## A Interface Visualization

Figure 1 illustrates the client-side integration of BackportCheck within the Gerrit environment.To demonstrate the tool's utility, we describe the complete decision-making workflow: The interaction is initiated when the Release Manager clicks the **"Analyze Eligibility"** button injected into the review interface. The system then displays the decision overlay, immediately presenting the **Decision Banner (A)**. This provides immediate feedback via a color-coded probabilistic verdict, allowing the maintainer to instantly triage the change. To understand the rationale, the user reviews the **AI Explanation (B)**, where the Large Language Model synthesizes complex risk factors into a coherent natural language justification.

To validate this assessment, **the Risk Dashboard (C)** displays the key metrics, such as *Author Reliability* and *Code Spread*, allowing maintainers to verify that the AI's reasoning is grounded in hard data rather than opaque logic. Furthermore, the interface provides **Threshold Controls (D)** to adjust the decision boundary in real-time according to the user's context. Lastly, the **Metric Definitions (E)** component provides on-demand explanations for each indicator, ensuring that the calculation logic remains accessible and transparent.

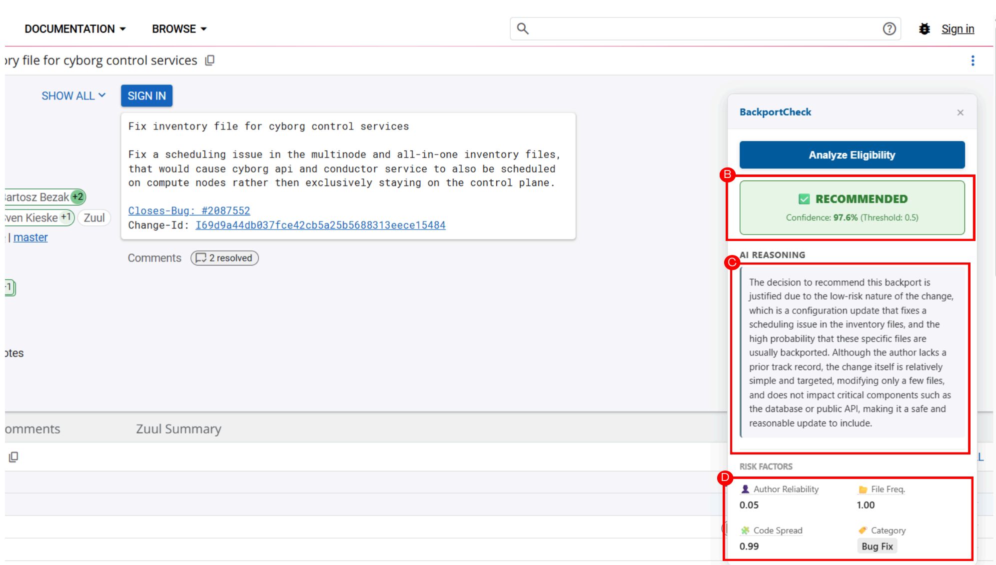


Fig. 1. **Annotated BackportCheck Interface.** The overlay augments the Gerrit code review screen with five decision-support layers: (A) The probabilistic verdict banner, (B) Natural language AI reasoning, (C) The quantitative risk dashboard, (D) Sensitivity threshold controls, and (E) Metric definitions for transparency.

## B BackportCheck features

Table 1 details the comprehensive list of 37 features extracted by the inference engine, categorized by the dimensions used in the risk reporting schema.

Table 1. **Taxonomy of State-Aware Features.** Features are extracted via static analysis of the changeset and historical metadata. Mathematical definitions are supplemented with Python implementation logic for reproducibility.

