Comprehensive Self-Assessment Guide

Dr. Yoram Segal – Version 2.0
22-11-2025

Table of Contents

---

1. General Introduction

This guide is intended to help you carry out a comprehensive self-assessment of your code project, both from an academic perspective and from a technical perspective.

Self-assessment is a vital part of the learning and development process. It enables you to:

- Identify strengths and weaknesses
- Develop reflective thinking
- Improve the overall quality of your work

1.1 Purpose of this Guide

This guide combines two dimensions:

1. **Academic Assessment**
   A framework for self-assessing the overall quality of your work, documentation, and research.
2. **Technical Assessment**
   A detailed review of technical aspects such as code organization, multiprocessing, and modular design.

1.2 How to Use this Guide

Go through each part carefully and answer the questions according to your project.

- Mark the items you have completed
- Note items that require improvement
- Take time for reflection and critical thinking about your work

---

Part I

Principles of Academic Self-Assessment

2. Fundamental Principles

Self-assessment is an important academic process that encourages:

- Reflective thinking
- Personal responsibility

- Awareness of the learning process

In this course, you are invited to determine the grade you believe you deserve. This grade is based on your own self-evaluation of the quality of your submitted work relative to the defined criteria.

2.1 Central Principle:

The level of scrutiny in grading will be influenced by the self-assigned grade

The higher the self-assigned grade, the higher the level of scrutiny and strictness in the review. This is a principle of **"Contract-Based Grading"** (promise–expectation):

- **High self-grade (90–100):**
  A very thorough and meticulous review; "looking for elephants in the barrel"; checking every tiny detail.
- **Medium self-grade (75–89):**
  A reasonable and balanced review with clear criteria.
- **Lower self-grade (60–74):**
  A flexible, sympathetic, and tolerant review—provided the submission makes sense and is reasonable.

---

3. Recommended Steps for Performing a Self-Assessment

You are not required to fill in and submit this guide together with your assignment, but it is highly recommended.

3.1 Step 1: Understanding the Criteria and Standards

Before you can evaluate your work, you must clearly understand the criteria against which it will be assessed:

- Read the assignment submission guidelines carefully
- Identify all required components (documentation, code, tests, analysis, etc.)
- Understand the expected quality level for each criterion
- Pay attention to the distinctions between different quality levels

---

3.2 Step 2: Mapping Your Work Against the Criteria

Use the following checklist to see which components you have included in your work.

3.2.1 Project Documentation (20%) – *Project Documentation (PRD, Architecture)*

PRD (Product Requirements Document)

- Clear description of the project's purpose and user problem
- Measurable goals and success metrics (KPIs)
- Detailed functional and non-functional requirements
- Dependencies, assumptions, and constraints
- Timeline and milestones

Architecture Documentation

- Block diagrams (C4 Model, UML)
- Operational architecture
- Architectural decision records (ADRs)
- API and interface documentation

Self-grade for this category: __ /20

3.2.2 README and Code Documentation (15%) – *README & Code Documentation*

Comprehensive README

- Step-by-step installation instructions
- Detailed usage instructions
- Example runs and screenshots
- Configuration guide
- Troubleshooting section

Code Comments and Documentation

- Docstrings for every function / class / module
- Explanations of complex design decisions
- Descriptive variable and function names

Self-grade for this category: __ /15

3.2.3 Project Structure & Code Quality (15%)

Project Organization

- Clear, modular folder structure (e.g., src/, tests/, docs/, data/, results/, config/, assets/)
- Separation between code, data, and results
- Files are not excessively long (roughly under ~150 lines, where reasonable)
- Consistent naming conventions

Code Quality

- Short, focused functions (Single Responsibility)
- Avoidance of duplicate code (DRY principle)
- Consistent coding style

Self-grade for this category: __ /15

3.2.4 Configuration & Security (10%)

Configuration Management

- Separate configuration files (.env, .yaml, .json, etc.)
- No hard-coded constants in the code
- Example configuration files (e.g., .env.example)
- Documentation of configuration parameters

Information Security

- No API keys in source code
- Use of environment variables
- Up-to-date .gitignore

