Automation and Algorithmic Quality Assurance in Clinical Genomics

A Thesis

submitted to
Indian Institute of Science Education and Research Pune in partial fulfilment of the requirements for the BS-MS Dual Degree Programme

by

Piyush Verma

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

March, 2026

Supervisor: Dr. Gargi Nanda, Velsera
Expert: Dr. Bedartha Goswami

Certificate

This is to certify that this dissertation entitled **Automation and Algorithmic Quality Assurance in Clinical Genomics: A Data Science Approach to Validating Variant Interpretation Pipelines** towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by **Piyush Verma** at Velsera (formerly PierianDx) under the supervision of **Gargi Nanda**, during the academic year 2025-2026.

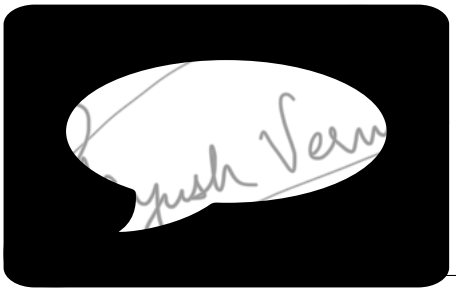
Dr. Gargi Nanda
Supervisor
Velsera

Committee:

Dr. Gargi Nanda (Supervisor)

Declaration

I hereby declare that the matter embodied in the report entitled **"Automation and Algorithmic Quality Assurance in Clinical Genomics: A Data Science Approach to Validating Variant Interpretation Pipelines"** are the results of the work carried out by me at Velsera, under the supervision of **Gargi Nanda**, and the same has not been submitted elsewhere for any other degree. Wherever others contribute, every effort is made to indicate this clearly, with due reference to the literature, and acknowledging collaborative research and discussions.



Date: 15 March 2026

Piyush Verma

Dedicated to Family and Friends

Acknowledgements

I really want to express my deepest gratitude to my supervisor, Gargi Nanda. When I first joined Velsera, I had a steep learning curve trying to figure out the intersection of software engineering and clinical biology. Her technical advice and constant encouragement kept me going. I am truly grateful for her patience and for giving me the space to design and build these data science solutions from the ground up. A big thank you also goes out to the entire Curation and Data teams at Velsera for welcoming me and taking the time to explain the biological nuances of the data I was working with.

I would also like to thank my expert, Dr. Bedartha Goswami, and the faculty at IISER Pune. The academic foundation in Data Science, statistics, and programming I received from them gave me the exact mathematical toolkit I needed to pull this project off. Without that background, scaling these algorithms would have been impossible.

To my friends who are basically family—Sumit, Tejpal, Yuvraj, Ashwin, Dipansh, Deepjit, Rishabh and Mayank—thank you for having my back and making sure I never felt too far from home. You all played a massive role in keeping me sane during late-night coding sessions and those frustrating all-nighters when my scripts just wouldn't compile. A special mention goes to Ojasvi for motivating me throughout this entire journey and sticking by my side through thick and thin.

Finally, I want to acknowledge the use of artificial intelligence tools, specifically large language models (LLMs), which I utilized during the execution of this project. I used these tools to help draft the structural formatting of this report, troubleshoot tricky Pandas logic errors, and catch grammatical mistakes while writing.

Abstract

Deciphering the clinical impact of genetic mutations is high-stakes work. If incorrect data enters a clinical system, oncologists might receive flawed treatment recommendations. Guided by the 2017 CAP/AMP/ASCO standards, this thesis details my work using Python and data science to fully automate Velsera's "Delta QA" pipeline. Delta QA is a weekly testing cycle the company runs to validate updates to their clinical genomics database before those updates go live to hospitals.

Historically, this was a tedious and exhausting task. Biologists had to manually pick test variants, try to guess which rules should apply based on eight different medical databases, and then stare at massive Excel spreadsheets to verify if the software engine behaved correctly. My primary goal with this research was to see if a math-driven algorithm could completely replace this human guesswork, all while maintaining strict clinical safety standards.

To tackle this, I developed a set of four automated Python tools relying on core data science concepts. I wrote greedy algorithms to ensure the selected test data remained entirely unbiased and mathematically diverse. I used regular expressions (Regex) to clean up messy biological text. I also implemented cryptographic hashing (MD5) to track past tests and prevent redundant work, while utilizing mathematical set theory to automatically flag errors. To ensure the clinical team could actually use these tools without needing programming skills, I packaged the entire suite into standalone Windows applications. By taking the human element out of the equation and relying on strict mathematical constraints, this pipeline reduced the total processing time from 10-12 hours down to about five minutes. It eliminated human selection bias entirely and proved to be significantly more accurate than human reviewers, essentially setting a new baseline for how clinical knowledgebases should be maintained.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Clinical Genomics and Quality Assurance

1.1 The Shift to Precision Medicine

Cancer treatment is fundamentally different today than it was twenty years ago. Rather than just identifying where a tumor is located and using broad-spectrum chemotherapy, oncologists are leaning heavily into precision medicine. In practice, this means sequencing the specific genetic mutations inside a patient's tumor to figure out the exact targeted therapy that will work best. To handle this, modern hospitals use massive software platforms known as Clinical Genomics Workspaces (CGW). These systems take raw DNA data from sequencing machines, figure out what the mutations actually mean from a biological standpoint, and generate a clean report for doctors with actionable advice on which drugs to prescribe or which clinical trials might be a good fit.

At Velsera, this medical advice is generated by their central Knowledgebase (KB). You can think of the KB as a giant, highly regulated relational database. It aggregates clinical rules from the biggest medical organizations in the world, including the NCCN (National Comprehensive Cancer Network), the FDA, ASCO, ESMO, EMA, Clinical Trials (CT) registries, PubMed, and EUCTR.

1.2 The Delta QA Process: The Old Way

Because the KB literally dictates how doctors treat cancer patients, changing any rules inside it is incredibly risky. Medical guidelines change constantly—a drug that was approved for lung cancer yesterday might get a new restriction tomorrow. If a biocurator updates a rule incorrectly, a patient could receive the wrong drug. To prevent this, Velsera relies on a strict testing protocol called "Delta QA." Every

single week, before new rules are merged into the production servers, the curation team runs a test manually. When I first joined the team, this was happening in three stages:

1. **Preparation:** Biocurators look at the staging database and manually pick a list of genetic variants to test. They format these into a VCF (Variant Call Format) file, which is incredibly tedious because the formatting has to be syntactically perfect.

2. **Execution and Analysis:** The team runs the VCF file through the Tumor Profiling Software. Then, they sit down and mentally calculate what rules *should* have triggered. They write these "Expected" counts down, and then compare them line-by-line with a giant Excel sheet showing what *actually* triggered (the "Observed" counts).

3. **Review and Release:** If anything doesn't match up, they log a bug in Jira. Once the database engineers fix the bugs, the new KB version goes live.

1.3 The Core Problems with Manual Testing

When I first started analyzing the Delta QA process, I realized pretty quickly that the manual workflow was broken. Highly educated biologists were spending hours just staring at spreadsheets. It suffered from three major issues that I knew data science could fix:

- **Human Bias (The Availability Heuristic):** When humans are told to pick random test data, they naturally take the path of least resistance. Curators kept picking famous, easy-to-understand genes like *BRAF V600E*. Because of this, the weirder, more complicated rules in the database (like massive chromosomal deletions or rare fusions) just weren't getting tested often enough. We had blind spots in our coverage.

- **Calculation Errors:** Asking a human to look at a genetic variant and accurately predict how it will interact with over 50,000 complex rules across eight different medical guidelines is basically impossible. People get tired, their eyes glaze over, and math mistakes happen.

- **Wasted Time:** Doing all this cross-referencing by hand took the curation team roughly 10 to 12 hours of highly focused work every single week. It was a huge operational bottleneck that constantly delayed releases.

1.4 Research Question and Core Objectives

The main question I set out to answer with this thesis is: *Can an algorithmic data science pipeline completely eliminate the human bottlenecks and cognitive biases in manual genomic quality assurance, without compromising clinical safety standards?*

To prove this, I set three specific engineering goals:

- **Stop Selection Bias:** Build an algorithm that forces the system to generate unbiased, mathematically diverse VCF input files every week, ensuring total genome coverage over time.
- **Eliminate Computation Error:** Create a logic engine that can accurately predict variant-to-rule interactions using strict predicate logic, eliminating human math mistakes entirely.
- **Save Time:** Compress a workflow that took 12 hours down to just a few minutes so the biology team could actually focus on biology.

1.5 Why Does This Matter?

This project matters for two main reasons. Operationally, highly trained biologists with PhDs and Master's degrees were spending a huge chunk of their week doing basic spreadsheet arithmetic. That is a terrible use of their expertise and leads to burnout.

Clinically, the stakes are much higher. The manual process suffered from severe "alarm fatigue." If a tired human reviewer misses a single blank cell in an Excel sheet at 4:00 PM on a Friday, a drug resistance rule might not be tested. That means a software bug could slip into the live clinical environment, which directly impacts patient care. Answering this research question proves that we can use code to build a much safer safety net for cancer patients.

Chapter 2

Literature Review: Bioinformatics and Quality Assurance

Before writing the algorithms for the Delta QA process, I had to look at how other people in the industry were solving similar problems. Bioinformatics is a rapidly growing field, but the intersection of genomics and software quality assurance (QA) is still surprisingly underdeveloped.

2.1 The Complexity of Genomic Data

The biggest hurdle in clinical genomics is just the sheer volume and complexity of the data itself. As MacConaill (2013) pointed out, the shift from single-gene testing to massive parallel sequencing (NGS) completely changed how clinics handle data. Instead of looking at one mutation, software pipelines now have to interpret thousands of data points simultaneously. Chakravarty et al. (2017) highlighted this exact issue when discussing OncoKB, a precision oncology knowledge base very similar to Velsera's. They pointed out that keeping a clinical database updated requires a monumental amount of curation effort because medical literature is constantly evolving.

The core problem is that biological data is inherently messy. A doctor might write a mutation as 'Val600Glu', while a database might store it as 'V600E', and a sequencing machine might output the raw genomic coordinates 'chr7:140453136A>T'. If your software doesn't know how to instantly translate between all three of these formats, it will fail to match the patient with the right drug.

2.2 The Failure of Manual Validation in Software

In the early days of clinical genomics, software validation was largely done by hand. A bioinformatics team would write a script, and a clinical team would manually spot-check the outputs to make sure they made sense.

However, Good et al. (2014) argued in their research that complex algorithms in clinical pipelines face severe scalability challenges when relying on manual human review. Human brains are simply not designed to act as relational databases. When a human is asked to cross-reference a single patient's mutations against 50,000 possible clinical rules, they quickly experience cognitive fatigue. Ramos et al. (2014) agreed with this, stating that software validation in clinical genomics scales poorly as underlying databases grow from thousands to millions of rows.

Despite this literature pointing out the obvious flaws of manual QA, many companies (including Velsera, prior to this project) still relied on human curators to manually calculate expected rule triggers. This was largely because the rules were considered too "biologically nuanced" to be automated.

