Democracy Developers
Colorado IRV RLA
Implementation Plan

Date: December 1, 2023
Version 1.0

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

Contents

Chapter 1

Introduction

We understand and support the principle of making minimal changes to existing `colorado-rla` code. We have tried to design with this as a priority, as much as possible. There are, unfortunately, several parts of the process where some interaction must be coded, because the logic is inherently collective.

- The *sample size calculation* must incorporate input from both the plurality RLAs and the IRV RLAs.

  This is relevant to both Round 1 samples and subsequent escalation samples. It is probably the most complex interaction between the two kinds of audits.

- The choice of which ballots to sample must include both IRV and plurality targeted contests.

  This is unavoidable because, at the UI end, the audit board must go and collect a single sample. It does not make sense to collect a plurality sample and a separate IRV sample—it is not necessary, and probably not acceptable, to double the workload for the audit board.

- The *escalation decision* at the end of Round 1 now needs to incorporate the IRV audits. This is relatively simple: if any audit, including any IRV audit, has not met its risk limit, then the audit needs to proceed to Round 2.

  Similarly, at the end of Round 2, there should be a statement of which (if any) audits have not yet met their risk limit, and this needs to be updated to include the IRV audits. This is straightforward.

- The *Audit reports* now need to include the IRV audits. The exact requirements here need to be discussed further with CDOS. There will be some reports that summarise all outcomes—these will need to incorporate IRV summaries too. There may also be a separate IRV audit report.

None of these work units are a consequence of IRV or RAIRE, just a necessity of incorporating audit data from two different sources.

The *colorado-rla* system contains an auditing engine, the vast majority of which is independent of the type of audit being undertaken. It would not be advisable, or necessary, to duplicate the functionality of this engine when integrating a new type of audit.

Integration of IRV audit functionality into *colorado-rla* will require use of classes, types, and utilities present in the existing system. IRV auditing functionality will need to make use of the existing colorado-rla model and utilities. Duplication of this extensive functionality would not be maintainable, and fortunately is not necessary.

1.1 Working prototype

The design in this document is accompanied by a working prototype, available at `https://github.com/DemocracyDevelopers/colorado-rla/tree/scratch/server/eclipse-project`.

It implements:

- CVR uploads that include IRV,

- sample size estimation for both plurality and IRV,

- assertion generation for IRV,

- assertion download as a json file for publication,

- upload of audit CVRs that include IRV,

- audit calculations (including discrepancy and risk calculations) that include IRV, and

- considerable, but not complete, tests.

This allows for a complete end-to-end run of a test audit that may include both plurality and IRV. We have run test audits through the entire process and would be delighted to give a demonstration.

Each section of this design document contains a pointer to the relevant part of the prototype code.

The prototype does *not* include:

- sophisticated user interfaces,

- IRV-specific reports,

- efficiency optimizations such as parallel RAIRE runs for assertion generation,

- complete logging,

- complete error handling, or

- complete test coverage.

It is important to understand that, although working, this prototype code is expected to be less robust than production code, and has not had the sort of QA or testing process that a production system would have. We are keen to receive feedback on its design and correctness, and expect that issues will arise as people try it out.

Note that the Democracy Developers branch of *colorado-rla* does not incorporate the changes pushed to *cdos-rla/colorado-rla* on Nov 18.

Downloading, compiling, running

The *Raire service* code is at `https://github.com/DemocracyDevelopers/raire-service`. It includes instructions for compiling and running both the raire service and the updated *colorado-rla* prototype—see the project README. We would be happy to help anyone compile and run it—please let us know if there are any issues.

Chapter 2

What colorado-rla does now

2.1   The CORLA Model

This explanation does not incorporate changes pushed to *cdos-rla/colorado-rla* on Nov 18, 2023.

Location: corla.model

This section provides a high level overview of how the existing colorado-rla system, at the time of writing, 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 may have implications for how IRV is integrated into colorado-rla.

2.1.1   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 (races) will involve only one county, where others are state-wide (voters from all 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,[1] 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 will be 10 for IRV and is typically equal to the number of allowed selections for plurality), and a sequence number. A *Choice* has a name, a textual description, and an boolean indicator of whether

---

[1]This seems to be unused—it may have been originally intended to allow for the description of non-plurality contest types.

the candidate is a qualified write in or is fictitious. Fictitious candidates are essentially ignored by the system.[2]

[Technical] Question 1. *What does the sequence number of a Contest refer to?*

Answer: This is probably a numeric identifier.

A *CountyContestResult* represents the result of a *Contest* that has taken place in a *County*. Note that state wide contests will involve a *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()*).

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 concepts to represent the total number of votes and ballots are defined–total votes accessed by the method *totalVotes()* and total number of ballots cast via the attribute *ballotCount*)

[Technical] Question 2. *What is the difference between* ballotCount *and the value returned by the method* totalVotes()? *The latter sums up all the first preferences for each candidate. Is the difference related to valid/invalid ballots?*

---

[2]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.

Answer: Most likely, this is related to votes being a count of valid votes sitting in each candidate's pile (which might include plurality contests where multiple selections are allowed, and would count zero for blank votes), where *ballotCount* is a count of ballots cast in the county for that contest (including blanks).

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 demoninator in the diluted margin computation.

A *ContestResult* is associated with a set of *County*s (multiple counties will participate in a state-wide contest). It also seems to be related to a set of *Contest*s. Unlike *CountyContestResult*, 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.

[Technical] Question 3. *How is a contest related to a set of contests (in the context of* ContestResult*)? Commenting suggests that the database has a separate* Contest *record for each county involved in the* Contest*.*

Answer: A *Contest* is created for each race taking place in a county, and if the race involves n counties, n *Contest* objects will be created, one for each county.

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.

2.1.2 Modelling of CVRs, Audited CVRs, and Discrepancies

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

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. Basically, we can think of a *CVRContestInfo* as capturing contest-specific details from a CVR. For each CVR and contest on that CVR, we can define a *CVRContestInfo* object.

[Technical] Question 4. *What is the meaning of the* ConsensusValue*?*

Answer: The audit board may disagree on what is on the ballot paper. This attribute is most likely used to indicate whether or not this was the case.

[Technical] Question 5. *Why are choices represented as Strings in* CVRContestInfo *(and other locations) but* Choices *in* Contest*? What are the implications for how they're stored in the database?*

Answer: It is most likely related to ease of seralization. Choices are represented as Strings in most places (except seemingly in *Contest*).

[Technical] Question 6. *Can we rely on the order of the list as a way of recording preference orderings? That is, if the list is stored in the database and then retrieved, is it guaranteed to have the same order?*

Answer: It seems to be serialized before it is stored, so this renders its reordering a non-issue. Nevertheless, it would be good to check.

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.

The actual selections are stored as a string. If there are multiple choices the choice strings are comma-delimited. 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.

A note on IRV integration IRV preferences are represented in the uploaded csv files with their ranks explicitly written, which can already be parsed by colorado-rla (after minor changes to acceptable headers). For example the sequence ["Alice(1)", "Bob(2)", "Chuan(3)", "Diego(4)"] Indicates a first preference for Alice, a second preference for Bob, etc. This is a workable format for storage.

However, in order to make canonicalization succeed, we believe it is better to store reported CVRs as an ordered list without explicit preferences. This would mean that the same preference sequence would be represented 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.

[Technical] Question 7. *What is the* sequence number *for a CVR?*

Answer: This 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.

[Technical] Question 8. *What is the* ballot type *attribute currently being used to represent?*

Answer: It most likely defines what contests are on the ballot. Not every voter is eligible to vote in every contest held in a county. (Note that some 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); and a PHANTOM_BALLOT.

[Technical] Question 9. *What is the difference between the record types PHANTOM_BALLOT and PHAN-TOM_RECORD?*

Answer: We believe the answer is that where a CVR exists for a ballot, but the matching paper ballot cannot be found, a PHANTOM_BALLOT is created to represent the ballot. Where a paper ballot exists, but there is no matching CVR, a PHANTOM_RECORD is created to form the CVR.

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.

[Technical] Question 10. *What is the* my_audit_flag *attribute used for? The commenting for this attribute is not quite clear.*

[Technical] Question 11. *What is the purpose of the* random number *stored in each* CastVoteRecord *object (as the attribute* rand*)? Is this related to ballot sampling?*

Answer: This most likely 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'.

[Technical] Question 12. *Are the* my_discrepancy *and* my_disagreements *attributes designed to represent the number of discrepancies and disagreements associated with this CVR to ACVR pair? What is the role of* AuditReason *in representing a discrepancy or disagreement?*

Answer: We think the answer is that 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'.

