Bachelor Data Science and Artificial Intelligence

Constraint-Verified Agentic Loop

for FPV Drone Component Compatibility

Matīss Bahšteins

First supervisor:
Marc Hilbert
Second supervisor:
Joost Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 1, 2026

Abstract

Background. First-Person View (FPV) drones are modular systems assembled from components such as flight controllers, motors, batteries, cameras, video transmitters, receivers, frames, and goggles. These parts are usually sold and documented separately, although a working build depends on shared electrical, mechanical, protocol, radio-control, and video-system requirements. The evidence needed to judge those requirements is scattered across product pages, datasheets, firmware documentation, guides, and community discussions. With access to large language model (LLM)-based assistants, people assembling or modifying FPV drones may try to answer compatibility questions by asking whether selected components work together instead of manually cross-referencing all of these sources. Retrieval-Augmented Generation (RAG) is one way for such an assistant to leverage the scattered evidence, because it can condition answers on retrieved product and knowledge context. However, retrieval alone does not guarantee that a generated compatibility verdict is faithful to the evidence or to the applicable constraints.

Aim. This thesis investigates whether deterministic compatibility checking can improve the reliability of retrieval-augmented FPV compatibility support.

Method. In this thesis, we propose the Constraint-Verified Agentic Loop (CVAL), a neuro-symbolic FPV compatibility assistant that combines product and knowledge retrieval with enriched DIYFPV.com product specifications and a Symbolic Knowledge Engine for deterministic rule execution. We evaluate CVAL against standard Retrieval-Augmented Generation and rule-prompted Retrieval-Augmented Generation on a 170-question benchmark with compatible, incompatible, and unverifiable verdicts across five language models.

Results. On this benchmark, CVAL gives the highest judged verdict accuracy for every evaluated model and reduces false-compatible approvals across the full model suite. GPT-5.5 high with CVAL gives the strongest absolute result, reaching 89.9% judged accuracy and reducing false-compatible approvals to three cases. Among the lower-cost models in the suite, DeepSeek V4 Flash and DeepSeek V4 Pro achieve lower absolute accuracy than GPT-5.5 high, but do so at a fraction of the evaluation cost, which is relevant for FPV support settings where many compatibility questions may need to be answered at scale.

Conclusion. CVAL improves FPV compatibility support by routing bounded compatibility decisions through explicit rule execution over structured specifications, while still using retrieval for product discovery and explanation. More generally, this case study provides evidence for the usefulness of a neuro-symbolic agent-collaboration pattern in constraint-bounded domains where natural-language assistance must respect explicit technical rules.

Contents

1 Introduction

First-Person View (FPV) drones are modular unmanned aerial systems whose pilots select and integrate many types of components, such as flight controllers, electronic speed controllers (ESCs), motors, propellers, batteries, cameras, video transmitters, receivers, frames, goggles, and related accessories. Although these components are usually sold and documented separately, they must satisfy compatibility requirements when combined in a complete build. These requirements include electrical, mechanical, protocol, firmware, radio-control, and video-system constraints, among others. Market reports indicate continued growth in FPV drone adoption, with the FPV market projected to reach USD 1.3 billion by 2032 [19, 26]. However, this growth does not remove the technical entry barrier for users who need to assemble or modify their own systems. Relevant knowledge is scattered across product pages, datasheets, firmware documentation, tutorials, and community discussions, creating a fragmented information environment in which users must cross-reference many sources before they can decide whether parts work together.

The resulting support problem is more specific than ordinary information retrieval because a compatibility question usually requires identifying the intended products or component types, finding the relevant specifications, and then applying a constraint across two or more components. A wrong product variant, a missing specification, or a rule applied to the wrong component category can change the final answer even when the explanation appears plausible. This matters because compatibility support is often verdict-oriented. A person asking whether a battery, receiver, video transmitter, or frame works with another part does not only need a list of related facts, but a decision that preserves the difference between approval, rejection, and insufficient evidence. As such, approving a combination when the required evidence is incomplete can be more harmful than giving a cautious answer, because the response may guide an actual purchasing or build decision.

Conversational AI offers a possible way to help with the retrieval and explanation parts of this problem. Retrieval-Augmented Generation (RAG) can ground responses in external product and knowledge evidence [15, 12, 25], while ReAct-style agents show how language models can combine reasoning with external tool calls that retrieve information, query structured sources, or invoke domain-specific procedures [29]. However, standard RAG remains vulnerable to hallucinations and faithfulness errors, where generated statements are unsupported by or contradictory to the retrieved evidence [16, 20]. In a constrained hardware domain, these errors can become compatibility violations. For example, a fluent answer may approve a component combination despite mismatched voltage limits, connector types, mounting dimensions, radio protocols, or video-system requirements.

One way to address this problem is to separate the parts of the task that benefit from language-model flexibility from the parts that require explicit rule execution. Neuro-symbolic AI studies this type of combination, where neural models are paired with symbolic representations and rule-based procedures [4]. This distinction is useful for FPV compatibility support because the language model can interpret the user's question, retrieve relevant product and knowledge evidence, and explain the result in natural language, while the compatibility verdict can be constrained by explicit checks over structured component data.

In this thesis, we propose the Constraint-Verified Agentic Loop (CVAL) to address this separation by adding an explicit verification step to a large language model (LLM)-based assistant for FPV compatibility support. Before the agent finalizes a compatibility answer, it can route selected

products to a Symbolic Knowledge Engine (SKE), which performs deterministic compatibility checking over structured product specifications. Rather than providing additional text for open-ended interpretation, the SKE executes compatibility rules and returns a bounded verdict signal that the agent must take into account. This shifts the compatibility decision away from retrieved text alone or rules written only into the prompt, and instead treats component compatibility as a constraint-checking problem.

We develop and evaluate the system using DIYFPV.com as its case-study environment. DIYFPV.com is an FPV community and product platform for drone pilots and builders, combining community discussions, troubleshooting, product discovery, price and stock comparison, a marketplace, and an interactive build-planning tool [7]. The platform has more than 15,000 members, and its catalogue contains more than 40,000 products from over 30 stores. This combination of catalogue scale, build tooling, and support context makes it suitable for studying compatibility assistance in a realistic service setting. The product records provide the basis for specification extraction and compatibility-rule development, and are enriched with structured compatibility specifications that can be retrieved by the agent and inspected by deterministic rules.

In our evaluation, we compare three agent modes that use the same prepared product data and the same benchmark questions. Standard RAG retrieves product and knowledge context, then answers directly. Rule-prompted RAG receives the same retrieval tools plus compatibility rule logic in the prompt, but it cannot execute the rules. CVAL can call the same RAG and product tools, and in addition can invoke the deterministic checker before giving a final answer.

The central claim tested here is not that CVAL produces more polished explanations, but that executing compatibility rules can improve verdict reliability compared with retrieval alone or rules written only into the prompt. FPV compatibility is a suitable setting for this comparison because many user questions can be reduced to bounded verdicts of compatible, incompatible, or unverifiable, where unverifiable means that there is insufficient reliable evidence to determine compatibility. The study therefore treats FPV compatibility both as a concrete application and as a case study for a broader neuro-symbolic agent pattern in constraint-bounded domains, where natural-language support must respect explicit rules or constraints.

2 Research Question

The introduction motivates CVAL as a way to move compatibility decisions from retrieved text alone toward executable rule checking. The research question therefore asks whether this additional verification step improves reliability when compared with two baselines that use retrieval without deterministic rule execution. Although the empirical evaluation is specific to FPV compatibility, the question is also motivated by the broader problem of using LLM agents in domains where answers must obey explicit constraints.

Main RQ: To what extent does a Constraint-Verified Agentic Loop improve compatibility-verdict reliability for FPV drone components compared with standard Retrieval-Augmented Generation and rule-prompted Retrieval-Augmented Generation?

The sub-questions connect this comparison to the main evaluation dimensions. The first asks how the effect varies across verdict classes. The second focuses on false approval, which is the most

safety-relevant failure mode in a hardware support setting. The third asks whether any reliability gain remains meaningful when SKE use, latency, token usage, and cost are considered across model categories.

RQ1 (Verdict classes): How does CVAL affect verdict accuracy for compatible, incompatible, and unverifiable benchmark cases compared with the two RAG baselines?

RQ2 (False approvals): Does CVAL reduce safety-relevant errors, especially false-compatible answers where incompatible or insufficiently supported combinations are approved?

RQ3 (Runtime and resource effects): How do CVAL's accuracy, SKE use, latency, token usage, and cost vary across different categories of language models?

3 Background and Related Work

Three strands of prior work frame the problem addressed here. Practical FPV compatibility knowledge defines the constraint structure of the domain, retrieval and faithfulness research explain how external evidence can help while still leaving room for unsupported generated claims, and neuro-symbolic tool use provides a way to connect language interaction with executable checks.

3.1 FPV Compatibility as Bounded Hardware Reasoning

At its core, FPV component compatibility is a bounded form of hardware reasoning. A user question may be written in natural language, but the decision often depends on a finite set of component roles and specifications. Practical FPV build guides describe component selection in terms of power, mounting, radio-link, and video-system compatibility, among other constraints [17, 27, 9]. A battery, for example, may need to be checked against the input range of an ESC or flight controller; a motor and propeller may need matching mounting or shaft dimensions; a radio and receiver may need overlapping protocols and frequency support; and a video transmitter must match the supported video system of a camera or display. These examples differ in detail, but they share the same basic structure because compatibility depends on whether known product properties satisfy explicit constraints.

This structure motivates the three verdict classes used later in the evaluation. A component combination is treated as *compatible* when the relevant specifications are available and satisfy the applicable compatibility rules. It is treated as *incompatible* when known specifications violate such a rule. Lastly, it is treated as *unverifiable* when the available product specifications, retrieved evidence, or rule inputs are insufficient to determine compatibility reliably. This class is important because incomplete evidence should not be treated as implicit compatibility approval.

3.2 Retrieval-Augmented Generation, Faithfulness, and Evaluation

Because FPV compatibility questions depend on both product evidence and domain-specific knowledge, an assistant needs access to information beyond the language model's parametric knowledge. RAG addresses this need by retrieving external context before generating an answer [15]. In many implementations, the retrieval step uses dense retrieval, where queries and documents

are represented as dense vector embeddings and compared by semantic similarity. This makes it possible to retrieve relevant passages even when they do not use the exact same wording as the user question [12]. For FPV support, this means that a model can answer with access to product descriptions, structured product records, and curated explanatory documents.

However, retrieval does not by itself guarantee that the generated answer is faithful to the retrieved evidence. Prior work shows that retrieval can reduce hallucination in conversation [25], but faithfulness errors can still occur when generated claims are unsupported by or contradictory to the provided evidence [16, 20]. In FPV compatibility support, such errors are especially problematic because a response can appear evidence-based while still reaching an incorrect compatibility verdict.

Evaluating such systems therefore also requires attention to whether generated answers reach the correct task outcome rather than only whether they sound plausible. ARES is one example of a RAG-specific evaluation framework that assesses generated answers against domain data, labelled examples, and judge models [24]. It fine-tunes lightweight judge models on synthetic training data and uses a small human-annotated set to mitigate prediction errors. More broadly, LLM-as-judge work shows that language models can be used to evaluate generated answers at scale, while also documenting biases such as position and verbosity effects that motivate human review of selected cases [30].

3.3 Neuro-Symbolic Tool Use and Executable Constraints

The limitations of retrieval-only generation motivate a broader class of systems that combine learned models with explicit symbolic structure. Neuro-symbolic AI studies the integration of machine learning with symbolic representations, rule systems, and automated reasoning as a way to combine the flexibility of neural models with the interpretability and constraint handling of symbolic methods [4]. One line of work incorporates symbolic knowledge during model training, for example by distilling first-order logic rules into neural networks [11]. This shows that symbolic constraints can guide neural predictions, but it still places the constraint mechanism inside the learned model.

CVAL follows the same high-level motivation while placing the symbolic component in the agent runtime. In a tool-using agent, the language model does not only generate an answer from prompt context, rather, it can also call a search function, inspect structured records, or run a domain-specific procedure and then use the returned result in its response. ReAct formalizes this style of agent by interleaving language-model reasoning with actions, including tool calls [29]. This makes tool use the practical mechanism by which an LLM can access external operations that it cannot reliably perform from text alone.

This runtime placement is also close to recent LLM systems that use the model to translate or organize a problem while delegating the final inference step to an external symbolic mechanism. Logic-LM, for example, uses an LLM to translate natural-language logical reasoning problems into symbolic formulations and then relies on a deterministic symbolic solver for inference [23]. In FPV compatibility support, the same idea motivates a separation between evidence gathering and verdict checking, where the model retrieves and organizes context while the SKE performs bounded compatibility checks over structured product specifications.

4 Solution Design

We implement CVAL through four main parts: the DIYFPV case-study data, the product-specification enrichment process, the deterministic compatibility-rule model, and the agent tools that connect these resources at runtime, as shown in Figure 1. The LLM agent retrieves product and knowledge context to interpret a user question, while the SKE evaluates selected products against explicit compatibility rules. Across the implementation, model calls are routed through OpenRouter, which provides a unified API for accessing language models from multiple providers and follows the widely used OpenAI-compatible chat and embedding request format [22]. Because the SKE cannot use free-form catalogue text as rule input, DIYFPV.com product data must first be prepared as structured specifications. The design description begins with the case-study data sources, then moves to specification enrichment and the deterministic rule model.