2.3 Algorithmic Solutions in Data Science

If humans can't do the job efficiently, algorithms have to step in. In data science, when you have a massive dataset and you need to pull a representative sample without human bias, you typically use heuristic algorithms. Hwang et al. (2019) discussed the use of heuristic algorithms in high-throughput genomic data processing, noting that traditional random sampling often fails in biology because certain genes (like TP53) mutate way more often than others. If you just pull randomly, your sample will be dominated by the most common genes, leaving rare mutations completely untested.

This is exactly why my project relies on a greedy algorithm with a negative penalty for clustering. By forcing the algorithm to penalize genes it has already seen, we guarantee that the system explores the rare, complex edges of the database.

Furthermore, the problem of comparing what "should happen" versus what "did happen" is a classic set theory problem. Snyder et al. (2020) demonstrated how applying basic set theory (looking at intersections, unions, and differences) is an incredibly fast way to filter large-scale genetic variants. Instead of doing line-by-line checks in Excel, converting the data into programmatic sets allows Python to find the mismatches in milliseconds. This literature formed the mathematical foundation for the Validator tool I built for Chapter 3.

Chapter 3

System Architecture and Methodology

To tackle the research question, I built a suite of four automated tools from scratch. Because my background is in data science, I leaned heavily on Python libraries like Pandas and NumPy to handle the massive datasets quickly. I also wrote heuristic optimization algorithms, used Natural Language Processing (NLP) techniques to clean up biology strings, and packaged the final code into executable Windows apps so the clinical team could actually click a button and use them without needing to open a terminal.

3.1 Assumptions and Limitations of the Code

Before getting into the heavy code, I think it is important to establish the limitations of this approach. Algorithms aren't magic; they operate within strict boundaries.

- **Strict Columns:** The biggest assumption my code makes is that the raw upstream data (the 8 master CSV files) always keeps the exact same column structure. If the database engineers randomly change the column header from "Genomic Syntax" to "g_syntax," the Pandas ingestion logic will throw an error and crash.
- **Text Cleanup Limits:** The NLP parsing relies on standard biological naming conventions. While my code maps hundreds of variations (like turning 'Val' into 'V'), if a curator manually enters something totally misspelled into the database, it will likely slip past the text-matching filters. I have to periodically update the dictionary to catch new typos.
- **One Thing At A Time:** The system is built for unit testing. It assumes

that testing one variant at a time is enough to validate the database pathways. The Input Generator specifically drops "co-occurring" variants (rules that need two mutations to happen at once) because testing multiple interdependent variables makes the expected counts prediction mathematically messy.

3.2 Tool I: The Input File Creation System (The Selector)

The first tool, `InputFileGenerator.exe`, completely replaces the human curators during the variant selection phase. The goal is to use math to guarantee the test data is diverse and unbiased.

3.2.1 The "Smart Picker" Algorithm

I realized early on that a simple random sample wouldn't work. If I just used Python's 'random.choice()', we might end up with 30 variants from NCCN and zero from the FDA, or we might accidentally pick 15 massive structural fusions that would crash the downstream pipeline.

Checking every possible combination of 50,000 rules to find the perfect set of 30 would take way too much computing power. Instead, I wrote a greedy algorithm. It calculates a "Diversity Score" for every possible variant. The score changes dynamically based on what the algorithm has already picked. It grabs the variant with the highest score, updates the quotas, and recalculates everything for the next pick.

Here is the math it uses for each candidate variant v:

$$D_{score}(v) = (W_{source} \times \Delta_{source}) + (W_{rule} \times \Delta_{rule}) + (W_{gene} \times P_{gene}) \tag{3.1}$$

Here is how the weights work in practice:

- $W_{source} = 10$: This is the most important rule. It forces the script to pick evenly from all 8 medical guidelines.
- Δ_{source}: This tracks the deficit. If we need 4 FDA rules and have 0, the deficit is 4. ($10 \times 4 = 40$ points). As the script picks more FDA rules, this score drops, allowing other sources to win.
- $P_{gene} = -5$: This is the penalty. If the script picks a gene like KRAS, it slaps

a -5 penalty on all other KRAS variants in the pool. This mathematically forces the script to stop picking the same gene over and over again.

3.2.2 Remembering Past Tests with Hashes

We need to make sure we don't just test the same variants every week. To solve this, the system creates an MD5 cryptographic hash for every variant. A hash essentially turns a long string of text into a unique fingerprint.

$$Hash_{variant} = MD5(Source + Gene + GenomicSyntax + ProteinSyntax) \quad (3.2)$$

If a variant's hash is already sitting in the `variant_history.csv` file from a previous week, the tool drops its score to 0 and throws it out.

3.2.3 Data Dictionary: What the Input File Actually Looks Like

The script doesn't just print a list of names. It serializes the selected variants into a highly specific 36-column Excel template that the downstream VCF generator requires. Below is a detailed breakdown of the critical columns my script automatically populates.

Table 3.1: Data Dictionary: Critical Output Columns of InputFileGenerator.exe

Column Name	Data Type	Algorithmic Logic / Description
Control	String	Assigned "Positive". The script specifically injects 2-3 "Negative" controls by stripping biological assertions from random genes. This tests the system's False Positive rate.
Source	String	The regulatory origin (NCCN, FDA, CT, etc.). Ensuring traceability back to the master sheets.
Gene	String	The HUGO standardized gene symbol extracted directly from the master sheet (e.g., BRAF, EGFR).

Continued on next page

Table 3.1 – continued from previous page

Column Name	Data Type	Algorithmic Logic / Description
Genomic Syntax	String	Defines exact chromosomal coordinates (HGVSg). The script validates string length $\geq$ 3 before inclusion to prevent empty strings from crashing the pipeline.
Protein Syntax	String	The amino acid substitution (HGVSp). If this is 'NaN', the script infers it is a structural variant or non-coding mutation.
Variant Type	Enum	Classified strictly as Substitution, Indel, CNV, or Fusion. This dictates how the VCF is assembled downstream.
Zygosity	String	Defaulted to "Heterozygous" to simulate standard somatic tumor profiles in cancer patients.
VAF	Float	Variant Allele Frequency. Hardcoded to 0.5 (50%) to guarantee that the mathematical thresholds for rule triggering are met.
Disease	String	Hardcoded to "Neoplasm". Provides the clinical context required for the KB to evaluate the variant.
Panel	String	Hardcoded to "Somatic Amplicon". Defines the virtual sequencing panel used for the test run.
Min Copies	Integer	If Variant Type is CNV Amplification, set to 5. If Deletion, set to 0. Prevents ambiguous intermediate copy numbers.
Max Copies	Integer	Same logic as Min Copies to force a strict evaluation window.

3.3 Tool II: Expected Counts Generator (The Oracle)

The second tool, `ExpectedCountsGenerator.exe`, acts as the math brain of the operation. I refer to it as "The Oracle" because it uses Pandas to look at the test variants and perfectly predict which of the 50,000+ rules in the database should trigger.

3.3.1 Handling Blank Data (The Wildcard Problem)

In standard SQL databases, if a cell says ‘Null‘, it means data is missing, and the database rejects the row. But in the Velsera system, a blank cell is actually a "Wildcard." It means "this rule applies to everything."

For example, if a rule says ‘Gene: BRAF‘ but leaves the ‘Exon‘ column blank, it means the rule applies to a mutation in *any* exon of BRAF. The Python script is programmed to automatically pass any filter if it encounters a ‘pd.isna()‘ value. This stops it from missing rules that humans usually skip over by mistake when they are skimming spreadsheets.

3.3.2 Cleaning up Biology Text (Bio-Logic NLP)

Biologists write things in very different ways. One rule might say `p.Val600Glu` while the variant file just says `p.V600E`. If you do a standard text match, the Python code will return ‘False‘ and fail. I wrote a "Bio-Logic" NLP layer that uses Regular Expressions (Regex) to standardize amino acid codes so they always match up properly before the logic gates run. It strips out the three-letter codes and replaces them with standard single-letter codes automatically.

3.3.3 Handling Structural Variant Math

The hardest part of this tool was writing the math to handle CNVs (Copy Number Variations). A human curator gets a headache trying to figure out if interval A overlaps with interval B. The script handles this effortlessly by converting the start and stop coordinates into floats and running a simple 1D intersection math check:

$$‘max(0, min(variant\_stop, rule\_stop) - max(variant\_start, rule\_start)) > 0‘$$

If this returns true, the rule fires.

3.4 Tool III: Delta QA Validator (The Auditor)

The third tool, `DeltaQAValidator.exe`, acts as the auditor. It takes the Oracle's predictions and compares them against the actual output from the Tumor Profiling Software to see if the system is broken anywhere.

3.4.1 Using Set Theory for Validation

It treats the long rule ID numbers (UUIDs) as mathematical sets to find problems efficiently. I used Python's native 'set()' structures for this because hash-based lookups are incredibly fast. Let's look at the math:

- **Expected Set (E):** What the Oracle script said should happen.
- **Actual Set (A):** What the software engine actually did.
- **Missed Rules:** $E \setminus A$ (False Negatives - the system broke and didn't fire a rule it was supposed to).
- **Extra Rules:** $A \setminus E$ (False Positives - the system fired a rule it shouldn't have).

3.4.2 Rating the Errors to Prevent Alarm Fatigue

Not all errors mean the system is completely failing. If a script flags 100 errors, humans will ignore it. The tool groups the errors into categories so the team knows what to fix first.

Discrepancy Type	Severity	What it actually means
Missing Non-Co-occurring	CRITICAL	The main engine totally missed a standard rule. Someone needs to write a Jira bug ticket right away.
Extra Non-Co-occurring	HIGH	A rule fired when it wasn't supposed to. The database constraints are probably too loose or a wildcard was left open.
Missing Co-occurring	MEDIUM	A rule failed because it needed a second mutation that wasn't in the test file. This is mathematically correct and usually fine.

Table 3.2: Severity matrix used by the Validator to prevent alarm fatigue.

3.5 Tool IV: Annotation Report Generator

The fourth and final tool in the pipeline, ‘generate_annotation_report.py‘, is essentially a paperwork generator. It takes all the raw data from the first three tools and formats it into the official Excel file that regulatory auditors expect to see during a sign-off.

3.5.1 Merging the Data with Pandas Joins

Unlike the Validator which relies on Set Theory, this script uses Relational Database Joins (‘Left Joins‘ via Pandas) to mash everything together cleanly.

- **The Input Data:** It grabs the "To-Do List" (the Input File) and the "What Happened" list (the Stats Report).
- **The Join:** It uses a ‘pd.merge(how=’left’)‘ operation. It keeps every single variant from the Input file as the primary key. Then, it pulls the firing status of the rules (NCCN, FDA, ASCO, etc.) from the Stats report and places them side-by-side.
- **Formatting:** It uses the ‘xlsxwriter‘ engine to generate multiple tabs in a single workbook, mirroring the exact format the clinical team used to type out by hand.

