Version 1.1

Software Test Plan

November 09, 2025

HapTech

Project Sponsor: Dr. Reza Razavian

Project Mentor: Karthik Srivathsan Sekar

Team Members: Landon Coonrod, Matthew Gardner, Peter Hilbert, Karissa Smallwood

Table of Contents

1. Introduction

Our team created a modular software framework designed to support human-robot interaction experiments using the Franka Research 3 robot. Our primary goal is to make research involving haptic touch and robotic systems more efficient, reusable, and consistent by providing researchers with an interface to configure, run, and monitor experiments. Our product integrates several key components: the robot hardware, CHAI3D simulation environment, and the command line interface all into one system that operates within a controlled laboratory setting.

Since our product functions as a research tool, ensuring system reliability and usability is very important. Errors, timing issues, or confusing designs could disrupt experiments or compromise data accuracy. To prevent these problems, our team developed a structured testing plan aimed at validating the system's performance, safety, and ease of use across all stages of development.

Software testing is the process of evaluating a system to confirm that it behaves as expected and meets its designed requirements. Testing helps detect issues early, verify that individual components work correctly, and ensure the system performs reliably under real conditions. Since our product has a combination of software, simulation, and robotic hardware, comprehensive testing is essential to make sure all modules work together seamlessly.

Our testing plan includes three primary levels of testing:

- Unit testing verifies the correctness of individual modules and ensures that core functionality behaves as intended.
- Integration testing validates the interaction between components such as the Client, Simulation Manager, and Interface, confirming stable data flow and timing.
- Usability testing evaluates how effectively researchers and participants can use the interface and complete experimental tasks.

Integration testing will be our main focus due to our product's distributed architecture and real-time communication requirements, where timing and safety are most critical. Usability testing will also be emphasized to confirm that the system supports efficient workflows and a positive user experience. Unit testing, while important, will mainly be used to make sure there is a stable foundation before higher-level testing begins.

The first phase of our testing plan focuses on unit testing, which will ensure each individual component functions correctly before proceeding to integration testing and user evaluations.

2. Unit Testing

2.1 Purpose

The purpose of unit testing is to verify that the smallest functional components of the HapTech system such as individual classes, methods, and functions, perform as intended in isolation. According to AWS, unit testing focuses on individual units of source code, ensuring that when a failure occurs, the issue can be traced to a specific and well-defined part of the program [1]. As IBM notes, unit testing accelerates development by detecting defects early, improving reliability, and ensuring long-term maintainability [2].

For HapTech, unit testing serves three primary goals:

1. Validate the correctness and stability of core functionality that can be executed independently.
2. Facilitate faster debugging by isolating bugs at the function or class level.
3. Catch bugs introduced by new changes during development rather than in production.
4. Build confidence in the integrity of modules before they are integrated into the full system.

2.2 Tools and Frameworks

The following tools and frameworks will be used to develop and execute unit tests:

Language	**Testing Framework**	**Description**
C++	Google Test (GTest)	A widely used, robust unit testing framework that integrates well with CMake/Make build systems. GMock (Google Mock) will also be used to simulate interfaces like the SimulationManager.
Python	PyTest	A simple, lightweight, and reliable testing framework for verifying client-side and utility modules. The pytest-cov plugin will be used for measuring code coverage.

These tools were selected for their reliability, simplicity of integration, and support for automated reporting and continuous testing.

2.3 Testing Process

Unit tests will be developed alongside their respective modules, following a test-driven development (TDD)-inspired approach when feasible. The general process includes:

1. **Test Setup:**
    - Create a dedicated testing directory (i.e. '/tests') within both the client and core repositories.
    - Each test file will mirror the directory structure of the main source tree.
2. **Test Implementation:**
    - Write unit tests for the smallest functional units possible (methods, constructors, or utility functions).
    - Use GMock to isolate dependencies like the SimulationManager during C++ testing.
3. **Automation:**
    - Automated test runs will be triggered through command-line scripts or CI workflows to ensure consistency.
4. **Reporting and Review:**
    - Test reports will be generated automatically by GTest and PyTest.
    - Failed tests will be logged with stack traces, and results will be reviewed before merging new code.

To evaluate unit testing effectiveness, the following metrics will be tracked:

- **Test Pass/Fail Counts:** Number of successful and failed test cases per module.
- **Code Coverage:** Measured using '--coverage' for GTest and the 'pytest-cov' plugin for PyTest.