[Technical] Question 13. *It is apparent that ballots can be reaudited, and other parts of the code suggest that new* CastVoteRecord *objects may be created each time an ACVR for a ballot is submitted. Does the* count_by_contest *attribute reference the number of times the ballot has been audited? Is the* my_acvr *attribute replaced with the most recently submitted ACVR for the ballot? What is the role of the attributes and methods relating to 'counts' in this class?*

[Technical] Question 14. *The concept of 'unauditing' appears in* CVRAuditInfo *(i.e., the method* resetCounted() *has the comment 'clear record of counts per contest, for unauditing'). How is this used?*

Answer: We believe that 'unauditing' ballots relates to reauditing (ie., when auditors reaudit a sample of ballots, they are 'unaudited' and then new ACVRs submitted for that sample–see Section 2.4).

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.

[Technical] Question 15. *What is the role of a* Tribute? *What is the meaning of the ballot position, random number, and sequence position attributes?*

Answer: It looks like 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. Basically, 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'.

2.1.3 Modelling of Audits

Relevant classes: ComparisonAudit; Round; (CountyContestComparisonAudit)

Discussions with CDOS suggest that 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*).

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 actual disagreements are stored in a set, *Set<CVRAuditInfo>*.

Note that 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 current rate.

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.

[Technical] Question 16. *Why are there two methods for computing the discrepancy between a CVR and audited ballot in* ComparisonAudit *(i.e.,* computeDiscrepancy *and* computeAuditedBallotDiscrepancy*)? The latter is substantially longer than the former. (There is also a third for computing discrepancies relating to phantom ballots, but it's purpose seems more clear).*

Answer: 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).

[Technical] Question 17. *Are* ComparisonAudit *objects created only for contests targeted for audit, or for all? Is it the case that discrepancies are identified for all contests, but risk computations only undertaken for those targeted for audit?*

Answer: It appears that 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.

A note on IRV integration The current interface of *ComparisonAudit* could be used to represent IRV audits as well as Plurality. The only problem is that one CVR-ACVR pair may result in more than one discrepancy for a single IRV audit (e.g., it may represent an overstatement for more than one assertion being audited). The *getDiscrepancy()*, *computeDiscrepancy()*, *recordDiscrepancy()*, and *removeDiscrepancy()* methods in *ComparisonAudit* are all designed with the assumption that each CVR-ACVR represents one kind of discrepancy for the audit. However, it may be possible to assign to each CVR-ACVR the *maximum* computed discrepancy across the assertions being audited, and that this would suffice for how these discrepancy values are being used *outside* of the *ComparisonAudit* class. If the result of these methods are going to be used outside of the *ComparisonAudit* class for risk computations or sample recalculations, then that would be problematic in the case where a *ComparisonAudit* was actually an IRV audit. (However, this would also break the OOP principle of encapsulation, and would not be ideal in any case). Note that for reporting, it may be important to detail how each CVR-ACVR represents a discrepancy for each assertion – however that could be achieved with the use of IRV-specific report generation. Note that there *is* one place where risk computations are performed out side of the *ComparisonAudit* class for a given audit.

In *ReportRows.java*, a call to the audit math (risk computation) is made, grabbing required data from *ComparisonAudit* objects. If this was internalized in the *ComparisonAudit* class, then this class could be used to represent *any* kind of assertion-based audit.

See Section 2.7 for details and an illustrative commit.

A note on IRV integration 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.

2.1.4   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*. We need to know what these reports are to understand if there are any implications relating to the addition of IRV audits. They are both essentially wrappers around a String, with some additional details (a timestamp and name).

- 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 OPPORTUNISTIC_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 will need 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. The exception is when forming the refresh response – it may be that the client will want to display/report information that is specific to the IRV audits being undertaken.

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

A note on IRV integration The *AuditBoardDashboard* will need to have new UI for entering all the preferences on sampled IRV ballots.

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

2.1.5 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.

2.1.6 Modelling of Audit Reports/Reporting

Relevant classes: IntermediateAuditReportInfo; AuditInvestigationReportInfo.

[Technical] Question 18. *Do these classes have anything to do with the production of audit reports or are they more about logging?*

2.1.7 Other

Relevant classes: LogEntry; UploadedFile; ImportStatus.

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

2.2   Audit Math

Location: corla.math

Relevant classes: Audit.

The *Audit* class contains a collection of static methods. This class contains:

- 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 2.9]

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.

2.2.1   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

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

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[3] 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.

[Technical] Question 19. *Where does the computation in* ComparisonAudit::scalingFactor *come from?*

2.2.2   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.

We assume that this is not currently causing serious problems, because it doesn't 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.

---

[3]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 overstatements.

A note on IRV integration Our IRV RLA integration proposal neither corrects nor exacerbates this problem.

2.3 Persistence and Database Queries

Location: corla.persistance and corla.queries

We will need to identify what additional pieces of data will need to be persisted and stored in the database when IRV audits are added to the colorado-rla system. This will likely include:

- Assertions generated by RAIRE for IRV contests;

  - These will likely be represented by either two subclasses of an *Assertion* class (*NENAssertion* and *NEBAssertion*) or two classes that implement an *Assertion* interface.

  - They will need to include not only the assertions, but also the metadata associated (specifically, the margin and difficulty) as output by RAIRE.

  - Possibly, it will help to retain the contest metadata from RAIRE, including the overall highest difficulty and min margin.

- Objects representing IRV versions of *ComparisonAudit*s.

Hibernate annotation attached to classes is used to automatically construct database tables.

2.4 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).

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 associ-

ated 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 2.6.

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 2.7 for further details.

2.5 CSV Parsing

Location: corla.csv and corla.json

Currently, *colorado-rla* can parse CSV CVR export files for single- and multi-winner plurality contests.

IRV integration See Section 4.4 for discussion of ballot parsing and CVR storage that includes IRV.

2.6 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).

2.7 Reports

Location: corla.report

A note on IRV integration *ReportRows* contains a method for computing the risk limit achieved for a given *ComparisonAudit*. It grabs discrepancy counts from the *ComparisonAudit* object. It would be much better if this computation is internalised into *ComparisonAudit* itself so that *ReportRows.riskMeasurement()* can be agnostic as to what type of audit is being undertaken by just calling a *riskMeasurement* method within *ComparisonAudit*.

Specifically:

- Move the function currently called *ReportRows::riskMeasurement()* into *ComparisonAudit*. It would hardly need to change at all because it only uses data from the *comparisonAudit* anyway.

- At *ReportRows:386* (the only call to *ReportRows::riskMeasurement()*), change it from *riskMeasurement(ca)* to *ca.riskMeasurement()*.

An illustrative commit is at https://github.com/DemocracyDevelopers/colorado-rla/commit/ 92a5b4d2ab81bbbfdb8381c428b40fab741a31b7

2.8 Endpoints

Location: corla.endpoints

StartAuditRound:

- This endpoint involves a lot of operations over lists of *ComparisonAudit*. All these activities that are currently being done over Plurality audits will need to also be done over IRV audits. Note that, as we explain when we discuss our proposed design for integrating IRV audits, this does not mean that the code in *StartAuditRound* needs 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()* will initialize the set of contests being audited (via *initializeContests()*, initialize the set of audits being undertaken (via *initializeAudits()*) and the set of *CountyDashboard*s (via *initializeCountyDashboard()*).

- The set of contests under audit are initialised (*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*. Note that IRV audits will need to be initialised and assigned to the relevant *CountyDashboard* also.

- 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 – it looks like rounds are synchronized across counties.

[Technical] Question 20. *It looks like rounds are synchronized across counties, but is this the case even for single county contests?*

Answer: 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 would need to capture both Plurality and IRV audits.

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 may need attention to understand what implications IRV CVRs have on this process.

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 currently 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.DoSDashboard-RefreshResponse and json.CountyDashboardRefreshResponse].

CountyDashboardRefresh:

- This endpoints creates a *CountyRefreshResponse* which contains data about all audits being undertaken to be used to refresh a County Dashboard. This would need to include 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 would need to include 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).

  A note on IRV integration: Auditors may enter IRV rankings that form an invalid vote. Exactly what is entered by auditors, and the way in which the system interprets the entered rankings to form a valid vote, will need to be recorded. See Section 4.6 for details.

