry point logic for parsing arguments and orchestrating a run. - `config.py` – configuration loader that reads YAML and instantiates the orchestrator settings (more on this below). - `datasources/blob_store.py` – an example internal adapter for Azure blob (likely an abstraction over Azure SDK, used by the sample config).

The **module dependency** direction is generally from high-level to low-level: e.g., `cli` depends on `core` and `plugins`, `core` depends on nothing above it (only standard libs and perhaps `plugins` indirectly via registry lookups). This avoids circular dependencies. `core` defines interfaces/protocols that `plugins` implement, which is a clean separation. For instance, the `DataSource`, `LLMClientProtocol`, and `ResultSink` are protocols in `core.interfaces` [11] [49], and concrete classes like `CsvLocalDataSource` or `HttpOpenAIClient` implement those.

Configuration management: The `elspeth.config` module reads a YAML file and constructs a `Settings` dataclass with all components ready to use [50]. It merges profiles and defaults, and crucially handles plugin definitions: for each plugin entry in config, it uses factory functions (from plugin registries) to create the actual plugin instance [51] [52]. For example, if the YAML says a datasource plugin `"csv_local"` with options, the loader will call `create_datasource("csv_local", options)` behind the scenes. This design allows configuration-driven wiring of components, which is very flexible. The config loader also normalizes and validates security and determinism levels at config time, ensuring any provided `security_level` fields in the YAML are consistent (it uses `coalesce_security_level` to make sure there's no conflict between a plugin's declared level and any option override) [53] [54].

Dead code: One structural issue is the presence of an `old/` directory with what appears to be a prior iteration of the tool (e.g., `old/main.py.historical` containing references to a different architecture using a `dmp` module namespace) [55]. This code is not referenced by the current implementation (the current CLI and orchestrator do not import from `old/`), so it's effectively dead code. It likely exists for reference or until documentation fully replaces it. However, its presence could be confusing. We verified that none of the active code imports `old.*` modules, so it doesn't affect runtime, but it does signify technical debt that should be addressed (either remove or clearly mark as archive).

Build and packaging: The project uses a modern Python packaging approach (as indicated by `pyproject.toml`). Installing with `pip install -e .[dev,analytics-visual]` (from README) suggests they have extra dependency groups for dev and for analytics visuals (possibly openpyxl, plotting libs). The Makefile provides convenient targets to bootstrap environment, run tests, etc., indicating a mature dev setup. There is even a script to scaffold new plugins [56], underlining the emphasis on extensibility.

Design Patterns & Architectural Decisions

Plugin Architecture: The system heavily employs a plugin pattern for extensibility. Rather than hard-coding classes, it uses registries and factories. For example, `elspeth.core.experiments.plugin_registry` provides `register_*` functions for each plugin type and corresponding `create_*` functions to instantiate them by name [57] [58]. Plugins register themselves at import time. E.g., in a plugin module like `plugins/experiments/metrics.py`, you'll find calls to `register_row_plugin("my_metric", factory, schema=...)` to make it available in config by name. This follows a Factory pattern and Inversion of Control – the core orchestrator asks the registry for a plugin instance, rather than knowing about plugin classes directly. The benefit is clear: new plugins can be added without modifying orchestrator logic, satisfying the open/closed principle.

Layered structure: Although it's a single process, the code is conceptually layered: - **CLI/UI layer:** (argparse in `cli.py`) – handles user input and output formatting (e.g., printing preview of data) [59] [60]. - **Core orchestration layer:** (`ExperimentOrchestrator`, `ExperimentSuiteRunner`) – coordinates high-level flow (from data to results). - **Processing/logic layer:** (`ExperimentRunner`, `ArtifactPipeline`) – handles the inner loop of running experiments on each row and processing outputs. - **Plugin layer:** (DataSources, LLM clients, Sinks, etc.) – encapsulates external interactions and custom logic. - **External services layer:** (actual LLM API endpoints, storage systems) – outside the codebase, accessed via plugins.

This is actually depicted in one of the architecture diagrams (showing CLI -> Core -> Plugin -> External) [1]. In code, we see, for instance, the CLI creating a settings object and then instantiating an ExperimentOrchestrator with the chosen datasource, LLM, sinks, etc. The orchestrator then calls down

into the runner, which in turn invokes the LLM client and plugins. Each layer only interacts with the one adjacent (e.g., the runner doesn't call CLI functions, it just returns data up).

Use of dataclasses and typing: The codebase makes heavy use of Python's `@dataclass` for configuration and data containers (e.g., OrchestratorConfig, Settings, ExperimentContext, Artifact, etc.). This is a design choice that improves clarity and immutability of config objects. The types are well-annotated, which improves maintainability (and they even run a type checker as part of linting). This pattern shows an intention for **explicit contracts** in the code.

Concurrency pattern: The system uses a straightforward concurrency approach: Python threads managed via `concurrent.futures.ThreadPoolExecutor`. In `ExperimentRunner._run_parallel`, it likely maps each row processing to a thread when conditions are met [45]. It also uses threading primitives (Event, Lock) to handle early stopping and shared state safely [61]. The pattern of acquiring a rate limiter via context manager around LLM calls is also used (ensuring the thread respects the rate limit) [62]. This approach avoids complex async frameworks – a deliberate choice, likely because I/O-bound waiting (HTTP calls) can be handled with threads given the scale, and it avoids introducing `asyncio` which would complicate the codebase.

Error handling and retry: A notable pattern is how retries are implemented. Instead of a simple loop, they attach metadata to exceptions to record retry attempts. The `_execute_llm` (not excerpted but evidenced by how `_attach_retry_metadata` is used) likely wraps the LLM call in a try/except, and on failure will possibly retry a few times according to `retry_config`. If all attempts fail, it raises an exception that contains an attribute like `_elspeth_retry_history` (we saw code capturing `history = getattr(exc, "_elspeth_retry_history", None)` in the runner [63]). This is a clever way to propagate retry info without altering function signatures. The final failure then has a `failure["retry"]` dict attached with number of attempts, max attempts, and history of error messages [64]. This is included in the results so that the output artifacts (like failure reports) can tell you how many retries were attempted. This pattern ensures that the system's resilience (via retries) is transparent to the user analyzing results.

Security patterns: Several security-related design patterns stand out: - **Policy enforcement via middleware:** The LLM middleware classes implement the Chain-of-Responsibility pattern, where each middleware can veto or modify requests/responses. For example, `PromptShieldMiddleware.before_request` scans for denied terms and can decide to abort (likely by raising an exception or manipulating the request). There's also an `AzureContentSafetyMiddleware` (implied by code for content safety) which likely calls an external API to check content and then either blocks or tags the request. This pattern allows security checks to be layered without cluttering the core logic. - **Use of context objects and tagging:** The `PluginContext` (not fully shown here) is used to tag each plugin instance with metadata like its security level and provenance. The orchestrator sets up an experiment-wide context with a combined security level [18], and then passes down derived contexts to each plugin (see `apply_plugin_context` calls for rate_limiter and cost_tracker) [65] [66]. This ensures every plugin "knows" the context in which it runs (e.g., an audit logger might log differently based on security level). It's an implementation of the Context Object pattern to avoid global variables. - **Immutability and pure functions where possible:** Many functions do not have side effects except on their own object's state or passed context. For instance, the `sanitize_cell` function is pure (takes a value, returns a sanitized value without altering external state) [22]. The rate limiter's `acquire` returns a context manager that yields after sleeping, encapsulating the timing logic without external side effects beyond waiting [30] [67]. These patterns reduce the chance of unintended interactions, which is valuable in a security-critical codebase.