100% of unit tests should be passed upon deployment/release. While not all code will be unit tested, coverage will focus on the most critical and isolated modules. A target of at least **80%** code coverage will be maintained for tested files.

2.4 Scope of Unit Testing

The table below outlines the scope of our unit testing.

Component	**Methods/Classes**	**Description**
Core – Object Classes	createObject()	Validates correct creation and initialization of 3D simulation objects.
Core – Condition Classes	Constructors, evaluate()	Tests condition validation and simulated environment triggers using mocked data.
SimulationManager	run() and close()	Confirms that the 3D simulation initializes and terminates without error.
Client	XML parser module	Ensures XML configuration files are interpreted correctly and invalid input is rejected.

Unit testing will focus on the modules that can be tested independently of the hardware and real-time haptic loop and without client-server dependence. The parts of the system that do require hardware or inter-process communication, like gRPC communication, trial flow, and haptic rendering, are better candidates for integration testing, and will be covered in the next section.

2.5 Detailed Unit Test Plans

2.5.1 Custom Objects - createObject()

Custom object logic can be tested in isolation without actually sending configuration over the network, so it is a strong candidate for unit testing. The goal of testing this logic is to validate correct object creation and parameter handling.

- **Equivalence Partitions:**
  - Size parameters: must be > 0
  - Color RGB values: $0 \leq$ value ≤ 1
  - Gravity scale: ≥ 0
  - Bounciness: $0 \leq$ value ≤ 1
- **Boundary Values:**
  - Test radius = 0 (invalid)
  - RGB values < 0 (invalid), = 0 and 1 (valid), > 1 (invalid)
  - Gravity scale < 0 (invalid), = 0 (valid)
  - Bounciness < 0 (invalid), = 0 or 1 (valid), > 1 (invalid)

Expected Results: Valid input should produce a valid object instance (not null). Invalid input should raise an exception or return null.

2.5.2 Trial Conditions - Constructor and evaluate()

Trial condition logic can be unit tested because it is mostly isolated logic that can be tested using a mocked SimulationManager. The goal of testing trial conditions is to verify that condition setup and evaluation behave as expected using mock simulation data.

- **Equivalence Partitions:**
  - Duration ≥ 0 (valid)
  - Duration < 0 (invalid)
- **Boundary Values:**
  - Duration = 0 (valid)
  - Trial time = duration (true)
  - Trial time = duration − 0.01 (false)
  - Trial time = duration + 0.01 (true)

Expected Results: Conditions should initialize successfully for valid input. The evaluate() method should return true only when criteria are met under mocked SimulationManager values.

2.5.3 SimulationManager - run() and close()

The basic functionality of SimulationManager can partially be unit tested, specifically, the setup and teardown of the 3D environment, as that only uses CHAI3D. However, the scene manipulation interface will fall under integration testing, as that partially depends on the custom object interface. The goal with the unit testing we will be doing for SimulationManager is to ensure the simulation environment can be created and destroyed without errors.

- **Procedure:**
    - Initialize a SimulationManager instance.
    - Call run() to start the simulation.
    - Call close() to end it.

Expected Results: The 3D environment window should open and close successfully without exceptions.

Additional Test Considerations: In addition to verifying that the SimulationManager can start and stop without errors, further input categories and edge cases should be defined to improve test completeness. Specifically, it should gracefully handle scenarios where run() is called while the simulation is already running, and where close() is called when the simulation is not running yet. These are invalid inputs, and should simply be rejected without crashing the program. Additionally, the simulation should not start if there is an issue with the connection to the robot in an effort to avoid crashes mid-trial. These tests ensure that the SimulationManager behaves predictable and fails safely under realistic error conditions.

2.5.4 Client XML Parser

The parsing of the XML files as well as the construction of the proper configuration data structures can be tested in isolation, since this functionality is separate from actually sending the trial request. The purpose of these unit tests is to ensure proper validation and parsing of XML configuration files.

Test Parameters:

- Each XML section (Trial Conditions, HUD Text, Scene Settings, Data Collection, Objects) will be tested with:
    - **Valid Input:** Proper tags and values.
    - **Invalid Input:** Missing or misstructured tags. Missing, malformed, or out-of-range attribute values.

