Lab 2: Diverse and Redundant Software Design

LLM-Powered N-Version Programming

YOUR NAME, Purdue University, USA

This is the first lab for the module on Design and Implementation. The topic is the use of modern large language models (LLMs), or agents based thereupon, to realize the old dream of N-version programming. See IEC 61508, Part 3, Table A.2, row 3d.

This lab has three parts: pre-lab, in-class component, and take-home component. All students enrolled in the course are expected to complete the pre-lab and the in-class component. The take-home component is one of your options for the "complete 3 take-home labs" aspect of the course.

1 Lab Purpose and Context

Earlier in this module, we examined IEC 61508's treatment of redundancy and diversity in hardware and software. We saw that:

- Hardware redundancy is often justified using probabilistic independence assumptions.
- Software design diversity (IEC 61508 Part 3, Table A.2, row 3d) — typically known as N**-version programming** — is recommended more cautiously.

We also reviewed the mathematical argument behind redundancy, which hinges on independence. Let F_i denote the event that version i fails on a particular input. **Independence** means that for all events $i \neq j$, the occurrence of one failure event does not imply the occurrence of another, *i.e.*:

$$P(F_i \cap F_j) = P(F_i)P(F_j)$$

If two channels each fail with probability p, and failures are independent, then the joint failure probability is p^2. Generalizing this model, voting architectures constructed out of components with independent failure rates (*e.g.* 1-out-of-2/1oo2, 2-out-of-3/2oo3) dramatically reduce the probability of failure.

However, as discussed in class, this independence assumption is questionable in software. Knight and Leveson (1990) spearheaded the experimental evaluation of N-version programming and found substantial correlation in failures across independently-developed programs based on the same specification. Their results challenged the idea that independent software development naturally produces independent software faults.

In this lab, you will run a modern version of that experiment.

1.1 From Knight & Leveson to LLMs

Knight and Leveson's original experiment required human teams independently implementing the same specification. Such arrangements are expensive and have not been found to offer the necessary failure independence to obtain the polynomial reduction in failure when combined. However, in the modern era, large language models (LLMs) allow us to generate multiple implementations of the same specification at very low marginal cost. This raises a natural question:

Can LLMs cheaply realize the old dream of N-version programming?

More precisely, do independently generated implementations by LLMs exhibit the failure independence required for voting-based reliability improvements?

Author's Contact Information: Your Name, Purdue University, West Lafayette, Indiana, USA, your.email@purdue.edu.

1.2 Experimental Questions

This lab investigates two claims:

(1) **Independence Claim.** Do LLM-generated implementations exhibit statistically independent failures, as assumed in classical N-version programming?
(2) **Reliability Improvement Claim.** Even if failures are correlated, as Knight & Leveson found, does N-version voting improve observed reliability relative to a single implementation?

1.3 System Under Study

You will be given a frozen specification for a program.

You may not modify the specification.

Using an LLM or agent system of your choice, you will generate N distinct implementations of the same specification. Each implementation must satisfy the same functional requirements, but may be produced using different prompting strategies or generation workflows. This follows the definition of *design diversity*: two or more items carrying out the same function but differing in design

1.4 Development and Evaluation Structure

To mirror the structure of Knight & Leveson's experiment:

- You will have access to a **visible test suite** during development.
- You may iteratively refine prompts and LLM/agent configurations using feedback from this suite.
- A **reserved test set** will be held back and used only for final evaluation.

The specification remains fixed throughout. The reserved inputs will be used to measure:

- Individual failure rates,
- Joint failure rates across versions,
- Observed reliability of k-out-of-N voting.

In this lab, you will empirically estimate:

- Marginal failure probability,
- Joint failure probability,
- Conditional failure probability.

You will compare observed joint failure rates to those predicted under independence. This directly operationalizes the mathematical reasoning behind redundant computation from the lecture.

