Verification and Validation Report: TPG

Team 3, Tangle
Calvyn Siong
Cyruss Allen Amante
Edward Gao
Richard Li
Mark Angelo Cruz

March 1, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Symbols, Abbreviations and Acronyms

symbol	description
TPG	Tangled Program Graphs
DNNs	Deep Neural Networks
RL	Reinforcement Learning
SRS	Software Requirement Specification
FR	Functional Requirement
NFR	Non-Functional Requirement
SLN	System Level Number
VnV	Verification and Validation

Contents

List of Tables

List of Figures

This document cohesively summarizes the results of each test as specified in the VnV Plan documentation.

3 Functional Requirements Evaluation

3.1 MuJoCo Integration

Table 1: MuJoCo Integration Tests

Test Id	**Notes**	**Result**
FR-SLN1	When executing the appropriate script, all MuJoCo environments can be run. The best-performing agent within the policy can be visualized using OpenGL or an MP4 file.	Pass
FR-SLN2	MuJoCo environments within the TPG framework can be successfully run within the Digital Research Alliance, enabling research to be conducted by executing experiments.	Pass

3.2 Experiment Visualization

Table 2: Experiment Visualization Tests

Test Id	**Notes**	**Result**
FR-SLN3	When an experiment is running or finished training, the best performing policy can be visualized using the TPG CLI tool.	Pass

3.3 Github Actions CI/CD Pipeline

Table 3: Github Actions CI/CD Pipeline Tests

Test Id	Notes	Result
FR-SLN4	Affirmed that the “Build TPG Project” pipeline properly builds the TPG framework with updated code when changes are pushed to any branch.	Pass
FR-SLN5	When the project building pipeline runs properly, the TPG unit test cases are also automatically ran, and the build will only pass if all the unit tests passes .	Pass
FR-SLN6	Tested that linting and Latex compilation pipeline works as expected.	Pass

3.4 Software Engineering Practices

Table 4: Software Engineering Practices Tests

Test Id	Notes	Result
FR-SLN7	Newly added code in the TPG codebase follows Google’s C++ Style Guide and software engineering best practices such as design patterns, and object-oriented design. This includes careful review and consideration of code readability, extendability, maintainability and scalability. A linter has also been implemented to check for such styling as discussed in 3.3.	Pass

4 Nonfunctional Requirements Evaluation

4.1 Usability

4.2 Performance

Table 5: Performance Tests

Test Id	**Result**
NFR-SLR4	Pass

For NFR-SLN4, test cases within TPG for the experimental environments have been implemented to check for the accuracy of the numerical computations associated during training. Declaration of variables with proper types (e.g. signed long or int, unsigned long or int) has also been taken into consideration to reduce issues in the future for extremely large or small numbers that may overflow. TPG has been comprehensively tested to guarantee that all computations with high numerical precision (e.g. during the runtime of an experiment) are accurate and contain an acceptable tolerance limit of 0.00001. The results were inspected manually by comparing the actual output to the anticipated output, and performing a calculation to check for quantitative error, and if such error meets the requirements for numerical precision.

4.3 Operational and Environmental

Table 6: Operational and Environmental Tests

Test Id	**Result**
NFR-SLR6	Pass

For NFR-SLN6, TPG now supports contributions from macOS, Windows, and Linux developers. Previously, only Linux was supported because TPG

```
TEST_CASE("Mujoco_Ant_v4 Reset Function", "[reset]") {
    std::unordered_map<std::string, std::any> params = createDefaultParams();
    Mujoco_Ant_v4 ant(params);
    std::mt19937 rng(1234);

    ant.step_ = 50;

    std::vector<double> qpos = {0.5, 0.8, -0.3};
    std::vector<double> qvel = {0.1, -0.05, 0.05};
    ant.set_state(qpos, qvel);

    std::vector<double> obs(ant.obs_size_, 1.0);
    ant.get_obs(obs);

    ant.reset(rng);

    REQUIRE(ant.step_ == 0);

    for (size_t i = 0; i < ant.state_.size(); i++) {
        REQUIRE(ant.state_[i] == Catch::Approx(0.0).margin(1e-2));
    }
}
```

Figure 1: Example of a Numerical Computation Test

used SCons for C++ builds and Linux-specific dependencies from requirements.txt. With VSCode Dev Containers, a Linux development environment is automatically launched for all developers, ensuring a standardized setup. Simply follow the Wiki instructions to download all necessary Linux dependencies and build the C++ code reliably. Onboarding on a Macbook has been reduced from 2 weeks to just 5 minutes.