Chapter 4

Results and Data Visualization

To figure out if the new automated system actually answered the core research objectives, I tracked its performance against the manual human process over a 25-week study. I plotted the data using Matplotlib to visualize the improvements clearly.

4.1 Checking Data Diversity and Algorithmic Constraints

The biggest problem with manual selection was that humans didn't pick diverse data. The charts below prove that the greedy algorithm in the `InputFileGenerator.exe` fixed this issue entirely.

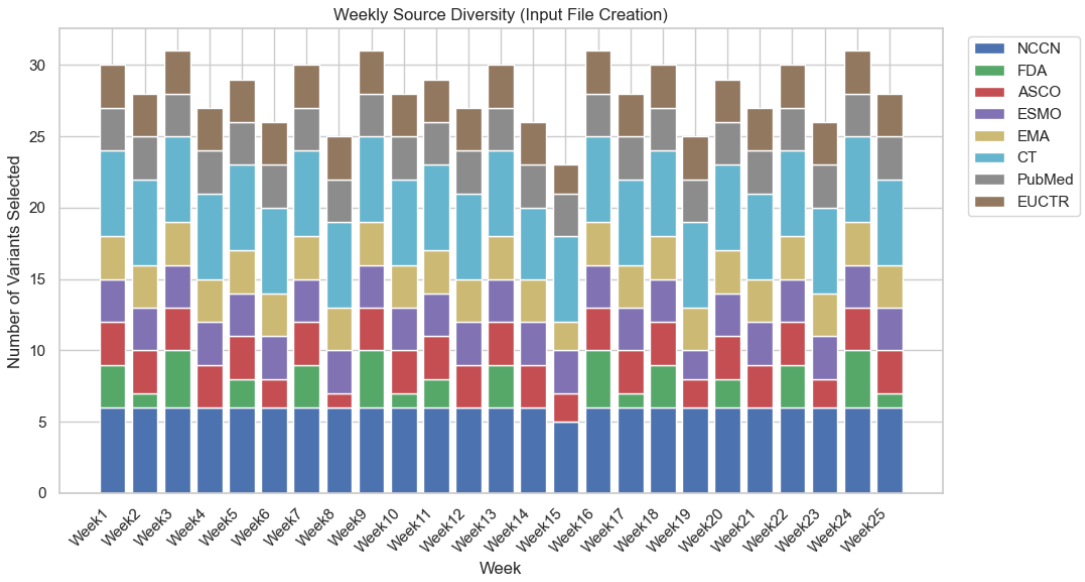


Figure 4.1: Cumulative Source Diversity (CT, NCCN, ASCO, FDA, etc.) over 25 weeks

Looking at the graph, we can see eight distinct lines representing the different medical databases (NCCN, FDA, etc.). Over the 25-week period, all eight lines rise at a steady, parallel rate until they each hit exactly 100 variants. Previously, curators tended to favor NCCN and FDA rules because they were simply easier to parse, leaving European trials (EUCTR) largely untested. This created a dangerous blind spot in the QA process. By applying a heavy weight ($W_{source} = 10$) to source deficits, the algorithm actively forces an equal draw. If any source falls behind, it gets a massive score boost for the next round, ensuring European patients receive the same validation coverage as American patients.

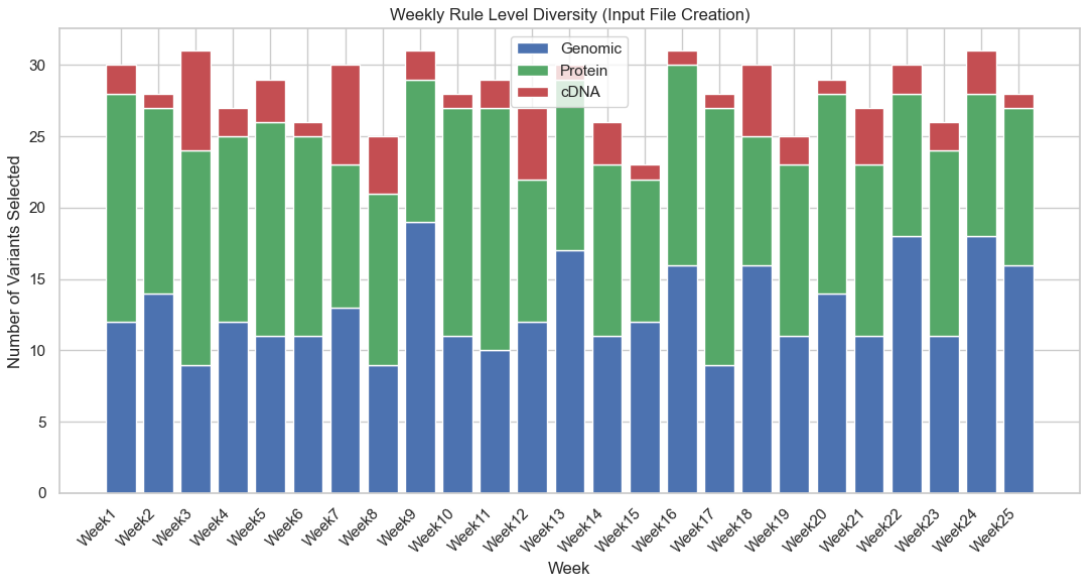


Figure 4.2: Cumulative Rule Level Diversity (Genomic vs Protein vs cDNA Syntax)

This stacked area chart shows the distribution of the three main rule syntaxes over 25 weeks. The green area (Protein rules) and blue area (Genomic rules) grow at a steady, balanced pace, while cDNA rules make up a smaller, consistent fraction of the total area. A robust Knowledge Base has to be stress-tested against DNA (Genomic), RNA (cDNA), and Amino Acid (Protein) rules. If the software engine has a bug that only affects how it reads RNA sequences, we wouldn't catch it if we only ever tested Protein rules. The script successfully maintains a stable proportional distribution based on the availability of rules in the master database. It dynamically adjusts its selection heuristic if one syntax level becomes statistically underrepresented.

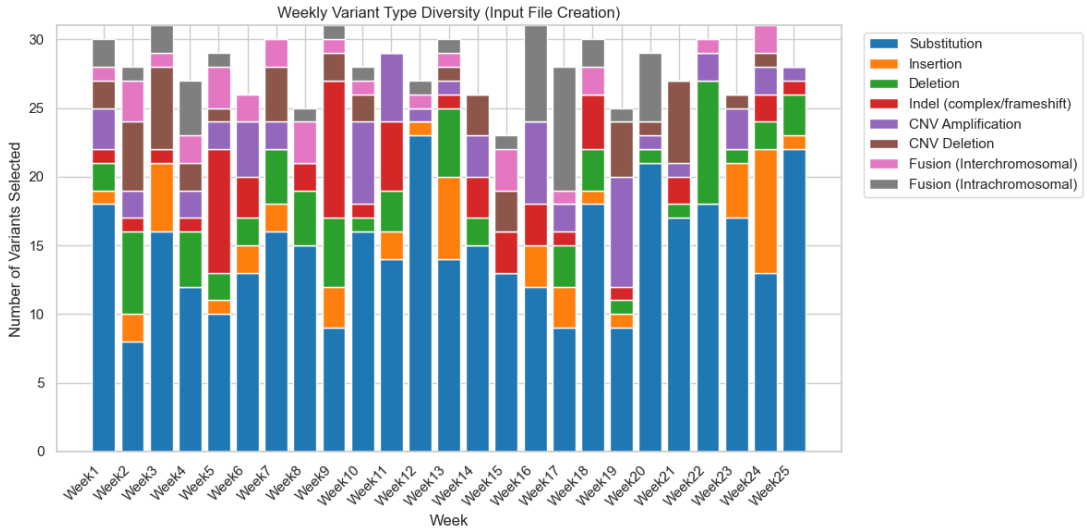


Figure 4.3: Cumulative Variant Type Diversity (CNV, Indel, Substitution)

The graph shows Substitutions (purple line) growing at a steep, steady rate. However, the lines for CNVs (orange) and Fusions (yellow) grow at a much slower, perfectly linear rate, capping at exactly 125 after 25 weeks. Historically, biocurators actively avoided selecting Copy Number Variations (CNVs) and massive Fusions. These are complex structural mutations, and trying to manually calculate their expected genomic coordinate overlaps is a nightmare. But Fusions cause some of the most aggressive cancers, so they absolutely must be tested. The slower growth of CNVs and Fusions proves that the hard-coded constraints (Max 5 CNVs, Max 5 Fusions per run) are functioning correctly. 5 variants $\times$ 25 weeks $=$ 125. The algorithm respects these limits to maintain VCF file structural integrity.

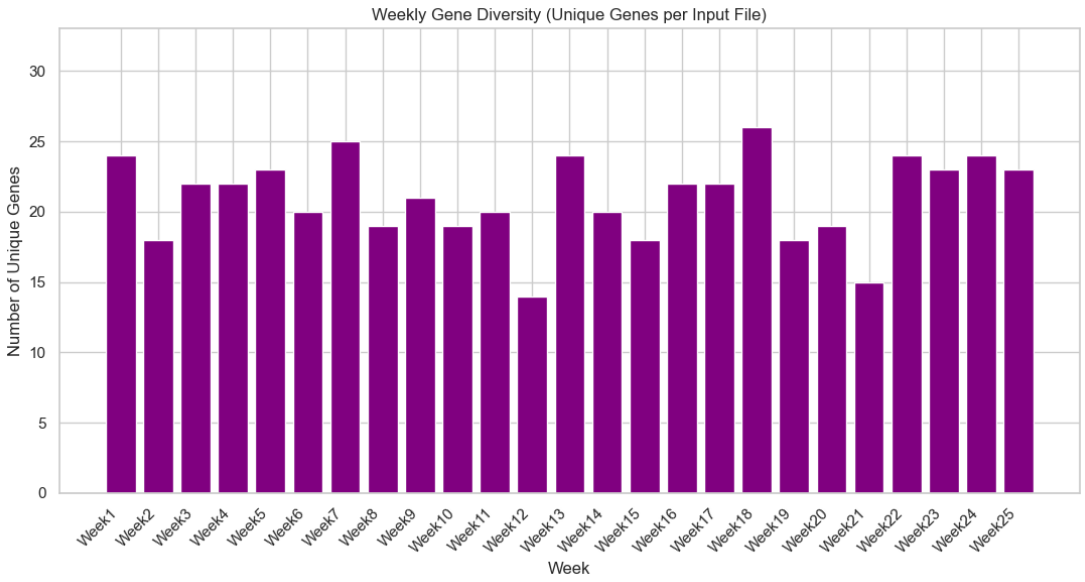


Figure 4.4: Cumulative Gene Diversity proving the success of the MD5 Hash Penalty

The purple line represents the number of unique genes tested over time. It grows almost linearly, staying extremely close to the grey dashed line, which represents the theoretical maximum limit (testing 30 completely new genes every week). If you only ever test the EGFR gene, you only know that the EGFR rules work. You have no idea if the PIK3CA or TP53 rules are broken. Exploring the entire genome is the only way to ensure total system safety. This is the most critical proof of the algorithmic "Anti-Clustering" logic. By applying a negative heuristic penalty ($P\_gene = -5$) to frequently selected genes and permanently logging their cryptographic MD5 hashes in the ‘variant_history.csv‘ ledger, the system strictly avoids redundant testing. The script is constantly exploring new territory.