4.1 Case-Study Environment and Data Sources

As introduced above, DIYFPV.com provides the case-study environment because it combines catalogue-scale FPV product data with a community support context [7]. This makes it suitable for studying compatibility support in a realistic setting, where questions concern concrete products, store listings, and build situations that resemble the decisions pilots face when selecting parts.

The platform distinguishes canonical product records from store-specific product listings. Canonical records provide the normalized product identity, while store-level listings preserve retailer-specific evidence such as listing wording, prices, stock, and URLs. In addition, structured product specifications form a product-linked layer over these records and provide the basis for later compatibility checking.

In addition to product and store-listing data, supporting evidence sources are used for the assistant and the evaluation design. Community support content, including troubleshooting questions and posts, provides realistic examples of how users ask compatibility-related questions, and pre-existing internal guide material from DIYFPV was converted into usable knowledge context about FPV concepts, compatibility criteria, and common build decisions. Product and catalogue data therefore form the basis for deterministic compatibility inputs, while community content and RAG documents mainly support explanation, context, and realistic question design.

4.2 Product Specification Enrichment

The compatibility rules used by the SKE can only operate on explicit and comparable product properties. A rule that checks whether a battery fits an ESC, for example, needs structured values for cell count, voltage limits, connector type, and related fields rather than a free-form product description. Product names and listing descriptions often contain this information, but they mix dimensions, protocols, voltage ranges, included accessories, marketing text, and store-specific wording in a form that cannot be checked directly. Catalogue products are therefore enriched with structured specifications before they are used as rule inputs.

At the database level, the enrichment pipeline uses only the product-specific parts of the DIYFPV data model. Canonical product records provide the main product identity and baseline descriptions, store listings add retailer-specific wording, categories determine which extraction schema applies,

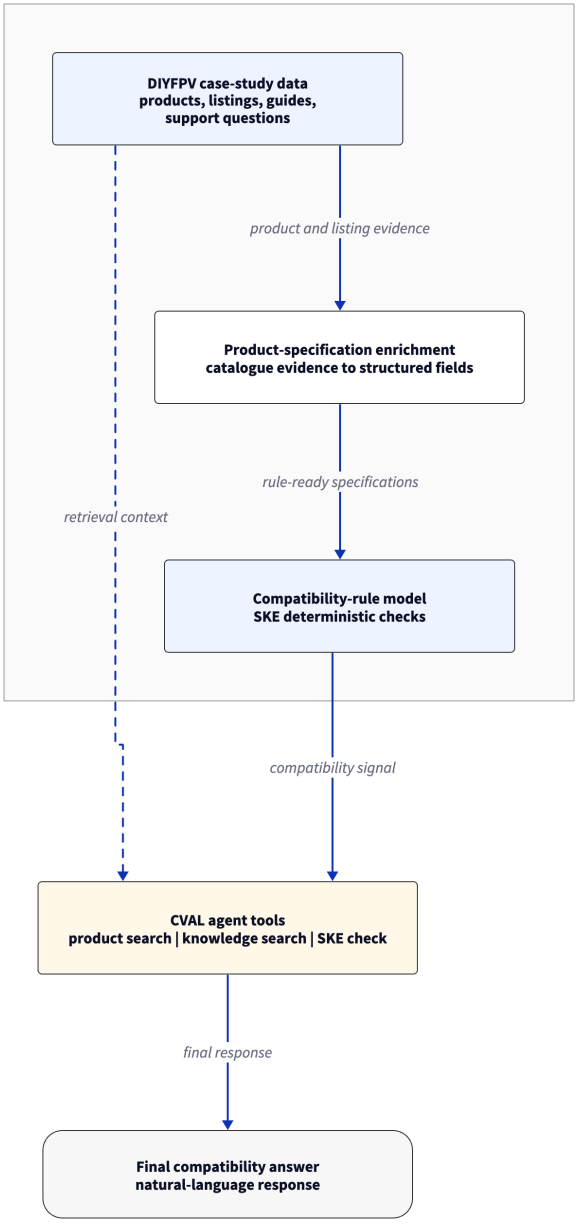

Figure 1: High-level overview of the CVAL design. Solid arrows show the compatibility-preparation path from DIYFPV product data to structured specifications, deterministic checks, and the final natural-language answer. The dashed arrow shows that the case-study data also supports retrieval context for the agent.

and accepted outputs are persisted as structured product specifications. Table 1 summarizes these entities and fields.

Entity	Relevant fields	Use in the enrichment pipeline
`Product`	`id`, `name`, `description`, `descriptionHtml`, `visible`, `hidden`, `categoryId`, `manufacturerId`	Canonical product record used as the main product identity and extraction source.
`ProductCategory`	`id`, `name`, `slug`, `type`, `parentCategoryId`	Maps products to FPV component categories and schema categories.
`Manufacturer`	`id`, `name`	Provides brand/manufacturer evidence for extraction and product display.
`StoreProduct`	`id`, `productId`, `storeId`, `name`, `description`, `descriptionHtml`, `url`, `price`, `stock`, `storeReference`	Store-level listing evidence, including retailer wording and availability data.
`Store`	`id`, `name`, `url`, `enabled`, `verified`, `region`, `baseCurrency`	Identifies the retailer context for store listings.
`ProductSpecification`	`id`, `productId`, `categoryId`, `type`, `value`	Stores the enriched structured specifications used by compatibility rules.

Table 1: DIYFPV database fields used by the product specification enrichment pipeline.

The specification schema covers 25 FPV-relevant product categories. In the database snapshot used for the extraction work, DIYFPV contained 71 product-category records, of which 60 contained visible products. The extraction schema selected 39 database categories and grouped them into the 25 schema categories used by the rule system. This grouping is necessary because the operational catalogue is organized partly for browsing and retail presentation. For example, frame products are split in the catalogue into separate categories for whoop frames, mini frames, five-inch frames, and larger frames, but they are treated as one *Frames* schema category for specification extraction. The same principle is applied to motors, propellers, and ready-to-fly drones, where retail subcategories are grouped under one extraction schema for the main component type. Categories that do not provide primary compatibility-rule inputs are excluded from specification extraction, including accessories, replacement parts, cables, tools, soldering supplies, bags, clothing, stickers, services, simulators, gift cards, racing gates, and broad miscellaneous categories.

Each schema category defines the fields that are meaningful for that component type rather than applying one universal product schema to the whole catalogue. The field selection is based on compatibility-relevant attributes in the DIYFPV catalogue and internal guide material, and cross-checked against practical FPV build references that discuss component selection, mounting, power, radio-link, and video-system constraints [17, 3, 27, 9]. For batteries, for example, relevant fields include `cell_count`, `capacity`, `battery_chemistry`, and `battery_connector`. For frames, the schema includes fields such as `max_propeller_size`, `supported_motor_mounting_patterns`, and

supported_board_mounting_patterns. Radio and video-system products require different fields again, including rf_protocols, module_bay_type, video_system, supported_frequency_bands, and control_protocols. The resulting values are normalized into typed representations, such as numbers with units, booleans, enumerated values, arrays, and ranges.

The enrichment pipeline is organized as the workflow shown in Figure 2. It starts by selecting product candidates from mapped FPV categories. For each candidate product, the extractor assembles evidence from the canonical product record, manufacturer name, product description, HTML description, and store-level listing titles and descriptions. The main full-scale extraction run used google/gemini-2.5-flash-lite through the shared model-calling interface. We selected this model after pilot extraction runs on sampled catalogue products. The first model-comparison pilots used a 117-product sample and compared Gemini 2.5 Flash Lite, Gemini 2.0 Flash, GPT-4o mini, GLM 4.7 Flash, and Qwen 3.5 Flash. The comparison considered JSON validity, extracted real fields, unsupported fields, and estimated catalogue-scale cost. GPT-4o mini extracted more fields on the early sample, but Gemini 2.5 Flash Lite also reached 100% valid JSON, improved field coverage over Gemini 2.0 Flash, and kept the estimated catalogue-scale cost lower than the other higher-coverage candidates. Larger catalogue-only pilots with the same model completed 500 and 1,224 products with 100% valid JSON and low junk-field rates, which supported using it for the full run. We use a primary-product filtering step before full extraction because DIYFPV aggregates retailer listings into catalogue categories, and those retailer/category signals can include secondary items alongside complete components. Accessories, mounts, cables, replacement parts, and bundles therefore need to be identified before schema extraction. This is important because a replacement arm in the frame category, for example, should not receive the same compatibility specification structure as a complete frame.

The prompt assembly step produces the schema prompt shown in Figure 2. It asks the model to return JSON only, to omit fields when the evidence is unavailable, and to avoid placeholder values such as unknown or not applicable. After the OpenRouter call returns JSON output, the JSON validation and normalization step applies the validation rules from the category schema. It attempts repair for malformed outputs when needed, removes unsupported fields, normalizes aliases and enum values, and writes the accepted fields to ProductSpecification as structured product specifications. This validation is necessary because deterministic rules should only receive fields that are supported by the source evidence and accepted by the schema.

In the clean full-scale extraction run, the pipeline completed 16,419 products. It extracted 227,813 real fields out of 308,536 possible fields, corresponding to an overall extraction rate of 73.8%. The run produced valid JSON for 98.6% of products; the remaining cases were outputs that could not be parsed or repaired into the required JSON structure. The junk-field rate was 3.4%, meaning that this share of extracted fields was rejected during validation because the field was unsupported by the schema or had an invalid value. The estimated LLM cost was USD 11.89. Figure 3 shows that extraction rates varied by category, with highly structured categories such as batteries and antennas producing higher field coverage than categories whose product records mix multiple subtypes or depend on less consistently reported fields.

Several evidence-source variants were also tested before settling on the final pipeline. These included enriched prompting, DataForSEO product information [6], Firecrawl page scraping [8], and combined external retrieval or scraping setups. On 117-product samples, DataForSEO and Firecrawl runs

produced field coverage in the mid-50% range, close to the strongest early model-only pilots. A larger DataForSEO run over 1,224 products reached 65.7% field coverage, compared with 63.2% for the catalogue-only pilot of the same size, but the estimated 19,000-product cost increased from about USD 8 to about USD 46. The external-source pilots also introduced operational dependencies and match-quality risks. For example, some DataForSEO matches were flagged as suspicious when a catalogue product such as *Flite Test FT EZ ID - Remote ID Module w/ Plug* was matched only to a generic title such as *EZ ID*. The final pipeline therefore uses internal catalogue and store evidence as the primary source, with primary-product filtering, strict schema prompting, validation, and repair as the main safeguards.

4.3 Compatibility Rule Model

After product specifications have been enriched into structured fields, they can be used as inputs to deterministic compatibility rules. Each rule reads a defined set of product specifications, applies a bounded compatibility condition, and returns a verdict that can be used by the agent when forming its final answer.

The current compatibility reference contains 48 active rules: 35 pairwise rules and 13 build-level rules. Pairwise rules evaluate compatibility between two selected components, such as a battery and ESC, a frame and motor, a radio and receiver, or a video transmitter (VTX) and antenna. Build-level rules evaluate constraints that depend on a larger component set, such as whether a build has enough universal asynchronous receiver-transmitters (UARTs), control outputs, motors, or regulated power capacity. Table 2 gives representative examples from both rule types, and Figure 4 shows the pairwise rule relationships.

Rules are designed to return one of three practical outcomes. A combination is *compatible* when the relevant specifications are known and satisfy the applicable rule. It is *incompatible* when known specifications violate the rule. It is *unverifiable* when required specifications are missing, ambiguous, or not represented by the current rule inputs. This treatment of unverifiable cases is important because a safe assistant should not present missing evidence as proof of compatibility.

Rules are grouped by their primary compatibility area for explanation and later analysis. These groups are not a strict taxonomy, especially for build-level checks, because a build-level rule can combine component presence, electrical limits, and resource constraints. The main areas are electrical compatibility, mechanical fit, radio-control protocol and frequency matching, video-system compatibility, antenna compatibility, frame and mounting constraints, and build-level resource constraints.

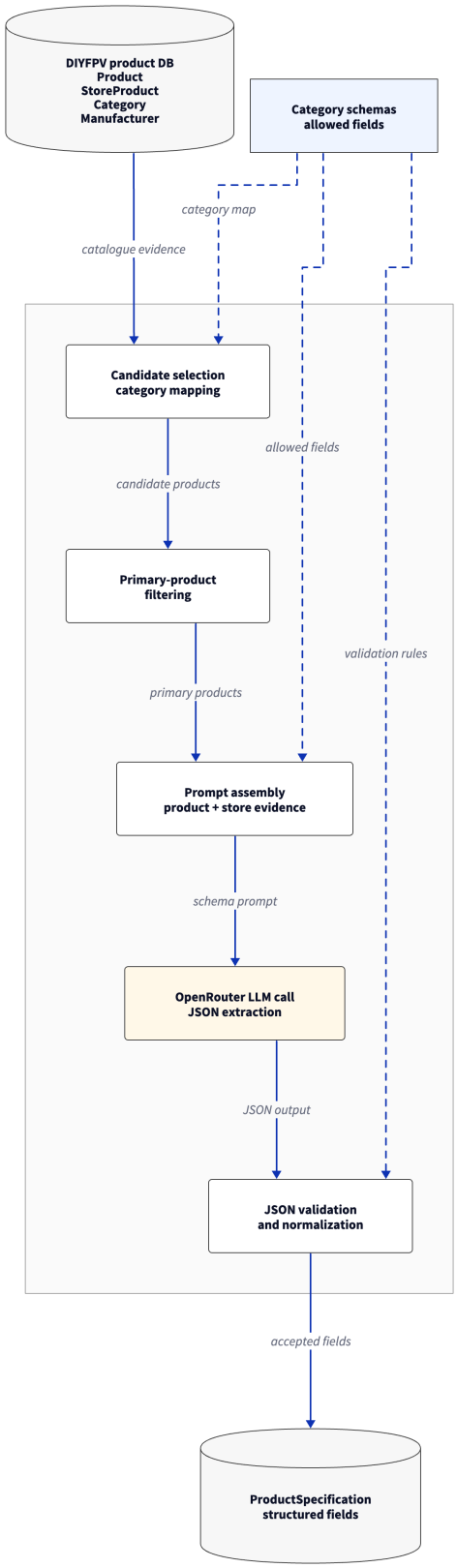

Figure 2: Product specification enrichment workflow. Solid arrows show the main data and processing flow. Dashed arrows show schema and configuration inputs.