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

2.9 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 AuditBoardDashboardASM.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 2.1 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 COM-PLETE_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 2.2 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.
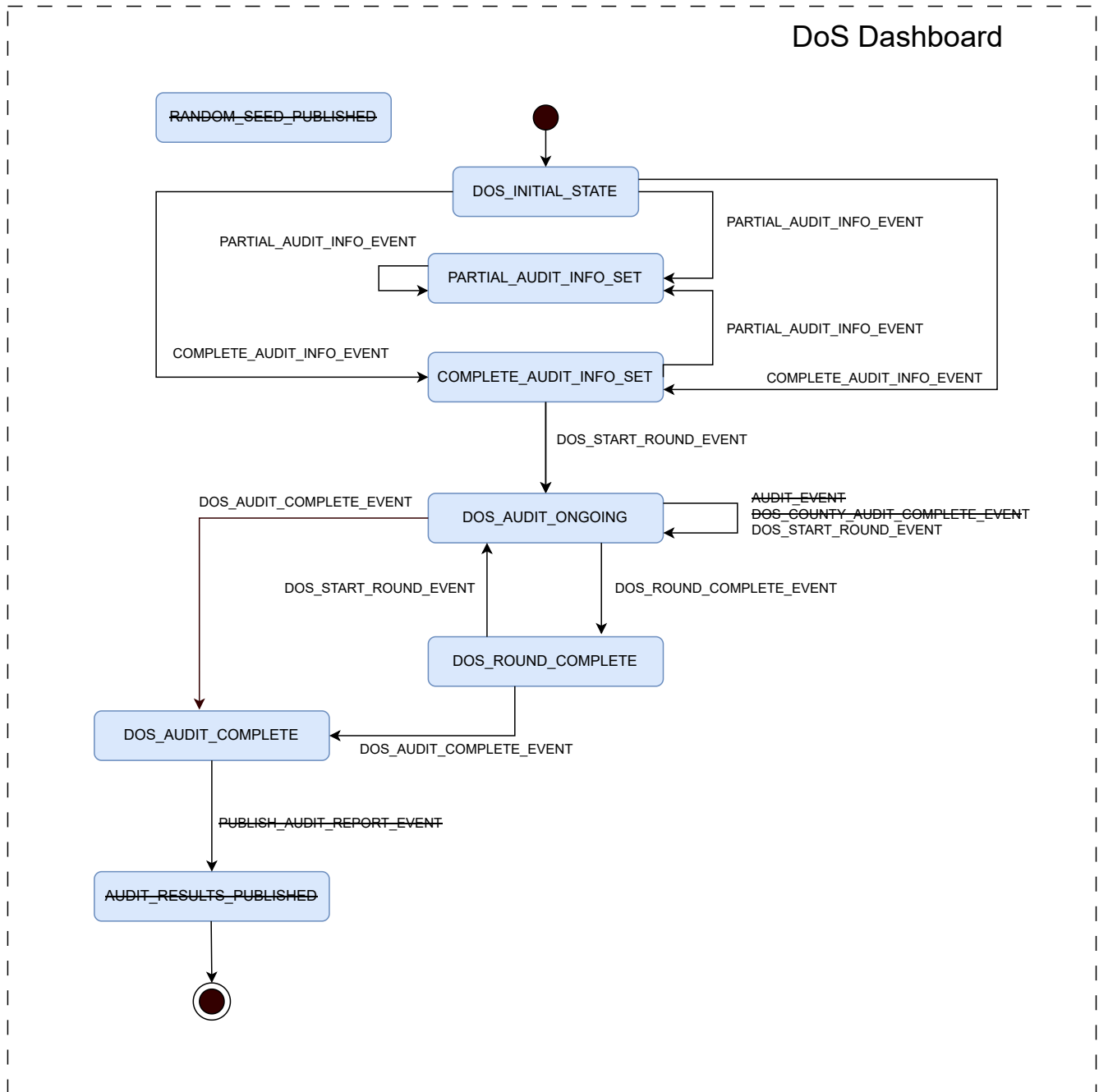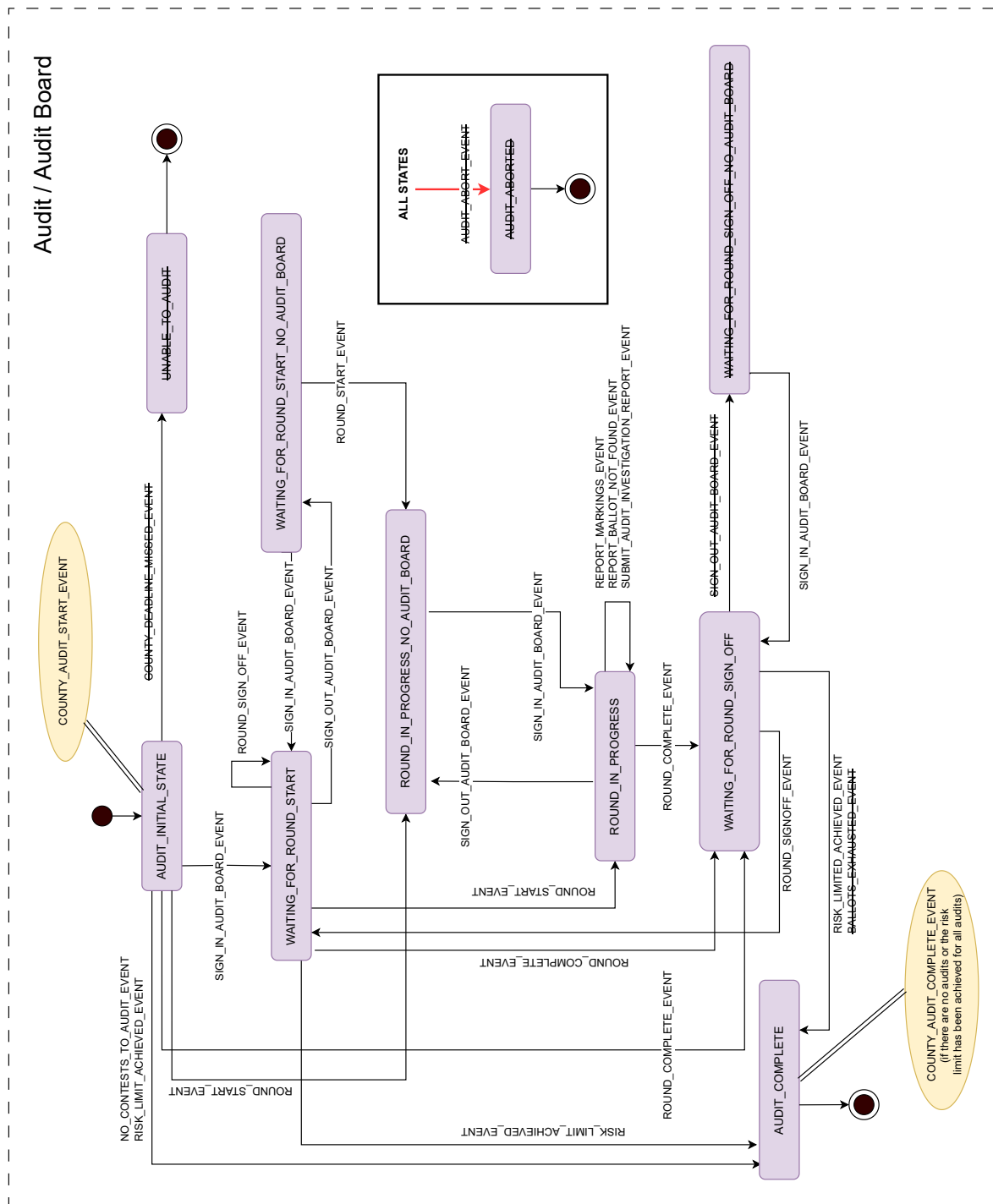
Figure 2.1: ASM for the DoS Dashboard (as extracted from the information in corla.asm). Crossed out states and events are those that are defined in the code but not used.