<table>
<tr><th>Dim.</th><th>Feature Variable</th><th>Description & Implementation Logic</th></tr>
<tr><td colspan="3">1. Complexity & Structure</td></tr>
<tr><td rowspan="3">Entropy</td><td>change_entropy</td><td>Quantifies the dispersion of modifications across the file set. High entropy indicates complex, scattered logic.<br>Logic: scipy.stats.entropy(lines_per_file)</td></tr>
<tr><td>safe_entropy_interaction</td><td>A risk-adjusted score that nullifies entropy penalties if the change is purely configuration.<br>Logic: change_entropy * is_pure_config</td></tr>
<tr><td>file_ext_entropy</td><td>Measures the heterogeneity of the technology stack (e.g., mixing SQL, Python, and Shell).<br>Logic: len(set(file_extensions))</td></tr>
<tr><td rowspan="4">Volume</td><td>churn_density</td><td>Proxies the "weight" of the edit; high density implies concentrated, heavy refactoring within few files.<br>Logic: total_lines / file_count</td></tr>
<tr><td>churn_log_size</td><td>Log-transformed total churn, normalizing the impact of massive automated refactors.<br>Logic: ln(1 + total_lines_changed)</td></tr>
<tr><td>file_count</td><td>Represents the "blast radius" or breadth of the modification footprint.<br>Logic: len(modified_file_paths)</td></tr>
<tr><td>deletion_ratio</td><td>Distinguishes subtractive refactoring (cleanup) from additive feature development.<br>Logic: deleted_lines / total_lines</td></tr>
<tr><td rowspan="3">Topology</td><td>directory_depth</td><td>Indicates architectural depth; deep changes often imply modifications to core business logic.<br>Logic: mean(path.count('/'))</td></tr>
<tr><td>config_line_ratio</td><td>Captures the dominance of configuration adjustments versus executable logic changes.<br>Logic: config_lines / total_lines</td></tr>
<tr><td>code_line_ratio</td><td>Captures the proportion of the changeset affecting source code files.<br>Logic: code_lines / total_lines</td></tr>
<tr><td colspan="3">2. Context & Environment</td></tr>
<tr><td rowspan="5">Meta</td><td>is_test_change</td><td>Identifies changes isolated to testing artifacts, implying minimal regression risk to production.<br>Logic: all('test' in f for f in files)</td></tr>
<tr><td>is_ci_change</td><td>Flags modifications to Continuous Integration pipelines or gate definitions.<br>Logic: files ∩ {.zuul.yaml,zuul.d,.gitlab-ci.yml, .travis.yml, tox.ini, bindep.txt}</td></tr>
<tr><td>is_deploy_project</td><td>Contextual flag indicating the repository serves a deployment role (e.g., Ansible) vs core logic.<br>Logic: project ∈ {kolla, kayobe, ...}</td></tr>
<tr><td>is_bot</td><td>Detects automated maintenance commits which exhibit distinct acceptance patterns.<br>Logic: author matches r'bot|zuul|jenkins'</td></tr>
<tr><td>has_gerrit_topic</td><td>Indicates if the change is linked to a broader semantic topic series in the review system.<br>Logic: metadata.topic is not NULL</td></tr>
<tr><td colspan="3">3. Semantics (NLP on Commit Message M & Subject S)</td></tr>
</table>