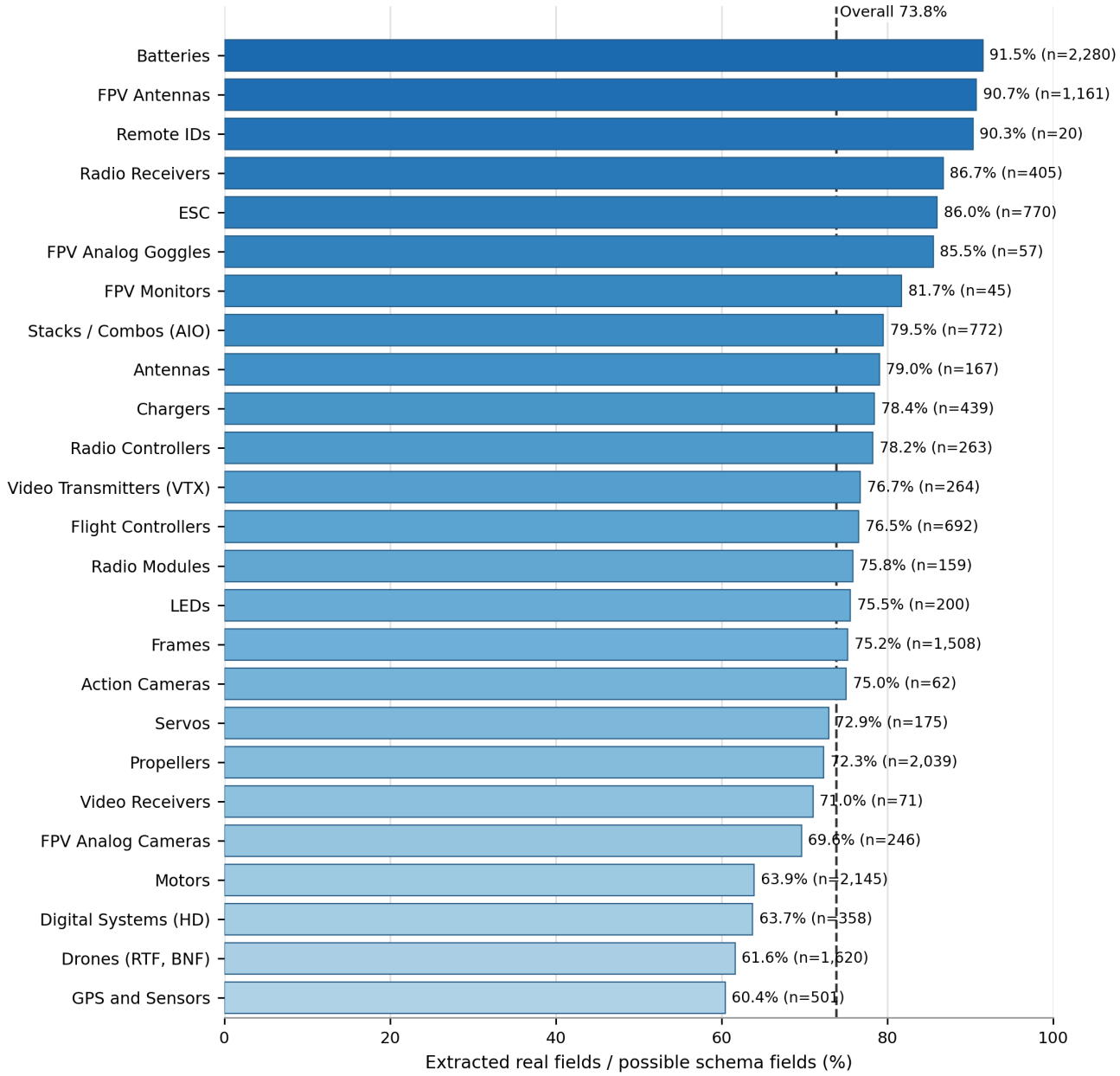

Figure 3: Specification extraction rates by product category for the clean full-scale extraction run. The dashed line marks the overall extraction rate of 73.8%.

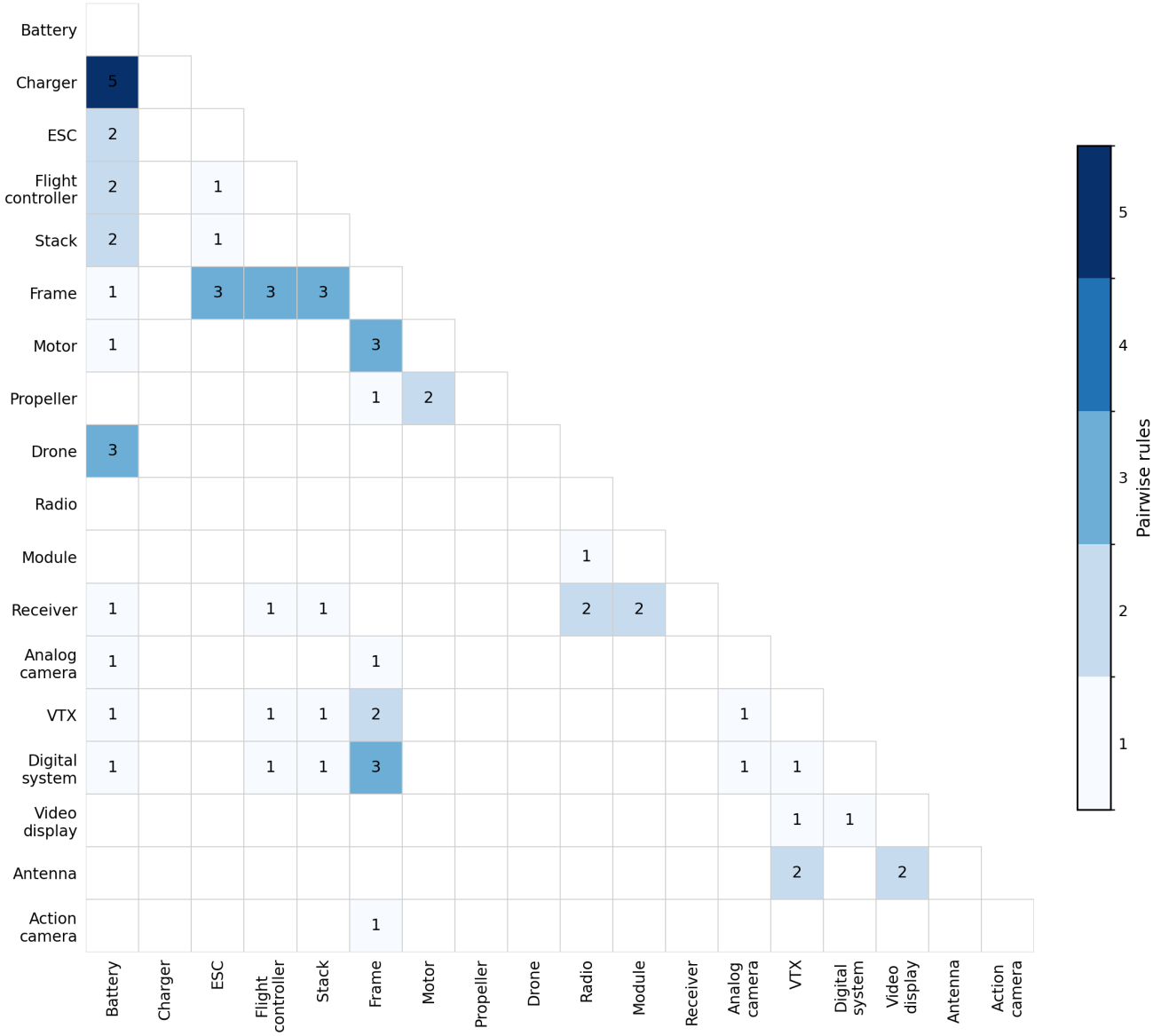

Figure 4: Pairwise compatibility-rule relationships between component roles. Non-empty cells mark role pairs covered by at least one deterministic pairwise rule; the number in each cell and the color intensity indicate how many pairwise rules cover that component pair.

Primary area	Rule identifier	Required inputs	Compatible outcome
Electrical	battery_esc_cell_count	battery.cell_count; esc.input_cell_count_min/max	Cell count in range.
Mechanical	frame_prop_max_size	frame.max_propeller_inches; propeller.diameter	Propeller fits frame.
Mechanical	motor_prop_mount_type	motor.prop_mount_type; propeller.prop_mount_type	Mounts match.
Radio control	radio_receiver_protocol	radio.rf_protocols; receiver.receiver_protocol	Protocol overlaps.
Video system	camera_vtx_signal	camera.video_system; vtx.video_system	Signal matches.
Build resource	build_uart_budget	flight_controller.uart_count; UART-consuming devices	UARTs suffice.

Table 2: Representative pairwise and build-level compatibility rules used by the Symbolic Knowledge Engine, grouped by primary rule area.

4.4 RAG Knowledge Curation and Retrieval Setup

With the rule inputs and compatibility rules defined, the agent also needs a retrieval source for explanatory FPV knowledge. We build the retrieval component from unpublished internal DIYFPV guide material that had been prepared for future knowledge and support features. We convert this material into a local knowledge corpus so that the agents can condition their answers on the same stable explanatory sources, following the general RAG idea of grounding generation in an external document collection [15]. The resulting corpus contains 54 guide documents across 26 FPV topic categories, covering areas such as batteries, antennas, video systems, control links, build skills, and getting-started concepts.

Each knowledge source was normalized into a concise local text document with explicit metadata. This follows common RAG ingestion practice, where documents are split into retrievable units and metadata such as titles, source identifiers, and topic labels help preserve context after chunking [28, 18]. In this corpus, each document begins with a title, category, and tags, followed by short guide-style sections that separate summaries, key concepts, and practical guidance. Listing 1 shows an excerpt from the *FPV Motor Selection Basics* document to make the source format explicit. A separate manifest records each source identifier, title, type, category, and file path, which makes the corpus reproducible and allows individual sources to be inspected without changing the retrieval code.

```
Title: FPV Motor Selection Basics
Category: motors
Tags: motor, brushless, brushed, kv, stator-size, thrust, thrust-to-weight,
    propeller, prop-size, pitch, battery-voltage, cell-count, esc-current,
    mounting-pattern, shaft, prop-mount, overheating

FPV motor selection summary:
Motor part selection depends on the complete power and mechanical system: frame
    prop size, battery voltage, motor KV, stator size, propeller load, ESC current
    rating, motor mounting pattern, prop mounting type, and aircraft weight. A
    motor is not "best" in isolation; it is only appropriate for a specific build
    goal.

FPV motor key concepts:
- Modern FPV drones usually use brushless motors.
- Brushed motors are mostly found on very cheap or older micro drones.
- Brushless motors have three wires connected to the ESC.
- Swapping any two motor wires reverses motor direction, and many ESCs can also
    reverse direction in software.
- Motor KV describes no-load RPM per volt.
```

Listing 1: Excerpt from a retrieval-ready DIYFPV guide document.

During ingestion, the guide documents are loaded into the knowledge index. They are compact, with a median length of 2,278 characters and a range from 1,188 to 3,861 characters. Each document is split on paragraph boundaries where possible and then grouped into chunks of up to 2,000 characters with a 200-character overlap. The overlap preserves nearby context across chunk boundaries while keeping each retrievable unit short enough for targeted retrieval [28, 13]. For this corpus, the configuration produces 100 chunks from 54 documents, usually one or two chunks per document. Each chunk is embedded with `google/gemini-embedding-2-preview` through OpenRouter [10, 21]. The model was selected because it was a recent Google embedding model released in public preview in March 2026, was available through the same model-calling interface used elsewhere in the system, supports semantic retrieval for RAG, and provides 3,072-dimensional embeddings that can be used consistently for both document ingestion and query-time retrieval. The manifest itself remains a local ingestion file rather than a database table; the persisted retrieval data is stored in the `KnowledgeChunk` table, as summarized in Table 3, and forms the knowledge base queried by the agent's retrieval tools at runtime.

