**Design and Subtask Breakdown for Rule Engine v2 (LegalTech Contract Analysis)**

**Decomposition of Tasks**

* **Policy Pack Loader & Rule Type Support:** Develop a unified **PolicyPack Loader** module that can discover and load rules written in three formats: *YAML-only*, *Python-only*, and *Hybrid (YAML+Python)*. The loader should detect if a hybrid version of a rule exists and give it precedence (hybrid > python > yaml). It will expose a common interface to execute any rule, regardless of format, and ensure all rule outputs conform to the unified Finding schema (see SSOT Finding below).
* **Rule Definition DSL (YAML) & RuleVM:** Design a declarative DSL for expressing rules in YAML. Each YAML rule will have sections like applies\_if (conditions to check applicability), checks (the logical conditions to evaluate), and on\_fail (the outcome if checks fail). The DSL should support logical operators (all, any, not), comparison ops (eq, not\_eq, in, range, etc.), and the ability to reference extracted evidence from the contract text. Implement a **Rule Virtual Machine (RuleVM)** to interpret this DSL with deterministic execution. The RuleVM should strictly validate YAML rules against a schema (to catch errors in rule definitions) and produce consistent results given the same inputs. Any complex logic that cannot be easily expressed in YAML DSL can be handled via hybrid rules (YAML structure calling into Python code for specific checks).
* **Unified Finding Schema (SSOT):** Implement a **Single Source of Truth (SSOT) Finding** data model as a Python class or schema that all rules use to return results. Define fields including: id (unique finding ID), pack (policy pack name), rule\_id (rule identifier within the pack), title (object with en and uk strings), severity (e.g. High/Medium/Low), category (e.g. “CrossCheck”, “ClausePresence”, etc.), message (en/uk), explain (en/uk), suggestion (en/uk), evidence (array of excerpts or data points from the contract), citation (array of reference URLs or IDs if any), flags (any boolean flags or tags for the finding), meta (additional metadata), version (rule version), created\_at (timestamp), and engine\_version (the rule engine version that generated the finding). This class will be used to format outputs from both YAML and Python rules, ensuring consistency.
* **Multilanguage Support (i18n):** Extend rule definitions to include bilingual text. Each rule’s user-facing strings (title, message, explanation, suggestion) must be provided in **English (EN)** and **Ukrainian (UK)**. Implement a fallback mechanism such that if a Ukrainian text is missing for a given field, the system defaults to English[i18next.com](https://www.i18next.com/principles/fallback#:~:text=Fallback%20to%20different%20languages). Create an **i18n validation tool** that scans all rules to ensure coverage (every English string has a corresponding Ukrainian translation). This can be integrated into tests or build process to prevent untranslated content. The Finding model will carry both languages, and the UI or output can choose the appropriate locale (with English as default).
* **Rule Refactoring & Migration Adapter:** Take the ~17 existing Python-based rules and ~3 YAML-based policies from Rule Engine v1 and rewrite them using the new policy-pack format (YAML or hybrid as appropriate). Each original rule’s logic needs to be preserved. Develop a **migration adapter** that can temporarily wrap old-format rules to produce new-format Findings, if needed, to ensure backward compatibility during the transition. Document the changes in a MIGRATION.md guide – explaining the new rule file structure, how to translate old rule code or YAML into the new DSL format, and noting any differences in behavior. The adapter will allow the engine to load legacy rules (if any remain) and convert their output to the SSOT Finding schema, to ease incremental migration.
* **Cross-Check Rules Implementation:** Implement a set of cross-check rules covering each pair in the given matrix of related clauses. For each relationship (e.g. **governing law ↔ jurisdiction**), write rules that verify the presence and consistency of both clauses:
  + *Governing Law vs. Jurisdiction:* Ensure that if a contract specifies a governing law, it also specifies the jurisdiction/forum for disputes. If, for example, a governing law is stated but no jurisdiction clause is present, the rule should flag this gap (and vice-versa). This prevents ambiguity since both the applicable law and the dispute forum should be clear[lawinsider.com](https://www.lawinsider.com/clause/governing-law-and-jurisdiction#:~:text=The%20governing%20law%20and%20jurisdiction,how%20disputes%20will%20be%20handled).
  + *Arbitration Clause vs. Dispute Resolution:* Ensure that if an arbitration clause is present, it aligns with or is referenced by the general dispute resolution clause. The contract should not contain conflicting dispute resolution methods (e.g. both court litigation and binding arbitration clauses without clarification). Flag if there's an arbitration clause but no overarching dispute resolution section, or if both exist but are contradictory.