1.5 Learning Objectives

After completing this lab, you should be able to:

- Explain the mathematical basis of redundancy under independence.
- Critically evaluate the independence assumption in software design diversity.
- Design and conduct an empirical study evaluating correlated failures.
- Use LLMs for simple tasks.
- Analyze the effectiveness of N-version voting in the presence of correlation.
- Connect empirical results to IEC 61508's guidance on design diversity.

2 Pre-Lab Assignment

Before the in-class session, complete the following:

(1) **Review the Lecture Slides.**

Re-read the slides from the unit on redundancy and diversity in software engineering. In particular, review:

- The probabilistic argument for reliability improvement under independence.
- The derivation for 1oo2 voting.
- IEC 61508 Part 3, Table A.2, row 3d (Design diversity).
- The summary of the Knight & Leveson experimental evaluation.

(2) **Derive the 2oo3 Reliability Formula.**

Suppose each version fails independently with probability p on a given demand. Let $q = 1 - p$ denote the probability of success.

In lecture, we derived the system failure probability for the 1-out-of-2 (1oo2) case, where the system is correct if at least one version is correct:

$$P(\text{system failure}) = p^2.$$

Now consider a 2oo3 (two-out-of-three) voter, *i.e.* the system is correct if at least two of the three versions are correct.

- Enumerate the failure cases for the voter.
- Use independence to compute the probability of each case.
- Derive an expression for:

  $$P(\text{system failure})$$

  as a function of p and q.

Your final expression should be written in closed form (no summation required at this stage). Show your reasoning clearly.

> **Deliverable:** Give your derivation of the 2oo3 system failure probability. Your answer must:
>
> - Clearly enumerate the failure cases,
> - Use the independence assumption explicitly,
> - Present the final expression as a simplified closed-form formula.

(3) **Set Up Your LLM Environment.**

Ensure that you have access to an LLM of your choice that you will use to generate program implementations for this lab.

You may use any of the following:

- **Gemini Pro**, available to you as a student.
- **Microsoft Copilot**, to which Purdue provides access.
- An **open-weight model** running locally on your laptop (e.g., via a system such as `ollama`).

Regardless of the system you choose, confirm that you can:

- Start a fresh session (to create independent "versions").
- Provide a specification and obtain generated source code.

- Run the generated code locally against a simple test suite.

(4) **Review the Experimental Structure.**

Review the experimental plan described in the lab handout:

- A frozen specification will be provided.
- You will generate N distinct implementations.
- A visible test suite will be available during development.
- A reserved test set will be used for final evaluation.

Make sure you understand:

- What is being measured (marginal failure, joint failure, voting reliability).
- Where the independence assumption enters the reliability argument.

In-Class Lab

1 In-Class Lab: 3-Version Development and Independence Assessment

In the in-class portion of this lab, you will construct three independently generated implementations of the tax engine specified in section A. The objective is to instantiate the 2-out-of-3 (2oo3) voting architecture analyzed in the pre-lab. By the end of the session, each team will have produced three distinct implementations of the same frozen specification and evaluated them on a visible test set.

To approximate independent development:

- Each implementation must be generated in a fresh LLM session.
- Team members may discuss high-level strategy before generation.
- During generation, team members may not share code across versions.
- Each implementation should intentionally vary along at least one dimension (*e.g.* language, architecture, numeric handling, library usage).

1.1 Team Structure and Requirement

All submissions (in-class and online) must contain three LLM-generated implementations.

In-class students:

Work in groups of two students. At most one group of three may be formed if necessary.

Online students:

Online students should work with those of their Team project partners who are also online students. This will result in groups of 2-3 online students for this submission.

Team members:

- NAME 1 (email)
- NAME 2 (email)

Team signup forum: Lab 2 Team Signup Discussion (Brightspace)
Please indicate your team members in the Brightspace discussion forum link above.

1.2 Version Independence Requirements

You may use prompting strategies such as those illustrated in section B. However, the specification in section A must remain unchanged.