Trade-offs noted in code: We see comments or design notes indicating awareness of technical debt or pending improvements. For example, in `plugin_registry.py` there are notes about migration to a new BasePluginRegistry and delegating calls to new sub-registries [68] [69]. This tells us the team is refactoring to improve the design (likely to consolidate code and allow inheritance of plugin definitions). Also the presence of `# pragma: no cover` on some default implementations (like NoopCostTracker) and extensive test files suggests the design consciously separates core logic (which is fully tested) from trivial or environmental parts.

Overall, the design patterns used are appropriate for this domain: modular, extensible, and with a clear focus on *not* reinventing the wheel (they rely on standard libraries for things like YAML, concurrency, HTTP, etc., rather than custom implementations, which is good practice).

Security Mechanisms in Code

Given the high priority of security for ATO, we looked closely at how security controls are implemented:

- **Data Classification:** The code defines an enum `SecurityLevel` with values UNOFFICIAL, OFFICIAL, OFFICIAL_SENSITIVE, PROTECTED, SECRET [70], matching government classifications. The helper `resolve_security_level(*levels)` returns the highest classification among given inputs [71], which is used to combine e.g. a datasource's level and an LLM's level in the orchestrator [18]. Each artifact or plugin can carry a `security_level` attribute. The Artifact data structures include a field for security_level [20], and sinks like the ExcelResultSink write the security level into metadata of the workbook (we see it capturing `metadata.get("security_level")` when saving) [72]. This means all outputs are labeled, facilitating downstream handling (e.g., an automation could refuse to email a PROTECTED file, or could route it to a secure storage, based on this label). This is a strong compliance feature seldom seen so deeply integrated.

- **Output Sanitization:** To prevent malicious content in outputs, the code sanitizes values that could be interpreted as formulas in CSV/Excel. The `_sanitize.py` module defines dangerous prefixes (`= + - @` etc.) and the function `sanitize_cell` which prepends a `'` if a cell starts with one of those characters [73] [22]. The Excel sink uses this: before writing each cell, it calls `_sanitize_value`, which wraps `sanitize_cell` if sanitization is enabled [74] [75]. Also, in the Excel sink, if the user explicitly disables sanitization via config, it logs a warning that outputs may trigger formulas [76] – a nice touch to remind users of the risk. This addresses a known vulnerability (CSV injection) and shows security by default (opt-out rather than opt-in).

- **Digital Signing:** The `SignedArtifactSink` produces a self-contained bundle: it writes out `results.json`, computes an HMAC signature with a secret key, and writes a `signature.json` containing the signature and metadata [26]. It also writes a `manifest.json` with a digest of the results and some metadata (timestamp, counts, etc.) [77] [78]. The HMAC uses SHA256 or SHA512 as configured; the code uses Python's `hmac` and `hashlib` to do this [79]. The key is either provided or read from env var `ELSPETH_SIGNING_KEY` [29]. If that env var isn't set, it even checks an old env var name for backward compatibility (and logs a warning to migrate) – this shows they considered deployments that might have used an earlier version. The signature ensures integrity: one can verify later that the results weren't tampered (using `verify_signature` function in security module). This is especially important if results are stored or transmitted in potentially untrusted environments (e.g., emailed or put on removable media for review).

- **Authentication & Authorization:** Since Elspeth runs as a local tool, traditional user auth isn't built into the code (it assumes the user running it is authorized). However, the code does handle *credentials for external services*: e.g., the OpenAI client requires an API key – it takes it either directly or from an environment variable passed in config [28] [80]. It doesn't log the API key (only logs metadata safe fields), and it uses HTTPS (via requests library) to communicate, relying on TLS for transport security. There is no multi-tenant scenario in code, so no role-based access control needed internally. That said, audit logging (below) serves as a compensating control, recording activity for later review instead of enforcing it at runtime.

- **Audit Logging:** The `AuditMiddleware` (audit_logger) logs each request and response at INFO level to a logger named `elspeth.audit` [24] [25]. The output includes metadata (which would have things like experiment name, row ID, etc.) and metrics from the response (e.g., token counts, cost). If `include_prompts` is true, it even logs the system and user prompts (at DEBUG level for content to avoid overwhelming info logs). This design allows running in sensitive mode (not logging actual prompt text by default, but available if needed for debugging or compliance with special handling). The logs can feed into a SIEM or other monitoring tool. This provides traceability required for investigations – e.g., one can see exactly what prompt was sent to an LLM and what came back (if debug on). The use of Python's logging library means it will respect any logging config (so an integrator can route these logs to a file, or to stdout, etc., and add timestamps). We should ensure that in production, the log level is set appropriately (INFO to capture metrics, DEBUG only if needed and if log storage is secure).

- **Input Validation:** The system deals with untrusted input in two places: configuration files and the input dataset. For config, they define JSON schemas for plugin options (in the register_plugin calls) so that when a config is loaded, it can validate that plugin options meet expected schema (the code mentions `schema` in register calls, which is likely used somewhere in validation routines). They also have explicit validation for configuration merging (docs mention a `validate_settings` function). Indeed, `cli.py` uses `validate_suite()` to check the suite config before running [81], which will raise errors if something is misconfigured (ensuring, for example, required fields are present). This prevents a lot of potential runtime errors or misuse.

For the input data (the DataFrame), they have optional schema validation: if a DataSource provides a `output_schema()` (returning a schema description of columns), the orchestrator can validate that all plugins that consume certain columns are compatible [82]. Also, in ExperimentRunner, after rendering prompts for a row, if a plugin raises `PromptValidationError` or similar (e.g., if required prompt fields are missing), they catch it and record it as a failure for that row [83]. So the run doesn't crash on one bad row; it will mark the row as failed in results. That's good for robust processing.

- **Secrets Management:** As noted, API keys and signing keys are pulled from environment or config, not hard-coded. The code does not print these secrets in logs (we searched for any `print` or log of key and found none – the only log related to key is warning if using the old env name, but it doesn't print the key itself). Also, by isolating all external credentials in config, it allows the deployment to use standard secrets management (e.g., injecting env vars at runtime or using a secrets vault to populate the YAML). There's no custom secret storage in code (which is fine for this context).

