Democracy Developers
Colorado IRV RLA
Implementation Report

Date: June 3, 2025
Version 2.0 (Previous versions were titled "Implementation Plan")

Authors: Dr. Michelle Blom and Associate Prof. Vanessa Teague

Contents

Edit history:

Version 1.0 Dec 1 2023

Version 1.1 March 1, 2024. Main changes:

- Reorganization of the assertion-generation and assertion-download API in the Raire service. IRV-relevant database interactions are now all inside the raire-service rather than colorado-rla.

- Reorganization of Democracy Developers' new modules into separate packages in *colorado-rla*.

Version 1.2 May 20, 2024. Main changes:

- Remove the `generate-and-get` endpoint from the Raire service.

- Add a `get-assertions-csv` endpoint to the Raire service.

- Update the data structure returned by `get-assertions-json` to incorporate risks in each assertion's state data.

- Change the name of `get-assertions` to `get-assertions-json`.

- Remove (`CountyID, ContestID`) pairs from the GenerateAssertionRequest. Describe how these will now be calculated inside the Raire service.

- Add `WRONG_CANDIDATE_NAMES` to description of error handling.

Version 1.2.1 Main changes:

- Correct computation of sample size estimate ()

- Add extra parameters in GetAssertionsRequest and GenerateAssertions request

- Add extra data (derived from request) to assertion export, csv and json (winner and total auditable ballots).

- Add NO_VOTES_IN_DATABASE to description of error handling.

Version 2.0 (this version) June 3, 2025. Main changes:

- Change title and content to describe completed work rather than a plan.

- Add raire-service hello endpoint.

- More details of testing, especially workflow testing.

Chapter 1

Introduction

This document describes extensions to *colorado-rla* to incorporate Instant Runoff Voting (IRV). Background theory on auditing IRV is contained in the *Guide to Raire* Part 1 and Part 2. Here we describe the implementation details, including both the new raire service and the extensions to existing *colorado-rla* software. The Development was conducted by Democracy Developers Ltd, with lead developers Dr. Michelle Blom, Dr. Andrew Conway and A/Prof. Vanessa Teague.

The information flow corresponds to the plan in our *Guide To RAIRE*: the RAIRE microservice generates assertions, which are audited using *colorado-rla*. Figure 1.1 provides a sequence diagram of the RLA process from election data upload to report generation. It shows the new components that have been added to support IRV audits and the existing components that required modification. The main auditing engine of *colorado-rla* remains unchanged. Conducting an IRV audit requires three pieces of software:

- the raire assertion generation engine generates assertions for each IRV contest,

- the Raire service receives requests from *colorado-rla* and communicates between *colorado-rla*, the database, and the raire assertion generation engine.

- the updated colorado-rla receives vote records from counties and conducts the audit, using the assertions it requests from the raire-service.

Instructions for compiling, installing and running these software components are in the raire-service README and colorado-rla developer instructions.

Chapter 2 describes the process of running an audit, emphasizing the changes to the user experience to incorporate IRV. This is intended for people who will use the system. Chapter 3 and Chapter 4 are intended for developers interested in the implementation details—they describe *colorado-rla*'s existing system and the changes we made to incorporate IRV. The final chapter describes testing and validation.
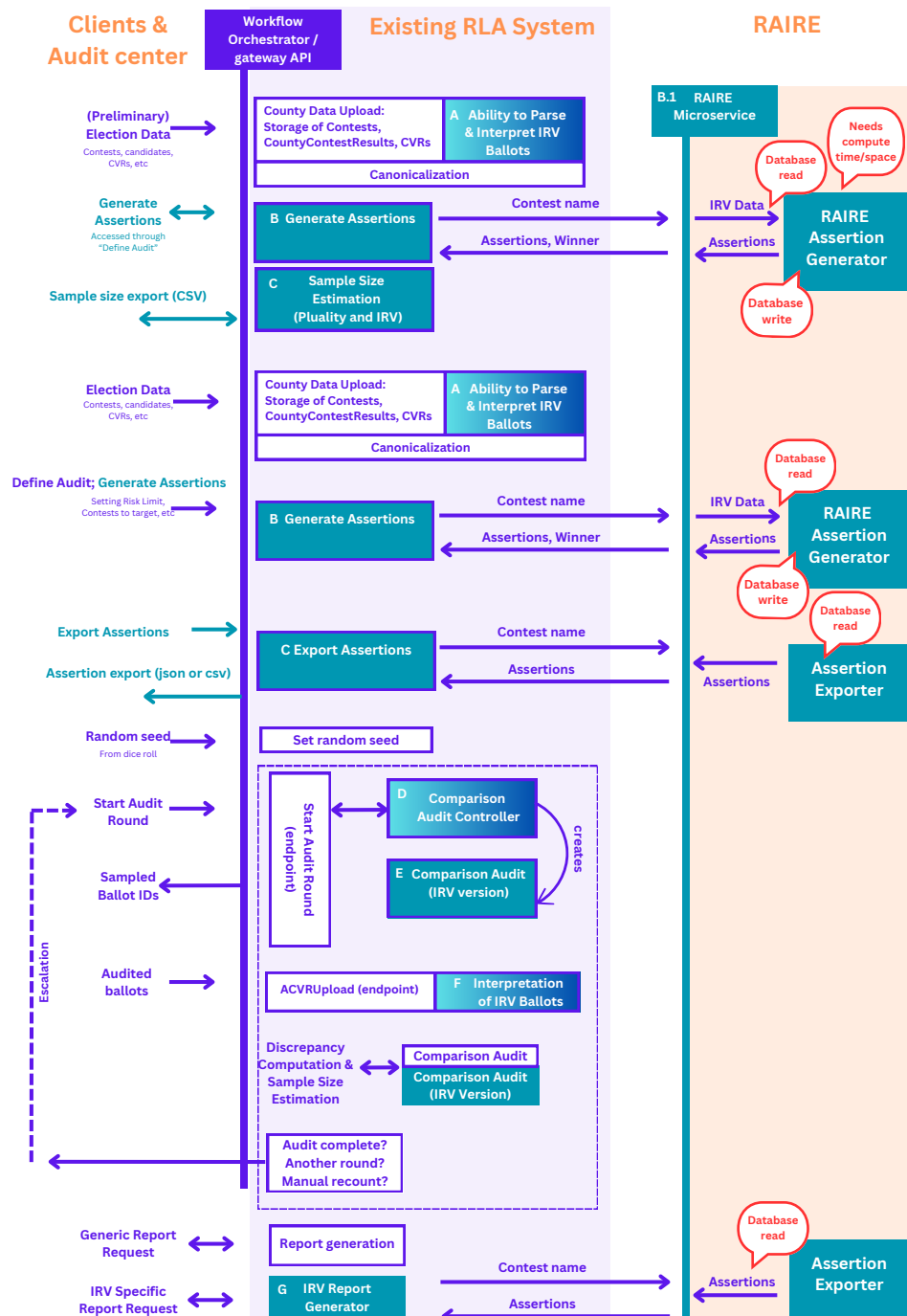
Figure 1.1: High level sequence diagram capturing the RLA process from election data upload to report generation, identifying new components required for IRV audits and instances where existing components required modifications. Purple denotes the original colorado-rla system, green the new components implemented for RAIRE IRV RLAs, and shaded gradients the existing code in colorado-rla that was modified to incorporate IRV contests.

Chapter 2

User Perspective: running an audit

This chapter steps through the audit process explaining how it has changed for IRV.

2.1   Setting up the audit data

The process for county administrators to upload their CVRs has not changed. A CVR may now include a mix of plurality and IRV contests. STV contests (ranked-choice voting with multiple winners) may also be uploaded, but are ignored. IRV (and STV) CVRs may include invalid ranks, for example votes that duplicate candidates, or repeat or skip ranks. These should be uploaded unaltered, and will be correctly interpreted by *colorado-rla*,[1] which produces a report on how it interpreted them.

2.2   Canonicalization

Canonicalization works in exactly the same way for IRV contests as it does for plurality contests—there was no need to edit the existing code in colorado-rla.

2.3   Generating and examining assertions

The process of defining an audit now includes the step of generating assertions for IRV contests. This takes place after the canonicalization of candidate and contest names, but before contests are selected for audit.

The RAIRE microservice is called from a new endpoint:

---

[1]It follows Colorado Election Rules 8 CCR 1505-1 Rule 26, with an extra skip-clearing step at the end.

Figure 2.1: The assertion generation screen with example data. When assertion generation succeeds, the IRV winner is displayed; when it fails, an explanation (such as Tied Winners) is given.

New Endpoint: 1. *Generate Assertions. This iterates through all IRV contests, uses the RAIRE microservice to generate assertions, and store them in the database.*

Colorado-rla then shows the user the results of assertion generation. This will usually consist of a success and a winner—it is important to check that these match the expected winners from vote tabulation. If assertion generation failed for any reason, such as a tie or a technical error, this failure will also be displayed. An example assertion generation summary, showing 3 successes and one failure due to a tied contest, is shown in Figure 2.1.

Sometimes assertion generation may take some time, though usually it should be very fast. If it does not complete within the default time limit (5 seconds), the user is prompted to re-run assertion generation with a longer time limit (in seconds). Fill the preferred value into the entry box and click "Generate Assertions" again. This can be repeated multiple times with progressively longer timeouts.

These assertions can be retrieved by clicking 'Export Assertions', either CSV or JSON. This calls the new *Get Assertions* endpoint.

New Endpoint: 2. *Get Assertions. This downloads assertions, in either csv or json form as requested, including some extra data such as the diluted margin and current risk.*

County	Contest Name	Contest Type	Ballots Cast	Total ballots	Diluted Margin	Sample Size
Adams	Adams COUNTY COMMISSIONER DISTRICT 3	PLURALITY	46	284	0.15	43
Alamosa	Alamosa COUNTY COMMISSIONER DISTRICT 3	PLURALITY	45	284	0.13	48
Arapahoe	Tied_IRV	IRV	10	284	0	0
Archuleta	Kempsey Mayoral	IRV	17585	17585	0.022	283
Boulder	Boulder County Ballot Issue 1A	PLURALITY	118525	118669	0.6	11
Boulder	Boulder County Ballot Issue 1B	PLURALITY	118525	118669	0.42	15
Boulder	Boulder Valley School District RE-2 Director District A (4 Years)	PLURALITY	74164	118669	0.082	76
Boulder	Boulder Valley School District RE-2 Director District C (4 Years)	PLURALITY	74164	118669	0.17	36
Boulder	Boulder Valley School District RE-2 Director District D (4 Years)	PLURALITY	74164	118669	0.066	95
Boulder	Boulder Valley School District RE-2 Director District G (4 Years)	PLURALITY	74164	118669	0.17	37
Boulder	City of Boulder Ballot Issue 2A	PLURALITY	34149	118669	0.14	46
Boulder	City of Boulder Ballot Question 2B	PLURALITY	34149	118669	0.18	34
Boulder	City of Boulder Ballot Question 302	PLURALITY	34149	118669	0.061	102
Boulder	City of Boulder Council	PLURALITY	34149	118669	0.00039	16061
Boulder	City of Boulder Mayoral Candidates	IRV	34149	118669	0.01	607
Boulder	City of Lafayette City Council	PLURALITY	11270	118669	0.00094	6656
Boulder	City of Longmont - City Council Member At-Large	PLURALITY	33460	118669	0.047	132

Figure 2.2: An example sample-size-estimate csv. 'Ballots Cast' is the number of votes cast in the contest; 'Total ballots' is the size of the universe for auditing. A sample size of 0 indicates that the contest is not auditable.

Both files describe basic information about all assertions for each contest, such as the winner and diluted margin. The CSV contains some extra summary data, while the json form should be validated in the RAIRE assertion visualizer. See Subsection 2.10.2 for more detail.

2.4 Estimating sample sizes

A third new *colorado-rla* endpoint handles sample-size estimation.

For the new sample size feature, counties will have to do an election-night preliminary upload. Closer to the audit, they will be able to upload a replacement. This did not require any new functionality.

Sample size estimation is expected to run on preliminary results very soon after election day, but of course could be run at any time after all counties for a given contest have uploaded their CVRs, and assertions have been generated for IRV contests.

New Endpoint: 3. *Estimate Sample Sizes* *This produces a CSV file containing the estimated sample size for each contest, derived from the diluted margin. Diluted margins are calculated for IRV contests using their assertions, and for plurality contests directly using the candidate tallies.*

This produces a csv file with sample size estimates for all (plurality and IRV) contests. An example is shown in Figure 2.2.

2.5   Redoing Canonicalization and assertion generation given final CVRs

CVRs may change between preliminary sample-size estimation and audit day. Canonicalization *must* be re-run in this case—*colorado-rla* already has logic for detecting when CVRs have changed and prompting for a re-run of the required canonicalization steps.

It is also advisable to run asssertion-generation again after the canonicalization re-run. Although RAIRE is not hypersensitive to small changes in the input data, different assertions might be generated if the contest is close or the changes are significant. The new assertions will simply replace the ones that were generated from preliminary data.

Alternatively, the database can be completely flushed between preliminary and final CVR upload. Canonicalization and assertion-generation then run again on the new data.

2.6   Starting the Audit

Subsequent audit steps, including choosing targeted contests, setting the seed and starting the audit, are unchanged. The software can target IRV contests at either state or county level.

A plurality contest is considered to be non-auditable if its diluted margin is zero. An IRV contest is non-auditable if RAIRE could not generate assertions for it. This may arise because the contest was a tie, or because assertions could not be generated for the contest within a given time limit. An IRV contest that is not auditable will have its target checkbox disabled.

The existing code for inputting a random seed and outputting a list of sampled ballots will be used for both plurality and IRV. The different types of contest will simply use the same samples, just as colorado-rla already uses the same samples when multiple plurality elections are targeted in one county. The IRV ones fit in with the existing sample selection.

Very little about the actual *audit* process needs to change for IRV.

2.7   Conducting the audit

2.7.1   Auditing IRV ballots

As the audit board enters data from the sampled ballots, they are uploaded to *colorado-rla*. For IRV votes, the audit board will now enter the full list of ranks. If the audit board sees an invalid IRV ballot,

they should enter exactly what they see. The system interprets this according to Colorado's ballot interpretation rules—see Section 4.5 for details. An example of the IRV ballot audit step is shown in Figure 2.3. The reaudit step looks similar.

2.7.2 CDOS view of the ongoing audit

As audit ballots are uploaded, some progress statistics are visible to the state admin dashboard.

Computation of Discrepancies   Discrepancy computation for a contest is currently performed within its associated *ComparisonAudit*. Our *IRVComparisonAudit* subclass will perform discrepancy computation for an IRV contest, implementing Definition 4 based on Table A.1 of the *Guide to RAIRE*.

When discrepancies need to be summarized for display, they are aggregated as described in Subsection 2.10.4.

2.8 Risk calculation

The functionality present in *Audit.math* for computing the current level of risk attained in an audit does not need to change. We describe the functionality in this part of the existing system in Section 3.2.

For an IRV contest, this functionality will be used to compute a risk for each of its assertions, with the largest of these risks representing the overall risk attained in the audit for the contest.

Note that risk computations are currently only being performed during the report generation process.

2.9 Finishing the Audit, starting another round or requiring Manual Recount

The main audit process now needs to decide what to do next: conclude the audit, escalate to another round, or perform a manual recount of some contests. This obviously depends on which round has passed and whether the audit has achieved the risk limit, for all the contests under audit.

As ballots are audited, and discrepancies identified, the estimated sample sizes required for each contest may increase. Once the number of ballots audited and the (optimistic) sample size for a contest match, its audit is complete.

This code did not change for IRV. The decisions that apply to IRV now incorporate the IRV-specific sample size estimations.

For each ballot contest:

Select exactly the same voting choices as the voter marked on the paper ballot you are examining.

Example 1: If the voter marked three candidates on their ballot in this contest, select the exact same three candidates below.

Example 2: If the voter did not vote for any of the candidates or choices in this contest, select "Blank vote – no mark"

Kempsey Mayoral
IRV

Candidate	Ranked Vote Choice							
HAUVILLE Leo	1	2	**3**	4	5	6	7	No Rank
EVANS Andrew	**1**	2	3	4	5	6	7	No Rank
BAIN Arthur	1	**2**	3	4	5	6	7	No Rank
CAMPBELL Liz	1	2	3	**4**	5	6	7	No Rank
SAUL Dean	1	2	3	4	5	6	7	**No Rank**
IRWIN Troy	1	2	3	4	**5**	6	7	No Rank
RAEBURN Bruce	1	2	3	4	5	6	7	**No Rank**

No audit board consensus

Blank vote - no mark

✐ Add comment

Figure 2.3: The audit step for an IRV ballot. For each mark on the ballot, the audit board selects the candidate by row and the rank by column.

2.10 Audit reports

There are three new reports specifically for IRV: *summarize_irv*, *assertions* and *ranked_ballot_interpretation*. There are also some small changes to existing reports.

2.10.1 IRV assertion generation summaries

summarize_irv.csv/json contains the results of running assertion generation, for each IRV contest. This is usually just the name of the winner, but will include any errors or warnings in the event that assertion generation produced them.

2.10.2 Assertion Export

assertions.csv/json contains the assertions along with up-to-date information about discrepancies. It also contains extra contest-related data such as the winner, overall (minimum) margin and current risk. The JSON assertions can be exported somewhere for the public to examine, for example to the audit center website.

It is important to verify that the assertions truly imply that the announced winner won. Although the assertion exports are human-readable, and can be validated manually, most people will find it easier to use a tool. The assertions exported by *colorado-rla* in JSON form can be visualised in the assertion visualizer. Simply paste the json into the window. The website provides an interactive process for verifying that the assertions imply a particular winner. Viewers should check that every candidate other than the announced winner is ruled out. Figure 2.4 shows an example of visualizing the assertions for the Guide to Raire Part 2 Example 1. Every candidate other than the announced winner (Chuan) is ruled out.

These reports contain a current risk estimate for each assertion, and a statement of the overall risk for the set (which is the maximum of any of the individual risks). This measure will be accurate for targeted contests and for single-county contests. However, the results are not accurate for cross-county non-targeted IRV contests, because in that case *colorado-rla* does not guarantee a uniform sample.

2.10.3 Ranked ballot interpretation

ranked_ballot_interpretation.csv/json records all invalid IRV ballots that were received, either during the initial csv upload or during the auditing phase. It lists the County, Contest name, CVR number, imprinted ID, and both the raw IRV vote and the valid interpretation made by the software.

2.10.4 Modifications to existing reports

Existing reports have slight modifications, mostly to omit IRV details when they do not make sense.

The assertions

Assertions - difficulty = 27 margin = 500

4.5	3000	NEN: Alice > Bob if only {Alice,Bob,Chuan,Diego} remain
27	500	NEN: Alice > Diego if only {Alice,Chuan,Diego} remain
27	500	NEN: Chuan > Alice if only {Alice,Chuan} remain
5.4	2500	NEN: Chuan > Diego if only {Alice,Chuan,Diego} remain
3.375	4000	Chuan NEB Bob
3	4500	NEN: Alice > Diego if only {Alice,Diego} remain

Explanation of why it works

☐ Show effect of each assertion sequentially (can be slow)
☑ Don't bother drawing the (technically unnecessary) trees for the winning candidate.
☑ Prevent text overlapping (by making trees wider if needed)
☑ Show the index of the assertion that eliminated a branch
☑ Show a text description of the assertion that eliminated a branch
☑ Split the text description over more lines

Demonstration by showing what eliminated each possibility

Alice is ruled out by the assertions
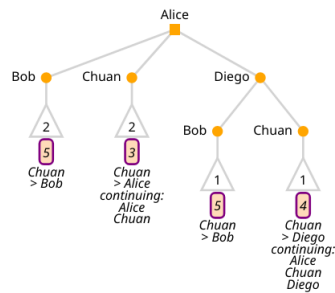
Bob is ruled out by the assertions

Diego is ruled out by the assertions
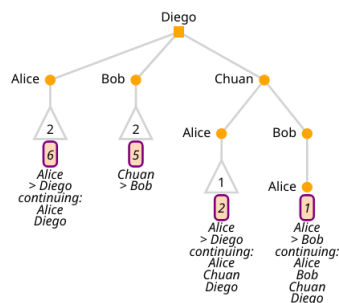
Figure 2.4: Visualizing the assertions from the Guide to Raire Part 2, Example 1. All candidates other than the apparent winner (Chuan) are ruled out.

- IRV contests are omitted from *tabulate.csv* and *tabulate_county.csv,* which have now been renamed to *tabulate_plurality.csv* and *tabulate_county_plurality.csv* respectively, because corla's tallies are designed for plurality and do not get the correct answers for IRV.

- IRV details are omitted from the summary sheet in the state report. The IRV contests still appear, but rather than including the winner, tallies, margin and diluted margin (as they do for plurality), only the contest name is listed. This data is available in the IRV-specific reports.

- *ResultsReport.xlsx* omits "votes for winner" and "votes for runner up" for IRV contests. The "total votes" are the total number of choices in all valid IRV interpretations for the contest.

Several reports summarize the discrepancies for each contest, across 5 columns headed "+2" (for 2-vote overstatements), "+1" (for one-vote overstatements), "zero value" (for discrepancies with no impact), "-1" and "-2" for understatements. These summaries have to be adapted for IRV, because each IRV contest has multiple assertions, which a given discrepancy might impact differently. For IRV, these reports count each discrepancy under the column for the *most impactful* type of discrepancy it causes any assertion. For example, a discrepancy that caused a +2 overstatement on one assertion, and a +1 overstatement on another assertion, would count in the +2 column; a discrepancy that caused a +1 overstatement on a few assertions, and a zero-value discrepancy on the rest, would count in the +1 column. These values are not used in auditing—they are simply used to give humans an easily-understood summary of the quantity of discrepancies. The decision to summarize the discrepancies in this way has two implications:

1. the total number of discrepancies for the contest is equal to the total of the values across the 5 columns,

2. the summary may overstate, but never understates, the overall impacts of the discrepancies compared with a plurality contest of the same diluted margin.

The last point is because, in plurality contests, all discrepancies are effectively counted against the same assertion, but in IRV different discrepancies might affect different assertions. It is much less of a concern to have three different overstatements all affecting different assertions, than three overstatements that all affect the same assertion.

The *assertions.csv* and *assertions.json* assertion export files give individual current risk assessments for each assertion.

Chapter 3

colorado-rla: main classes and processes

3.1   The CORLA Model

Location: corla.model

This section provides a high level overview of how the colorado-rla system models election and audit concepts through the classes in corla.model. The intention is not to provide a formal specification, but to identify which aspects of the system have implications for how IRV is integrated into colorado-rla.

3.1.1   Modelling of Contests

Relevant classes: County; Contest; Choice; CountyContestResult; ContestResult; ContestToAudit.

County: The state of Colorado is comprised of a set of counties. Some contests will involve only one county, where others are cross-county (voters from multiple counties vote in these contests).

A *County* has a name and a numeric identifier. A *Contest* has a name, is associated with a *County*, has a textual description, a list of candidates (*List<Choice>*), the number of winners (this is 1 for IRV contests, and usually 1 for plurality but sometimes higher for committees), maximum number of cast votes allowed (which matches the number of candidates for IRV and is equal to the number of allowed selections for plurality), and a sequence number. A *Choice* has a name, a textual description, and a boolean indicator of whether the candidate is a qualified write in or is fictitious. Fictitious candidates are essentially ignored by the system.[1] A cross-county contest will correspond to several different *Contest* objects, one for each county that runs that contest. These should all have the same name after canonicalization.

---

[1] These are the candidates called "Write-in", which are used to indicate the beginning of the write-in candidate columns in the CVRs, but are not actually valid choices.

Note on IRV integration. The *description* in a contest is used to identify the social choice function: either IRV or PLURALITY. These must be consistent for all contests of the same name.

A *CountyContestResult* represents the result of a *Contest* that has taken place in a *County*. Cross-county contests will involve a separate *CountyContestResult* for each county. (The *ContestCounter* controller tallies the votes across all *CountyContestResult*s relevant to each *Contest* forming a list of *ContestResult*s). The attributes of a *CountyContestResult* include: the *County* and *Contest*; the number of allowed winners; the set of winners (*Set<String>*); the set of losers (*Set<String>*); vote totals for each candidate (*Map<String,Integer>*); the minimum pairwise margin between a winner and loser; the maximum pairwise margin between a winner and a loser; the number of ballots cast in the *County*; and the number of ballots cast in the *County* that contain the *Contest*. The vote totals and ballots cast counters are updated through the addition of CVR data (method *addCVR()* is called for each *CastVoteRecord*). After CVR data is provided, a method is defined to compute vote totals and pairwise margins (*updateResults()*).

Note on IRV integration. The tallies computed in CountyContestResult for IRV contests are meaningless and are ignored.