1.3 Plan for Diversity

Before generating code, discuss how you will induce diversity across the three implementations. You may draw on prompting strategies such as those illustrated in section B.

- How you will induce diversity across the three implementations,
- What dimensions you expect might influence failure behavior,
- Whether floating-point handling, architecture, or library choice might affect correctness.

Team Strategy: In 2–3 sentences, describe how your team will intentionally induce diversity across the three implementations.

1.4 Generate Three Implementations

Generate your three implementations.

(1) Start a new LLM session.
(2) Provide the full specification from section A.
(3) Develop a complete implementation that reads the input and produces tax outputs.
(4) Ensure the implementation runs locally.

Note 1: *I have fed the specification to ChatGPT, Microsoft CoPilot, and Google Gemini and prompted them to "Write a program to answer this spec". All of them choked, had network issues, or timed out. For the in-class part, you will need to break things down for them so you can complete the implementation in a timely manner.*

Note 2: *I advise you to complete two implementations and do a 1oo2 check before you develop the third implementation.*

A small smoke test set (Test Set 1) is provided on Brightspace. Because we do not provide a reference implementation for this lab, Test Set 1 cannot help you establish correctness (nor is that the point of this lab). Instead, it is used to reduce noise by ensuring that all implementations:

- parse inputs successfully,
- produce well-formed outputs, and
- behave consistently enough to proceed with meaningful analysis of disagreement and potential correlated failures.

Run all three implementations on Test Set 1. If any implementation crashes, produces malformed output, or disagrees with the other implementations, revise it (*e.g.* via debugging and prompting) and re-run Test Set 1. Proceed only once your team has a stable outcome on Test Set 1. You should aim for unanimous agreement across all cases, thus indicating each version is (seemingly) correct. Settling for 2oo3 agreement on every case would suffice if needed.

1.5 Independence Analysis on Test Set 2

A second, larger "full" test set (Test Set 2, approximately 10,000 inputs) is provided on Brightspace. A "runner" program is also supplied to let you drive multiple implementations and get results. Run all three implementations on Test Set 2 and record, for each input, whether the three implementations agree.

Because no external oracle is provided, we treat disagreement among implementations as evidence of potential fault events. This allows us to study the structure of divergence without assuming any version is correct.

Construct an agreement table with one row per input from Test Set 2. For each input, record the outputs produced by each version:

$$\text{Input} \longrightarrow (\text{Output}_{V1},\ \text{Output}_{V2},\ \text{Output}_{V3})$$

Using this data, compute and report:

- The fraction of inputs with unanimous agreement (3/3).
- The fraction of inputs with exactly one dissenting version (a 2–1 split).
- The fraction of inputs where multiple versions diverge from one another.

- For each version, how often it disagrees with the other two.

Discuss:

- Whether disagreement events appear isolated or clustered on specific inputs.
- Whether the same implementation tends to disagree repeatedly.
- Whether disagreement patterns suggest independence or correlated behavior.

At this stage, you are not estimating true failure probabilities. Rather, you are examining the empirical structure of disagreement as preliminary evidence about independence assumptions.

This analysis evaluates *consistency* and *correlated divergence* among independently generated implementations. It does not, by itself, establish correctness, since we have not included correct answers in the test set.

1.6 Preliminary Analysis

Within your team, discuss:

- Are disagreement events concentrated on the same inputs across implementations, or do they appear scattered?
- When disagreements occur, do they tend to implicate the same implementation repeatedly (one version often being the outlier), or do outliers rotate across versions?
- What aspects of the specification appear to trigger disagreements? For example, are disagreements concentrated on inputs that activate the floating-point rules (e.g., the EWMA-based surcharge), threshold boundaries, or phaseouts?
- Based on the observed disagreement patterns, is the independence assumption plausible, or do you see evidence of correlated behavior (common-mode mistakes)?