Expected XML Structures:

- Trial conditions
    - There must be StartConditions and EndConditions present
    - For each Condition within StartConditions and EndConditions:
        - It must have a name (any string), success (true/false), and priority (int) value.
        - It must have a Parameters tag within it, which can contain any attributes.

- HUD Text
    - There must be StartMessage, EndMessage, and FailureMessage present within the HUDText.
- Scene Settings
    - There must be a BackgroundColor, which should contain r, g, and b int values between 0 and 255.
    - There must be a Camera, which should have a Position, LookAt, and Up, each having x, y, and z float values.
- Data Collection Settings
    - Within DataCollection, there must be a Device, which contains Position, Velocity, AngularVelocity, Orientation, ProxyPosition, and Force, each with an "enabled" attribute with a value of either true or false.
    - Within Objects, there must be Position, Orientation, Contact, ContactPoint, ContactNormal, PenetrationDepth, Force, SpringForce, and Damping force, each also with an "enabled" attribute with a value of either true or false.
- Objects
    - Within Objects, each Object must have an id and type attribute, both of which can be any string.
    - Each Object's Position and Orientation must have x, y, and z float values.
    - Each Object's Color must have r, g, and b float values between 0 and 1.
    - Each Object must have a Parameters tag, which can have any attributes.
- Each of these sections will be tested with all valid input, some invalid input, and some omitted input.
    - Invalid input should produce an error or null output.
    - Valid input should produce data structures with values expected in the input configuration. For example, if an object's position was specified to be x=0.5, y=0.2, and z=0.1 in the configuration, we should expect that the resulting data structure has that object's position as a Vector3 with values (0.5, 0.2, 0.1).

Expected Results: If valid input is provided, valid input data structures should be constructed with correct values (e.g., Object position of (0.5, 0.2, 0.1) produces a Vector3(0.5, 0.2, 0.1)). If invalid input, the parser should return an error or null output.

2.6 Logging and Reporting Plan

- **GTest Reporting:** Automatically prints pass/fail results; can output XML reports for CI.
- **PyTest Reporting:** Summarizes test counts, failed cases, and durations.
- **Coverage Reports:**
    - C++: compile with '--coverage' and process using gcov or lcov.
    - Python: use 'pytest --cov=src/ --cov-report=html'.

Reports will be reviewed upon completion of coding tasks to identify bugs and untested areas of the codebase.

3. Integration Testing

3.1 Purpose

Integration testing is to verify that separately tested components combine together correctly to form the whole system, and that the data flows across boundaries correctly. Completely, ordered, and timely. In practice, integration testing joins modules and exercises their interfaces to ensure they behave as a coherent system.

3.2 Rationale for Integration Testing

Haptechs architecture makes integration testing essential as there are so many different key parts:

- Real Time Constraints: The haptic/robotic loop and CHAI3D control paths are timing sensitive and massively important to the studies being carried out.
- Cross process boundaries: Client - Simulation Manager - Robot Interface communicative via gRPC/Streams. All can be very common failure points affecting the entire system.
- Safety: Wrong commands or wrong timing could cause unsafe robot behavior.

3.3 Integration Testing Approach

Style: Hybrid (top down + bottom up)

- Bottom-up on device/robot paths (Mock Franka → CHAI3D loop → Robot Interface) to harden cycle timing, queueing, and safety interlocks before adding upstream flows.
- Top-down on user flows (Client/UI → Simulation Manager → Trial lifecycle) to validate setup, execution, monitoring, and export.

Assembly order

1. Pairwise boundaries
2. Triads
3. Full-stack scenarios (incl. soak).

Contract-first

- Freeze gRPC/Protobuf + XML contracts.
- Build positive and negative suites (missing fields, bad ranges, version mismatches).
- Validate with schema checks (XSD/JSON Schema) and protobuf conformance tests before scenario runs.

Fault Injection

- Network: drop, jitter, reorder, bandwidth cap.
- Timing: synthetic step overruns, loop jitter.
- Configs: malformed XML, incompatible versions, illegal ranges.
- Robot faults (mock/HIL): limit breach, torque spike, sensor dropout.

Soak runs

- 2 to 4 hour sustained trials to catch lifecycle leaks, cumulative jitter drift, and I/O back-pressure.