Source	Relevant fields	Use in retrieval setup
`rag-sources.json` manifest	`id`, `title`, `type`, `category`, `path`	Local ingestion configuration that records which guide documents are embedded. It is not stored as a database entity.
`KnowledgeChunk` database entity	`id`, `content`, `embedding vector(3072)`, `sourceTitle`, `sourceType`, `chunkIndex`, `metadata`, `createdAt`	Stores the retrievable text unit, its embedding vector, and source-level fields used in retrieval output.
`KnowledgeChunk.metadata` JSON field	`source_id`, `category`, `content_hash`, `embedding_model`, `embedding_dimensions`, `chunk_size`, `chunk_overlap`, `ingested_at`	Records ingestion settings so the corpus can be inspected, refreshed, or compared across embedding runs.

Table 3: Manifest fields and database fields used by the RAG ingestion pipeline.

4.5 Agent Architecture and Modes

The runtime design has to support both open-ended evidence gathering and bounded verification. The evaluated modes therefore share the same basic agent loop, while differing in the prompts, tools, and verification behavior made available to the language model.

4.5.1 Graph-Based Agent Loop

Figure 5 gives an overview of the graph-based agent loop. We implement the tool-calling workflow with LangGraph, which represents agent execution as explicit nodes and conditional edges [14]. This design makes the tool loop and the CVAL-specific verification step easier to control than in a single linear chain. The graph maintains a shared agent state containing the conversation messages, user and conversation identifiers, the selected agent mode, and optional model configuration. The main assistant node invokes the chat model with a mode-specific system prompt and a mode-specific tool set. When the model emits tool calls, a tool-execution node runs the requested tools and returns their outputs to the assistant node, allowing the agent to iterate before producing a final answer. We use this structure because FPV compatibility support is not a single-pass retrieval task. The agent may need to search for candidate products, inspect product specifications, retrieve explanatory knowledge, run deterministic checks, and then decide whether the available evidence supports a compatibility verdict.

The graph exposes four tools to the assistant. `search_products` retrieves candidate catalogue products through semantic product search and optional category or brand filters. `get_product_details` retrieves the full saved product record and structured specifications for a selected product. `search_knowledge` retrieves curated guide chunks from the RAG corpus for technical background

and explanation. `check_compatibility` invokes the Symbolic Knowledge Engine on selected catalogue products or manually specified products and returns deterministic rule outcomes. The first three tools are available to all agent modes, while `check_compatibility` is exposed only in CVAL mode.

CVAL also adds a guard node after the assistant node. If the assistant attempts to answer a compatibility question without having called `check_compatibility`, the guard injects a corrective message and routes the state back to the assistant. The guard is not part of the SKE itself, but a control mechanism that enforces the intended CVAL workflow when the question appears to require deterministic verification. Operationally, the graph detects completed checker use through the presence of a `check_compatibility` tool message. If the assistant still reaches a final answer without such a tool result, the guard emits an unverifiable fallback answer. Separately, when the checker is called but no applicable rule or sufficient input data is available, the checker output itself is unverifiable.

4.5.2 Agent Modes

The graph structure shown in Figure 5 is instantiated in three agent modes. All three modes use the same product data, RAG corpus, benchmark questions, and basic assistant-tool loop, but they differ in how compatibility rules are made available to the language model. This split separates retrieval quality from rule execution. Standard RAG tests whether retrieved product and guide context is enough for reliable verdicts. Rule-prompted RAG tests whether explicit rule text improves reasoning without changing the available tools. CVAL keeps the same retrieval tools but adds deterministic rule execution and the guard node, making it possible to measure whether executable constraints improve compatibility-verdict reliability over prompt guidance alone. Table 4 summarizes the differences between the modes. In the rule-prompted mode, rule logic is provided as text in the system prompt, but the model must apply the rule itself, as shown in Listing 2.

Mode	Purpose	Callable tools
`standard_rag`	Retrieval baseline for product and knowledge-grounded answers.	`search_products`, `get_product_details`, `search_knowledge`.
`rule_prompted_rag`	Prompt-only rule baseline with compatibility rules written into the system prompt.	`search_products`, `get_product_details`, `search_knowledge`.
`cval`	Constraint-verified agent mode with executable SKE checks and the CVAL guard node.	`search_products`, `get_product_details`, `search_knowledge`, `check_compatibility`.

Table 4: Agent modes compared in the evaluation.

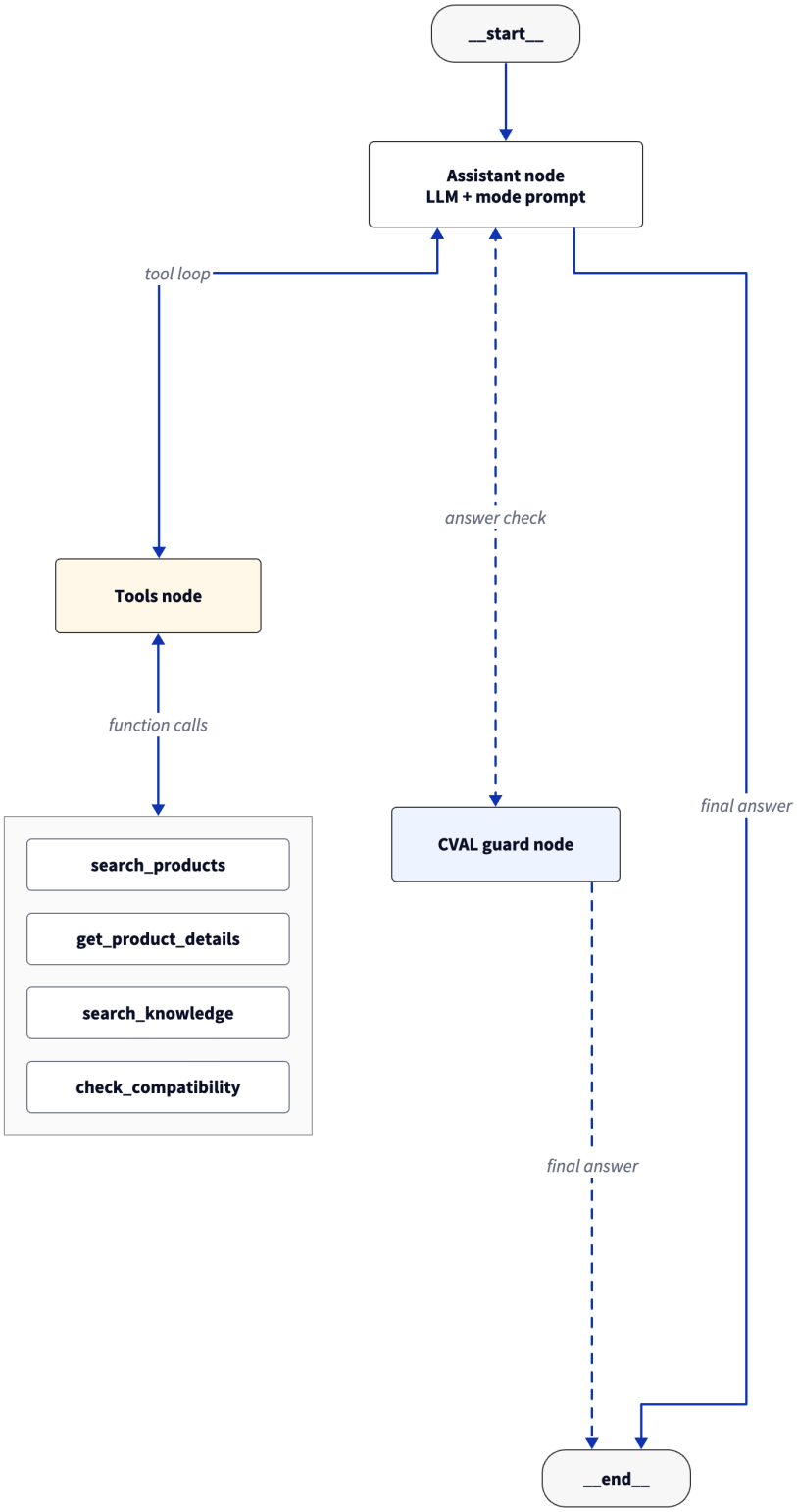

Figure 5: Agent graph structure based on the LangGraph implementation. Function boxes list the tool calls exposed through the tools node. Solid arrows indicate the standard assistant-tool loop and final-answer path, while dashed arrows indicate CVAL-specific guard transitions.

```
Rule: frame_motor_mount_pattern
APPLIES: frame + motor
READ: frame.supported_motor_mounting_patterns,
     motor.motor_mounting_patterns
UNVERIFIABLE: either pattern set is missing
PASS: intersection(frame_patterns, motor_patterns) is not empty
FAIL: both pattern sets are known and intersection is empty
```

Listing 2: Example compatibility-rule text included in the rule-prompted RAG system prompt.

4.5.3 Product Retrieval Tools

Product retrieval is separated into candidate search and product resolution. The `search_products` tool is used when the agent needs to identify likely catalogue products from a natural-language query. It embeds the query with the same embedding configuration used for product vectors and ranks visible, non-hidden catalogue products by vector similarity over `Product.embedding`. Optional category and brand arguments act as filters. The returned payload remains intentionally lightweight, containing product identifiers, catalogue metadata, a similarity score, and a short description excerpt. This makes the tool suitable for candidate discovery, but not for final compatibility decisions.

After a candidate product has been selected, `get_product_details` resolves the product by ID or slug and returns the full product context together with its saved structured specifications. The lookup joins the canonical product record with category and manufacturer data, then loads the corresponding `ProductSpecification` rows as specification key-value pairs. This second step is necessary because compatibility reasoning depends on structured fields rather than on search-result snippets. In CVAL mode, resolved product identifiers or specifications can then be passed to `check_compatibility`. In the RAG baselines, they form the product evidence available to the language model. Table 5 summarizes the two product retrieval tools.

Tool	Input	Retrieval role
`search_products`	Natural-language query, optional category and brand filters, and result limit.	Candidate discovery over embedded catalogue products, returning lightweight product identifiers and metadata.
`get_product_details`	Product ID or slug from a selected candidate.	Product resolution through exact lookup, returning full product context and saved structured specifications.

Table 5: Product retrieval tools exposed to the agent.

4.5.4 RAG Knowledge Retrieval Tool

At runtime, `search_knowledge` queries the curated FPV guide corpus introduced in Section 4.4. The tool receives a natural-language query and an optional category hint, embeds the query, and

searches the stored `KnowledgeChunk` records. The implementation uses a hybrid ranking scheme because dense retrieval and lexical retrieval capture different signals in RAG systems [12, 28]. Dense similarity helps when the user's wording differs from the guide text, while full-text ranking helps preserve exact FPV terms such as protocol names, connector types, and component labels. The final score uses 75% semantic similarity and 25% normalized full-text relevance. If the optional category hint matches the chunk metadata, the chunk receives a small boost, but the category is not used as a hard filter. This avoids hiding useful adjacent guidance when the model's category guess is plausible but too narrow.

The tool returns a small set of ranked chunks, with three results by default and at most five. Each result includes source metadata, retrieval scores, and retrieved chunk content, so the retrieved guidance remains traceable to a specific corpus entry. More complex retrieval variants, such as separate rank fusion or a learned reranking stage, were not used in this implementation [5, 28]. The guide corpus is small, and the goal was to keep the retrieval path inspectable while still combining semantic and lexical evidence.

4.5.5 Compatibility Checker Tool

The `check_compatibility` tool is the CVAL-only interface between the language-model agent and the Symbolic Knowledge Engine. Figure 6 gives an overview of how this tool resolves selected products, applies pairwise and build-level rules, and returns structured guidance to the agent. Unlike the product and knowledge retrieval tools, it does not retrieve additional evidence for free-form interpretation. Instead, it receives selected products, maps them to the component roles used by the SKE, such as battery, ESC, receiver, or frame, and executes deterministic compatibility rules over the corresponding structured specifications.

The preferred input is a list of product IDs or slugs obtained from `search_products`. The tool resolves these identifiers to canonical product records and saved `ProductSpecification` fields before checking compatibility. For cases where the user describes a synthetic build rather than selecting catalogue products, the tool can also accept explicit product objects with manually stated specifications. An optional build profile can be passed when the question concerns a larger parts list rather than a single product pair. When no build profile is provided, the SKE evaluates all unique product pairs and applies the pairwise rules that match their component roles. When a profile is provided, it selects build-level checks such as the generic `build`, partial `five_inch`, or fuller `five_inch_complete` profiles. These checks cover multi-part constraints such as required roles, control-plane structure, component counts, serial resources, and power capacity.

The SKE gives priority to blocking failures when producing the verdict exposed to the agent. If any applicable required rule fails, the tool returns an incompatible result. If the applicable rules do not fail but require fields that are missing or ambiguous, the tool returns an unverifiable result. It returns a compatible result only when the applicable checks pass. A product pair or build for which no deterministic rule applies is not treated as compatible. In the three-class evaluation, such cases are handled as unverifiable rather than as approvals. Table 6 summarizes the main inputs and outputs of the tool.