You are not required to perform formal statistical testing at this stage. The goal is to build intuition about correlated behavior and the independence assumption using empirical disagreement data. If you choose to complete the take-home portion of the lab, you will perform a larger version of this experiment and may incorporate more formal modeling of the underlying hypothesis of independence.

Deliverable for the In-Class Portion: Submit:

- The three implementations (distinct zipped folders labeled `impl-1.zip`, `impl-2.zip`, `impl-3.zip`),
- Your disagreement table for Test Set 2 (as a CSV with columns `input_id`, `out_v1`, `out_v2`, `out_v3`),[a]
- A brief analysis describing observed agreement or disagreement patterns, including at least one table or figure.

[a] For simplicity, you can sum the federal and state taxes for those outputs and just report results in the aggregate

Take-Home Lab

1 Take-Home Portion: Empirical Evaluation of N-Version Programming

In this take-home component, you will extend the in-class experiment in two directions:

- Develop the general k-out-of-N reliability model under the assumption of independence.
- Empirically evaluate whether LLM-generated implementations conform to that model.

1.1 Theory: Derivation of the k-out-of-N Model

Suppose each version fails independently with probability p on a given demand. Let $q = 1 - p$ denote the probability of success.

Let X be the number of successful versions out of N. Under the independence assumption, X follows a binomial distribution:

$$X \sim \text{Binomial}(N, q).$$

For a k-out-of-N voting architecture, the system is correct if at least k versions are correct. Equivalently, the system fails if fewer than k versions are correct.

Your task: Derive an expression for the system failure probability $P(\text{system failure})$ for a general k-out-of-N voter. Your final answer should be written in summation form using binomial coefficients (a closed-form simplification is *not* required). Clearly indicate where and how the independence assumption is used in your derivation.

Your derivation should clearly specify:

- The definition of the random variable X (what does it count?),
- The probability that exactly i versions are correct,
- The condition under which the k-out-of-N voter fails,
- The resulting summation expression for $P(\text{system failure})$.

1.2 Evaluation: Empirical Comparison to the Independence Model

In the in-class portion, you initiated N-version development using LLM-generated implementations, created manually. For the take-home component, you will scale this up.

1.2.1 Step 1: Generate Versions. Generate at least $N = 100$ distinct implementations of the provided specification.

You may use your choice of:

- LLM-based agents,
- Interactive LLM sessions,
- Multiple fresh sessions with varied prompts,
- Any system available to you (*e.g.* Gemini Pro, Microsoft Copilot, open-weight models).

If you have not used these types of systems before, here are some options with references to tutorials on how to get started:

(1) CoPilot Skills on VS Code[2]. Pay close attention to the examples on the drop-down for this.

(2) A programmatic agent system such as python Transformers [1]. This may allow for more control than the one above, but will require more programming
(3) An SDK such as Claude's Agent SDK[3], which strikes a balance between the two above at the cost of vendor lock-in.

Each implementation must:

- Use the same frozen specification,
- Be developed independently (fresh session),
- Be runnable against the test suite.

Hint: I picked a large enough N that you should automate this with an agentic approach, or at minimum a wrapper around a series of LLM calls.

Deliverable: Give your derivation of the $k$oo$N$ system failure probability.

1.2.2 Step 2: Measure Failure Behavior. We will release the full test suite on Brightspace.

For each input in the test suite:

- Record which versions produce incorrect output.
- Estimate:
  - Marginal failure probability p,
  - Pairwise joint failure probabilities,
  - Empirical distribution of the number of failures per input.

Deliverable: Provide your analysis, illustrated with at least one table or figure.

1.2.3 Step 3: Compare to the k-out-of-N Model. Using your derived formula from the Theory section:

- Compute the predicted system failure probability under the assumption of independent failures.
- Compute the observed system failure rate under k-out-of-N voting.
- Compare the predicted and observed values.