4.4 Maintanability

Table 7: Maintainability Tests

Test Id	**Result**
NFR-SLR7	Pass

To satisfy the testing requirements for NFR-SLR7 - establishing a secure and robust repository management system, the team has implemented checks to ensure the repository prevents unauthorized access and defective code integration. The repository where our team is working on GitHub (whereas the base TPG repository is based in Gitlab), and access is controlled through a combination of two-factor authentication (2FA), and a main branch that is protected to ensure that merge requests can only be performed after the TPG project GitHub workflow action pipeline has successfully completed. This pipeline validates the building and testing process, ensuring that only code that passes all checks can be merged into the main branch. Any critical build errors or warnings, create blocking pull request conversations that must be resolved before merging. There is also a specific GitHub workflow that is used to automatically pull changes from GitLab, eliminating the need for manual merging and risk of human error.

4.5 Security

Table 8: Security Tests

Test Id	**Result**
NFR-SLR8	Pass

For NFR-SLN8, the .csv, .txt, .png and .mp4 files that are generated within Classic Control and MuJoCo experiments are ignored by Git when making commits to the public repositories in GitHub and GitLab to reduce chance of oversharing sensitive data. Currently, none of these files generate sensitive data, but to follow best practice and to keep the repository at a clean state, these are not recognized when synchronizing code to each respective repository. Additionally, the team has also manually checked all stored .csv, .txt, .png and .mp4 files along with others that may contain textual information to see if data within them are sensitive and must be kept private.

4.6 Compliance

Table 9: Compliance Tests

Test Id	Result
NFR-SLR9	Pass

The modified codebase is successfully analyzed using Clang-Tidy and Clang-Format within the CI/CD pipeline. Code change discussions take place through pull request conversations made to the main branch. All errors and warnings are generated based on the C++ Style Guidelines. Any critical errors found during the linting process create blocking pull request conversations that must be resolved before merging into the main branch.

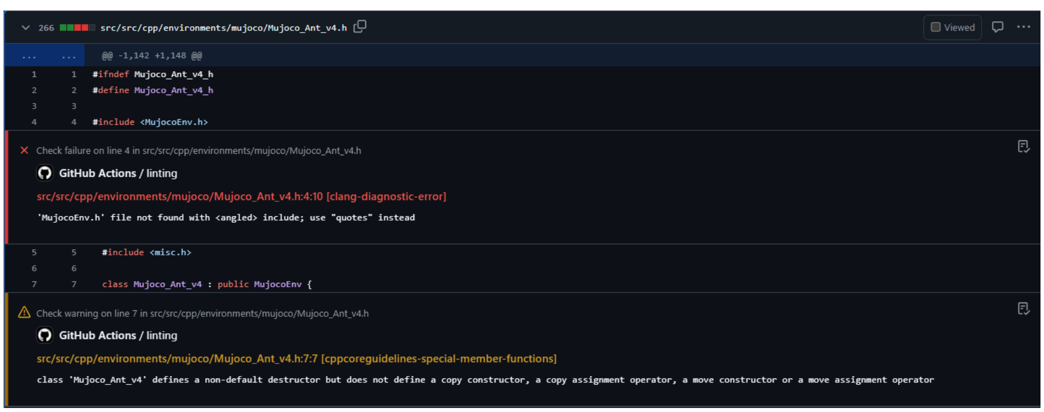

Figure 2: Example of a Linter Error

5 Unit Testing

5.1 Behaviour-Hiding Module

5.1.1 RegisterMachine Crossover Tests

Type: Automatic, Functional

Initial State: The `TPG` and `RegisterMachine` objects are initialized with default parameters and state.

Test Case Derivation: The expected behavior is derived from the correct crossover functionality, chunk splitting, and recombination of `RegisterMachine` objects, ensuring valid instruction patterns and segment lengths.