Self-grade for this category: __ /10

3.2.5 Testing & QA (15%)

Test Coverage

- Unit tests with at least ~70% coverage for new code
- Tests for edge cases
- Coverage reports

Error Handling

- Documented edge cases with description and handling behavior
- Comprehensive error handling
- Clear error messages
- Logs for debugging

Test Results

- Documentation of expected results
- Automated testing reports where relevant

Self-grade for this category: __ /15

3.2.6 Research & Analysis (15%)

Experiments and Parameters

- Systematic experiments with parameter variation
- Sensitivity analysis
- Experiment table with results
- Identification of critical parameters

Analysis Notebook

- Jupyter Notebook or similar analysis tool
- Methodical and in-depth analysis

- Mathematical formulas in LaTeX (where relevant)
- References to academic literature

Visual Presentation

- High-quality charts (bar charts, line charts, heatmaps, etc.)
- Clear labels and legends
- High resolution

Self-grade for this category: __ /15

3.2.7 UI/UX & Extensibility (10%)

User Interface

- Clear and intuitive interface
- Screenshots and workflow documentation
- Accessibility considerations

Extensibility

- Defined extension points / hooks
- Documentation for plugin development
- Clear interfaces

Self-grade for this category: __ /10

3.3 Step 3: In-Depth Analysis and Uniqueness

Answer the following questions to evaluate the depth and uniqueness of your work.

Technical Depth

- Did I use advanced AI agent techniques?
- Did I add mathematical or theoretical analysis?
- Did I conduct a comparative study between different approaches?

Uniqueness and Innovation

- Does the project include original ideas or an innovative approach?
- Did I develop a solution to a complex or challenging problem?
- Did I add value beyond the basic requirements?

Prompt Book

- Did I document the development process using AI?
- Did I include examples of significant prompts?
- Did I add best practices learned from experience?

Costs and Pricing

- Did I calculate token usage?
- Did I present a detailed cost table?
- Did I propose optimization strategies?

---

4. Determining the Self-Assigned Grade – Level-Based Guide

4.1 Level 1: Grade 60–69 (Basic Pass)

Description:
A reasonable submission that covers the minimum requirements.

Characteristics:

- Working code that performs the required tasks
- Basic documentation (README with installation and run instructions)
- A reasonable project structure (though not necessarily ideal)
- Basic tests or partial coverage
- Results exist but without in-depth analysis

Review Level You Will Receive:
Flexible, sympathetic, and tolerant. The reviewers will look for logic and reasonableness in the work and will not dwell on minor details.

Recommendation:
Choose this level if you invested a reasonable effort but know the work is not perfect, or if your time was limited.

4.2 Level 2: Grade 70–79 (Good)

Description:
A high-quality project with good documentation and orderly structure.

Characteristics:

- Well-structured code with comments and modularization
- Comprehensive documentation: good README, architecture documentation, basic PRD
- Proper project structure with separation of code, data, and results
- Tests with 50–70% coverage
- Result analysis with basic graphs
- Proper configuration and secured API keys

Review Level You Will Receive:
Reasonable and balanced. Reviewers will check compliance with the main criteria but will be forgiving about small mistakes.

Recommendation:
Choose this level if you met most requirements and produced an organized, quality piece of work.

4.3 Level 3: Grade 80–89 (Very Good)

Description:
An excellent project with a high academic standard.

Characteristics:

- Professional code with high modularity and clear separation of responsibilities
- Complete and detailed documentation: comprehensive PRD, architecture with C4 diagrams, high-quality README, and user manual
- Ideal project structure aligned with best practices
- Extensive tests with 70–85% coverage
- Substantial research: parameter sensitivity analysis, analysis notebook with formulas
- Impressive visual presentation of results
- High-quality user interface
- Documented costs and optimization analysis

Review Level You Will Receive:
Thorough and meticulous. Reviewers will check full compliance with the criteria and insist on a high quality level.

Recommendation:
Choose this level if you made significant effort, met all requirements, and carried out genuine research.

4.4 Level 4: Grade 90–100 (Exceptional Excellence)

Description:
MIT-level quality – work at the level of an academic or industry publication.