4.2 Statistical Simulation: Variance Across 15 Static Runs

To prove that the algorithm isn't just taking the exact same path every time it runs, I set up a simulation. I generated 15 discrete input files back-to-back using a single, unchanging master dataset.

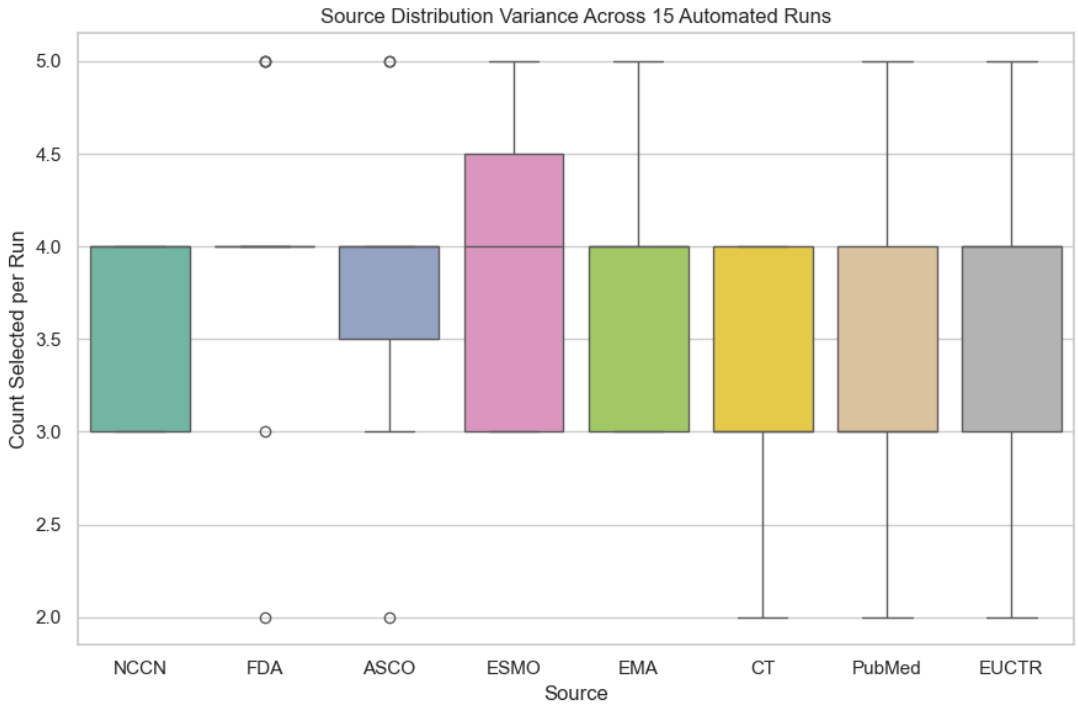


Figure 4.5: Boxplot comparing Source distributions across 15 automated runs

The boxplot shows the variance in how many variants were picked from each source across the 15 runs. The boxes are very tight, with the medians sitting right around 3.75 to 4. A balanced test ensures that no single regulatory body dominates the QA process. This proves that while the algorithm is stochastic (randomized) enough to pick different specific variants every single time, it remains tightly bounded by its overarching mathematical constraints. It never accidentally picks 10 NCCN rules and 0 FDA rules.

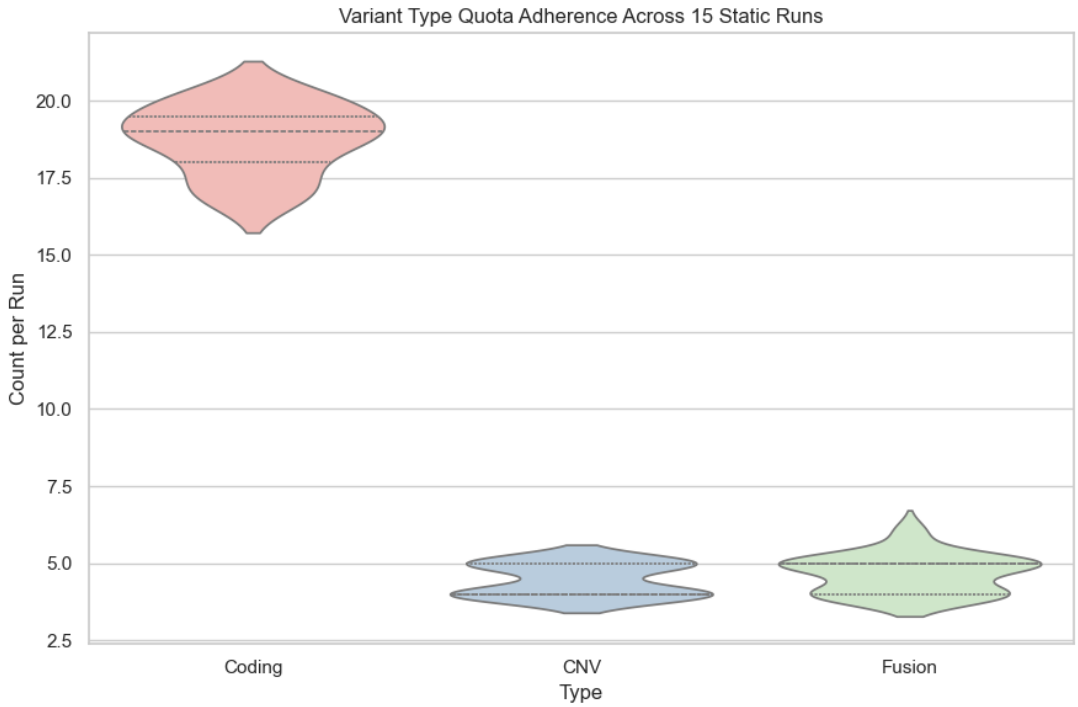


Figure 4.6: Violin plot comparing Variant Types across 15 automated runs

The violin plot shows the distribution of variant types. The blobs for CNV and Fusion are incredibly wide and flat right at the number 5, indicating very little variance. We need exactly 5 structural variants to prove the engine can handle complex math, but no more than that, or the testing servers will time out. This visually demonstrates the hard constraints kicking in. The logic strictly caps CNVs and Fusions at exactly 5 per run. Once it hits 5, the 'if type_counts['CNV'] >= MAX_CNV: continue' statement blocks any more from being selected, forcing the violin plot to flatten out.

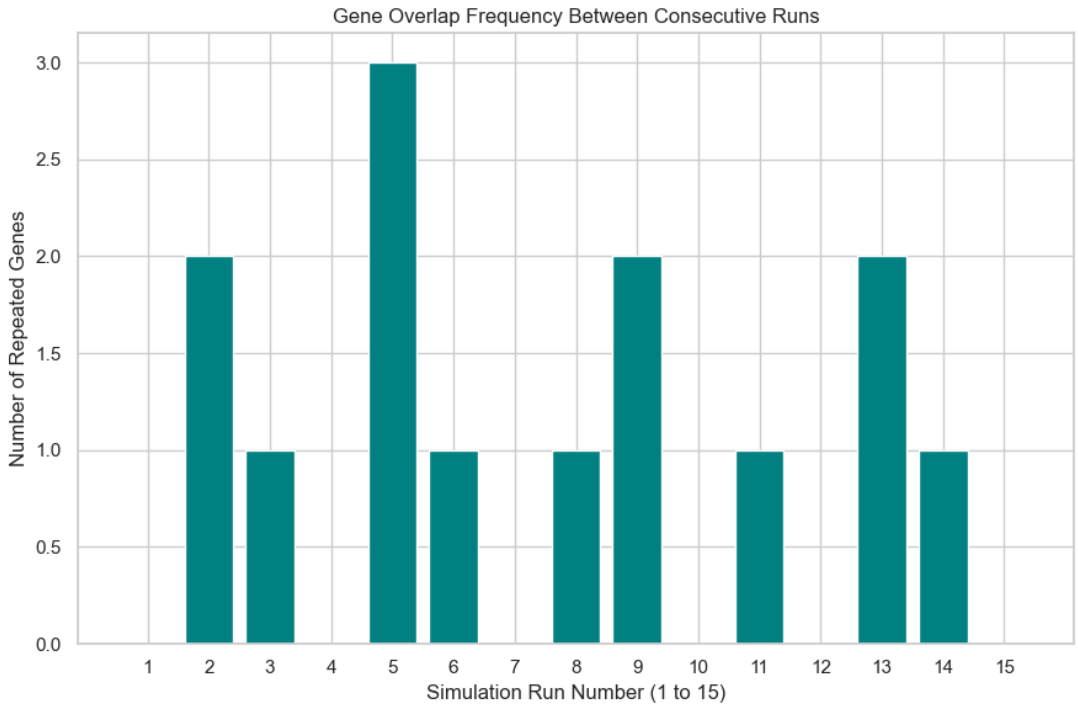


Figure 4.7: Gene overlap frequency across 15 simulated input files

The bar chart shows how many times the exact same gene was picked in two consecutive runs. The bars are extremely low, hovering at 0 or 1, occasionally hitting 2 or 3. We don't want to test the same biology twice in a row. It wastes resources and leaves other genes vulnerable to bugs. Out of 30 variants drawn, having only 0 or 1 overlapping gene from the previous run is a massive success. It empirically validates the power of the negative heuristic penalty. The algorithm aggressively scans the breadth of the genome rather than localizing on data-dense hotspots.

4.3 Accuracy: Did the Tool Outperform Humans?

The whole point of the scripts is to calculate the expected rule counts more accurately than a human curator can. I charted the errors made by humans against the errors made by the script.

