Here's the **English translation** of the uploaded document **"Guidelines for Submitting Excellent Software for M.Sc. in Computer Science"** by **Dr. Yoram Segal** (© 2025).

Guidelines for Submitting Excellent Software for M.Sc. in Computer Science

1. Overview

This document defines the academic and technical requirements for submitting a high-quality software project for an M.Sc. in Computer Science. It provides detailed expectations for documentation, code structure, testing, research presentation, and software quality.

---

2. Project and Design Documents

2.1 Product Requirements Document (PRD)

Defines the project's purpose, scope, and objectives, including:

- Stakeholders, KPIs, and success criteria.
- Acceptance tests and measurable outcomes.
- Functional and non-functional requirements, user stories, and use cases.
- Scalability, performance, limitations, and dependencies.
- Timeline, milestones, and deliverables.

2.2 Architecture Document

Describes system architecture, diagrams, and design decisions:

- Use of the **C4 model** (Context, Container, Component, Code).
- UML diagrams and deployment architecture.
- Architectural Decision Records (ADRs) summarizing trade-offs and alternatives.
- API definitions, data schemas, and interface contracts.

---

3. Code Documentation and Project Structure

3.1 Comprehensive README

The README serves as the main manual. It must include:

- Installation instructions and environment setup.
- Usage examples and troubleshooting.
- Configuration details and contribution guidelines.
- Licensing, authorship, and references.

3.2 Modular Project Structure

The project must have a clear modular organization with logical separation of concerns (e.g., feature-based or layered architecture). Example structure:

project-root/

├— src/ (source code)

```
│   ├── agents/
│   ├── utils/
│   └── config/
├── tests/
├── data/
├── results/
├── docs/
├── assets/
├── notebooks/
├── README.md
├── requirements.txt
└── .gitignore
```

3.3 Code Quality and Comments

All functions, classes, and modules must include **docstrings** and inline comments.
Follow DRY (Don't Repeat Yourself) and single-responsibility principles.
Code should be self-explanatory, well-structured, and consistent in naming.

---

4. Configuration Management and Security

4.1 Configuration Files

Store configuration separately (e.g., .json, .yaml, .env).
Never hard-code secrets. Include an example configuration (env.example).
Use .gitignore to exclude sensitive files.

4.2 Information Security

Protect API keys and credentials with environment variables or secret-management tools.
Apply least-privilege access and rotate secrets periodically.

---

5. Software Testing and Quality

5.1 Unit Tests

Unit tests must cover **at least 70–80 %** of the codebase.
Use frameworks such as pytest or unittest.
Automate test execution in CI/CD pipelines and include coverage reports.

5.2 Handling Edge Cases and Errors

Test critical and edge cases.
Ensure graceful degradation, defensive programming, and informative error handling.

5.3 Expected Test Outputs

Provide clear test results, success/failure rates, and error logs.

---

6. Research and Results Analysis

6.1 Parameter Sensitivity Analysis

Analyze how parameter changes affect outcomes using methods like OAT (one-at-a-time) or variance-based sensitivity analysis.
Present findings with visualizations (heatmaps, line charts, etc.).

6.2 Analysis Notebooks

Include **Jupyter Notebooks** or equivalent showing code, metrics, and visualizations.
Document methodology, results, and interpretation clearly.

6.3 Visual Presentation of Results

Provide well-designed visual outputs — bar, line, scatter, and waterfall charts — with captions and explanations for every figure.

---

7. User Interface (UI) and User Experience (UX)

7.1 Usability Criteria

Evaluate software usability using **Nielsen's 10 Heuristics**, including:

- Visibility of system status
- Match between system and real world
- User control and freedom
- Consistency, error prevention, and minimalist design
- Help, documentation, and recovery from errors

7.2 Interface Documentation

Include screenshots or diagrams illustrating workflows, accessibility features, and interaction flow.

---

8. Version Control and Development Documentation

8.1 Git Best Practices

Use clear commit messages, meaningful branches, pull requests, and tagging.
Keep a readable commit history.

8.2 Prompt Engineering Log

Document AI prompts used in code generation or development.
For each prompt, record its purpose, context, and results.

---

9. Costs and Budget

9.1 Cost Analysis

Estimate computational or API token usage costs.
Example cost table (for GPT-4, Claude, etc.) should include input/output tokens and total cost.

9.2 Budget Management

Track costs over time, monitor API usage, and optimize cost-effectiveness.

---

10. Extensibility and Maintainability

10.1 Extension Points

Design the architecture for extensibility using plugins or middleware.
Define clear interfaces and lifecycle hooks (e.g., beforeCreate, afterUpdate).

10.2 Maintainability

Ensure modular, testable, and reusable code.
Support easy updates, analysis, and replacement of modules.

---

11. International Quality Standards

11.1 Software Quality Attributes (ISO 25010)

The project should comply with ISO/IEC 25010 standards, covering:

- Functional suitability
- Performance efficiency
- Compatibility and interoperability
- Usability and reliability
- Security and maintainability
- Portability and adaptability

---

12. Final Submission Checklist

Before submission:

- Include PRD, architecture, and README documents.
- Provide source code with modules, tests, and docstrings.
- Attach configuration examples and .gitignore.
- Verify test coverage ≥ 70 %, run edge-case tests, and include visual results.
- Document cost, research findings, and extension design.

---

13. Additional References and Standards

Includes references to **MIT ACIS**, **ISO/IEC 25010**, **Google Engineering Practices**, **Microsoft REST API Guidelines**, and **Nielsen UX Heuristics**.