  + *Limitation of Liability vs. Indemnity:* These clauses should be consistent in scope. Implement rules to check that if an indemnity clause is broad, an appropriate limitation of liability clause exists to cap or exclude certain liabilities, and vice versa. If one is present without the other, or if their terms are misaligned (e.g. indemnity covers certain losses that the liability clause caps differently), raise a finding.
  + *Confidentiality vs. Data Protection (UK GDPR):* If the contract includes a confidentiality clause, check for a data protection clause addressing GDPR or privacy, especially if personal data is involved. Conversely, if there's a GDPR/data protection clause, ensure a confidentiality clause exists. The rule should trigger if one is present without the other relevant protection, as both confidentiality and data handling terms are typically needed in modern contracts.
  + *Termination for Convenience vs. Early Termination Penalties:* If a party can terminate for convenience, the contract may specify an early termination fee or penalty. Cross-check that whenever a **termination for convenience** clause appears, any associated penalty or compensation clause is present (and reasonable), and vice versa (if an early termination fee clause exists, there should be a clause allowing termination for convenience). Flag missing corresponding clauses.
  + *IP Ownership vs. Licensing/Work-for-Hire:* Ensure that intellectual property ownership clauses are coupled with appropriate licensing or work-for-hire provisions. For example, if the contract states that all IP developed is owned by the client, verify there's no conflicting licensing clause granting rights back to the developer (or that work-for-hire language supports the ownership clause). If IP ownership is mentioned without clarifying how rights are transferred (work-for-hire) or what licenses are granted, the rule should flag it.
  + *OGUK MSA Clauses (Knock-for-Knock, Consequential Loss, Pollution Liability):* In Oil & Gas UK Model Service Agreements, certain risk-allocation clauses appear together. Implement rules to verify that if the contract is identified as an OGUK MSA (perhaps via metadata or keywords), it contains the standard **knock-for-knock indemnities**, an exclusion of **consequential losses**, and provisions for **pollution liability**. If any of these standard clauses is missing or significantly deviates from the expected form, flag it. These cross-checks ensure the package of clauses is complete and consistent with industry practice.
* **Performance Optimization:** Ensure the rule engine can handle **200+ rules with a 95th percentile execution time under 300 ms**. Achieving this will involve writing efficient code for the RuleVM (e.g., avoiding excessive nested loops), using compiled regex or pre-parsed contract representations for text-heavy operations, and possibly lazy-loading or caching results of expensive checks. All rules should be implemented as pure functions (no side effects or external I/O), and no randomness should be used, so that performance is predictable and results are deterministic. Consider grouping or short-circuiting checks where possible (e.g., if applies\_if fails, skip the rest of the rule evaluation early). We may also implement an optional parallel execution for independent rules to leverage multiple CPU cores (if needed to meet performance), but given the 300ms target, a single-thread optimized approach might suffice. Include performance tests to validate the engine meets this requirement.
* **Versioning, Deprecation & Feature Flags:** Introduce a versioning scheme for policy packs (using semantic versioning). Each policy pack definition will include its own version number and an engine\_min\_version requirement. The loader should check this to ensure compatibility; if the engine’s version is lower than required, the pack should be skipped or raise a clear error. Build in support for deprecating rules or packs: e.g., allow a rule metadata flag for deprecation (the engine can log a warning or omit deprecated findings unless explicitly enabled). Implement **feature flags** support such that certain rules can be toggled on/off based on configuration or environment (useful for rolling out new rules gradually or disabling ones that don't apply in certain jurisdictions). This likely involves a registry or configuration file where each rule/pack has an enabled flag or required feature tag. Make sure the system gracefully handles unknown or disabled rules (they simply don’t execute).
* **Documentation & Schema Definitions:** Write a **README\_rules.md** that explains how to write new rules in the system: covering the YAML DSL syntax (applies\_if/checks/on\_fail structure, available operations, how to include evidence hooks, etc.), how to add translations, severity levels, and examples of a simple rule and a hybrid rule. Provide a **JSON Schema** (or YAML schema) file for the DSL to help with validation and editor auto-complete when writing rules. This schema should enforce required fields and allowed keywords/operators in YAML rules. Also update the central **registry** of rules/packs, which could be a simple index (like a Python list or a YAML manifest) listing all policy packs to load. The registry ensures the engine knows what packs to load (or it can load all packs in a directory automatically).