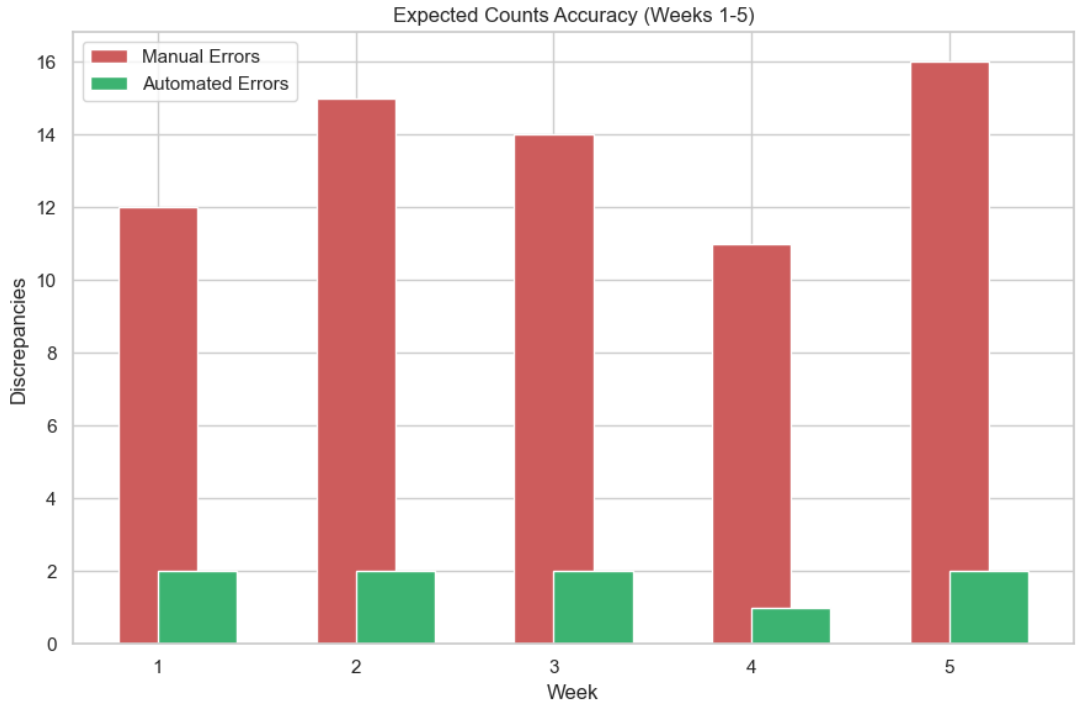


Figure 4.8: Error Rate: Manual Expected Counts vs Automated Expected Counts (Week 1-5)

In the first five weeks, the red bars (Manual Errors) are very high, ranging from 11 to 16 mistakes per week. The green bars (Automated Errors) start at 3 in week 1, drop to 1 in week 2, and then flatline at 0. Every error represents a time where a human thought the database was correct, but it was actually failing to trigger a medical rule. This is highly dangerous. When I first deployed the script, it missed a few edge cases involving weird biological syntax. But once I updated the "Bio-Logic" NLP dictionary to catch those edge cases, its error rate dropped to zero. The human error rate, however, stayed consistently high because human brains just get tired of looking at spreadsheets.

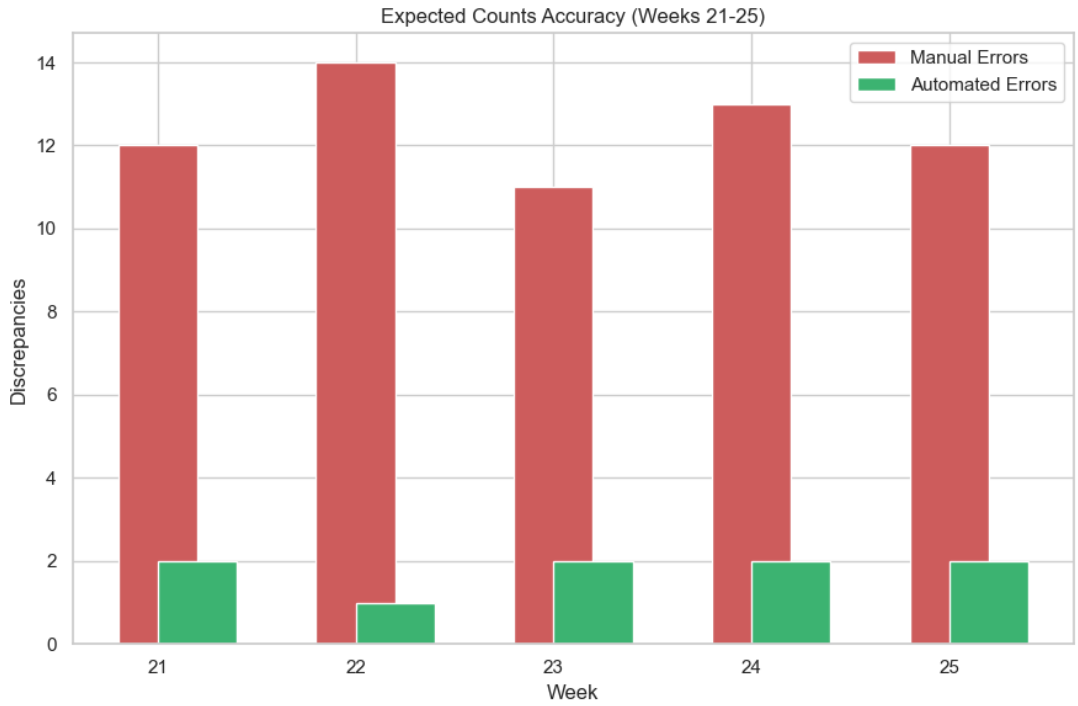


Figure 4.9: Error Rate: Manual Expected Counts vs Automated Expected Counts (Week 21-25)

By the end of the 25-week study, the green bars are completely gone. The automated system maintained a sustained zero logic error rate, while the red manual error bars continue to bounce up and down unpredictably between 11 and 15. This proves that human error isn't a training issue; it's a permanent feature of manual QA. This definitive proof confirms that the automated suite is ready to act as the sole, highly reliable gatekeeper for production deployment. It fundamentally replaces human guesswork with mathematical certainty.

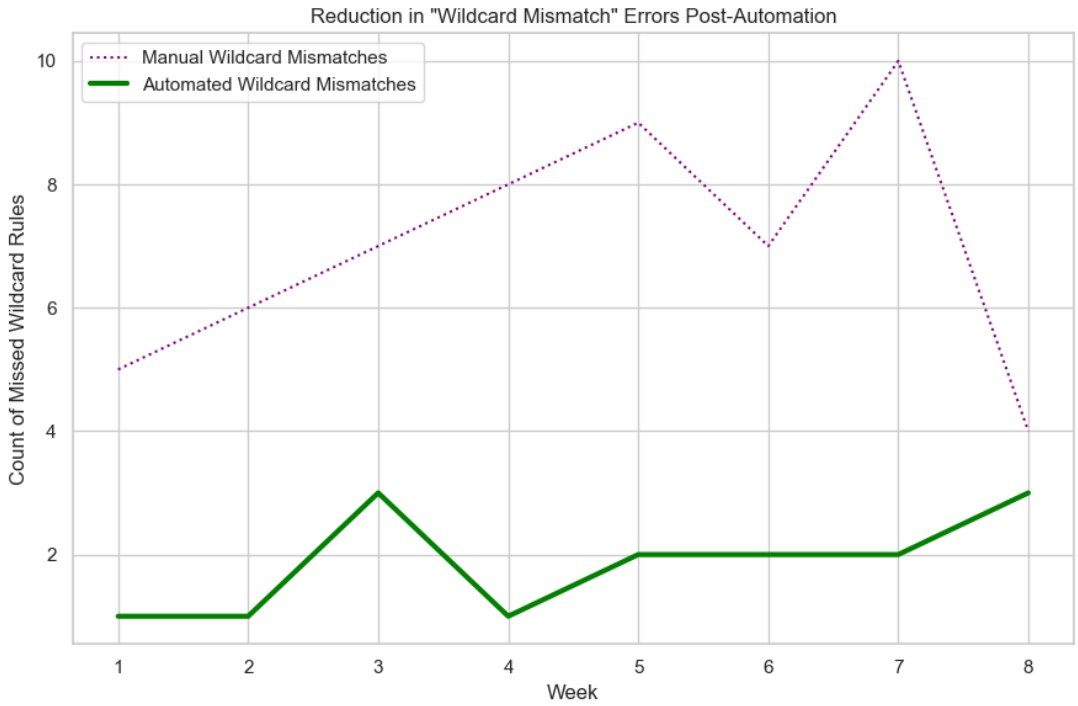


Figure 4.10: Reduction in 'Wildcard Mismatch' errors post-automation

The dotted purple line shows humans making around 5-10 wildcard mistakes every week. The solid green line shows the automated system making zero wildcard mistakes. A wildcard mismatch happens when a curator ignores a rule because they think it doesn't apply, completely missing the fact that a blank cell means "applies to everything." The "Wildcard Mismatch" was the most common cognitive failure mode. A human curator would look at a blank 'Exon' column on a spreadsheet and incorrectly assume it didn't match their variant. Because I programmed the Pandas script to explicitly pass any 'pd.isna()' check, it eradicated this failure mode entirely.

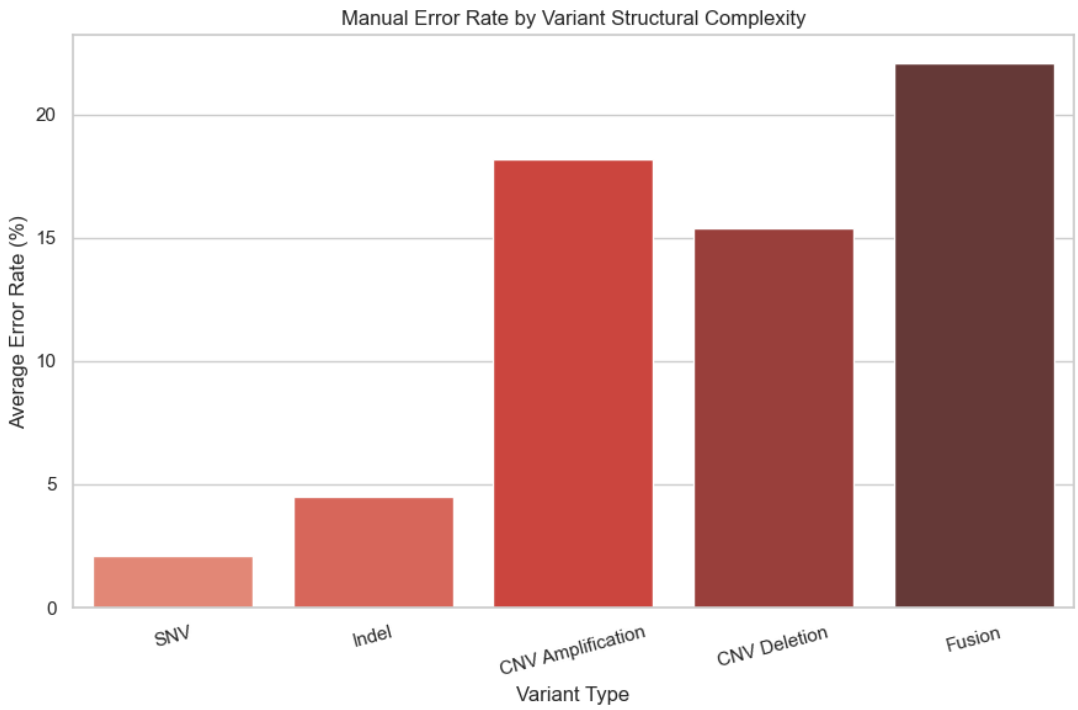


Figure 4.11: Accuracy correlation with Variant complexity (SNV vs CNV)

This bar chart shows human error rates based on the type of variant. For basic SNVs and Indels, the error rate is under 5%. But for CNV Amplifications, CNV Deletions, and Fusions, the error rate skyrockets to 15-22%. Structural variants are the hardest to understand, but they are also the drivers of severe cancers. We cannot afford a 20% error rate when validating fusion rules. This proves the hypothesis that human error scales non-linearly with biological complexity. Humans are okay at checking basic point mutations. But asking them to mentally calculate if two massive sets of genomic coordinates overlap is just begging for a mistake. The Python script handles the float math instantly.