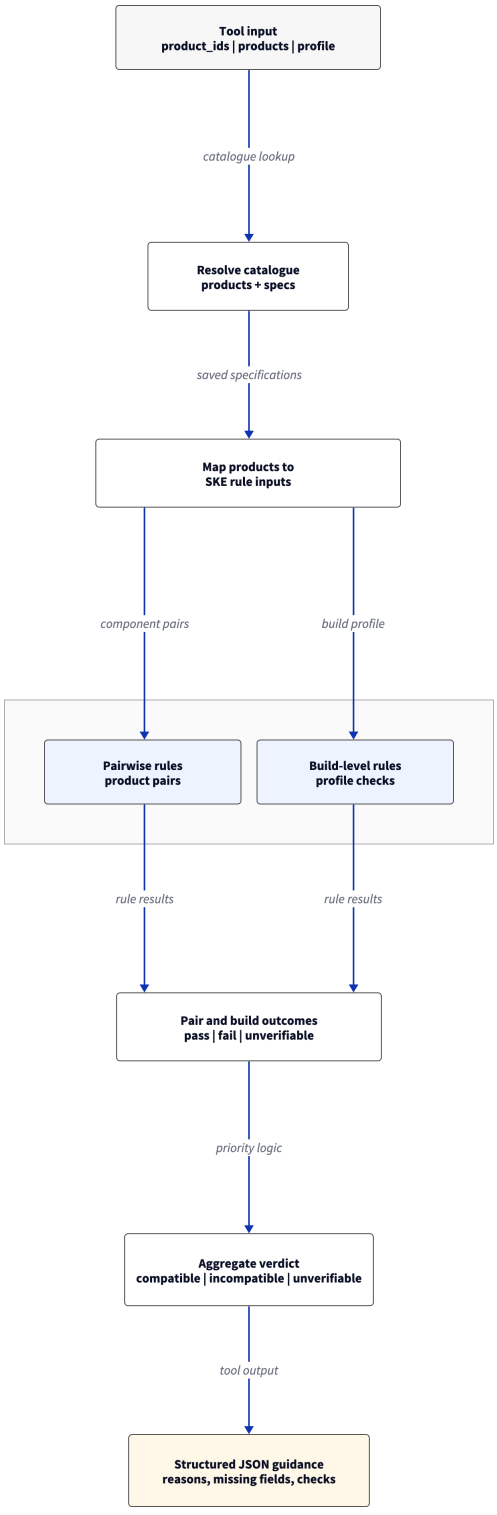

Figure 6: Compatibility-checking flow exposed through `check_compatibility`.

Part	Field	Purpose
Input	`product_ids`	Catalogue IDs or slugs resolved to saved product records and specifications.
Input	`products`	Manual product objects with explicit specifications for synthetic or user-stated builds.
Input	`profile`	Optional build profile that enables build-level checks in addition to pairwise rules.
Output	`decision`	Agent-facing verdict in the three-class task space: compatible, incompatible, or unverifiable.
Output	`blockingReasons`	Failed deterministic checks that prevent compatibility approval.
Output	`missingInformation`	Required fields or checks that prevent reliable verification.
Output	`supportingChecks`	Passed checks that support the returned verdict within the checked scope.

Table 6: Main input and output fields of the `check_compatibility` tool.

5 Benchmark Method

Reliable comparison requires more than a collection of example questions. Each benchmark item needs a visible support question, an expected verdict, and enough metadata to relate the answer back to the compatibility rules and data sources used in the evaluation.

5.1 Benchmark Objective and Composition

To make the comparison between agent modes measurable, the evaluation uses a 170-question benchmark focused on compatibility-verdict reliability rather than general conversational quality. Each item presents a natural FPV support question and requires the agent to reach one of the three verdicts defined earlier: compatible, incompatible, or unverifiable. The visible question text is written in user-facing FPV language and does not expose internal database identifiers to the evaluated model. Additional benchmark metadata is retained alongside the questions for reproducibility, scoring, and later result analysis.

The benchmark size is intended to balance rule coverage with the runtime and cost constraints of evaluating multiple models and agent modes. ARES similarly treats a target-domain validation set of about 150 annotated datapoints as sufficient for efficient RAG evaluation when combined with model-based scoring [24]. With 170 questions, an overall accuracy estimate has a worst-case binomial standard error of about 3.8 percentage points. Under the simplifying assumption that benchmark items behave like independent correct-or-incorrect trials, this corresponds to an approximate 95% interval of about ± 7.5 percentage points. The benchmark is therefore used to compare broad performance patterns between agent modes, while also retaining coverage of the 48 active rules

described in Section 4.3.

Each benchmark item is stored as a visible user question together with evaluation metadata. The metadata records the expected verdict, relevant rule labels, source and scope information, optional product or specification references, and a short expected reason. We assign the expected verdicts during benchmark construction rather than inferring them with the judge. For controlled items, the verdict follows the intended rule outcome encoded in the test case. For catalogue-grounded items, the verdict is derived from the enriched product specifications and the relevant compatibility rule or missing required field. For DIYFPV user questions, the support question is anonymized and mapped to a compatibility relation in the active rule set before assigning the expected verdict. We manually reviewed the final suite so that each item has a visible question, expected verdict, target rules, and expected reason. These fields keep internal identifiers and scoring information outside the evaluated prompt while supporting later analysis of retrieval errors, missing evidence, rule coverage, and verdict selection. Table 7 summarizes the fields stored for each item.

Field	Purpose
`id`	Stable question identifier used in evaluation logs.
`question`	User-facing question shown to the evaluated agent.
`expected_verdict`	Target verdict for scoring, one of compatible, incompatible, or unverifiable.
`target_rules`	Compatibility rules expected to be relevant for the item.
`question_type`	Broad item type, such as pair compatibility, build compatibility, or multi-rule edge case.
`source_type`	Source category used for benchmark construction.
`rule_scope`	Coverage scope of the item, such as pairwise, build-level, or real-user-post-substitution behavior.
`profile`	Optional build profile used for build-level checks.
`expected_reason`	Short rationale for the expected verdict.
`products`	Optional product references or controlled product descriptions, used when the item is tied to specific products.
`required_specs`	Specification fields expected to matter for the compatibility decision.

Table 7: Main fields stored for each benchmark item.

5.2 Composition and Coverage

The first composition dimension is the question source. The benchmark combines three sources so that the evaluation covers both rule-controlled cases and realistic user phrasing. DIYFPV user questions are derived from visible community and troubleshooting posts, with identifying details removed from the benchmark surface. Catalogue-grounded product questions use real DIYFPV product records and their enriched specifications. These items test whether the agent can retrieve the right products and use the available product evidence correctly. Lastly, controlled user-style product questions are constructed from rule test cases, but their visible wording states the relevant component information in conversational form rather than as internal field names. The resulting source split is summarized in Table 8, and example questions from the benchmark are shown in Listing 3.

Source type	Count	Role in the benchmark
DIYFPV user questions	26	User-style compatibility questions derived from visible community and troubleshooting posts.
Catalogue-grounded DIYFPV product questions	84	Real catalogue products with hidden product metadata and enriched specifications.
Controlled user-style product questions	60	Constructed product cases that cover rule outcomes not cleanly available from catalogue examples.

Table 8: Question sources in the 170-question benchmark.

```
DIYFPV user question:
Will a TBS Crossfire module in a RadioMaster Boxer let me control a drone that has
    a TBS Nano RX?

Catalogue-grounded DIYFPV product question:
I am considering TBS Graphene 4S 1500mAh 75C for DJI Air 3S. Can I use that
    battery with this drone?

Controlled user-style product question:
Would a 5 inch build with an F7 stack and a CRSF receiver work if I can only
    confirm that the receiver uses CRSF, but I cannot find the stack UART count?
```

Listing 3: Example benchmark questions from the three source categories.

After source type, the next composition dimension is compatibility scope. Pairwise questions ask about the compatibility of two selected components. Build-level questions ask whether a larger parts list satisfies constraints that depend on the build as a whole. Multi-rule edge cases use support questions in which the answer may depend on several component relationships at once. The resulting scope split is shown in Table 9.

Question scope	Count	Role in the benchmark
Pairwise compatibility	105	Questions about the compatibility of two selected components.
Build-level compatibility	39	Questions about constraints that depend on a larger parts list.
Multi-rule edge cases	26	Questions where several component relationships may affect the answer.

Table 9: Question scopes in the 170-question benchmark.

The final composition dimension is the expected verdict. Its distribution is intentionally not uniform because the benchmark emphasizes safety-relevant failure modes. Incompatible and unverifiable

cases are more frequent to test whether the assistant avoids unsupported approvals of component combinations, which matter in FPV support because such approvals can lead users toward parts that do not fit, bind, power, or communicate correctly. Compatible cases are still included to check that the system does not become overly conservative, but their smaller count means that compatible-class accuracy has higher uncertainty than the other verdict classes. Table 10 shows the resulting verdict distribution.

Expected verdict	**Count**	**Benchmark role**
Compatible	29	Questions where the stored evidence and applicable rules should justify approval.
Incompatible	73	Questions where a known mismatch or rule violation should block approval.
Unverifiable	68	Questions where missing or ambiguous information should block confident approval.

Table 10: Expected verdict distribution in the 170-question benchmark.

Together, these dimensions allow results to be interpreted beyond overall accuracy. They separate real catalogue retrieval, controlled rule coverage, and user-style support questions, while also distinguishing direct component checks from build-level and multi-rule reasoning.

6 Experimental Setup

The evaluation pipeline runs each benchmark question for each selected language model and agent mode, then stores the complete result for later analysis. Within one model configuration, the same question is submitted to standard RAG, rule-prompted RAG, and CVAL under the same benchmark conditions. Each run produces a final answer as well as execution metadata, including tool calls, retrieved or checked evidence, latency, token usage, estimated cost, and any runtime error. This setup keeps the benchmark question fixed while varying the agent architecture, making the comparison focus on whether retrieval alone, prompt-level rule guidance, or executable rule checking leads to more reliable compatibility verdicts.

We give each agent run a 180-second timeout and a LangGraph recursion limit of 64 graph steps. The timeout bounds wall-clock runtime and cost while still allowing several retrieval, product-lookup, and compatibility-checking calls for tool-heavy questions. The recursion limit bounds the agent loop itself, so a run cannot continue indefinitely if the model keeps requesting tools or fails to reach a final answer. LangGraph raises a recursion error when a graph exceeds this limit without reaching a stop condition [14]. Runtime failures such as timeouts or recursion-limit errors are recorded as execution failures rather than judged answers, and are reported separately from judged verdict accuracy.

As discussed in Section 3.2, generated answers are scored against predefined expected verdicts rather than by open-ended preference. Because the evaluated agents produce natural-language support answers rather than a required JSON verdict field, the expressed verdict is first extracted from the answer text. A separate LLM judge performs this extraction and records whether the extracted

verdict matches the benchmark label. The judge also records diagnostic scores for consistency, rule reasoning, evidence use, unsupported claims, handling of missing information, overstated certainty, and explanation quality. Following the rationale in ARES, where a small human-annotated set is used to mitigate prediction errors in model-based evaluation [24], the automated scores are complemented by manual inspection of selected outputs, especially mismatches, unverifiable cases, and safety-relevant false approvals.

6.1 Model Selection and Run Configuration

We choose the final model suite to cover different price and capability levels while keeping the full benchmark run feasible. Because each model is evaluated on 170 questions across three agent modes, adding one model adds 510 agent runs before judging. The suite therefore favors a small number of current models that span a lower-cost small-model baseline, cost-efficient reasoning models, balanced production candidates, and a stronger ceiling model. Where a model provider exposes reasoning or effort settings, the evaluation uses the reasoning-oriented configuration because the agents perform multi-step tool use and structured compatibility decisions.

We use external Intelligence vs. Price data from Artificial Analysis to guide the model-suite selection [2]. Figure 7 shows the selected candidate models on the Artificial Analysis view. The vertical axis is the Artificial Analysis Intelligence Index v4.0, a composite score over ten evaluations covering agentic tasks, coding, instruction following, knowledge, long-context reasoning, and scientific reasoning [1]. As a reference point, Claude Opus 4.8 has the highest AAII value in the comparison, with an index of 61.4. The horizontal axis in Figure 7 is the blended model price in USD per million tokens. Artificial Analysis computes this price using a 7:2:1 cache, input, and output token weighting, which approximates workloads where repeated prompt prefixes are served partly through cheaper prompt-cache hits [2]. Table 11 lists the corresponding release dates and numeric values.

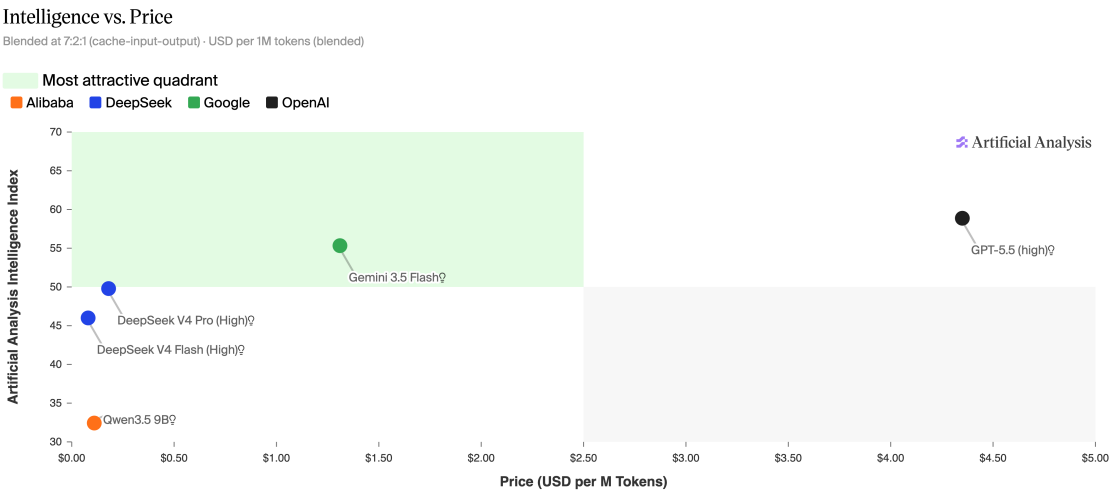

Figure 7: Intelligence vs. Price view for the selected model-suite candidates, taken from Artificial Analysis [2] and accessed on 2026-06-02.

