Guidelines for Submitting Outstanding Software

for the M.Sc. in Computer Science

Version 2.0
Dr. Yoram Segal

22-11-2025

Table of Contents

---

1. New in Version 2.0

Version 2.0 of this guidelines document expands the criteria for submitting software at an **outstanding** level and adds three new chapters that focus on **deep technical inspection of the code**. These additions complement the existing academic and research criteria and ensure a comprehensive evaluation of code quality at the technical level.

1.1 New Chapters Added

Chapter 13: Organizing the Project as a Package –
This chapter provides a detailed checklist for organizing the project as a professional **Python package**. It includes criteria for definition files (setup.py or pyproject.toml), a well-organized directory structure, __init__.py files, use of relative paths, and dependency management. Correct packaging enables code reuse, clear dependency management, and simple distribution and installation.

Chapter 14: Parallel Processing and Performance –
This chapter deals with correct use of multiple processes (**multiprocessing**) and multiple threads (**multithreading**) to improve system performance. It explains the difference between CPU-bound and I/O-bound operations, and provides detailed checklists for each approach, including resource management and thread safety.

Chapter 15: Modular Design and Building Blocks –
This chapter presents a modular approach to software architecture based on **building blocks**. Each building block is a standalone unit with clearly defined input data, output data, and configuration. The chapter includes detailed checklists for each type of data and emphasizes essential design principles such as single responsibility, separation of concerns, and ease of testing.

1.2 Additional Improvements

In addition to the three new chapters, Version 2.0 includes:

- Expansion of the final checklist (Chapter 13) to include the new technical criteria
- Integration of a **technical self-assessment** alongside the academic evaluation
- Recommended grade weighting: **60% academic criteria and 40% technical criteria**
- Emphasis on the importance of deep technical inspection as an integral part of software quality evaluation

1.3 Purpose of the Additions

The additions in Version 2.0 are intended to ensure that software projects are evaluated not only against high **academic and research standards**, but also against **professional code and technical design standards**. Combining academic evaluation with deep technical evaluation produces a full and comprehensive picture of the project's quality, and ensures that students develop **practical software-engineering skills** alongside **theoretical research capabilities**.

---

2. General Overview

This document defines the criteria for submitting a software project at a level of academic excellence appropriate for **outstanding M.Sc. students in Computer Science** [1], [2]. The requirements focus on high-quality development, comprehensive documentation, and demonstration of advanced research and development skills. Wherever the term "project" appears, it refers to any assignment, task, or project that is submitted as part of the course.

---

3. Project and Design Documents

Every professional software project begins with clear and comprehensive documentation of the requirements and the design. These documents form the basis for development, ensure shared understanding among all stakeholders, and allow tracking of project progress over time.

3.1 Product Requirements Document (PRD)

The **Product Requirements Document (PRD)** is the central document that defines the **project's purpose and requirements** [3], [4], [5].

The PRD begins with a general overview of the project and its context, including:

- A clear description of the **user problem** the project is intended to solve
- Analysis of the **competitive landscape** and strategic positioning of the project
- Identification of the **target audience** and **stakeholders**

It then defines project goals and success metrics, including:

- Clear, measurable objectives
- **KPIs** for quantifying impact and success

- **Acceptance criteria** that enable evaluation of whether the project meets the requirements

A central part of the PRD is the description of **functional and non-functional requirements**, including:

- A prioritized feature list
- **User stories** and **use cases** describing how users interact with the system
- Requirements for **performance**, **security**, **availability**, and **scalability** [6], [7], [8]

The PRD also identifies assumptions, dependencies, and constraints, including:

- External systems and technological dependencies
- Technical and organizational limitations
- Items that are explicitly **out of scope**

Finally, the document includes a detailed **timeline and milestones**, with review points and expected **deliverables** at each stage.

