Cline AI Assistant Execution Guide for Financial Calculator Project

Table of Contents

.clinerules Configuration

Create a `.clinerules` file in your project root with the following content:

Unset

```
# Financial Calculator Project Rules for Cline AI Assistant

project_context: |
  This is a financial calculator module in Python for educational purposes.
  Focus on accuracy, readability, and proper financial formula implementation.

coding_standards:
  - "Follow PEP 8 styling guidelines strictly"
  - "Use type hints for all function parameters and return values"
  - "Include comprehensive docstrings in Google format"
  - "Implement proper error handling with custom exceptions"
  - "Use NumPy for mathematical operations where possible"
  - "Validate all financial inputs (no negative principal, reasonable rates)"

testing_requirements:
  - "Write pytest tests for every function"
  - "Aim for 80%+ test coverage"
  - "Include edge case testing (zero values, extreme inputs)"
  - "Use pytest fixtures for common test data"
  - "Test mathematical properties with hypothesis library"

documentation_style:
  - "Write clear, concise docstrings with examples"
  - "Include parameter validation information"
  - "Provide usage examples in docstrings"
  - "Document AI assistance with inline comments"

financial_accuracy:
  - "Double-check all financial formulas against standard references"
  - "Handle floating-point precision issues appropriately"
  - "Use appropriate rounding for currency calculations"
  - "Validate results against known financial calculators"

ai_collaboration:
  - "Mark AI-generated code sections with comments"
  - "Explain optimization suggestions clearly"
  - "Document alternative approaches considered"
  - "Track changes and improvements in development log"

prohibited_actions:
  - "Do not use hardcoded magic numbers without explanation"
  - "Do not skip input validation"
  - "Do not implement financial formulas without verification"
  - "Do not create functions without corresponding tests"
```

Prompt Execution Order

PHASE 1: Foundation Setup (Prompts 1-3)

Execute these prompts **sequentially** - wait for each to complete before proceeding.

PHASE 2: Core Implementation (Prompts 4-9)

Execute these prompts **sequentially** - each builds on the previous module.

PHASE 3: Testing & Documentation (Prompts 10-12)

Can execute **simultaneously** after Phase 2 completion.

PHASE 4: Quality Assurance (Prompts 13-15)

Execute these prompts **sequentially** after Phase 3.

PHASE 5: Enhancements (Prompts 16-18)

Optional - can execute **simultaneously** or sequentially based on preference.

Sequential Prompts (Phase-by-Phase)

PHASE 1: Foundation Setup

Prompt 1: Project Structure Setup

Unset

```
Create a Python financial calculator project with the following
structure:

financial_calculator/
├── __init__.py
├── compound_interest.py
├── loan_calculator.py
├── investment_projections.py
├── utils/
│   ├── __init__.py
│   ├── validators.py
│   └── formatters.py
├── tests/
│   ├── __init__.py
│   ├── test_compound_interest.py
│   ├── test_loan_calculator.py
│   └── test_investment_projections.py
└── examples/
    └── usage_examples.py

Also create:
- requirements.txt with numpy, pandas, matplotlib, pytest,
pytest-cov, pydantic, hypothesis
- README.md with basic project description
- development_log.md for tracking AI collaboration
- .gitignore for Python projects

Initialize each __init__.py file appropriately and add basic
module imports.

Here you have to just create a base structure. In the succesive
prompt I will guide you the details for fuctioon creation and
others
```

Prompt 2: Utility Functions Foundation

Unset

```
Implement the utility modules for the financial calculator:

1. In utils/validators.py:
   - Create custom exception classes: InvalidFinancialInput,
NegativeValueError, InvalidRateError
   - Implement input validation functions:
     - validate_positive_amount(amount: float) -> float
     - validate_interest_rate(rate: float) -> float
     - validate_time_period(time: float) -> float
     - validate_compounding_frequency(n: int) -> int
   - Add comprehensive error messages and type hints

2. In utils/formatters.py:
   - Implement formatting functions:
     - format_currency(amount: float) -> str
     - format_percentage(rate: float) -> str
     - format_financial_result(result_dict: dict) -> str
   - Support for different currency symbols and locale formatting

Include full docstrings, type hints, and basic unit tests for
validation.
```