**Acceptance Criteria**

1. **Unified Loader Functionality:** The system can load and execute rules from all three formats. For example, if a certain rule is defined in both YAML and Python, the engine chooses the hybrid definition (if present) or the Python definition over YAML, as per priority. A successful load of the rule packs yields a collection of rule objects ready for evaluation. *Verification:* Create dummy rules in each format and ensure the loader picks the correct one and executes without error, returning a Finding.
2. **Correct Finding Output Schema:** Every rule execution returns a Finding in the standardized JSON structure containing all required fields (id, pack, rule\_id, title, severity, category, message, explain, suggestion, etc. in both languages). For instance, running a cross-check rule on a contract that is missing a Jurisdiction clause (while having a Governing Law clause) should produce a Finding similar to the example below:

{

"id": "CC-001",

"pack": "CrossChecks",

"rule\_id": "jurisdiction\_missing",

"title": {

"en": "Missing Jurisdiction Clause",

"uk": "Відсутнє положення про юрисдикцію"

},

"severity": "Medium",

"category": "CrossCheck",

"message": {

"en": "The contract specifies a governing law but lacks a jurisdiction clause.",

"uk": "У контракті визначено застосовне право, але відсутнє положення про юрисдикцію."

},

"explain": {

"en": "Without a jurisdiction clause, it may be unclear which court or forum will handle disputes, leading to uncertainty.:contentReference[oaicite:2]{index=2}",

"uk": "Без положення про юрисдикцію незrozуміло, який суд чи форум вирішуватиме спори, що призводить до невизначеності."

},

"suggestion": {

"en": "Add a jurisdiction clause to clarify the dispute resolution forum.",

"uk": "Додайте положення про юрисдикцію, щоб визначити форум вирішення спорів."

},

"evidence": [

"Clause 12. Governing Law: New York"

],

"citation": [],

"flags": [],

"meta": {},

"version": "2.0.0",

"created\_at": "2025-08-28T12:08:13Z",

"engine\_version": "2.0.0"

}

*Verification:* Check that the Finding includes the English and Ukrainian text for all message fields, and that evidence captures the relevant contract excerpt (in this case, the governing law clause text). The severity and category match the rule definition. All timestamps and version fields are correctly populated. The example above illustrates the expected output format.

