ChatGPT

Combining Pass-Driven Development and Runtime Type Safety for Resilient Software

By Dinis Cruz and ChatGPT Deep Research

Introduction: Challenges in Dynamic, Evolving Codebases

Modern software systems often evolve rapidly, with frequent changes driven by new requirements and bug fixes. In such dynamic environments – especially when using languages like Python or JavaScript – traditional *test-first* approaches and static type systems can fall short. **Test-Driven Development (TDD)** prescribes writing failing tests before code, but in practice this can disrupt continuous integration when known bugs cause tests to fail. Teams may hesitate to add tests for known issues because it would break the build, leaving important cases untested. Likewise, static type systems (or optional type hints in dynamic languages) catch certain mistakes at compile time, but cannot enforce deeper semantic constraints or prevent runtime errors in evolving architectures. Many high-profile failures have shown the limits of static types alone. For example, the Log4Shell vulnerability (2021) exploited a raw string in a log message to trigger remote code execution – a **type-correct** string that passed static checks but contained malicious content [1]. Similarly, the Equifax breach (2017) was enabled by an unvalidated string parameter that allowed SQL injection [2]. In both cases, a statically-typed "string" field provided no protection, and traditional tests had not anticipated these exploits. These incidents underscore a critical gap: **traditional static typing and upfront tests do not guarantee runtime safety in the face of evolving code and novel inputs**.

Dynamic codebases demand a development approach that can keep pace with change without sacrificing quality. This is where **Pass-Driven Development (PDD)** and **runtime type-safe primitives** come in. PDD flips the script of TDD to ensure the test suite captures real bugs without constantly red lighting the build. Meanwhile, runtime type safety (as implemented in OSBot-Utils' **Type_Safe** primitives and collections) acts as a powerful second line of defense, enforcing correct usage and data integrity at runtime beyond what static types or early tests can check. Together, these techniques enable teams to build **high-quality, resilient, and consumer-aligned software** that catches issues early and adapts gracefully to change. The following sections introduce PDD and Type_Safe primitives in turn, then show – with examples from Dinis Cruz's HTML Graph and MGraph-DB projects – how their combination yields robust software systems.

Pass-Driven Development (PDD): Testing without Breaking the Build

Pass-Driven Development is an evolutionary testing practice that addresses the main pain point of test-first methods in fast-moving projects: keeping the build green. The core idea of PDD is to **start with a passing test, even when targeting a known bug**. In practice, this means writing a test that *succeeds* under the current buggy behavior (for example, asserting the presence of the bug's symptoms or the incorrect output) instead of a test that immediately fails. This counter-intuitive strategy ensures that adding the test will not break the CI pipeline [3]. The known issue is captured in the test suite without halting development momentum. Once the bug is fixed in code, that test will naturally begin to fail – alerting developers that the behavior has changed – at which point the test can be updated to assert

the now-correct behavior. In effect, the test has flipped from a “bug existence” test to a “regression” test for the fix.

This PDD workflow offers several key advantages. First, it guarantees that the testing framework is capable of reproducing the bug scenario in the first place [4]. Writing a passing bug test often involves simulating the exact conditions that trigger the issue, which validates that your tests are hitting the right code paths. Second, it **integrates known bugs into the CI process** in a non-disruptive way [5]. Instead of leaving a bug undocumented until a fix is ready (or commenting out a failing test), the team has a living test case in place. Third, PDD provides immediate feedback when the bug is resolved [6] – the previously passing test will fail as soon as the code no longer exhibits the buggy behavior, signaling that it’s time to update expectations. In summary, *every bug gets a test, and the test suite stays green throughout*. This approach preserves development velocity while dramatically increasing coverage of edge cases.

Dinis Cruz has applied PDD extensively in his development workflow. For example, in the context of OSBot-Utils’ Type_Safe collections, he created tests named for specific issues like `test_bug_dict_dont_support_type_checks`. Such tests were written to pass under the old limitations of the system, effectively **documenting the edge case** and capturing the known limitation in code [7]. When the underlying code was improved (e.g. adding type-checking support to a dict structure), these tests started failing – confirming that behavior had changed – and could then be moved into a “regression tests” suite or updated to assert the new correct behavior [8]. The end result is a test suite that tells a story of the code’s evolution. Each bug-related test becomes a permanent regression test, a record of a real-world scenario that once failed and now must always pass. *“Instead of just fixing the issues and moving on, [we] create a permanent record of the problems that existed… As we evolve the code, we leave behind a large set of regression tests confirming that we’re not regressing or adding new bugs”* [9]. In other words, PDD turns every resolved bug into a safety net. Over time, the accumulation of these targeted tests leads to extremely high coverage of critical paths **without any explicit mandate** – coverage grows naturally as a byproduct of addressing issues. Perhaps more importantly, there is full traceability between features/bugs and tests: for every fix or implemented feature, there is at least one test capturing its expected behavior or past pitfalls. This traceability is invaluable for maintenance and audits, since one can map test cases directly to user-facing requirements or bug reports.