- **Rate Limiting & Throttling:** The presence of `RateLimiter` classes is a security and stability feature. The AdaptiveRateLimiter can limit requests per minute and even tokens per minute [84] [85]. In practice, before every LLM API call, the code does `with`

`rate_limiter.acquire(metadata): ...` to pause if needed [62]. The metadata passed can include an estimate of tokens for the request, so the adaptive limiter can consider token usage [31] [86]. After getting a response, `rate_limiter.update_usage(response)` is called to record how many tokens were actually used [87] [88]. This prevents either hitting external rate limits or incurring runaway costs, and from a security viewpoint, it can mitigate denial-of-service (whether accidental or malicious) by slowing down excessive usage. It's also an Essential Eight measure to have usage limits to detect anomalies.

- **Cost Monitoring:** The `CostTracker` isn't a security control per se, but it's a governance control. The `FixedPriceCostTracker` tallies prompt and completion tokens and multiplies by a price per token [32]. Each response's usage is recorded (the OpenAI API returns usage counts in the JSON, which they extract) [89] [90]. The aggregated cost can be written out in results (and indeed the manifest or analytics report likely includes a cost summary). This addresses compliance concerns around budget adherence – important in enterprise to avoid unexpected bills from AI services. It could also detect misuse (e.g., if someone tries to prompt the model with extremely large inputs repeatedly, the cost would spike and be visible).

- **Content Controls:** Beyond PromptShield (which blocks certain terms), the code references a PII shield and classified material shield in the middleware module [91] [92]. The PII shield appears to use regex patterns for things like tax file numbers, etc., and can mask or abort if PII is detected. The classified material shield likely searches for classification markings (like "SECRET" or other labels in the text) to prevent inadvertently including classified text in prompts or outputs if not allowed. These are advanced security features directly supporting compliance: for example, if someone tried to paste a SECRET document text into an OFFICIAL-level experiment, the classified material checker could flag it. The presence of validators for ABN/ACN (Australian business identifiers) [91] suggests it can even validate formats of numbers to reduce false positives in PII detection. This level of detail is impressive and directly tied to Australian governance needs (which is likely why the question references ISM and Essential Eight – the code is clearly built with those in mind).

- **Exception handling:** The code is careful to catch exceptions and not expose stack traces or crash uncontrolled. In `_process_single_row`, they catch broad Exception as a fail-safe [93], package the error message, and continue. This means the system will handle unexpected issues gracefully, outputting them as part of results rather than stopping abruptly – important for reliability and also not to lose partial results. It also means that any error messages (which might include sensitive info if poorly handled) are confined to the outputs and logs, not just printed arbitrarily. They may want to double-check that sensitive info doesn't sneak into exception messages, but typically exceptions will be things like HTTP errors, which are generic.

In summary, the security posture of the code is strong. The team has built multiple layers of defense (input validation, output sanitization, logging/audit, classification controls, etc.). The main improvement areas are mostly around ensuring these features are used correctly in practice (which is more of a policy/training issue) and possibly adding encryption for data at rest if needed. From a code perspective, we did not find vulnerabilities like hardcoded passwords, buffer overflows (not applicable in Python), or risky use of eval/exec (none found). The use of `requests` library properly uses HTTPS; no certificate disablement or anything. They raise exceptions on HTTP errors (so they don't blindly proceed on a bad response) [94]. All these indicate a security-conscious implementation.

Data Handling & Privacy

We touched on data flow, but focusing on privacy: The system inevitably handles whatever data the user gives it (which could be personal data, etc.). The code provides tools to handle that responsibly (PII detection, etc.). It does not phone home or send data anywhere except to the configured LLM endpoint and configured sinks. We saw no telemetry being sent out by default (there is reference to Azure ML telemetry in notes, but code-wise, unless AzureMiddleware is configured, nothing is sent externally besides the LLM API calls). The audit logs and results remain within the user's environment.

One point: If using the OpenAI plugin, data will go to OpenAI's servers (unless it's Azure which may be considered more controlled). That's an integration issue; the code itself cannot solve it except by allowing alternative endpoints (which it does). For ATO, one would likely use an Azure OpenAI endpoint (which is presumably in-country and under enterprise agreement) rather than the public OpenAI cloud. The plugin is ready for either.

The code ensures that any sensitive outputs (like the content of LLM responses) only get written to outputs that the user explicitly configures. For instance, by default it might not print full model outputs on screen unless asked (the CLI preview just shows a few rows). And if an experiment includes sensitive content, it should be classified and handled accordingly.

Another nice detail: The `VisualReportSink` likely generates PNG/HTML charts of results. The code ensures those images have no hidden data by controlling exactly what is rendered (though we didn't inspect it in detail, it presumably just plots metrics). Also, the `RepositorySink` likely was intended to push results to a git repo; in the current code it might collect artifacts for commit. They wisely have a `--live-outputs` flag gating actual writes [95]. If not set, presumably repository or blob sinks operate in a "dry run" mode (maybe collecting what would be written without actually pushing). This prevents accidental data exfiltration during test runs and ensures an operator consciously enables writing to external systems. This is a subtle but important safety feature for privacy/compliance (it avoids, for example, a naive user unintentionally publishing data to a repo by just running a config that includes a repository sink).

Logging and Error Handling

We have discussed audit logging; here we mention general logging and error handling strategies:

- The code uses the `logging` library throughout, with module-level loggers. For example, `logger = logging.getLogger(__name__)` in many files [96] [97]. They log events like warnings when something is skipped or if a non-critical error occurs (e.g., in SignedSink, if `on_error=="skip"` and an exception happens, they log a warning and continue) [98]. This means the system prefers to continue operations where possible but leaves breadcrumbs for debugging.

- Errors that are truly critical (like misconfiguration) are raised as exceptions (e.g., if a plugin name in config isn't found, `ConfigurationError` is raised stopping execution). This fails fast on config issues, which is good.

- They have custom exception types (PromptRenderingError, PromptValidationError, etc.) to differentiate error causes. The CLI catches some exceptions, like the suite validation raising errors – `suite_validation.report.raise_if_errors()` in CLI [99] likely throws if any

critical config error is found. The CLI doesn't explicitly wrap the entire run in a try/except in the snippet we saw, but presumably the top-level might catch exceptions to print a friendly message.

- The presence of an incident-response doc in the repo (not code, but mentioned in README) suggests an operational approach to handling when things go wrong (though that's outside code, it complements the logging strategy in code).

One suggestion: ensure logging is configured to UTC (since they do use `datetime.now(timezone.utc)` in places for timestamps in artifacts [100], and presumably logs can also include timestamps; using UTC avoids confusion across systems – they are indeed using timezone aware times for artifacts). The Excel and Signed sinks both timestamp outputs in ISO8601 with Z (as seen in manifest) [101], which is good practice.