Model	Role in suite	Release	AAII	AA blended price
Qwen3.5 9B	Small low-cost model	2026-03-02	32.4	0.105
DeepSeek V4 Flash	Cost-efficient reasoning model	2026-04-24	46.0	0.058
DeepSeek V4 Pro	Cost-efficient stronger reasoning model	2026-04-24	49.8	0.177
Gemini 3.5 Flash	Balanced production candidate	2026-05-19	55.3	1.305
GPT-5.5	Frontier ceiling model	2026-04-23	58.9	4.350

Table 11: Candidate model suite used to select the final evaluation models. AAII denotes the Artificial Analysis Intelligence Index. The blended price is the Artificial Analysis USD price per one million tokens under its 7:2:1 cache-input-output weighting; actual benchmark costs are calculated separately from logged evaluation runs.

6.2 Automated Judging and Manual Inspection

After each agent run has produced a final answer, we use an automated LLM judge to convert the natural-language response into structured scoring fields. This makes it possible to score hundreds of answers consistently before manual inspection; for the five-model suite, the full run contains 170 questions across three agent modes and five models, or 2,550 generated answers. A purely deterministic comparison would be possible if every agent answer were forced to return a machine-readable verdict field. The evaluation instead keeps the answers in natural support style, so the judge is used to extract the expressed verdict from text such as approval, rejection, or insufficient-evidence wording. The judge is not asked to infer the correct compatibility decision independently, but receives the benchmark fields described in Section 5.1, together with the agent mode, compact tool outputs, and the final answer. Once the judge has extracted the expressed verdict, the match against the predefined benchmark target is a deterministic comparison.

The judge configuration uses DeepSeek V4 Pro with high reasoning effort. This choice is cost-aware: in Table 11, DeepSeek V4 Pro sits between DeepSeek V4 Flash and Gemini 3.5 Flash in Artificial Analysis Intelligence Index while keeping a much lower blended price than the higher-price models. It therefore represents a balanced approach between judge capability and evaluation cost.

Beyond the verdict match, the judge records diagnostic fields for later error analysis. These fields indicate whether the answer is internally consistent, uses relevant evidence, reasons correctly about the compatibility issue, handles missing information appropriately, avoids unsupported claims, avoids overstated certainty, and provides a useful explanation. Table 12 summarizes the main judge outputs.

Judge field	Meaning
judge_extracted_verdict	Verdict expressed by the agent answer.
judge_verdict_matches_expected	Whether the extracted verdict matches the benchmark verdict.
answer_is_internally_consistent	Whether the answer contradicts itself.
rule_reasoning_correct	Whether the answer reasons correctly about the expected compatibility issue.
used_relevant_evidence	Whether the answer uses relevant question, product, or tool evidence.
unsupported_claims	Whether the answer adds claims not supported by available evidence.
missing_info_handled_correctly	Whether missing information is treated appropriately.
overstated_certainty	Whether the answer is more certain than the evidence allows.
explanation_quality	Integer score from 1 to 5 for explanation usefulness.
short_reason	One-sentence judge rationale.

Table 12: Main fields returned by the automated LLM judge.

We use automated judging for scale, but we do not treat it as a complete replacement for human review. ARES addresses judge prediction errors through prediction-powered inference, using a small human-annotated validation set to correct model-based evaluation estimates [24]. This evaluation does not implement that statistical correction. Instead, selected runs are inspected manually after automated scoring by reading the benchmark item, expected verdict, agent answer, extracted judge verdict, and compact tool trace together. The inspection checks whether the judge extracted the expressed verdict correctly and whether an observed error is more plausibly connected to product retrieval, missing specifications, rule coverage, SKE use, or final answer wording. The inspected cases include judge mismatches, unverifiable items, safety-relevant false approvals, runtime failures, and examples where the diagnostic fields suggest unsupported or overconfident reasoning.

7 Results

Result reporting separates two concerns: whether the experimental runs completed and could be judged, and what compatibility behavior appeared once a usable answer was produced. Run completeness is reported first because the accuracy denominators depend on completed runs with usable judge verdicts.

7.1 Evaluation Completeness

The final evaluation attempted the full 170-question benchmark for each of the five language models and each of the three agent modes, giving 2,550 model–mode–question runs. Each run used the fixed limits described in Section 6: a 180-second timeout to bound wall-clock runtime

and a LangGraph recursion limit of 64 graph steps to stop agent loops that did not converge to a final answer. Table 13 summarizes completed runs, usable judge verdicts, and execution failures by model. Across the final suite, no runs failed because of OpenRouter rate limits, insufficient credits, or database/network errors. All execution failures were timeouts, recursion-limit failures, empty-answer runs, or provider-side content-filter rejections where the upstream model provider rejected the generated output.

Model	Attempted	Completed	Judged	Correct	Failures
Qwen 3.5 9B	510	406	403	300	104
DeepSeek V4 Flash	510	489	488	388	21
DeepSeek V4 Pro	510	458	455	365	52
Gemini 3.5 Flash	510	491	488	388	19
GPT-5.5 high	510	498	490	434	12

Table 13: Evaluation completeness across the five final model runs. Attempted runs cover 170 benchmark questions across three agent modes for each model. Judged runs are completed runs for which the automated judge produced a usable extracted verdict.

7.2 Overall Verdict Accuracy

The main comparison is verdict accuracy across the three agent modes. Accuracy is computed over completed runs for which the automated judge extracted a usable verdict. This denominator follows from Section 7.1, so runtime failures are not counted as incorrect judged answers but remain visible in the completeness table. Across all five evaluated models, CVAL produced the highest verdict accuracy among the three modes. Figure 8 shows the same pattern visually, while Table 14 gives the exact counts.

The improvement from CVAL is largest for the lower- and mid-cost models. DeepSeek V4 Flash improves from 73.3% with standard RAG to 85.2% with CVAL, while DeepSeek V4 Pro improves from 73.2% to 85.7%. Gemini 3.5 Flash also improves from 75.6% to 83.3%. GPT-5.5 high starts from a stronger standard-RAG baseline, so the absolute gain is smaller, but GPT-5.5 high with CVAL still gives the highest achieved accuracy across all model–mode combinations at 89.9%.

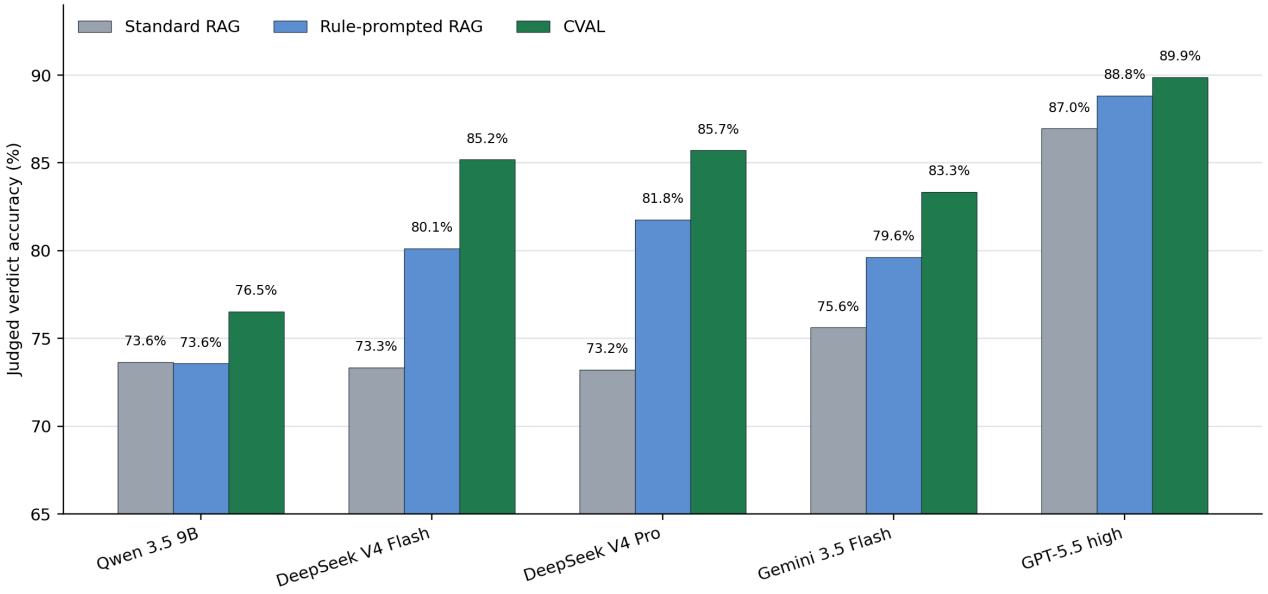

Figure 8: Overall judged verdict accuracy by model and agent mode.

Model	Agent mode	Judged	Correct	Acc.	False comp.
Qwen 3.5 9B	Standard RAG	148	109	73.6%	17
Qwen 3.5 9B	Rule-prompted RAG	140	103	73.6%	20
Qwen 3.5 9B	CVAL	115	88	76.5%	6
DeepSeek V4 Flash	Standard RAG	165	121	73.3%	26
DeepSeek V4 Flash	Rule-prompted RAG	161	129	80.1%	16
DeepSeek V4 Flash	CVAL	162	138	85.2%	9
DeepSeek V4 Pro	Standard RAG	153	112	73.2%	21
DeepSeek V4 Pro	Rule-prompted RAG	148	121	81.8%	12
DeepSeek V4 Pro	CVAL	154	132	85.7%	11
Gemini 3.5 Flash	Standard RAG	164	124	75.6%	21
Gemini 3.5 Flash	Rule-prompted RAG	162	129	79.6%	11
Gemini 3.5 Flash	CVAL	162	135	83.3%	7
GPT-5.5 high	Standard RAG	161	140	87.0%	8
GPT-5.5 high	Rule-prompted RAG	161	143	88.8%	5
GPT-5.5 high	CVAL	168	151	89.9%	3

Table 14: Overall judged verdict accuracy by model and agent mode. False-compatible counts refer to judged cases where the benchmark verdict was incompatible or unverifiable, but the agent answer was judged as compatible.

7.3 Accuracy by Verdict Class

Verdict-class accuracy shows whether the overall gains come from approving compatible cases, rejecting incompatible cases, or withholding approval when the evidence is insufficient. This distinction matters because, as discussed in Section 5.2, the benchmark is not uniformly distributed across labels, and because an FPV support agent should not treat missing compatibility evidence as approval. Table 15 reports the CVAL results by expected verdict.

Model	Compatible	Incompatible	Unverifiable
Qwen 3.5 9B	12/17 (70.6%)	39/50 (78.0%)	37/48 (77.1%)
DeepSeek V4 Flash	23/28 (82.1%)	64/70 (91.4%)	51/64 (79.7%)
DeepSeek V4 Pro	22/27 (81.5%)	62/68 (91.2%)	48/59 (81.4%)
Gemini 3.5 Flash	20/27 (74.1%)	63/69 (91.3%)	52/66 (78.8%)
GPT-5.5 high	22/29 (75.9%)	66/72 (91.7%)	63/67 (94.0%)

Table 15: CVAL judged verdict accuracy by expected verdict class. Each cell reports correct judged answers over judged answers for that class.

The strongest CVAL class-level results occur on incompatible cases for the four larger models, each of which reaches approximately 91% accuracy on that class. Unverifiable cases also improve compared with the standard-RAG baseline for every model, with GPT-5.5 high reaching 94.0% under CVAL. For compatible cases, CVAL is lower than standard RAG for all five models: Qwen 3.5 9B changes from 84.6% to 70.6%, DeepSeek V4 Flash from 96.3% to 82.1%, DeepSeek V4 Pro from 92.6% to 81.5%, Gemini 3.5 Flash from 85.7% to 74.1%, and GPT-5.5 high from 93.1% to 75.9%. This shows that the checker-based mode is more cautious about approving combinations when the required evidence is incomplete or a rule does not support approval.

7.4 False-Compatible Approvals

The most safety-relevant error type is a false-compatible approval. In this evaluation, a false-compatible approval occurs when the benchmark verdict is incompatible or unverifiable, but the agent answer is judged as compatible. The rate is therefore computed over judged non-compatible cases rather than over all judged answers. Figure 9 shows that CVAL reduces this rate for every evaluated model, and Table 16 gives the corresponding counts and rates.

The largest absolute reductions occur for DeepSeek V4 Flash and Gemini 3.5 Flash. DeepSeek V4 Flash falls from 26 false-compatible approvals under standard RAG to 9 under CVAL, while Gemini 3.5 Flash falls from 21 to 7. GPT-5.5 high has the lowest false-compatible count in all three modes, and CVAL reduces it further to 3 cases.