Figure 2.3 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:
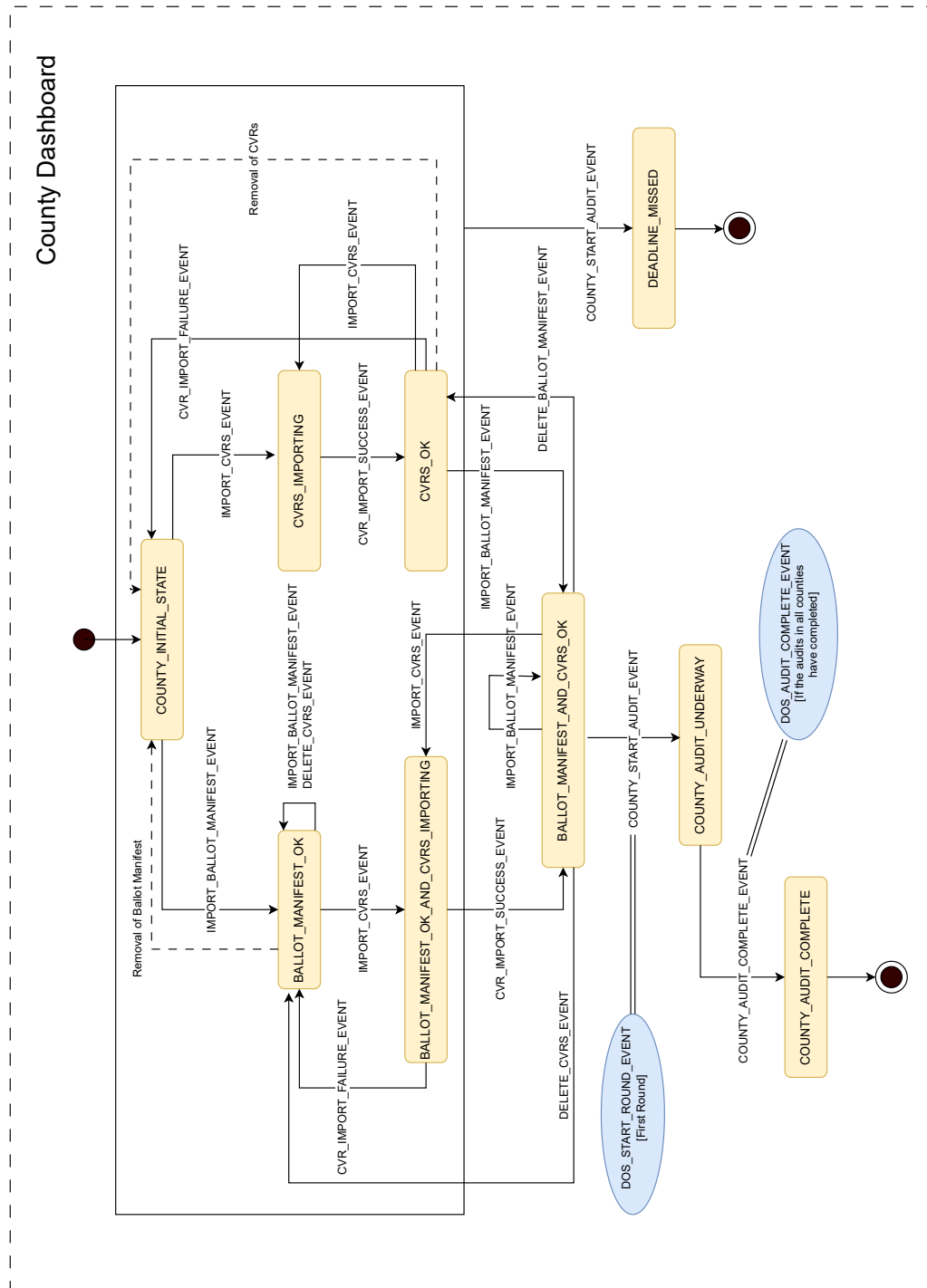
Figure 2.2: ASM for a County Dashboard (as extracted from the information in corla.asm). The dashed lines indicate transitions that are made in the code but not present in the ASM's transition function. The circled elements represent events from the DoS Dashboard ASM that are linked with events in a County Dashboard ASM (under certain conditions).

Figure 2.3: 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).

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 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.

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).

2.10 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—this is very simple and will not need to change for IRV. 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.

A note on IRV integration Because preferences are incorporated into choice names in the contest CVRs, (e.g. "Alice(2)" meaning a second preference for Alice), these steps will not automatically work for IRV if the IRV choices are simply stored as they appear in the CVRs.

Instead, a better option is to update the ballot parsing to remove the parenthesized preferences and use the order of choice names to retain preferences implicitly, for example, we would store ["Alice", "Bob", "Chuan", "Diego"] and interpret it to mean that Alice is the first preference, Bob is second, etc. This is described in Section 4.4.

This means that the canonicalization code will work for IRV ballots without needing any changes.

[Technical] Question 21. *Are the sampled ballots uploaded by the audit boards already canonicalized? That is, do they have contest and choice names that match the DOS-specified choice and contest names?*

Answer: Yes.

Chapter 3

Design proposal: User Perspective

3.1   Overview

Given the design of the existing system, IRV functionality is something that needs to be integrated into it as opposed to something that can sit entirely in a separate service. IRV audits will not be taking place independently of Plurality audits. They will be proceeding at the same time, and operating over the same ballots. The points at which the system needs to consider both Plurality and IRV audits include all instances where interaction with *ComparisonAudit* objects takes place, and where data collected from *ComparisonAudit* objects is used. The *ComparisonAudit* class is described in Section 2.1.3. Our proposed design has been conceived with the motivation of minimizing changes to existing Java classes, the overall audit logic, and the database structure. Our design also aims to utilize existing functionality where possible to avoid unnecessary duplication of code or the addition of redundant code.

At a very high level, the intended flow corresponds to the plan in our Guide To RAIRE. The RAIRE microservice generates assertions, which are audited using *colorado-rla*.

Since colorado-rla already involves significant complexity, careful thought is needed to understand how to introduce IRV audits in a way that minimally disrupts existing processes.

This section gives a high-level view of the proposed design, with an emphasis on the user experience. Figure 3.1 provides a high level sequence diagram depicting the RLA process from election data upload to report generation, identifying the components that will need to be added to support IRV audits and the existing components that will require modifications. Chapter 4 describes in detail how each new component will be implemented, and how any existing component will be altered.
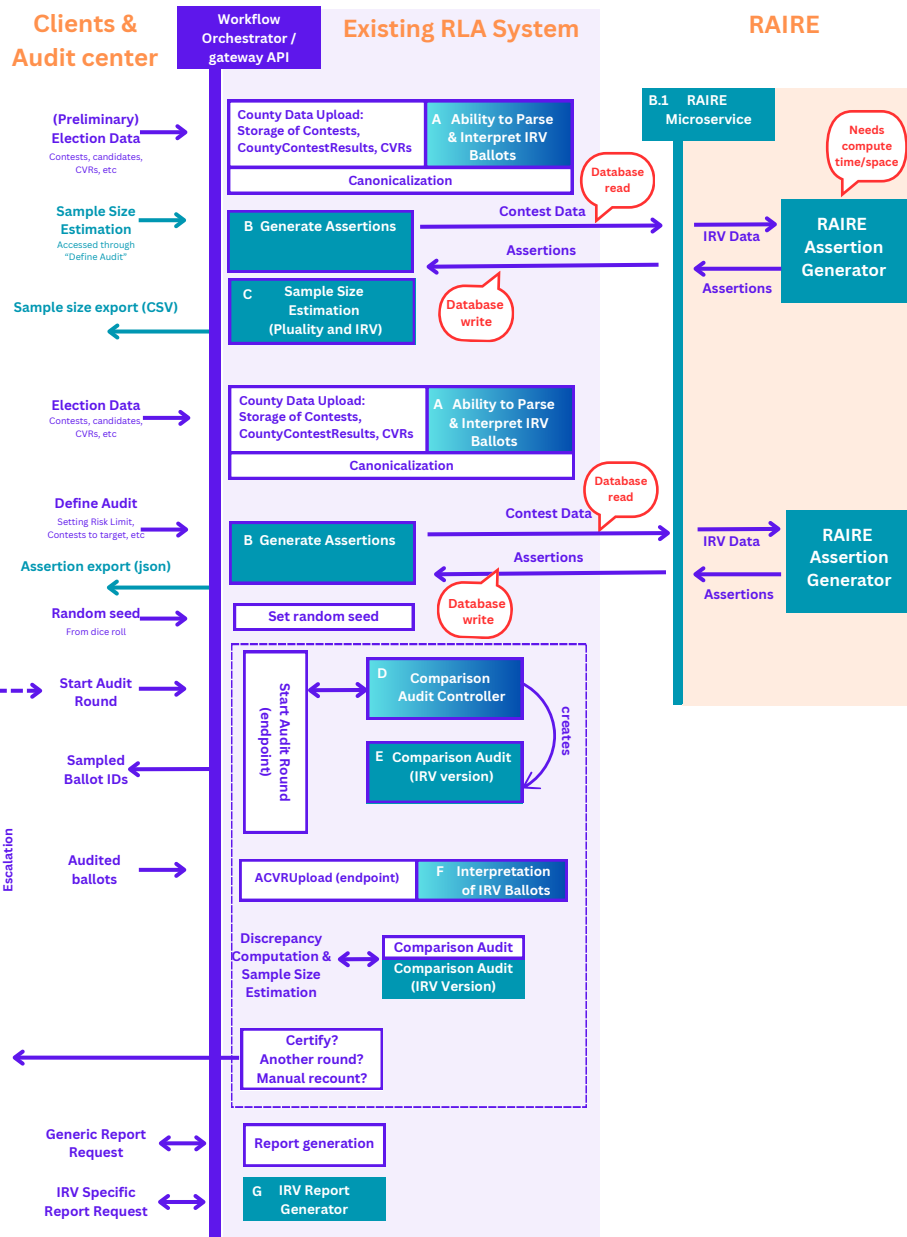
Figure 3.1: High level sequence diagram capturing the RLA process from election data upload to report generation, identifying new components required for IRV audits and instances where existing components required modifications. Purple denotes the existing colorado-rla system, green new components required for RAIRE IRV RLAs and shaded gradients denote existing code in colorado-rla that will need to be modified to incorporate IRV contests.