Test Procedure: The test will be performed as follows:

- **Basic Crossover Functionality Test:**
  - **Input:** Two parent `RegisterMachine` objects.
  - **Output:** Two child `RegisterMachine` objects with valid instructions and actions.
  - **Test Derivation:** Verifies that crossover produces children with reasonable sizes, valid actions, and instruction patterns derived from both parents.
- **Chunk Splitting and Recombination Test:**
  - **Input:** Two parent `RegisterMachine` objects with predefined instruction sequences.
  - **Output:** Two child `RegisterMachine` objects with instruction counts and patterns derived from both parents.
  - **Test Derivation:** Ensures that crossover produces children with valid instruction counts and different instruction patterns.
- **Crossover Constraints Test:**
  - **Input:** Two parent `RegisterMachine` objects with predefined instruction sequences.

- **Output:** Two child `RegisterMachine` objects adhering to crossover constraints.
- **Test Derivation:** Verifies that crossover points, segment lengths, and resulting program lengths adhere to predefined constraints (`dcmax`, `lsmax`, `dsmax`, `lmin`, `lmax`).

5.1.2 Team Crossover Tests

Type: Automatic, Functional

Initial State: The `TPG` and `team` objects are initialized with default parameters and state.

Test Case Derivation: The expected behavior is derived from the correct crossover functionality of `team` objects, ensuring valid team sizes, atomic program preservation, and adherence to team size limits.

Test Procedure: The test will be performed as follows:

- **Single Program Teams - Linear Crossover Test:**
  - **Input:** Two parent teams with single programs.
  - **Output:** A child team with one program.
  - **Test Derivation:** Verifies that crossover produces a child team with a single program and valid atomic count.
- **Multi-Program Teams - Team Crossover Test:**
  - **Input:** Two parent teams with multiple programs.
  - **Output:** A child team with programs derived from both parents.
  - **Test Derivation:** Ensures that crossover produces a child team with a valid size (within `max_team_size`) and at least one atomic program.
- **Atomic Program Preservation Test:**
  - **Input:** Two parent teams with atomic and non-atomic programs.
  - **Output:** A child team with at least one atomic program.

  - **Test Derivation:** Verifies that crossover preserves atomic programs in the child team.
- **Team Size Limits Test:**
  - **Input:** Two parent teams with the maximum number of programs.
  - **Output:** A child team with a size within `max_team_size`.
  - **Test Derivation:** Ensures that crossover produces a child team adhering to the predefined team size limit.

5.2 Mujoco Module

5.2.1 Mujoco Environment Test

Type: Automatic, Functional

Initial State: The MuJoCo environment is initialized using the `MockMujocoEnv` class with the appropriate model path.

Test Case Derivation: The expected behavior is derived from the correct initialization, state setting, and simulation execution of the MuJoCo environment.

Test Procedure: The test will be performed as follows:

- **Simulation Initialization Test:**
  - **Input:** Model path determined by the `determine_tpg_env()` function.
  - **Output:** Successful initialization of the MuJoCo environment.
  - **Test Derivation:** Verifies that the `initialize_simulation()` function correctly initializes the MuJoCo environment, ensuring that the model (`m_`) and data (`d_`) pointers are not null.
- **Set State Test:**
  - **Input:** Position vector `qpos` set to `{0.5, 0.5, ...}` and velocity vector `qvel` set to `{0.1, 0.1, ...}`.

  - **Output:** Updated state in the MuJoCo environment.
  - **Test Derivation:** Ensures that the `set_state()` function correctly updates the position and velocity states in the MuJoCo environment, verifying that `d_->qpos` and `d_->qvel` match the input values.

- **Do Simulation Test:**
  - **Input:** Control vector `control` set to `{0.2, 0.2, ...}` and a step count of 5.
  - **Output:** Updated control values in the MuJoCo environment.
  - **Test Derivation:** Confirms that the `do_simulation()` function correctly applies the control inputs and updates the simulation state, ensuring that `d_->ctrl` matches the input control values.

Test Cases

Test Case 1: Simulation Initialization

- **Description:** Tests the initialization of the MuJoCo simulation environment.
- **Steps:**
  1. Create a `MockMujocoEnv` object with the model path.
  2. Call `initialize_simulation()`.
  3. Verify that `m_` and `d_` are not null.