Crucially, PDD keeps the **CI pipeline resilient**. Even if developers discover a serious bug, they can write and commit a test for it immediately (ensuring it won’t be forgotten) without fear of “breaking the build” and blocking their team [3]. The pipeline remains green as bugs are catalogued in tests. Then, fixes can be done on separate branches or iterations, at which point the failing of those tests serves as a clear indicator that the code behavior has changed as expected. This workflow encourages an attitude of *“no bug left behind”* – even if you can’t fix something this sprint, you can at least write a test to guard against it. The resulting software is **battle-hardened** with respect to known issues: every production bug or edge case found becomes a test, and thus the same problem can never slip through twice. In summary, Pass-Driven Development complements traditional TDD by addressing its blind spot in bug-fixing scenarios. It lets teams **achieve high test coverage and knowledge preservation without compromising continuous integration**. PDD ensures that a dynamic, evolving codebase always has an up-to-date, passing test suite that reflects both the intended features and the catalog of known quirks or past bugs (now turned into regression tests).

OSBot-Utils Type_Safe Primitives: Enforcing Runtime Type Safety

While PDD fortifies the test strategy, an equally important part of building resilient software is strengthening the code's correctness guarantees. **OSBot-Utils' Type_Safe primitives and collections** provide a runtime type safety layer that goes far beyond what static typing or basic tests can offer. In languages like Python, static type checkers (e.g. mypy) might warn about type mismatches at development time, but they do not enforce anything at runtime – and they certainly don't validate the *content* of variables (for instance, a string containing malicious data is still just a `str` to the type checker). Type_Safe primitives were created to address this gap by introducing **strongly-typed, validated data classes** that developers use in place of raw primitives. In fact, Dinis Cruz advocates using "strongly types for everything and *banning the use of primitives like string, int and float*" in favor of Type_Safe variants [10]. OSBot-Utils is built on a "Type-Safe First" philosophy: it provides a suite of classes like `Safe_Str`, `Safe_Int`, `Safe_Float`, etc., each with built-in validation and sanitization logic [11]. These serve as the foundation for domain-specific types (for example, `Safe_UInt__Port` for network port numbers or `Safe_Str__Html` for HTML content) that encode business rules and security checks *in the data types themselves*.

Runtime enforcement is the distinguishing feature here. With Type_Safe classes, checks happen **continuously whenever data is created or manipulated**, not just at boundaries or compile-time [12] [13]. For instance, if you define a data model using Type_Safe, any attempt to assign a wrong-type or out-of-range value will raise an error immediately, right at the operation that violates the contract [13] [14]. This is in stark contrast to conventional Python, where a function annotated to take a `float` might still be called with a string at runtime and lead to a failure far downstream. With Type_Safe, such mistakes are caught at the source. Consider a simple comparison: a traditional function `process_payment(amount: float, user_id: str)` might be called incorrectly as `process_payment("99.99", 12345)` and nothing will stop it – it will run and likely cause a failure deep in the system. With Type_Safe, one would model this as a class with `amount: Safe_Float__Money` and `user_id: Safe_Str__UserId`, and then any improper assignment (e.g. passing a string where a Money float is expected, or an int for a string ID) would either be auto-converted or trigger an immediate exception [13] [15]. In short, **Type_Safe turns type hints into actual runtime guarantees**. The table below (from the OSBot-Utils documentation) highlights how this approach compares to other frameworks:

[16]

Table: Comparison of when and how different approaches validate types. Here we see that Type_Safe checks types on **every operation** (not just at object creation or not at all), and supports domain-specific primitives with automatic sanitization, unlike standard dataclasses or static type checkers [16].

The benefits of this runtime strictness are immense for producing robust, secure code. **Entire categories of bugs and vulnerabilities are prevented by construction.** For example, Type_Safe strings can automatically sanitize dangerous characters or patterns, closing the door on injection attacks. If a developer uses `Safe_Str__Username` (a type for user names that permits only alphanumeric and underscore characters), any attempt to include SQL metacharacters or script tags will be neutralized. The string "`admin' OR '1'='1`" when passed into `Safe_Str__Username` might come out as `"admin_OR_1_1"`, with quotes and spaces replaced by safe characters [17]. The code using this value can proceed knowing it's safe – the risk of SQL injection or HTML injection is eliminated by the data type itself. In effect, **security and data validation are built-in**, not bolted on. A developer writing an SQL query with a `Safe_Str__Username` can be confident that it won't contain rogue quotes or semicolons [17]. OSBot-Utils provides many such specialized types out of the box:

`Safe_Str__Html` will ensure an HTML snippet is free of disallowed tags or scripts (preventing XSS), `Safe_Str__Url` validates and sanitizes URLs (e.g. stripping `javascript:` exploits) [18], and file-related strings like `Safe_Str__File__Name` can automatically remove path traversal sequences (`"../"`) from filenames [19]. On the numeric side, `Safe_Int` and `Safe_UInt` enforce bounds and non-negativity. A `Safe_UInt__Port` is guaranteed to be between 0 and 65,535 – assigning a value outside that range throws an error [20]. Similarly, a `Safe_UInt__Percentage` would enforce 0–100%. The code snippet below illustrates these domain rules in action:

[21]

Code: Defining a server config with domain-specific safe types ensures invalid values cannot be assigned. In this example, `port` must be ≤65535 and `cpu_limit` ≤100, so any out-of-range assignment triggers a `ValueError` [20].

By using such types pervasively, **impossible states are made impossible to represent**: you cannot even create a `ServerConfig` with an invalid port or an over-100% CPU limit – the Type_Safe system won't allow it. This dramatically reduces the need for defensive checks scattered throughout the code (the class itself is the gatekeeper) and thus reduces the chances of developer oversight. In fact, the OSBot-Utils team reports that adopting Type_Safe can *reduce the overall testing burden by about 40%* [22], because many edge cases (like negative values, overly long strings, bad formats) are inherently handled by the type system rather than requiring separate test cases. Code becomes more **self-documenting** as well: when you see a function parameter of type `Safe_Str__Html`, it's explicitly clear that this function expects sanitized HTML – far clearer than a mere comment or naming convention, and enforced by the runtime checks [23].

Moreover, the Type_Safe system extends to collections, enabling **continuous validation even in compound data structures**. For example, OSBot-Utils provides a `Type_Safe__List` which acts like a Python list but tracks the expected type of its elements. If you have a `List[Safe_Str__ProductId]` and try to append a `None` or an integer to it, a `TypeError` will be raised immediately [24]. Contrast this with typical use of dataclasses or Pydantic models: those might validate a list of strings on creation, but once the object exists, someone could still call `order.items.append(12345)` and corrupt the data with no warning. Type_Safe prevents that: *"catches corruption at the source, prevents cascade failures"* by validating on every insertion [25]. Early catching of such errors vastly simplifies debugging (finding the bug at the exact operation, rather than many steps later when the bad data is finally used) – in one case study it reduced debugging time by 75% [25]. And from a security perspective, having safe types means that any data crossing a trust boundary (user input, external API, etc.) can be converted to a Safe type at the boundary. If the conversion succeeds, the data is clean and valid going forward; if it fails, the system rejects the input early. This provides **end-to-end data integrity** within the system, a crucial property for high-reliability and safety-critical software.

In summary, OSBot-Utils' runtime type safety brings the rigor of a strong type system and the protection of validation logic into a dynamic runtime. It plugs the holes left by static typing, especially in a dynamically typed language, by ensuring that *if the code runs, it runs on correct and safe data*. It also frees developers from writing boilerplate checks or duplicating validation logic in every function – the primitives handle it consistently. The resulting codebase is significantly more **robust to unexpected inputs and misuse**, which is a perfect complement to the exhaustive scenario coverage that PDD testing provides.

Synergy in Practice: Resilient and Consumer-Aligned Software

Individually, PDD and runtime Type_Safe primitives each enhance software quality; combined, they reinforce each other to create a development paradigm optimized for resilience. This approach has been adopted in Dinis Cruz's projects like **HTML Graph** (a service that converts and analyzes web content in graph form) and **MGraph-DB** (an open-source semantic graph database). In these complex, evolving systems, the PDD + Type_Safe combination enables rapid iteration with confidence in correctness.

Early Failure, Early Fix: Both PDD and Type_Safe promote surfacing failures as soon as possible. Under PDD, whenever a new edge case or requirement is discovered (often through a user bug report or a new feature request), a test is immediately written to capture it. Under Type_Safe, whenever a developer (or an automated process) introduces data of the wrong type or format, the code throws an error at the point of violation. Together, this means that the *first time* a potential failure enters the system, it is caught – either by a test that reproduces it or by a runtime check that guards against it. For example, in the HTML Graph environment, suppose a particular HTML snippet with nested malformed tags once caused the graph parser to crash. Instead of silently ignoring it or patching it without a test, the team would add a PDD test that *expects the old crash or error condition* (thus passing until the parser is fixed). At the same time, they might introduce a `Safe_Str__Html` type for HTML content inputs, so that any grossly malformed or script-injected HTML is sanitized upfront. The next time that scenario arises – whether in development or production – the system will either sanitize it or fail fast (and the test will ensure it's handled as expected). No issue remains hidden or delayed; everything fails early on controlled terms, which leads to quicker diagnosis and fixes. This significantly increases the **resilience** of the software: it behaves predictably under stress, and when something does go wrong, it's immediately apparent and documented.