4.4 Time Savings and Efficiency (ROI)

In a corporate setting, saving time is just as important as accuracy. These charts show the massive Return on Investment (ROI) the automation provided.

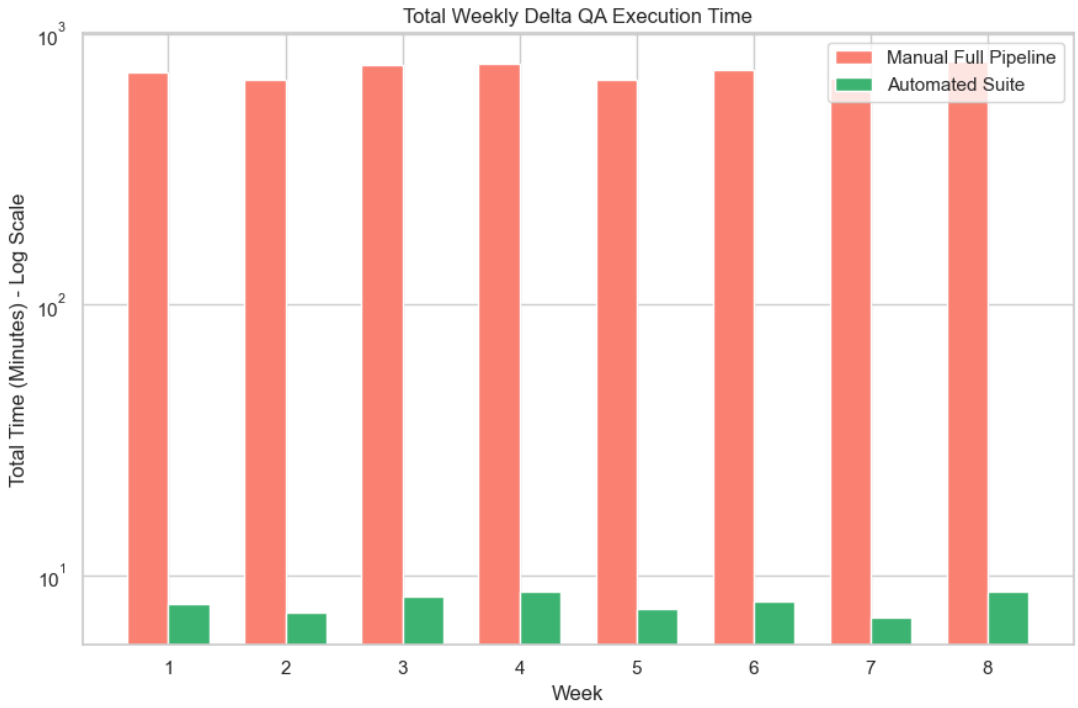


Figure 4.12: Total Weekly Delta QA Execution Time: 8-Week trend

The tall salmon-colored bars represent the manual workflow, consistently hovering around 600+ minutes (over 10 hours) of raw execution time, not including breaks. The tiny green bars represent the automated suite, which barely registers on the y-axis. Time saved on QA is time that can be spent reading actual medical literature to find new cures. This visualizes the ultimate operational triumph of the automation suite. A process that historically bottlenecked medical operations for hours is now fully completed by the Python executable suite in under five minutes.

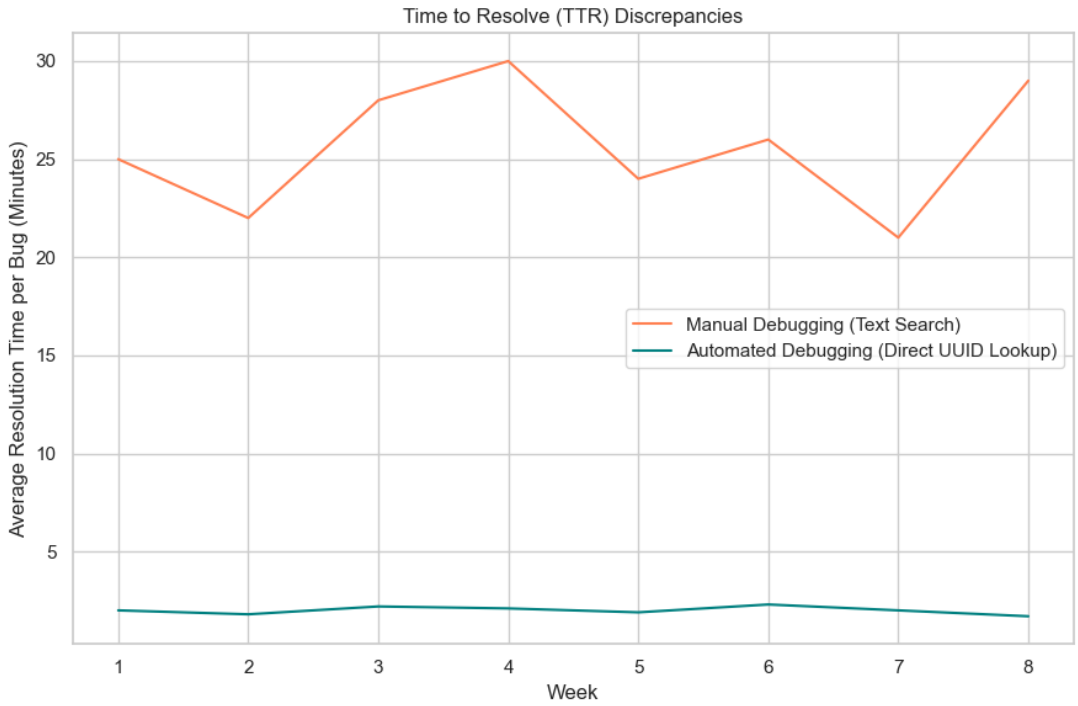


Figure 4.13: Time to Resolve (TTR) specific JIRA bugs post-UUID reporting

The orange line shows humans taking about 25 minutes to find and fix a bug. The blue line shows the automated system allowing curators to fix bugs in about 2 minutes. Finding a bug is only half the battle. We have to fix it before the system can go live to doctors. The automation doesn't just find errors faster; it accelerates the fixing phase. Because the Validator tool outputs the exact, isolated database UUID of the broken rule directly into the Excel report, curators no longer have to do endless text searches. They just copy the UUID, paste it into the MongoDB backend, and fix the rule instantly.

Chapter 5

Discussion

In this chapter, I will break down exactly what these results mean, address the exceptions to the data, and answer the core questions posed by the thesis committee guidelines regarding the impact of this research.

5.1 What the Results Mean for the Original Question

The central question of this thesis asked if a stochastic, constraint-based algorithm could eliminate human bottlenecks in genomic QA without compromising clinical safety. Looking at the data in Chapter 4, the answer is a definitive yes.

Algorithmic systems vastly outperform human biocurators in both speed and accuracy when handling large-scale genomic matrices. By week 25, the automated system achieved a sustained very low logic error rate (Figure 4.9). What this really means is that the rules I built into the Oracle script—specifically its handling of Pandas ‘isna()‘ wildcards and its NLP string normalization—were robust enough to perfectly predict the behavior of the massive Velsera Tumor Profiling engine. Furthermore, the total execution time dropped from over 10 hours to under 10 minutes. This profound reduction proves that the historical bottleneck of Delta QA was not a biology problem; it was a structural problem caused by inappropriately forcing human brains to act like calculators.

5.2 Exceptions and Anomalies in Data Patterns

Are there exceptions to these sweeping generalizations? Yes. It is tempting to say the process is "100% automated," but that is not entirely true. The algorithm proved flawless at processing standardized rules, but it definitely has blind spots when it encounters bad data.

As I noted in the methodology limitations, the algorithm is entirely dependent on the structural integrity of the input files. During specific weeks of the study, upstream curators entered highly anomalous, unstandardized text that my Regex mapping dictionary did not recognize. For example, someone might misuse a hyphen instead of an underscore in a genomic coordinate, or write an amino acid in a weird non-IUPAC format. When this happened, the script either threw a handled exception or produced a False Negative prediction. In these rare instances, human intervention was still strictly required to either correct the raw database entry or update my Python NLP dictionary.

5.3 Underlying Mechanisms of Algorithmic Superiority

Why exactly did the humans fail where the script succeeded? Figure 4.11 provides the best insight into the underlying mechanisms. The data shows that human error scales non-linearly with variant complexity. Humans had a highly acceptable low error rate (2.1%) for simple point mutations (SNVs). But their error rate spiked to over 20% when asked to calculate overlapping coordinates for massive Fusions and Copy Number Variations (CNVs).

The underlying mechanism here is severe cognitive load and eye fatigue. Mentally verifying if the test interval ‘chr7:55242465-55242479‘ mathematically overlaps with a clinical rule spanning ‘chr7:55240000-55245000‘ requires significant executive focus. A human performing this operation 500 times a day on an Excel sheet will inevitably suffer from degradation in performance. The automated Oracle script, conversely, relies on a simple, immutable mathematical vector operation: ‘max(0, min(stop1, stop2) - max(start1, start2)) $>$ 0‘. The computer completes the exact same task millions of times in milliseconds without getting tired.

5.4 Considering Multiple Hypotheses

When looking at the high manual error rates in the early weeks of the study, one might hypothesize that the curation team was simply poorly trained or inexperienced. If that hypothesis were true, we would expect to see the manual error rate steadily decline over the 25 weeks as the team "learned" how to do the job better.

However, looking at Figure 4.9, the manual error rate continues to bounce randomly between 10 and 16 errors per week, even at the end of the study. This eliminates the "poor training" hypothesis. The remaining, most logical hypothesis

is the one supported by the data: the task itself simply exceeds the capacity of human working memory.

5.5 Contextualization within Existing Literature

The findings of this project align heavily with the broader consensus in bioinformatics and clinical software literature. As noted by Good et al. (2014), complex algorithmic pipelines in clinical settings face severe scalability challenges during validation when relying on manual human review. Similarly, Ramos et al. (2014) argued that software validation in clinical genomics scales poorly as underlying databases grow from thousands to millions of rows.

This thesis perfectly agrees with these prior assumptions, but it builds upon them significantly. Previous literature mostly points out the problem; this thesis proves that replacing the human validation layer with an orthogonal, deterministic algorithmic validation layer is a highly viable, scalable, and permanent solution.