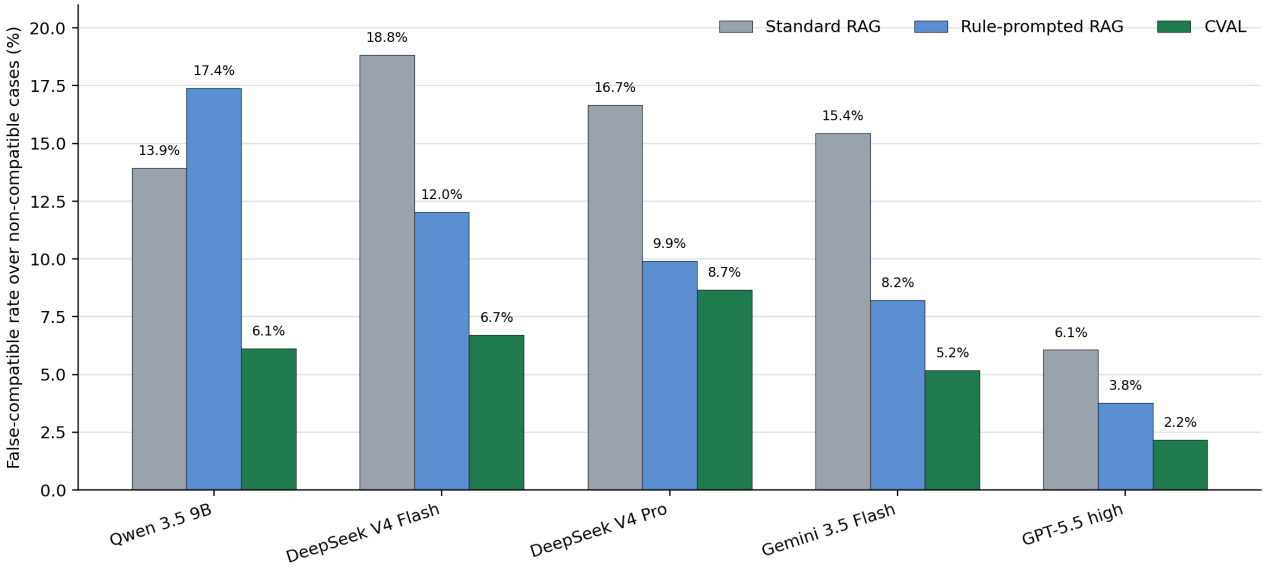

Figure 9: False-compatible approval rate by model and agent mode. The denominator is judged cases whose expected verdict is incompatible or unverifiable.

Model	**Standard RAG**	**Rule-prompted RAG**	**CVAL**
Qwen 3.5 9B	17 (13.9%)	20 (17.4%)	6 (6.1%)
DeepSeek V4 Flash	26 (18.8%)	16 (12.0%)	9 (6.7%)
DeepSeek V4 Pro	21 (16.7%)	12 (9.9%)	11 (8.7%)
Gemini 3.5 Flash	21 (15.4%)	11 (8.2%)	7 (5.2%)
GPT-5.5 high	8 (6.1%)	5 (3.8%)	3 (2.2%)

Table 16: False-compatible approval counts and rates. Each cell reports false-compatible approvals, with the rate in parentheses over judged non-compatible cases.

7.5 SKE Use and Agent Behavior

The CVAL results depend on whether the agent actually invokes the SKE through `check_compatibility`, introduced in Section 4.5.5. Table 17 reports SKE use over completed CVAL runs. For the four larger models, the checker is called in at least 96.8% of completed CVAL runs. Qwen 3.5 9B is the exception, with checker use in 74.1% of completed CVAL runs, which is consistent with its higher number of empty-answer and runtime failures in Table 13. The checker-use rates provide the execution context for the CVAL accuracy and false-compatible results reported above.

Model	Completed CVAL	SKE used	Use rate
Qwen 3.5 9B	116	86	74.1%
DeepSeek V4 Flash	162	158	97.5%
DeepSeek V4 Pro	156	151	96.8%
Gemini 3.5 Flash	162	161	99.4%
GPT-5.5 high	168	167	99.4%

Table 17: SKE use in completed CVAL runs. SKE use is measured by whether the run invoked `check_compatibility`.

7.6 Runtime, Token Usage, and Cost

Runtime and cost are reported descriptively because the evaluated agent modes differ not only in prompt content, but also in how often they call product, knowledge, and compatibility tools. Median latency is used as the main runtime measure because tool-using runs have long-tailed runtimes, with many runs finishing after a small number of steps while a smaller number approach the timeout or require repeated tool calls. Runtime failures are reported separately in Section 7.1. Table 18 summarizes median latency, total token usage, estimated cost, and average cost per attempted question by model and mode.

Model	Agent mode	Median lat.	Tokens	Total cost	Cost per q.
Qwen 3.5 9B	Standard RAG	53.4s	2.07M	$0.55	$0.0032
Qwen 3.5 9B	Rule-prompted RAG	64.5s	5.19M	$0.69	$0.0041
Qwen 3.5 9B	CVAL	75.3s	4.82M	$0.63	$0.0037
DeepSeek V4 Flash	Standard RAG	33.7s	4.89M	$1.12	$0.0066
DeepSeek V4 Flash	Rule-prompted RAG	38.0s	9.67M	$1.58	$0.0093
DeepSeek V4 Flash	CVAL	32.6s	7.29M	$1.28	$0.0075
DeepSeek V4 Pro	Standard RAG	55.7s	3.02M	$1.91	$0.0112
DeepSeek V4 Pro	Rule-prompted RAG	47.8s	5.14M	$2.71	$0.0159
DeepSeek V4 Pro	CVAL	54.3s	5.28M	$2.78	$0.0164
Gemini 3.5 Flash	Standard RAG	20.6s	3.74M	$8.95	$0.0526
Gemini 3.5 Flash	Rule-prompted RAG	20.1s	7.54M	$14.54	$0.0856
Gemini 3.5 Flash	CVAL	20.3s	5.86M	$11.76	$0.0692
GPT-5.5 high	Standard RAG	34.1s	2.18M	$16.31	$0.0960
GPT-5.5 high	Rule-prompted RAG	32.5s	4.37M	$27.19	$0.1599
GPT-5.5 high	CVAL	29.4s	3.96M	$24.64	$0.1450

Table 18: Runtime, token usage, and estimated cost by model and agent mode. Total cost includes model-call cost and automated-judge cost for the corresponding runs. Cost per question is total cost divided by the 170 attempted questions in that model–mode run.

Gemini 3.5 Flash has the lowest median latency across all modes, with all three medians close to 20 seconds. GPT-5.5 high gives the highest accuracy in Section 7.2, but it is also the most expensive

model in the suite, with a total evaluation cost of $68.15 across all modes. The DeepSeek models are substantially cheaper, with total evaluation costs of $3.98 for DeepSeek V4 Flash and $7.40 for DeepSeek V4 Pro. CVAL is not consistently slower than the baselines; for DeepSeek V4 Flash, Gemini 3.5 Flash, and GPT-5.5 high, the CVAL median latency is comparable to or lower than the corresponding standard-RAG median.

7.7 Error Pattern Summary

Execution failures are reported separately from judged-answer accuracy. Table 19 summarizes the failure categories observed in the final suite. The most common failures are timeouts, where a run did not finish within the 180-second limit, and empty-answer runs. Empty-answer failures occurred when the model-driven agent loop completed after tool-call steps, but no final natural-language assistant response was present for judging. They are therefore answer-completion failures in the tool-using loop rather than judged compatibility mistakes. Recursion-limit failures occur when the LangGraph run exceeds the configured graph-step limit before reaching a stop condition.

Model	Failed	Timeout	Recur.	Empty	Filter
Qwen 3.5 9B	104	45	5	54	0
DeepSeek V4 Flash	21	16	4	0	1
DeepSeek V4 Pro	52	34	17	0	1
Gemini 3.5 Flash	19	17	1	1	0
GPT-5.5 high	12	12	0	0	0

Table 19: Execution failure categories in the final evaluation suite. Failure counts are over 510 attempted runs per model.

Qwen 3.5 9B has the highest failure count, with 104 failed runs out of 510 attempts. More than half of these are empty-answer runs, which also helps explain the lower SKE-use rate reported in Section 7.5. The other four models have substantially fewer failures, with GPT-5.5 high producing the fewest failed runs at 12 out of 510.

8 Discussion

The results indicate that adding deterministic rule execution improves compatibility-verdict reliability, but the effect is not uniform across models, verdict classes, or resource constraints. The main pattern is that CVAL raises overall judged accuracy for every evaluated model, while also reducing the most safety-relevant error type, namely false-compatible approval. At the same time, the class-level results show a tradeoff because CVAL is more cautious on compatible cases than standard RAG. The discussion therefore separates verdict reliability, safety behavior, model effects, and resource cost before turning to broader implications for bounded compatibility support.

8.1 Answering the Research Questions

The main research question in Section 2 asked whether CVAL improves compatibility-verdict reliability compared with standard RAG and rule-prompted RAG. The answer is yes for the evaluated benchmark, measured over completed runs with usable judge verdicts. As shown in Table 14, CVAL achieves the highest accuracy for all five evaluated models. The improvement is especially visible for the DeepSeek models and Gemini 3.5 Flash, where CVAL raises accuracy by several percentage points over both baselines. GPT-5.5 high starts from a stronger baseline, so its absolute gain is smaller, but GPT-5.5 high with CVAL still achieves the highest accuracy across all model–mode combinations.

For the verdict-class sub-question, the class-level results in Table 15 show that CVAL is strongest on incompatible and unverifiable cases. For the four larger models, incompatible-case accuracy is around 91%, and GPT-5.5 high reaches 94.0% on unverifiable cases. Compatible cases are weaker under CVAL, most clearly for GPT-5.5 high, where 22 of 29 compatible cases are judged correctly. This suggests that the verification step improves safety-oriented withholding and rejection, but can also make the system more conservative when a correct answer requires approval.

For the false-approval sub-question, the results support a clear positive answer. CVAL reduces the false-compatible rate for every model, as shown in Figure 9 and Table 16. This matters because false-compatible approvals are the cases where an assistant approves a combination that is either known to be incompatible or not supported by enough evidence. The reduction is largest in absolute count for DeepSeek V4 Flash and Gemini 3.5 Flash, while GPT-5.5 high has the lowest remaining count under CVAL with three such cases.

For the runtime and resource sub-question, the results show that model choice remains important. GPT-5.5 high is the strongest absolute model, but it is also the most expensive. DeepSeek V4 Flash and DeepSeek V4 Pro reach lower absolute accuracy than GPT-5.5 high, but their total evaluation costs are much lower, making them important cost-performance candidates. In CVAL mode, DeepSeek V4 Flash costs approximately $0.0075 per attempted question and about $0.009 per correct judged answer, while GPT-5.5 high costs approximately $0.1450 per attempted question and about $0.163 per correct judged answer. For a platform context such as DIYFPV, with more than 15,000 users, this difference matters because per-question cost scales directly with repeated support use. Gemini 3.5 Flash has the lowest median latency, while Qwen 3.5 9B has the weakest execution reliability and the lowest checker-use rate. The SKE-use results in Table 17 show that, for the four larger models, completed CVAL runs nearly always reached the checker, so the CVAL comparison mostly reflects the intended architecture rather than prompt-only behavior.

8.2 Reliability and Conservatism

The main reliability gain from CVAL is that it changes the cost of uncertainty. In standard RAG, the model may retrieve relevant-looking product evidence and still produce a confident compatibility approval when the decisive field is missing, ambiguous, or contradicted elsewhere. In CVAL, the SKE introduces a bounded verdict signal that can block approval when the relevant structured inputs do not support it. This is why the false-compatible rate falls across all evaluated models. The same mechanism also helps explain the compatible-class pattern reported in Section 7.3, since correct approval requires product resolution, specifications, and checks to align.

Manual inspection of selected correct and incorrect answers, as described in Section 6.2, shows that the remaining errors are not explained by a single failure point. CVAL does not turn FPV compatibility support into a solved classification problem, and even GPT-5.5 high with CVAL reaches 89.9% overall judged accuracy rather than 100%. The deterministic checker is only one part of the pipeline. The agent must still identify the intended products, retrieve or resolve the correct catalogue records, pass the relevant inputs to the checker, and then express the returned signal correctly in the final answer. Errors can therefore still arise before the SKE is called, for example when product names are ambiguous or catalogue variants are difficult to distinguish. They can also arise after the SKE returns a result, if the language model phrases the final answer too strongly or fails to preserve the checker verdict.

Specification availability is another source of residual error. As described in Section 4.2, the extraction pipeline achieved 73.8% overall field coverage in the clean full-scale run, so not every potentially relevant product field is available as a structured rule input. The benchmark is designed around the active rule set, but the rules still depend on enriched product fields being present and reliable for the products used in each question. When a product record lacks a relevant field, CVAL may correctly avoid approval but still miss a compatible benchmark case. This is visible in the compatible-class results, where GPT-5.5 high with CVAL correctly answers 22 of 29 compatible cases. The result is therefore not only a measure of rule execution, but also a measure of how well retrieval, product resolution, specification enrichment, and final response generation work together.