3.2   Setting up the audit data

County administrators will be able to upload their CVRs exactly as they currently do. A CVR may include a mix of plurality and IRV contests. New functionality will be added to ensure that IRV votes can be parsed, interpreted and stored appropriately. Importantly, this new functionality will take IRV votes that consists of a list of choice names with ranks, and store the vote as a list of candidate names *without* ranks. Similarly, where candidate names are stored in contest data structures, they will be stored without attached ranks. This design choice will allow the existing canonicalization process to proceed *without modification* for IRV contests.

3.2.1   Canonicalization

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

3.3   Before the audit: generating assertions and estimating sample sizes

The process of defining an audit will be modified to include the step of generating assertions for IRV contests, and (optionally) estimating the sample sizes required to audit all contests (both IRV and Plurality). These two steps will take place after the canonicalization of candidate and contest names, and before contests are selected for audit.

A microservice will be developed for the purpose of generating assertions following the algorithm described in the Guide To RAIRE: Part 2. The RAIRE microservice will be called from a new endpoint:

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

Canonicalization needs to run before assertion generation. In order to generate assertions, RAIRE needs all the Instant Runoff Voting (IRV) Cast Vote Records (CVRs) and the associated metadata (the number of votes, names of candidates, etc) for each IRV contest. This is stored in the colorado-rla database after CVR upload—the RAIRE microservice can either retrieve it from the database or receive it in an http request—the current design receives all the data for each contest in an http request.

In order to implement the new feature of sample size estimation, we will add a second new endpoint to colorado-rla.

New Endpoint: 2. *Estimate Sample Sizes* *This will produce 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.*

3.3.1   Database flush

After learning the estimated sample sizes for all contests, the database can be flushed. We can design the sample size estimation endpoint to avoid altering the database. The endpoint for generating assertions, however, will result in new data (assertions) being stored. When counties re-upload their data, any assertions that have been previously generated for contests involving that county will need to be cleared, and assertion generation repeated. If the deletion of a data file from the county side triggers the removal of assertions associated with contests in the county, then we could avoid the need for a separate database reset step.

3.3.2   Final CVR upload

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 does 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.

3.4   Defining the audit

3.4.1   Canonicalization and assertion generation

CVRs may change between preliminary sample-size estimation and audit day. It is advisable to run asssertion-generation again when final CVRs are uploaded. 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.

Canonicalization will have to be re-applied to the freshly-uploaded CVRs before the second round of assertion generation can occur.[1]

If assertion generation is done as part of defining an audit, then we can ensure that the correct sequence of steps (setting the risk limit, standardization of names, and then assertion generation) will always take place.

---

[1]It is possible that the canonicalization code could be edited to store and then re-apply some canonicalization instructions, but this is a general new feature rather than an IRV-specific one. If the CDOS Java development team were interested in implementing it, we would be happy to advise, but we do not feel it fits within the scope of the IRV feature.

In order to allow for accurate counting of meaningful discrepancies, assertions should be generated for all IRV contests that appear on any card that might be audited, which in practice probably means all contests. This can start as soon as all final CVRs are uploaded.

Sometimes assertion generation may take some time, though usually it should be very fast.

[Requirements] Question 22. *How should RAIRE's possible failure modes be communicated to the CDOS dashboard?*

These may include:

- *that the outcome is a tie and therefore cannot be audited*

- *that RAIRE timed out without finding a valid set of assertions (this is very unlikely but needs to be considered)*

Answer: Communicate the error back to the user in the corla/cdos UI. Log progress, including good progress. Consider how RAIRE can give an idea of its progress.

[Requirements] Question 23. *Should RAIRE provide status updates that are communicated to the CDOS dashboard? What should they include?*

Observation: Based on the prototype implementation, it appears that the overwhelming majority of the time is spent retrieving CVRs from the database, rather than performing the RAIRE assertion generation itself. This is actually much easier to design progress-related UI for.

The output is a list of assertions in JSON (or whatever other format is required). A previous example is the San Francisco DA assertions.

The final assertions will be:

- stored in the *colorado-rla* database for use during the audit, and

- returned as JSON to the CDOS UI.

The JSON assertions can be exported somewhere for human validation and visualisation, for example to the audit center website.

3.4.2   Choosing Target contests

Choosing target contests would work the same for IRV as for plurality, except that it would help if the UI showed the CDOS UI which contests were IRV.

3.4.3 Non-auditable contests

Currently, a 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.

3.5 Starting and Running the Audit

3.5.1 'Controlling' Comparison Audits

The set of comparison audits being undertaken by *colorado-rla* is represented by a collection of instances of the *ComparisonAudit* class. Functionality for estimating sample sizes and computing discrepancies between CVRs and audited ballots for a contest sit within this class.

A single static class, *ComparisonAuditController*, is responsible for creating these instances. We describe this controller in Section 2.4. This class will need to be updated to control IRV comparison audits too. Under our proposed design, this will not be difficult. Each comparison audit's calculations are performed as part of the class *ComparisonAudit*, for which we will need an IRV child class: *IRVComparisonAudit*. The controller will need to create an instance of *ComparisonAudit* for Plurality contests and an instance of *IRVComparisonAudit* for IRV contests. Where the system performs operations over a collection of *ComparisonAudit*s, it does not need to know which of these are IRV audits and which are Plurality. See Section 4.3.3 for implementation details.

3.5.2 Choosing a random seed, Selecting random ballots

The existing code for inputting a random seed and outputting a list of sampled ballots will be used for both plurality and IRV. Indeed, they'll be the same samples, just as colorado-rla already takes the same samples when multiple plurality elections are targeted. The IRV ones will simply fit in with the existing sample selection.

Very little about the actual *audit* process needs to change for IRV. The size of the samples required by each contest is determined within its associated instance of the *ComparisonAudit* or *IRVComparisonAudit* class.

[Technical] Question 24. *Should we assume that the predicted numbers of over- and under-statements will be the same for plurality and IRV, or should we ensure that* IRVComparisonAudit *can have different assumptions from the parent?*

Answer: Let's get empirical data from Boulder '23.

This is relevant to the computation of optimistic sample sizes.

The audit is still valid if incorrect guesses are made, but it may require more samples, or may require a second round rather than completing successfully with one.

3.5.3 Auditing ballots

When the audit board has finished entering all the data from the sampled ballots, they upload them.

The UI team will need to update the client UI to allow the audit board to input what they see on sampled IRV ballots. We'll discuss this with the UI team. The audit board will have to enter the full list of preferences that they see.

The resulting vote records, for all sampled IRV ballots, need to be uploaded to colorado-rla. The existing ACVR endpoint will need to be expanded to incorporate and interpret IRV ballots, though the user experience will be almost unchanged.

Interpretation of invalid IRV preferences   If the audit board sees an invalid IRV ballot, they should enter exactly what they see. We will write code to interpret this according to Colorado's ballot interpretation rules. See Section 4.5 for details.

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*.

3.6 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 2.2.

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

3.7 Certification, Round 2 or Manual Recount

The main audit process now needs to decide what to do next: certify the result, escalate to Round 2, 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.

This code will not need to change for IRV - it simply needs to incorporate the IRV-specific sample size estimations computed by *IRVComparisonAudit*. The existing code that does this by interacting with collections of *ComparisonAudit*s will be able to do this for IRV as well, since *IRVComparisonAudit* can be present in those lists as a subclass of *ComparisonAudit*s.

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.

3.8   Audit reports

We will discuss with CDOS as to what these reports should contain and how they should be exported. Again, it might be valuable to try to reuse some of the code in the current system.

Assertions can be exported as JSON as soon as they are generated, or at any subsequent point.

[Requirements] Question 25. *What should be in the IRV audit reports?*

Chapter 4

Design Proposal: 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 preferences, 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 incorporates *raire-java* as a submodule. It connects the details of Colorado RLAs with the general implementation of the RAIRE assertion generation. It is a REST API that receives vote data from *colorado-rla*, translates it into a form appropriate for RAIRE assertion generation, calls the RAIRE assertion-generation API, and returns the results via http.

  DD Answer: We have decided to implement it as a REST API, because this makes automated testing much easier. We are happy to revise this decision if a different structure is preferred. See Section 5 for details of testing for this component.

An implementation of the RAIRE microservice is available at: https://github.com/DemocracyDevelopers/raire-service, which includes *raire-java* as a submodule. Instructions for building, testing and running it are in the README.