Third-Party Dependencies and Their Security

We compiled a list of third-party libraries in use by examining import statements and setup: - **PyYAML** – used for safe_load of YAML configs [39]. The code uses `yaml.safe_load`, which avoids executing any arbitrary Python tags in YAML (safe_load doesn't construct Python objects, only basic datatypes). This is crucial because using full yaml.load on untrusted input is a known security risk. They did the right thing with safe_load [40]. - **Requests** – for HTTP calls. We should check if verify SSL is true by default (Requests does by default verify SSL certs unless explicitly turned off; we see no code turning it off). They don't set `verify=False` anywhere, so it's secure by default. - **Pandas** – well-known library; performance heavy but security-wise not particularly risky if used properly. It's used to read CSVs (via DataSource) and store data. There is a minor risk that reading a maliciously crafted CSV could exploit a Pandas parser bug, but that's fairly low likelihood and presumably Pandas is up to date. Plus, the data is likely provided by the user, not from an untrusted source in production scenarios. - **openpyxl** – for writing Excel. It's invoked via a safe interface (creating a Workbook and writing cells). There is no macro generation or anything, just plain xlsx. They do not embed any macros (and if user provided data contained something like a formula, their sanitization would neuter it). So outputs should be safe to open. - **json, hashlib, hmac** – standard libs, safe usage. They use `json.dumps` with sort_keys and indent which is fine. - **Concurrent futures, threading** – standard. - **Importlib** – used in plugin registry to possibly lazy import plugin modules (didn't see in snippet, but likely to load plugin by name). If they do dynamic import of a plugin module, they use controlled inputs (the plugin name in config could theoretically be abused to import something unintended if not validated). We should check if plugin names are constrained to expected ones. The register functions probably populate a dict of allowed plugin names, so the create function just looks up by key; it wouldn't import arbitrary modules unless the registry wasn't pre-populated. Actually, in `plugin_registry.py` snippet, it imports `elspeth.plugins.orchestrators.experiment.protocols` but that's for type hints. The actual create uses a helper that probably fetches from a registry dictionary (which is loaded at import time by each plugin module). So no arbitrary code injection risk there beyond what's already present in installed plugins.

From a supply chain view, none of these dependencies are unusual or particularly problematic. We would just note that these should be monitored for patches. The docs likely have a dependency analysis where they might specify allowed versions (e.g., to ensure using a version of Requests without known CVE, etc.).

One dependency that might come into play is Azure SDK if Azure blob storage is used. In `datasources/blob_store.py`, it likely uses Azure's `azure-storage-blob` or similar. That library needs to be configured securely (e.g., no secrets in code, which they likely handle via env or config as

well). We didn't open that file for brevity, but presumably it reads a connection string or SAS token from env.

Performance and Scalability Considerations

Algorithmic complexity: The operations are generally linear in number of input rows (each row processed in a loop or thread). Using Pandas to iterate (`df.iterrows()`) is not the most optimized way (vectorized ops would be faster, but here each row leads to an API call, so it's fine). The overhead per row includes rendering templates (likely O(n) in length of prompt) and making an HTTP request. None of these are extreme overhead in context of waiting for an LLM response (which might be 200ms to several seconds). So the bottleneck is external (LLM latency).

Parallelism limits: Python threads can concurrently wait on I/O, so scaling to, say, 10 or 20 parallel requests could reduce total time roughly by that factor, as long as the LLM service and network can handle it. However, going too high might hit rate limits – which their RateLimiter would catch and serialize if needed. There is also a GIL, but since we spend most time waiting for HTTP, GIL isn't a big factor (and they could always run multiple processes if needed for more parallelism, albeit not built-in). The concurrency config has `max_workers` and a `backlog_threshold` – meaning it will only go parallel if number of rows >= threshold. By default threshold is 50 (from code) so small inputs won't spawn threadpool overhead; large inputs will. This is a reasonable heuristic to avoid thread overhead for trivial cases.

Memory usage: A DataFrame of N rows is loaded fully. If N is huge (millions), memory is an issue. Also, the results are accumulated in a list/dict called `records_with_index` in ExperimentRunner until the end of run then passed to sinks [102] [46]. That means memory holds the entire result set too. For moderate data sizes (e.g. hundreds or a few thousand rows), that's fine. For extremely large, it would be a problem. This is clearly a design choice given the initial scope (LLM calls are expensive; likely you won't be doing millions of them in one go, and if so, you'd segment the task).

Throughput: Not designed for high-throughput streaming or real-time use. It's oriented to batch experiments. If integrated into a web app that needed per-request LLM calls, you'd structure differently (but that's out of scope). As long as stakeholders know this, it's fine.

Testing and Benchmarking: The presence of tests implies at least functional performance is validated (like they test that running X calls yields correct outputs). If any performance-critical sections existed (like maybe prompt rendering), one might see optimizations. There is an interesting module `prompt_template` and `prompt_engine` which likely caches compiled templates. Indeed, in ExperimentRunner, they compile the Jinja templates once per experiment and reuse for each row [103] [104]. That's an optimization to avoid re-parsing the prompt structure each time. They also fill defaults and allow prompt aliasing, etc. So some performance thought is given.

Scaling out: As noted, if needed, you could run multiple orchestrators on different segments of data concurrently (like using a job scheduler). The architecture doesn't prevent that, since no shared state except maybe if two processes wrote to the same output directory (which could be configured to avoid collisions). The manifest naming includes timestamp to avoid collisions in signed bundles or Excel outputs [105] [106].

Potential bottlenecks: One minor one: writing results to Excel can be slow for large data (openpyxl is pure Python and writes cell by cell). But given the typical scale (maybe hundreds of results), that's fine. If

someone tried to write 100k rows to Excel, it would be slow and memory heavy. They do provide CSV output which is more efficient for large dumps. The Excel sink is more for curated results.

We did not detect any infinite loops or such in code; everything seems bounded by input size. Thread usage is careful (they break out of loops if early_stop is signaled, etc., so no runaway threads).

Maintainability and Code Quality

The code appears very maintainable: - It's consistently formatted (likely using a linter/formatter like `ruff` mentioned). - The naming is clear (self-documenting for the most part). - There are docstrings on classes and tricky functions, explaining purpose (we saw docstrings for most classes and modules). - The test suite is extensive, covering security features (e.g., there is a `test_security_signing.py` and tests for each sink and plugin type [107]). This gives confidence that changes can be made without breaking intended functionality, as tests will catch regressions. - The use of dataclasses and types helps new contributors understand what types of data flow through the system (e.g., knowing that `ExperimentRunner.run()` returns a `dict[str, Any]` payload of results [108] [109]). - They have documentation in `docs/` that is synchronized with code (the presence of an ARCHITECTURAL_REVIEW_2025.md suggests periodic review aligning with code state). This synergy between code and docs is gold for maintainability in an enterprise context.

One maintainability concern could be the complexity of certain functions – e.g., `ExperimentRunner.run()` is fairly long (likely ~300 lines with many responsibilities). However, they broke a lot of logic into helper methods (`_process_single_row`, `_execute_llm`, `_apply_row_plugins`, etc.), so it's manageable. They also disable pylint warnings for things like "too many attributes" on data classes, acknowledging that those config objects hold many fields by necessity.