Prompt 3: Base Module Setup

Unset

```
Set up the base structure for the three main financial calculator
modules:

1. compound_interest.py - Create empty function stubs with full
docstrings:
   - calculate_compound_interest()
   - calculate_future_value()
   - calculate_present_value()
   - calculate_continuous_compound()
   - calculate_effective_annual_rate()

2. loan_calculator.py - Create empty function stubs with full
docstrings:
   - calculate_monthly_payment()
   - generate_amortization_schedule()
   - calculate_total_interest()
   - calculate_payoff_time_with_extra_payments()

3. investment_projections.py - Create empty function stubs with
full docstrings:
   - project_investment_growth()
   - calculate_dollar_cost_average_returns()
   - project_retirement_savings()
   - simulate_portfolio_scenarios()

Include proper imports, type hints, and placeholder
implementations that raise NotImplementedError with descriptive
messages.
```

PHASE 2: Core Implementation

Prompt 4: Compound Interest Implementation

Unset

```
Implement all functions in compound_interest.py with full
mathematical accuracy:

1. calculate_compound_interest(principal, rate, time, n=12):
   - Support annual, semi-annual, quarterly, monthly, daily
compounding
   - Return dict with future_value, interest_earned,
effective_rate
   - Use formula: A = P(1 + r/n)^(nt)

2. calculate_future_value() - wrapper for compound interest
calculation
3. calculate_present_value() - reverse calculation for present
value
4. calculate_continuous_compound() - using math.e for continuous
compounding
5. calculate_effective_annual_rate() - calculate EAR from nominal
rate

Requirements:
- Full input validation using utils.validators
- Proper error handling and custom exceptions
- NumPy for mathematical operations
- Comprehensive docstrings with examples
- Handle edge cases (zero rates, very long time periods)

Mark any AI-suggested optimizations with comments.
```

Prompt 5: Compound Interest Testing

Unset

```
Create comprehensive tests for compound_interest.py in
test_compound_interest.py:

1. Unit tests for each function with typical values
2. Edge case testing (zero rates, negative inputs, extreme
values)
3. Property-based tests using hypothesis for mathematical
properties
4. Parameterized tests with known financial calculation results
5. Performance tests for large arrays of calculations

Use pytest fixtures for common test data and ensure all functions
have 100% test coverage.
Include tests for error conditions and validation.

Also create example usage in examples/usage_examples.py showing
compound interest calculations.
```

Prompt 6: Loan Calculator Implementation

Unset

```
Implement all functions in loan_calculator.py with accurate
amortization formulas:

1. calculate_monthly_payment(principal, annual_rate, years):
   - Use standard amortization formula: M = P[r(1+r)^n]/[(1+r)^n-1]
   - Handle edge case of zero interest rate
   - Return detailed payment information

2. generate_amortization_schedule(principal, annual_rate, years, extra_payment=0):
   - Return pandas DataFrame with columns: Payment_Number, Payment_Amount, Principal_Payment, Interest_Payment, Remaining_Balance
   - Support extra payment scenarios
   - Include running totals

3. calculate_total_interest() - sum of all interest payments
4. calculate_payoff_time_with_extra_payments() - calculate early payoff scenarios

Requirements:
- Full input validation and error handling
- Use pandas for schedule generation
- Support various payment frequencies
- Include detailed financial breakdowns
- Mark AI optimizations with comments
```

Prompt 7: Loan Calculator Testing

Unset

```
Create comprehensive tests for loan_calculator.py in test_loan_calculator.py:

1. Test monthly payment calculations against known values
2. Verify amortization schedule accuracy (balances, totals)
3. Test extra payment scenarios and early payoff calculations
4. Edge case testing (zero interest, very short/long terms)
5. DataFrame structure validation for amortization schedules
```

```
Include fixtures for common loan scenarios and ensure
mathematical accuracy.
Add example loan calculations to examples/usage_examples.py.
```