Characteristics:

- Production-grade code with extensibility, hooks, and plugin architecture
- Perfect and detailed documentation in all aspects: comprehensive PRD, full architecture documentation, professional README
- Full compliance with standard ISO/IEC 25010
- Extensive tests with >85% coverage, documented and handled edge cases
- Deep research: systematic sensitivity analysis, mathematical proofs, data-driven comparison
- Top-level visualizations with an interactive dashboard
- Detailed, well-documented prompt book
- Comprehensive cost analysis with optimization recommendations
- Innovation and uniqueness: original ideas, solution to a complex problem
- Contribution to the community: open-source code, reusable documentation

Review Level You Will Receive:
Extremely meticulous – “looking for elephants in the barrel”. Reviewers will check every tiny detail, search for the smallest omissions, and insist on absolute precision.

Warning:
This level is intended only for those who are completely confident that their work is at the highest level of excellence. If gaps are found, the grade may drop significantly.

Recommendation: Choose this level only if:

- You have covered all criteria without exception
- You performed a deep and rigorous self-review and everything is truly polished
- There is significant innovation and uniqueness
- You are prepared for a very strict review

---

5. Summary Self-Assessment Table

Use the following table to calculate your self-assigned grade:

Category	Weight	My Grade	Weighted Grade
Project Documentation (PRD, Architecture)	20%		
README & Code Documentation	15%		
Project Structure & Code Quality	15%		
Configuration & Security	10%		
Testing & QA	15%		
Research & Result Analysis	15%		
UI/UX & Extensibility	10%		
Total	100%		

---

6. Self-Assessment Submission Form

Please fill in the following details and submit together with the project:

- Student name(s):
- Project title:
- Submission date:
- My self-assigned grade:

6.1 Justification for the Self-Assessment (Required – 200–500 words)

In this section, explain why you chose this grade. Include:

- **Strengths:** What did you do particularly well? Which components are of high quality?
- **Weaknesses:** What is missing or could be better? (Honesty is appreciated!)
- **Effort:** How much time and effort did you invest?
- **Innovation:** Is there anything unique or special in the project?

- **Learning:** What did you learn from the project?

6.2 Desired Level of Scrutiny in the Review

Based on the self-grade I gave, I understand that the review will be:

- 60–69: Flexible, sympathetic, tolerant – basic logic and fit check
- 70–79: Reasonable and balanced – checking main criteria
- 80–89: Deep and meticulous – full review of all criteria
- 90–100: Extremely meticulous – "looking for elephants in the barrel", attention to every detail

---

6.3 Academic Integrity Declaration

I hereby declare that:

- My self-assessment is honest and genuine
- I have checked the work against all the criteria before deciding on the grade
- I understand that a higher self-grade will lead to a more thorough review
- I accept that the final grade may differ from the self-assigned grade
- The work is my/our own (of the group), and I/we are responsible for all of it

Signature: _____ Date: _____

---

7. Tips for a Successful Self-Assessment

7.1 DO:

- Be honest – an accurate self-assessment is more beneficial than an inflated grade
- Use the criteria – go systematically through each item in the guidelines
- Document the process – keep a list of what you did and what is missing
- Get feedback – ask peers to review the work before submission
- Take time for reflection – think about what you learned and where you can improve

7.2 DON'T:

- Don't inflate your grade – an overly high grade will lead to a tough review and potential disappointment
- Don't belittle your work – even if it's not perfect, it may be better than you think
- Don't skip the justification – a poor explanation makes it harder for reviewers to understand your assessment
- Don't wait until the last minute – high-quality self-assessment takes time
- Don't forget to sign – the academic integrity declaration is mandatory

---

8. Frequently Asked Questions (FAQ)

Q: What happens if the grade I give myself is different from the grade given by the reviewer?
A: That's completely normal. Self-assessment is a learning tool, not the final grade decision. The reviewer will take the self-assessment into account but will determine the final grade based on their professional judgment.

Q: Is it always better to choose a low grade so that the review will be lenient?
A: No. A low self-grade can also lead to a lower final grade even if the work is good. The choice should reflect the actual quality of the work, not a strategy.

Q: Can I appeal the grade?
A: Yes, but the basis for appeal must be substantial. If you evaluated yourself honestly, appeals should be rare.