At this stage, you are performing a *descriptive model comparison*, not a formal statistical hypothesis test. Your goal is to assess whether the empirical behavior appears consistent with the binomial independence model, and to characterize any discrepancies you observe.

Discuss:

- Whether the observed joint behavior is qualitatively consistent with independence.
- Whether voting improves reliability relative to a single version.
- How strongly the observed behavior deviates from the binomial prediction.
- Plausible explanations for any deviation.

You are not required to compute p-values or conduct formal hypothesis tests here. If you wish to perform a rigorous statistical test of independence, see the optional subsection below.

1.2.4 Optional: Statistical Analysis in the Style of Knight & Leveson. In the required portion of this lab, you compared observed reliability to the prediction of the k-out-of-N model under independence. That comparison is descriptive: you assessed whether the empirical data appear to conform to the binomial model, but you did not perform a formal statistical hypothesis test.

Knight and Leveson (1990), by contrast, conducted formal statistical analyses to evaluate whether observed joint failure behavior was consistent with independence. Replicate their statistical framing, including:

- Testing whether observed joint failure rates significantly exceed p^2 (or the appropriate independence prediction for the $k$oo$N$ scenario),
- Performing formal hypothesis tests of independence,
- Estimating correlation coefficients between version failure indicators,
- Computing confidence intervals for marginal and joint failure probabilities.

If you choose this option, clearly state:

- Your null hypothesis (H_0),
- The statistical test used,
- Any assumptions required,
- Your interpretation of the result.

Deliverable: Provide your statistical analysis and interpretation.

This deliverable is entirely optional, but instructive. You are welcome to use ChatGPT or other tools to assist with statistical calculations, provided that you understand and clearly explain the methodology used. You may complete any part of it that you see fit.

1.3 Deliverables

Provide your empirical results and a written interpretation (1–2 pages) addressing whether the data conform to the $k$oo$N$ independence model. You may place your solutions in the colorboxes above, or in a new section here as you like.

1.4 A Note on Interpreting "No Failures Observed"

Design redundancy is most suitable when single-version reliability cannot be assured, or when residual risk remains unacceptably high even after best-effort assurance activities. If single-version reliability is already extremely high on the input distribution represented by our test suite, then an N-version experiment may observe few or no failures, and therefore may have limited ability to evaluate failure independence empirically.

To see why, suppose a single generated implementation fails on approximately 1 in 10^6 inputs (*i.e.* $p = 10^{-6}$). Our controlled test suite contains $T = 10{,}000$ inputs. The expected number of failures for a single implementation on this suite is:

$$\mathbb{E}[\text{single-version failures}] = Tp = 10{,}000 \cdot 10^{-6} = 0.01.$$

Thus, even if a given implementation is imperfect, we should expect to observe zero failures most of the time.

Under the independence model, a 1oo2 voter fails only when both versions fail on the same input, which occurs with probability p^2. The expected number of 1oo2 system failures across the test suite is therefore:

$$\mathbb{E}[\text{1oo2 system failures}] = Tp^2 = 10{,}000 \cdot (10^{-6})^2 = 10^{-8}.$$

This value is effectively zero. In such a regime, the absence of observed failures is not strong evidence that failures are independent, nor that redundancy is unnecessary; it primarily indicates that the experiment, as instantiated by the available test suite, has limited statistical power to observe rare events.

If you observe very few (or zero) failures, you should report this outcome explicitly and discuss what it implies—and does not imply—about (i) single-version reliability, (ii) failure independence, and (iii) the value of design diversity for this task.

A Specification: The Tax Code

A.1 Background

You live in the country of Avalon. Spring is here and the smell of taxes fills the air. Like the United States, citizens of Avalon pay taxes both to their federal government and to the state in which they reside.

Avalon has recently modernized its tax system. There is a new federal tax code (section A.2) and a new tax code for each state (section A.3). However, the official tax calculator is proprietary and not publicly available. You would like to implement your own tax calculator so that you do not have to compute your taxes by hand.