Prompt 8: Investment Projections Implementation

Unset

```
Implement all functions in investment_projections.py for
portfolio analysis:

1. project_investment_growth(initial, monthly_contrib,
annual_return, years):
   - Calculate future value with regular contributions
   - Handle variable contribution schedules
   - Support different compounding frequencies

2. calculate_dollar_cost_average_returns():
   - Model periodic investment purchases
   - Handle market volatility scenarios
   - Calculate average cost basis

3. project_retirement_savings():
   - Include inflation adjustments
   - Support varying contribution rates over time
   - Calculate required monthly savings for retirement goals

4. simulate_portfolio_scenarios():
   - Monte Carlo-style scenario modeling
   - Bear/bull market simulations
   - Risk analysis with different return assumptions

Requirements:
- Use NumPy for array operations and statistical functions
- Support pandas DataFrames for time series data
- Include inflation adjustment capabilities
- Comprehensive input validation
- Mark AI-generated algorithms with comments
```

Prompt 9: Investment Projections Testing

Unset

```
Create comprehensive tests for investment_projections.py in
test_investment_projections.py:

1. Test investment growth calculations with various scenarios
2. Verify dollar-cost averaging calculations
3. Test retirement planning projections
4. Validate scenario simulation outputs
5. Performance testing for large-scale simulations

Include statistical validation of Monte Carlo simulations and
ensure mathematical properties hold.
Add investment projection examples to examples/usage_examples.py.
```

PHASE 3: Testing & Documentation (Simultaneous Execution)

Prompt 10: Overall Test Suite Integration ⚡ *Can run simultaneously with Prompt 11*

Unset

```
Create a comprehensive test suite integration:

1. Update tests/__init__.py with test discovery
2. Create pytest.ini configuration file with coverage settings
3. Add test fixtures in conftest.py for shared test data
4. Implement integration tests that use multiple modules together
5. Add performance benchmarks for all calculations
6. Set up test data generation for property-based testing

Run full test suite and ensure 80%+ coverage. Generate coverage
report and document any gaps.
Create run_tests.py script for easy test execution.
```

Prompt 11: Complete Documentation ⚡ *Can run simultaneously with Prompt 10*

Unset

```
Complete all project documentation:

1. Update README.md with:
   - Complete installation instructions
   - Quick start guide with code examples
   - API reference for all modules
   - Usage examples and common scenarios
   - Contributing guidelines

2. Ensure all docstrings are complete and follow Google format
3. Add inline comments for complex financial formulas
4. Create API documentation using docstring extraction
5. Update development_log.md with detailed AI collaboration notes

Include mathematical formula explanations and references to
financial literature.
```

Prompt 12: Examples and Usage Demonstrations ⚡ *Can run simultaneously with Prompts 10-11*

Unset

```
Create comprehensive usage examples:

1. Complete examples/usage_examples.py with:
   - Real-world financial scenarios
   - Step-by-step calculation walkthroughs
   - Comparison examples between different calculation methods
   - Error handling demonstrations

2. Create Jupyter notebook (optional) with:
   - Interactive financial calculations
   - Visualization of results
   - Educational explanations of financial concepts

3. Add command-line usage examples in README
```

```
4. Include sample data files for testing
```

PHASE 4: Quality Assurance (Sequential Execution)

Prompt 13: Code Quality Analysis

Unset

```
Perform comprehensive code quality analysis and improvements:

1. Run flake8 or black for PEP 8 compliance
2. Check all type hints are properly implemented
3. Verify error handling is comprehensive
4. Optimize performance where possible using NumPy vectorization
5. Review and improve function design for reusability
6. Ensure all magic numbers are properly documented or eliminated

Fix any code quality issues found and document improvements made.
Update development_log.md with optimization details.
```

Prompt 14: Mathematical Accuracy Verification

Unset

```
Verify mathematical accuracy of all financial calculations:

1. Cross-reference formulas with standard financial references
2. Test calculations against online financial calculators
3. Validate edge cases and boundary conditions
4. Check floating-point precision handling
5. Verify rounding behavior for currency calculations
6. Test large number handling and overflow conditions

Document any discrepancies found and corrections made.
Create accuracy_verification.md with test results against known
values.
```

Prompt 15: Final Integration Testing

Unset

```
Perform final integration and system testing:

1. Test all modules working together in realistic scenarios
2. Verify data flow between functions
3. Test error propagation and handling across modules
4. Performance testing with large datasets
5. Memory usage optimization check
6. Cross-platform compatibility verification

Run complete test suite and ensure all tests pass.
Prepare final submission checklist and verify all requirements
met.
```