1. **Bilingual Messaging & Fallback:** For every rule, when generating a Finding, the message, explain, and suggestion fields appear in the appropriate language. If a Ukrainian translation is missing for a given rule field, the system falls back to English gracefully[i18next.com](https://www.i18next.com/principles/fallback#:~:text=Fallback%20to%20different%20languages). *Verification:* Temporarily remove or blank out a message.uk in a rule definition and run that rule – the resulting Finding should show the English text in both message.en and message.uk (or message.uk may repeat the English content as fallback). Also, run the i18n coverage checker on the rule set: it should report no missing translations (or only expected fallbacks). The build should fail if any required translation is absent.
2. **Cross-Check Rule Outcomes:** Each cross-check in the matrix triggers the correct Finding when conditions are met. Examples of acceptance scenarios:
   * If a contract has a **Governing Law** clause but no **Jurisdiction** clause, a Medium-severity Finding is produced warning about the missing jurisdiction clause (as shown above). Conversely, if a jurisdiction/forum clause exists without a governing law, a similar Finding is raised for missing governing law. If both clauses are present and consistent, no finding from this rule should appear.
   * If a contract contains an **Arbitration** clause but also a separate **litigation** clause (e.g. naming courts) in a way that conflicts, a Finding is raised noting the inconsistency in dispute resolution clauses. If an arbitration clause is present *and* no general dispute resolution clause is found, that should also be flagged (perhaps as a different message).
   * For **Limitation of Liability and Indemnity**, if an indemnity clause is present but no liability cap or exclusion clause is found (or vice versa), a Finding is generated pointing out the imbalance (e.g. high severity if unlimited liability risk). If both are present, the rule might also check for logical consistency (though exact consistency checking could be complex, basic presence is minimum).
   * **Confidentiality vs Data Protection:** On an NDA or services contract that has a confidentiality clause, ensure a data protection clause is present if personal data is involved (for test, maybe flag if certain keywords like "personal data" or "GDPR" appear without a dedicated clause). The Finding should suggest adding the missing clause.
   * **Termination vs Penalties:** If "termination for convenience" is found with no mention of an early termination fee where one would be expected, flag it. Similarly, if a contract mentions a termination fee but doesn't actually allow termination for convenience, that should be flagged as odd (though that scenario is rare, it could be a clause leftover).
   * **OGUK MSA trio:** Given a contract marked or recognized as an OGUK Master Service Agreement, remove one of the three key clauses in a test and ensure the engine flags that omission. For example, omit the consequential loss exclusion – the rule should produce a Finding like "Missing Consequential Loss Exclusion in OGUK MSA", with explanation and suggestion to include the standard clause. If all three clauses are present in an OGUK contract, no findings (for those rules) occur.  
     *Verification:* Create test contract snippets for each scenario and run the full rule set. Confirm that each specific cross-check rule produces the intended Finding (with correct id, messages, severity) only when the conditions are met. No false alarms should trigger when both parts of a pair are present properly.
3. **YAML DSL Execution & Logic:** The YAML-defined rules using the DSL operate correctly. Complex boolean logic in applies\_if and checks is evaluated as expected. For example, define a sample rule in YAML where applies\_if: all: [cond1, cond2] and checks: any: [cond3, cond4]. Ensure that if the context meets cond1 and cond2, the rule runs, and it fails if neither cond3 nor cond4 is true, triggering the on\_fail. Also test negation: a condition with not: cond5 in YAML properly inverts the logic. Range checks (e.g. a number falls within a range) and membership checks (in lists) function as intended.  
   *Verification:* Unit-test the RuleVM with a variety of synthetic rules: e.g., a rule that triggers only if a numeric value is outside a range, or a rule that gathers evidence using an evidence hook (like capturing the actual contract text that matched a regex). Confirm that the evidence hook collects the correct snippet from the input document (e.g., if a rule checks for the presence of "GDPR", the evidence might be the sentence containing "GDPR" in the contract). All such rules should yield deterministic outcomes given fixed input.
4. **Strict Schema Validation:** If a YAML rule file has an incorrect structure or unknown field, the loader should refuse to load it (or log an error) rather than quietly misinterpreting it. *Verification:* Introduce a deliberate error in a test YAML (e.g. a typo in a keyword like applies\_iff or an unsupported operator) and confirm that the engine catches it during loading or validation, producing a meaningful error message pointing to the schema or rule definition issue.
5. **Performance Benchmarks:** When running the full suite of ~20 rules (and anticipating scalability to 200+), the engine meets the performance target. *Verification:* Using a representative large contract (or a synthetic document with many clauses), time the execution of 200 rules. The 95th percentile of run times should be under 300 ms on a typical modern CPU. If needed, run the test multiple times and ensure consistent performance. Additionally, verify no network calls or external API calls are made during rule execution (inspect that all rules use pure functions and local data only). Memory usage should remain modest (since rules are stateless and mostly string comparisons/regex, etc.).
6. **Test Coverage Metrics:** The test suite covers ≥85% of lines of code and ≥95% of the critical logic branches. *Verification:* Run the coverage tool after all unit and integration tests. Key modules like the PolicyPack Loader, RuleVM interpreter logic, multilingual handling, and each cross-check rule’s logic should show near-100% coverage. Any critical branch (for example, both outcomes of an if-else in a complex function) should be exercised by some test. The pipeline should report failure if coverage drops below the threshold.
7. **Versioning & Compatibility:** The system correctly handles policy pack version metadata:
   * If a policy pack’s version is below the engine’s current version or marked deprecated, it still loads but might log a deprecation warning.
   * If a pack requires a higher engine version (engine\_min\_version), the loader skips it and the system logs an informative message (or raises an exception in strict mode) that the pack is not compatible.  
     *Verification:* Create a dummy pack with engine\_min\_version set higher than the current engine version in tests and confirm it does not execute (and a warning is emitted). Mark a pack or rule as deprecated in metadata and ensure that info is carried into the Finding (perhaps via a flag in the Finding or a log) but does not stop execution. Similarly test feature flagging: define a rule that is tied to a feature flag (simulate by a config setting); toggle the flag in tests to see that with flag off the rule is skipped, and with flag on the rule runs.
8. **Migration Accuracy:** The new Rule Engine v2 produces findings equivalent to the old version for the same input contracts. *Verification:* Run a set of 5 sample contracts through both the old engine (v1) and the new engine (v2 with migrated rules via adapter) and compare the outputs. The findings list should be identical or with only minor expected differences (e.g., improved wording in messages if that was intentionally changed). Document any differences in MIGRATION.md. All migrated rules (the 17 Python and 3 YAML from v1) should be traceable in the new system (same or improved logic, same identifiers if possible). The MIGRATION.md should be reviewed to ensure it clearly instructs how to update custom rules: for example, it might show a before/after of a simple rule in old Python style vs new YAML DSL format.