Your task is to implement a tax computation engine that reads an input file describing taxpayers' financial information and outputs the total tax that each one owes under the 2026 Avalon tax code (federal + state).

A.2 Avalon Federal Tax Code

Taxes are taken on the adjusted income, which is derived from the gross income and a series of deductions.

A.2.1 Gross Income. Gross income is defined as the sum of all taxable income sources before deductions or credits are applied. Gross income consists of the following components:

- **Wage Income (W2).** The total annual wage income reported by the employer. This value is provided directly in the input and is treated as fully taxable.
- **Capital Gains and Losses.** Net income resulting from the sale of financial assets. Capital gain for a transaction is defined as:

  $$\text{Gain} = \text{Sale Price} - \text{Purchase Price}.$$

  Losses occur when this quantity is negative.

Gross income is computed as:

$$\text{Gross Income} = \text{Wage Income} + \text{Net Capital Gain}.$$

Net capital gain combines the gains and losses from the reporting year.

A.2.2 Capital Gains Computation. Taxpayers may have invested in one of the three publicly traded companies in Avalon: Ananya's Wool Whimsy, Miko's Coal Collaborative, and Esteban's Timberfell. If a taxpayer sells any portion of an investment during the reporting year, the gain or loss from that sale (delta from the purchase price) is taxable.

Realized Gain or Loss. For each sale of an asset, the realized gain (or loss) is defined as:

$$\text{Gain} = q \cdot (p_s - p_b),$$

where:

- q is the quantity of shares sold,
- p_s is the sale price per share,
- p_b is the purchase price per share.

Cost Basis Matching Rule. If shares in one of the companies were purchased at different times of the year, the purchase price may have varied. To calculate gains, sales must be matched to prior purchases using the **FIFO** (first-in, first-out) rule. That is, shares sold are matched against the earliest purchased shares that have not yet been sold.

Net Capital Gain. The taxpayer's **Net Capital Gain** is the sum of realized gains and losses across all assets and all sales during the reporting year:

$$\text{Net Capital Gain} = \sum_{\text{all realized transactions}} \text{Gain}.$$

All transactions are conducted to three decimal places. Results should be rounded to the nearest Avalon dollar.

A.2.3 Adjustments and Deductions. After computing gross income, the taxpayer may be eligible for certain adjustments and deductions. These reduce income before the application of tax brackets and surcharge calculations.

Deductions are applied in the order specified below. All deductions reduce gross income to produce **Taxable Income**.

Any excess deduction beyond gross income is discarded and does not carry forward.

A taxpayer may either take the Standard Deduction or itemize deductions.

Standard Deduction. The Standard Deduction is a fixed dollar amount: $10,000. If elected, it reduces gross income directly. If the Standard Deduction is taken, no itemized deductions may be applied.

Itemized Deductions. Instead of taking the Standard Deduction, a taxpayer may elect to itemize deductions. If itemizing, the taxpayer may claim:

- Charitable Donation Deduction,
- Child Tax Deduction.

Itemized deductions reduce gross income before bracket computation.

Charitable Donation Deduction. The taxpayer may report a list of charitable donations made during the reporting year. Each donation is specified as a monetary amount in dollars.

The Charitable Donation Deduction is equal to the sum of all reported donations.

$$\text{Charitable Deduction} = \sum_{i=1}^{n} d_i$$

where d_i are the individual donation amounts.

Child Tax Deduction. A taxpayer may claim a deduction based on the number of qualifying children. The deduction percentage increases with each additional child, up to a maximum of 10 children.

The applicable percentage schedule is:

- 1% for the first child,
- 2% for the second child,
- 3% for the third child,
- ...
- up to 10% for the tenth child.

The deduction is computed as a percentage of taxable income before the child deduction is applied.