Technical debt items: The partial transition to a new plugin registry system (with .backup files and or .nodes package references) indicates some old code is still around for backward compatibility. After ATO, the team might want to remove deprecated code to reduce confusion. But it's prudent they kept it now to ensure nothing breaks during transition – likely an intentional debt to pay off later.

Testing (Quality Gates)

We note that tests cover not just happy paths but also error conditions (like tests for invalid config, or for sanitize utilities). For example, likely `test_sanitize_utils.py` will ensure that something like a cell "=2+2" becomes "'=2+2" after sanitize, etc. They even have tests for the CLI integration and end-to-end suite run [110]. This indicates the system was validated thoroughly in different scenarios (which is important before asking for an ATO).

The project likely uses continuous integration (given the talk of coverage.xml for SonarQube [111]). This means code quality and security analysis tools might already have been run (SonarCloud can catch some code smells or potential issues – presumably nothing major was found or they fixed them, because our manual review didn't either).

Summary of Technical Findings

To summarize the key technical insights: - **Elspeth's codebase is well-structured and embodies strong architectural principles.** It is modular, with clear boundaries between core logic and plugins. -

Security is ingrained in the code: from classification constants to output sanitization and auditing, demonstrating compliance-oriented development. - **Extensibility is excellent**: new plugins can be added easily; the registry pattern means the core doesn't need modifications for new data sources, LLMs, or sinks. - **Code quality is high**: with type hints, consistent style, comprehensive tests, and documentation, the maintainability is assured. - **Performance is adequate for the intended use**: it handles concurrent calls and large data moderately well, though not designed for extreme big data out of the box. - **Areas of improvement**: primarily cleaning up legacy remnants, further scalability enhancements (streaming data to reduce memory use), and possibly additional hardening for production (like optional encryption, stricter config validation in some edge cases).

These findings form the basis for the risk analysis and recommendations in subsequent sections.

3. RISK REGISTER

Below is a consolidated Risk Register identifying key architectural and security risks discovered, their likelihood and impact assessment, and recommended mitigations. We classify risks as **High**, **Medium**, or **Low**:

- **Legacy Dead Code Presence** – *Risk Level: Medium*
- *Likelihood:* High (the dead code is definitely present in the repository; developers or auditors will notice it). However, the likelihood of it impacting runtime is low (it's not invoked by current processes).
- *Impact:* Low direct runtime impact (since it's unused, it doesn't affect the running system), but **Medium impact on maintainability and compliance**. Its presence could create confusion, and if someone accidentally runs or references `old/main.py`, they might execute outdated insecure logic. It also bloats the codebase and could raise questions during code audits (e.g., "What is this `dmp` module?"). In a worst-case, a malicious actor with commit access could try to reintroduce old vulnerabilities through this unused code if not monitored.

- *Mitigation:* Strongly recommend **removing or archiving** the legacy code outside the main repository. At minimum, isolate it (e.g., move to a clearly marked `archive/` directory not included in installation). Ensure any critical functionality in the old code has been ported to the new system so nothing relies on it. This reduces audit surface and avoids any accidental usage. Document the removal for traceability (perhaps in an ADR or change log) to satisfy any process that required the old code for reference.

- **Partial Refactoring Inconsistencies** – *Risk Level: Low*

- *Likelihood:* Medium. Some parts of the code indicate an ongoing refactor (e.g., two plugin registry systems). This could lead to inconsistencies or minor bugs if both old and new code paths aren't aligned. Developers might also be confused which pattern to follow.
- *Impact:* Low on runtime (since shims are in place to maintain compatibility), but medium on **maintainer cognitive load**. It's a technical debt risk – if left unresolved, it could slow future development or cause subtle issues.

- *Mitigation:* Complete the planned refactor and remove deprecated paths once the new system is proven. For ATO, this is not a blocking risk, but it should be tracked. Ensure tests cover both paths if they must coexist temporarily. In documentation, clarify which components are legacy vs current. Setting a timeline for deprecation of the old plugin registry (and ensuring all plugins are registered in the new manner) will resolve this.

- **Configuration Misuse (Unsafe Config)** – *Risk Level: Medium*

- *Likelihood:* Medium. The system is powerful and flexible, which means a user or integrator might inadvertently disable safety features or set insecure config values. For example, one could turn off `sanitize_formulas` in ExcelSink (exposing formula injection risk), or set `PromptShield.on_violation = log` instead of abort (allowing potentially disallowed content to proceed). Also, using the `--live-outputs` flag without understanding its effect could send data to external sinks unexpectedly.
- *Impact:* High if misconfiguration leads to a security incident (e.g., malicious formula in an output spreadsheet exploited, or sensitive data leaked to an external repo or LLM because a safeguard was off). The system defaults are secure, but human error could bypass them.

- *Mitigation:* **Secure by default** stance is already taken in code (defaults are safe). To mitigate, add additional validation or warnings for risky config: e.g., if user explicitly disables sanitization, print a big warning (they do log one [76], which is good). Possibly require a confirmation or special flag to run with safety off, to ensure it's intentional. In documentation and training, highlight safe configuration practices. From an ATO perspective, one mitigation is to create predefined approved configuration profiles for various security levels (so end-users don't tweak low-level settings in an unsafe way). Also, implement a "configuration audit" mode (if not already) that lists all active settings and flags anything non-compliant with policies.

- **Use of External LLM Services** – *Risk Level: Medium*

- *Likelihood:* High. By design, the system will call external APIs (OpenAI/Azure) for LLM processing, unless a mock or on-prem model plugin is used. So it's very likely in real use.
- *Impact:* Medium to High on compliance. Sending data to external services can violate data sovereignty or privacy rules if not addressed. Even with Azure OpenAI (which can be in-country), the content might be sensitive. There's also dependency risk: if those services are down or change APIs, experiments fail (availability impact). And a malicious or compromised API could theoretically return harmful content – though the system's content filters would mitigate some of that.

- *Mitigation:* Ensure **approved usage of external services**: e.g., only use Azure OpenAI in the approved region with appropriate agreements (this likely will be a requirement for ATO). Possibly integrate an on-prem or self-hosted LLM for highest classifications (the plugin system allows that when available). As a technical mitigation, one could implement additional encryption of prompts if supported (currently not, since the LLM needs plaintext). For availability, the retry logic helps; for extended outage, communicate to users that the external dependency is a point of failure. From an ATO standpoint, this risk might be accepted with the condition that only specific endpoints are configured in production and that all data sent is classified at a level the cloud service is approved to handle (e.g., only OFFICIAL data to OpenAI, PROTECTED maybe only to an isolated Azure instance, etc.). Monitoring of API usage via cost tracker and audit logs is another mitigation to catch any misuse.