4.1.1   RAIRE Assertion Generation

Endpoint: raire-rs or raire-java   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
}
```

Each vote record consists of a count, followed by a list of integer preferences in order (first preference 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 who listed Chuan as first preference, Bob as second preference, and Alice as third preference.

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.

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 are described in Section 4.1.4. Once RAIRE finds assertions, it will apply internal algorithms for filtering out those that are *redundant*. This process is referred to as *trimming*.

```
'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]

---

[1], but this is not always true for other audit types.

Constraints: Assertions may be regenerated any time before *COMPLETE_AUDIT_INFO_SET* is true. Newly generated assertions will simply replace older assertions for the same contest. After *COMPLETE_AUDIT_INFO_SET* is true, no assertion regeneration may occur. The current prototype does not (yet) enforce this constraint.

4.1.2 RAIRE microservice

Endpoint: raire-service::cvr/audit

Input: The *RAIRE microservice* has one API endpoint, at */cvr/audit*. It takes a description of a single IRV contest in JSON, including a contest name, time limit for assertion generation, list of candidates, and (optionally) universe size. Votes are given as a list of lists of candidate names. An example is shown below.

```
{ "contestName": "GuideToRAIREExample1",
"totalAuditableBallots": 15,
"timeProvisionForResult": 10,
"candidates": ["Alice","Bob","Chuan","Diego"],
"votes": [
  ["Alice","Bob","Chuan","Diego"],
  ["Alice","Bob","Chuan","Diego"],
  ["Alice","Bob","Chuan","Diego"],
  ["Alice","Bob","Chuan","Diego"],
  ["Alice","Bob","Chuan","Diego"],
  ["Bob","Alice","Chuan"],
  ["Bob","Diego","Alice"],
  ["Chuan"],
  ["Chuan"],
  ["Chuan","Diego"],
  ["Chuan","Diego"],
  ["Diego","Chuan"],
  ["Diego","Chuan"],
  ["Diego","Chuan"],
  ["Diego","Chuan"]
  ]
}
```

Translation to RAIRE API call The *totalAuditableBallots* field will be 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 will be 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.

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

Output: The output is the same as the RAIRE assertion generation step, except that each assertion's *winner*, *loser* and *assumed-continuing* candidates are referred to by name (a String) rather than by index.

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.[2]

4.1.3   Parallelism and efficiency

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

The prototype does not currently take advantage of this parallelism. We find that the database retrieval takes much longer than assertion generation anyway, so that step is the priority for parallelization.

4.1.4   Error handling

RAIRE produces several possible errors.

Indications of invalid input

The following indicate programming errors. These should cause a runtime exception because they are not expected to occur.

- **InvalidTimeout:** the given timeout was not a positive integer.

- **InvalidCandidateNumber:** the candidate numbers were invalid.

---

[2]The current version of the explainer and the current format exported from the prototype do not quite align, so this functionality does not work right now, but will be updated soon.

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

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.

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

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

- **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.

- **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.

- **InternalErrorDidntRuleOutLoser:** similar to being unable to rule out a specific elimination sequence with an unexpected winner.

- **InternalErrorRuledOutWinner, WrongWinner** should not occur in our setting where we do not give RAIRE an expected winner.

There is also a flag **warning_trim_timed_out,** which is true if the assertion-trimming phase timed out. This means that the assertions are valid, and are efficient to audit, but may contain a large number of redundant assertions. In this case, if there is time, it is worthwhile to rerun the trimming process.

These should generally be returned to the user, because the user can act upon them—generally by deciding that the only way to verify that contest is by a full hand count. For some cases (such as *warning_trim_timed_out*) the user may decide to rerun the trimming algorithm with more time.

4.2   New Endpoints in colorado-rla

4.2.1   Generate Assertions

Endpoint: colorado-rla::generate-assertions

Purpose: For the set of all IRV contests, identified by their name, this endpoint will access the RAIRE microservice (Section 4.1) to generate assertions for those contests. The RAIRE microservice will return these assertions in JSON format, which will subsequently be stored by this endpoint into 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 microservice.

Input: None. It reads the list of contests from the database and finds all the IRV ones.

Output: Either a success value or an error.

The assertions are also stored in the database.

Considerations:

- For opportunistic reporting of discrepancies for all IRV contests, assertions will need to be 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 cleared from the database and replaced.

A prototype of the Assertion Generation endpoint is available at: `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/endpoint/GenerateAssertions.java`

4.2.2   Export Assertions

Endpoint: colorado-rla::export-assertions

Purpose: Provide a json download of the assertions generated by the Generate Assertions endpoint, along with associated metadata including contest name, candidate names, minimum margin and expected difficulty.

Input: None. It reads the assertions from the database.

Output: The assertions and associated metadata, in the same (or similar) JSON format produced by RAIRE.

Considerations:

- These assertions are intended for public display and should be human-readable and compatible with an assertion explainer.

A prototype of the Assertion Export endpoint is available at: https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/endpoint/ExportAssertions.java

4.2.3 Sample Size Estimation

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

Purpose: Compute estimate 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. However, if there is a need or desire to store the estimated sample sizes in the database we can do that.

Input: None.

Output: A CSV file will be produced and automatically downloaded. The file will, at least, contain 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;

- Diluted margin;

- Estimated sample size (assuming no discrepancies will arise);

For IRV contests, each associated assertion will have its own diluted margin. We intend to report the smallest diluted margin assigned to an assertion for an IRV contest, in this CSV file export.

Design: This endpoint will implement the following steps:

1. Create *ContestResult* objects for each contest in the database. This is achieved using the existing *ContestCounter* 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 2.1.1. The existing *ContestCounter* controller is described in Section 2.4.

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:

- 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).

A prototype of the new sample-size estimation endpoint is available at: `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/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.

A new endpoint will be required for generating IRV-specific reports, such as exporting IRV contest assertions in an accessible format (such as JSON, see Section 4.2.2).

The nature and content of these reports will require further discussion with CDOS.

4.3   New Classes

4.3.1   Contest Type (Enumeration)

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

A prototype is at https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/ContestType.java.

4.3.2   Assertions

Three classes will be added to represent assertions:

- An abstract base class *Assertion*

  Prototype:

  https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/Assertion.java

- A subclass of *Assertion*, *NEBAssertion*

  Prototype:

  https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/NEBAssertion.java

- A subclass of *Assertion*, *NENAssertion*

  Prototype:

  https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/NENAssertion.java.

Assertion base class

Key Attributes:

- Id (Long)

- Contest name (String)

- Winner (String)

- Loser (String)

- Margin (int)

- 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:

- (Constructor) *public Assertion(String contestName, String winner, String loser, int margin, long universeSize, double difficulty, List<String> assumedContinuing)*

  The diluted margin for the assertion is computed in the constructor by dividing its margin with the universe size.

- *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 (BigDecimal riskLimit, 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 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.

- *public void recordDiscrepancy(final CVRAuditInfo the_record)*

  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.

- *public void removeDiscrepancy(final CVRAuditInfo the_record)*

  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).

- *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, Integer one_vote_overstatements, Integer two_vote_overstatements)

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

The computation of discrepancies is assisted by a private method:

private OptionalInt computeDiscrepancyPhantomBallot(final CVRContestInfo cvrInfo)

This method evaluates the worst case discrepancy that could arise given that we know the CVR vote for a contest but are missing the physical paper ballot.

Assertion subclasses: NEBAssertion; NENAssertion

Each assertion subclass will provide its own implementations 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 will be captured in a subclass of the existing class *ComparisonAudit*. This will allow 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

Prototype:

`https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/IRVComparisonAudit.java`.

Extends: ComparisonAudit

Key Attributes:

- Universe size (long)

- Assertions (List<Assertion>)

Key Methods:

- (Constructor) *public IRVComparisonAudit(final ContestResult contestResult, final BigDecimal riskLimit, final BigDecimal gamma, 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. It will set the universe size for the contest to the total number of ballots stored in the given *ContestResult*. 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.

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 auditedCVR)*

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 the_type)*

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.

4.3.4   Assertion-related Queries

An *AssertionQueries* class will be added to the *queries* folder of colorado-rla.

Prototype:

https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/
src/main/java/us/freeandfair/corla/query/AssertionQueries.java

Key methods:

- *public static List<Assertion> matching(final String contestName)*

   Returns a list of all assertions generated for the contest with the given name.

4.4   Parsing and storage of (IRV) CVRs

Colorado-rla already incorporates an option for multiple candidate selections. This can be 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 preference first, second preference second, etc.

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

A prototype of the parser edits is available at: https://github.com/DemocracyDevelopers/
colorado-rla/blob/scratch/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-preference 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 preferences 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 preferences that are not valid unambiguous IRV preferences. 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 preference list. The above example would be "Alice(1), Bob(2)."