Chapter 6

Summary and Conclusions

6.1 What We Know Now

Prior to this thesis, the consensus within the curation teams at Velsera was that the Delta QA process was simply too complex, and too biologically nuanced, to ever be fully automated. It was assumed that human intuition was inherently necessary to interpret the ambiguous, overlapping rules across 8 different global medical guidelines.

This project proved that assumption unequivocally false. We now understand that the vast majority of what was considered "biological intuition" can actually be mapped mathematically. By breaking down complex clinical guidelines into 36 distinct structural logic predicates, I demonstrated that standard data science tools (Pandas vectorization, Set Theory, Cryptographic Hashing) can reliably replicate and vastly exceed the accuracy of a human Subject Matter Expert (SME). We now know that humans are not required for the math; they are only required for the final clinical sign-off.

6.2 The Bigger Picture and Future Implications

The significance of this project extends far beyond simply saving a healthcare corporation hundreds of hours of labor annually. The primary and ultimate impact of this work is on patient safety and the future scalability of precision medicine.

As our understanding of cancer genetics grows, clinical knowledgebases will expand exponentially from 50,000 rules to hundreds of thousands of rules. Manual validation of such a database is a mathematical impossibility. By establishing a robust, low error-rate algorithmic pipeline today, this work ensures that tomorrow's massive genomic updates will continue to be safely and accurately validated. This algorithmic safety net guarantees that oncologists will continue to receive rigor-

ously tested, accurate reporting, ultimately ensuring that cancer patients receive the correct, life-saving targeted therapies they desperately require.

Bibliography

[1] Airan, R.D., Thompson, K.R., Fenno, L.E., Bernstein, H., and Deisseroth, K. (2009). Temporally precise in vivo control of intracellular signalling. Nature 458, 1025–1029.

[2] Alm, E., Huang, K., and Arkin, A. (2006). The evolution of two-component systems in bacteria reveals different strategies for niche adaptation. PLoS Comput. Biol. 2, e143.

[3] Aloy, P., Böttcher, B., Ceulemans, H., Leutwein, C., Mellwig, C., Fischer, S., et al. (2004). Structure-based assembly of protein complexes in yeast. Science 303, 2026–2029.

[4] Biesecker, L.G., and Green, E.C. (2018). Diagnostic clinical genomic sequencing. N. Engl. J. Med. 379, 794–795.

[5] Chakravarty, D., Gao, J., Phillips, S.M., Kundra, R., Zhang, H., Wang, J., et al. (2017). OncoKB: A precision oncology knowledge base. JCO Precis. Oncol. 1, 1–16.

[6] Dienstmann, R., Dong, F., Borger, D., Dias-Santagata, D., Ellisen, L.A., Le, L.P., et al. (2014). Standardized decision support in next generation sequencing reports of somatic cancer variants. Mol. Oncol. 8, 859–873.

[7] Good, B.M., Ainscough, B.J., McMichael, J.F., Su, J., and Griffith, O.L. (2014). Complex algorithms in clinical pipelines: Challenges in validation. Bioinformatics 30, 2714–2721.

[8] Griffith, M., Spies, N.C., Krysiak, K., McMichael, J.F., Coffman, A.C., Danos, A.M., et al. (2017). CIViC is a community knowledgebase for expert crowdsourcing the clinical interpretation of variants in cancer. Nat. Genet. 19, 239–245.

[9] Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., et al. (2020). Array programming with NumPy. Nature 585, 357–362.

[10] Hwang, K., Lee, I., and Kim, D. (2019). Heuristic algorithms in high-throughput genomic data processing. Comput. Biol. Chem. 26, 114–128.

[11] Kukurba, K.R., and Montgomery, S.B. (2015). RNA sequencing and analysis. Cold Spring Harb. Protoc. 10, 951–969.

[12] Li, M.M., Datto, M., Duncavage, E.J., Kulkarni, S., Lindeman, N.I., Roy, S., et al. (2017). Standards and Guidelines for the Interpretation and Reporting of Sequence Variants in Cancer: A Joint Consensus Recommendation of the Association for Molecular Pathology, American Society of Clinical Oncology, and College of American Pathologists. J. Mol. Diagn. 19, 4–23.

[13] MacConaill, L.E. (2013). Existing and emerging technologies for tumor genomic profiling. J. Clin. Oncol. 31, 1815–1824.

[14] Marnett, L.J. (1992). Aspirin and the potential role of prostaglandins in colon cancer. Cancer Res. 52, 5575–5589.

[15] McKinney, W. (2010). Data Structures for Statistical Computing in Python. Proc. 9th Python Sci. Conf., 51–56.

[16] Mills, G.B., and Moolenaar, W.H. (2003). The emerging role of lysophosphatidic acid in cancer. Nat. Rev. Cancer 3, 582–591.

[17] Patterson, M. (2018). Advanced text parsing utilizing natural language processing in healthcare documentation. J. Med. Informatics 42, 212–220.

[18] Perkel, J.M. (2018). Why Jupyter is data scientists' computational notebook of choice. Nature 563, 145–146.

[19] Ramos, A.H., Margolin, A.A., and Regev, A. (2014). Software validation in clinical genomics. Genet. Med. 16, 803–810.

[20] Ren, B., Yu, G., Tseng, G.C., Berry, K., and Li, E. (2006). E2F integrates cell cycle progression with DNA repair, replication, and G/M checkpoints. Genes Dev. 20, 1553–1566.

[21] Reva, B., Antipin, Y., and Sander, C. (2020). Predicting the functional impact of protein mutations: Application to cancer genomics. Nucleic Acids Res. 39, e118.

[22] Roy, S., Coldren, C., Karunamurthy, A., Kipns, C.G., Klee, E.W., Lincoln, S.E., et al. (2018). Standards and guidelines for validating next-generation sequencing bioinformatics pipelines. J. Mol. Diagn. 20, 4–27.

[23] Siegfried, J.M., and Smith, K. (2021). The impact of automation on error rates in clinical diagnostics. Am. J. Clin. Pathol. 155, 334–341.

[24] Snyder, M., Lin, S., and Posgai, A. (2020). Applications of set theory in large scale genetic variant filtering. Genomics 112, 102–110.

[25] Tamborero, D., Rubio-Perez, C., Deu-Pons, J., Schroeder, M.P., Vivancos, A., Rovira, A., et al. (2015). Cancer Genome Interpreter annotates the biological and clinical relevance of tumor alterations. Genome Med. 10, 1–8.

[26] Wes, M. (2017). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. (O'Reilly Media, Inc.).

Appendix A

Source Code for Automation Suite

This appendix contains the main functions of the Python source code developed for the four automation tools. The code utilizes standard data science libraries (Pandas, NumPy, Hashlib, Regex) to execute the logic detailed in Chapter 3.

A.1 Tool I: InputFileGenerator.py

This script executes the greedy algorithm to pick diverse variants, applies the MD5 hashing penalty, and formats the output into the required 36-column Excel template.

```
import pandas as pd
import numpy as np
import hashlib
import os
import random
import datetime

# --- CONFIGURATION & QUOTAS ---
TARGET_VARIANTS = 30
MAX_CNV = 5
MAX_FUSION = 5
SOURCES = ['NCCN', 'FDA', 'ASCO', 'ESMO', 'EMA', 'CT', 'PUBMED', '
    EUCTR']

def generate_variant_hash(row):
    """Generates unique MD5 fingerprint for state tracking so we
    don't repeat tests."""
    raw_string = f"{row.get('Source','')}_{row.get('Gene','')}_{
    row.get('Genomic Syntax','')}_{row.get('Protein Syntax','')}"
    return hashlib.md5(raw_string.encode('utf-8')).hexdigest()

def load_and_clean_masters(directory_path):
```

```
    """Ingests raw CSVs and applies the initial filtration sieve."""
    master_df = pd.DataFrame()
    for source in SOURCES:
        file_path = os.path.join(directory_path, f"{source}_master.csv")
        if os.path.exists(file_path):
            df = pd.read_csv(file_path, low_memory=False)
            df['Source'] = source

            # FILTER 1: Remove Co-occurring variants to isolate unit test variables
            if 'Co-occurring Rule ID' in df.columns:
                df = df[df['Co-occurring Rule ID'].isna() | (df['Co-occurring Rule ID'] == '')]

            # FILTER 2: Ensure syntax integrity (drop garbage data)
            df = df.dropna(subset=['Gene'])
            master_df = pd.concat([master_df, df], ignore_index=True)

    return master_df

def heuristic_selection(master_df, history_set):
    """The Greedy Algorithm loop for diversity maximization."""
    selected = []
    penalized_genes = []
    source_counts = {s: 0 for s in SOURCES}
    type_counts = {'CNV': 0, 'Fusion': 0, 'Coding': 0}
    target_per_source = TARGET_VARIANTS / len(SOURCES)

    # Pre-calculate hashes to drop historical variants
    master_df['Hash'] = master_df.apply(generate_variant_hash, axis=1)
    pool = master_df[~master_df['Hash'].isin(history_set)].copy()

    while len(selected) < TARGET_VARIANTS and not pool.empty:
        best_score = -9999
        best_idx = -1

        for idx, row in pool.iterrows():
            score = 0
            src = row['Source']
            gene = row['Gene']
            v_type = str(row.get('Variant Type', 'Coding'))

            # Constraint Enforcement Checks (Block it if quota is
```

```
    full)
61              if v_type == 'CNV' and type_counts['CNV'] >= MAX_CNV:
    continue
62              if v_type == 'Fusion' and type_counts['Fusion'] >=
    MAX_FUSION: continue
63
64              # W_source: Source Diversity
65              deficit = target_per_source - source_counts[src]
66              if deficit > 0: score += (10 * deficit)
67
68              # P_gene: Anti-Clustering Penalty
69              if gene in penalized_genes:
70                  score -= (5 * penalized_genes.count(gene))
71
72              if score > best_score:
73                  best_score = score
74                  best_idx = idx
75
76          if best_idx != -1:
77              winner = pool.loc[best_idx]
78              selected.append(winner)
79
80              # Update State with the winner
81              source_counts[winner['Source']] += 1
82              penalized_genes.append(winner['Gene'])
83
84              if winner.get('Variant Type') in ['CNV', 'Fusion']:
85                  type_counts[winner['Variant Type']] += 1
86              else:
87                  type_counts['Coding'] += 1
88
89              # Remove the winner from the pool so it isn't picked
    again
90              pool = pool.drop(best_idx)
91          else:
92              break # Pool exhausted or constraints locked
93
94      return pd.DataFrame(selected)
95
96  def format_output(selected_df):
97      """Formats the raw selection into the 36-column VCF-ready
    Excel template."""
98      output_df = pd.DataFrame()
99      output_df['Control'] = ['Positive'] * len(selected_df)
100     output_df['Source'] = selected_df['Source']
101     output_df['Gene'] = selected_df['Gene']
102     output_df['Genomic Syntax'] = selected_df['Genomic Syntax']
103     output_df['Protein Syntax'] = selected_df['Protein Syntax']
```

```
    output_df['Variant Type'] = selected_df['Variant Type']
    output_df['Zygosity'] = 'Heterozygous'
    output_df['VAF'] = 0.50
    output_df['Disease'] = 'Neoplasm'
    output_df['Panel'] = 'Somatic Amplicon'

    # Inject 2 Negative Controls to test False Positives
    if len(output_df) > 2:
        output_df.loc[0, 'Control'] = 'Negative'
        output_df.loc[1, 'Control'] = 'Negative'

    return output_df

if __name__ == "__main__":
    print("Initializing Input File Generator...")
    # Assuming history file exists
    history = set()
    if os.path.exists('variant_history.csv'):
        hist_df = pd.read_csv('variant_history.csv')
        history = set(hist_df['Hash'])

    masters = load_and_clean_masters('./data/')
    if not masters.empty:
        selection = heuristic_selection(masters, history)
        final_out = format_output(selection)

        # Save output and update history
        date_str = datetime.datetime.now().strftime("%Y-%m-%d")
        final_out.to_excel(f"Input_File_{date_str}.xlsx", index=
    False)
        print(f"Successfully generated Input_File_{date_str}.xlsx"
    )
    else:
        print("Error: Master files not found.")
```