**Test Plan**

**Unit Tests:**

* *Rule Logic Units:* Write unit tests for each rule’s logic (especially for complex Python-only or hybrid rules). For YAML rules, this can be done by feeding the RuleVM a controlled input. For example, test a **governing law vs jurisdiction** rule by simulating contract text that has one clause but not the other, and assert that the Finding returned matches expected values (and conversely test with both clauses present to assert no finding or a specific outcome). Repeat for each cross-check rule and a representative selection of other rules to ensure individual correctness.
* *DSL Condition Evaluations:* Create small test cases for the DSL interpreter:
  + A rule with all conditions vs any conditions (ensure that all requires all sub-conditions True, any requires at least one).
  + Test not operator wrapping another condition, confirming it inverts the result.
  + Test comparison operators: eq (equal), not\_eq (not equal), numeric comparisons, in (membership in list of values), and range (value within a range). For each, craft a context that should pass and one that should fail.
  + Evidence capture: If a rule uses an evidence hook (e.g., capturing the exact text that matched a regex in checks), simulate a contract text and ensure the evidence array in the Finding contains the expected snippet or value.
  + Schema validation: Feed an invalid YAML definition (via a unit test that calls the loader on a bad rule) and assert that a schema validation error is raised. Also test a valid rule passes without error.
* *PolicyPack Loader:* Test the loader with various scenarios:
  + Only YAML rules present -> all loaded.
  + Only Python rules present -> all loaded.
  + Both YAML and Python for the same rule -> ensure the loader loads it as hybrid (perhaps by designating that the YAML references a Python function, or by naming conventions). Check that the loaded rule’s type is correctly identified.
  + Mixed version compatibility: loader should skip or flag a pack with incompatible version as per design (simulate engine version lower/higher).
  + Feature flag filtering: simulate environment or config where a certain feature is disabled, and verify the loader (or execution stage) skips rules marked for that feature.
* *Multilanguage Fields:* Unit test the i18n fallback: create a Finding with a missing Ukrainian message and run a function that applies the fallback, expecting the Ukrainian field to be filled with the English text. Also test the function that verifies translation coverage: feed it a set of rule definitions where one is intentionally missing a translation and assert that it reports this error.
* *Performance Micro-benchmarks:* Although full performance is an integration concern, also include a micro-benchmark test for the RuleVM by running a moderately complex rule many times on a small input to ensure the loop overhead is minimal. Use Python’s timing or profiling to catch any obvious slow spots. (These tests might be marked separately or run in a profiling mode rather than every CI run.)