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

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

The second step requires specific code to interpret invalid preferences according to Colorado's statute (`https://www.sos.state.co.us/pubs/rule_making/CurrentRules/8CCR1505-1/Rule26.pdf`) and CDOS clarification rules. Prototype code at `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/IRVBallotInterpretation.java` removes duplicates first (that is, when multiple preferences have been given to the same candidate) and subsequently removes every preference after a skipped or overvoted preference. Core IRV parsing utilities are provided in prototype code at `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/util/IRVVoteParsing.java`. This class contains static methods that make use of the interpretation rules in *IRVBallotInterpretation* to translate a vote "Alice(1), Bob(2), Chuan(3), Diego(3)" to its valid and final form "Alice, Bob."

The resulting valid preference list can then be stored unambiguously as an ordered list of candidate names.

New Classes

IRVPreference: `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/IRVPreference.java`

Stores a (name, rank) pair.

IRVChoices: `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/IRVChoices.java`

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

IRVBallotInterpretation: `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/model/IRVBallotInterpretation.java`

Contains a static method (*InterpretValidIntent*) that takes an *IRVChoices* and returns a new *IRVChoices* representing a valid sequence of choices (candidate names without ranks, sorted by preference). An instance of this class can also be created to store an interpretation–raw choices and interpreted choices– for the purpose of persisting a record of this interpretation in the database.

IRVVoteParsing `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/corla/util/IRVVoteParsing.java`

Contains a series of static methods providing utilities for parsing IRV votes in CVR export data files, and on audited ballots.

4.4.2 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 preference n.

4.5 Ballot Interpretation Database storage

Key idea: the Audit Board will enter the IRV ballot data exactly as they see it; the system will record both the raw data and the valid 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 preferences. Instead, the audit system will store both:

- 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").

New classes

IRVBallotInterpretation, as above.

4.6 Running the Audit

IRV comparison audits, with correct implementations of sample size estimation, discrepancy management, and risk computation, can be 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. Provided the correct constructor is used when creating comparison audit objects (*ComparisonAudit* for Plurality and *IRVComparisonAudit* for IRV), most other audit logic that deals with *ComparisonAudit*s need not change with the introduction of IRV. This includes the logic for starting and signing off an audit round (*StartAuditRound* and *SignOffAuditRound*).

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 will be important to have a record of exactly what the audit board said was on the ballot, and how the system interpreted this information. We propose to do this by introducing a new database table to store this information, as described in Sections 4.5 and 4.7.

Discrepancy calculation will then act 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 will 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 propose 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, will be 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 propose 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 will 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) will be stored in the database as per Section 4.7.4.

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

Prototype code for the IRV-inclusive *ACVRUpload* endpoint is at `https://github.com/` `DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/` `us/freeandfair/corla/endpoint/ACVRUpload.java` where the method *recordIntepretations* takes a *SubmittedAuditCVR*, and for every IRV vote recorded on the audited ballot, records the interpretation to the database.

4.7   Database Tables

The integration of IRV into colorado-rla will result 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) will need to ensure that data in any new tables is reset as desired. Note this is not implemented in the prototype.

4.7.1   Table: comparison_audit

A single table will be used to store all comparison audits (Plurality and IRV). In light of the addition of a *ComparisonAudit* subclass, the *comparison_audit* table will now include two additional columns: a discriminator column used to identify the type of audit being undertaken (IRV or Pluarlity); and a column for the size of the ballot universe for the audit. This latter column arises as this universe size is an attribute of the *IRVComparisonAudit* class.

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 will be used to store the type of contest (IRV or Plurality). Consequently, this table will not change when IRV is integrated into colorado-rla.

4.7.3   Assertion-related tables

Three tables will be 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 will be 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 will be 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_cvr_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.

- Table: *assertion_context*

  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 will be stored in a table

assertion_context 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 propose adding a table to store the details of the auditor-entered ranks for IRV contests, and how the system interprets those votes to form a valid ranking. This table will need to store: 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.

4.8   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.

It is important, however, that if the user has generated assertions during *Define Audit*, that they are either not permitted to repeat this step (unless counties have uploaded new data and existing assertions have been cleared from the database), or if they want the ability to re-generate assertions then existing assertions must be cleared from the database. We must also ensure that neither sample size estimation, or launching of an audit, can take place without assertions having been generated.

4.9   UI Changes

We anticipate that the following changes will need to be incorporated into the colorado-rla client:

- In the *Define Audit* step, access to assertion generation and sample size estimation will need to be provided.

- When uploading an audited ballot, auditors will need appropriate UI to enter what they see on the ballot for an IRV contest.

  - The *ACVRUpload* endpoint will need to 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. A record will be kept of the exact data provided by the audit board, as well as how the system interpreted this data.

  Prototype code is at `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/client/src/component/County/Audit/Wizard/BallotAuditStage.tsx`.

- For IRV-specific reports, the UI will need to provide access to the generation of those reports.

4.10   Additional Changes to Existing Code

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

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

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

- Several attributes in *ComparisonAudit* will need to be made *protected* as opposed to *private* so they can 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). Alternatively, the *ComparisonAudit* class will need protected methods for setting these attributes. The *recalculateSamplesToAudit()* method in *ComparisonAudit* is currently private. It will need to be made protected so that it can be overridden in *IRVComparisonAudit*. By overriding this method, many of the other methods implemented in *ComparisonAudit* will not need to be reimplemented for IRV, but can be used as they are.

  Prototype code for *IRVComparisonAudit* is at `https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/src/main/java/us/freeandfair/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, en-

capsulation is broken by not bundling this method with the data that it uses. Data from *Comparison Audit*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 the prototype classes.

· 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 method 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.

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

Chapter 5

Testing and Verification

5.1   Unit tests

All our new classes will have substantial unit test coverage. We are currently using JUnit, but would be happy to switch to any other open test platform.

As an example, the code for valid interpretation for IRV ballots already has near-complete test coverage, including

- tests that validity is correctly assessed for some valid and invalid examples,

- tests that the correct sorted list of preferences are returned for some valid examples,

- tests that the correct valid interpretation is made for some invalid examples, including those in the Voter Intent Guide and others generated by us.

This is at:

https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/
src/test/java/us/freeandfair/corla/model/IRVBallotInterpretationTest.java.

- a systematic test for all possible valid and invalid ballots on 5 candidates, to check that the valid interpretation procedure always produces valid output.

This is at:

https://github.com/DemocracyDevelopers/colorado-rla/blob/scratch/server/eclipse-project/
src/test/java/us/freeandfair/corla/model/SlowExhaustiveIRVBallotInterpretationTest.java.

Other IRV-specific tests have been included in the existing test folder at https://github.com/DemocracyDevelopers/colorado-rla/tree/scratch/server/eclipse-project/src/test/java/us/freeandfair/corla.

5.2 API tests

The RAIRE assertion generator is a REST API, but the other components generally retrieve information from the database and then do something. This means that unit tests need to include database setup (and teardown).

5.2.1 Test data

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 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 endpoint being tested, we produce the data either as CSVs, to match the expected upload from Colorado counties, or as json, to test the RAIRE assertion-generation endpoints.

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 preferences are invalid (with duplicates, overvotes, or skipped preferences) the valid interpretation is stored.

Real IRV examples from Australian IRV data

The Australian state of New South Wales makes detailed vote preference 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.

Extreme test cases to assess limits of behaviour

In general, the RAIRE algorithm finds assertions for most contests quickly, even those with a large number of candidates. There are wicked data sets that we have collected on which RAIRE does not terminate. Some of these are STV data sets that we have reimagined as IRV, and others are ranked-choice data sets obtained from Preflib[1] that did not originate from any kind of election. Australian STV contests typically have a large number of candidates (over 100). The use of STV data, reimagined as IRV, will allow us to stress test our RAIRE implementation on contests with extremely large candidate numbers.

Combination tests

We will generate 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.

5.2.2   Test targets

There are several APIs for which test cases will be constructed:

- RAIRE assertion generation This is a REST API. There is already a significant test suite at `https://github.com/DemocracyDevelopers/raire-java/tree/main/src/test/`.

- RAIRE microservice. Also a REST API.

- New colorado-rla endpoints

- Existing colorado-rla endpoints

We intend to work with CDOS QA on these tests, particularly the tests of existing *colorado-rla* endpoints to ensure that existing tests all still pass.

---

[1]www.preflib.org

5.3 Use of Selenium for automated end-to-end system testing

Tests will include testing of each error mode, normal operation, stress testing with lots of contests in parallel, testing instances that we know will result in a very small, and very large, number of assertions, testing with instances that we know will result in only NEB, only NEN, and a mix of both.