A note on margins Different margins are calculated in *CountyContestResult*. These are: the *countyDilutedMargin()* defined as the minimum pairwise margin divided by the number of ballots cast in the *County*; the *contestDilutedMargin()* defined as the minimum pairwise margin divided by the number of ballots cast in the *County* that contain the *Contest*; and *minMargin()* and *maxMargin()* referring to the smallest and largest pairwise margins between a winner and loser, respectively.

maxMargin, *ContestResult::getMaxMargin* and *CountyContestResult::maxMargin* seem to be unused. For reporting, *ContestResult::getMinMargin* is used.

Where *CountyContestResult* represents the results for a *Contest* in a single *County*, the *ContestResult* class captures the result of a contest across all counties. The *CountyContest* class captures some of the same type of information that *CountyContestResult* does: the number of winners allowed; the set of winners (*Set<String>*); the set of losers (*Set<String>*); vote totals for candidates (as a *Map<String, Integer>*); the diluted margin; and the minimum and maximum pairwise margins. Two different concepts need to be distinguished.

- *ballotCount* is the total number of ballots cast in all counties that ran the contest (including ballots that do not contain the contest).

- *totalVotes()* is the total number of votes cast in the contest. For example, in a vote-for-3 plurality contest, some ballots might contribute 3 votes to this tally. This value counts all the ranks chosen on each IRV ballot, and is not used for IRV audits.

Important fact: the *ballotCount* in *ContestResult* is the correct universe size for audit purposes. This is the total number of ballots (actually, the total number of cards) in all counties involved in the audit

of this contest. This is calculated in *ballotManifestInfoQueries* by simply summing the ballots in each relevant county's manifest. It is used in the *ContestCounter* as the denominator in the diluted margin computation.

A *ContestResult* is associated with a set of *County*s (multiple counties will participate in a state-wide contest) and a corresponding set of *Contest*s (one for each county that ran the contest). A *ContestResult* is also associated with an *AuditReason*–an enumeration over the options STATE_WIDE_CONTEST, COUNTY_WIDE_CONTEST, CLOSE_CONTEST, TIED_CONTEST, GEOGRAPHICAL_SCOPE, CONCERN_REGARDING_ACCURACY, OPPORTUNISTIC_BENEFITS, and COUNTY_CLERK_ABILITY; and an *AuditSelection*–an enumeration over AUDITED_CONTEST and UNAUDITED_CONTEST.

The *ContestToAudit* class bundles a *Contest* with an *AuditReason* and an *AuditType*. The *AuditType* enumeration consists of the options: COMPARISON; HAND_COUNT; NOT_AUDITABLE; and NONE.

3.1.2 Modelling of CVRs, Audited CVRs, and Discrepancies

Relevant classes: CVRContestInfo; CastVoteRecord; CVRAuditInfo; Tribute; BallotManifestInfo.

A *CVRContestInfo* captures contest-specific details from a CVR. For each CVR and contest on that CVR, *colorado-rla* makes a *CVRContestInfo* object. A *CVRContestInfo* is associated with a cast vote record (although the associated cast vote record is not identified in the *CVRContestInfo* object), a *Contest*, a textual comment, a *ConsensusValue* (an enumeration over YES and NO) and a list of choices (*List<String>*). Note that choices here are represented as Strings rather than *Choice* objects. This class is used for both the CVRs uploaded by counties at the beginning of the audit, and CVRs uploaded by the audit board—the *ConsensusValue* is set to 'NO' if the audit board members disagree about the contents of the ballot.

A *CastVoteRecord* represents a single ballot cast by a voter, in which votes across one or more contests are made. This class is used to represent both reported CVRs (those scanned by tabulators) and audited ballots (those collected by auditors and entered into the auditing system). The latter are referred to as ACVRs in the code. Irrespective of whether the *CastVoteRecord* is capturing a reported CVR or an audited CVR (ACVR), it is associated with a *RecordType*, a county id (numeric), CVR number (numeric), sequence number (numeric), scanner id (numeric), batch id (String), record id (numeric), imprinted id (String), a ballot type (String), and a list of *CVRContestInfo* objects for the contests that are included on the ballot. A CVR's *sequence number* represents the order in which CVRs for a county are uploaded. A sequence number of n denotes that the CVR was the $(n-1)^{th}$ imported for the county.

The actual selections are stored as a string. For example, the string *["Alice","Bob","Chuan","Diego"]* would be stored as the choices indicating that four-candidate selection in a multi-choice plurality contest.

IRV preferences are represented in the uploaded csv files with their ranks explicitly written. For example the sequence ["Alice(1)", "Bob(2)", "Chuan(3)", "Diego(4)"] Indicates a first preference for Alice, a second preference for Bob, etc.

Note on IRV integration.   In order to make canonicalization succeed, we store IRV CVRs as an ordered list without explicit ranks, with the first-ranked candidate first, then the second-ranked candidate, etc. The example vote above is stored as ["Alice", "Bob", "Chuan", "Diego"]. Because ballots may include invalid preferences (such as duplicates, repeated or skipped preferences) they must be appropriately interpreted before storage—see Section 4.4.

Audit CVRs require equivalent processing because they also may not be valid. We discuss processing of audited ballots in Section 4.6.

The *ballot type* attribute defines what contests are on the ballot. Not every voter is eligible to vote in every contest held in a county. (Note that some pre-existing test data has curious conditions where some CVRs of the same ballot type include votes for a given contest but others do not).

The *RecordType* of the *CastVoteRecord* indicates whether the object is representing: an UPLOADED CVR, an AUDITOR_ENTERED ballot (ACVR), a REAUDITED ballot (used to indicate that the ACVR is not the latest revision—the ballot has been reaudited with the current ACVR represented by the AUDITOR_ENTERED record), a PHANTOM_RECORD (CVR), a PHANTOM_RECORD_ACVR (ACVR), or a PHANTOM_BALLOT.

A PHANTOM_RECORD means that the manifest indicates a CVR that is not present in the CVR export file. This automatically creates PHANTOM_RECORD (CVR), and also PHANTOM_RECORD_ACVR (ACVR) if that ballot is sampled. A PHANTOM_BALLOT means that the CVR is present but the audit board were unable to find the corresponding paper record.

Where the *CastVoteRecord* is capturing an *audited* ballot (an ACVR), additional attributes become relevant: an audit board index (numeric, defining *who* submitted the audited CVR); the ID of the uploaded CVR matching the ballot being audited; a textual comment indicating the reason why the ballot was reaudited (if relevant); boolean flags to indicate whether the CVR has been audited; and the number of the round in which it was audited; and a timestamp (presumably indicating *when* the ACVR was submitted). The attribute *revision* (numeric) is relevant in the context where multiple revisions to an ACVR are submitted by auditors. We assume that new *CastVoteRecord* objects are created for each revision.

A random number called *rand* is stored in each *CastVoteRecord* object This relates to the fact that when sampling ballots, we think of all ballots as forming a long sequence. A series of random numbers are generated, and the ballots at those indices in our sequence are the ones that form our sample.

The *CVRAuditInfo* class pairs together a CVR to audit (a *CastVoteRecord*) and a submitted audited ballot (a *CastVoteRecord* representing an ACVR). Attributes define an ID (the ID of the CVR being audited), two *CastVoteRecord* objects (the CVR and ACVR), an indication of how many times the associated CVR appears in the sample of a comparison audit being undertaken (as a map between comparison audit ID and this number of times, *Map<Long,Integer>*, called *multiplicity_by_contest*), and three further attributes whose meaning is somewhat less clear–*count_by_contest*, *my_discrepancy*, and *my_disagreement*. The latter two are modelled as sets of *AuditReason*, an enumeration over the reasons why a contest may be

chosen for audit. Commenting associated with these attributes indicate that they represent the number of discrepancies and disagreements found in the audit 'thus far'.

Modelling discrepancies and disagreements as a set of *AuditReason* objects keeps track of how many of the discrepancies and disagreements associated with this CVR-ACVR pair are in contests being audited for different reasons. (Although, ultimately all the discrepancies and disagreements are incorporated into audit *Round* objects in a map with keys *AuditSelection.UNAUDITED_CONTEST* and *AuditSelection.AUDITED_CONTEST*).

Commenting attached to the *count_by_contest* attribute indicates that it refers to the 'number of times this CVRAuditInfo has been counted/sampled in each ComparisonAudit'. This is because corla conducts sampling *with* replacement, so a given ballot may occur multiple times in the same sample. (It is never counted in the election result more than once.) The ballot is requested only once from the audit board—any repeats simply reuse the data that was uploaded in the first ACVR.

The concept of 'unauditing' appears in *CVRAuditInfo* (i.e., the method *resetCounted()* has the comment 'clear record of counts per contest, for unauditing'). This relates to reauditing (ie., when auditors reaudit a sample of ballots, they are 'unaudited' and then new ACVRs submitted for that sample—see Section 3.4).

A *Tribute* is defined in the code as a theoretical CVR that may or may not exist. It is associated with a an id, county id, scanner id, batch id, ballot position (offset), a random number, a sequence position, and a contest name. A 'Tribute' is created for a CVR, containing all the "metadata needed to find a CVR from a manifest given a sample position". They are created by the method *addTribute* in *BallotSelection.Segment*. This method is called by the method *Selection.addBallotPosition*, which itself is called from the method *Selection.selectTributes(Selection, Set<Long>, Set<BallotManifestInfo>*, which is called from *Selection.randomSelection* (through a few other *selectTribute* methods). Within *Selection.randomSelection*, the tributes are 'resolved' by calling the (static) method *Selection.resolveSelection* and the CVRs for those tributes are collected and stored within the *Selection* object provided as input. When sampling ballots the code forms a list of random numbers which each of those numbers maps to a particular CVR. The 'Tribute' contains the metadata to required to identify the CVR for a given 'randomly generated number'. *Selection.randomSelection* is called from *StartAuditRound.makeSelections*. Note that a *Selection* object represents the random sample of CVRs for a given contest. *Selection*s are combined into *Segment*s, where a *Segment* contains the CVRs for a given county to audit (capturing all the contests being audited, relevant to that county).

The *BallotManifestInfo* class represents storage information relating to a batch of ballots. A complete ballot manifest for a county is represented by a set of *BallotManifestInfo* objects. A *BallotManifestInfo* is associated with a batch number, scanner ID, batch size, and storage location. *BallotManifestInfo* also contains attributes for sequence start and end number, ultimate sequence start and end, and a method to 'translate a generated random number from contest to county scope, then from county to batch scope'.

3.1.3   Modelling of Audits

Relevant classes: ComparisonAudit; Round; (CountyContestComparisonAudit)

The class *CountyContestComparisonAudit* is not being used, so we will not discuss it here.

Each audit being undertaken across the state of Colorado is represented by a *ComparisonAudit* object. (Note that these are created by the *ComparisonAuditController*). Functionality for estimating sample sizes and computing discrepancies between CVRs and audited ballots for a contest sits within this class.

A *ComparisonAudit* is associated with an *AuditReason* (an enumeration, as described earlier), a *ContestResult*, an *AuditStatus* (an enumeration over the options NOT_STARTED, NOT_AUDITABLE, IN_PROGRESS, RISK_LIMIT_ACHIEVED, ENDED, and HAND_COUNT), the diluted margin of the contest, gamma value (for audit math), a risk limit, running counts of one and two vote under and over statements, running counts of other discrepancies that do not fall into those categories, running counts of disagreements (differences of opinion on how an audited ballot was interpreted by auditors), optimistic and estimated samples to audit, number of samples already audited, flags to indicate whether various sample counts need to be updated, and IDs for CVRs relevant to the contest under audit (*contestCVRIds*). The actual discrepancies are stored in a map, *Map<CVRAuditInfo,Integer>*–this assumes that a CVR-ACVR pair can result in only one discrepancy for a contest. The disagreements are stored in a set, *Set<CVRAuditInfo>*.

The 'optimistic' sample count assumes that no further overstatements will occur. It represents how many ballots we expect we will need to sample so that the risk for the audit falls below our desired threshold (the risk limit), assuming that we will see no (further) discrepancies. The 'estimated' sample count assumes that overstatements will continue to occur at the rate observed already.

A *ComparisonAudit* object has methods for: updating the status of the audit (recalculating the values of the optimistic/estimated sample counts if required, which determines whether the risk limit has been met), computing optimistic/estimated sample sizes for the audit, computing the discrepancy of a *CVRAuditInfo* with respect to the contest being audited (note that there are two methods for this–*computeDiscrepancy* and *computeAuditedBallotDiscrepancy*), signalling that a sample has been audited (or 'unaudited') and that sample size counts may need to be recalculated, and recording and removing discrepancies and disagreements. It appears that *computeDiscrepancy* is not used.

An audit round is represented by the *Round* class. A *Round* has a start and end time, a number of attributes pertaining to the number of ballots to be audited in the round (estimated and actual), and attributes and methods used to identify/provide access to the set of CVRs to be audited in the round. Discrepancies and disagreements occurring during a round are stored in two maps (*Map<AuditSelection, Integer>*), where *AuditSelection* is a enumeration over AUDITED_CONTEST and UNAUDITED_CONTEST. This relates to the opportunistic computation of discrepancies for contests that are not being targeted by the audit (the *ComparisonAudit*s for those contests will have the *AuditReason* OPPORTUNISTIC_BENEFITS).

A *ComparisonAudit* object is created for every contest, even those that are not being targeted for audit. However, non-targeted contests will have the *AuditReason* OPPORTUNISTIC_BENEFITS. The *Round* class must capture data relating to all audits being undertaken: CVRs under audit; ballots audited; and discrepancies and disagreements (totals for targeted and non-targeted contests). It does not need to *know* what audits are Plurality and which are IRV, however, and did not change to incorporate IRV.

Note on IRV integration. We moved the function for computing risk measurement from *ReportRows::riskMeasurement()* into *ComparisonAudit* This allows the *ComparisonAudit* class to represent *any* kind of assertion-based audit.

Note on IRV integration. We have implemented a new *IRVComparisonAudit* class, which inherits from *ComparisonAudit*. This allows the IRV audits to fit into colorado-rla's existing logic, by providing an interface identical to that of *ComparisonAudit* and keeping the IRV-related calculations internal to the subclass. See Subsection 4.3.3.

3.1.4   Modelling of Dashboards

Relevant classes: CountyDashboard; DoSDashboard, AuditBoardDashboard.

The state and counties access audit related information through two different dashboards.

CountyDashboard:

- Discrepancy and disagreement counts are represented as they are in the *Round* class, as a map between *AuditSelection* and Integer.

- *CountyDashboard* contains a method *addDiscrepancy* that models discrepancies as a set of *AuditReason*. Possibly the reason for modelling discrepancies as *AuditReason* is that it identifies whether the discrepancy is for an audited or unaudited contest – however, if this is the case, why would they not be represented as *AuditSelection*s? (Possibly, there was an idea to have a more fine grained count of discrepancies per audited contest type, but that ultimately this was not pursued).

- Contains a method for starting an audit round. This method, *startRound* creates a *Round* object and adds it to the attribute *my_rounds*.

- Contains a list of *AuditInvestigationReportInfo* and *IntermediateAuditReportInfo*. They are both essentially wrappers around a String, with some additional details (a timestamp and name). We have not made any changes for IRV.

- Methods for addition and removal of audited ballots.

- A method for 'setting' the collection of *ComparisonAudit*s being undertaken.

- Methods for indicating which contests are the *driving* contests of the audit (assumption is that these are the set of contests that are being audited for a reason other than OPPORTUNIS-TIC_BENEFITS). The driving contests are identified by their names (a set of Strings).

The *AuditBoardDashboard* is a simple dashboard, accessible from the *CountyDashboard*, allowing uploads of sampled ballots when the audit board logs in.

A note on IRV integration As information in the *CountyDashboard* for a county is used to update the information presented to the county in the client, it needs to have visibility of: *all* audits being undertaken in the county (including IRV); all CVRS under audit (including those only being audited for IRV contests); and all discrepancies and disagreements that have arisen across both Plurality and IRV audits [see *CountyDashboardRefreshResponse.java*]. This does not mean that the *CountyDashboard* needs to *know* which audits are for IRV and which are for Plurality, except that the audit dashboard needs to present IRV ballots in a table format (as shown in Figure 2.3).

DoSDashboard is simpler than *CountyDashboard*, with attributes and methods responsible for maintaining a collection of *ContestToAudit* objects across the state (addition, removal, and updating).

When the DoS Dashboard is refreshed in the client, information from *DoSDashboard* relating to *all* audits being undertaken across the state (including IRV) is used when forming the *DoSDashboardRefreshResponse*. [See *DoSDashboardRefreshResponse.java*]

Note on IRV integration. We have added GenerateAssertionsSummaries to the DoSDashboard—this tells the DoS dashboard basic data about the success or failure of assertion generation for each IRV contest. See Figure 2.1 for a display.

3.1.5   Modelling of Users

Relevant classes: Administrator; AuditBoard; Elector.

The details of these classes are not relevant to the addition of IRV audits to colorado-rla.

3.1.6   Other

Relevant classes: LogEntry; UploadedFile; ImportStatus; IntermediateAuditReportInfo; AuditInvestigationReportInfo.

The details of these classes are not relevant to the addition of IRV to colorado-rla.

3.2   Audit Math

Location: corla.math

Relevant classes: Audit.

The *Audit* class contains a collection of static methods related to the mathematics of risk calculations.

- Two methods for computing the diluted margin for a contest given the margin and total number of auditable ballots relating to that contest. They differ in the types used to represent the margin and ballot count (i.e., Integer/Long and BigDecimal). The diluted margin is computed by dividing the input margin by the ballot count.

- A method called *totalErrorBound* which returns $\frac{2\gamma}{\mu}$ where μ denotes a diluted margin and γ is a parameter relating to the risk function being used in the audit.

- Two methods called *optimistic* (one calls the other), responsible for computing the total number of ballots we expect to need to sample to audit a given contest (based on a given risk limit, diluted margin, gamma value, and count of one/two vote under and overstatements experienced thus far) assuming we see no further discrepencies between CVRs and the paper ballots.

- A method *pValueApproximation* that computes the current level of risk achieved for a given contest being audited, given: the number of ballots audited; the diluted margin; gamma value; and number of one and two vote under and over statements. It appears that this method is only used when creating reports of conducted audits, rather than to signal that an audit should move from the IN_PROGRESS to RISK_LIMIT_ACHIEVED states. [For more detail on audit states, see Section 3.9]

A note on audit state transitions: The transition between IN_PROGRESS to RISK_LIMIT_ACHIEVED for an audit seems to be determined by recalculation of the *optimistic* sample size as ballots are sampled, and discrepancies are recorded. Once the number of ballots audited reaches this optimistic ballot sample count, audit states shifts to the RISK_LIMIT_ACHIEVED state. If the risk limit has not yet been achieved for a contest after a sample has been audited, the optimistic ballot sample count will increase when it is recomputed. We can think of (optimistic ballot sample count - audited sample count) as representing the number of ballots that we still have to sample for a contest in order to achieve the risk limit, assuming that we will see no further discrepancies. If we do see further discrepancies, the optimistic ballot sample count may increase when it is recomputed.

3.2.1   Estimating sample sizes

The estimate of the Round 1 sample size is calculated in *Audit::optimistic*, which is calculated separately for each *ComparisonAudit*. The comment at the top of the function suggests it is meant to take the equation from Lindeman and Stark's "Gentle Introduction to Risk Limiting Audits," but that paper in turn refers back to Stark's "Super simple simultaneous risk limiting audits." It implements a variant of Equation [17] of Stark's "Super Simple Simultaneous Risk Limiting Audits," but with a slightly more general version derived from Stark's Equation [10] incorporating not only one-vote overstatements, but also two-vote overstatements and understatements.

If α is the risk limit, μ is the diluted margin, o_1 and o_2 are the numbers of one- and two-vote overstatements respectively, and u_1 and u_2 are the one- and two-vote understatements, *Audit::optimistic* calculates

$$-2\gamma \left[\log \alpha + u_1 \log(1 + \frac{1}{2\gamma}) + u_2 \log(1 + \frac{1}{\gamma}) + o_1 \log(1 - \frac{1}{2\gamma}) + o_2 \log(1 - \frac{1}{\gamma})\right] \cdot \frac{1}{\mu}$$

Audit::optimistic returns the maximum of the above and $o_1 + o_2 + u_1 + u_2$.

Note that this is a little more general than Equation [17] of "Super Simple"—it includes terms for understatements and 2-vote overstatements, derived in the same way as Equation [17]. Also note that it produces the same results[2] as Equation [1] of "A gentle introduction to Risk Limiting Audits" with $\gamma = 1.03905$ and $\alpha = 0.1$.

For escalation audit rounds, the code multiplies the "optimistic" value by a scaling factor computed in *ComparisonAudit::scalingFactor*. We are unsure where this calculation comes from—the code does not include a reference. It is probably just a best-effort guess to generate decent approximate sample sizes. Because the scaling factor is always at least one, this strategy is always conservative.

The scaling factor is intended to be computed as:

$$1 + (o_1 + o_2)/auditedSamples$$

when the number of audited samples, 'auditedSamples', is at least 1, and 1 otherwise. However, there seems to be a mistake in the implementation: the input values of o_1 and o_2 are always zero, so the scaling factor is always 1. This does not impact the accuracy of the audit, but it means there is actually no scaling.

3.2.2   Using sampled ballots past the estimated size needed

Based on our discussion online and on examining the code, we believe that if two different comparison audits have the same universe, but different sample sizes, then they will each do risk calculations up to their expected sample size *and no further* even if the other audit causes more ballots to be drawn from the same universe.

This is not wrong, but may cause an unnecessary escalation to Round 2, because the contest that expected a smaller sample may not actually confirm its result, but possibly would have been able to do so using the further samples motivated by the other audit.

---

[2]Actually the coefficient of u_2 isn't quite the same, but this is irrelevant for our purposes because Super Simple almost never produces 2-vote understatements.

We assume that this is not currently causing serious problems, because it does not often arise during current CO audit practices. However, if Colorado decides to target a large number of contests simultaneously it will matter, and may result in much more escalation to Round 2 than necessary.

A note on IRV integration Our IRV RLA integration changes neither correct nor exacerbate this problem.

3.3   Persistence and Database Queries

Location: corla.persistance and corla.queries

Hibernate annotations attached to classes are used to automatically construct database tables. Changes to existing tables, and descriptions of new database tables for IRV, are in Chapter 4.

3.4   Controllers

Location: corla.controller

The *ComparisonAuditController* is the workhorse for running audits across the set of counties. *ComparisonAuditController* contains static methods for:

- Returning a list of CVRs to audit in a given audit round, or those that remain to audit in a given round (across all counties, or those specific to a given county).

- Creating a *ComparisonAudit* object for a contest (the method takes the contest's *ContestResult* and the risk limit of the audit).

- Submitting audited ballots (this method takes as input the *CountyDashboard* and two *CastVoteRecord* objects representing the CVR of the ballot and the ACVR generated by an auditor). These CVR-ACR pairs, stored within a *CVRAuditInfo* object, are stored in the database.

- A method that processes audited ballots, currently stored in the database but that have not yet been processed, and calls the *audit* method on those *CVRAuditInfo* objects.

- A method, *audit*, that takes a *CVRAuditInfo* object representing a CVR-ACVR pair, computes discrepancies and disagreements with respect to contests relevant to that CVR-ACVR pair (by accessing methods in the *ComparisonAudit* objects for those contests), stores those discrepancies and disagreements in the *Round* object for the audit round in progress, and signals to each relevant *ComparisonAudit* that ballots relevant to their audit have been sampled.

- Re-auditing a ballot (this involves 'unauditing' the ballot – removing any discrepancies and disagreements relating to the ballot from all counters and data structures that store discrepancies

and disagreements; removing the audited ballot from dashboard counters; and triggering comparison audits that had that ballot in their sample to update their own internal data structures and estimates of remaining ballots to sample – and then adding the new interpretation of the ballot to the database).

- Update data structures storing discrepancies and disagreements that have occurred during each audit round (in the relevant *Round* object).

- Returning the estimated number of ballots to audit, by iterating over the driving contests under audit, taking the maximum of their estimated sample sizes (note that the *estimated* sample number, obtained from a *ComparisonAudit* object, assumes that overstatements will continue to occur at the current rate).

Note on IRV integration. This class was updated to control IRV comparison audits also. The controller now creates an instance of *ComparisonAudit* for Plurality contests and an instance of *IRVComparisonAudit* for IRV contests. Where the system performs operations over a collection of *ComparisonAudits*, it does not need to know which of these are IRV audits and which are Plurality. See Section 4.3.3 for implementation details.