Test Case 2: Set State

- **Description:** Tests the ability to set the state of the MuJoCo environment.
- **Steps:**
  1. Create a `MockMujocoEnv` object and initialize the simulation.
  2. Set `qpos` to `{0.5, 0.5, ...}` and `qvel` to `{0.1, 0.1, ...}`.
  3. Call `set_state(qpos, qvel)`.
  4. Verify that `d_->qpos` and `d_->qvel` match the input values.

Test Case 3: Do Simulation

- **Description:** Tests the execution of a simulation step with control inputs.
- **Steps:**
  1. Create a `MockMujocoEnv` object and initialize the simulation.
  2. Set `control` to `{0.2, 0.2, ...}`.
  3. Call `do_simulation(control, 5)`.
  4. Verify that `d_->ctrl` matches the input control values.

5.2.2 Mujoco Ant Test

Type: Automatic, Functional

Initial State: The `Mujoco_Ant_v4` environment is initialized.

Test Case Derivation: The expected value is based on the logic that the environment should be terminal when the step count reaches 200, as per the environment's design.

Test Procedure: The test will be performed as follows:

- **Healthy Reward Test:**
  - **Input:** None.
  - **Output:** Returns `healthy_reward_`.
  - **Test Derivation:** Verifies that the `healthy_reward()` function correctly returns the predefined `healthy_reward_` value.
- **Control Cost Test:**
  - **Input:** Action vector `{0.1, -0.1, 0.2, 0.3}`.
  - **Output:** Calculated control cost.
  - **Test Derivation:** Ensures the `control_cost()` function computes the cost using `control_cost_weight_` and the squared sum of action values.

- **Contact Cost Test:**
  - **Input:** None.
  - **Output:** Non-negative contact cost.
  - **Test Derivation:** Confirms that the `contact_cost()` function always returns a non-negative value.
- **Is Healthy Test:**
  - **Input:** Modify `qpos[2]` to test health conditions.
  - **Output:** Boolean indicating health status.
  - **Test Derivation:** Validates that `is_healthy()` returns `true` when `qpos[2]` is within the healthy range and `false` otherwise.
- **Simulation Step Test:**
  - **Input:** Action vector `{0.1, -0.1, 0.2, 0.3}`.
  - **Output:** Finite reward and incremented `step_`.
  - **Test Derivation:** Checks that `sim_step()` processes actions correctly, updating the environment state and returning a valid reward.
- **Get Observation Test:**
  - **Input:** None.
  - **Output:** Non-zero observation vector.
  - **Test Derivation:** Ensures `get_obs()` reflects the current state of the environment in the observation vector.
- **Reset Function Test:**
  - **Input:** Random number generator.
  - **Output:** Reinitialized environment state.
  - **Test Derivation:** Verifies that `reset()` brings the environment back to its initial state, setting `step_` to 0 and state values close to zero.

5.2.3 Mujoco Half Cheetah Test

Type: Automatic, Functional

Initial State: The `Mujoco_Half_Cheetah_v4` environment is initialized with default parameters.

Test Case Derivation: The expected behavior is derived from the correct initialization, terminal condition, control cost calculation, simulation step execution, and reset functionality of the environment.

Test Procedure: The test will be performed as follows:

- **Initialization Test:**
  - **Input:** Default parameters.
  - **Output:** Correct initialization of environment variables.
  - **Test Derivation:** Verifies that `n_eval_train_`, `n_eval_validation_`, `n_eval_test_`, and `max_step_` are set correctly.
- **Terminal Condition Test:**
  - **Input:** Step count set to 200.
  - **Output:** `terminal()` returns `true`.
  - **Test Derivation:** Ensures the environment terminates when the step count reaches 200.
- **Control Cost Test:**
  - **Input:** Action vector `{0.1, -0.1, 0.2}`.
  - **Output:** Calculated control cost.
  - **Test Derivation:** Confirms that `control_cost()` computes the cost using the squared sum of action values.
- **Simulation Step Test:**
  - **Input:** Action vector `{0.1, -0.1, 0.2}`.
  - **Output:** Finite reward and incremented step count.

- **Test Derivation:** Verifies that `sim_step()` processes actions correctly and updates the step count.

- **Reset Function Test:**
  - **Input:** Random number generator and modified state.
  - **Output:** Reinitialized environment state.
  - **Test Derivation:** Ensures `reset()` resets the step count and state values to initial conditions.