Continued on next page

**Table 1 – continued from previous page**

<table>
<tr><th>Dim.</th><th>Feature Variable</th><th>Description & Implementation Logic</th></tr>
<tr><td rowspan="6">Intent</td><td>`is_fix`</td><td>Captures corrective intent, signaling a bug patch or crash resolution.<br>*Logic:* `S matches r'\b(fix|resolve|repair|patch|correct|handle|mitigate|prevent)\b'`</td></tr>
<tr><td>`is_feature`</td><td>Signals the introduction of novel functionality or capabilities.<br>*Logic:* `S matches r'\b(add|implement|support|introduce|new|feat|enable|allow|provide)\b'`</td></tr>
<tr><td>`is_refactor`</td><td>Identifies structural reorganization without functional alteration.<br>*Logic:* `S matches r'\b(refactor|clean|remove|move|rename|delete|drop)\b'`</td></tr>
<tr><td>`is_revert`</td><td>Explicitly flags a rollback operation, indicating immediate regression mitigation.<br>*Logic:* `S startswith "Revert "`</td></tr>
<tr><td>`is_maintenance`</td><td>Detects routine housekeeping, dependency bumps, or version pinning.<br>*Logic:* `S matches r'\b(update|bump|upgrade|downgrade|pin|unpin|sync)\b'`</td></tr>
<tr><td>`is_deployment`</td><td>Signals intent related to environment configuration or installation logic.<br>*Logic:* `S matches r'\b(config|conf|deploy|install|set|use|default|variable|param|role)\b'`</td></tr>
<tr><td rowspan="4">Clarity</td><td>`readability_ease`</td><td>Evaluates the cognitive effort required to comprehend the commit message.<br>*Logic:* `textstat.flesch_reading_ease(M)`</td></tr>
<tr><td>`msg_gunning_fog`</td><td>Estimates the formal linguistic complexity of the descriptive text.<br>*Logic:* `textstat.gunning_fog(M)`</td></tr>
<tr><td>`ref_bug_tracker`</td><td>Indicates strict process adherence by linking to an external issue tracker.<br>*Logic:* `M matches r'(Closes-Bug|Bug): #\d+'`</td></tr>
<tr><td>`has_subject_tag`</td><td>Detects the use of conventional brackets for categorizing changes.<br>*Logic:* `S matches r'\[.*?\]'`</td></tr>
<tr><td colspan="3">4. Risk Flags (Heuristic Domain Detection)</td></tr>
<tr><td rowspan="4">High Risk</td><td>`modifies_db`</td><td>Critical flag for database schema alterations, implying high compatibility risk.<br>*Logic:* `path matches r'.*(alembic|migration).*'`</td></tr>
<tr><td>`modifies_api`</td><td>Critical flag for modifications to external-facing interfaces.<br>*Logic:* `path matches r'.*(api/|v1/|v2/).*'`</td></tr>
<tr><td>`modifies_deps`</td><td>Flags alterations to the project's build graph or library requirements.<br>*Logic:* `files ∩ {requirements.txt, setup.py}`</td></tr>
<tr><td>`has_security`</td><td>Detects explicit mentions of vulnerabilities or CVE identifiers.<br>*Logic:* `M matches r'CVE-\d+|Security'`</td></tr>
<tr><td rowspan="3">Safety</td><td>`is_pure_config`</td><td>Identifies changes composed almost entirely (> 99%) of configuration data.<br>*Logic:* `config_line_ratio > 0.99`</td></tr>
<tr><td>`is_doc_only`</td><td>Identifies risk-neutral changes limited to documentation files.<br>*Logic:* `all(f.endswith(.rst|.md) for f in files)`</td></tr>
<tr><td>`modifies_config`</td><td>Flags the presence of any configuration modification within the changeset.<br>*Logic:* `any(f.endswith(yaml|yml|.json|.ini|.conf|.toml|.xml|.j2|.rst|.md|.erb) for f in files)`</td></tr>
<tr><td colspan="3">5. Historical Context (Computed recursively at time t)</td></tr>
</table>

Continued on next page

**Table 1 – continued from previous page**

| Dim. | Feature Variable | Description & Implementation Logic |
|---|---|---|
| Author | `author_trust` | Proxies the author's efficiency: high submission volume relative to churn. *Logic:* `count / (1 + cumulative_churn)` |
| | `author_success` | The author's historical acceptance probability within the project. *Logic:* `accepted_commits / total_submissions` |
| | `author_subs` | Quantifies the author's cumulative experience level. *Logic:* `len(history[author])` |
| Env | `project_rate` | Establishes the baseline acceptance probability for the specific repository. *Logic:* `project_accepted / project_total` |
| | `file_risk_prob` | Estimates the "riskiness" of files based on their specific modification history. *Logic:* `max(acceptance_rate(f) for f in files)` |

## B.1 Feature Interpretation and Rationale

To interpret the raw feature vectors extracted by the system, we provide illustrative examples demonstrating how specific metrics serve as proxies for software risk.