The *ContestCounter* controller is responsible for collating *CountyContestResult*s across counties, forming a single *ContestResult* for each contest. This controller contains static methods for:

- Collecting *CountyContestResult* objects (from the database), grouping them by contest name, and tallying them to produce state-wide results (*countContest, CountAllContests, accumulateVoteTotals, addVoteTotal*). The state-wide results, or county results if the contest is county specific, are contained in a resulting *ContestResult* object.

- Computing pairwise margins between winners and losers given a data structure containing tallies for those candidates (a map between candidate name and their vote total). This assumes the plurality context, where candidates each have a single tally.

DeleteFileController and *ImportFileController* do not appear to contain any details that would need to be different when IRV auditing is added to the system.

The functionality present in the *BallotSelection* controller appears to be independent of the type of contests that each *CastVoteRecord* contains votes for. This controller contains static methods for combining CVR records with information from a ballot manifest, which details how to find the paper ballots associated with the CVRs (creating *json::CVRToAuditResponse* objects for these CVRs), and combining CVRs to be audited for a single contest into a *Segment* object. Ballot sampling is described in more detail in Section 3.6.

The *AuditReport* controller contains the functionality for creating:

- Activity reports for a single contest or all contests being targeted for audit;

- Result reports for a single contest or all contests being targeted for audit.

See Section 3.7 for further details.

3.5 CSV Parsing

Location: corla.csv and corla.json

Note on IRV integration. The parsing of CSV CVR export files needed to be extended to incorporate IRV. See Section 4.4 for discussion of ballot parsing and CVR storage that includes IRV.

3.6 Ballot Sampling

This should not need to change for IRV. (Computing sample sizes is different for Plurality and IRV, but the process of collecting the sample given the sample size is not dependent on the type of contest).

3.7 Reports

Location: corla.report

Additional report options *summarize_irv*, *assertions* and *ranked_ballot_interpretation* were added.

3.8 Endpoints

Location: corla.endpoints

StartAuditRound:

- This endpoint involves a lot of operations over lists of *ComparisonAudit*. All these activities that were previously being done over Plurality audits now iterate over all (IRV and plurality) audits. The code in *StartAuditRound* did not need to change.

- If the ASM state is at the stage where all audit information has been provided, but the first round has not yet started (state is COMPLETE_AUDIT_INFO_SET), the endpoint initializes the *DoSDashboard* by calling *initializeAuditData()*, and then calls the method *startRound()*. As we shall see, *initialiseAuditData()* initializes the set of contests being audited (via *initializeContests()*, the set

of audits being undertaken (via *initializeAudits()*) and the set of *CountyDashboard*s (via *initialize-CountyDashboard()*).

· The set of contests under audit are initialized (*ContestResult*s are created for each such contest) in the method *initializeContests*. This method calls the method *countAndSaveContests*, which in turn calls *ContestCounter.countAllContests()*, persisting the results.

· Plurality comparison audits are currently initialized by the method *initializeAuditData*, which takes as input the *DoSDashboard*, and subsequently calls the method *initializeAudits*. These *ComparisonAudit*s are then assigned to the relevant *CountyDashboard* through the method *initializeCountyDashboard*. .

· When initializing a County Dashboard (via *initializeCountyDashboard*), it collects the names of the driving contests (these are the contests that are being audited for a reason other than OPPORTUNISTIC_BENEFITS), assigns *ComparisonAudit*s relevant to that County to their dashboard, and creates and initializes the ASM for the County (*CountyDashboardASM*).

· The *startRound* method does the following:

  – It selects ballot samples for each *ComparisonAudit* being undertaken, using the method *makeSelections*(). This method takes the set of *ComparisonAudit* objects, the seed, and the desired risk limit as inputs.

  – It then collects the set of County Dashboards for which an audit is 'ready to start' (by calling *dashboardsToStart()* as part of the condition in a for loop).

  – A *CountyDashboard* is 'ready to start' if it is not in its initial state or in its final (finished or deadline missed) state. We expect that the dashboard will have transitioned through several states as its ballot manifest and CVRs have been successfully imported and integrated.

  – For each *CountyDashboard* with an audit 'ready to start', it looks at its state to see if its audits are complete. If so, it raises an RISK_LIMIT_ACHIEVED_EVENT and updates its ASM state. (This county dashboard will no longer appear in the list of 'ready to start' dashboards).

  – If all audits are not yet complete for the county associated with the *CountyDashboard*, it then computes the total number of disagreements that have arisen.

  – If the county is in the ASM state COUNTY_AUDIT_UNDERWAY, the following conditions will be checked. If there are *no disagreements*, and not all audits for the county have finished, it then calls *updateStatus* on each *ComparisonAudit* being undertaken in the county that has not yet finished (and then persists the *ComparisonAudit* object). If there are *no disagreements* and all audits *have* finished for the county, it raises the RISK_LIMIT_ACHIEVED_EVENT and updates

the *CountyDashboard*'s ASM state. In the latter case, we then move on to the next county dash board 'ready to start'.

 - Provided there is a non-empty list of audits being undertaken for the county, *startRound* then moves on to form a ballot sample *Segment* for the county. This essentially combines the ballot selections for all contests that the county is participating in. The created *Segment* is then distilled into a list of *CastVoteRecord*s that the audit boards (for the county) have to collect (phantom ballots/records are removed, duplicates are removed). This distilled list is called *ballotSequenceCVRs*. It is distilled further into a list containing just the CVR IDs, this is called *ballotSequence*. If this list is not empty, *ComparisonAuditController.startRound()* is called with the set of *ComparisonAudit*'s associated with the county and the ballot sequence/segment data passed as inputs. The ROUND_START_EVENT occurs for the *CountyDashboard*. Note that *ComparisonAuditController.startRound()* will call the *startRound* method of the relevant *CountyDashboard*. This will move audits for that county to the next round (if rounds have already been undertaken).

 - We then move on to the next *CountyDashboard* that is 'ready to start'.

 - The endpoint will be accessed for each round – rounds are synchronized across all counties.

SignOffAuditRound:

 - The method *logAuditsForCountyDashboard()* logs information relating to each comparison audit being undertaken in a given county (contest name, audit reason, audit status, and whether it is being targeted). This did not change for IRV.

SetContestNames:

 - This endpoint deals with canonicalization (of contest names and choice names within a contest). It updates choice names in *CVRContestInfo*s in the database for CVRs [it calls CastVoteRecord-Queries.updateCVRContestInfos()]. This did not change for IRV—canonicalization of IRV contest and candidate names can proceed using the same code as plurality.

SelectContestsForAudit:

 - This endpoint interacts with the *DoSDashboard*, which will need to have a view of all audits being undertaken. Some of the contest set to be audited will be IRV. However, none of the code in this endpoint needs to care whether the contests are IRV or Plurality.

IndicateHandCount:

- The functionality in this endpoint is relevant to all audits being undertaken (including Plurality and IRV), but does not need to know which are Plurality and which are IRV. It It provides the functionality to 'un target' a contest being audited, and change the reasons why a contest is being audited.

DoSDashboardRefresh:

- This endpoints creates a *DoSDashboardRefreshResponse* which contains data about all audits being undertaken to be used to refresh the DoS Dashboard. This data contains audit ASM state, the reasons why audits are being undertaken, estimated and optimistic sample counts, refresh responses for each county, and which contests are being hand counted. [See json.DoSDashboardRefreshResponse and json.CountyDashboardRefreshResponse].

CountyDashboardRefresh:

- This endpoint creates a *CountyRefreshResponse* which contains data about all audits being undertaken to be used to refresh a County Dashboard. This includes data for both Plurality and IRV audits.

- A *CountyRefreshResponse* contains data that includes the contests under audit, CVRs, ballots audited, discrepancies, disagreements, and round data (as *Round* objects).

CVRToAuditList:

- Once audit rounds are under way, this endpoint requests and returns the list of ballots to audit from the *ComparisonAuditController*. When IRV contests are being audited, this list includes the ballots being audited for those contests as well.

ACVRUpload:

- This endpoint provides uploaded audited CVRs to the *ComparisonAuditController*. This is a relevant task for all kinds of audit (Plurality and IRV).

Note on IRV integration. If the audit ballot contains IRV rankings that form an invalid vote, auditors should simply enter exactly what they see. The system records both the raw data and the way in which the system interpreted the entered rankings to form a valid vote. See Section 4.6 for details.

The UI correspondences of some of the most important endpoints are given in Appendix A.

3.9   The State Machine

Location: corla.asm

The code files ASMState.java, ASMEvent.java, and ASMTransitionFunction.java detail the states for the DoS Dashboard ASM, the ASMs for each County Dashboard, and the ASMs for each Audit Board, the transition function over those states, and the events that cause those transitions. The files AuditBoard-DashboardASM.java, CountyDashboardASM.java, and DoSDashboardASM.java identify the intial and set of final states for those entities. Through examination of the codebase, it becomes apparent that states in each of the following ASMs are 'connected' – i.e., events raised in connection with one are associated with events raised in others. (By 'raised', we mean that the events are 'executed', potentially resulting in a transition between states in the ASM).

Figure 3.1 depicts the ASM for the DoS Dashboard. The events are raised as described below:

PARTIAL_AUDIT_INFO_EVENT
> This event is raised when any of the following pieces of information are provided and, after providing that piece of information, at least one of the remaining pieces have not yet been provided:
>
> - Contest names (through the endpoint *SetContestNames*)
>
> - Random seed (through the endpoint *SetRandomSeed*)
>
> - Contests to audit (through the endpoint *SelectContestsForAudit*)
>
> - Risk limit (through the endpoint *RiskLimitForComparisonAudit*)

COMPLETE_AUDIT_INFO_EVENT
> This event is raised when any of the following pieces of information are set and, after providing that piece of information, all remaining pieces (including the risk limit) have been provided:
>
> - Contest names (through the endpoint *SetContestNames*)
>
> - Random seed (through the endpoint *SetRandomSeed*)
>
> - Contests to audit (through the endpoint *SelectContestsForAudit*)

> Note that one of the above events (PARTIAL_AUDIT_INFO_EVENT and COMPLETE_AUDIT_INFO_EVENT) are returned by the *UpdateAuditInfo* endpoint depending on whether all or only some of the audit information has been set.

DOS_START_ROUND_EVENT
> This event is raised when the *StartAuditRound* endpoint is called, after checking that the DoS Dashboard ASM is in the COMPLETE_AUDIT_INFO_SET state and initialising audit data structures (if not done already), but before entering the main logic of the endpoint (i.e., calling the *startRound* method that performs ballot selection, and monitors the state of the County Dashboard and Audit/Audit Board ASMs).

DOS_AUDIT_COMPLETE_EVENT

This event is raised in the *SignOffAuditRound* endpoint, when the method *notifyAuditCompleteFor-DoS* is called *and* the audits in all counties have completed. (Note that the endpoint is accessed when a County signs off on an audit round. This method is called when all of the counties audits have completed).

Figure 3.2 depicts the ASM for a County Dashboard. The events are raised as described below:

IMPORT_BALLOT_MANIFEST_EVENT

This event is raised by the *BallotManifestImport* endpoint.

IMPORT_CVRS_EVENT

This event is raised by the *CVRExportImport* endpoint.

DELETE_BALLOT_MANIFEST_EVENT

This event is raised by the *DeleteFileController* in the method *reinitializeCBD* (if there are CVRs but no ballot manifest file). This method is only called in one place in the code (in the method *resetDashboards* in *DeleteFileController*, which in turn is called once in the code by the method *deleteFile* in *DeleteFileController*, which in turn is called in the *DeleteFile* endpoint body. So, this event will be raised whenever the *DeleteFile* endpoint is accessed and there are CVRs but no ballot manifest file. The ASM will be reset to its initial state if either the CVRs or ballot manifest is deleted and the ASM is in either the BALLOT_MANIFEST_OK or CVRS_OK state. These transitions are not explicitly defined in *ASMTransitionFunction* but are possible in the code.

DELETE_CVRS_EVENT

This event is raised by the *DeleteFileController* in the method *reinitializeCBD* (if there are no CVRs but there exists ballot manifest file). Again, the ASM state will revent to its initial state if it is in the state CVRS_OK and the CVRs are removed.

CVR_IMPORT_SUCCESS_EVENT

This event is raised in the method *success* located in the *ImportFileController*. This method is called from *runOnThread* in the event that parsing of the uploaded file was successful, which is called from *run*, in *ImportFileController*.

CVR_IMPORT_FAILURE_EVENT

This event is raised in the method *error* located in the *ImportFileController*. This method is called from *runOnThread* in the event that parsing of the uploaded file was not successful, which is called from *run*, in *ImportFileController*.

COUNTY_START_AUDIT_EVENT

This event is raised in the *StartAuditRound* endpoint in the method *initializeCountyDashboard*, provided the County Dashboard ASM is in the state BALLOT_MANIFEST_AND_CVRS_OK. This method

is called from the method *initializeAuditData*, which in turn is called from the endpoint body on the start of the first round of auditing.

COUNTY_AUDIT_COMPLETE_EVENT

This event is raised in both the *SignOffAuditRound* and *StartAuditRound* endpoints. In the former, the event is raised in the method *notifyAuditCompleteForDoS* which is itself called from the endpoint body with respect to a given County Dashboard. It is also raised in the method *markCountyAsDone* which is called from *notifyRoundCompleteForDoS* when all audits in all counties except that identified in the input argument have finished. (The event is raised for the identified county in *notifyAuditCompleteForDoS*). In *StartAuditRound*, the event is raised when: all audits are complete for the County and there are no disagreements; and where there are no contests to audit for the County.

Figure 3.3 depicts the ASM for an Audit Board (County). We believe there is one Audit Board ASM for each County. The events are raised as described below:

NO_CONTESTS_TO_AUDIT

This event is raised in the *StartAuditRound* endpoint (in the method *initializeCountyDashboard* in the context where the County has not uploaded either a ballot manifest or CVRs, i.e, they have missed the file upload deadline). This event is also raised in the *StartAuditRound* endpoint body if the set of comparison audits for the County is empty.

REPORT_MARKINGS_EVENT

This event is raised in the body of the *ACVRUpload* endpoint in the context where an ACVR has been uploaded by the County, it does not represent a re-audited ballot, and there are further ballots remaining in the current audit round (for which ACVRs need to be uploaded). Otherwise, the ROUND_COMPLETE_EVENT is raised.

REPORT_BALLOT_NOT_FOUND_EVENT

This event is raised in the body of the *BallotNotFound* endpoint in the context where a ballot that auditors were requested to collect was not found (a Phantom Ballot is created for these), and there are further ballots remaining in the current audit round (for which ACVRs need to be uploaded). Otherwise, the ROUND_COMPLETE_EVENT is raised.

SUBMIT_AUDIT_INVESTIGATION_REPORT_EVENT

This event is raised in the *AuditInvestigationReport* endpoint. This endpoint is accessed when an 'audit investigation report' is submitted. An *AuditInvestigationReportInfo* object (containing a report as a String) is created and added to a list of such objects in the CountyDashboard.

SUBMIT_INTERMEDIATE_AUDIT_REPORT_EVENT

This event is raised in the *IntermediateAuditReport* endpoint. This endpoint is accessed when
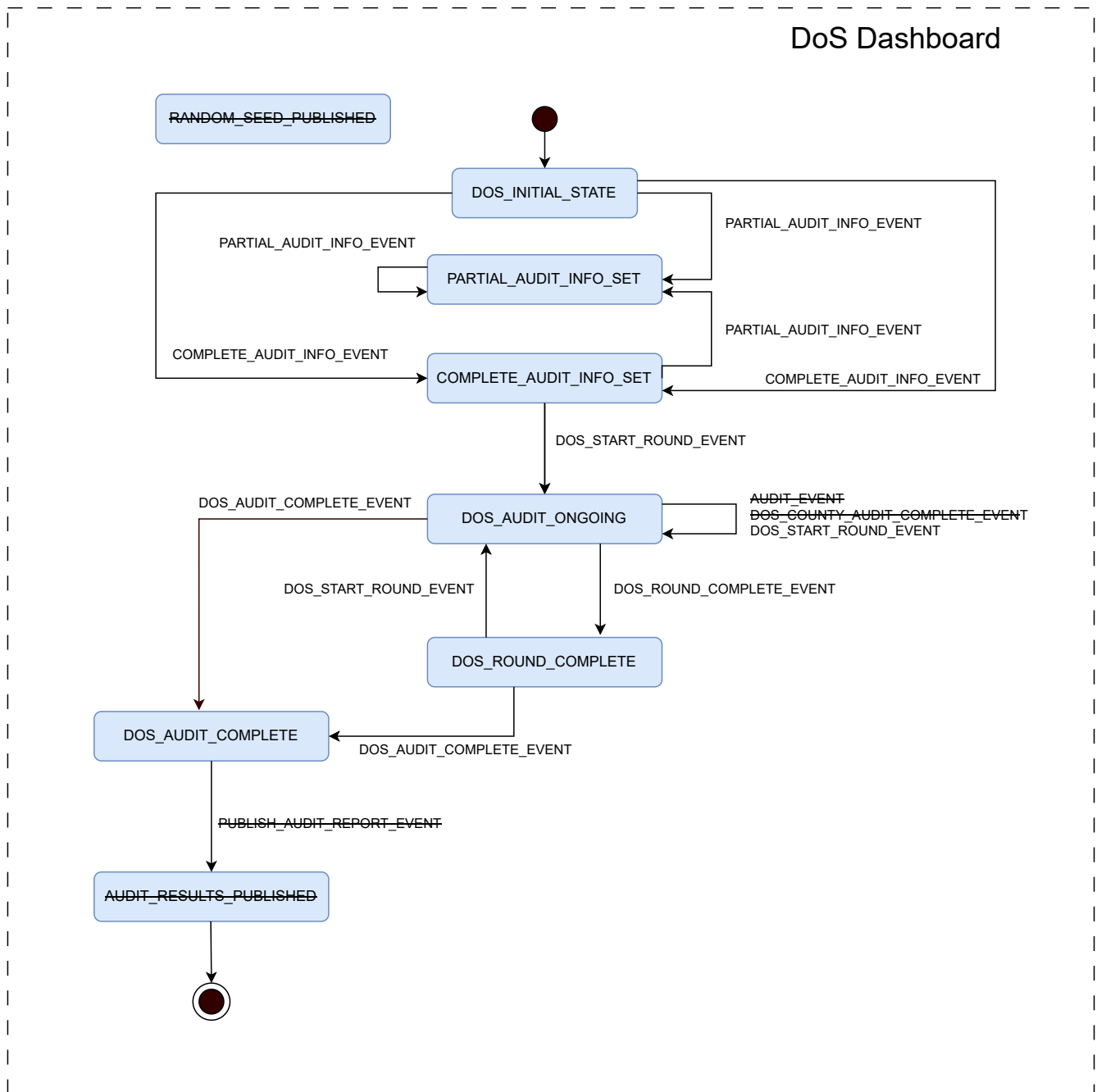
Figure 3.1: ASM for the DoS Dashboard (as extracted from the information in corla.asm). Crossed out states and events are those that are defined in the code but not used.

an 'intermediate audit report' is submitted. An *IntermediateAuditReportInfo* object (containing a report as a String) is created and added to a list of such objects in the CountyDashboard.
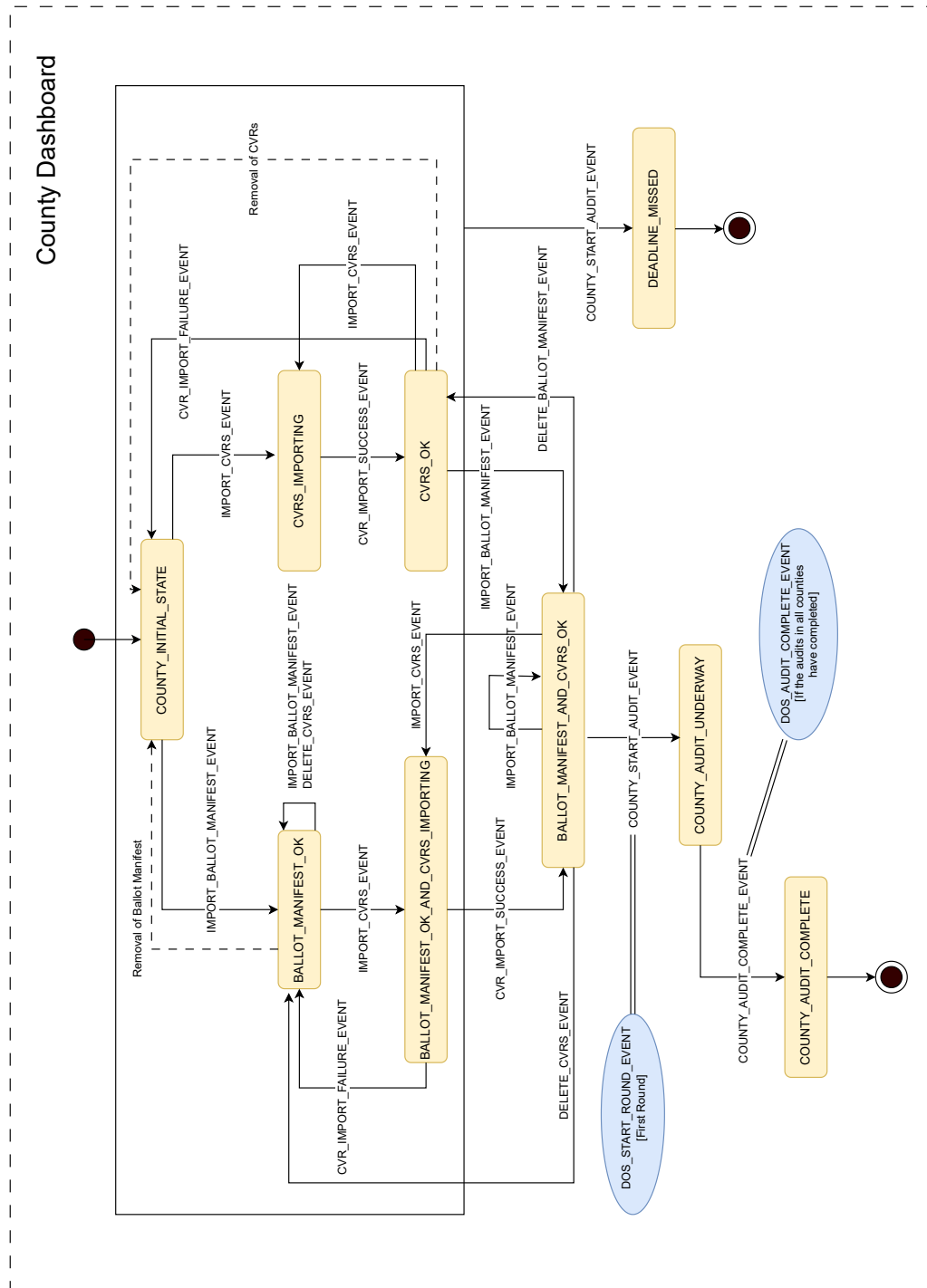
Figure 3.2: ASM for a County Dashboard (as extracted from the information in corla.asm). The dashed lines indicate transitions that are made in the code but not present in the ASM's transition function. The circled elements represent events from the DoS Dashboard ASM that are linked with events in a County Dashboard ASM (under certain conditions).
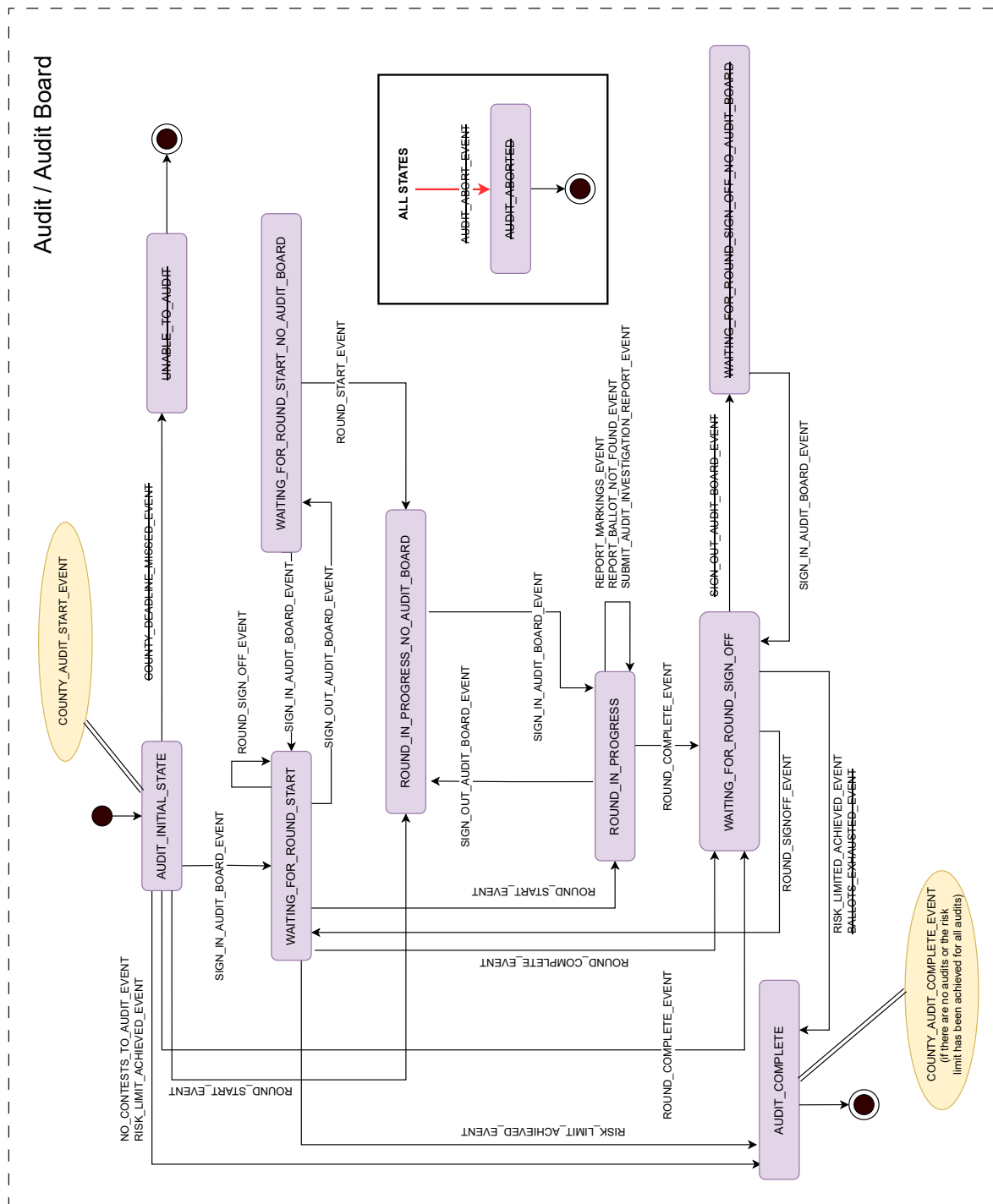
Figure 3.3: ASM for the audits for a County, as extracted from the information in corla.asm. Crossed out states and events are those that are defined in the code but not used. The circled elements represent events from the CountyDashboard ASM that are linked with events/states in the Audit/AuditBoard ASM (under certain conditions).