---

14. Important Note

This document summarizes best academic and engineering practices for M.Sc. software submissions.
Projects are evaluated for depth, rigor, and adherence to modern software-engineering standards, including AI-assisted development practices.

---

15. English References

A full list of 25 sources from MIT, ISO, Google, Microsoft, Monday.com, Aha!, Codacy, Stack Overflow, and others is provided for best-practice frameworks and templates.

Guidelines for Submitting Excellent Software for the M.Sc. in Computer Science

Table of Contents

---

1. Executive Summary

This document defines the criteria for producing a high-quality academic software project.
The expectations are based on leading industry standards and academic best practices [1], [2].
The goal is to ensure that the project is research-oriented, well-documented, maintainable,
and includes a robust software-engineering foundation.

---

2. Project Documentation

2.1 Product Requirements Document (PRD)

The PRD is the central document of the project. It defines:

- The problem the project solves
- Business and user needs
- Stakeholders
- Success metrics (KPIs)
- Functional and non-functional requirements
- User stories and use cases
- Constraints, dependencies, scalability aspects
- Out-of-scope items
- Milestones and timeline
- Deliverables

2.2 Architecture Document

The architecture document should include:

- Overall system design
- C4 Model diagrams (Context, Container, Component, Code)
- UML diagrams
- Technology stack
- Deployment architecture
- API design and schemas
- Architectural decision records (ADR)
- Trade-offs and alternatives considered

---

3. Project Code Documentation

3.1 README File

The README is the face of the project and must include:

- Installation instructions
- System requirements
- Usage examples (CLI/GUI)

- Configuration guide
- Troubleshooting section
- Contribution guidelines
- Credits and license

3.2 Modular Project Structure

A clean folder structure such as:

```
project-root/
├── src/
│   ├── agents/
│   ├── utils/
│   └── config/
├── tests/
├── data/
├── results/
├── docs/
├── assets/
├── notebooks/
├── README.md
├── requirements.txt
└── .gitignore
```

3.3 Code Comments and Documentation

- Follow code comment standards [11–13]
- Use meaningful naming conventions
- Add docstrings for every module, class, and function
- Explain logic for non-trivial code
- Keep comments updated and relevant

---

4. Configuration and Environment Management

4.1 Configuration Files

Use separate configuration files such as:

- .env
- .json
- .yaml

Avoid hard-coded values. Use example files like example.env.

4.2 Environment Variables

- Always store API keys using environment variables
- NEVER commit secrets
- Follow security best practices [14–16]

---

5. Software Quality and Testing

5.1 Unit Testing

- Recommended coverage: 70–80%
- Include edge cases
- Use automated testing (pytest, unittest)
- Integrate into CI/CD pipeline

5.2 Edge & Critical Case Handling

- Validate inputs
- Handle unexpected states
- Use defensive programming
- Provide meaningful error messages
- Ensure graceful degradation

5.3 Automated Test Results

- Provide test reports
- Show pass/fail metrics
- Highlight failure patterns

---

6. Experimental & Research Results

6.1 Parameter Sensitivity Analysis

- Show how parameter variations affect results
- Use methods like OAT, variance-based analysis
- Provide graphs and interpretations

6.2 Results Analysis

Use:

- Statistical summaries
- Comparison tables
- Jupyter notebooks
- Reproducible code

6.3 Visualization

Recommended visualizations:

- Line charts
- Bar charts
- Scatter plots
- Box plots
- Heatmaps
- Waterfall charts

---

7. UX & UI

7.1 Usability Criteria

Based on Nielsen's 10 Heuristics:

- Learnability
- Efficiency
- Memorability
- Error prevention
- User satisfaction
- Aesthetics
- Accessibility

7.2 UX Documentation

Should include:

- User flows
- Screenshots
- Interaction diagrams
- Accessibility considerations

---

8. Development Workflow & Version Control

8.1 Git Best Practices

- Clean commit history
- Use branches
- Pull requests + code review
- Tags for releases

8.2 Prompt Engineering Log

If using AI tools:

- Document prompts

- Include prompt evolution
- Note improvements and failures
- Maintain reproducibility

---

9. Cost & Resource Management

9.1 Token/Compute Cost Table

Example:

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**

9.2 Cost Monitoring

- Track usage
- Optimize prompts
- Reduce unnecessary runs

---

10. Extensibility & Maintainability

10.1 Extension Points

Include:

- Plugin architecture
- API-first design
- Lifecycle hooks (beforeCreate, afterUpdate)
- Middleware layers

10.2 Maintainability Principles

- Modularity
- Reusability
- Testability
- Easy debugging
- Clean separation of concerns

---

11. Software Quality Standards

11.1 ISO 25010

Include compliance with:

- Functional suitability

- Reliability
- Usability
- Performance efficiency
- Security
- Maintainability
- Portability

---

12. Submission Checklist

Includes:

- Full documentation
- Architecture diagrams
- README
- API docs
- Coding standards
- Configuration files
- Tests + coverage
- Research analysis
- Visualizations
- Git hygiene
- Deployment instructions

---

13. Additional Standards

Includes:

- ISO/IEC 25010
- MIT software quality guides
- Google engineering practices
- Microsoft REST API guidelines
- Nielsen usability heuristics

---

14. Important Note

This document represents a high academic and professional standard for M.Sc. software projects. Generative AI tools (LLMs) can support rapid prototyping, but **human validation and rigorous testing are mandatory**.

---

15. English References

(Full list preserved exactly as in original.)