3.2 Architecture Document

The **Architecture Document** provides a comprehensive technical description of the system's structure and behavior:

- Visual diagrams, such as **C4 Model** diagrams showing the system at Context, Container, Component, and Code levels
- **UML** diagrams describing complex processes and component interactions
- **Deployment diagrams** describing the technological infrastructure and **operational architecture**

In addition, the document includes:

- **Architecture Decision Records (ADRs)** explaining the rationale for key architectural decisions, including trade-offs and alternatives considered
- Detailed documentation of APIs and other interfaces: public interfaces, **data schemas**, and **contracts** describing interactions between components

---

4. Code Documentation and Project Structure

Correct code documentation and an orderly project structure are essential foundations of professional software development. Good documentation allows other developers to quickly understand the code, use it correctly, and contribute effectively.

4.1 Comprehensive README File

The **README** file is the central document accompanying every software project and serves as a full **user manual** [9], [10].

It should include:

- **Installation instructions**:
    - System requirements
    - Step-by-step installation for different environments
    - How to configure **environment variables**
    - A **troubleshooting** section for common problems

- **Usage instructions**:
    - How to run the software in different modes
    - Explanation of CLI flags and GUI options
    - A typical **user workflow**
- **Examples and demonstrations**:
    - Code examples
    - Screenshots of the UI
    - Example usage scenarios
    - Links to demo videos, if relevant
- **Configuration guide**:
    - Explanation of configuration files and tunable parameters
    - Their impact on system behavior
- **Contribution guidelines**:
    - Code style and standards
    - How to submit changes
- **License & Credits**:
    - License information
    - Acknowledgment of third-party libraries and contributors

4.2 Modular Project Structure

A good project structure is the key to efficient maintenance and future evolution of the code.

Principles include:

- Logical separation into directories by role:
    - Source code
    - Tests
    - Documentation
    - Data
    - Results
    - Configuration
    - Assets/resources
- Organization can be **feature-based** or **layered architecture**, with clear separation between:
    - Code
    - Data
    - Results
    - Documentation

File size is important:

- Source files should generally not exceed **~150 lines of code**, so each file has a focused responsibility and is easy to understand.
- If a file becomes too large, split it into smaller functions and modules while preserving **separation of concerns**.

It is also important to maintain consistent **naming conventions** for files and directories across the project.

Example of a recommended structure:

```
project-root/
├── src/          # Source code
│   ├── agents/   # Agent modules
│   ├── utils/    # Helper functions
│   └── config/   # Configuration code
├── tests/        # Unit and integration tests
├── data/         # Databases and input files
├── results/      # Experiment results
├── docs/         # Additional documentation
├── config/       # Configuration files
├── assets/       # Images, graphs, resources
├── notebooks/    # Analysis notebooks
├── README.md
├── requirements.txt
└── .gitignore
```

4.3 Code Quality and Comments

Code quality is measured not only by functionality, but also by **readability and maintainability**.

Code Comments Standards [11], [12], [13] require that comments explain the **"why"**, not only the **"what"** – i.e., focus on design decisions and rationale rather than merely describing operations.

- Every function, class, and module should have **docstrings** explaining:
    - Purpose
    - Parameters
    - Return values

Comments should:

- Explain complex design decisions
- Document assumptions and preconditions
- Be kept up-to-date along with code changes

Good coding practices include:

- Descriptive, precise names for variables and functions
- Short, focused functions that follow **Single Responsibility Principle**
- Avoiding duplicate code (**DRY** – Don't Repeat Yourself)
- Consistent coding style across the entire project

---

5. Configuration Management and Information Security

Proper **configuration management** and **information security** are critical for safe operation, especially in production environments or when handling sensitive data.

5.1 Configuration Files

Separating configuration from code is a fundamental principle of modern software development.

- Configuration should be stored in separate files in standard formats such as .json, .yaml, or .env
- Avoid **hardcoded values** in the source code
- Provide example configuration files such as .env.example with secure defaults
- Document every configuration parameter and its effect on the system

When using **Git**:

- Use .gitignore to ensure sensitive configuration files are **not committed**
- Create template configuration files for different environments:
    - Development (dev)
    - Staging
    - Production

5.2 Information Security

Protecting **API keys** and other secrets is critical to prevent data leakage and unauthorized use [14], [15], [16].