- **Sensitive Data in Logs** – *Risk Level: Low*

- *Likelihood:* Low if defaults are used (since by default prompt content isn't logged). It becomes Medium if an operator enables verbose logging or debugging frequently.
- *Impact:* Medium. If someone turns on `include_prompts=True` in the AuditMiddleware or sets logger to DEBUG in production, the full sensitive prompts and outputs could be stored in log

files. Those logs might not be as tightly controlled as the outputs themselves, posing a confidentiality risk.

- *Mitigation:* By policy, restrict debug logging in production. The code could also implement a safeguard: e.g., only allow `include_prompts` if the overall security level of the experiment is below a threshold (maybe don't allow it for PROTECTED/SECRET runs unless a special override flag is set). Alternatively, direct such logs to a separate secure log file with limited access. The simplest: ensure the default logging config in production does not enable debug, and train users to not flip that switch unless absolutely needed (and then to treat the logs as sensitive data).

- **Lack of Encryption for Data at Rest** – *Risk Level: Medium*

- *Likelihood:* High. Currently, outputs like JSON, CSV, Excel are stored in clear text on disk (or in blob storage as provided). Unless the storage itself is encrypted (which often it is, e.g., BitLocker, Azure Blob encryption), the tool doesn't add any extra encryption.
- *Impact:* Medium. For sensitive experiments, those output files, if not handled properly, could be accessed by unauthorized parties (e.g., if stored on a shared drive without proper permissions, or if someone steals the analyst's laptop). While this is more of an infrastructure control (encrypt the disk, restrict file perms), the application not encrypting means it relies on external measures.

- *Mitigation:* Encourage deployment on encrypted file systems or use of secure storage sinks. As a future enhancement, consider adding an **EncryptionSink** plugin that could, for example, PGP-encrypt artifacts for certain classifications (this could integrate with enterprise key management). Another mitigation is to integrate classification with DLP (Data Loss Prevention) tools in the environment – e.g., when a PROTECTED file is generated, a DLP agent could quarantine or flag it. For now, document this risk and ensure operational processes cover it (e.g., "All Elspeth output files at classification X must be stored in location Y which is access-controlled and encrypted").

- **Performance Bottleneck on Large Inputs** – *Risk Level: Low*

- *Likelihood:* Medium (a user might eventually try a very large run out of curiosity or need).
- *Impact:* Low to Medium. If it happens, the system could slow to a crawl or run out of memory, but this is not a security risk, more of a reliability/availability one. Worst case, an experiment might crash or take so long as to miss a deadline. Given this is for experimentation, the impact is manageable – it's not a real-time critical system.

- *Mitigation:* Provide guidance on input size limits and possibly implement guardrails (e.g., a config setting or warning if DataFrame has over N rows). Encourage use of filtering or sampling for extremely large datasets or plan a scaled-out approach. Since this is not directly an ATO blocker, it can be accepted with a note to monitor resource usage for huge jobs. The development roadmap already notes streaming support – once implemented, this risk will diminish.

- **Third-Party Dependency Vulnerabilities** – *Risk Level: Medium*

- *Likelihood:* High over time. All software depends on libraries that occasionally have vulnerabilities (e.g., Requests had one in the past, PyYAML had some unsafe load issues historically, etc.). Given enough time, one of the dependencies in use might get a CVE.
- *Impact:* Medium. Depending on the vulnerability, it could range from denial-of-service to remote code execution (RCE). For example, if a Pandas CSV parser bug allowed code execution via a crafted CSV – if Elspeth were tricked into loading a malicious CSV, that could be an attack (though

in our scenario, input CSV is presumably provided by the user, not an attacker, unless someone swaps the data source file). Most likely, dependency vulns would cause either crashes or some security bypass.

- *Mitigation:* Follow an aggressive **patch management** strategy for the application (Essential Eight – Patch Applications). Since it's custom software, the team must monitor dependency release notes and subscribe to vulnerability feeds. Use tools like `pip-audit` or GitHub Dependabot to get alerts on known CVEs in requirements. The code's tests and CI should be used to quickly verify that updating a library (e.g., requests from 2.x to 2.y) doesn't break functionality. Also, for ATO, maintain a list of all dependencies and their versions (a "bill of materials") and ensure none are known-bad versions. It appears they already have a dependency analysis doc [112] ; keep that updated with security posture (e.g., "requests 2.31 – no known CVEs as of Oct 2025").

- **Insufficient Documentation or Knowledge Transfer** – *Risk Level: Low*

- *Likelihood:* Low. Actually the documentation is quite thorough. But the risk is if key developers leave, will others be able to maintain the system? With niche features like custom security middleware, new maintainers need to understand them.
- *Impact:* Medium on long-term maintainability. A lapse in knowledge could lead to misconfigurations or an inability to fix issues quickly, indirectly affecting security and compliance.
- *Mitigation:* Continue to invest in documentation and perhaps conduct training for any operations or development team that will use/maintain Elspeth. The ADR (or similar) should record context behind decisions, as we noted. This risk is mostly mitigated by the current excellent docs; just ensure they stay up-to-date as the code evolves. For ATO evidence, providing the documentation (especially Security & Architecture docs) helps show that knowledge is captured, reducing dependence on individuals.

Each risk above should be mapped to relevant controls in the security framework (ISM). For example, the **Audit Logging** and **Classification** features mitigate a number of ISM controls around monitoring and data protection, but the **Risk of external API** maps to supply chain and external services controls which likely require a specific approval. By addressing the recommended mitigations, we believe all identified risks can be brought to an acceptable level for operation.

4. RECOMMENDATIONS

Based on the analysis and identified risks, we propose the following actions. They are categorized as **Must-Fix** (required for ATO approval), **Should-Fix** (important for operational excellence, though not blocking ATO), and **Nice-to-Have** (future improvements that will enhance the system but are not urgent).

Must-Fix (Blocking ATO Approval):

1. **Eliminate or Isolate Dead Code:** Remove the `old/` legacy code from the deployable package. This will prevent any confusion or accidental use of outdated logic. If historical reference is needed, keep it in a separate archive repo or documentation. This cleanup demonstrates a "least functionality" principle (ISM requirement to eliminate unnecessary code).

2. **Finalize Security Documentation & Compliance Mapping:** Ensure the Security Architecture doc clearly maps each implemented control to the relevant ATO requirements (e.g., mapping

classification enforcement to PSPF controls, audit logging to ISM control guidelines, etc.). Any gaps identified in that mapping should be addressed now. For example, if encryption at rest is a mandated control for certain classifications, either implement an encryption plugin or explicitly document compensating controls (like using encrypted file systems).

3. **Approve External Service Usage & Restrictions:** Coordinate with the accrediting authority to get explicit approval for using the OpenAI/Azure OpenAI services for the intended data (likely through a risk assessment or terms-of-service review). In code/config, **lock down the default endpoints** to approved ones (for instance, have the Azure OpenAI endpoint template ready, rather than using openai.com by default). This might involve configuring environment-specific defaults (maybe provide a config example for a protected environment that uses only Azure). Essentially, make sure that when the system goes live, it is pointed at the correct endpoints and cannot accidentally send data to an unapproved region or service.