*B.1.1 Complexity: Entropy as a Proxy for Coupling.* The `change_entropy` feature differentiates between "atomic" changes and "cross-cutting" modifications.

- **Scenario A (Low Entropy):** A commit modifies 100 lines within a single file (e.g., `main.py`). The distribution is $[1.0]$, resulting in an entropy of 0.0.
  *Interpretation:* The change is highly cohesive. If a regression occurs, it is likely isolated to this specific module, lowering the system-wide risk.
- **Scenario B (High Entropy):** A commit modifies the same 100 lines, but splits them evenly between a logic file (`main.py`) and a utility library (`utils.py`). The distribution is $[0.5, 0.5]$, resulting in an entropy of $\approx 0.69$.
  *Interpretation:* The logic bridges multiple modules. This indicates tighter coupling and requires the reviewer to verify that the interaction between these distinct files remains valid, increasing the cognitive load and risk of side effects.

*B.1.2 Volume: Churn Density vs. Log Size.* While total lines changed measures magnitude, `churn_density` measures complexity concentration.

- **High Density (Concentrated Risk):** A change of 200 lines inside just 1 file yields a density of 200. This suggests a deep, complex rewrite of a specific algorithm, requiring careful logical verification.
- **Low Density (Scattered Risk):** A change of 200 lines scattered across 200 files yields a density of 1. This pattern is characteristic of mechanical refactoring (e.g., renaming a variable globally) or formatting fixes, which are generally lower risk despite the large total volume.

*B.1.3 Topology: Directory Depth.* The `directory_depth` feature acts as a heuristic for the architectural significance of the change.

- **Surface Level (Depth $\approx 1$):** Changes occurring at the root (e.g., `./README.md` or `./requirements.txt`) often affect project metadata or dependencies rather than core execution logic.
- **Core Level (Depth $> 4$):** Changes occurring deep in the directory tree (e.g., `./src/drivers/net/api/v` typically target specific, implementation-heavy business logic that is less visible but critical for system stability.

*B.1.4 Context: The Author Trust Score.* The `author_trust_score` penalizes high churn coming from inexperienced contributors.

- **High Trust:** An author who has made 50 submissions with a cumulative churn of only 100 lines (mostly small fixes) receives a high score. They have a proven track record of safe, incremental changes.
- **Low Trust:** An author who submits their first patch containing 5,000 lines of code receives a low score. The system flags this as high-risk because the author is unproven in the ecosystem, yet is attempting a massive modification.

## C Explainability Prompt Structure

To ensure consistent explanations, we inject the XGBoost predictions into the structured template shown in Listing 2.

```
SYSTEM: Act as a Senior OpenStack Release Manager. Justify the decision to {VERDICT} this backport.
=== DECISION ===
VERDICT: {VERDICT} (Confidence: {PROB}%, Threshold: {THRESHOLD})
CATEGORY: {UI_DISPLAY_TYPE}
=== CONTEXT: HOW TO INTERPRET THE DATA ===
- Author Trust Score: Ratio of accepted backports. 0.0=New, >0.5=Trusted.
- Historical File Prob: Probability these specific files are usually backported.
- Change Entropy: Code complexity (0=Simple, >4=Complex/Scattered).
- Churn Density: Lines changed per file (High = Dense/Risky).
- Modifies DB/API: Critical risk factors.
=== FULL FEATURE VECTOR (Internal Data) ===
{FEATURE_VECTOR_JSON}
=== CHANGE ARTIFACTS ===
Commit Message: "{COMMIT_MESSAGE}"
Files Modified: {FILE_LIST_STRING}
=== INSTRUCTIONS ===
Write a professional, 2-3 sentence justification.
    (1) Analyze the Vector: Look at the "Full Feature Vector" above. Find the anomalies or strong signals.
    (2) Synthesize: Do not list the numbers. Explain their meaning.
          • Instead of "Is Pure Config is Yes", say "The change is a low-risk configuration update."
          • Instead of "Trust is 0.0", say "The author lacks a prior track record."
    (3) Explain the Verdict:
          • If REJECTED: Is it the Author? The Complexity? The specific Files?
          • If ACCEPTED: Is it the Safety (Config/Doc)? The High Trust?
RESPONSE:
```

Fig. 2. **The Prediction-Aligned Prompt Template.** The system dynamically populates the placeholders (in brackets) with the inference results and the raw feature vector before sending the request to the LLM.