5.2.4 Mujoco Hopper Test

Type: Automatic, Functional

Initial State: The `Mujoco_Hopper_v4` environment is initialized with default parameters.

Test Case Derivation: The expected behavior is derived from the correct initialization, terminal condition, healthy reward, control cost calculation, health check, simulation step execution, observation retrieval, and reset functionality of the environment.

Test Procedure: The test will be performed as follows:

- **Initialization Test:** Similar to Half Cheetah or Ant test, but the input is default parameters.
- **Terminal Condition Test:** Similar to Half Cheetah or Ant test, but the input includes modifying `qpos[1]` to test the healthy z range and step count.
- **Healthy Reward Test:** Similar to Half Cheetah or Ant test, but the input is none, and the output is `healthy_reward_`.
- **Control Cost Test:** Similar to Half Cheetah or Ant test, but the input is action vector `{0.1, -0.1, 0.2}`.
- **Is Healthy Test:** Similar to Half Cheetah or Ant test, but the input includes modifying `qpos[1]` and `qpos[2]` to test the healthy z range and angle range.

- **Simulation Step Test:** Similar to Half Cheetah or Ant test, but the input is action vector `{0.1, -0.1, 0.2}`.
- **Get Observation Test:** Similar to Half Cheetah or Ant test, but the input includes manually setting `qpos` and `qvel` to non-zero values.
- **Reset Function Test:** Similar to Half Cheetah or Ant test, but the input includes modifying `qpos`, `qvel`, and step count before resetting.

5.2.5 Mujoco Humanoid Standup Test

Type: Automatic, Functional

Initial State: The `Mujoco_Humanoid_Standup_v4` environment is initialized with default parameters.

Test Case Derivation: The expected behavior is derived from the correct initialization, terminal condition, simulation step execution, observation retrieval, and reset functionality of the environment.

Test Procedure: The test will be performed as follows:

- **Initialization Test:** Similar to Hopper or Half Cheetah test, but the input is default parameters.
- **Terminal Condition Test:** Similar to Hopper or Half Cheetah test, but the input is step count set to 200.
- **Simulation Step Test:** Similar to Hopper or Half Cheetah test, but the input is action vector `{0.1, -0.1, 0.2}`.
- **Get Observation Test:** Similar to Hopper or Half Cheetah test, but the input includes verifying non-zero observation values.
- **Reset Function Test:** Similar to Hopper or Half Cheetah test, but the input includes modifying `qpos`, `qvel`, and step count before resetting.

5.2.6 Mujoco Inverted Double Pendulum Test

Type: Automatic, Functional

Initial State: The `Mujoco_Inverted_Double_Pendulum_v4` environment is initialized with default parameters.

Test Case Derivation: The expected behavior is derived from the correct initialization, terminal condition, simulation step execution, observation retrieval, and reset functionality of the environment.

Test Procedure: The test will be performed as follows:

- **Initialization Test:** Similar to Humanoid Standup or Hopper test, but the input is default parameters.
- **Terminal Condition Test:** Similar to Humanoid Standup or Hopper test, but the input includes modifying `site_xpos[2]` to test the terminal threshold and step count.
- **Simulation Step Test:** Similar to Humanoid Standup or Hopper test, but the input is action vector `{0.1}`.
- **Get Observation Test:** Similar to Humanoid Standup or Hopper test, but the input includes manually setting `qpos` and `qvel` to non-zero values.
- **Reset Function Test:** Similar to Humanoid Standup or Hopper test, but the input includes modifying `qpos`, `qvel`, and step count before resetting.

5.2.7 Mujoco Inverted Pendulum Test

Type: Automatic, Functional

Initial State: The `Mujoco_Inverted_Pendulum_v4` environment is initialized with default parameters.

Test Case Derivation: The expected behavior is derived from the correct initialization, terminal condition, simulation step execution, observation retrieval, and reset functionality of the environment.

Test Procedure: The test will be performed as follows:

- **Initialization Test:** Similar to Inverted Double Pendulum or Humanoid Standup test, but the input is default parameters.