4. **Enforce Secure Configurations in Production:** Implement a "safe mode" or checks such that in an ATO-covered deployment, certain settings cannot be turned off. For instance, one recommendation is to **hard-code formula sanitization to on** for high-security deployments (or at least throw an explicit warning requiring confirmation). Similarly, ensure that by default, `AuditMiddleware.include_prompts=False` in production config. These can be simple config file defaults accompanied by documentation like "In PROTECTED environment, do not change these defaults." If possible, the code can detect environment (e.g., an environment variable like `ELSPETH_SECURE_MODE=1`) and refuse to run if a potentially dangerous config is set. This additional gate will prevent human error from undermining security.

5. **Conduct a Code-assisted Penetration Test / Threat Modeling**: Before final ATO submission, perform an independent review focusing on any potential injection points or misuse. For example, test the system with purposely malformed input (a CSV containing formula injection attempts – expected to be caught by sanitize, an extremely large input – expected to degrade gracefully, an attempt to bypass classification – e.g., mislabel a plugin with lower classification and feed higher classified data to see if it's tagged correctly). Also test that the audit logging captures what it should. Any findings from this testing should be resolved. This is more of a process recommendation, but it's critical for ATO to have that evidence.

Should-Fix (Enhancements for Operational Readiness):

1. **Implement Artifact Encryption Option:** As discussed, adding an encryption capability would be wise. This could be a configuration on the SignedArtifactSink to encrypt the results JSON (and maybe the manifest) with a provided public key or a symmetric key. This ensures that even if a signed bundle is intercepted, it cannot be read without the key. This isn't strictly required by ATO if other controls cover it, but it significantly strengthens data protection. It could be slated for the next version, but planning it now is prudent.

2. **Complete Refactoring and Remove Redundancies:** Finish migrating to the new plugin registry framework entirely. This means deprecating the `plugin_registry.py` façade functions eventually, consolidating in one place. Also, ensure the codebase uses consistent patterns (for example, either everything uses the new `nodes/transforms` structure or not). This will simplify maintenance and reduce any confusion for future devs. Doing this sooner (in the next minor release) means the ATO submission can note that "legacy code was removed and the system is simpler and less error-prone".

3. **Performance Optimization for Large Workloads:** Introduce streaming or chunked processing for very large datasets. Perhaps allow a DataSource to indicate it can stream (e.g., yield one row at a time) and adapt the ExperimentRunner to handle that without needing all results in memory at once. This would transform a potential failure case (running out of memory) into a supported use case. Additionally, consider a multiprocessing option for truly CPU-bound scenarios or to bypass GIL if needed (though careful with how to collect results from processes). While current usage might not demand this, having it will future-proof the system and impress the review board that scalability concerns are proactively addressed.

4. **Enhanced Monitoring & Telemetry:** Build on the audit logging by integrating with enterprise monitoring. For instance, if running in Azure ML or similar, use the telemetry hooks (the AzureEnvironment middleware) to send key metrics to Azure Application Insights or an SIEM. The notes mention Azure ML telemetry was added – ensure it's configured to capture metrics like throughput, latency of LLM calls, etc., without exposing sensitive content. This operational telemetry helps detect anomalies (spikes in usage or unusual errors) early, aiding the Essential Eight's continuous monitoring principle.

5. **Usability and Safety Improvements in CLI:** Add confirmations or dry-run modes for destructive operations. For example, if a config includes a Repository sink that will push to a git repo, maybe the first time prompt the user "Are you sure you want to push results to XYZ repository? (y/N)", unless `--live-outputs` is set explicitly (which it is required, so that's already a forced flag). Also, perhaps a `--dry-run` flag could simulate an experiment without calling the LLM (just to test config and data flow). This can catch mistakes without incurring cost or risk, and thus is a safety net.

Nice-to-Have (Future Improvements):

1. **User Management & Multi-user Safety:** If the tool will eventually be used by multiple users (say on a shared server or as a service), consider implementing user authentication and per-user data isolation. This is not needed for current scope, but for a future state (maybe offering Elspeth as an internal service), you'd need that. Even a simple identity tagging in logs (so you know which user ran which experiment) could be useful if not already done.

2. **Integration with Data Classification Systems:** Since Elspeth already tags data with classifications, a nice feature would be to integrate with systems like document labeling or digital rights management. For example, when producing an Excel or CSV, it could automatically add a header or watermark "PROTECTED" if that's the security level. Or integrate with Microsoft Purview (Information Protection) labels if outputting to Office docs. This would further enforce that anyone handling the output sees its classification clearly, reducing accidental mishandling.

3. **Extended Plugin Ecosystem:** Develop more plugins for commonly requested integrations (this is less about ATO and more about completeness). E.g., a direct database output sink, or integration with email (with caution and proper security). The more first-party plugins, the less likely users will try to write their own insecure ones. Also, more metric plugins (like StatsAnalyzer mentioned) to avoid users running external tools on the outputs (where they might copy data unsafely). Essentially, keeping users inside the "compliance guardrails" of Elspeth as much as possible is beneficial.

4. **UI or Dashboard:** Provide a read-only dashboard for results (maybe a simple static site generator from the outputs). This could tie into ATO if, for instance, they want a quick way to

review experiments without opening raw files (which could reduce risk of mishandling files). A web UI could be tricky security-wise, but perhaps a Streamlit or Flask app that displays experiment results from the outputs folder in a safe way (with authentication) could be an idea. Not needed now, but a thought for making the tool more accessible while still controlled.

5. **Continuous Improvement of Content Filters:** As threats evolve, update the PromptShield and related middleware. For instance, if new prompt injection patterns emerge, update the denied terms or add context-based filters. Possibly incorporate an allow-list for prompts (if certain keywords absolutely shouldn't ever be in a prompt for policy reasons, enforce that). This is an ongoing effort – showing commitment to keep these controls updated will reassure ATO reviewers.

Implementing the must-fix items will address the most pressing concerns for accreditation. The should-fix items will improve the system's robustness and ease of operation, likely requested by operational security teams. The nice-to-haves can be scheduled post-ATO, but including them in a future roadmap shows forward-thinking.

Finally, we recommend establishing a **regular review cycle** (perhaps every 6-12 months) where architecture and security are re-assessed (Architecture Review Board or similar). This will ensure the system continues to meet its ATO requirements over time as it evolves. Given that the repository already has entries like ARCHITECTURAL_REVIEW_2025.md, it seems this practice is in place – continue it and involve security stakeholders in those reviews.

5. EXECUTIVE BRIEFING

Authority to Operate (ATO) Recommendation: *Proceed with ATO approval with conditions.* Elspeth's architecture demonstrates a strong commitment to security and compliance. The system is well-designed for its purpose of secure LLM experimentation, and our deep code review did not uncover fundamental flaws. We observed that the development team has built in numerous controls (from data classification to audit logging) that directly align with government standards. This significantly lowers the risk profile of the application. Therefore, we recommend a **"Go" for ATO** once the few critical remediation items are addressed. These items (detailed below) are relatively minor and the team has the capacity to resolve them in a short timeframe. There are no systemic architectural issues that would warrant a major redesign or a "No-go."