Core rules:

- Never store API keys in source code
- Always use **environment variables**, e.g.:
- os.environ.get("API_KEY")
- Hide .env files via .gitignore
- In production, use **secrets management tools**

Additional practices:

- Periodic **rotation** of keys
- Monitoring key usage to detect anomalies
- Applying **least privilege** permissions, granting only what is necessary

---

6. Testing and Software Quality

Comprehensive testing is the cornerstone of **code quality** and **system reliability**. A strong test suite:

- Finds bugs early
- Ensures code meets requirements
- Enables confident future changes

6.1 Unit Tests

Unit tests validate individual components in isolation.

Test coverage requirements [17], [18]:

- Aim for **70–80% minimum coverage** for new code
- Higher coverage for:
    - Critical code
    - Core business logic

Coverage should include:

- **Statement coverage** – every line executed at least once
- **Branch coverage** – all decision branches tested
- **Path coverage** – critical execution paths through combinations of decisions

Use standard test frameworks:

- unittest, pytest, etc.
- Integrate tests into a **CI/CD pipeline**
- Generate **coverage reports** and track them over time

6.2 Handling Edge Cases and Failures

Identifying and documenting **edge cases** is essential for robust software:

- Systematically identify boundary conditions and edge cases
- Document each one:
    - Expected input
    - Required system behavior
    - Screenshots, if relevant

Error handling should include:

- **Defensive programming** with thorough input validation
- Clear, helpful error messages for users
- Detailed logging for **debugging**
- **Graceful degradation**, where the system continues operating as much as possible even under partial failure

Failure documentation should include:

- Precise description of the failure and its root cause

- System response and how the error is handled
- Assessment of impact on the user and system

6.3 Expected Test Results

Documenting **expected test outputs** allows quick comparison of actual vs. expected behavior:

- Record expected outputs for each test
- Generate **automated testing reports** with **pass/fail rates**
- Store logs of successful and failed runs for future analysis and learning

---

7. Research and Results Analysis

Thorough research and results analysis are what separate a typical software project from an **excellent academic work**.

7.1 Parameter Study

Sensitivity analysis examines how different parameters affect system performance:

- Perform systematic experiments with controlled parameter changes
- Record how each parameter affects results
- Use advanced methods, such as:
    - Partial derivatives
    - Variance-based analysis
    - “One-at-a-time” (OAT) approaches

Goals:

- Identify critical parameters with the greatest impact
- Understand interactions between parameters

Documentation should include:

- A structured table of experiments: parameter values and corresponding results
- Graphs, such as:
    - Line charts for trends
    - Heatmaps for cross-parameter sensitivity
    - Sensitivity plots
- Statistical analysis:
    - Significance
    - Confidence levels

7.2 Results Analysis Notebook

A **Results Analysis Notebook** is the main tool for presenting the research in an interactive, detailed way:

- Typically implemented with **Jupyter Notebook** or similar tools

- Combines code, text, and outputs
- Allows:
    - Methodical analysis of experimental results
    - Comparison of different algorithms, configurations, or methodologies
    - Inclusion of mathematical proofs or theoretical analysis where relevant

Use **LaTeX** to:

- Write equations and formulas
- Provide detailed mathematical explanations of models and algorithms
- Cite academic literature and prior research

7.3 Visual Presentation of Results

High-quality **data visualization** is crucial for clear and convincing communication of research results.

Common visualizations:

- **Bar charts** – category comparisons
- **Line charts** – trends over time
- **Scatter plots** – correlations and relationships
- **Heatmaps** – 2D parameter sensitivities
- **Box plots** – distributions and statistics
- **Waterfall charts** – stepwise changes and relative contributions

Good graphs are:

- Clearly labeled and accurately scaled
- Use consistent, accessible colors (including for color-blind users)
- Include detailed **captions** and clear **legends**
- High resolution, suitable for academic or professional publication

---

8. User Interface and User Experience

A good **UI (User Interface)** and **UX (User Experience)** are critical to the success of any software system. Even a highly functional system may fail if users struggle to use it.