8.3 Neuro-Symbolic Agents for Constraint-Bounded Support

The results support the broader design idea that retrieval and generation are useful for user-facing support, but are not always sufficient for bounded verdict tasks. In FPV compatibility, the language model is valuable for interpreting natural-language questions, finding likely products, calling tools, and explaining the outcome. However, the final verdict often depends on structured constraints that should not be inferred from retrieved prose alone. CVAL therefore uses the SKE as a symbolic constraint layer rather than as another source of text. This division of labor matches the neuro-symbolic framing introduced in Section 3.3, where the neural component handles language and tool orchestration while the symbolic component executes explicit rules over structured inputs.

This pattern is most appropriate when a support task has bounded decisions, identifiable entities, structured attributes, and rules that can be made explicit. FPV compatibility has these properties for many component relationships, which is why the SKE can improve verdict reliability. The same pattern may be useful for other domains such as parts selection, device configuration, policy eligibility, or regulated product guidance, where natural-language support must respect constraints that can be checked deterministically. The results also show the boundary of the approach. A neuro-symbolic agent is only as reliable as the full path from user question to structured rule input and final answer. Better rules alone are not enough if product retrieval, specification extraction, or final response generation fails.

9 Limitations

While the benchmark is large enough to compare the evaluated model–mode combinations, the results should still be read in light of the study design. The 170-question suite combines catalogue-based, controlled, and community-derived questions, but it is designed specifically for FPV compatibility rather than for general DIYFPV support. It evaluates whether generated answers preserve the expected compatibility verdict, not whether people would find the assistant helpful, trustworthy, or easy to use. The compatible class is also smaller than the incompatible and unverifiable classes because the benchmark emphasizes safety-relevant non-approval cases, but this does make compatible-class estimates less stable.

The reported accuracy and false-compatible rates are therefore used as descriptive results for this benchmark. They show the observed differences between agent modes and models, but the evaluation does not apply formal significance testing to determine whether small gaps between individual model–mode pairs are distinguishable from benchmark-level variation. This matters most for close comparisons among the strongest configurations, especially GPT-5.5 high, where only a few changed answers can move the percentage by several points.

Second, the evaluation relies on an automated LLM judge. The judge extracts the verdict expressed by the agent and compares it with the expected benchmark label, which makes large-scale scoring feasible. However, the judge can still misread ambiguous natural-language answers or disagree with a human evaluator about whether an answer expressed enough certainty. Manual inspection of selected correct and incorrect cases reduces this risk, but it does not provide a full human annotation pass over all 2,550 generated answers.

A related limitation is the degree of domain-expert validation. The benchmark labels, rule interpretations, and manual inspections were developed with practical FPV knowledge and access to DIYFPV product and support context, but they were not independently validated by multiple external FPV experts. Some expected verdicts or rule interpretations may therefore reflect the current project understanding of FPV compatibility rather than a consensus expert standard.

Third, CVAL depends on the quality of the enriched product specifications. The extraction pipeline validates schema conformance and rejects unsupported fields, but the complete specification database is not manually verified field by field. Missing specifications may cause CVAL to withhold approval where a complete datasheet would allow a compatible verdict, while incorrect specifications could produce misleading rule outcomes. The reported results therefore evaluate CVAL as an end-to-end system, not the rule engine in isolation.

Fourth, the deterministic rule set covers the compatibility relations targeted in this evaluation, while leaving room for further expansion through expert review and real support use. The current rules focus on relations that could be represented with available product specifications and checked deterministically. Additional FPV expertise may reveal further compatibility cases, exceptions, or rule refinements that are not captured by the current implementation. The results therefore show the value of executable rules for the evaluated compatibility relations, not a complete rule model for all FPV build decisions.

Finally, the evaluation is intentionally grounded in DIYFPV.com and FPV drone compatibility. This focus makes the case study concrete, but it also means that claims about transfer to other domains

should remain cautious. The broader neuro-symbolic pattern may transfer to other constraint-bounded domains, but that transfer would require domain-specific entity resolution, structured data extraction, executable rule design, and evaluation benchmarks. The results support the value of CVAL in this FPV case-study environment rather than proving general reliability for all hardware or policy support systems.

10 Future Work

The most immediate next step is to review and extend the FPV compatibility knowledge base with additional domain expertise. The current rule set is grounded in DIYFPV product data, internal guide material, and practical FPV references, but expert review by experienced FPV builders could further refine the rules, add missing cases, and clarify where deterministic checking should stop. This is especially relevant for cases involving firmware configuration, physical layout, vendor-specific behavior, and build-practice assumptions that are difficult to infer from catalogue text alone. Future work should also include independent validation of benchmark labels and rule interpretations by multiple FPV experts, so that the expected verdicts reflect a broader domain consensus rather than only the current project interpretation.

The second direction is to improve the structured product-specification layer. CVAL can only execute rules over fields that have been extracted, normalized, and linked to the correct catalogue products. Better extraction prompts, manufacturer-document ingestion, targeted web evidence, and human-in-the-loop correction could raise specification coverage and reduce the number of compatible cases that become unverifiable because one adjacent field is missing. A practical version of the system could also record which missing fields most often prevent verification, helping maintainers prioritize catalogue improvements. Over time, this would make the SKE more useful not only as an answering tool, but also as a diagnostic layer for catalogue quality.

Future evaluation should also move beyond offline verdict accuracy. The benchmark used here measures whether generated answers preserve expected compatibility verdicts, but it does not measure whether real users find the assistant helpful, understandable, or trustworthy during an actual build-planning workflow. A follow-up study could evaluate CVAL with DIYFPV users, compare user confidence and decision quality, and test whether cautious unverifiable answers are perceived as useful rather than frustrating. Future work could also add statistical tests and confidence intervals to check whether small differences between the strongest model–mode combinations are meaningful or mainly caused by a few changed answers.

Finally, CVAL should be understood as one component of a larger FPV assistant rather than as a complete support system. A production assistant would also need broader troubleshooting support, build planning, product recommendation, firmware guidance, and user-specific context such as budget, skill level, existing parts, and intended flying style. The same pattern could also be tested outside FPV in other constraint-bounded domains, such as PC component compatibility, smart-home device configuration, insurance eligibility screening, or regulated medical-device guidance. In each case, the open research question could be when it is worth converting natural-language support problems into structured inputs and explicit rule logic, which introduce additional data-preparation and maintenance work but can make bounded decisions more reliable than language-model reasoning alone.

11 Conclusion

The work presented here examined whether deterministic compatibility checking can improve the reliability of retrieval-augmented FPV compatibility-support agents. The implemented CVAL system combines product and knowledge retrieval with a Symbolic Knowledge Engine that executes compatibility rules over enriched product specifications. The evaluation compared CVAL against standard RAG and rule-prompted RAG on a 170-question FPV compatibility benchmark across five language models.

The results support the main claim that executable rule checking improves compatibility-verdict reliability. CVAL achieved the highest judged verdict accuracy for every evaluated model and reduced false-compatible approvals across the full model suite. The strongest absolute result was GPT-5.5 high with CVAL, which reached 89.9% judged accuracy and reduced false-compatible approvals to three cases. The results also show that the improvement is not only a frontier-model effect. DeepSeek V4 Flash and DeepSeek V4 Pro achieved strong CVAL performance at much lower cost, making them important candidates for practical deployment settings.

The central tradeoff is that CVAL improves safety-oriented rejection and withholding, but can also become more conservative on compatible cases. This behavior follows from the design of the system. Missing or ambiguous evidence is not treated as proof of compatibility, and the final answer must respect the deterministic checker when the checker can evaluate the selected products. The remaining errors show that compatibility support is still an end-to-end problem involving product retrieval, specification quality, rule execution, and final response generation.

Overall, FPV compatibility provides a useful case study for neuro-symbolic LLM agents in constraint-bounded domains. Retrieval remains valuable for finding products and explaining evidence, but bounded compatibility verdicts benefit from explicit rule execution. The broader implication is that LLM assistants can be made more reliable when language-model flexibility is paired with symbolic checks for the parts of a task where explicit constraints are available.

Declaration of AI Tool Use

Codex was used during the preparation of this thesis as programming and analysis support. It was used to discuss thesis structure, review wording for clarity, reason through presentation of results, inspect code, modify scripts, debug evaluation tooling, and assist with LaTeX formatting. Codex was also used during product and specification exploration, for example when inspecting product records and compatibility-relevant fields while preparing benchmark questions.

AI tools are also part of the implemented and evaluated system itself, as described in the Solution Design and Experimental Setup sections. This includes language-model agents, product and knowledge retrieval, specification extraction, and automated judging. All final research decisions, benchmark design choices, code changes, interpretations, source selection, and thesis text were reviewed and approved by the author. The author remains responsible for the correctness of the work, including the reported results, citations, and conclusions.

References

[1] Artificial Analysis. Artificial Analysis Intelligence Benchmarking Methodology. https://artificialanalysis.ai/methodology/intelligence-benchmarking, 2026. Accessed: 2026-06-02.

[2] Artificial Analysis. Comparison of AI models across intelligence, performance, and price. https://artificialanalysis.ai/models/, 2026. Accessed: 2026-06-02.

[3] Joshua Bardwell. The FPV shopping list — 5 inch freestyle FPV drones and parts. https://www.fpvknowitall.com/fpv-shopping-list-five-inch-freestyle/, 2026. Accessed: 2026-05-26.

[4] Tarek R. Besold, Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Pedro Domingos, Pascal Hitzler, Kai-Uwe Kühnberger, Luis C. Lamb, Daniel Lowd, Priscila Machado Vieira Lima, Leo de Penning, Gadi Pinkas, Hoifung Poon, and Gerson Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. *arXiv preprint arXiv:1711.03902*, 2017.

[5] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Buettcher. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 758–759, 2009.

[6] DataForSEO. DataForSEO apis. https://dataforseo.com/apis, 2026. Accessed: 2026-05-26.

[7] DIYFPV. DIYFPV.COM — FPV drone community and product catalog. https://www.diyfpv.com, 2026. Over 10,000 users and over 40,000 products from 30+ stores. Accessed: 2026-05-23.

[8] Firecrawl. Firecrawl documentation. https://docs.firecrawl.dev/, 2026. Accessed: 2026-05-26.

[9] GetFPV. FPV beginner guide. https://www.getfpv.com/learn/new-to-fpv/fpv-beginner-guide/, 2026. Accessed: 2026-05-26.

[10] Google. Gemini Embedding 2: Our first natively multimodal embedding model. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-embedding-2/, 2026. Published: 2026-03-10. Accessed: 2026-05-26.

[11] Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing deep neural networks with logic rules. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 2410–2420, 2016.

[12] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, 2020.

[13] LangChain. LangChain text splitters documentation. https://python.langchain.com/docs/concepts/text_splitters/, 2026. Accessed: 2026-05-26.

[14] LangChain. LangGraph graph api documentation. https://docs.langchain.com/oss/python/langgraph/graph-api, 2026. Accessed: 2026-05-26.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[16] Wei Li, Wenhao Zhao, Sujian Li, and Xiaojun Sun. Faithfulness in natural language generation: A systematic survey of analysis, evaluation and optimization methods. *arXiv preprint arXiv:2203.05227*, 2022.

[17] Oscar Liang. Quadcopter hardware overview — every component explained. https://oscarliang.com/build-a-quadcopter-beginners-tutorial-1/, 2024. Accessed: 2026-05-26.

[18] LlamaIndex. LlamaIndex metadata extraction documentation. https://developers.llamaindex.ai/python/framework/module_guides/indexing/metadata_extraction/, 2026. Accessed: 2026-05-26.

[19] Market.us. FPV racing drone market growth, CAGR impact by 17.80%. https://market.us/report/fpv-racing-drone-market/, 2025. Accessed: 2026-02-18.

[20] Cheng Niu, Yuanhao Wu, Juno Zhu, Siliang Xu, Kashun Shum, Randy Zhong, Juntong Song, and Tong Zhang. RAGTruth: A hallucination corpus for developing trustworthy retrieval-augmented language models. *arXiv preprint arXiv:2401.00396*, 2024.

[21] OpenRouter. Google: Gemini Embedding 2 Preview api quickstart. https://openrouter.ai/google/gemini-embedding-2-preview/api, 2026. Released: 2026-04-17. Accessed: 2026-05-26.

[22] OpenRouter. OpenRouter — the unified interface for LLMs. https://openrouter.ai/, 2026. Accessed: 2026-05-25.

[23] Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, 2023.

[24] Jon Saad-Falcon, Omar Khattab, Christopher Potts, and Matei Zaharia. ARES: An automated evaluation framework for retrieval-augmented generation systems. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024.

[25] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation. *Findings of the Association for Computational Linguistics: EMNLP*, pages 3784–3803, 2021.

[26] Stats Market Research. Global FPV drone forecast market. https://www.statsmarketresearch.com/global-fpv-drone-forecast-market-8058586, 2025. Accessed: 2026-02-18.

[27] Unmanned Tech. FPV drone component buying guide — parts compatibility. https://help.unmannedtech.co.uk/portal/en-gb/kb/articles/fpv-drone-component-buying-guide-parts-compatibility, 2026. Accessed: 2026-05-26.

[28] Xiaohua Wang, Zhenghua Wang, Xuan Gao, Feiran Zhang, Yixin Wu, Zhibo Xu, Tianyuan Shi, Zhengyuan Wang, Shizheng Li, Qi Qian, Ruicheng Yin, Changze Lv, Xiaoqing Zheng, and Xuanjing Huang. Searching for best practices in retrieval-augmented generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 17716–17736. Association for Computational Linguistics, 2024.

[29] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[30] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023.