SIGN_IN_AUDIT_BOARD_EVENT

This event is raised in the body of the *AuditBoardSignIn* endpoint (provided the audit board has at least a quorum of members, and the ASM is in one of the states AUDIT_INITIAL_STATE, WAITING_FOR_ROUND_START_NO_AUDIT_BOARD, ROUND_IN_PROGRESS_NO_AUDIT_BOARD, WAITING_FOR_ROUND_SIGN_OFF_NO_AUDIT_BOARD). Note that we don't believe the ASM can be in the state WAITING_FOR_ROUND_SIGN_OFF_NO_AUDIT_BOARD.

ROUND_START_EVENT

This event is raised in the body of the *StartAuditRound* endpoint, after checks have been made to ascertain whether the county has audits yet to complete, and ballot selection has taken place.

ROUND_COMPLETE_EVENT

This event is raised in a number of situations:

- In the *ACVRUpload* endpoint if an ACVR was uploaded (not a re-audited ballot) and it was the last ballot to process.

- In the *BallotNotFound* endpoint, if the last ballot to process could not be found (a PHANTOM_BALLOT is created in its place).

- In the body of the *SignOffAuditRound* endpoint, provided all audit boards for the county have signed off, and the current ASM state is ROUND_IN_PROGRESS.

- In the *startRound* method of the *StartAuditRound* endpoint, in the context where ballot selection has been performed for the County and round, but the ballot sequence is empty (the county has no ballots to collect). (Perhaps this occurs when there are disagreements preventing the County Dashboard ASM from moving into the COUNTY_AUDIT_COMPLETE state?)

ROUND_SIGN_OFF_EVENT

This event is raised in the body of the *SignOffAuditRound* endpoint if:

- All audit boards for the County have signed off the round.

If it is the case that all audits for the County are complete when the *SignOffAuditRound* endpoint is called, the *notifyAuditCompleteForDoS* and *notifyRoundCompleteforDoS* methods will be called, which will trigger the County Dashboard ASM's COUNTY_AUDIT_COMPLETE_EVENT and may trigger the DoS Dashboard ASM's DOS_AUDIT_COMPLETE_EVENT (if all audits for all counties have completed).

3.10   Canonicalization

Sometimes different counties may write contest and candidate names differently, even if they are intended to be the same. For example, different punctuation, capitalization or abbreviations might look obviously the same to a human, but are not matched as strings. For this reason, colorado-rla requires CDOS to upload a canonical list of contest and choice names. After the counties have uploaded their data files, an endpoint called *SetContestNames* allows the CDOS dashboard to specify which uploaded contest and choice names match which ones from the canonical list. These are then rewritten to match.

SetContestNames::changeNames does four main things. The first updates contest names, the other three change choice names (i.e. generally candidate names).

1. It calls *Contest::updateChoiceName*, which simply resets the string if it matches the old one.

2. It calls *CastVoteRecordQueries::updateCVRContestInfos*. This identifies all relevant CVRs from the database and searches their choice strings for complete quote-delimited matches with the old choice—if found, it is replaced with the new choice. (For example, if the old choice is Ali and the new choice is Bob, it will replace "Ali" with "Bob" but will not replace "Alice" with "Bobce".) This may take a long time.

3. It calls *CountyContestResult::updateChoiceName*, which replaces the vote total record with one with the new name. These new tallies are then updated in the database.

Because IRV ballots are stored as ordered lists of candidate names, rather than with explicit ranks, the canonicalization code works for IRV ballots without needing any changes. This is described in Section 4.4.

Sampled ballots uploaded by the audit boards are already canonicalized. That is, they have contest and choice names that match the DOS-specified canonical choice and contest names.

Chapter 4

Implementation Details

4.1 RAIRE Microservice

The RAIRE microservice consists of two main components.

- The *actual RAIRE assertion-generation* is independent of any of the details of Colorado elections. It simply inputs a list of IRV ranks, along with metadata such as candidate names and intended audit method, and returns a collection of assertions.

  We have two implementations with the same API:

  - Java: https://github.com/DemocracyDevelopers/raire-java,

  - Rust: https://github.com/DemocracyDevelopers/raire-rs.

- The *raire-service* is a Springboot microservice that uses *raire-java* as a library. It connects the details of Colorado RLAs with the general implementation of the RAIRE assertion generation. It receives requests by contest name from *colorado-rla* and interacts with the database to return or store the result, as required.

  The implementation of the RAIRE microservice is available at:
  https://github.com/DemocracyDevelopers/raire-service/.

  Instructions for building, testing and running it are in the README. The API specification is in the repository at *raire_service-openapi.yaml*.

  See Section 5 for details of testing for this component.

4.1.1   raire-java Assertion Generation

function: raire-java::RaireResult   The Java (and Rust) implementations of the RAIRE assertion generation follow the main algorithmic idea described in the Guide to RAIRE, Part 2, with some optimizations and extra features designed to enhance error handling. They have the same API, described in the RAIRE github repository and reproduced here.

Input: The input is a description of IRV election data, consisting of JSON as in the following example:

```
{
  "metadata": {
    "candidates": ["Alice", "Bob", "Chuan","Diego" ],
    "note" : "Anything can go in the metadata section. Candidates names are used below
              if present. "
  },
  "num_candidates": 4,
  "votes": [
    { "n": 5000, "prefs": [ 2, 1, 0 ] },
    { "n": 1000, "prefs": [ 1, 2, 3 ] },
    { "n": 1500, "prefs": [ 3, 0 ] },
    { "n": 4000, "prefs": [ 0, 3 ] },
    { "n": 2000, "prefs": [ 3 ]  }
  ],
  "audit": { "type": "OneOnMargin", "total_auditable_ballots": 13500  },
  "time_limit_seconds": 10
}
```

In the *raire-service*, *raire-java* is used as a library and the data simply sent as a Java object that is input to the *RaireResult* function.

Each vote record consists of a count, followed by a list of integer ranks in order (first rank first). The numbers refer to the candidates in the "candidates" metadata list, starting from 0. In the example, the first record of votes states that there were 5000 votes that listed Chuan as first rank, Bob as second rank, and Alice as third rank.

The *total_auditable_ballots* field is used to compute the estimated difficulty of each assertion, assuming that it describes the number of ballots in the relevant universe. If omitted, it defaults to the sum of the entered votes. (This default is usually *not* appropriate for Colorado because super-simple uses all the votes cast in the county.)

The *time_limit_seconds* field gives RAIRE an upper limit on the amount of time it can spend trying to generate assertions. It returns an error if it did not find a sufficient set of assertions within that time.

Note this is *elapsed* time, not compute time—if RAIRE has to share resources with other processes, it is less likely to find solutions within a given time.

Output: The output contains two fields: *metadata*, which is a repeat of the input metadata, and *solution*, which is either an error or a list of assertions in JSON, together with some auxiliary data. The following example is also taken from the *raire-rs* README. Error outputs and their handling by the service are described in Section 4.1.2. Once RAIRE finds assertions, it will apply internal algorithms for filtering out those that are *redundant*. This process is referred to as *trimming*. The *solution* field is as follows.

```
'assertions' : an array of assertions. Each of these is an assertion object
    'assertion' : an object containing fields
    'type' : either the string 'NEN' or 'NEB' specifying what type of assertion it is.
        'NEB' (Not Eliminated Before) means that the 'winner' always beats the 'loser'.
        'NEN' (Not Eliminated Next) means that the 'winner' beats the 'loser' at the
            point where exactly the 'continuing' candidates are remaining. In particular,
            it means that the 'winner' is not eliminated at that exact point.
    'winner' : A candidate index
    'loser' : A candidate index.
    'continuing' : Only present if 'type' is 'NEN'. An array of candidate indices.
    'difficulty' : a number indicating the difficulty of the assertion.
    'margin' : an integer indicating the difference in the tallies associated with the
            winner and loser.

'difficulty' : a number indicating the difficulty of the audit.
'margin' : an integer indicating the smallest margin of the audit. This is the minimum
        of the margins in the assertions array.
'winner' : The index of the candidate who won - an integer between '0' and 'num_candidates-1'.
'num_candidates' : The number of candidates (an integer).
'warning_trim_timed_out' : If present (and true), then the algorithm successfully found
        some assertions but was unable to do the desired trimming in the time limit
        provided. Instead the untrimmed assertions are returned. Some of them may be
        redundant.

'time_to_determine_winners', 'time_to_find_assertions', and 'time_to_trim_assertions' :
    Objects describing how long each stage of the algorithm took. Fields are:
        'seconds' : The number of seconds taken at this stage.
        'work' : An integer indicating the number of steps taken in this stage. For
                finding winners, it is states in the elimination order. For finding
                assertions, it is the number of elements passing through the priority
                queue. For trimming, it is the number of nodes of the tree searched
                (some may be searched twice).
```

Note that the overall margin is the minimum margin of any assertion in the array. For ballot-level comparison audits, the overall difficulty is the same as the maximum difficulty of any assertion.[1]

Constraints: Assertions may be regenerated any time before *COMPLETE_AUDIT_INFO_SET* is true, but it is the responsibility of *colorado-rla* to enforce this constraint—see Subsection 4.2.1. Newly generated assertions will simply replace older assertions for the same contest.

4.1.2   RAIRE service assertion generation

Endpoint: raire-service::raire/generate-assertions

Input: The specification of one contest as a *GenerateAssertionsRequest*, which includes:

- Contest name (string)

- Total auditable ballots (int)

- Time allowed (s) (double)

- Candidates (List<String>)

Output: A *GenerateAssertionsResponse*, which includes:

- Contest name (string)

- response: either OK, or an error.

Database Interaction: The (ContestID, CountyID) pairs are retrieved from the Contest table (by looking up the contest by name). Cast Vote Records are then retrieved from the database using the given (ContestID, CountyID) pairs. If assertion generation is successful, the assertions are stored in the database.

Translation to RAIRE API call   The *totalAuditableBallots* field is set to the size of the universe, as defined in Stark's "Super Simple Simultaneous Risk-Limiting Audits." This is the sum of the card counts of all counties involved in the contest. This is calculated by colorado-rla as the ballot count of the relevant contest—see Section 4.2.1 for details.

The audit type is set to *"OneOnMargin"*, which is appropriate for Colorado RLAs, because it characterises the estimated sample sizes for ballot-level comparison audits. If Colorado ever changed to a different auditing method (for example, ballot polling, though this is not recommended), the audit type sent to RAIRE would have to change.

---

[1]This is not always true for other audit types.

The *RAIRE microservice* also gathers together any different instances of the same rank list, in order to send them efficiently to RAIRE. For example, if there are 20 CVRs which all contain the same rank list, the *RAIRE microservice* will create a single record with "n: 20" and the appropriate list of ranks. It also converts the candidate names into candidate indexes (each one being the position of that candidate in the *candidates* input list).

Errors

RAIRE produces several possible errors, which need to be appropriately handled by the assertion generation endpoint.

Errors for user feedback The following indicate that the voting data is hard to understand. The election may be a tie, or it may be very complex and close to a tie. They are returned by *raire-service* unchanged.

- **TiedWinners:** the election is a tie, and therefore definitely not auditable. This error also outputs the complete list of tied winners.

- **TimeoutFindingAssertions:** which also returns an estimated difficulty at the time of stopping. If the difficulty is infinite, it means that no sufficient set of assertions was found. If it is finite, the audit can proceed, and the assertions are valid, but they may not be efficient to audit.

- **TimeoutTrimmingAssertions:** the audit can continue, but may have redundant assertions. This happens if *raire-java* returned the **warning_trim_timed_out** flag, which is true if the assertion-trimming phase timed out. The assertions are valid, and are efficient to audit, but may contain a large number of redundant assertions. Rerunning the trimming process is advised. **TimeoutTrimmingAssertions** is returned as an error code, though there is also a valid GenerateAssertionsResponse (with the winner) returned.

- **TimeoutCheckingWinner:** the election is either tied, or is very complex and cannot be distinguished from a tie.

- **CouldNotRuleOut:** a certain elimination sequence could not be ruled out, despite having a different winner. This also implies that the outcome was a tie and the contest inherently cannot be audited.

- **InvalidCandidateNumber:** arises if the candidate list entered in the request has the right number of names but wrong names relative to what is stored in the database.

These errors should generally be returned to the user, because the user can act upon them—in some cases, by deciding that the only way to verify that contest is by a full hand count. For the timeouts, the user may decide to rerun the algorithm with more time.

Indications of invalid input

The following indicate programming errors (in either raire-service or raire-java) and are summarised as INTERNAL_ERROR.

- InvalidTimeout: the given timeout was not a positive integer. This should be caught before being sent to raire.

- InternalErrorDidntRuleOutLoser:

- InternalErrorRuledOutWinner:

- InternalErrorTrimming: for some reason, trimming assertions did not work.

- InvalidNumberOfCandidates: an empty candidate list or invalid candidate count was sent to raire.

- WrongWinner: the given winner is not consistent with the CVRs. This should never happen in Colorado because an explicit winner is not provided.

INTERNAL_ERROR is also returned if database storage fails.

The *raire-service* can also return:

- BAD_REQUEST: Input validation errors, for example the contest does not exist or is not IRV, or the candidate list is null or the timeout is non-positive.

- WRONG_CANDIDATE_NAMES: if *raire-java* returned the InvalidCandidateNumber error. The request was valid, but at least one vote contained a candidate who was not in the request's list of expected candidates. (The opposite is not an error: a valid candidate might get no votes.).

- NO_VOTES_IN_DATABASE: if there are no votes in the database for the (validly requested IRV) contest.

4.1.3   RAIRE service assertion download

The raire service can export assertions in either json or CSV. The json form is designed to be compatible with a raire assertion visualizer. The csv is intended to be read in a spreadsheet. The raw data is the same, but the csv contains a summary of which assertions meet certain extrema such as the lowest margin or highest estimated samples to audit.

Endpoint: raire-service::raire/get-assertions-json   This endpoint exports assertions in json format.

Input: a *GetAssertionsRequest,* which includes:

- contestName (String)

- candidates (List<String>)

- riskLimit (BigDecimal)

- totalAuditableBallots.

These are used to generate the metadata for the returned assertions. TotalAuditableBallots is the total number of ballots in the universe (which is generally larger than the number of ballots that contain the contest).

Output: A *GetAssertionsResponse,* which is either assertions for the contest, updated with data on audit progress, or an error.

The output is the same as the RAIRE assertion generation step, except that each assertion has a *status* field to describe the current risk measurement, as a *BigDecimal*.[2] The *risk_limit* field contains a single float stating the risk limit (typically 0.03 in Colorado). It is then easy to calculate whether the risk limit has been met for any given assertion. The *solution* field is shown in Subsection 4.1.1.

An example, for a contest with two assertions, is given below.

```
{
  "metadata": {
    "candidates": ["Alice","Bob","Chuan","Diego"],
    "contest":"One NEN NEB Assertion Contest",
    "risk_limit": 0.05,
    "total_auditable_ballots":42353
  },
  "solution": {
    "Ok": {
      "assertions": [
        {"assertion":{"type":"NEB","winner":2,"loser":0},"difficulty":0.1,
          "margin":112,"status":{"risk":0.08}},
        {"assertion":{"type":"NEN","winner":2,"loser":1,"continuing":[0,1,2]},"difficulty":3.17,
          "margin":560,"status":{"risk":0.70}}
      ],
      "difficulty":3.17,
      "margin":112,
      "winner":2
    }
  }
}
```

---

[2]*Also, winner* and *number_of_candidates* are omitted.

These assertions are intended to be made public. For example, they could be exported to Colorado's public audit website. Interested citizens could examine and visualize them with the RAIRE assertion explainer.

Endpoint: raire-service::raire/get-assertions-csv   This exports assertions in csv format. Each assertion's row has the same information as the json above, with some extra data derived from *colorado-rla:* the estimated samples to audit, optimistic samples to audit, discrepancies and diluted margin. Each assertion is given an index starting at 1 (and incrementing by 1), which is guaranteed to be consistent for different calls to the endpoint. The csv file also contains a summary stating the extremal values for interesting statistics, and listing the indices of the assertions that attain those statistics. The statistics are: margin, diluted margin, difficulty (as estimated by raire), risk (as currently estimated by colorado-rla), optimistic samples to audit and estimated samples to audit.

Input: a *GetAssertionsRequest,* which includes:

- contestName (String)

- candidates (List<String>)

- riskLimit (BigDecimal)

Output: a CSV file containing:

- a preface with the contest name, candidate list, winner, total auditable ballots, and risk limit;

- a statistics summary listing the extremal values for margin, diluted margin, difficulty (as estimated by raire), risk (as currently estimated by colorado-rla), optimistic samples to audit and estimated samples to audit. This also lists the indices of all assertions that attain the extremum (max or min, as appropriate);

- data for each assertion, as a csv row containing: ID, type, winner, loser, assumed continuing, difficulty, margin, diluted margin, risk, estimated samples to audit, optimistic samples to audit, two vote over count, one vote over count, other discrepancy count, one vote under count, two vote under count.

The database interactions and errors are the same for both endpoints.

Database interaction: Retrieves Assertions from the database.

Errors: BAD_REQUEST, INTERNAL_ERROR, WRONG_CANDIDATE_NAMES all have the same meaning as for the *generate-assertions* endpoint.

NO_ASSERTIONS_PRESENT means that the request was valid but no assertions have been generated for the contest. This may be because assertion generation has not (yet) been requested, or because it failed for some reason such as tied winners.

4.1.4   RAIRE service hello

A super-simple endpoint that does nothing except return an HTTP OK. This allows colorado-rla to poll the raire service to check that it is reachable.

| Contest name | Lots of assertions with ties Contest |
| Candidates | Alice, Bob, Chuan, Diego |

Extreme item	Value	Assertion IDs
Margin	220	2, 5, 6
Diluted margin	0.22	2, 5, 6
Raire difficulty	3.1	3
Current risk	0.23	2, 3
Optimistic	910	4
Estimated	430	2, 5

ID	Type	Winner	Loser	Assumed continuing	Difficulty	Margin	Diluted Margin	Risk	Estimated	Optimistic
1	NEB	Alice	Bob		2.1	320	0.32	0.04	110	100
2	NEB	Chuan	Bob		1.1	220	0.22	0.23	430	200
3	NEB	Diego	Chuan		3.1	320	0.32	0.23	50	110
4	NEN	Alice	Bob	Alice, Bob, Chuan	2.0	420	0.42	0.04	320	910
5	NEN	Alice	Diego	Alice, Diego	1.1	220	0.22	0.07	430	210
6	NEN	Alice	Bob	Alice, Bob, Diego	1.2	220	0.22	0.04	400	110

Table 4.1: Example CSV export. "Estimated samples to audit" and "Optimistic samples to audit" are cut to "Estimated" and "Optimistic" in this table, but not in the real csv file.

Endpoint: raire-service::raire/hello   No inputs or errors.

Output: HTTP OK.

4.1.5   Parallelism and efficiency

Each call to the API will automatically execute in a separate thread, which should make parallelism easy.

4.2   New Endpoints in colorado-rla

4.2.1   Generate Assertions

Endpoint: colorado-rla::generate-assertions

Purpose: For IRV contests, identified by their description ("IRV") in the database, this endpoint will access the RAIRE microservice (Section 4.1) to generate assertions. The RAIRE microservice will store these assertions in the database. The database tables used to store assertions are described in Section 4.7.3. This endpoint will handle all error types produced by the RAIRE service.
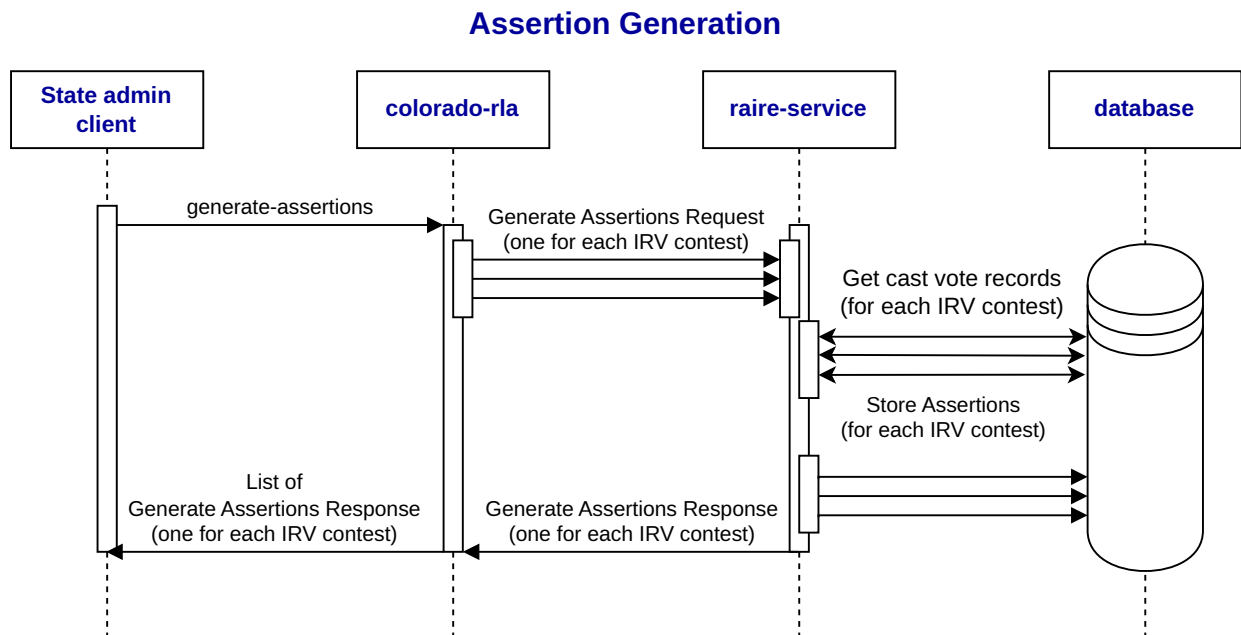
Assertion Generation

Figure 4.1: Sequence diagram for Assertion Generation. The data types are defined in 4.1.2.

Input: Two options:

- **None** It reads the list of contests from the database, finds all the IRV ones, and requests the raire-service to generate assertions for all of them using a default time limit.

- **Contest or timeLimit (seconds)** It makes a single request to the raire-service to generate assertions for the specified contest or time limit.

  Example: */generate-assertions?contest="Boulder Mayoral",timeLimitSeconds=5*

Output: Either the list of *GenerateAssertionsResponse*s from the *raire-service*, for all IRV contests, or an error. Note that some individual responses for particular contests in the list may be errors (for example, TIED_WINNERS) while the overall computation is not an error. If a single contest is specified, but is not an IRV contest, that is an error.