## D Validation via Formal Concept Analysis (FCA)

To empirically ground the feature taxonomy and justify the dashboard layout, we employed a data-driven approach inspired by **Formal Concept Analysis (FCA)** [3]. We followed a three-step methodology to extract stable decision patterns from the historical dataset of 3,422 changes:

1. **Data Discretization (Context Creation):** Since association rule mining necessitates categorical data, continuous feature values were transformed into boolean attributes using statistical quantiles. Values $\leq Q_1$ were mapped to *Low*, values between $Q_1$ and $Q_3$ to *Medium*, and values $> Q_3$ to *High*. This transformation preserves the relative distribution of the data while enabling categorical analysis.
2. **Rule Extraction:** We utilized the **Apriori algorithm** [1] to identify frequent itemsets and association rules. To ensure the extracted rules were statistically significant, we enforced the following constraints:

- **Minimum Support = 0.02:** Focusing on patterns appearing in at least 2% of the dataset.
- **Minimum Confidence = 0.65:** Ensuring a high probability of the rule's validity.
- **Minimum Lift ≥ 1.0:** We utilized the Lift metric [2] to filter out coincidental correlations, retaining only positive dependencies.

(3) **Semantic Grouping:** The resulting dominant rules (Confidence ≈ 1.0, Lift > 1) were clustered into semantic families. As shown in Table **??**, these families correspond to the dashboard components: rules predicting rejection formed the *Risk Patterns* panel, while rules predicting acceptance formed the *Intrinsic Safety* and *Logic Safety* panels.

Table 2. Key Association Rules Derived from Historical Data. These stable patterns justify the grouping of features into the "Risk" and "Safety" dashboard panels.

| Antecedents (Contextual Signals) | Outcome | Conf. | Lift |
|---|---|---|---|
| ***Risk Patterns (Rejection Signals)*** | | | |
| `Low Success Rate` + `Low Readability` + `Is BugFix` + `High File Count` | **Rejected** | 1.00 | 31.4 |
| `Low Code Ratio` + `Med Trust` + `High File Count` + `Is BugFix` | **Rejected** | 1.00 | 30.8 |
| `High Entropy` + `Deep Directory` + `New Author` | **Rejected** | 1.00 | 28.5 |
| `CI Change` + `Low Readability` + `Modifies DB` | **Rejected** | 1.00 | 25.1 |
| ***Intrinsic Safety (Configuration & Documentation)*** | | | |
| `Config Only` + `Low Entropy` + `High Trust` + `Med Readability` | **Accepted** | 1.00 | 5.9 |
| `Low File Count` + `Config Only` + `High Success Rate` | **Accepted** | 1.00 | 5.9 |
| `Low Code Ratio` + `Config Only` + `High Trust` | **Accepted** | 1.00 | 5.9 |
| ***Logic Safety (Trusted Code Patterns)*** | | | |
| `High Trust` + `Med Readability` + `Low Entropy` + `Low File Count` | **Accepted** | 1.00 | 5.9 |
| `High Success Rate` + `Low Directory Depth` + `Med Readability` | **Accepted** | 1.00 | 5.9 |
| `High Trust` + `Med Config Ratio` + `Low Entropy` | **Accepted** | 1.00 | 5.9 |

# E Experimental Details

## E.1 Large Language Model Configuration

To evaluate the reasoning capabilities of Generative AI without the cost of fine-tuning, we benchmarked four state-of-the-art open-weights Large Language Models (LLMs). We utilized the Ollama runtime environment to execute these models locally. Based on the available hardware constraints, we utilized 4-bit quantized versions (Q4_0) for efficient inference:

- **Meta Llama 3 8B-Instruct** (4.7 GB): A highly capable general-purpose reasoning model.
- **Google Gemma 2 9B-Instruct** (5.4 GB): Google's lightweight open model optimized for logic tasks.
- **Mistral 7B-Instruct v0.3** (4.4 GB): A high-performance model known for strict instruction following.
- **Microsoft Phi-3 Mini 3.8B** (2.2 GB): A compact model optimized for high-quality reasoning at low latency.