Q: Can I update my self-grade after submission?
A: No. The self-grade is part of the submission and determines the level of review. It cannot be changed afterward.

Q: What if I find it hard to assess myself?
A: Use the checklist, ask group members or peers for help, and rely on the detailed criteria. Self-assessment is a skill that improves with practice.

Part II

Detailed Technical Review of the Code

9. Introduction to the Technical Review

This part of the guide focuses on the **technical aspects** of your code project and provides detailed checklists for:

- Organizing the project as a package
- Using multiple processes and threads
- Designing with building-block-based architecture

---

10. Check A: Organizing the Project as a Package

10.1 Background

Organizing code as a **package** is a core principle in professional software development. A well-structured package enables:

- Code reuse across multiple projects
- Clear dependency management
- Simple distribution and installation
- Built-in testing structure

10.2 Checklist: Package Structure

Review the following items and check your project:

1. **Package Definition File – setup.py or pyproject.toml**
    - Is there a package definition file (setup.py or pyproject.toml)?
    - Does it contain all required information (name, version, dependencies)?

- Are dependencies fully listed with version numbers?

2. **__init__.py File**
   - Is there an __init__.py file in the package's main folder?
   - Does it export the public interfaces of the package?
   - Are constants such as __version__ defined in this file?
3. **Organized Folder Structure**
   - Is the source code in a dedicated folder (e.g., src/ or the package name)?
   - Are tests located in a separate tests/ folder?
   - Is documentation located in a separate docs/ folder?
4. **Relative Paths**
   - Do imports use relative paths or package names rather than hard-coded file paths?
   - Does the code avoid absolute paths?
   - Are file read/writes relative to the package path, not the script's execution location?
5. **Hash Code Placeholder (if applicable)**
   - Are there locations in the code intended for hash calculation?
   - Are these locations clearly marked (comments or dedicated functions)?
   - Is the hashing implementation consistent across the project?

10.3 Examples of a Good Structure

Recommended folder structure:

```
my_project/
 src/
  my_package/
   __init__.py
   core.py
   utils.py
 tests/
  __init__.py
  test_core.py
 docs/
 setup.py
 README.md
 requirements.txt
```

10.4 Additional Notes

Write here any findings, identified issues, or required improvements.

---

11. Check B: Use of Multiple Processes and Multiple Threads

11.1 Background

Using **multiple processes (multiprocessing)** and **multiple threads (multithreading)** is essential for optimal performance in modern software.

- **Multiprocessing:** Suitable for CPU-bound operations
- **Multithreading:** Suitable for I/O-bound operations (network, disk, etc.)

11.2 Checklist: Multiprocessing

1. **Identifying Suitable Operations**
    - Have you identified CPU-intensive operations?
    - Are these operations appropriate for parallelization?
    - Have you estimated the potential benefit of parallel execution?
2. **Implementation Using multiprocessing**
    - Does the code use Python's multiprocessing module?
    - Is the number of processes set dynamically according to the number of cores?
    - Is data sharing between processes handled correctly?
3. **Resource Management**
    - Are processes properly closed when work completes?
    - Is there proper exception handling within parallel processes?
    - Have you avoided memory leaks?

11.3 Checklist: Multithreading

1. **Identifying Suitable Operations**
    - Have you identified I/O-bound operations?
    - Do these operations involve waiting (e.g., network or disk access)?
    - Have you estimated the benefit of concurrent execution?
2. **Implementation Using Threads**
    - Does the code use Python's threading module?
    - Are threads managed in an orderly manner?
    - Is synchronization between threads handled correctly (e.g., locks, semaphores)?
3. **Thread Safety**

- Have you avoided race conditions?
- Are shared variables protected with locks?
- Have you avoided deadlocks?

11.4 Additional Considerations

- Have you considered using asyncio for asynchronous operations instead of threading?
- Did you choose the correct tool (processes vs. threads) for each task?
- Did you run performance benchmarks to validate the improvement?

11.5 Additional Notes

Write here any findings, identified issues, or required improvements.