Action: Calls the *raire-service::generation-assertions* endpoint for each IRV contest in the database, or (in the case of the specific contest request) for the specified contest.

Considerations:

- For opportunistic reporting of discrepancies for all IRV contests, assertions are generated for all these contests, not just those being targeted for audit.

- When counties upload new CVR data for their contests, any previously generated assertions associated with contests in those counties will need to be regenerated. This is the responsibility of the CDOS user—it is not triggered automatically.

  If the assertions are not regenerated, the RLA remains valid—it will not confirm a wrong result except with probability bounded by the risk limit. However, it may fail to confirm a correct result because the old assertions may not be true of the new CVRs.

Constraints: Assertions may *not* be regenerated after 'COMPLETE_AUDIT_INFO_SET' becomes true. If the endpoint is hit after that time, it will return an error.

Errors:

- *Bad Data Contents* if the contest name or time limit is present but cannot be parsed as a valid parameter, for example if the contest name is blank or the time limit is negative.
- *Illegal Transition* if the endpoint is hit after 'COMPLETE_AUDIT_INFO_SET'.
- *Internal Server Error* for other errors related to processing data or contacting the *raire-server*.

Note that most of the reasons *raire-service* could return an error do *not* produce an error in the *colorado-rla::generate-assertions* endpoint—they simply result in a single-contest record of a failed assertion generation attempt (e.g. Tied Winners). These are recorded, and generally cause that contest to be 'NOT_AUDITABLE', but do not prevent the rest of the audit from proceeding (with other contests targeted).

Code for the Assertion Generation endpoint is available at: https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/endpoint/GenerateAssertions.java

4.2.2   Export Assertions

Endpoint: colorado-rla::export-assertions

Purpose: Provide a zip file containing a json or csv file for each contest that defines the assertions generated for that contest, along with associated metadata including contest name, candidate names, minimum margin and expected difficulty. The format of the json file is compatible with an assertion explainer.

Input: An optional *format* parameter, which may be either "json" or "csv". Defaults to json.
Example: /get-assertions?format=csv

Output: The assertions and associated metadata for each contest bundled into a zip file, or an error. Note that some individual responses for particular contests in the zip may be errors (for example, NO_ASSERTIONS_PRESENT) while the overall computation is not an error.

Action: Calls the *raire-service::get-assertions* endpoint for each contest, and returns the generated files in a zip file.
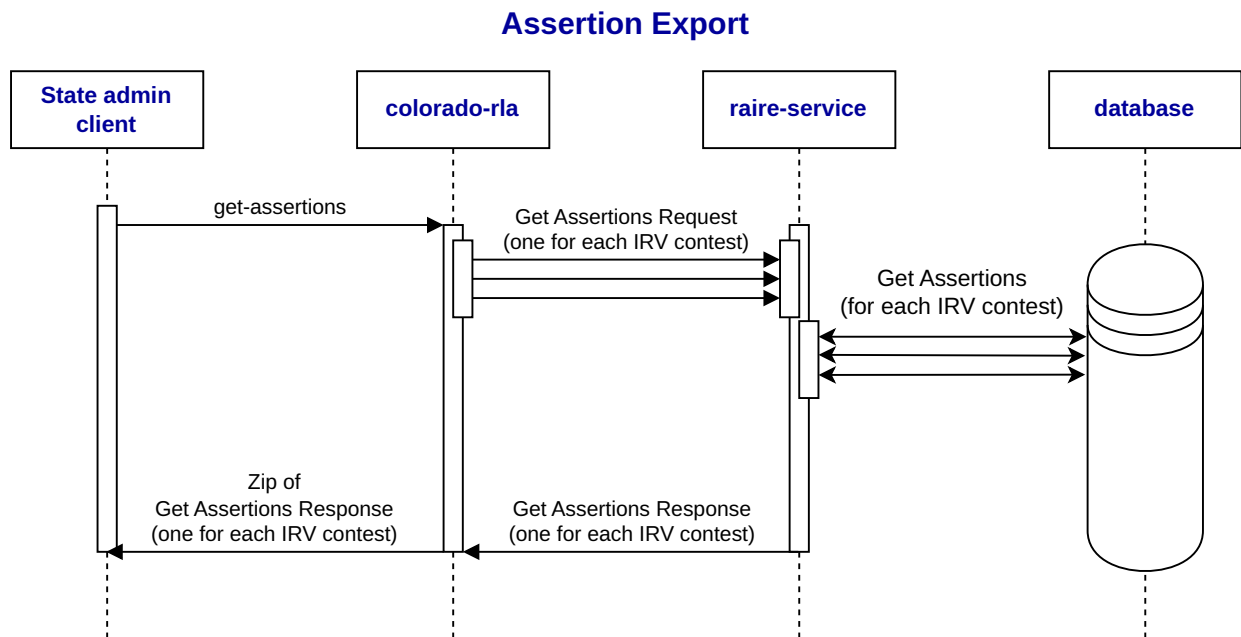
Assertion Export

Figure 4.2: Sequence diagram for Assertion Export. The data types are defined in 4.1.3.

Considerations: These assertions are intended for public display and are human-readable. The json form is compatible with an assertion explainer.

Code for the Assertion Export endpoint is available at: `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/endpoint/GetAssertions.java`

4.2.3 Sample Size Estimation

Endpoint: colorado-rla::estimate-sample-sizes

Purpose: Compute estimated sample sizes for each contest in the database, both Plurality and IRV, and export these sample sizes to a CSV file. We have designed this endpoint to avoid storing or persisting new data to the database.

Input: None.

Output: A CSV file is produced and automatically downloaded. The file contains the following data points for each contest:

- County name (or indication that contest is multi-jurisdictional);
- Contest name;
- Contest type (IRV or Plurality);

- Number of ballots cast;
- Total universe size;
- Diluted margin;
- Estimated sample size (assuming no discrepancies will arise).

For IRV contests, each associated assertion has its own diluted margin. This CSV file reports the smallest diluted margin assigned to an assertion for a given IRV contest.

Design: This endpoint implements the following steps:

1. Create *ContestResult* objects for each contest in the database. This is achieved using the existing *Contest-Counter* controller. It collects *CountyContestResult* objects for each distinct contest name, and integrates them into a single *ContestResult*, tabulating county-level vote totals where appropriate. For Plurality contests, these vote totals are used for computing diluted margins. For IRV contests, vote tabulation within colorado-rla is unnecessary as margins are computed by the RAIRE microservice when forming assertions. Diluted margin computation then only requires division by the total number of ballots in the contest's universe. The existing *ContestResult* and *CountyContestResult* classes are described in Section 3.1.1. The existing *ContestCounter* controller is described in Section 3.4.

2. Create *ComparisonAudit* objects for each Plurality contest and *IRVComparisonAudit* objects for each IRV contest using the *ContestResult*s constructed in Step 1. As *IRVComparisonAudit* is a subclass of *ComparisonAudit*, the existing colorado-rla system can interact with it as if it was a *ComparisonAudit*. For example, through a list: *List<ComparisonAudit>*. This means that in each location where colorado-rla iterates over a set of *ComparisonAudit*, the existing code will work for IRV just as it does now for Plurality, without required changes.

3. Call each comparison audit's *estimatedSamplesToAudit()* method to compute sample sizes.

4. Export sample size estimates, along with additional data points for each contest, to a CSV file.

Considerations:

- This endpoint does *not* use manifests when calculating the universe size or the estimated sample sizes. Instead, it simply counts the total number of CVRs in each county. If the manifests, which must be uploaded later for the audit step, list a larger number of ballots than the CVRs, then this may lead to significantly different sample sizes.

- For a single contest involving multiple counties, CDOS will sometimes take one of these counties and audit the contest at a single-county level. If this is achieved by creating a single-county version of a state-wide contest, with a distinct name, no changes to this design will be required to support this functionality.

- Sample size estimates will only be generated for IRV contests with generated assertions. If assertions have been generated (via the Generate Assertions endpoint, Section 4.2.1) for only a subset of IRV contests, then sample size estimates will only be generated for those contests (in addition to all Plurality contests in the database).

Code for the new sample-size estimation endpoint is available at: `https://github.com/DemocracyDevelopers/`
`colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/`
`endpoint/EstimateSampleSizes.java`

4.2.4   IRV Specific Reports/Assertion Export

Traditional reports generated by the existing system can be generated for IRV contests by virtue of the fact that *IRVComparisonAudit*s extend the *ComparisonAudit* class, and all data accessible through the *ComparisonAudit* interface is accessible for IRV audits.

The two new IRV-related reports that are generated within *colorado-rla* are *ranked_ballot_interpretation* and *summarize_IRV*. These are generated by adding appropriate queries into the existing collection of sql-generated reports at */src/main/resources/sql*.

4.3   New Classes

4.3.1   Contest Type (Enumeration)

A new enumeration has been added to the system to capture the different kinds of contest under audit: Plurality; and IRV. The *description* attribute of the *Contest* class has been used to store the contest's type.

4.3.2   Assertions

Three classes have been added to represent assertions:

- An abstract base class *Assertion*

  `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/`
  `main/java/au/org/democracydevelopers/corla/model/assertion/Assertion.java`

- A subclass of *Assertion*, *NEBAssertion*

  `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/`
  `main/java/au/org/democracydevelopers/corla/model/assertion/NEBAssertion.java`

- A subclass of *Assertion*, *NENAssertion*

  `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/`
  `main/java/au/org/democracydevelopers/corla/model/assertion/NENAssertion.java`

Assertion base class

Key Attributes:

- Id (Long)
- Contest name (String)

- Winner (String)

- Loser (String)

- Raw margin (int)

- Risk (BigDecimal)

- Diluted margin (double)

- Difficulty (double, as computed by RAIRE)

- Context: candidates assumed to be continuing (List<String>)

- Optimistic samples to audit (Integer)

- Estimated samples to audit (Integer)

- Two vote under count (Integer)

- One vote under count (Integer)

- One vote over count (Integer)

- Two vote over count (Integer)

- Count of discrepancies that are neither understatements or overstatements (Integer)

- Map between CVR ID and the discrepancy value associated with that CVR. We keep track of this information for each assertion as a single CVR can be involved in different discrepancies across the set of assertions for a contest (Map<Long,Integer>).

Key Methods:

- *public Integer computeOptimisticSamplesToAudit (BigDecimal riskLimit)*
  Computes the estimated sample size under the assumption that no further discrepancies will be encountered. This method will call *Audit.optimistic()* with appropriate inputs.

- *public Integer computeEstimatedSamplesToAudit (int auditedSampleCount)*
  Computes the estimated sample size under the assumption that discrepancies will continue to arise at their current rate. This method will apply a scaling factor to the current optimistic sample size estimate.

- *public OptionalInt computeDiscrepancy(final CastVoteRecord cvr, final CastVoteRecord auditedCVR)*
  Computes the over/understatement (or other discrepancy) represented by the specified CVR and ACVR. This method returns an optional int that, if present, indicates a discrepancy. There are 5 possible types of discrepancy: -1 and -2 indicate 1- and 2-vote understatements; 1 and 2 indicate 1- and 2- vote overstatements; and 0 indicates a discrepancy that does not count as either an under- or overstatement for the RLA algorithm, but nonetheless indicates a difference between ballot interpretations. When this method is called, the result will be returned and also recorded in the Assertion's map of CVR ID to discrepancy value. Table 4.2 lists the formulae used to compute discrepancies according to the properties of the CVR and audited ballot.

| | **Audited Ballot** | | | |
CVR	Phantom	Consensus: No Contest Present	Consensus: Yes Contest Present	Contest Not Present
Phantom	2	2	1 - audited ballot score	1
Contest Present	CVR score + 1	CVR score + 1	CVR score - audited ballot score	CVR score
Contest Not Present	1	1	- (audited ballot score)	–

Table 4.2: Discrepancy formulae used for a CVR and its matching paper ballot (audited ballot) in the context of a specific contest and assertion, in each combination of possibilities: phantom/not phantom; consensus on the audited ballot (yes/no); contest present/not present. Assertions are scored as per Appendix A in Part 2 of the Guide to RAIRE.

- *public boolean recordDiscrepancy(final CVRAuditInfo theRecord)*

  This method will look up the Assertion's internal record of discrepancies for the relevant CVR ID (extracted from the input *CVRAuditInfo*) to determine if it was involved in a discrepancy, and if so, it will increment the Assertion's internal counters. The reason we have two methods (*computeDiscrepancy()* and *recordDiscrepancy()*) is that this is how the colorado-rla system deals with discrepancies for Plurality audits. The type of discrepancy for a given CVR and audited ballot will be computed, and then the *recordDiscrepancy()* method called n times, where n denotes the number of times the CVR appears in the ballot sample. This method will return 'true' if the assertions internal discrepancy counts were incremented, and 'false' otherwise.

- *public boolean removeDiscrepancy(final CVRAuditInfo theRecord)*

  When a ballot is re-audited, any discrepancies associated with the CVR/audited ballot comparison for that ballot in a given contest are removed from that contest's audit. This method will look up the type of discrepancy associated with the record in the Assertion's internal map, and then decrement the relevant counter by 1. (Note that as with *recordDiscrepancy()*, this method is designed to be called n times if the relevant discrepancy has been recorded n times). This method will return 'true' if the assertions internal discrepancy counts were decremented, and 'false' otherwise.

- *protected abstract int score(final CVRContestInto info)*

  As described in the Guide to RAIRE, an audited ballot or CVR is given a score of -1, 0 or 1 with respect to a given assertion, based on the ranked choices on the ballot and CVR. We subtract the score given to the ballot from the score given to the CVR, resulting in an discrepancy value between -2 and 2. This is the method that will have a different implementation for the two different kinds of assertions involved in IRV audits.

The computation of estimated samples to audited is assisted by a private method:

private BigDecimal scalingFactor(int auditedSamplesInt, BigDecimal overstatements)

This method scales the suggested number of samples to audit according to the rate of overstatements seen thus far in the audit.

Discrepancy Computation

Table 4.2 lists the formulae used to compute discrepancies for each of the different configurations of CVR and audited ballot. For example, when both the CVR and audited ballot are Phantoms, the worst case discrepancy of

2 is assigned to the CVR/ACVR pair for any assertion. Where the contest to which the assertion belongs is not on the CVR, but is on the paper ballot, with consensus, then the discrepancy is equal to the negation of the audited ballot score (i.e., we treat the CVR score as 0). It is important to ensure that if the CVR and paper ballot have mismatching contests (i.e., a contest that should be on the CVR is not, or a contest that shouldn't be on the CVR is) can be detected as a discrepancy.

Assertion subclasses: NEBAssertion; NENAssertion

Each assertion subclass provides its own implementation of the base classes abstract methods. No additional attributes are defined within each subclass.

4.3.3   IRV Comparison Audits

The complexity of IRV audits is captured in a subclass of the existing class *ComparisonAudit*. This allows IRV audits to be seamlessly integrated into the existing colorado-rla system while at the same time, separating sample size estimation, discrepancy and risk computation for IRV into a single component.

IRVComparisonAudit

Source:

https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/model/IRVComparisonAudit.java.

Extends: ComparisonAudit

Key Attributes:

- Assertions (List<Assertion>)

Key Methods:

- (Constructor) *public IRVComparisonAudit(final ContestResult contestResult, final BigDecimal riskLimit, final AuditReason auditReason)*

  The base class's constructor will be called with the given *ContestResult*, risk limit, gamma value, and *AuditReason*.

  This constructor will collect all assertions stored in the database for this contest, and populate the assertions attribute. If no assertions have been generated for the contest, it will set the audit's status to *AuditStatus.NOT_AUDITABLE*. Otherwise, the audit's status will be set to *AuditStatus.NOT_STARTED*.

  The smallest diluted margin across all assertions for the contest will be assigned to the base class's diluted margin attribute. It will use the total number of ballots stored in the given *ContestResult* as the universe size for the contest.

  As per the design of the base class constructor, optimistic and estimated sample sizes will be computed.

- (Overridden) *protected void recalculateSamplesToAudit()*

  Recalculates the overall number of ballots to audit, setting the optimistic and estimated samples to audit attributes in the base class.

- (Overridden) *public OptionalInt computeDiscrepancy(final CastVoteRecord cvr, final CastVoteRecord audit-edCVR)*

  This method will call the discrepancy computation method of each of its assertions, recording discrepancies that arise within the assertions themselves. For the purposes of reporting, if the CVR and ACVR pair represents a discrepancy for any assertion, the maximum discrepancy type (a number between -2 and 2) will be recorded in the totals maintained in the base class.

- (Overridden) *public BigDecimal riskMeasurement()*

  A method to compute the current level of risk for the given contest will be added to *ComparisonAudit* and overridden in *IRVComparisonAudit* for IRV audits. In the current codebase, risk measurement for the purposes of reporting collects data from a *ComparisonAudit* object and passes it directly to a method in *Audit.math*. This is only suitable for Plurality, however. We have shifted risk measurement into *ComparisonAudit* so that each type of audit can interact with *Audit.math* in a way that is appropriate for the audit type.

- (Overridden) *public void recordDiscrepancy(final CVRAuditInfo the_record, final Integer theType)*

  As described in our discussion of the methods associated with our *Assertion* class, once the system calls the *ComparisonAudit*/*IRVComparisonAudit* method for computing the discrepancy (if any) between a CVR and audited ballot, it then *records* that discrepancy by calling *recordDiscrepancy()*. The IRV version of this method will call the *recordDiscrepancy()* method of each of its assertions.

- (Overridden) *public void removeDiscrepancy(final CVRAuditInfo theRecord, final int theType)*

  This method will call the *removeDiscrepancy()* method of each of its assertions.

4.4   Parsing and storage of (IRV) CVRs

Colorado-rla already incorporates an option for multiple candidate selections. This is used directly to store *valid* IRV ballots. It requires no change to the database schema.

Key idea: store valid IRV votes as an ordered list of candidate names, first rank first, second rank second, etc.

We have implemented parsing of Dominion IRV CVR fields, based on examples provided by CDOS. This includes automatic checking for validity (that is, no duplicated, repeated or skipped ranks) and warning the user if the CVR CSV contains invalid ranks.

Parser edits are available at: https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/us/freeandfair/corla/csv/DominionCVRExportParser.java

4.4.1   Parsing and storage details

Dominion CVRs store IRV votes by having a separate column for each candidate-rank pair. For example, if there are three candidates called Alice, Bob and Chuan (and no write-ins), the CVRs will have 9 columns for this contest:

`Alice(1), Bob(1), Chuan(1), Alice(2), Bob(2), Chuan(2), Alice(3), Bob(3), Chuan(3)`

The number of ranks is capped at 10, even if there are more than 10 candidates.

This allows for the expression of any pattern of selections that the voter may have made, including ranks that are not valid unambiguous IRV ranks. So IRV CVR storage needs three phases.

1. Parse the the CVR row as a (possibly invalid) collection of selections with their explicit rank in parentheses, e.g. "Alice(1), Bob(2), Chuan(3), Diego(3)."

2. Interpret the selections to the corresponding valid IRV rank list. The above example would be "Alice(1), Bob(2)."

3. Store the valid rank list as an ordered list of choices, where the ranks are not recorded explicitly but are encoded by position in the list. The above example would be "Alice, Bob."

The first step uses the existing colorado-rla code to do the parsing.

The second step requires specific code to interpret invalid ranks according to Colorado's statute (`https://www.sos.state.co.us/pubs/rule_making/CurrentRules/8CCR1505-1/Rule26.pdf`) and CDOS clarification rules. Code available at: `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/model/vote/IRVBallotInterpretation.java` and `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/model/vote/IRVChoices.java`. Duplicates are removed first (that is, when multiple ranks have been given to the same candidate) and subsequently removes every rank after a skipped or overvoted rank.

Write-ins

Existing write-in parsing code can be used almost unchanged, except that instead of searching only for the column "Write-in" it needs also to check for column headings of the form "Write-in(n)" for each rank n.

New Classes

IRVPreference:

`https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/model/vote/IRVPreference.java`

Stores a (name, rank) pair.

IRVChoices:

`https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/model/vote/IRVChoices.java`

Stores a list of (name, rank) pairs. Includes methods to check whether the rank list is valid (i.e. without duplicated, skipped or repeated ranks) and methods to apply Colorado's rules for dealing with such invalidities.

The *IRVChoices* class contains a method *getValidIntentAsOrderedList* which applies the rules in [Colorado Election Rules [8 CCR 1505-1]](#) to produce a valid IRV vote, and returns that valid IRV vote as an ordered list of candidate names, with the highest-preference candidate first.[3] This version of the IRV vote is stored in the database and used throughout the audit.

4.5   Ballot Interpretation Database storage

Key idea: the Audit Board enters the IRV ballot data exactly as they see it; the system records the valid interpretation, and the raw data if it differs from the interpretation.

Ballot interpretation for audited ballots is the same as ballot interpretation for initial CVRs. The audit board will be told to make a decision about what marks are on the ballot paper, but *not* to apply the interpretation rules for overvoted, repeated or skipped ranks. Instead, the audit system will store:

- the raw (possibly invalid) vote for a given IRV contest (for example, "Alice(1), Bob(2), Alice(3), Bob(3)"), and
- the valid interpretation, as an ordered list of candidate names (in this example, "Alice, Bob"),
- the contest ID,
- the CVR number,
- the imprinted ID,
- the record type (an uploaded CVR, an audit CVR, or an audit CVR that has been reaudited).

New classes

IRVBallotInterpretation:

https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/model/vote/IRVBallotInterpretation.java

This table stores a record of each invalid IRV ballot and its interpretation. This is useful for record-keeping. Only the valid interpretation is used in the audit.

4.6   Running the Audit

IRV comparison audits, with correct implementations of sample size estimation, discrepancy management, and risk computation, are treated the same as Plurality comparison audits in the audit logic of colorado-rla. Our design hides these complexities of IRV audits within a subclass of the Plurality *ComparisonAudit* class. As the correct constructor is used when creating comparison audit objects (*ComparisonAudit* for Plurality and *IRVComparisonAudit* for IRV), most audit logic that deals with *ComparisonAudit*s did not change with the introduction of IRV. This includes the logic for starting and signing off an audit round (*StartAuditRound* and *SignOffAuditRound*).

---

[3]The implementation runs the removal of choices after skipped ranks (Rule 26.7.2) last, in order to ensure the result is a valid ranked list. See the comment in the file for details.

The exception is in the upload of audited ballots by audit boards. The server handles this upload in the *ACVRUpload* endpoint. Currently, the uploaded ballot is provided to the endpoint in JSON that is used to form a *SubmittedAuditCVR* object. The information captured is: CVR ID; a *CastVoteRecord* object capturing the details of the audited ballot; a boolean indicating whether this is a re-audited ballot; a string comment; and the index of the audit board. The vote on this ballot will be described by a string of choices (with ranks). Internally, colorado-rla will represent IRV votes as a string of choices (*without* ranks). At this point in the upload, we will need to both interpret the vote to see if it is invalid, and if so convert it into its valid representation (see Section 4.5), and then convert what was entered by the auditor into the string of choices without ranks. It is important to have a record of exactly what the audit board said was on the ballot, and how the system interpreted this information. We did this by introducing a new database table to store this information, as described in Sections 4.5 and 4.7.

Discrepancy calculation then acts on the valid interpretation of each ballot—if the CVR and the audit ballot have the same ballot interpretation, there will be no discrepancy.

IRV ballot interpretation does not occur directly in the *ACVRUpload* endpoint, but rather where an IRV vote on an audited ballot is parsed. This is in the JSON adapter class for *CVRContestInfo* (*CVRContestInfoJsonAdapter.java*).

We implemented the following changes to this adapter class, and *CVRContestInfo*, to support the processing of IRV votes on audited ballots, and the storing of how the system interpreted those votes in the database.

- An IRV vote, as recorded by auditors for a given contest, is passed to the server as a list of choice names and their ranks (i.e., "Alice(1), Bob(2)"). It is necessary for the ranks to be included as the vote could be invalid (for example, with repeated or skipped ranks). Without rankings, the server will not be able to identify whether a vote is invalid.

- In *CVRContestInfoJsonAdapter::read()*, a *CVRContestInfo* object is created for a vote in a given contest. We check if the contest is IRV, and if so, we call *IRVVoteParsing::IRVVoteToValidInterpretationAsSortedList* to translate a vote of the form "Alice(1), Bob(2)" to "Alice, Bob."

- We need to perform this translation at this point, because this JSON adaptor creates the *CVRContestInfo* for this vote. During this process, a validity check will take place to ensure that the choices recorded in the *CVRContestInfo* are a subset of those in the associated *Contest*.