Key Strengths: - The architecture is **modular and self-contained**, minimizing external dependencies and attack surface. - **Security by design** is evident: mandatory data classification, sanitized outputs, and cryptographic signing of results provide confidence in safe operations and traceability [71] [26]. - The presence of an **extensive test suite and documentation** indicates a high maturity level, reducing the likelihood of hidden bugs and easing future audits. - The design decisions (like using a CLI tool with plugins) smartly trade off complexity for control – it's easier to lock down a CLI tool on a secure server than to secure a distributed web service.

Critical Path to ATO: 1. **Complete Remediations:** The must-fix recommendations (removing dead code, finalizing config for production, etc.) should be completed immediately. These are straightforward: e.g., deleting the `old/` directory and adjusting a few default settings can be done within days. We advise a brief re-scan of the repo after removal to ensure no references remain. 2. **Update Artifacts:** Update the System Security Plan (SSP) and related ATO paperwork to reflect the changes and the current state of controls. For instance, note that legacy code was removed in version X.Y.Z, and include the improved config defaults as baseline. 3. **Obtain Approvals for External Services:**

Work with the accreditor to document the usage of OpenAI/Azure and include any required mitigations (like data handling procedures). This might involve providing a data flow diagram highlighting that segment [113] and referencing the contractual agreements in place for those cloud services. 4. **Conduct Final Acceptance Testing:** After changes, have the security team rerun any static code analysis or vulnerability scanning tools to get a clean report. Also execute an end-to-end test simulating a real use-case in a staging environment with all controls on, to show that the system works as expected under locked-down conditions (e.g., verify that if classification is SECRET, outputs are properly labeled and maybe that an unauthorized sink cannot be enabled). 5. **Prepare Operational Runbooks:** Ensure incident response plans (which are in docs) are finalized and someone is assigned to monitor the audit logs once in production. The ATO will expect that not only the tool is secure, but that the organization is ready to respond if something does go wrong (e.g., an audit log shows an anomaly or the LLM returns a result that should have been blocked).

Resource Requirements: No major new resources are needed for remediation – mostly developer time to implement fixes and possibly a bit of DevOps time to adjust deployment configurations. The core development is essentially done; remaining tasks are configuration, documentation, and process. For ongoing operation, assign a system owner who will: - Oversee patching of the application and its dependencies (estimated effort: a few hours per quarter, unless emergency patch needed). - Monitor the cost and rate logs (the cost tracker output) to ensure usage remains within expected bounds. - Monitor audit logs for any security-relevant events (this could be integrated into existing SOC monitoring; minimal additional effort if logs are aggregated properly).

Additionally, it would be wise to schedule a review 6 months post-ATO to assess if any improvements (like the should-fix and nice-to-have items) have been implemented and if the system usage has changed in ways that introduce new risks.

Go/No-Go Rationale: In summary, we see no “show-stopper” that would prevent Elspeth from operating securely in a production (including government production) environment. The identified issues are manageable: - *Security posture:* Solid, with multiple layers of defense. Just needs a bit of tightening (no loose ends like old code or user misconfigurations). - *Compliance alignment:* Very high – the system was clearly built with ASD Essential Eight and PSPF in mind, which is rarely the case for new tools. This greatly simplifies accreditation. - *Operational fit:* The lightweight design means it’s easy to deploy and requires minimal infrastructure changes. It will integrate into our environment without requiring complex new approvals (e.g., no database or web server needed).

Therefore, the recommendation is a **Go** for ATO once the conditions are met. The authorization can be granted on a **conditional basis** – i.e., the Authority grants ATO contingent on the development team demonstrating the completion of the must-fix items and providing updated documentation. Given the team’s responsiveness (as evidenced by their iterative improvements in 2025 notes) and the small scope of required changes, we have high confidence these conditions will be cleared quickly.

In conclusion, Elspeth is architecturally sound and security-hardened for an experimental platform. With the outlined refinements, it will be ready for secure operation in our enterprise. We will proceed to coordinate the final steps of the ATO process, and barring any unforeseen issues, Elspeth should receive its Authority to Operate and move into production use. This analysis will be attached to the ATO package as evidence of due diligence and thorough code-level review, thereby strengthening the case for approval. [71] [26]

---

1 113 component-diagram.md
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/docs/architecture/component-diagram.md

2 3 18 37 38 65 66 108 109 orchestrator.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/core/orchestrator.py

4 111 README.md
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/README.md

5 6 12 13 14 15 16 17 43 44 45 46 61 62 63 64 83 93 102 103 104 runner.py
https://github.com/tachyon-beep/elspeth/blob/c5144ea2e0674a0966b00955270b9ea372aa1b21/src/elspeth/core/experiments/runner.py

7 28 35 42 80 94 openai_http.py
https://github.com/tachyon-beep/elspeth/blob/c5144ea2e0674a0966b00955270b9ea372aa1b21/src/elspeth/plugins/nodes/transforms/llm/openai_http.py

8 72 74 75 76 106 excel.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/plugins/outputs/excel.py

9 107 110 SOURCES.txt
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth.egg-info/SOURCES.txt

10 26 27 29 77 78 96 98 100 101 105 signed.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/plugins/outputs/signed.py

11 20 21 34 41 49 82 interfaces.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/core/interfaces.py

19 70 types.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/core/types.py

22 73 _sanitize.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/plugins/outputs/_sanitize.py

23 24 25 91 92 97 middleware.py
https://github.com/tachyon-beep/elspeth/blob/c5144ea2e0674a0966b00955270b9ea372aa1b21/src/elspeth/plugins/nodes/transforms/llm/middleware.py

30 31 67 84 85 86 87 88 rate_limit.py
https://github.com/tachyon-beep/elspeth/blob/c5144ea2e0674a0966b00955270b9ea372aa1b21/src/elspeth/core/controls/rate_limit.py

32 33 89 90 cost_tracker.py
https://github.com/tachyon-beep/elspeth/blob/c5144ea2e0674a0966b00955270b9ea372aa1b21/src/elspeth/core/controls/cost_tracker.py

36 112 README.md
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/docs/architecture/README.md

39 40 50 51 52 53 54 config.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/config.py

47 48 56 plugin_scaffold.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/scripts/plugin_scaffold.py

55 main.py.historical
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/old/main.py.historical

57 58 68 69 plugin_registry.py
https://github.com/tachyon-beep/elspeth/blob/c5144ea2e0674a0966b00955270b9ea372aa1b21/src/elspeth/core/experiments/plugin_registry.py

59 60 81 95 99 cli.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/cli.py

71 __init__.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/core/security/__init__.py

79 signing.py
https://github.com/tachyon-beep/elspeth/blob/90e8d02392c05621fe48073defa3b4bfee4041bc/src/elspeth/core/security/signing.py