3.4 Test Environments

Three tiers to control risk and isolate defects:

1. Headless simulation: No robot only sim manager
2. CHAI3D Simulation: CHAI3D engine, simulation manager, No robot
3. Hardware in loop: Full hardware run with all systems

3.5 Critical Integration Points and Test Design

3.5.1 Client to/from Simulation Manager

1. What's tested:
    a. Trial lifecycle
    b. XML config loading, scene starting
    c. Data Collection, lossless CSV
2. Expected data flow:
    a. XML file to Simulation Manager

    b. Data stream from Simulation Manager to Client
3. Error Handling validation:
    a. Inject miscreated XML, missing required field, bad values
    b. Drop connection mid trial
    c. Start another trial while a trial is already running
4. Logging & Evaluation:
    a. JSON logs
    b. Store RPC timeline (request/response ts), state transitions, and scene diff
    c. Pass if: all contracts verified, no unhandled exceptions, no orphaned threads, latency/jitter under thresholds.

3.5.2 Simulation Manager to/from Robot Interface

1. What's Tested:
    a. Pose/torques from Sim Manager
    b. Realtime feedback
    c. Safety interlocks
2. Expected data flow:
    a. Real time data from Robot Interface to Simulation Manager
        i. Joint Position
        ii. Joint Torque
        iii. Robot Relative Position
3. Logging & Evaluation:
    a. Per cycle metrics (latency, jitter, queue depth) safety events
    b. Pass if: no missed data, no limits broken, jitter within bounds

3.6 Logging and Reporting Plan

- Structured JSON logs across all components with correlation IDs
- Artifacts per test:
    - Input config (XML, Protobuf versions), seeds, and environment
    - RPC traces (timestamps, status codes), latency/jitter, error/fault events
- Automated Summaries: one page per test case (ID, environment, key metrics, pass/fail, defects)
- Review cadence: after each integration session

3.7 Metrics for Success

- Contract coverage: All defined RPCs and XML fields tested with positive and negative cases
- Data integrity: No schema violations, no losses, contiguous sequence numbers
- Timing & Stability

- Robustness: All injected faults produce correct, user visible errors and safe states
- Safety: All limit checks and interlocks verified, and safety event leads to safe stop within bounds

3.8 Criteria for Progression to Usability Testing

We will proceed when all of the following hold:

- Critical tests pass twice consecutively
- Timing budgets met in simulation
- No defects
- End to end demo scenario
- Safety gates cleared

4. Usability Testing

4.1 Purpose

Usability testing is when the design is tested by a group of users. The purpose of usability testing is to find flaws within the design that may have been overlooked previously [3]. When observing the users testing the design, you'll gain valuable feedback about how well your product works, which you can use to improve the system [3].

For HapTech, usability testing is very important for this project because it is research based. This means our product's purpose is to be used for experiments that involve participants interacting with the system. Thus, it is important to have the design thoroughly tested to avoid any issues popping up during an experiment.

4.2 Intended Users and Testing Environment

Primary Users:

- Research Members: they will set up experiments, monitor the robot's behaviors, and collect data
- Experiment participants: they will interact with the haptic system as part of a study

User Environment:

- Laboratory setting with the Franka Research 3 robot, a computer running our systems server and a separate computer running the client side.
- Controlled experimental environment with both physical (robot) and digital (UI) components
- Limited distractions

4.3 Usability Risks and Challenges

Complexity:

- Researchers may need to configure multiple parameters within the XML files

Unfamiliar UI:

- Users unfamiliar with the system may find the setup confusing at first.

Safety and precision:

- Incorrect user input (pushing too hard, going out of bounds) could cause physical robot errors and interrupt experiments.

Mixed User types:

- The interface must work for both researchers (technical users) and participants (non-technical users).

Limited testing time:

- There might be strict timing for experiments, leaving little time for troubleshooting errors.

4.4 Testing Approach and Rationale

A combination of expert reviews and user studies will be our usability testing approach.

- Expert reviews can be used to discover high-level usability issues early.
- While user studies will provide feedback from both researchers and participants
- Since our system is used for research and experiments, using task-based observation and acceptance testing is the most ideal.
- Will do direct observation testing with realistic lab settings to mirror how the product would actually be used.

4.5 Testing Methods

4.5.1 Expert Reviews

- Participants: 2-3 usability experts familiar with UI design principles.
- Environment: Will take place in the Reza Lab, using the system prototype.
- Tasks: Evaluate the UI screens for clarity, consistency, and error prevention.
- Recording: They will take notes and screenshots of usability violations.
- Metrics: Number of issues found, the severity rating, and adherence to usability principles.