A.2.4 Taxable Income. Formal definition:

$$\text{Taxable Income} = \text{Gross Income} - \text{Allowable Deductions}.$$

The taxpayer is obligated to select whichever option (Standard or Itemized) results in the lower total tax owed.

A.2.5 Progressive Tax Brackets. National income tax is applied to the Taxable income according to the following marginal bracket structure:

Taxable Income Range	**Marginal Rate**
$0 \leq I \leq 100{,}000$	5%
$100{,}000 < I \leq 200{,}000$	10%
$200{,}000 < I \leq 300{,}000$	15%
$I > 300{,}000$	20%

Policy Note: This bracket structure is intended to incentivize citizens to make more money by maintaining relatively modest increases in marginal tax rates at higher income levels.

A.2.6 High-Income Surcharge. In addition to the national income tax computed under the progressive bracket structure, a High-Income Surcharge may apply.

This surcharge is computed independently of deductions and bracket calculations. It is applied directly to **Gross Income**.

Five-Year Income History. Each taxpayer provides reported gross income values for the previous five tax years:

$$Y_{t-1},\ Y_{t-2},\ Y_{t-3},\ Y_{t-4},\ Y_{t-5}.$$

These values are expressed in dollars and may include fractional cents.

Exponentially Weighted Moving Average (EWMA).. The five-year average income is computed using an exponentially weighted moving average with smoothing factor $\alpha = 0.6$.

The EWMA is defined recursively as:

$$E_1 = Y_{t-5}$$

$$E_k = \alpha Y_{t-(6-k)} + (1-\alpha)E_{k-1} \quad \text{for } k = 2, 3, 4, 5.$$

The final EWMA value is E_5.

Threshold for Surcharge. If

$$E_5 > 1{,}000{,}000,$$

the taxpayer is subject to the High-Income Surcharge.

Surcharge Computation. If applicable, the surcharge is computed as:

$$\text{Surcharge} = 0.02 \times \text{Gross Income}.$$

If the EWMA threshold is not exceeded, the surcharge is zero.

Total National Tax. The total national tax is:

$$\text{National Tax} = \text{Bracket Tax (on Taxable Income)} + \text{Surcharge}.$$

A.3 Avalon State Tax Code

Each taxpayer resides in exactly one state. They cannot be employed in any other state, nor leave the state at any time, thus simplifying the tax code (although perhaps not their lives).

State tax is computed independently of national tax and added to the total tax owed. The two states of Avalon are:

- California
- Texas

A.3.1 California State Tax. California computes state tax based on **Gross Income**, but treats income components differently.

State tax is defined as:

$$\text{CA Tax} = 0.04 \cdot \text{Wage Income} + 0.06 \cdot \max(0,\ \text{Net Capital Gain})$$

Notes:

- Wage income is taxed at a flat rate of 4%.
- Positive net capital gain is taxed at 6%.
- If net capital gain is negative, it contributes 0 to the capital-gains component of state tax.
- No deductions (standard, charitable, or child) apply to California state tax.
- If the federal High-Income Surcharge applies to a California resident, a 5% surcharge is also applied at the state level.

A.3.2 Texas State Tax. Texas computes state tax based on **Taxable Income** (after deductions).

Texas uses the following marginal bracket structure:

Taxable Income Range	**Marginal Rate**
$0 \le I \le 90{,}000$	3%
$90{,}000 < I \le 200{,}000$	5%
$I > 200{,}000$	7%

Tax is computed marginally.

Notes:

- Texas state tax applies to Taxable Income as defined in the national tax section.
- If their federal deductions exceed 15,000, then citizens shall apply a 1% deduction to their comptued state tax.
- In Texas, the High-Income Surcharge does not affect state tax computation.

A.3.3 Total Tax Liability. Total tax owed is:

$$\text{Total Tax} = \text{National Tax} + \text{State Tax}.$$

A.4 Numerical Precision

All taxes shall be paid in whole dollars.