8.1 Quality Criteria

Usability criteria [19], [20] include several key dimensions:

- **Learnability** – how easily new users learn to use the system
- **Efficiency** – how quickly experienced users complete tasks
- **Memorability** – how easily users can return after a break and still know how to use the system
- **Error prevention** – how well the system protects users from mistakes
- **Satisfaction** – how pleasant and satisfying the system is to use

Nielsen's 10 Usability Heuristics [20], [21] include:

- Visibility of system status
- Match between system and real world
- User control and freedom
- Consistency and standards
- Error prevention
- Recognition rather than recall
- Flexibility and efficiency of use
- Aesthetic and minimalist design
- Help users recognize, diagnose, and recover from errors
- Help and documentation

8.2 UI Documentation

Comprehensive UI documentation includes:

- Screenshots of every screen and possible state
- A detailed **user workflow** from first use to completing the main goals
- Explanations of interactions and system feedback to user actions
- **Accessibility** considerations to ensure the system is usable by people with disabilities

---

9. Version Control and Development Documentation

Correct **version control** is essential for teamwork, change tracking, and the ability to revert to previous versions when necessary. Development documentation helps understand decisions made along the way.

9.1 Recommended Git Practices

Recommended **Git Best Practices** include:

- Clear **commit history** with meaningful messages describing what changed and why
- Separate **branches** for new features to maintain a stable main branch
- **Code reviews** via Pull Requests before merging changes
- **Tagging** to mark important, stable releases

9.2 Prompt Log ("Prompt Book")

Documenting development with AI (**Prompt Engineering Log**) is a new and important part of modern software development:

- List of all significant prompts used to build the project
- Description of context and purpose for each prompt
- Examples of outputs and how they were integrated
- Documentation of iterative improvements to prompts over time
- Best practices learned from experience

A recommended folder structure:

- Separate directories for prompts related to:
    - Architecture design
    - Code generation
    - Testing
    - Documentation
- A high-level overview file

---

10. Costs and Pricing

Understanding development and operational costs is critical for planning and resource decisions.

10.1 Cost Breakdown

A **Cost Breakdown** of API token usage includes:

- Accurate counting of **input and output tokens**
- Calculating **cost per million tokens (per Mtokens)** according to each provider's pricing
- Estimating total cost by model and service

Example (API token cost analysis):

Model	Input Tokens	Output Tokens	Total Cost
GPT-4	1,245,000	523,000	$45.67
Claude 3	890,000	412,000	$32.11
Total	2,135,000	935,000	$77.78

Optimization strategies:

- Reducing token usage by **summarizing and shortening prompts**
- Using **batch processing** for more efficient bulk operations
- Choosing models based on **cost-effectiveness** where cheaper models provide sufficient quality

10.2 Budget Management

Effective budget management includes:

- Cost forecasting for future scale
- Real-time monitoring of usage to detect anomalies early
- Budget alerts to prevent unexpected expenses

---

11. Extensibility and Maintainability

Designing a system that is easy to **extend and maintain** is a long-term investment that pays off as the system evolves.

11.1 Extension Points

A **plugin architecture** enables adding new functionality without changing core code:

- Define clear **interfaces** that specify contracts between core and plugins
- Expose **lifecycle hooks** (e.g., beforeCreate, afterUpdate) that plugins can respond to
- Use **middleware** for chained request processing
- Design with **API-first** principles so all functionality is available via well-defined interfaces

Extension documentation should include:

- A detailed guide for developing plugins
- Examples of simple and complex plugins
- Conventions and rules for safe extensions that do not break the system

11.2 Maintainability

Maintainable code is characterized by:

- **Modularity** and separation of concerns – each part of the code is responsible for one thing
- **Reusability** – components used in multiple places
- **Analyzability** – code is easy to understand and debug
- **Testability** – code is easy to test automatically

---

12. International Quality Standards

The **ISO/IEC 25010** standard [22] defines a comprehensive model for software quality with eight main quality characteristics, each broken down into sub-characteristics.