  It is desirable to represent candidates in IRV contests only be their name (with no ranks) in the *Contest* and *CastVoteRecord* data structures. This allows the existing canonicalization process to proceed without modification for IRV contests, and supports easier discrepancy calculations during an audit.

  So, the IRV vote passed to the server will contain candidate names and ranks, and the *CVRContestInfo* requires a valid choice list of candidate names without ranks.

- We implemented a small modification to the *CVRContestInfo* class in which an additional piece of information (the raw choices entered by an auditor when uploading a vote on a ballot) is stored as a transient attribute. The addition of a new constructor allows for this additional piece of information to be supplied. These changes do not impact how *CVRContestInfo*'s are used throughout the codebase for Plurality audits.

- In the *ACVRUpload* endpoint, where a *CVRContestInfo* has been formed for an IRV contest, we create, and persist, an instance of *IRVBallotInterpretation* to store the details of how the system interpreted the IRV vote to the database. The details of this interpretation (the ID of the CVR being audited, contest name, raw vote, and interpreted vote) are stored in the database as per Section 4.7.4.

These changes can be found in https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/us/freeandfair/corla/json/CVRContestInfoJsonAdapter.java and https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/us/freeandfair/corla/model/CVRContestInfo.java.

The IRV-inclusive *ACVRUpload* endpoint is at https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/us/freeandfair/corla/endpoint/ACVRUpload.java.

4.7   Database Tables

The integration of IRV into colorado-rla has resulted in a number of additional database tables to store assertions generated for each contest, and a small change to the table storing comparison audits. The database structure is defined by a mapping between Java classes and database tables using an implementation of the JPA API, Hibernate. So, as new Java classes are added to the system, the database structure will naturally change. The *ResetDatabase* endpoint (if in use) has been updated to ensure that data in any new tables is reset as desired. Note this is was not implemented in the prototype.

4.7.1   Table: comparison_audit

A single table is used to store all comparison audits (Plurality and IRV). In light of the addition of a *ComparisonAudit* subclass, the *comparison_audit* table now includes one additional column: a discriminator column used to identify the type of audit being undertaken (IRV or Pluarlity)

For more detail on Java Persistence and Inheritance, refer to: https://en.wikibooks.org/wiki/Java_Persistence/Inheritance.

4.7.2   Table: contest

The description field in the *contest* table is used to store the type of contest (IRV or Plurality). Consequently, this table did not change when IRV was integrated into colorado-rla.

4.7.3   Assertion-related tables

Four tables were added to the database to support the storage of assertions, and the mapping of assertions to comparison audits.

- Table: *assertion*

  As with comparison audits, a single table was used to store all assertions generated by RAIRE. Each row in the *assertion* table defines: an assertion type (discriminator value for each subclass); a unique numeric id; name of the contest to which the assertion belongs; difficulty and margin, as computed by RAIRE; winner and loser; estimated and optimistic samples to audit; and counts for each type of discrepancy. (An additional *version* column is present, arising from the fact that the *Assertion* class implements the *PersistentEntity* interface).

There are two options for how we can store the CVR-discrepancy map associated with each *Assertion* class. The first utilises the existing *LongIntegerMapConverter* class to translate the CVR-discrepancy map into a single text-based column in the *assertion* table. The second is to create a new table *assertion_discrepancies* with columns corresponding to assertion ID, CVR ID, and discrepancy value. In general, the number of discrepancies associated with any given assertion is likely to be small. We chose to implement the second approach, introducing a new database table *assertion_discrepancies.*

- Table: *assertion_discrepancies*

  For each discrepancy computed for an assertion and a CVR, this table records the assertion ID, CVR ID, and discrepancy value.

- Table: *assertion_assumed_continuing*

  Each assertion has an attribute defining the candidates (identified by their names) that are assuming to be 'continuing' in the assertion's context. This information is stored in a table *assertion_assumed_continuing* where each row contains the numeric identifier of the assertion, and the name of an assumed continuing candidate.

- Table: *audit_to_assertions*

  As each *IRVComparisonAudit* contains a list of assertions as an attribute, this information is stored in the database in a table that lists, for each assertion belonging to the audit, the the numeric identifier of the *IRVComparisonAudit* alongside the numeric identifier of the assertion.

4.7.4   Ballot interpretation

We added a new database table to store the details of the auditor-entered ranks for IRV contests, and how the system interpreted those votes to form a valid ranking. This table stores: a unique numeric identifier for the interpretation itself; the ID of the CVR being audited; the contest name; the choices and their ranks as entered by the audit board; and the ordered choices defining the system's interpretation of the ballot for the IRV contest. While it would be ideal to also store the revision number associated with the ballot being audited, this is not attached to the ACVR record at the point of its creation, when the interpretation is taking place and being stored in the database.

4.7.5   Assertion Generation Summaries

The new *generate_assertions_summary* table contains, for each contest:

- the contest name,
- the winner, if successfully calculated by raire,
- the error from raire, if there was one (e.g. *TIED_WINNERS*),
- the message associated with the error, if present (e.g. the names of the tied winners),
- any other warning (e.g. *time_out_trimming_assertions*)

This information is displayed in the Generate Assertions page of the CDOS dashboard, and is also available in the *generate_assertions_summary* report.

4.8 Abstract State Machines

We do not believe that any of the ASMs need to change with the addition of IRV audits to the system. While assertions must be generated prior to the start of an IRV audit, this will take place during the *Define Audit* step. An audit round cannot start until the completion of this step. Consequently, even without introducing new states and events in the DoS Dashboard ASM relating to assertion generation, it can be assured that the system will never start an IRV audit without assertions being generated.

4.9 UI Changes

The main UI changes were as follows.

- In the *Define Audit* step, a sample size estimation page was added. This does *not* require counties to have uploaded manifests.

- A new *Generate Assertions* page was added, as shown in Subsection 4.2.1

- When uploading an audited ballot, auditors have appropriate UI to enter what they see on the ballot for an IRV contest. item The *ACVRUpload* endpoint can receive IRV vote descriptions with choice names *and* how they were ranked. This vote will then be interpreted by the endpoint (Section 4.5), before the audited CVR is stored in the database.

- Report-download UI now includes the IRV-specific reports.

4.10 Additional Changes to Existing Code

The following additional changes to colorado-rla were required to support IRV audits:

- When constructing *ComparisonAudit*s, the *ComparisonAuditController* needs to use the *ComparisonAudit* constructor for Plurality audits and the *IRVComparisonAudit* constructor for IRV audits. As each *Contest* now have a *ContestType* within its description attribute, the controller has the information it needs to determine which constructor to use.

  Code is available at `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/us/freeandfair/corla/controller/ComparisonAuditController.java`.

- Several attributes in *ComparisonAudit* were made *protected* as opposed to *private* so they could be accessed in the *IRVComparisonAudit*. These are the diluted margin, the optimistic and estimated ballots to sample, audited ballot count, and discrepancy totals (for reporting). The *recalculateSamplesToAudit()* method in *ComparisonAudit* was private. It was made protected so that it could be overridden in *IRVComparisonAudit*. By overriding this method, many of the other methods implemented in *ComparisonAudit* did not need to be reimplemented for IRV, but could be used as they are.

Code for *IRVComparisonAudit* is available at `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/au/org/democracydevelopers/corla/model/IRVComparisonAudit.java`.

- A method to compute the current risk for an audit has been added to *ComparisonAudit* and *IRVComparisonAudit*. In the current system, risk is computed for reporting purposes. However, encapsulation is broken by not bundling this method with the data that it uses. Data from *ComparisonAudit*s is extracted and passed to a risk measurement function in *Audit.math*. As slightly different procedures are followed for risk computation for both audit types, it is best to follow the OOP principle of encapsulation and incorporate a risk calculation function in these classes. We have incorporated this into our implementation.

- During development of the prototype integration of IRV into the public colorado-rla repository, it was found useful to add a method to the *CastVoteRecord* class:

  public Optional<CVRContestInfo> contestInfoForContestResult(final String contestName)

This returns the *CVRContestInfo* relating to a specific contest on the CVR, identified by its name. There are a number of variations of *contestInfoForContestResult()* present in *CastVoteRecord*. These could all be replaced with this method, as the other variations either extract the contest name from the given input (e.g., a *ContestResult*) or compare the *Contest* associated with each *CVRContestInfo* with a given *Contest* object.

Changes are available at `https://github.com/DemocracyDevelopers/colorado-rla/blob/main/server/eclipse-project/src/main/java/us/freeandfair/corla/model/CastVoteRecord.java`.

Chapter 5

Testing and Verification

Section 5.1 describes the data that we have used for testing Democracy Developer's changes to colorado-rla and the RAIRE microservice. The approach used for testing changes to colorado-rla is described in Section 5.2, and for testing the RAIRE microservice in Section 5.3.

5.1   Test Data

To test Democracy Developer's changes to colorado-rla and the RAIRE microservice, we have three different kinds of test data: simple examples with human-computable assertions, real-world data from the Australian state of New South Wales and Boulder, Colorado, and extreme test cases designed to test operational limits. The first and third categories are generated manually, while the middle category is derived in bulk from public data.

Depending on the functionality being tested, we produce the data either as CSVs, to match the expected upload from Colorado counties, as json, to test the RAIRE assertion-generation endpoints, or in SQL, for automated tests requiring specific data in the database.

5.1.1   Simple IRV examples with known outputs

These are tiny examples, typically 3 candidates and about 10 votes, to test assertion generation on an election that even humans could generate assertions for. We have also generated some invalid input files to test that parsing behaves as expected on invalid input data. In particular, when the input ranks are invalid (with duplicates, overvotes, or skipped ranks) the valid interpretation is stored. This data is located at: `colorado-rla/server/eclipse-project/src/test/resources/CSVs/Tiny-IRV-Examples`.

5.1.2   Real IRV examples from Australian IRV data

The Australian state of New South Wales makes detailed vote rank data available under an open license. Many of these contests are IRV mayoral contests for New South Wales local government. Although nobody is certain

what voting patterns will appear in Colorado IRV data, the New South Wales dataset is probably the best practical example that resembles what Colorado IRV elections might be like. It includes more than 40 IRV contests, with candidate counts ranging from 3 to more than 10. Ballot counts range from a few thousand in some rural areas to more than 100,000 for Sydney mayor. We have translated the New South Wales data into the CSV form expected by colorado-rla and will use this as the basis for a typical test suite. This data is located at: `colorado-rla/server/eclipse-project/src/test/resources/CSVs/NewSouthWales21`.

5.1.3   Boulder 2023

CSV ballot data from the Boulder election in 2023. This includes an IRV contest for Mayor. This data is located at: `colorado-rla/server/eclipse-project/src/test/resources/CSVs/Boulder23`.

5.1.4   Test cases of unusual data, or data that produces errors or problems

Normal expected operation includes some unusual but not impossible situations. The test suite includes:

- a tied IRV contest (not auditable),
- a contest that times out checking winners (a huge multi-way IRV tie called "time out checking winners contest"),
- a contest (Byron Mayoral '21) that can time out generating assertions (given a very short time limit),
- IRV write-ins,
- IRV contests with non-ascii characters in the candidate names,
- IRV votes with invalid preferences, to test that they are correctly interpreted according to regulation.

5.1.5   Combination tests

The *CSVs* folder also contains a series of combinations of the above, to test that the components work together. This includes but is not limited to:

- combinations of IRV and Plurality for each county;
- tests with a realistic number of contests per County and realistic IRV to Plurality split;
- tests with a large number of IRV contests, *i.e.* for stress testing;
- tests with state-wide and county-specific IRV contests.

5.2   Colorado RLA

5.2.1   Tools

The following tools have been used in the development of an automated test suite for Democracy Developer's changes to colorado-rla:

- The *TestNG* framework for the development of automated tests. These tests are located at: `colorado-rla/server/eclipse-project/src/test/java/au/org/democracydevelopers/corla`.
- *TestContainers* for the construction of test database states across all automated tests.
- *Mokito* for creating mocked representations of various objects in the construction of unit tests. This allows us to test specific functionality in isolation from other features in the code.
- *RestAssured* for API testing and automated testing of audit workflows.
- *GitHub Workflows* for running the automated test suite when each pull request is made into the main branch of Democracy Developers fork of the colorado-rla code base. All tests must pass before changes can be merged into the main branch.

5.2.2   Unit and Integration Tests

Each class added to colorado-rla by Democracy Developers has an associated test class containing unit and integration tests. These tests consider each unit of functionality in these code classes, testing them in the context of both valid and invalid inputs. These tests have been designed to cover the range of contexts in which each unit of functionality may be executed. For functionality that involves database access, *TestContainers* has been used to create and initialize a test version of the colorado-rla database. SQL scripts for initializing the test database are located in `colorado-rla/server/eclipse-project/src/test/resources/SQL`.

5.2.3   API and Endpoint Tests

`colorado-rla/server/eclipse-project/src/test/java/au/org/democracydevelopers/corla/communication` and `colorado-rla/server/eclipse-project/src/test/java/au/org/democracydevelopers/corla/endpoint` contain tests of the proper functioning of *colorado-rla*'s new IRV-related endpoints, *get-assertions* and *generate-assertions*. These generally mock the raire-service, and test that *colorado-rla* properly validates requests and deals appropriately with errors from raire.

The endpoint test folder also includes extensive tests that the *EstimateSampleSizes* endpoint correctly estimates sample sizes for a series of known test cases with independently-calculated data.

5.2.4   Audit Workflow Tests

The audit workflow tests simulate a complete workflow in a test container, with an ephemeral database initialized to colorado-rla's expected initial state.

These tests automate audit run-throughs by accessing colorado-rla endpoints in the order they would be accessed when these audits are performed via the UI. These tests do not test the functioning of the UI, or make a call to the RAIRE microservice. At the point at which the RAIRE microservice would be called, the workflow testing infrastructure loads assertions into the test database from a provided SQL file.

These tests conduct complete run-throughs of IRV and plurality audits, including but not limited to:

- A selection of 'toy' IRV contests that exhibit a range of properties (including tied contests, county contests, multi-county contests, and all-county contests);

- Audits for state and county contests that complete in a single round;

- Canonicalization of IRV contests;

- Audits in which the full spectrum of discrepancies arise;

- An audit for a state contest that goes into a second round;

- An audit for a county contest that goes into a second round;

- A combination of state and county contest audits in various contexts (those that complete in a single round and those that require a second round);

- A combination of IRV and Plurality audits in various contexts (state and county, single and multi-round).

- Ballot reaudits and disagreements.

The workflows specify CVRs and Manifests to upload. These are loaded in for counties in the order they are specified in the workflow, i.e. the first CSV file is uploaded for county 1 (Adams), the second file for count 2 (Arapahoe), etc. There must be the same number of CVRs and Manifests.

The workflow runner completes the following steps:

1. uploads all the specified CVRs (county login),

2. uploads all the specified manifests (county login),

3. uploads the canonicalization file (CDOS login),

4. sets the risk limit, and other audit data such as dates,

5. conducts canonicalization of contest or candidate names,

6. simulates assertion generation by loading the specified SQL file into the database,

7. chooses contests to target for audit,

8. sets the seed,

9. gets the sample size estimates,

10. starts the audit round,

11. for each county with audit work to do,
    (a) signs in the audit board,
    (b) uploads audited ballots as specified (if any),
    (c) reaudits any ballots specified,
    (d) signs off on the round,

12. assesses whether the audit is complete and, if not, returns to Step 10,

13. downloads several reports, including *summarize_IRV, contest, contest_by_county, contest_selection, seed, tabulate_plurality* and *tabulate_county.*

At each step, any values it learns are compared against the expected values specified in the workflow json. For example, when the audit is complete, it checks whether the number of rounds matched the rounds (optionally) specified in the workflow. In the case of infinite audits, it detects certain circumstances that definitely lead to infinite audits, and if these occur it verifies that the workflow specified -1 rounds. However, there may be other circumstances that produce infinite audits that the system is not intelligent enough to recognize. That is, it may be that a workflow that truly is infinite, runs indefinitely, but is not properly identified as infinite.

5.2.5   Workflow JSON Format

Each workflow is defined in a JSON file with a specific format. The key sections of this workflow are defined below, alongside examples. An example of a complete workflow is then presented.

- **Risk Limit**: Defines the risk limit to be used across the audits defined in the workflow.

  ```
  "RISK_LIMIT" : 0.03,
  ```

- **Seed**: Provides the seed to be used when defining the audits.

  ```
  "SEED" : "24098249082409821390482049098",
  ```

- **SQLS**: Defines a list of SQL files, containing assertion data to be loaded into the database at the point where the RAIRE microservice would normally be called to generate assertions.

  ```
  "SQLS" : [
      "SQL/WollongongRichmondValleyAssertions.sql"
  ],
  ```

- **CVRS**: For each county, in order of their ID, provide the location of its CVR export CSV. The first CSV will be used as the CVR export for the first county (with ID 1), the second CSV will be used as the CVR export for the second county (with ID 2), and so on.

  ```
  "CVRS" : [
      "src/test/resources/CSVs/.../City_of_Wollongong_Mayoral_PlusRVM_Q3_R1.csv",
      "src/test/resources/CSVs/.../Richmond_Valley_Mayoral_FirstHalf.csv",
      "src/test/resources/CSVs/.../Richmond_Valley_Mayoral_LastQuarter.csv"
  ],
  ```

- **Manifests**: For each county, in order of their ID, provide the location of its ballot manifest. The first CSV will be used as the manifest for the first county (with ID 1), the second CSV will be used as the manifest for the second county (with ID 2), and so on.

  ```
  "MANIFESTS" : [
      "src/test/resources/CSVs/.../City_of_Wollongong_Mayoral.manifest.csv",
      "src/test/resources/CSVs/.../Richmond_Valley_Mayoral_FirstHalf.manifest.csv",
  ```

```
        "src/test/resources/CSVs/.../Richmond_Valley_Mayoral_LastQuarter.manifest.csv"
    ],
```

· **Canonical list**: Provide the path to the canonicalization file to be used when defining the audits.

```
    "CANONICAL_LIST" : "src/test/resources/CSVs/.../WollongongAndSplitRichmond_Canonical_List.csv
```

· **Contest name change**: Identify the contests, by name, whose name should be changed by canonicalization, and their new name.

```
    "CONTEST_NAME_CHANGE" : {
        "City of Boulder Mayoral Candidates" : "City of Boulder Mayoral"
    },
```

· **Candidate name change**: Identify the candidates, by name, whose name should be changed by canonicalization, and their new name.

```
    "CANDIDATE_NAME_CHANGE" : {
        "Regent of the University of Colorado" : {
            "Clear Winner" : "Distant Winner"
        }
    },
```

· **Contests by selected counties**: Identify, for selected counties, the list of contests in that county. This information, if provided, will be used when verifying the content of produced reports.

```
    "CONTESTS_BY_SELECTED_COUNTIES" : {
        "Adams" : [
            "Wollondilly Mayoral",
            "Adams COUNTY COMMISSIONER DISTRICT 3",
            "Regent of the University of Colorado"
        ],
        "Alamosa" : [
            "Alamosa COUNTY COMMISSIONER DISTRICT 3",
            "Regent of the University of Colorado"
        ],
        "Arapahoe" : [
            "Ballina Mayoral"
        ],
        "Archuleta" : [
            "Bellingen Mayoral"
        ],
        "Baca" : [
            "Burwood Mayoral"
```

```
        ],
        "Bent" : [
            "Byron Mayoral"
        ],
        "Boulder" : [
            "Canada Bay Mayoral"
        ]
    },
```

- **Targets**: Identify the contests to be targeted for audit, and the reason. Note that this information is only used in reporting.

```
    "TARGETS" : {
        "Richmond Valley Mayoral" :  "STATE_WIDE_CONTEST",
        "City of Wollongong Mayoral" : "COUNTY_WIDE_CONTEST"
    },
```

- **Winners**: Identify the winners of selected contests. This information is used to check the contents of reports generated at the end of the audit.

```
    "WINNERS" : {
        "Richmond Valley Mayoral" : "MUSTOW Robert",
        "City of Wollongong Mayoral" : "BRADBERY Gordon"
    },
```

- **Raw margins**: Identify the raw margins of selected contests. This information is used to check the contents of reports generated at the end of the audit.

```
    "RAW_MARGINS" : {
        "Richmond Valley Mayoral" : 5821,
        "City of Wollongong Mayoral" : 2666
    },
```

- **Diluted margins**: Identify the diluted margins of selected contests (at least those targeted for audit).

```
    "DILUTED_MARGINS" : {
        "Richmond Valley Mayoral" : 0.43424095,
        "City of Wollongong Mayoral" : 0.02095253
    },
```

- **Expected samples**: Identify the expected sample sizes, pre-audit, for selected contests (at least those targeted for audit).

```
        "EXPECTED_SAMPLES" : {
            "Richmond Valley Mayoral" : 17,
            "City of Wollongong Mayoral" : 348
        },
```

· **Final expected audited ballots**: Identify the expected number of ballots checked at the end of the audit for targeted contests.

```
        "FINAL_EXPECTED_AUDITED_BALLOTS" : {
            "Richmond Valley Mayoral" : 17,
            "City of Wollongong Mayoral" : 674
        },
```

· **Final expected optimistic samples**: Identify the final optimistic sample count for targeted contests.

```
        "FINAL_EXPECTED_OPTIMISTIC_SAMPLES" : {
            "Richmond Valley Mayoral" : 17,
            "City of Wollongong Mayoral" : 674
        },
```

· **Final expected estimated samples**: Identify the final estimated sample count for targeted contests.

```
        "FINAL_EXPECTED_ESTIMATED_SAMPLES" : {
            "Richmond Valley Mayoral" : 17,
            "City of Wollongong Mayoral" : 675
        },
```

· **Plurality tabulation**: One of the CSV reports that colorado-rla produces provides the tallies of each candidate in each Plurality contest in uploaded data. For state-wide contests, the tallies take into account votes across all counties involved in the contest. We can specify the workflow JSON the plurality contests whose reported tallies we want to verify as shown below.

```
    "PLURALITY_TABULATION" : {
      "Adams COUNTY COMMISSIONER DISTRICT 3" : {
        "Jeff Baker" : 43,
        "Janet Lee Cook" : 1
      },
      "Alamosa COUNTY COMMISSIONER DISTRICT 3" : {
        "Jeff Baker" : 38,
        "Janet Lee Cook" : 1
      },
      "Regent of the University of Colorado" : {
        "Distant Winner" : 244,
        "Distant Loser" : 28
```

```
    }
  },
```

- **Plurality tabulation by county** Another CSV report colorado-rla produces outlines candidate tallies by county for each Plurality contest. This differs from 'Plurality tabulation' by reporting the votes each candidate received from voters in each county separately.

```
  "PLURALITY_COUNTY_TABULATION" : {
    "Adams" : {
      "Adams COUNTY COMMISSIONER DISTRICT 3": {
        "Jeff Baker": 43,
        "Janet Lee Cook": 1
      },
      "Regent of the University of Colorado" : {
        "Distant Winner" : 122,
        "Distant Loser" : 14
      }
    },
    "Alamosa" : {
      "Alamosa COUNTY COMMISSIONER DISTRICT 3": {
        "Jeff Baker": 38,
        "Janet Lee Cook": 1
      },
      "Regent of the University of Colorado" : {
        "Distant Winner" : 122,
        "Distant Loser" : 14
      }
    }
  },
```

- **IRV contests**: Identify all the IRV contests in the CVR exports loaded in the workflow.

```
      "IRV_CONTESTS" : [
          "Richmond Valley Mayoral",
          "City of Wollongong Mayoral"
      ],
```

- **Expected rounds**: Identify the expected number of rounds that will be performed in the audit (maximum across all targeted contests).

```
      "EXPECTED_ROUNDS" : 2,
```

The special value -1 is used to indicate an expected *infinite* number of rounds. Although `EXPECTED_ROUNDS` is usually optional, an infinite workflow must have an explicit prediction of -1 in order to pass.