---

12. Check C: Building-Blocks-Based Design

12.1 Background

Building-blocks-based design is a modular approach to software architecture. Each **building block** is an independent unit with:

- **Input data** – information required to perform its function
- **Output data** – the results it produces
- **Setup data** – configuration and parameters for that block

---

12.2 Design Principles

Good building-block design should adhere to:

1. **Single Responsibility:**
   Each block is responsible for one well-defined task.
2. **Separation of Concerns:**
   Each block handles one aspect of the system.
3. **Reusability:**
   Building blocks can be reused in different contexts.
4. **Testability:**
   Each block can be tested independently.

12.3 Checklist: Identifying Building Blocks

1. **System Mapping**
   - Did you create a flow diagram of your system?
   - Did you identify all main building blocks?
   - Did you map the relationships and dependencies between blocks?
2. **Defining Building Blocks**

- Is each block implemented as a separate class or function?
- Does each block have a clear, descriptive name?
- Does each block have a detailed docstring?

12.4 Checklist: Input Data

Check the input data for each building block:

1. **Clear Definition**
    - Are all input parameters clearly documented?
    - Are data types specified?
    - Is the valid range for each parameter defined?
2. **Validation**
    - Is there validation for all input data?
    - Do checks correctly handle invalid input?
    - Are clear error messages returned to the user?
3. **Dependencies**
    - Are all external dependencies identified?
    - Are dependencies supplied via dependency injection?
    - Is the block independent of application-specific code?

12.5 Checklist: Output Data

1. **Clear Definition**
    - Are all outputs clearly documented?
    - Are data types specified?
    - Is the output format well-defined and consistent?
2. **Consistency**
    - Does the output match the definition in all cases?
    - Are all edge cases handled?
    - Is output deterministic for the same input (assuming no randomness)?
3. **Error Handling**
    - Do failed operations return clear error information?
    - Is error output distinguishable from normal output?
    - Are errors properly logged?

12.6 Checklist: Setup Data

1. **Configurable Parameters**
    - Have you identified all configurable parameters?
    - Does each parameter have a reasonable default value?
    - Are parameters loaded from configuration files or environment variables?
2. **Separation of Configuration**
    - Is configuration separated from code?
    - Can configuration be changed without modifying code?
    - Are there different configurations for different environments (dev, test, production)?
3. **Initialization**
    - Is the block correctly initialized before use?
    - Is there a dedicated setup or initialize function where needed?
    - Are potential initialization errors properly handled?

12.7 Example of a Good Building Block

```
class DataProcessor:
  """
  Building block for processing data

  Input Data:
  - raw_data: List[dict]
  - filter_criteria: Dict with filtering rules

  Output Data:
  - processed_data: List[dict] (processed records)

  Setup Data:
  - processing_mode: str ('fast' or 'accurate')
  - batch_size: int (default: 100)
  """

  def __init__(self, processing_mode='fast', batch_size=100):
    # Setup / configuration
    self.processing_mode = processing_mode
    self.batch_size = batch_size
```

```
def process(self, raw_data, filter_criteria):
    # Input validation
    # Processing logic
    # Return output
    pass
```

12.8 Additional Notes

Write here any findings, identified issues, or required improvements.

Part III

Summary and Recommendations

13. Overall Grade

After going through all checks—both academic and technical—calculate an overall grade for your project.

13.1 Academic Grade (from Part I)

- Weighted academic grade: __ /100

13.2 Technical Grade (from Part II)

Check how many technical checklist items passed:

- Total technical items passed: __
- Total technical items: __
- Technical success rate: __ %

13.3 Overall Grade

Suggested weighting:

- Academic grade: 60%
- Technical grade: 40%
- Final overall grade: __ /100

---

14. Areas for Improvement

Write the three main areas that require improvement in your project:

1. __
2. __
3. __

---

15. Action Plan

For each area of improvement, define concrete action steps.

---

16. Conclusion

Self-assessment is an ongoing process combining:

- Academic reflection
- Deep technical review

Return to this process regularly (e.g., monthly or at the end of each development phase) to ensure your project continues to improve and meet the highest standards — in both documentation and research, and in code quality and technical design.

Good luck!
Remember: an honest and accurate self-assessment is a sign of academic maturity and professional excellence.

Version 2.0 – 22-11-2025