PHASE 5: Enhancements (Optional)

Prompt 16: CLI Interface ⚡ *Can run simultaneously with Prompt 17*

Unset

```
Example usage:
python -m financial_calculator compound --principal 1000 --rate
0.05 --time 10
python -m financial_calculator loan --principal 200000 --rate
0.04 --years 30
```

Prompt 17: Data Visualization ⚡ *Can run simultaneously with Prompt 16*

Unset

```
Add visualization capabilities using matplotlib:

1. Create visualization.py module with plotting functions:
   - Compound interest growth curves
   - Loan amortization charts
   - Investment projection graphs
   - Portfolio scenario comparisons


2. Add visualization options to main functions
```

```
3. Support different chart types and customization
4. Include interactive plotting features
5. Export capabilities for charts (PNG, PDF)

Integrate visualization calls into examples/usage_examples.py.
```

Prompt 18: Advanced Features

Unset

```
Implement advanced features:

1. Excel export functionality using openpyxl:
   - Export amortization schedules
   - Investment projection tables
   - Formatted financial reports

2. Basic Monte Carlo simulation for investment risk analysis
3. Data import capabilities (CSV financial data)
4. Configuration file support for default parameters
5. Logging system for calculation tracking

Document all advanced features in README and provide usage
examples.
```

Simultaneous Prompts Guidelines

When to Execute Simultaneously:

✅ **Safe to run together:**

- Testing + Documentation (Prompts 10-12)
- CLI + Visualization (Prompts 16-17)
- Any prompts working on different files/modules

❌ **Must run sequentially:**

- Implementation prompts (4-9) - each builds on previous
- Quality assurance prompts (13-15) - each depends on previous fixes
- Foundation setup (1-3) - creates dependencies

Simultaneous Execution Tips:

1. **Monitor Resource Usage**: Multiple AI tasks may slow down individual responses
2. **Check Dependencies**: Ensure prompts don't modify the same files
3. **Merge Conflicts**: Be prepared to resolve any conflicts if prompts modify related areas
4. **Progress Tracking**: Keep separate development_log.md entries for parallel tasks

---

Quality Control Prompts

Error Recovery Prompt:

Unset

```
I'm getting errors in [specific module]. Please:
1. Analyze the error messages and identify root causes
2. Fix the implementation issues while maintaining functionality
3. Update tests to prevent similar issues
4. Document the fixes in development_log.md
5. Verify all tests still pass after fixes
```

Performance Optimization Prompt:

Unset

```
Review the current implementation for performance improvements:
1. Identify bottlenecks in calculations
2. Suggest NumPy vectorization opportunities
3. Optimize memory usage for large datasets
4. Profile critical functions and suggest improvements
5. Maintain mathematical accuracy while optimizing
```

Code Review Prompt:

Unset

```
Perform a comprehensive code review of the entire project:
1. Check adherence to Python best practices
2. Verify proper error handling throughout
3. Ensure consistent coding style and documentation
4. Identify opportunities for code reuse
5. Suggest architectural improvements
6. Verify all requirements are met
```

Troubleshooting Prompts

Test Failures:

Unset

```
Tests are failing in [module name]. Please:
1. Analyze failing test cases and identify issues
2. Fix the underlying implementation problems
3. Update tests if requirements have changed
4. Ensure test coverage remains above 80%
5. Verify mathematical accuracy of fixes
```

Import/Dependency Issues:

Unset

```
Having import or dependency issues. Please:
1. Check and fix all import statements
2. Verify requirements.txt has correct versions
3. Update __init__.py files for proper module exposure
4. Test import functionality across all modules
5. Document any dependency changes
```

Mathematical Accuracy Issues:

Unset

```
Financial calculations seem incorrect. Please:
1. Review all mathematical formulas against standard references
2. Check for floating-point precision issues
3. Verify edge case handling
4. Test against known financial calculator results
5. Document formula sources and validation methods
```

Final Execution Summary

Total Prompts: 18 main + 6 troubleshooting Estimated Time: 3-5 hours with AI assistance Critical Path: Prompts 1-3 → 4-9 → 13-15 Parallel Opportunities: Prompts 10-12, 16-17

Remember to maintain your development_log.md throughout the process, documenting AI suggestions, optimizations, and any issues encountered!