CDOS QA suggested Selenium as the most appropriate testing framework—we are happy with this, or any other open framework.

Chapter 6

Timeline

A draft timeline is available through our Project Plan at `https://github.com/orgs/DemocracyDevelopers/projects/1/views/2`.

The timeline will be updated once the design is agreed upon and development can commence.

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"
```

Figure A.1: Administrator login to DoS Dashboard.

```
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.

Endpoint: DoSDashboardRefresh.java (/dos-dashboard)

```
No Request payload.
```

```
Response:
asm_state "PARTIAL_AUDIT_INFO_SET"
audited_contests Object { }
```

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

```
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"
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
```

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

```
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):

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

```
Request payload:
election_date "2023-09-07T02:37:39.945Z"
election_type "general"
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.
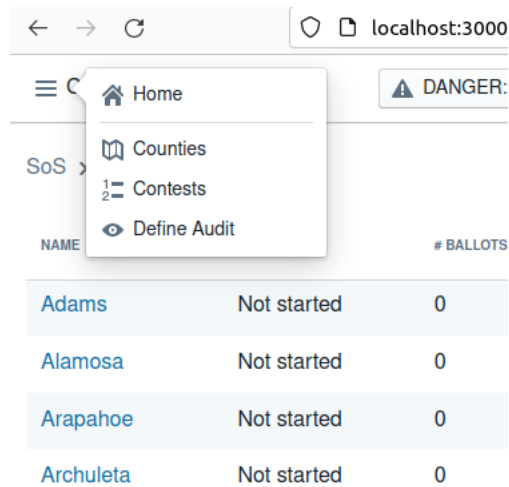
Figure A.5: DoS Dashboard: Defining the audit.

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"
```

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
```

Administer an Audit

Election Info

Enter the date the election will take place, and the type of election.

Election Date

9/7/2023

Election Type

○ Coordinated Election
○ Primary Election
● General Election
○ Recall Election

Public Meeting Date

Enter the date of the public meeting to establish the random seed.

Public Meeting Date

9/8/2023

Risk Limit

Enter the risk limit for comparison audits as a percentage.

Comparison Audits (%)

5

Contests

Drag and drop or click here to select the file you wish to use as the source for standardized contest names across jurisdictions.

File requirements:

- File must be CSV formatted, with a *.csv* or *.txt* extension. Other file types are not accepted
- The file must contain a header row consisting of *CountyName* and *ContestName*.
- The file may not contain duplicate records

Ready to import:

IRV_Test_Canonical_List.csv (2032 bytes.)

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

Figure A.7: Selecting contests for audit.

Figure A.8: Choosing the random seed.

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

Figure A.9: Launching the audit.

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)

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

```
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"
enabled_ui_events []
```

Endpoint: ContestDownload.java (/contest)

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

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
imprinted_id "1-3-1"
cvr_number 79
db_id 14195
ballot_type "Ballot 1 - Type 1"
storage_location "Bin 1"
audited false
```

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
```
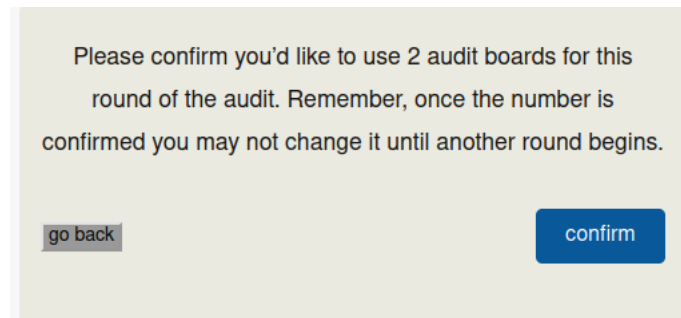
Figure A.11: Confirming audit boards.

```
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]
```

Hello, Boulder County!

> ℹ **You may now perform round 1 of the audit.**

Ballot Manifest

File name	IRV-test-manifest.csv
SHA-256 hash	C60B60B5F8E29C867AA3EE5968AC0E399A331E1138B029076...

✔ File successfully uploaded 07/09/2023, 12:50:53 pm [Download]

CVR Export

File name	CVR_Export_IRV_Example.csv
SHA-256 hash	75A0ECA27564BB9419120C04BF53C9A964FBB3C80AA248AA8...

✔ File successfully uploaded 08/09/2023, 11:11:33 am [Download]

Intermediate audit report (CSV)

[Download]

List of ballots to audit (CSV)

[Download]

How many audit boards will be auditing?

[2] [Enter]

There are **249** ballot cards to audit in this round.

Sign in to an audit board

[👥 Audit Board 1] [👥 Audit Board 2]

Figure A.12: Audit board sign up buttons.

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.

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

Figure A.13: Audit board sign in.

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

```
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]]"
```

I'm a little confused about 'index 1' because I chose Audit board 2.

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

It seems to accept any names - they don't need to be in the database already (in contrast to stateadmin and countyadmin roles).

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"
```

Figure A.14: Audit board dashboard.

```
batch_id "4"
contest_info [...]
0 {...}
```
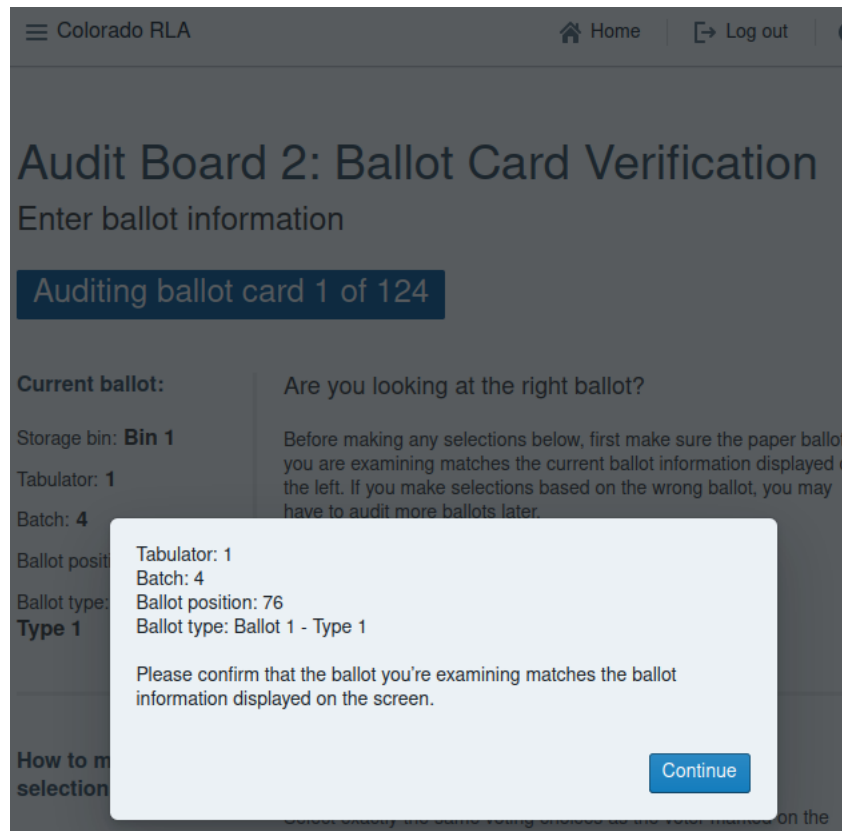
Figure A.15: Ballot card verification popup.

```
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 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.

```
timestamp "2023-09-13T12:16:22.015Z"
auditBoardIndex 1
cvr_id 14348

Response:
```

Ballot Card Verification
Review ballot

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**

Confirm that the information displayed accurately reflects its interpretation for each contest and choice from the corresponding paper ballot.

If there are any discrepancies, click the **Edit** button located to the right of the audited contest's selections, and reenter the voter markings for the ballot.

If the review page accurately reflects the audit board's interpretation of all votes in all contests, click the **Submit** button at the bottom of the page.

IRV for Test County

✎ Edit

Candidate 4(2)   Candidate 10(2)

Candidate 8(2)   Candidate 4(3)

Candidate 8(3)

Submit & Next Ballot Card

Figure A.17: Audit ballot submission summary.

```
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
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
choices [ "Candidate 1(1)", "Candidate 2(2)", "Candidate 3(3)", "Candidate 4(4)", "Candidate 5(5)",
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.

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"
```

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.

Q 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.

```
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, ...}
```

I'm confused about the sample numbers. I've audited one ballot, and the expected_count is not close to estimated_ballots_to_audit or to half of it. Looks like optimistic_ballots_to_audit - estimated_ballots_to_audit is the number that I have actually audited.

Ans: estimated and optimistic are already updated because of the discrepancy that you just entered.