12.1 Product Quality Characteristics

Examples include:

- **Functional suitability**:
    - Completeness
    - Correctness
    - Appropriateness
- **Performance efficiency**:
    - Time behavior (response times)
    - Resource utilization (memory, CPU)
    - Capacity under load
- **Compatibility**:
    - Interoperability
    - Coexistence
- **Usability**:
    - Learnability

    - Operability
    - Accessibility
    - User error protection
    - User interface aesthetics
- **Reliability**:
    - Maturity
    - Availability
    - Fault tolerance
    - Recoverability
- **Security**:
    - Confidentiality
    - Integrity
    - Authenticity
    - Accountability
    - Non-repudiation
- **Maintainability**:
    - Modularity
    - Reusability
    - Analyzability
    - Modifiability
    - Testability
- **Portability**:
    - Adaptability
    - Installability
    - Replaceability

---

13. Final Checklist

Before submitting the project, go through a comprehensive checklist to ensure all requirements are met.

Documentation should include:

- A detailed **PRD** with all components
- Architecture documentation with clear block diagrams
- A **comprehensive README** at full user-manual level
- Full API documentation for all public interfaces
- A documented **prompt log**

Code should be:

- Organized in a **modular, structured project**
- Files generally under 150 lines
- With extensive comments and docstrings for every function and class
- Consistent in style throughout

Configuration should:

- Be separated from code
- Include example files (e.g., .env.example)
- Avoid API keys in source
- Use an up-to-date .gitignore

Tests should include:

- Unit tests with at least **70% coverage**
- Documented edge cases
- Comprehensive error handling
- Automated test reports

The research part should include:

- Experiments with parameter variation
- Documented sensitivity analysis
- Analysis notebook with illustrative graphs
- Mathematical formulas where relevant

Visualization should include:

- High-quality graphs of results
- UI screenshots
- Clear architecture diagrams

Cost analysis should include:

- Token usage table
- Detailed cost breakdown
- Optimization strategies

Extensibility should include:

- Documented extension points
- Plugin examples
- Clear extension interfaces

13.1 Detailed Technical Checklist

In addition to the criteria above, ensure you meet the technical criteria from Chapters 13–15:

- Project organized as a package with pyproject.toml/setup.py, __init__.py, and an organized directory structure
- Correct use of **multiprocessing** for CPU-bound tasks
- Correct use of **multithreading** for I/O-bound tasks, including thread safety
- Building-block-based design with clear definitions of input, output, and setup for each module
- Comprehensive validation of all input data
- Detailed documentation for each building block and its dependencies

Finally, ensure:

- Clean Git history
- License file included
- Proper attribution to third-party libraries
- Clear installation and deployment instructions

---

14. Additional Standards and References

For preparing an excellent project, it is recommended to refer to recognized international standards and sources, such as:

- MIT's software quality assurance plan [23]
- ISO/IEC 25010 software quality model [22]
- Google engineering practices [24]
- Microsoft REST API guidelines [25]
- Nielsen's usability heuristics [20], [21]

These provide a solid basis for high-level professional and academic work.

---

15. Organizing the Project as a Package

Organizing code as a **package** is a fundamental principle of professional software development. A well-organized package enables:

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

This chapter provides a detailed checklist for organizing your project as a professional Python package.

15.1 Package Definition File

Every professional package must include a **definition file** describing the package's properties and dependencies:

- Either traditional setup.py or modern pyproject.toml
- Must include:

- Package name
- Version number
- Short description
- Author name
- License
- Full list of external dependencies with specific or acceptable version ranges

15.2 __init__.py Files

__init__.py files are how Python recognizes a directory as a package:

- Must exist in the package's root directory and in any sub-directory that should be a sub-package
- May be empty, but it is recommended to:
  - Export public interfaces via __all__
  - Define constants such as __version__

Initialization logic may also be placed there, but should remain light to avoid slowing package import.