- **Phantom ballots**: Identify the ballots that will be treated as phantoms during the audit. Ballots are identified by their imprinted ID and county ID. In the following example, two ballots from county I will be treated as phantoms, those with imprinted IDs 1-7-54 adn 1-167-63.

```
"PHANTOM_BALLOTS" : {
    "1" : [
        "1-7-54", "1-167-63"
    ]
},
```

- **Disagreements**: Identify the CVRs that, when their associated paper ballot is audited, will result in a disagreement between auditors. We specify these CVRs with a mapping between county ID in which the ballot was cast, imprinted ID of the ballot, and the names of the contests identifying the votes that will result in a disagreement. In the example below, there will be a disagreement only with the vote in the Uralla Mayoral contest, but we may specify a list of contests in this field.

```
"DISAGREEMENTS" : {
    "4" : {
        "1-5-9" : [
            "Uralla Mayoral"
        ]
    }
},
```

- **Discrepant audited ballot choices**: When each CVR is audited, the contents of the CVR, and any interpretations of that CVR stored in the IRV interpretations database table, will be used to construct the physical ballot. This will mean that the votes on all CVRs and physical ballots will match, unless we specify to inject varied choices for a CVR's physical ballot. We specify these discrepant choices in this part of the workflow. The first-level map specifies the county name, followed by the round number, then the imprinted ID and contents of the physical ballot. In the example below, county 32 has one discrepant ballot, in round 1, for the Byron Mayoral Contest; county 64 has two discrepant ballots, both in round 2.

```
"DISCREPANT_AUDITED_BALLOT_CHOICES" : {
  "32" : {
    "1-2-4": {
      "Byron Mayoral": [
        "COOREY Cate(1)",
        "SWIVEL Mark(2)",
        "PUGH Asren(3)",
        "LYON Michael(4)",
        "CLARKE Bruce(5)"
      ]
    }
  },
  "64" : {
```

```
     "1-3-50": {
       "Byron Mayoral": [
         "SWIVEL Mark(1)",
         "COOREY Cate(2)"
       ]
     },
     "108-1-87": {
       "Byron Mayoral": [
       "LYON Michael(1)",
       "DEY Duncan(2)"
       ]
     }
   }
 }
```

· **Reaudits**: Specify which CVRs are to be re-audited, and what choices the auditors will enter when the CVRs are re-audited. The structure is the same as for discrepant ballots, except that after the imprinted ID, each set of choices may also include a statement of whether there was consensus, for example:

```
     "1-2-4": [
       {
         "Byron Mayoral": {
           "choices": [
             "COOREY Cate(1)",
             "SWIVEL Mark(2)",
             "PUGH Asren(3)",
             "LYON Michael(4)",
             "CLARKE Bruce(5)"
           ],
           "consensus": "YES"
         }
       }
     ]
```

· **County results**: We check for each expected round, and for selected counties, selected fields in the county status of the DoS dashboard response. One field is the discrepancy count map for the county. This *discrepancy_count* may contain up to two types of keys – *AUDITED_CONTEST* and *UNAUDITED_CONTEST*. The values for each key represent the number of (non-resolved) discrepancies arising for ballots audited within the county for contests that were audited (*AUDITED_CONTEST*) and those that were not targeted for audit (*UNAUDITED_CONTEST*).

```
   "COUNTY_RESULTS" : {
     "1" : {
       "32" : {
         "discrepancy_count" : {
           "AUDITED_CONTEST" : 1
```

```
        }
      },
      "64" : {
        "discrepancy_count" : {
          "AUDITED_CONTEST" : 1
        }
      },
      "7" : {
        "discrepancy_count" : {
          "AUDITED_CONTEST" : 1
        }
      }
    },
    "2" : {
      "32" : {
        "discrepancy_count" : {
          "AUDITED_CONTEST" : 1
        }
      },
      "64" : {
        "discrepancy_count" : {
          "AUDITED_CONTEST" : 1
        }
      },
      "7" : {
        "discrepancy_count" : {
          "AUDITED_CONTEST" : 1
        }
      }
    }
  }
}
```

- **Contest results**: For each round, we define the expected discrepancy and disagreement counts for selected contests. In the following example, we expect that there will be no discrepancies recorded for Richmond Valley Mayoral in the first and second rounds, while there will be one two-vote overstatement introduced in the first round for the City of Wollongong Mayoral.

```
"CONTEST_RESULTS" : {
    "1" : {
      "Richmond Valley Mayoral" : {
        "one_over_count" : 0,
        "two_over_count" : 0,
        "one_under_count" : 0,
        "two_under_count" : 0,
        "other_count" : 0,
```

```
          "disagreements" : 0
        },
        "City of Wollongong Mayoral" : {
          "one_over_count" : 0,
          "two_over_count" : 1,
          "one_under_count" : 0,
          "two_under_count" : 0,
          "other_count" : 0,
          "disagreements" : 0
        }
      },
      "2" : {
        "Richmond Valley Mayoral" : {
          "one_over_count" : 0,
          "two_over_count" : 0,
          "one_under_count" : 0,
          "two_under_count" : 0,
          "other_count" : 0,
          "disagreements" : 0
        },
        "City of Wollongong Mayoral" : {
          "one_over_count" : 0,
          "two_over_count" : 1,
          "one_under_count" : 0,
          "two_under_count" : 0,
          "other_count" : 0,
          "disagreements" : 0
        }
      }
    }
  }
```

An example full workflow JSON is provided below:

```
{
  "NAME" : "SmallerStateAndCountyDiscrepanciesTwoRounds",

  "RISK_LIMIT" : 0.03,

  "SEED" : "2409824908240982139048204909",

  "SQLS" : [
    "SQL/WollongongRichmondValleyAssertions.sql"
  ],

  "CVRS" : [
```

    "src/test/resources/CSVs/WollongongAndSplitRichmondValley/City_of_Wollongong_Mayoral_PlusRVM_Q3_R1.csv",
    "src/test/resources/CSVs/WollongongAndSplitRichmondValley/Richmond_Valley_Mayoral_FirstHalf.csv",
    "src/test/resources/CSVs/WollongongAndSplitRichmondValley/Richmond_Valley_Mayoral_LastQuarter.csv"
],

"MANIFESTS" : [
    "src/test/resources/CSVs/NewSouthWales21/City_of_Wollongong_Mayoral.manifest.csv",
    "src/test/resources/CSVs/WollongongAndSplitRichmondValley/Richmond_Valley_Mayoral_FirstHalf.manifest.csv",
    "src/test/resources/CSVs/WollongongAndSplitRichmondValley/Richmond_Valley_Mayoral_LastQuarter.manifest.csv"
],

"CANONICAL_LIST" : "src/test/resources/CSVs/.../WollongongAndSplitRichmond_Canonical_List.csv",

"CONTEST_NAME_CHANGE" : {
},

"CANDIDATE_NAME_CHANGE" : {
},

"CONTESTS_BY_SELECTED_COUNTIES" : {
},

"TARGETS" : {
    "Richmond Valley Mayoral" :   "STATE_WIDE_CONTEST",
    "City of Wollongong Mayoral" : "COUNTY_WIDE_CONTEST"
},

"WINNERS" : {
    "Richmond Valley Mayoral" : "MUSTOW Robert",
    "City of Wollongong Mayoral" : "BRADBERY Gordon"
},

"RAW_MARGINS" : {
    "Richmond Valley Mayoral" : 5821,
    "City of Wollongong Mayoral" : 2666
},

"DILUTED_MARGINS" : {
    "Richmond Valley Mayoral" : 0.43424095,
    "City of Wollongong Mayoral" : 0.02095253
},

"EXPECTED_SAMPLES" : {
    "Richmond Valley Mayoral" : 17,
    "City of Wollongong Mayoral" : 348
},

"FINAL_EXPECTED_AUDITED_BALLOTS" : {
    "Richmond Valley Mayoral" : 17,

```
    "City of Wollongong Mayoral" : 674
},

"FINAL_EXPECTED_OPTIMISTIC_SAMPLES" : {
    "Richmond Valley Mayoral" : 17,
    "City of Wollongong Mayoral" : 674
},

"FINAL_EXPECTED_ESTIMATED_SAMPLES" : {
    "Richmond Valley Mayoral" : 17,
    "City of Wollongong Mayoral" : 675
},

"PLURALITY_TABULATION" : {
},

"PLURALITY_COUNTY_TABULATION" : {
},

"IRV_CONTESTS" : [
    "Richmond Valley Mayoral",
    "City of Wollongong Mayoral"
],

"EXPECTED_ROUNDS" : 2,

"PHANTOM_BALLOTS" : {
},

"DISAGREEMENTS" : {
},

"DISCREPANT_AUDITED_BALLOT_CHOICES" : {
    "1" : {
        "1-587-10" : {
            "City of Wollongong Mayoral" : [
                "BROWN Tania(1)"
            ]
        }
    }
},

"REAUDITS" : {
    "1" : {
        "1-587-10" : [
            {
                "City of Wollongong Mayoral": {
                    "choices" : [
                        "BROWN Tania(1)"
```

```
        ],
        "consensus" : "YES"
      }
    }
  ]
}
},

"CONTEST_RESULTS" : {
  "1" : {
    "Richmond Valley Mayoral" : {
      "one_over_count" : 0,
      "two_over_count" : 0,
      "one_under_count" : 0,
      "two_under_count" : 0,
      "other_count" : 0,
      "disagreements" : 0
    },
    "City of Wollongong Mayoral" : {
      "one_over_count" : 0,
      "two_over_count" : 1,
      "one_under_count" : 0,
      "two_under_count" : 0,
      "other_count" : 0,
      "disagreements" : 0
    }
  },
  "2" : {
    "Richmond Valley Mayoral" : {
      "one_over_count" : 0,
      "two_over_count" : 0,
      "one_under_count" : 0,
      "two_under_count" : 0,
      "other_count" : 0,
      "disagreements" : 0
    },
    "City of Wollongong Mayoral" : {
      "one_over_count" : 0,
      "two_over_count" : 1,
      "one_under_count" : 0,
      "two_under_count" : 0,
      "other_count" : 0,
      "disagreements" : 0
    }
  }
}
}
```

5.2.6   Provided Workflows

We have provided the following workflows in `colorado-rla/server/eclipse-project/src/test/resources/workflows/instances`. Unless otherwise stated, all workflows define audits with a 3% risk limit.

TinyIRV

> This is a workflow version of a demonstration audit that can be manually stepped through, with instructions provided at `colorado-rla/server/eclipse-project/src/test/resources/workflow/README.md` under *Running it manually*. Data (CVR exports and manifests) is uploaded for the first four counties, and consists of a small IRV contest across two counties, two small plurality contests across two counties, a small tied county-wide IRV contest, and a small county-wide IRV contest. The two IRV contests, one across multiple counties and the other county-wide, are targeted for audit. Two discrepancies are introduced in the first round of auditing for the county-wide IRV contest.
>
> This workflow runs infinitely. Although one targeted contest (TinyExample1) has met the risk limit after the first round, the other (Example3) has enough discrepancies introduced that it cannot meet the risk limit.

AllCountyWidePluralityAndIRVDiscrepanciesReaudited

> This workflow uses six New South Wales local government mayoral contests and three plurality contests drawn from existing colorado-rla test data. Note that this workflow is described as 'all county wide' as all of the IRV contests in the uploaded data are county-wide contests. One of Plurality contests, however, is still a state-wide contest (albeit across only two counties). The uploaded data defines the following contests:

Adams	– Wollondilly Mayoral (NSW, IRV, county-wide)
	– Adams COUNTY COMMISSIONER DISTRICT 3 (colorado-rla, Plurality, county-wide)
	– Regent of the University of Colorado (colorado-rla, Plurality, state-wide)
Alamosa	– Alamosa COUNTY COMMISSIONER DISTRICT 3 (colorado-rla, Plurality, county-wide)
	– Regent of the University of Colorado (colorado-rla, Plurality, state-wide)
Arapahoe	– Ballina Mayoral (NSW, IRV, county-wide)
Archuleta	– Bellingen Mayoral (NSW, IRV, county-wide)
Baca	– Burwood Mayoral (NSW, IRV, county-wide)
Bent	– Byron Mayoral (NSW, IRV, county-wide)
Boulder	– Canada Bay Mayoral (NSW, IRV, county-wide)

> The state-wide contest for Regent of the University of Colorado was called "Regent of the University of Colorado - At Large" in the uploaded data, but was renamed to "Regent of the University of Colorado" during canonicalisation. The candidate "Clear Winner" was renamed to "Distant Winner" during canonicalisation.
>
> The contests Regent of the University of Colorado, Alamosa COUNTY COMMISSIONER DISTRICT 3, Wollondilly Mayoral, and Burwood Mayoral, were targeted for audit. This workflow introduces two discrepancies in the first round – one for Wollondilly Mayoral and the other for Burwood Mayoral – but re-audits the discrepant ballots to remove those discrepancies. The audit completes one round.

AllCountyWidePluralityAndIRVTwoRounds

> This workflow creates the same scenario as **AllCountyWidePluralityAndIRVDiscrepanciesReaudited**, but *does not* re-audit the two discrepant ballots, resulting in a one vote overstatement in Wollondilly Mayoral, and a two vote overstatement in Burwood Mayoral, in the first round. The audit proceeds to the second round, in which no discrepancies are discovered and the audit completes.

AllDiscrepanciesReauditedSingleRound

> This workflow uploads CVR and manifests across all 64 counties. The New South Wales Byron Mayoral contest CVRs are split across all 64 counties, creating a state-wide IRV contest. The Adams and Alamosa counties contain the contests described for the **AllCountyWidePluralityAndIRVDiscrepanciesReaudited** workflow, in addition to a portion of the Byron Mayoral IRV contest. The Archuleta county includes a county-wide IRV contest – Kempsey Mayoral – alongside the state-wide Byron Mayoral and a tied IRV contest. Boulder includes past contests (a mix of IRV and Plurality) and a portion

of Byron Mayoral. In total, there are four IRV contests in this workflow – Byron Mayoral, Kempsey Mayoral, Tied IRV, and City of Boulder Mayoral. Byron Mayoral, Kempsey Mayoral and City of Longmont Mayor (Plurality) are targeted for audit. In the first round of auditing, seven discrepancies are introduced across Byron and Kempsey Mayoral, and are all re-audited. The audit completes in one round with no (remaining) discrepancies. This example consists of a mix of IRV and Plurality contests, and a mix of county and state-wide contests.

AllPluralityDiscrepanciesReaudited

This workflow uploads data for the first two counties, Adams and Alamosa, containing the contests: Regent of the University of Colorado, Adams COUNTY COMMISSIONER DISTRICT 3, Alamosa COUNTY COMMISSIONER DISTRICT 3. All contests are Plurality, and are targeted for audit. The Regent of the University of Colorado spans both counties, and the county commissioner contests are county-wide. Two discrepancies are introduced in the first round of auditing. The discrepant ballots are re-audited and the discrepancies removed. The audit completes in one round.

AllPluralityTwoVoteOverstatementTwoRounds

This workflow creates the same scenario as **AllPluralityDiscrepanciesReaudited**. In this workflow, however, one two vote overstatement is introduced in the first round of auditing for Regent of the University of Colorado. It is not removed through re-auditing. The audit proceeds to a second round.

LastBallotInRoundDiscrepancy

This workflow identifies a bug in the auditing logic of colorado-rla, in which the last ballot in a county's sample for a round cannot be re-audited. The data uploaded is the same as that used in the **AllDiscrepanciesReauditedSingleRound** workflow. One discrepancy is introduced, for the last ballot in one of the county's sample. This discrepant ballot is supposed to be re-audited, but colorado-rla produces an error and prevents the re-audit.

StateAndCountySingleRound

In this workflow, the data uploaded is the same as that for the **AllDiscrepanciesReauditedSingleRound** workflow. No discrepancies are introduced, and the audit completes in a single round.

StateAndCountyDisagreements

In the StateAndCounty workflows, the data uploaded is almost the same as that used for the **AllDiscrepanciesReauditedSingleRound** workflow. The only difference is that in this series of workflows, the Uralla Mayoral contest, from the New South Wales data set, is used in place of Kempsey Mayoral in the Archuleta county. Three contests are targeted for audit – City of Longmont Mayor (Plurality, county-wide), Byron Mayoral (IRV, state-wide), and Uralla Mayoral (IRV, county-wide). In this series of workflows, it is the state-wide IRV contest (Byron) that is the driving contest in the audit. In this workflow, a disagreement over a ballot interpretation for the Uralla Mayoral contest is introduced in the first round of auditing. This results in a one vote overstatement. The audit completes in two rounds.

StateAndCountyPhantomBallots

Using the same data as the **StateAndCountyDisagreements** workflow, this workflow is designed to test the auditing logic in the presence of phantom ballots in the sample. One ballot, from the Archuleta county, is treated as a phantom, introducing a one vote overstatement for Uralla Mayoral. The audit completes in two rounds.

StateAndCountyTwoRounds

Using the same data as the **StateAndCountyDisagreements** workflow, this workflow introduces three discrepancies in the first round of auditing (for Byron Mayoral), with each of the discrepant ballots being re-audited. For one of these three ballots, the re-audit results in a disagreement. For the other two, the discrepancies are confirmed. The end result is a one vote overstatement, a two vote overstatement, and a one vote understatement, for Byron Mayoral. No new discrepancies are introduced in the second round, and the audit completes in two rounds.

StateAndCountyUnbalanced

This workflow uses the same data as the **AllDiscrepanciesReauditedSingleRound** workflow. This workflow results in an 'infinite' (never terminating) audit. It introduces a one and a two vote overstatement for Byron Mayoral in the first round, and a one vote understatement for Kempsey Mayoral. Given that in the Archuleta county, the Byron contest is only on a very small proportion of the total ballots cast in that county, each new sample generated, for subsequent rounds, has a lot of previously seen ballots. This may be the reason that the audited sample count in each round *never* reaches the

optimistic sample count, continually triggering new rounds. However, in lines 280 to 282 of StartAuditRound.java, the result of *BallotSelection.auditedPrefixLength(comparisonAudit.getContestCVRIds())* for the contest exceeds the number of optimistic samples, indicating that no further ballots need to be selected. It is not clear whether this indicates a problem in the computation of the audited sample counter, and whether colorado-rla includes or excludes duplicates from this count. For this scenario, the targeted contests meet the risk threshold, but new rounds are continually triggered.

SmallerStateAndCountyDisagreements

In the SmallerStateAndCounty workflows, data is uploaded for three counties:

Adams	– City of Wollongong Mayoral (IRV, county-wide)
	– Richmond Valley Mayoral (IRV, state-wide)
Alamosa	– Richmond Valley Mayoral (IRV, state-wide)
Archuleta	– Richmond Valley Mayoral (IRV, state-wide)

These workflows only contain IRV contests, one state-wide (across the three counties) and one county-wide (in Adams). Richmond Valley Mayoral and City of Wollongong Mayoral are targeted for auditing. In this case, it is the county-wide contest that has the smaller margin and is driving the audit. Two disagreements in ballot interpretation are introduced, and one discrepant ballot, for City of Wollongong Mayoral, with two of these ballots re-audited (one of the ballots resulting in a disagreement and the discrepant ballot). For the discrepant ballot, a disagreement is introduced during the re-audit. The end result is two disagreements recorded and two one vote overstatements in the first round. The audit proceeds to a second round, and completes in two rounds.

SmallerStateAndCountyDiscrepanciesReaudited

This workflow uses the same scenario as **SmallerStateAndCountyDisagreements**. Four discrepant ballots are defined, resulting in five discrepancies. Three of the discrepant ballots are reaudited, and four of the discrepancies are removed. The remaining discrepancy falls under the 'other' category (not an overstatement or understatement). The audit concludes in a single round.

SmallerStateAndCountyDiscrepanciesSingleRound

This workflow uses the same scenario as **SmallerStateAndCountyDisagreements**. Two discrepancies are introduced during the first round – an overstatement and an 'other' discrepancy for Wollongong Mayoral. These discrepancies do not influence the sample size required, however, as the overstatement is recorded against an assertion that is not 'driving' the audit (i.e., even with the overstatement, the assertion is not the most difficult to audit assertion in the audit). The audit concludes in one round.

SmallerStateAndCountyDiscrepanciesTwoRounds

This workflow uses the same scenario as **SmallerStateAndCountyDisagreements**. One two vote overstatement is introduced for Wollongong Mayoral in the first round. The ballot is reaudited, but the discrepancy is confirmed. The audit requires two rounds to complete.

SmallerStateAndCountyNoDiscrepanciesSingleRound

This workflow uses the same scenario as **SmallerStateAndCountyDisagreements**. No discrepancies are introduced, and the audit concludes in one round.

SmallerStateAndCountyPhantomBallots

This workflow uses the same scenario as **SmallerStateAndCountyDisagreements**. Two paper ballots corresponding to sampled CVRs in the first round are treated as phantoms. This results in a one vote overstatement and an 'other' discrepancy. The audit concludes in two rounds.

SmallerStateAndCountyThreeRounds

This workflow uses the same scenario as **SmallerStateAndCountyDisagreements**. This workflow tests discrepancies being introduced across two rounds. In the first round, a two vote overstatement is introduced for Wollongong Mayoral. The ballot is reaudited, but the discrepancy is confirmed. The audit proceeds to a second round, where a one vote overstatement is introduced. The ballot is reaudited, and the discrepancy is confirmed. The audit proceeds to a third round, where no new discrepnacies are added, and the audit concludes.

IRVContestOneCandidateTest

In this workflow, we take the TinyIRV workflow and modify it to include an IRV contest with only one candidate. We check that the raire-service will not fail in this context, but will not produce assertions.

NonTargetedIRVContestDiscrepancies

This workflow uploads the same data as in the SmallerStateAndCounty series. We target only the City of Wollongong Mayoral contest, and leave the City of Richmond Valley Mayoral contest un-targeted. We introduce discrepancies for both contests and check that they are all recorded appropriately.

There are also two extra workflows that do not fit in to the general structure, and are expressed as a dedicated test rather than a json instance. These test unusual features that are not part of standard workflows.

UploadAndDeleteIRVCVRs

This workflow tests that it is possible to upload and delete CSV files that contain IRV ballots.

EstimateSampleSizesVaryingManifests

This workflow tests that sample size estimation occurs correctly without manifests (in which case, it uses the total number of CVRs as the total number of ballot cards), and with manifests that indicate more ballots than the CVR export file.

5.2.7 Running a workflow in a UAT setting and generating assertions using RAIRE

As an experimental feature, we have provided some support for running workflows in a UAT setting.

It tests workflows in the format described above, but instead of running them inside a test container, it interacts with a UAT setup, including the colorado-rla server and the raire service, executing the steps that the client would otherwise execute manually. This allows for more realistic testing, including using raire to generate the assertions, and being able to inspect the client UI as the workflow progresses.

This requires significant setup including

- java (and maven) installed on the machine,
- a copy of colorado-rla's source code on the machine,
- a colorado-rla server running at an accessible url,
- a raire-server running at an accessible url,
- a corla database running at an accessible url.

See */src/test/java/au/org/democracydevelopers/corla/workflows/WorkflowRunnerWithRaire.java* for setup instructions.

Testing timeout generating assertions   A related test at *src/test/java/au/org/democracydevelopers/corla/workflows/WorkflowRunnerWithRaireWithTimeoutAndAssertionReplacement.java* checks for proper recovery from a timeout. The test first sends *raire-service* an assertion generation request with an extremely small timeout, checks that it times out, resends the request with a longer timeout, and then checks that the successful generation record replaces the timeout failure.

Both these tests also contain instructions for running the test manually using the client.

Please contact us if there are any issues—these tests are less predictable than the others because they run on an existing machine rather than inside a test container.

5.2.8   Running a workflow – automated steps

We have implemented a generic workflow runner designed to take a JSON workflow, as described in the preceding sections, run through the steps of the defined audit, and verify that the results were as expected. The workflow runner executes the following steps in sequence:

1. It loads the workflow JSON into an Instance object.
2. Collects the identified list of CVR and manifest CVS files and checks that there is a manifest for each CVR CVS file.
3. Each county uploads their manifest CSV file.
4. Each county uploads their CVR CSV file.
5. A DoS dashboard refresh response is generated, and the initial state of the dashboard is checked.
6. The risk limit is provided and the canonicalization file provided.
7. Canonicalization is performed as specified in the workflow JSON.
8. Assertions are generated. This may involve a call to the raire-service (if WorkflowRunnerWithRAIRE is being run) or storage of pre-defined assertions (in specified SQL files) in the database (if WorkflowRunner is being run).
9. A check is performed to make sure that right number of IRV contests are present, and that we have assertion summaries for each of those contests.
10. Selected contests are targeted for audit.
11. The seed is provided.
12. A check is performed to ensure that we are in the COMPLETE_AUDIT_INFO_SET ASM state.
13. Sample sizes for each contest are estimated, and checked against expected values.
14. Audit rounds are performed until the audited sample count reaches or exceeds the optimistic sample size for each targeted contest. At the end of each round, the discrepancy and disagreement counts are checked for selected contests.
15. At the conclusion of the audit, the number of performed rounds is cross-checked against the number of expected rounds, and the audited sample count for selected contests cross-checked against expected values.
16. Selected CSV reports are generated, and their contents checked. These are: summarize_IRV; contest; contest_by_county; contest_selection; seed; tabulate_plurality; tabulate_county_plurality.

5.3   RAIRE Microservice

5.3.1   Tools

The following tools have been used in the development of an automated test suite for the RAIRE microservice:

- The *JUnit* framework for the development of automated tests. These tests are located at: `raire-service/src/test/java/au/org/democracydevelopers/raireservice`.
- *@SpringbootTest* along with *TestContainers* to run a variety of integration tests, including those that generate a fresh independent database instance, and run a version of the service for interaction.
- *GitHub Workflows* for running the automated test suite when each pull request is made into the main branch of the RAIRE microservice. All tests must pass before changes can be merged into the main branch.