**Integration Tests:**

* *Full Contract Analysis:* Prepare about **5 typical contracts** (as text fixtures) covering various scenarios: e.g. an NDA (with confidentiality, no data protection – should trigger that rule), a Service Agreement with governing law but missing jurisdiction (trigger cross-check), a Software Development contract covering IP (ensure IP vs licensing rules), a contract with arbitration clause (for dispute resolution cross-check), and an OGUK-style contract. For each contract, run the entire rule engine and collect all Findings.
  + Assert that the expected Findings appear (e.g., known issues in those contracts are caught). For instance, in the NDA fixture, we expect a finding about missing GDPR clause if personal data is mentioned.
  + Assert that no unexpected Findings appear (no false positives). If the NDA has both law and jurisdiction, the governing law/jurisdiction rule should not flag anything.
  + These tests essentially simulate real usage, ensuring that the combination of rules doesn’t produce duplicates or conflicts. (If two rules could potentially flag the same issue, we might see duplicate Findings – tests should check for and fail on unintended duplicate entries).
* *Cross-Checks End-to-End:* Specifically for cross-check pairs, create mini-contracts as strings focusing on those clauses. For example:
  + A snippet containing a governing law clause but missing jurisdiction. The integration test running all rules on this snippet should result in exactly one Finding (Missing Jurisdiction). It should not, for example, produce a null reference or crash if it tries to find a jurisdiction clause that isn’t there.
  + A snippet with an arbitration clause but also a clause submitting to courts – should result in one Finding about conflicting dispute clauses.
  + A snippet with an indemnity but no liability cap – expect the indemnity vs liability rule triggers.
  + And so on for each relationship. This ensures that in a realistic parsing of text, the rule conditions indeed pick up the patterns/keywords as intended.
* *Aggregate Output Validation:* After running the full pipeline on a sample contract, validate that every Finding in the list conforms to schema (both languages present, all required fields not null, etc.). This can be done by serializing findings to JSON and validating against the Finding JSON schema. Also verify sorting or ordering if the engine is supposed to sort findings by severity or document order – for example, ensure the output list is stable and logically ordered.
* *Concurrent Execution (if applicable):* If the engine supports parallel rule evaluation, test it by running with concurrency (e.g., a flag to execute rules in threads or processes). Ensure that the results are the same as single-thread (order might differ, but we can sort by finding id for comparison). Also ensure thread-safety of any shared structures (the design is mostly pure functions, so likely fine).
* *Coverage Enforcement:* The test suite should include a final assertion on coverage. Using a coverage tool, we can have an integration test (or part of CI) that fails if coverage % is below the target. This isn’t a typical runtime test, but part of the quality gates.

By executing this comprehensive test plan, we ensure each component (rules, loader, DSL, output, etc.) works in isolation and as a whole. The plan also checks for regressions of old rules (via comparing v1 vs v2 outputs) and verifies non-functional requirements like performance and localization.