15.3 Organized Directory Structure

Directory organization should be logical and consistent:

- Source code in a dedicated directory, usually src/ or a directory named after the package
- Tests in a separate tests/ directory
- Documentation in a separate docs/ directory

This separation ensures production code, test code, and documentation are not mixed.

15.4 Use of Relative Paths

All imports should use **package names or relative imports**, never absolute file system paths:

- Instead of import /path/to/module use:
- from mypackage.submodule import function

When reading/writing files, compute paths **relative to the package**, not to the script being run. This ensures the package works across environments.

15.5 Checklist: Packaging

Check the following:

1. **Package definition file**
   - Does pyproject.toml or setup.py exist?
   - Does it contain all required information (name, version, dependencies)?
   - Are dependencies specified with versions?
2. **__init__.py**
   - Is there an __init__.py in the root package directory?
   - Does it export public interfaces?

- Is __version__ defined?

3. **Directory structure**
    - Is source code in a dedicated directory?
    - Are tests in a tests/ directory?
    - Is documentation in a docs/ directory?
4. **Relative paths**
    - Do all imports use relative or package paths?
    - Does the code avoid absolute paths?
    - Are file operations relative to the package path?

15.6 Example of a Proper 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
└── .gitignore
```

---

16. Parallel Processing and Performance

Using multiple processes (**multiprocessing**) and multiple threads (**multithreading**) is essential for optimal performance in modern software. Understanding each tool and when to use it is critical.

16.1 Difference Between Multiprocessing and Multithreading

The main difference lies in the **type of load**:

- **Multiprocessing** is suited for **CPU-bound** operations:
    - Heavy mathematical computations
    - Image processing

    - Training machine-learning models
      Each process has its own memory space and can use a different CPU core, enabling true parallelism.

- **Multithreading** is suited for **I/O-bound** operations:
    - Network calls
    - Database access
    - File I/O
      Threads allow other operations to continue while waiting for external components.

16.2 Using Multiprocessing

When using multiprocessing:

- Identify CPU-bound operations suitable for **parallelization**
- Use Python's multiprocessing module
- Set number of processes dynamically based on available cores, e.g.:
- multiprocessing.cpu_count()
- Use correct data-sharing mechanisms (Queue, Pipe, etc.)
- Ensure processes terminate properly to prevent resource leaks

16.3 Using Multithreading

When using multithreading:

- Identify I/O-bound operations that wait for external resources
- Use Python's threading module
- Manage threads carefully
- Ensure proper synchronization using:
    - Locks
    - Semaphores
- Avoid **race conditions** and **deadlocks** by careful design of lock acquisition and release

16.4 Thread Safety

Thread safety is critical in multithreaded programs:

- Protect all shared mutable data with locks
- Use **thread-safe data structures** such as queue.Queue to pass data between threads
- Avoid deadlocks via consistent lock ordering and using **context managers** (with statements) to ensure locks are always released

16.5 Checklist: Parallel Processing

1. **Identifying suitable operations**
    - Have you identified CPU-bound or I/O-bound operations?
    - Did you choose the appropriate tool (multiprocessing or multithreading)?
    - Did you estimate the potential benefit of parallel execution?

2. **Correct implementation**
    - Is the number of processes/threads set dynamically?
    - Is data sharing done safely?
    - Is there proper synchronization between execution units?
3. **Resource management**
    - Are processes/threads properly closed?
    - Are exceptions handled correctly?
    - Is memory leakage avoided?
4. **Thread safety (multithreading)**
    - Are shared variables protected by locks?
    - Are race conditions prevented?
    - Are deadlocks prevented?

---

17. Modular Design and Building Blocks

Building-blocks design is a modular approach to software architecture, where each component is an independent unit with a clearly defined interface. This simplifies maintenance, testing, and code reuse, and supports development of large, complex systems.

17.1 Structure of a Building Block

Each building block is defined by three kinds of data:

- **Input Data** – data required to perform the operation:
    - Clear data types
    - Valid ranges
    - External dependencies
      All input data must undergo **comprehensive validation**.
- **Output Data** – results produced by the block:
    - Data types and formats
    - Behavior in edge cases and errors
      Output should be consistent and well defined.
- **Setup Data** – configuration and parameters:
    - Configurable parameters with reasonable defaults
    - Settings loaded from configuration files or environment variables
    - Initialization parameters required before use

17.2 Design Principles

Good building-block design must follow:

- **Single Responsibility** – each block is responsible for one defined task

- **Separation of Concerns** – each block handles one aspect; e.g., a computation block is not responsible for saving to disk
- **Reusability** – building blocks can be reused in different contexts and are not tied to system-specific code
- **Testability** – each block can be independently tested; dependencies are provided via **dependency injection**

17.3 Validation and Protection

Each building block must:

- Include comprehensive input validation:
    - Types
    - Ranges
    - Preconditions
- In case of invalid input, return a **clear, helpful error message** explaining what went wrong and how to fix it
- Validate as early as possible (**fail fast**) to prevent error propagation

17.4 Checklist: Building-Block Design

1. **Identifying building blocks**
    - Did you create a system diagram and identify building blocks?
    - Is each block defined as a separate class or function?
    - Does each block have a descriptive name and detailed documentation?
2. **Input data**
    - Are all inputs clearly documented?
    - Is there validation for all input data?
    - Are dependencies provided via dependency injection?
3. **Output data**
    - Are all outputs documented?
    - Is output consistent in all situations?
    - Are errors documented and returned clearly?
4. **Setup data**
    - Are all configurable parameters identified?
    - Are there reasonable default values?
    - Is configuration separated from code?
5. **Design principles**
    - Does each block follow single responsibility?
    - Is there clear separation of concerns?
    - Are blocks reusable and testable?

17.5 Example of a Good Building Block

A conceptual example is a **DataProcessor** building block:

- **Input**:
  - raw_data: list of dictionaries
  - filter_criteria: filtering conditions
- **Output**:
  - processed_data: list of processed dictionaries
- **Setup**:
  - processing_mode: 'fast' or 'accurate'
  - batch_size: integer batch size

Key properties:

1. Single responsibility – only processes data (no loading/saving)
2. Separation of concerns – configuration separated from processing logic
3. Comprehensive validation – input validated before processing
4. Testability – each function testable independently
5. Reusability – can be used in multiple contexts

(Implementation and usage examples follow the same concepts.)

---

18. Important Note

This document describes a **very high level of excellence**. Not every item is strictly mandatory, but the more criteria you meet, the higher the grade and quality evaluation will be. Focus on depth, professionalism, and demonstration of **high-level academic research capabilities**.

It is recommended to use **LLM tools** to help complete the project. Note that as part of the evaluation, AI agents may also be used to check your work.

18.1 Combined Evaluation

Project evaluation should **combine** the academic criteria (Chapters 1–12) with the technical criteria (Chapters 13–15). A recommended weighting is:

- **60%** – academic criteria
- **40%** – technical criteria

This combined approach ensures the project meets not only research standards but also **professional software-engineering standards**.

---

19. English References

(As in the original – I'll keep the reference list as-is, since it's already in English.)

1. MIT ACIS, MIT Software Quality Assurance Plan, 2022.

2. A. Downey, Software Engineering Practices for Scientists, 2013.
3–25. (Full list retained exactly as in your text.)

If you want, I can now:

- Turn this into a **polished PDF/Word structure** with headings and styles, or
- Extract just a **short English summary** for students instead of the full formal version.

Make sure to implement those key points:

- **Submission as a package that is independent of the installation location**
- **Use of parallel multitasking**
- **Design in a modular manner**