A.5 Input and Output Format Specification

A.5.1 Input Format (NDJSON). The input file shall be in **NDJSON** (newline-delimited JSON) format. Each line of the file is a valid JSON object representing a single household.

Each household object contains four conceptual components:

- Household data,
- Earned income history,
- Investment data,
- Charitable giving.

Household Data.

- `taxpayer_id` (string)
- `state` (string: `"California"` or `"Texas"`)
- `w2_income` (number)
- `num_children` (integer)

Earned Income History.

- `prior_five_years_income` (array of exactly five numbers)

The array is ordered from oldest to most recent year:

$$[Y_{t-5}, Y_{t-4}, Y_{t-3}, Y_{t-2}, Y_{t-1}]$$

Investment Data. Investment activity is represented using two arrays:

- `purchases`: array of objects
- `sales`: array of objects

Each purchase object contains:

- `asset_id` (string)
- `date` (string, ISO-8601 format: `YYYY-MM-DD`)
- `quantity` (number)
- `unit_price` (number)

Each sale object contains:

- `asset_id` (string)
- `date` (string, ISO-8601 format: `YYYY-MM-DD`)
- `quantity` (number)
- `unit_price` (number)

Charitable Giving. Charitable giving is represented as:

- `charitable_donations`: array of numbers

Each element of the array represents the monetary value (in dollars and cents) of a single charitable donation made during the reporting year.

A.5.2 Output Format (NDJSON). The output file shall also be in **NDJSON** (newline-delimited JSON) format. Each line of the output corresponds one-to-one with a household object in the input file.

For every input household object, exactly one output JSON object must be produced.

Each output object contains the following fields:

- `taxpayer_id` (string),
- `federal_tax` (number),
- `state_tax` (number).

The order of output taxes due must match the order of households in the input file.

A.5.3 CLI. Each implementation should be `chmod +x`'d so the runner can invoke it, and must accept the following command-line arguments:

- `-inputFile HOUSEHOLDS.NDJSON` : specifies the path to an NDJSON input file (one JSON object per line).
- `-outputFile TAXES.NDJSON` : specifies the path where the implementation must write its NDJSON output.

The program must read from the given input file and write all results to the given output file. No other input/output behavior should be required.

B Sample Prompts

This appendix provides sample prompts you may use when interacting with an LLM. You are not required to use these prompts verbatim. However, you should keep the specification fixed and avoid importing external tax rules beyond what is stated in section A.

B.1 Prompt 1: Increasing Design Diversity

> **Prompt:** Hi ChatGPT, I'm interested in developing diverse software implementations of a spec. I am thinking about ways to prompt you that will increase the variation in the implementations. I am wondering about programming language and library selection (vs. custom implementations of components). Are there other things I should think about? Please give concrete strategies and examples, and explain how each strategy might affect the diversity of failure modes.

B.2 Prompt 2: Starting from a System Design

> **Prompt:** The full spec is available as a file in your context. Can we start with a system design? I want the design to have an NDJSON parser, an analysis engine, and an output writer. I would also like to use a database to store intermediate computations, so that the tax computation pipeline is explicit and debuggable. Please propose a modular architecture and data model first, before writing any code. Then, after I confirm, we will implement the modules one at a time.

References

[1] Mark AI. 2025. Transformers Agents Tutorial: Building AI Assistants with Tool Use. https://markaicode.com/transformers-agents-tutorial-ai-assistants-tool-use/

[2] Microsoft Corporation. 2026. Example Skills. https://code.visualstudio.com/docs/copilot/customization/agent-skills#_example-skills

[3] Claude Inc. 2026. Agent SDK: Overview. https://platform.claude.com/docs/en/agent-sdk/overview

[4] Firstname LASTNAME and Firstname LASTNAME. YYYY. TITLE OF SOURCE. https://example.com/ Accessed: YYYY-MM-DD.