**Risks & Mitigations**

* **Rule Overlap & Duplication:** With multiple rule formats and cross-checks, there's a risk of duplicate findings or overlapping logic (e.g., the same issue flagged by two rules). *Mitigation:* Use unique rule IDs and carefully scope each rule’s conditions. The policy packs should be reviewed for overlaps; if two rules could trigger on the same condition, consider merging them or using the more specific rule only. The loader’s priority (hybrid > python > yaml) also avoids duplicate execution of essentially the same rule defined in different ways. Integration tests will catch any duplicate outputs, and we can refine rules to eliminate redundancy.
* **False Positives/Negatives:** Rules based on patterns might misfire (false positive) or miss issues if phrasing varies (false negative). For example, the cross-check might not catch a jurisdiction clause if the contract uses unusual wording, or it might flag something as missing when the concept is addressed in an unexpected section. *Mitigation:* Improve the DSL conditions with more robust text matching (e.g., use regex patterns for synonyms, or preprocess the contract into a structured format like a clause list). We can incorporate clause detection or keywords in multiple languages. Start with clear patterns (to minimize false positives) and iteratively refine rules based on test contracts. Provide a mechanism to suppress a finding if needed (so users can mark a false positive and not be alerted again). Use a diverse set of test documents to broaden coverage of language variations.
* **Multi-language Clause Handling:** If the system needs to analyze contracts in both English and Ukrainian, rules might need to detect clauses in either language. There’s a risk that our rules (and regex patterns) are initially English-centric. *Mitigation:* Identify if Ukrainian contracts are in scope. If yes, extend the DSL or rule definitions to include language-specific keywords (possibly the YAML DSL can allow different regex for different locales). Alternatively, maintain separate policy packs for Ukrainian documents. Ensure the evidence hooks and patterns are Unicode-safe and can handle Cyrillic text. For output, ensure the Ukrainian translations of findings are accurate and contextually appropriate; have bilingual legal experts review the message.uk contents to avoid mistranslation of legal terms.
* **Performance Degradation:** With the addition of a VM and more rules, the engine might slow down, especially if many rules scan the same document independently. *Mitigation:* Optimize by parsing the contract once and sharing results. For instance, if multiple rules need to know if a "Governing Law" clause exists, do a single search for that clause and store it (perhaps in a context object) that rules can query, instead of each rule doing its own search. The RuleVM could support a pre-processing step to extract common features (like a dictionary of found clause types). Also, profile the system to find bottlenecks (e.g., regex matching could be costly; we might precompile regexes at load time). If needed, apply parallel execution for independent rules to utilize multi-core CPUs, but ensure thread safety. Avoid heavy computations in Python rules; if a complex analysis is needed, consider simplifying it or offloading it to pre-processing. Given the no I/O constraint and pure functions, most time will be CPU-bound string processing, which we will keep efficient (e.g., by using Python's built-in regex and string methods which are in C). Regular performance tests will guide if further optimization (or rule limiting) is required.
* **Migration and Backward Compatibility Issues:** There is a risk that in rewriting the rules, some logic might change unintentionally, causing different findings than before. Users might rely on specific wording or severity from the old engine. *Mitigation:* Use the migration adapter to run old and new rules side-by-side on sample documents and diff the results. Any differences must be intentional and documented (e.g., maybe we decided to raise severity of a certain issue – note that in MIGRATION.md). Keep as many rule IDs and categories the same as in v1 so that any consumers of the results (downstream systems) aren’t confused by renamed findings. MIGRATION.md will enumerate any removed or merged rules and how to update them. During the migration phase, support both formats (via adapter) so we can do a phased rollout and quickly switch back if a critical bug is found in a rewritten rule.
* **Complex DSL or Hybrid Failures:** The introduction of hybrid rules (mixing YAML and Python) and a custom DSL means potential for new kinds of bugs (e.g., a bug in the RuleVM interpreting a complex condition might cause a rule to never fire). *Mitigation:* Extensive testing of the DSL, and also providing an escape hatch: if something is too tricky to do in DSL reliably, allow the rule to be a Python-only implementation. The rule loader priority ensures that if a hybrid rule is problematic, we could disable the YAML part and let a Python version run. Also, include logging in the RuleVM – if a rule fails to evaluate due to an error, catch it and log which rule failed, so it's easier to debug. Provide good error messages for rule writers (e.g., "Rule X in pack Y has invalid syntax at ...").
* **Maintaining Multi-language Content:** Ensuring the English and Ukrainian texts stay aligned over time can be challenging (e.g., someone updates the English message but forgets to update the Ukrainian). *Mitigation:* The i18n validation as part of the test pipeline will catch missing translations. We could also implement a script to list all message pairs for easy review by translators. Possibly maintain a central dictionary of common terms (legal glossary) to ensure consistency in translations. Another mitigation is to use a translation key approach (store keys in rules, and have separate locale files), but given the requirements, likely the text will be directly in the rule definitions for simplicity. Regular reviews of the Ukrainian text by legal bilingual staff is recommended to ensure quality.
* **Feature Flag Misconfiguration:** If feature flags are used to toggle rules, there's a risk a rule might be unintentionally turned off (missing an important check) or on (an experimental rule affecting results prematurely). *Mitigation:* Clearly document the feature flags in README\_rules.md. Maybe provide a command-line or config dump that lists which rules/packs are enabled/disabled for a run. In tests, explicitly exercise both flag states. Default configurations should enable all stable rules, with new ones having to be explicitly toggled on – thereby avoiding surprises.
* **Deprecation Handling:** Marking rules or packs as deprecated but still present could confuse developers or users (e.g., two findings for same issue until deprecated is removed). *Mitigation:* Use deprecation sparingly and communicate it. The engine could tag deprecated findings with a flag or special note so that users know it's from a rule that will be removed. Have a plan to actually remove or update deprecated items in a future version to avoid lingering dead code. Test that deprecation warnings appear as expected when using a deprecated pack.
* **OGUK Identification Risk:** The OGUK-specific rules need a reliable way to know the contract is intended to follow the OGUK standard, otherwise they might flag any contract that lacks those clauses (which would be a false positive if the contract isn’t an oil & gas contract at all). *Mitigation:* Perhaps require an explicit marker (metadata or user input) that a document is an "OGUK MSA" type, or detect certain telltale phrases in the document (like referencing "Knock for knock" or industry terms). The applies\_if for these rules can include such conditions. This way, an unrelated contract won't trigger OGUK clause warnings. Test with non-OGUK contracts to ensure these rules stay silent.

By anticipating these risks and implementing the mitigations above, we aim to ensure the Rule Engine v2 is robust, accurate, and maintainable in the long term.