All inference was conducted with a `temperature` of 0.0 to maximize determinism and a constrained output format set to JSON to ensure parsable results.

*E.1.1 Prompting Strategies.* We evaluated the models using two distinct prompting strategies to measure the impact of In-Context Learning (ICL):

***Zero-Shot Prompting***. In this setting, the model is provided with the role definition (Release Manager), the decision rules, and the target commit, but is given no prior examples. The prompt structure is defined as follows:

> *"You are a strict OpenStack Release Manager. Task: Decide if the following code change should be backported to a stable branch. Rules: 1. ACCEPT if it fixes a bug, crash, or build failure. 2. ACCEPT if it is a necessary maintenance update (OS compatibility). 3. REJECT if it is a new feature, refactoring, or style fix. Target Change: Subject: "subject" Message: "message" Respond strictly in JSON: "decision": "YES" or "NO" "*

***Few-Shot Prompting (In-Context Learning)***. To mitigate the models' tendency to be overly optimistic, we implemented a 6-shot prompting strategy. We extracted **six real-world examples** (3 Accepted and 3 Rejected) directly from the training partition of our dataset. These examples were selected to cover edge cases, such as dependency bumps (Maintenance) versus feature additions. The prompt structure is defined as follows:

> TRAINING DATA: 1. Subject: "ansible-lint: fix unnamed-task" -> REJECT (Style fix) 2. Subject: "Add TLS support" -> REJECT (New Feature) 3. Subject: "Correct lock path" -> REJECT (Minor follow-up) 4. Subject: "Add Debian 12 setup" -> ACCEPT (OS Compatibility) 5. Subject: "Support pagination for list API" -> ACCEPT (Fixes broken API) 6. Subject: "Ensure services stay disabled" -> ACCEPT (Fixes regression) TASK: Subject: "subject" Message: "message" Instructions: 1. Linter/Style/Typo -> NO 2. New Feature (unless OS compat) -> NO 3. Fixes Crash/Breakage/Failure -> YES Respond in JSON: "decision": "YES" or "NO" "

### E.2 Machine Learning Configuration

To evaluate the predictive power of historical and process metadata, we benchmarked five classical machine learning algorithms. Prior to training, all feature vectors were standardized using Z-score normalization (`StandardScaler`) to ensure optimal convergence for the linear and probabilistic models.

Given the class imbalance in backporting data (where accepted changes typically outnumber rejections), we implemented cost-sensitive learning across all models. We calculated a dynamic positive scale weight ($w_{pos}$), defined as the ratio of negative to positive samples in the training set:

$$w_{pos} = \frac{N_{negative}}{N_{positive}} \tag{1}$$

The models were instantiated with the following hyperparameter configurations, which were selected based on a preliminary Grid Search optimization:

- **XGBoost (Gradient Boosting):** As our primary candidate, this model was configured with 300 estimators, a maximum tree depth of 8, and a conservative learning rate of $\eta = 0.05$ to prevent overfitting. We applied stochastic regularization using a subsample ratio of 0.8 and column sampling of 0.8. The `scale_pos_weight` parameter was set to $w_{pos}$ to penalize false positives.
- **Random Forest:** An ensemble of 300 trees constrained to a maximum depth of 10. We utilized the `balanced` class weight mode to automatically adjust weights inversely proportional to class frequencies.
- **Logistic Regression:** A linear baseline configured with L2 regularization and a maximum of 2,000 iterations to ensure convergence on the high-dimensional feature set.
- **Decision Tree:** A single CART estimator capped at a depth of 10 to serve as a simple, interpretable baseline.