4.5.2 User Studies

- Participants: 3-5 researchers and 3-5 experiment participants.
- Environment: Will take place in the Reza Lab where the environment will be controlled and the software will be running.
- Tasks:
    - Researcher: Load experiment XML, start trial, monitor robot state, export and analyze data.
    - Participant: Interact with the Franka Research 3 robot adhering to the on-screen instructions.
- Recording: Screen recordings, observer notes, timing data, post-test interviews.
- Metrics: Task completion rate, error count, time on task, subjective ease-of-use ratings on a scale of 1 to 5.

4.5.3 Acceptance Testing

- Participants: Our client, Dr. Reza Razavian and his research team.
- Goal: Verify that the usability meets expectations for deployment in actual research settings.
- Metrics: Confirmation of core usability requirements (setup <5 min, data collection verified, clear feedback to participants).

4.6 Data Analysis

- Compile quantitative metrics (task success, time, errors) and qualitative feedback from comments.
- Identify recurring usability problems and prioritize them by severity.
- Compare expert findings vs real user feedback to validate patterns.
- Implement design refinements (UI layout, terminology, workflow improvements)
- Retest updated versions as part of iterative refinement.

4.7 Testing Timeline

Phase	Activity	Duration	Notes
Week 1	Expert review of current prototype.	1 week	Identify critical usability issues early.
Week 2-3	User testing with researchers.	2 weeks	Observe usability in a near-final environment.
Week 4	Participant usability sessions.	1 week	Evaluate ease of use for study participants.
Week 5	Analyze results & document findings.	1 week	Compile report, propose UI improvements.

5. Conclusion

This test plan validates HapTechs framework for the Franka Research 3 for correctness, timing safety, and usability in a controlled lab environment. It prioritizes integration and usability given the system's real-time nature, while maintaining a solid foundation for unit tests.

- Unit testing confirms core modules behave as designed, speeds debugging, and guards against failure. We use Google Test/Mock for C++ and PyTest for python. The target is for 100% test pass and ≥80% coverage on tested files, with a focus on components that can be isolated from hardware. Together these check critical logic before high level tests.
- Integration testing verifies contracts and data flow across the client, simulation, manager, and robot interface with a hybrid top-down/bottom-up approach. It locks configs first, then exercises trial lifecycles, timing, and error paths with fault injection and soak runs in three environments (headless sim, CHAI3D only, hardware-in-loop). Passing here demonstrates stable, ordered, and timely interfaces and safe behavior under stress.
- Usability testing combines expert reviews, researcher/participant studies, and acceptance testing in realistic lab conditions. We capture task success, time, errors, and qualitative feedback, then iterate. Acceptance criteria include fast setup (<5 min), verified data collection, and clear participant feedback, ensuring the tool is effective for real research workflows.

By chaining unit correctness, integration reliability/safety, user effectiveness, this plan provides evidence that the system is error-tolerant, and ready for research deployment. It explains how each layer contributes to overall quality and why we can be confident in the software's reliability, functionality, and usability.

As of November 2025, the team has done high-level integration testing and acceptance testing with our client throughout the course of development. In the coming weeks, there will be more detailed integration testing of edge case scenarios, unit testing, and usability testing to be done. In working towards these next testing goals, the team has completed the design of all test plans and procedures, prepared the testing environment, and finalized the unit and integration testing frameworks using Google Test and PyTest, and we are ready to move forward.

References

[1] AWS, “What is Unit Testing? - Unit Testing Explained - AWS,” *Amazon Web Services, Inc.*, 2024. https://aws.amazon.com/what-is/unit-testing/ [accessed Nov. 05, 2025].

[2] P. Powell and I. Smalley, “Unit testing,” *Ibm.com*, Jun. 02, 2025. https://www.ibm.com/think/topics/unit-testing [accessed Nov. 05, 2025].

[3] “What is Usability Testing?,” *The Interaction Design Foundation*, Jun. 02, 2016. https://www.interaction-design.org/literature/topics/usability-testing?srsltid=AfmBOook8tRM7uxFh69ur7Pz2sbKjT8fUzw-GS9OgcyDKQbwN6C4Jdzj [accessed Nov. 05, 2025].