**Deliverables**

By the end of this project, the following artifacts will be delivered in the pull request (PR):

* **MIGRATION.md:** A markdown document explaining how to migrate rules from the old system to the new policy-pack format. It will list changes in the rule structure (YAML schema vs old format), examples of old vs new rule definitions, and any changes in rule IDs or behavior. This is for developers maintaining custom rules or understanding the impact of the upgrade.
* **README\_rules.md:** Documentation for the Rule Engine v2, including an overview of the rule types, how to write YAML DSL rules, how to integrate Python logic, how multilingual fields work, and guidelines for adding new rules or packs. It will likely include code snippets and maybe a quick start on adding a new policy pack.
* **Rewritten Rule Packs (rules/packs/...):** The new set of rule definitions. Likely organized as multiple YAML files (and possibly some Python modules for hybrid logic). For example, there may be a general\_rules.yaml (with general contract rules), cross\_checks.yaml (for the cross-check matrix rules), oguk\_rules.yaml, etc., or possibly one big YAML containing all. Each pack file will contain multiple rules defined in the new DSL format, with appropriate metadata (version, name, engine\_min\_version, etc.). A total of around 20 rules (migrated 17+3 and new cross-check ones if those are additional) will be present. Python code for any rules that require it (hybrid or full Python rules) will also be included in this directory structure (e.g., a rules/packs/common\_checks.py if needed).
* **Policy Pack Registry:** An updated registry (could be a JSON/YAML file or Python enumeration) that lists all available policy packs and their metadata (name, version, maybe a short description). This registry is used by the loader to know what to load. If the design chooses auto-loading (e.g., load all YAML files in rules/packs), this may be just implicit. But we will provide the mechanism for the engine to recognize new packs easily.
* **DSL Interpreter & RuleVM Code:** The implementation of the DSL evaluation engine (RuleVM). Likely delivered as one or more Python modules (e.g., rulevm.py, or under engine/dsl\_interpreter.py). It will include the logic to parse/validate YAML rule structures and execute conditions. Also, any helper libraries for evaluating expressions (for example, functions to evaluate all, any, etc., and to perform comparisons). This also includes the YAML schema definition file (e.g., rule\_schema.json) used for validation.
* **Unified Finding Schema Definition:** The Finding class or schema will be part of the deliverables (maybe in a file like models/finding.py or similar). If a JSON Schema for findings is created (to validate outputs), include that as well. This schema ensures that any consumer of the rule engine results knows the exact format.
* **Migration Adapter Code:** A module that can ingest old-format rules or findings and output the new format (if this is needed for backward compatibility). This might be a temporary piece of code, but it will be included to facilitate a smooth transition (and possibly removed in a later version once migration is complete).
* **Test Suite:** All unit tests and integration tests as described, likely under a tests/ directory. This includes:
  + Unit test files for each core component (loader, DSL, each type of rule logic, i18n, etc.).
  + Integration test files that run the engine on sample contracts.
  + Test fixtures, such as sample contract texts or clause snippets, perhaps stored in tests/fixtures/ or generated within tests.
  + A coverage report configuration (e.g., coverage.cfg or just settings in CI) to enforce the 85%/95% criteria.
* **Fixtures (Sample Contracts):** At least five representative contract samples (or excerpts) to be used in integration testing. These may be included in the repository (scrubbed of any sensitive info, purely synthetic or from public templates). They will be documented or named in a way that indicates their type (e.g., fixtures/NDA\_sample.txt, fixtures/ServiceAgreement\_sample.txt, fixtures/OGUK\_MSA\_sample.txt, etc.). These help demonstrate the engine’s capability on real-world data.
* **Continuous Integration Configuration:** (If applicable) Updated CI pipeline scripts to run the tests and coverage, ensuring that the quality gates (performance, coverage, etc.) are checked on every run. This ensures ongoing compliance with the requirements.
* **Engine Version Bump:** The version of the rule engine will be updated (e.g., to 2.0.0) to reflect the new features. This might be in a \_\_init\_\_.py or a version file, which will be part of the PR. All policy packs will use this version or specify their compatible engine version.

All the above deliverables will be organized and referenced in the PR such that reviewers can easily find the new rule definitions, see the documentation for usage, and run the test suite to verify everything passes. The end result is a robust Rule Engine v2 ready for integration into the LegalTech contract analysis system, fulfilling the outlined requirements.