- **Gaussian Naive Bayes:** Included to evaluate the performance of a probabilistic classifier assuming feature independence.

## E.3 Deep Learning Model: CodeBERT

To complement metadata-based approaches, we utilized `microsoft/codebert-base`, a Transformer model pre-trained on bimodal data (Natural Language and Programming Language).

*E.3.1 Smart Input Compression.* Due to the 512-token limit of BERT architectures, raw diffs cannot be ingested directly. We developed a *Smart Diff Compressor* that parses the diff structure to discard syntactic noise (e.g., Git hashes, file indices) while strictly preserving semantic signals such as Hunk Headers (function context) and modified lines. All project-specific identifiers were masked to prevent data leakage. The inputs were concatenated as: Subject $\oplus$Message $\oplus$ Diff.

*E.3.2 Architecture.* We replaced the standard classification head with a custom 1D Convolutional Neural Network (CNN) to capture local dependencies in code. The architecture consists of:

(1) **Encoder:** CodeBERT outputs a sequence of vectors ($L \times 768$).
(2) **Convolution:** A 1D Conv layer (256 filters, kernel size 3) scans the sequence for local risk patterns.
(3) **Activation: LeakyReLU** ($\alpha = 0.1$) is applied to maintain gradient flow.
(4) **Pooling:** Global Max Pooling extracts the most significant features.
(5) **Classification:** A dense layer maps the pooled features to the binary output (Backport/Reject).

*E.3.3 Training Dynamics.* The model was trained for 100 epochs using the AdamW optimizer ($lr = 2e^{-5}$). We observed peak generalization performance between Epochs 14 and 40 (F1 $\approx 0.80$). Beyond this point, the model exhibited clear signs of overfitting, where training loss converged to near-zero ($< 0.01$) while validation accuracy declined. The best-performing checkpoint was selected for the final evaluation.

## E.4 Dual-Stream Architecture: RoBERTa + CodeBERT

A significant limitation of standard Transformer models is the 512-token context window. In complex backports, the combination of a detailed commit message and the associated code diff often exceeds this limit, leading to truncation and information loss. To mitigate this, we implemented a **Dual-Stream Multi-Modal Architecture** that processes natural language (intent) and programming language (implementation) in parallel.

*E.4.1 Architecture Design.* The system consists of two independent encoders that do not share weights:

(1) **Text Stream (Intent):** The commit subject and message are tokenized (max length 128) and passed through `roberta-base`. We utilize the `[CLS]` token embedding to capture the semantic intent of the change (e.g., distinguishing a "Bug Fix" from a "Feature Request").
(2) **Code Stream (Implementation):** The smart-compressed diff is tokenized (max length 400) and passed through `microsoft/codebert-base`. This stream focuses on identifying risky technical patterns (e.g., dependency changes or high-churn modifications) independent of the author's description.

*E.4.2 Feature Fusion.* The output vectors from both streams, $v_{text} \in \mathbb{R}^{768}$ and $v_{code} \in \mathbb{R}^{768}$, are concatenated to form a unified representation $v_{fused} \in \mathbb{R}^{1536}$. This vector is passed through a Multi-Layer Perceptron (MLP) with dropout ($p = 0.3$) for final classification.

$$y = \sigma(W_2 \cdot \mathrm{ReLU}(W_1 \cdot [v_{text} \oplus v_{code}] + b_1) + b_2) \tag{2}$$

*E.4.3 Results.* This architecture achieved an accuracy of **76.6%**, significantly outperforming the single-stream CodeBERT baseline (58% on raw data). This confirms that decoupling the representation of *what was said* (RoBERTa) from *what was done* (CodeBERT) allows the model to retain critical context that would otherwise be lost to truncation.

## References

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann, Santiago, Chile, 487–499.

[2] Sergey Brin, Rajeev Motwani, and Craig Silverstein. 1997. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 265–276.

[3] Bernhard Ganter and Rudolf Wille. 1999. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Berlin, Heidelberg.