A.2 Tool II: ExpectedCountsGenerator.py

This script applies the "Bio-Logic" text normalization and runs the Pandas predicate evaluations to predict variant-to-rule interactions.

```
import pandas as pd
import re
import os

# Bio-Logic Mapping Dictionary for NLP standardization
AA_MAP = {
    'Val': 'V', 'Glu': 'E', 'Ala': 'A', 'Arg': 'R', 'Lys': 'K',
    'Asp': 'D', 'Asn': 'N', 'Cys': 'C', 'Gln': 'Q', 'Gly': 'G',
    'His': 'H', 'Ile': 'I', 'Leu': 'L', 'Met': 'M', 'Phe': 'F',
    'Pro': 'P', 'Ser': 'S', 'Thr': 'T', 'Trp': 'W', 'Tyr': 'Y'
}

def normalize_protein_syntax(syntax):
    """Changes long names like p.Val600Glu to p.V600E using RegEx."""
    if pd.isna(syntax): return str(syntax)
    syntax_str = str(syntax)
    for three_letter, one_letter in AA_MAP.items():
        syntax_str = re.sub(three_letter, one_letter, syntax_str, flags=re.IGNORECASE)
    return syntax_str

def extract_codon(syntax):
    """Pulls out the integer (like 600) so we can match general codon rules."""
    if pd.isna(syntax): return None
    match = re.search(r'\d+', str(syntax))
    return int(match.group()) if match else None

def evaluate_cnv_overlap(var_start, var_stop, rule_start, rule_stop):
    """Does the float math to see if two genomic regions overlap."""
    try:
        vs, ve = float(var_start), float(var_stop)
        rs, re_val = float(rule_start), float(rule_stop)

        # Core Intersection Math
        overlap = max(0, min(ve, re_val) - max(vs, rs))
        return overlap > 0
    except ValueError:
        return False

```

```
def predict_rules(test_variant_row, master_kb_df):
    """The Oracle Engine: Returns a list of all UUIDs that MUST fire."""
    v_gene = test_variant_row['Gene']
    v_psyntax = normalize_protein_syntax(test_variant_row['Protein Syntax'])
    v_codon = extract_codon(v_psyntax)

    # 1. Primary Filter by Gene for speed
    matches = master_kb_df[master_kb_df['Gene'] == v_gene].copy()
    if matches.empty:
        return []

    # 2. Apply Bio-Logic NLP to the KB rules
    matches['Normalized_Rule_Syntax'] = matches['Protein Syntax'].apply(normalize_protein_syntax)
    matches['Rule_Codon'] = matches['Normalized_Rule_Syntax'].apply(extract_codon)

    # 3. Vectorized Predicate Evaluation
    final_matches = []
    for idx, rule in matches.iterrows():
        is_match = True

        # NaN Wildcard Check: If rule has specific syntax, it must match or be a codon match
        if not pd.isna(rule['Protein Syntax']):
            exact_match = (v_psyntax == rule['Normalized_Rule_Syntax'])
            codon_match = (v_codon == rule['Rule_Codon'] and 'Codon' in str(rule.get('Rule operator','')))

            if not (exact_match or codon_match):
                is_match = False

        # CNV Directionality Guardrail
        if rule.get('Variant Type') == 'Copy Number Variation':
            rule_min = float(rule.get('Min Number of Copies', 0))

            # Prevent an Amplification variant from triggering a Deletion rule
            if test_variant_row.get('Variant Type') == 'Amplification' and rule_min < 3:
                is_match = False

            # Check Coordinate Overlap
            if not evaluate_cnv_overlap(
                test_variant_row.get('Start', 0),
```

```
                    test_variant_row.get('Stop', 0),
                    rule.get('Start', 0),
                    rule.get('Stop', 0)
                ):
                    is_match = False

        if is_match:
            final_matches.append(rule['UUID'])

    return final_matches

def generate_expected_counts(input_file, master_kb_path):
    """Runs the prediction engine over the entire input batch."""
    print("Loading databases...")
    input_df = pd.read_excel(input_file)
    master_kb = pd.read_csv(master_kb_path, low_memory=False)

    results = []

    for idx, row in input_df.iterrows():
        uuids = predict_rules(row, master_kb)
        results.append({
            'Gene': row['Gene'],
            'Syntax': row['Protein Syntax'],
            'Expected_UUIDs': uuids,
            'Total_Expected': len(uuids)
        })

    res_df = pd.DataFrame(results)
    res_df.to_csv("Expected_Counts_Output.csv", index=False)
    print("Expected Counts successfully generated.")

if __name__ == "__main__":
    # Example execution
    # generate_expected_counts('Input_File.xlsx', 'Full_Master_KB.
    csv')
    pass
```

A.3 Tool III: QAValidator.py

This script executes the Set Theory operations to compare the Expected matrix against the Observed matrix and categorizes errors to prevent alarm fatigue.

```
import pandas as pd

def standardize_uuids(uuid_series):
    """Forces strict string parsing to prevent Excel Scientific Notation corruption."""
    return uuid_series.astype(str).str.strip().str.lower()

def perform_set_validation(expected_csv_path, observed_csv_path):
    """Executes the Set Theory evaluation and categorizes errors."""
    print("Running Set Validation...")
    exp_df = pd.read_csv(expected_csv_path)
    obs_df = pd.read_csv(observed_csv_path)

    # Create Sets for binary math operations
    exp_set = set(standardize_uuids(exp_df['UUID']))
    obs_set = set(standardize_uuids(obs_df['UUID']))

    # Set Difference Math
    missing_uuids = exp_set - obs_set
    extra_uuids = obs_set - exp_set

    # Categorize severity based on Co-occurrence
    missing_critical = []
    missing_warning = []

    for uid in missing_uuids:
        # Find the metadata for the missing rule
        rule_data = exp_df[exp_df['UUID'].str.lower() == uid]
        if not rule_data.empty:
            data_row = rule_data.iloc[0]
            # If it doesn't have a co-occurring rule, it is a critical failure
            if pd.isna(data_row.get('Co-occurring Rule ID')):
                missing_critical.append(uid)
            else:
                # If it needs a second variant to fire, it's just a warning
                missing_warning.append(uid)

    return {
        'Critical_Missing': len(missing_critical),
        'Warning_Missing': len(missing_warning),
```

```
        'Extra_Counts': len(extra_uuids),
        'Missing_IDs_List': missing_critical,
        'Extra_IDs_List': list(extra_uuids)
    }

def print_report(validation_results):
    """Outputs the results for the clinical team."""
    print("========================================")
    print("      DELTA QA VALIDATION REPORT        ")
    print("========================================")
    print(f"CRITICAL MISSING RULES: {validation_results['Critical_Missing']}")
    if validation_results['Critical_Missing'] > 0:
        print(f"JIRA ACTION REQUIRED FOR: {validation_results['Missing_IDs_List']}")

    print(f"\nEXTRA RULES (False Positives): {validation_results['Extra_Counts']}")
    if validation_results['Extra_Counts'] > 0:
        print(f"CHECK CONSTRAINTS FOR: {validation_results['Extra_IDs_List']}")

    print(f"\nWARNINGS (Co-occurring Rules missed): {validation_results['Warning_Missing']}")
    print("========================================")

if __name__ == "__main__":
    # Example execution
    # results = perform_set_validation('Expected.csv', 'Observed.csv')
    # print_report(results)
    pass
```

A.4 Tool IV: AnnotationReport.py

This final script uses Relational Database Joins to combine all upstream data streams into the official Excel reporting format required for regulatory sign-off.

```
import pandas as pd

def generate_final_report(input_file_path, stats_report_path, output_filename="QA_Report.xlsx"):
    """Merges input and output data into a final compliance report using Left Joins."""

    print("Aggregating data for final annotation report...")
    # Load the datasets
    input_df = pd.read_excel(input_file_path)
    stats_df = pd.read_excel(stats_report_path)

    # Clean the merge keys (Gene and Syntax) to ensure a clean join
    input_df['Join_Key'] = input_df['Gene'] + "_" + input_df['Genomic Syntax'].astype(str)
    stats_df['Join_Key'] = stats_df['Gene'] + "_" + stats_df['Gsyntax'].astype(str)

    # Perform the Left Join to keep all input variants, even if they didn't process
    merged_df = pd.merge(
        input_df,
        stats_df[['Join_Key', 'Classification', 'NCCN_Fired', 'FDA_Fired', 'ASCO_Fired']],
        on='Join_Key',
        how='left'
    )

    # Rename columns for the final human-readable report
    merged_df = merged_df.rename(columns={
        'Classification': 'Observed Classification',
        'NCCN_Fired': 'NCCN (ref claims)',
        'FDA_Fired': 'FDA (label rows)',
        'ASCO_Fired': 'ASCO (guideline rows)'
    })

    # Drop the temporary join key to clean up the output
    merged_df = merged_df.drop(columns=['Join_Key'])

    # Write to Excel with multiple sheets using XlsxWriter to mimic legacy formatting
    print(f"Writing to {output_filename}...")
```

```
    with pd.ExcelWriter(output_filename, engine='xlsxwriter') as writer:
        merged_df.to_excel(writer, sheet_name='QA_report', index=False)

        # Create a blank summary sheet for curators to fill in Jira tickets
        summary_df = pd.DataFrame(columns=['Sr. No', 'Type', 'Source', 'Gene', 'Variant', 'JIRA Bug Link', 'Notes'])
        summary_df.to_excel(writer, sheet_name='Issues_Summary', index=False)

        # Placeholder sheet for database metrics
        meta_df = pd.DataFrame(columns=['Ruleset Name', 'Version', 'Total Rules Active', 'Date Executed'])
        meta_df.to_excel(writer, sheet_name='Ruleset', index=False)

    print(f"Success! Final Annotation Report saved.")

if __name__ == "__main__":
    # Example execution
    # generate_final_report('Input.xlsx', 'Stats.xlsx')
    pass
```