5.3.2   Unit and Integration Tests

Each class in the RAIRE microservice has an associated test class containing unit and integration tests. These tests consider each unit of functionality in these code classes, testing them in the context of both valid and invalid inputs. These tests have been designed to cover the range of contexts in which each unit of functionality may be executed. This means that unit tests often need to include database setup (and teardown). For functionality that involves database access, *TestContainers* has been used to create and initialize a test version of the colorado-rla database. SQL scripts for initializing the test database are located in `raire-service/src/test/resources`.

5.3.3   API and Endpoint Tests

The test suite at `/src/test/java/au/org/democracydevelopers/raireservice/controller` contains extensive tests for assertion generation and retrieval, including

- small known test cases,
- the complete set of New South Wales 2021 mayoral contests,
- tests that induce errors or warnings, such as tied winners or timeouts,
- tests of repeated operations such as repeat calls to generate assertions with different timeouts
- tests that the correct data is returned in either json or csv format from the *get-assertions* endpoints.

Appendix A

UI-Endpoint correspondence

The following shows the main UI features with the corresponding endpoint.

A.1 The DoS View

Login (Administrator) Figure A.1
Endpoint: AuthenticateAdministrator.java (/auth-admin)

```
Request payload:
username: "stateadmin1"
password ""

Response:
role: STATE
stage "TRADITIONALLY_AUTHENTICATED"
challenge "[B,6] [G,2] [I,3]"
```

Grid Challenge   Figure A.2
Endpoint: AuthenticateAdministrator.java (/auth-admin)

```
Request payload:
second_factor: "a s d f"
username: "stateadmin1"

Response:
role "STATE"
stage: "SECOND_FACTOR_AUTHENTICATED"
```

This then passes to the DoS dashboard, with the County Updates tab selected.

DoS dashboard: /sos   The following API endpoints are called before any user interaction.
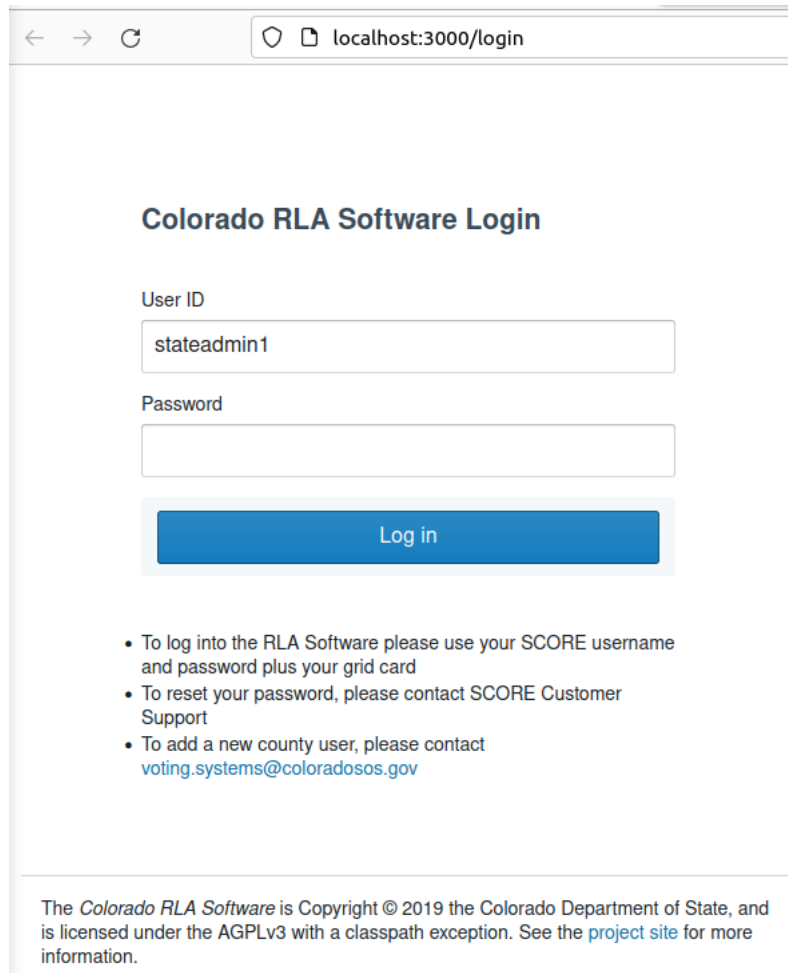
Figure A.1: Administrator login to DoS Dashboard.

Endpoint: DoSDashboardRefresh.java (/dos-dashboard)

```
No Request payload.

Response:
asm_state "PARTIAL_AUDIT_INFO_SET"
audited_contests Object { }
estimated_ballots_to_audit Object { }
optimistic_ballots_to_audit Object { }
discrepancy_count Object { }
county_status Object { 1: {...}, 2: {...}, 3: {...}, ... }
hand_count_contests []
audit_info Object { election_type: "general", election_date: "2023-09-07T02:37:39.945Z",
    public_meeting_date: "2023-09-07T14:00:00Z", ... }
election_type "general"
```

Colorado RLA Software Login

Enter a response to the grid challenge **[B6] [G2] [I3]** for the user **stateadmin1**.

Grid challenge response

`••••`

Log in

Figure A.2: Administrator login to DoS Dashboard; Grid challenge.

```
election_date "2023-09-07T02:37:39.945Z"
public_meeting_date "2023-09-07T14:00:00Z"
risk_limit 0.05
canonicalContests Object { Boulder: [...] }
canonicalChoices Object { "IRV for Test County": [...] }
audit_reasons Object { }
audit_types Object {}

Each county_county status is something like
id 3
asm_state "COUNTY_INITIAL_STATE"
audit_board_asm_state "AUDIT_INITIAL_STATE"
audit_boards Object { }
estimated_ballots_to_audit 0
optimistic_ballots_to_audit 0
ballots_remaining_in_round 0
ballot_manifest_count 0
cvr_export_count 0
```

Endpoint: ContestDownload.java (/contest)

```
No Request payload.

Response:

id 14115
name "IRV for Test County"
county_id 7
description ""
choices [ {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, ... ]
votes_allowed 10
winners_allowed 10
```

```
sequence_number 0
```

```
Each choice is something like
name "Candidate 6(1)"
description ""
qualified_write_in false
fictitious false
```

Note the contest response is actually an array with only one element. Probably if multiple counties have uploaded, there are multiple elements.

The client continues to poll /dos-dashboard and /contest.

If the 'Contest Updates' tab is selected instead, the API calls are the same.

Selecting a particular County   Figures A.3 and A.4

When a particular county is selected (Boulder in this example) it visits /sos/county/[countyIDName]. For example, Boulder is /sos/county/7

SOS Dashboard single-county view   Figure A.4

This has two buttons (at least, two now visible): Download and Delete File.

Endpoint: FileDownload.java (/download-file)

```
Request: /download-file?fileId=4092
```

```
No Request header.
Response: the csv file
```

Endpoint: DeleteFile.java (/delete-file)

```
(Didn't test this. Assume it deletes the file.)
```

Defining the Audit   Figure A.5

Clicking 'Define Audit' doesn't seem to actually do anything other than call /dos-dashboard again, presumably to get up-to-date status.

Nor does dragging and dropping a file.

The Audit admin dashboard appears as shown in Figure A.6.

Clicking 'Save and Next' accesses the endpoint UpdateAuditInfo.java (/update-audit-info):

```
Request payload:
election_date "2023-09-07T02:37:39.945Z"
election_type "general"
```

Figure A.3: DoS Dashboard: selecting a County.

```
public_meeting_date "2023-09-07T14:00:00.000Z"
risk_limit 0.05
upload_file [The contents of the canonicalization csv]

Response:
result: "audit information updated."
```

Selecting Contests Figure A.7

Filtering, selection etc are done locally.

Clicking 'Save and Next' accesses the endpoint SelectContestsForAudit.java (/select-contests):

```
Request payload:
audit "COMPARISON"
contest 14115
reason "COUNTY_WIDE_CONTEST"

Response:
result "Contests selected"
```

Boulder County Info

NAME	STATUS	AUDITED DISCREPANCIES	NON-AUDITED DISCREPANCIES	SUBMITTED
Boulder	Ballot man…	—	—	0

Ballot Manifest

File name	IRV-test-manifest.csv
SHA-256 hash	C60B60B5F8E29C867AA3EE5968AC0E399A331E1138B0290765339…

✓ File successfully uploaded   07/09/2023, 12:50:53 pm   Download   Delete File

CVR Export

File name	CVR_Export_IRV_Example.csv
SHA-256 hash	75A0ECA27564BB9419120C04BF53C9A964FBB3C80AA248AA8982…

✓ File successfully uploaded   08/09/2023, 11:11:33 am   Download   Delete File

Audit boards

BOARD MEMBER #1	BOARD MEMBER #2	SIGN-IN TIME

Figure A.4: DoS Dashboard: selecting a County (Boulder).

Choosing the seed   Figure A.8

Clicking 'Save and Next' accesses the endpoint SetRandomSeed.java (/random-seed):

```
Request payload:
seed "12341234123412341234"

Response:
result "random seed set to 12341234123412341234"
```

Launching the Audit   Figure A.9

This then transitions to an 'Audit Definition Review' screen.

Clicking 'Launch Audit' accesses the endpoint StartAuditRound.java (/start-audit-round):

```
No payload
```
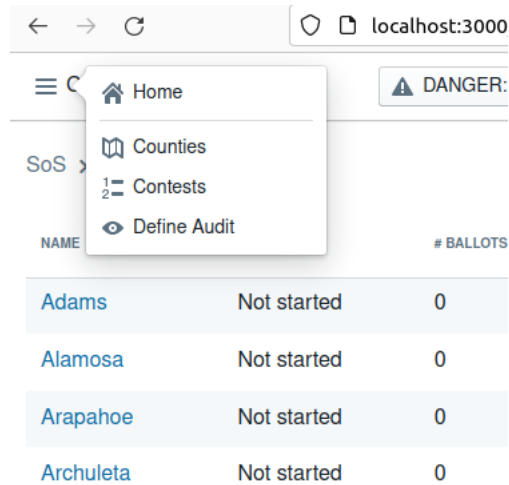
Figure A.5: DoS Dashboard: Defining the audit.

```
Response:
result "round started"
```

A.2   The County's view

Now that the audit is defined, the county is told that they may begin (Figure A.10).

It is already polling /county-dashboard, /county-asm-state, /audit-board-asm-state, /contest and /cvr-to-audit-list?round=1.

Endpoint: CountyDashboardRefresh.java (/county-dashboard)

```
No payload.
Response:
id 7
asm_state "COUNTY_AUDIT_UNDERWAY"
and a lot of other data
```

Endpoint: CountyDashboardASMState.java (/county-asm-state)

```
No Payload.
Response:
current_state "COUNTY_AUDIT_UNDERWAY"
enable_ui_events []
```

Endpoint: AuditBoardDashboardASMState.java (/audit-board-asm-state)

```
No Payload.
Response:
current_state "ROUND_IN_PROGRESS_NO_AUDIT_BOARD"
```

Figure A.6: Audit admin dashboard after DoS defines the audit.

```
enabled_ui_events []
```

Figure A.7: Selecting contests for audit.

Figure A.8: Choosing the random seed.

Endpoint: ContestDownload.java (/contest)

```
Request: /contest/county?7
Response: Looks the same as when SOS calls it.
```

Figure A.9: Launching the audit.

Endpoint: CVRToAuditList.java (/cvr-to-audit-list)

```
Request: /cvr-to-audit-list?round=1
```

```
Response:
A long list of audit items (equal in number to the 'x ballot cards to audit' message, like:
```

```
audit_sequence_number 63
scanner_id 1
batch_id "3"
record_id 1
```

Figure A.10: County dashboard, after DoS launches the audit.

```
imprinted_id "1-3-1"
cvr_number 79
db_id 14195
ballot_type "Ballot 1 - Type 1"
storage_location "Bin 1"
audited false
```
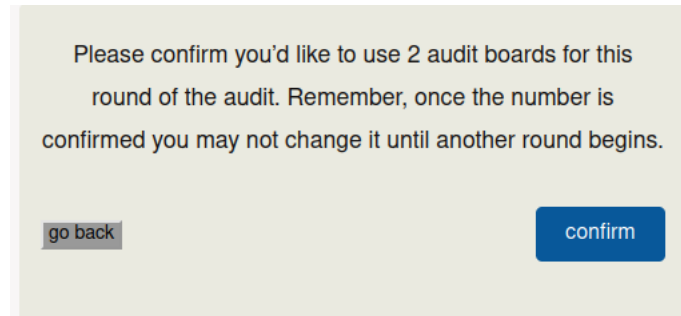
Figure A.11: Confirming audit boards.

Clicking 'Enter' after audit board number raises a popup (Figure A.11).

Clicking 'confirm' in Figure A.11 accesses the endpoint SetAuditBoardCount.java (/set-audit-board-count):

```
Request payload:
count 2

Response:
result "set the number of audit boards to 2"
```

Audit board sign in   Then buttons for Audit board sign up appear as shown in Figure A.12.

When I click 'Audit Board 2' it does something I don't understand (Endpoint CVRDownloadByID.java, /cvr/id/[id])

```
Endpoint: /cvr/id/14348

No payload.
Response:
id 14348
record_type "UPLOADED"
county_id 7
cvr_number 232
sequence_number 231
scanner_id 1
batch_id "4"
record_id 76
imprinted_id "1-4-76"
uri "cvr:7:1-4-76"
ballot_type "Ballot 1 - Type 1"
contest_info [ {...} ]
0 Object { contest: 14115, choices: [...] }
contest 14115
choices [The list of candidate/choice names]
```

This does not make any sense because the audit board hasn't even signed in yet. But it turns out this is the next ballot to be audited, so maybe it does make sense at that point.

Figure A.12: Audit board sign up buttons.

It shifts to /county/board/1 and displays the screen shown in Figure A.13.

When filled in with some names and party affiliations, upon clicking 'Sign in', the endpoint AuditBoardSignIn.java (/audit-board-sign-in) is accessed:

Figure A.13: Audit board sign in.

```
Request payload:
audit_board [...]
0 {...}
first_name "Alice"
last_name "Alice"
political_party "Republican Party"
1 {...}
first_name "Bob"
last_name "Bob"
political_party "Democratic Party"
index 1
```

```
Response:
result "audit board #1 for county 7 signed in: [Elector [first_name=Alice, last_name=Alice,
    political_party=Republican Party], Elector [first_name=Bob, last_name=Bob,
    political_party=Democratic Party]]"
```

Also continues to poll /county-dashboard, /audit-board-asm-state, /county-asm-state, /contest/county?7, /cvr-to-audit-list?round=1

The Audit board dashboard   Figure A.14

Clicking 'Download ballot list as CSV' does the following (Endpoint CVRToAuditDownload.java, /cvr-to-audit-download):

```
Request: /cvr-to-audit-download?round=1

Response: the CSV file
Basically the same as what's displayed on the screen
```

Clicking 'Start audit' pops up a 'ballot card verification' popup (Figure A.15).

Clicking 'Continue' hides the popup and displays the CVR-entry dashboard (Figure A.16).

Selecting some candidates and clicking 'Review' prints a nice summary as shown in Figure A.17 (this is /county/audit/1).

Note: Sometimes the 'Auditing ballot card' number shifts from n to n+[more than 1] if there are duplicates that have actually already been audited.

Clicking 'Submit & Next ballot card' does the following (Endpoint ACVRUpload.java, /upload-audit-cvr):

```
Request payload:
audit_cvr {...}
ballot_type "Ballot 1 - Type 1"
batch_id "4"
contest_info [...]
0 {...}
choices [...]
0 "Candidate 4(2)"
1 "Candidate 10(2)"
2 "Candidate 8(2)"
3 "Candidate 4(3)"
4 "Candidate 8(3)"
comment ""
consensus "YES"
contest "14115"
county_id 7
cvr_number 232
id 14348
imprinted_id "1-4-76"
record_id 76
record_type "UPLOADED"
scanner_id 1
```

Audit Board 2: Ballot Cards to Audit

The Secretary of State has established the following risk limit(s) for the following ballot contest(s) to audit:

- IRV for Test County – 5%

The Secretary of State has randomly selected 249 ballot cards for the Boulder County Audit Board(s) to examine in Round 1 to satisfy the risk limit(s) for the audited contest(s).

The Audit Board(s) must locate and retrieve, or observe a county staff member locate and retrieve, the following randomly selected ballot cards for the initial round of this risk-limiting audit:

Audit Board 2: Click Start audit to begin reporting the votes you observe on each of the below ballot cards that have been assigned to you.

Start audit

Download ballot list as CSV

Storage bin	Scanner	Batch	Ballot position	Ballot type	Audited	Audit board
Bin 1	1	3	1	Ballot 1 - Type 1		1
Bin 1	1	3	2	Ballot 1 - Type 1		1
Bin 1	1	3	4	Ballot 1 - Type 1		1
Bin 1	1	3	6	Ballot 1 - Type 1		1
Bin 1	1	3	7	Ballot 1 - Type 1		1

Figure A.14: Audit board dashboard.

```
timestamp "2023-09-13T12:16:22.015Z"
auditBoardIndex 1
cvr_id 14348
```
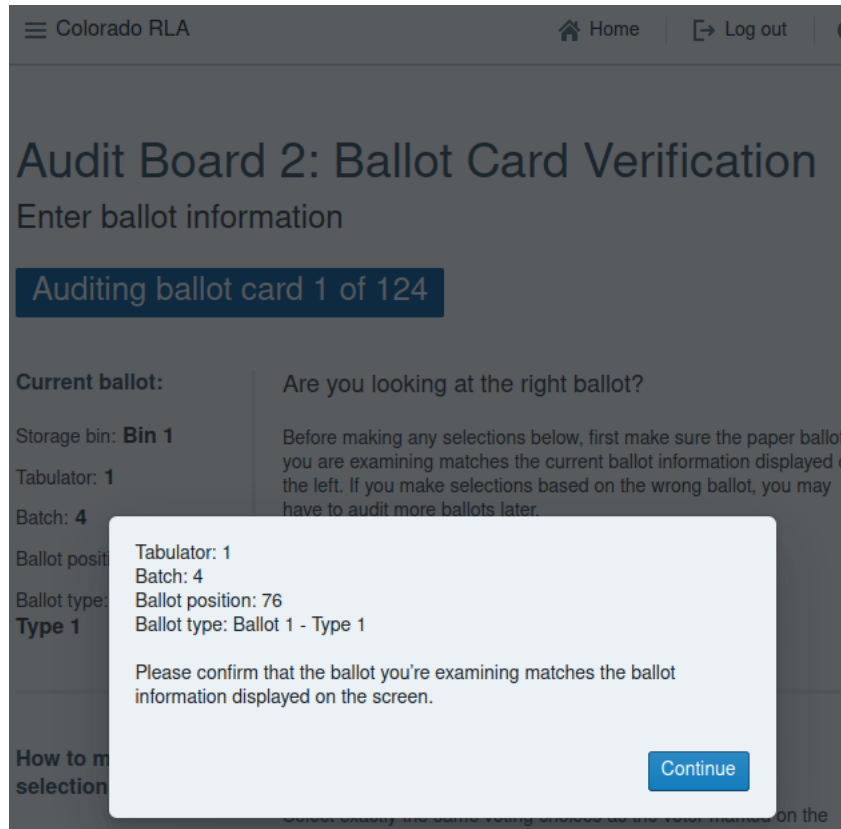
Figure A.15: Ballot card verification popup.

```
Response:
result "ACVR submitted"
```

Obviously the request matches what the audit board member chose as selections.

It then calls for the next one:

```
Endpoint: /cvr/id/14352
```

```
No request payload
```

```
Response:
d 14352
record_type "UPLOADED"
county_id 7
cvr_number 236
sequence_number 235
scanner_id 1
batch_id "5"
record_id 2
```

Audit Board 2: Ballot Card Verification

Enter ballot information

Auditing ballot card 1 of 124

Current ballot:

Storage bin: **Bin 1**

Tabulator: **1**

Batch: **4**

Ballot position: **76**

Ballot type: **Ballot 1 - Type 1**

Are you looking at the right ballot?

Before making any selections below, first make sure the paper ballot you are examining matches the current ballot information displayed on the left. If you make selections based on the wrong ballot, you may have to audit more ballots later.

Ballot not found - move to next ballot

How to match selections with ballot

Overvote	>
Undervote	>
Blank vote	>
Write-in	>
We can't agree	>

For each ballot contest:

Select exactly the same voting choices as the voter marked on the paper ballot you are examining.

Example 1: If the voter marked three candidates on their ballot in this contest, select the exact same three candidates below.

Example 2: If the voter did not vote for any of the candidates or choices in this contest, select "Blank vote – no mark"

IRV for Test County

Candidate 1(1)	Candidate 2(1)	Candidate 3(1)
Candidate 4(1)	Candidate 5(1)	Candidate 6(1)
Candidate 7(1)	Candidate 8(1)	Candidate 9(1)

Figure A.16: Audit CVR entry.

```
imprinted_id "1-5-2"
uri "cvr:7:1-5-2"
ballot_type "Ballot 1 - Type 1"
contest_info [ {...} ]
0 Object { contest: 14115, choices: [...] }
contest 14115
```

Figure A.17: Audit ballot submission summary.

```
choices [ "Candidate 1(1)", "Candidate 2(2)", "Candidate 3(3)", "Candidate 4(4)", "Candidate 5(5)", "Candidate 6(6)",
0 "Candidate 1(1)"
1 "Candidate 2(2)"
2 "Candidate 3(3)"
3 "Candidate 4(4)"
4 "Candidate 5(5)"
5 "Candidate 6(6)"
6 "Candidate 7(7)"
7 "Candidate 8(8)"
8 "Candidate 9(9)"
```

Looks the same as the one when audit was just starting, but now incremented for next sample ballot.

Round 1 in progress

0 of 1 counties have finished this round.

Choose report to download

County Updates   Contest Updates

County Updates

Click on a column name to sort by that column's data. To reverse sort, click on the column name again.

Filter by county name

COUNTY NAME ▼	STATUS ⇕ ▼	AUDITED DISCREPANCIES ⇕	NON-AUDITED DISCREPANCIES ⇕	DISAGREEMENTS ⇕	SUBMITTED ⇕	REMAINING IN ROUND ⇕
Adams	File upload deadline missed	—	—	—	—	—
Alamosa	File upload deadline missed	—	—	—	—	—
Arapahoe	File upload deadline missed	—	—	—	—	—
Archuleta	File upload deadline missed	—	—	—	—	—
Baca	File upload deadline missed	—	—	—	—	—
Bent	File upload deadline missed	—	—	—	—	—
Boulder	● Round in progress	1	0	0	1	248
Broomfield	File upload deadline missed	—	—	—	—	—

Figure A.18: DoS Dashboard: Audit in progress.

CDOS's view

The CDOS dashboard now shows Boulder's audit in progress, including the number of observed discrepancies (Figure A.18).

The /dos-dashboard call now includes information about the audit in progress.

```
Endpoint: /dos-dashboard

No payload.
Response:
Same as before, except that county_status 7 is updated:

7 Object { id: 7, asm_state: "COUNTY_AUDIT_UNDERWAY", audit_board_asm_state: "ROUND_IN_PROGRESS", ... }
id 7
asm_state "COUNTY_AUDIT_UNDERWAY"
```

audit_board_asm_state "ROUND_IN_PROGRESS"
audit_board_count 2
audit_boards Object { 1: {...} }
1 Object { sign_in_time: "2023-09-13T11:55:08.200720Z", members: [...] }
members [{...}, {...}]
0 Object { first_name: "Alice", last_name: "Alice", political_party: "Republican Party" }
1 Object { first_name: "Bob", last_name: "Bob", political_party: "Democratic Party" }
sign_in_time "2023-09-13T11:55:08.200720Z"
ballot_manifest_file Object { approximateRecordCount: 5, countyId: 7, fileName: "IRV-test-manifest.csv", ... }
cvr_export_file Object { approximateRecordCount: 316, countyId: 7, fileName: "CVR_Export_IRV_Example.csv", ... }
estimated_ballots_to_audit 591
optimistic_ballots_to_audit 592
ballots_remaining_in_round 248
ballot_manifest_count 312
cvr_export_count 312
cvr_import_status Object { import_state: "SUCCESSFUL", timestamp: "2023-09-08T01:11:35.165961Z" }
audited_ballot_count 1
discrepancy_count Object { AUDITED_CONTEST: 1 }
disagreement_count Object { }
audited_prefix_length 0
rounds [{...}]
current_round Object { number: 1, start_time: "2023-09-13T07:25:24.379146Z", expected_count 249, ...}