- **Terminal Condition Test:** Similar to Inverted Double Pendulum or Humanoid Standup test, but the input includes modifying `qpos[1]` to test the terminal threshold and step count.
- **Simulation Step Test:** Similar to Inverted Double Pendulum or Humanoid Standup test, but the input is action vector `{0.1}` and the expected reward is `1.0`.
- **Get Observation Test:** Similar to Inverted Double Pendulum or Humanoid Standup test, but the input includes manually setting `qpos` and `qvel` to non-zero values and verifying the observation vector.
- **Reset Function Test:** Similar to Inverted Double Pendulum or Humanoid Standup test, but the input includes modifying `qpos`, `qvel`, and step count before resetting.

5.2.8 Mujoco Reacher Test

Type: Automatic, Functional

Initial State: The `Mujoco_Reacher_v4` environment is initialized with default parameters.

Test Case Derivation: The expected behavior is derived from the correct initialization, terminal condition, control cost calculation, distance retrieval, simulation step execution, observation retrieval, and reset functionality of the environment.

Test Procedure: The test will be performed as follows:

- **Initialization Test:** Similar to Inverted Pendulum or Inverted Double Pendulum test, but the input is default parameters.
- **Terminal Condition Test:** Similar to Inverted Pendulum or Inverted Double Pendulum test, but the input is step count set to 200.
- **Control Cost Test:** Similar to Hopper or Half Cheetah test, but the input is action vector `{0.1, -0.1}`.
- **Get Distance Test:** Unique to Reacher, the input is none, and the output is a distance vector of size 2.

- **Simulation Step Test:** Similar to Inverted Pendulum or Inverted Double Pendulum test, but the input is action vector `{0.1, -0.1}`.
- **Get Observation Test:** Similar to Inverted Pendulum or Inverted Double Pendulum test, but the input includes verifying non-zero observation values.
- **Reset Function Test:** Similar to Inverted Pendulum or Inverted Double Pendulum test, but the input includes modifying `qpos`, `qvel`, and step count before resetting.

6 Changes Due to Testing

6.1 Feedback from Rev 0

The feedback given by the instructor and teaching assistant during Revision 0 was essential in guiding the next steps as the team looks toward the final demonstration. Emphasis was placed on ensuring that usability testing was executed systematically rather than in the more ad-hoc manner initially planned by the team. Some additional changes to be made include ensuring that unit testing and benchmarking of the implemented environments are cohesively executed and investigating whether the integration of deployment within the DRA is possible.

7 Automated Testing

As a result of the team's conversion from building the project using SCons to CMake, automated testing became significantly easier to execute and debug. To run any automated tests within a developer's local environment, a developer can simply execute a command to build the project. This not only compiles everything but also runs all automated tests. If a developer wishes to run only the tests, they must navigate to the directory where the tests were already compiled (typically `/build/tests`). From there, the command `ctest` can be entered into the command prompt. Similar to the compilation process, all automated tests are executed once this command is run.

From the repository's point of view, tests are executed using GitHub Actions or GitLab CI (depending on which repository is being viewed). Both linting and compilation are performed using the same commands that would be executed within a developer's local environment. These tests run when a new pull request is made to the main branch, ensuring that all tests pass before merging. The compiler workflow is also executed after merging into the main branch to ensure no errors or unintended changes in code behaviour have occurred. If any test or workflow fails, the logs of the workflow can be reviewed, providing a detailed summary of the reason for failure. This not only allows for easier debugging but also resolves the "works on my machine" issue.

8 Trace to Requirements

9 Trace to Modules

10 Code Coverage Metrics

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Reflection.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

   One part that went well for this deliverable is that valuating the functional and non functional requirements was relatively simple, as we were able to trace it back to our VNV plan and SRS report.

2. What pain points did you experience during this deliverable, and how did you resolve them?

3. Which parts of this document stemmed from speaking to your client(s) or a proxy (e.g. your peers)? Which ones were not, and why?

4. In what ways was the Verification and Validation (VnV) Plan different from the activities that were actually conducted for VnV? If there were differences, what changes required the modification in the plan? Why did these changes occur? Would you be able to anticipate these changes in future projects? If there weren't any differences, how was your team able to clearly predict a feasible amount of effort and the right tasks needed to build the evidence that demonstrates the required quality? (It is expected that most teams will have had to deviate from their original VnV Plan.)