Full Traceability and Documentation: PDD's practice of linking tests to real bugs/features and Type_Safe's use of semantically rich types both contribute to making the system more transparent and aligned with domain needs. Every test written in the PDD style is effectively a requirement or use-case captured in code. In MGraph-DB's test suite, for instance, one will find tests that correspond to specific graph behaviors or past integration bugs (e.g. a test ensuring that adding a node with an invalid ID format is handled gracefully might stem from a real incident). These tests act as living documentation of how the system should behave in all these corner cases. Likewise, the code itself, through the use of descriptive safe types, communicates its intent clearly – a function that takes a `PatientId` and a `DoctorId` (both subclasses of `Safe_Id`) cannot mix them up by accident [26], and a reader of the code immediately understands those parameters are distinct identifiers in the medical domain. In this way, there is **traceability from the business domain down to the code and tests**. Features and fixes are tied to tests (you can trace a user story or bug report to one or more test cases), and domain concepts are tied to types (you can trace a high-level concept like "Port number" or "HTML content" to a specific class enforcing its rules). This traceability is invaluable for collaborating in large teams and for onboarding new developers – it's always clear *why* a test exists (it names the scenario or bug) and *why* a type exists (it encapsulates certain assumptions or rules). As Dinis Cruz noted, *every bug or regression test represents a real-world use case encountered and understood* [27]. The development process therefore stays tightly aligned with **consumer needs and real usage patterns**. Instead of theoretical tests or types, we have tests derived from actual user scenarios and types reflecting actual business rules, which keeps the software focused on the consumer's perspective.

High Coverage and Fewer Gaps: A direct outcome of this approach is very high test coverage and fewer blind spots in quality. PDD virtually guarantees that for every feature implemented or bug fixed, test code accompanies it. Over time, the accumulation of tests results in broad coverage of the

codebase without teams having to chase an arbitrary percentage target. In Dinis Cruz's OSBot-Utils project, for example, the practice of writing a test for every bug (and every new capability) led to a comprehensive test suite protecting the Type_Safe functionality [28]. When he undertook a major refactoring of the Type_Safe list and dict classes (splitting one class into three), he did so with the confidence that the existing tests would flag any regression. Indeed, after the refactor, a suite of tests failed – specifically those tests designed to assert the previously buggy behaviors around dict type-checking [28]. This was *intentional and welcomed*: the failing tests proved that his changes had altered those behaviors, effectively pointing out exactly what needed to be addressed next. He then updated the implementation and tests accordingly, eventually reaching a state where all tests passed and the new design was confirmed to handle all the previously problematic cases. The **CI pipeline remained unbroken throughout** – at no point were other contributors blocked – because the interim steps were done in isolation. Once complete, the changes were pushed and the CI could run a fully passing suite across multiple environments [29]. This story highlights how high coverage combined with PDD methodology yields *fearless refactoring*. The team can undertake sweeping improvements or migrations (even leverage automated tools or AI for large-scale refactors) with the safety net of tests that will catch anything deviating from expected behavior. Developers gain the confidence to evolve the system's architecture or internals aggressively, knowing that if they miss something, a regression test will snap and alert them. This reduces technical debt and stagnation – codebases can stay healthy and adaptable over time, rather than becoming brittle due to fear of breaking things.

Consumer-Aligned Design: An often overlooked benefit of combining these practices is how it keeps development aligned with consumer expectations. The term "consumer-aligned software" implies building software that meets users' needs and handles the realities of how it's used in the field. PDD contributes here by ensuring that *every time a user finds a bug or an edge case, it gets turned into an automated test*. The software literally learns from each failure in the field, and the test suite grows to cover the way real consumers are interacting with it. Type_Safe primitives contribute by encoding the assumptions about input and output that make sense for the consumer. For instance, if a web API expects an `email` field, using a `Safe_Str__Email` type (hypothetically) would ensure that any client-provided email is validated – the system won't accept just any string, only one that matches the format of an email address. This means the software is less likely to accept garbage data or produce nonsensical results for the user. In HTML Graph's case, using `Safe_Str__Html` for content means the graphs generated will exclude potentially malicious content, aligning with end-user security expectations for any rendered output. In MGraph-DB, using safe types for node identifiers and relationship types means the database APIs won't allow operations that mix up IDs or violate the ontology schema – thus aligning with the user's expectation of consistency in the knowledge graph. In short, these mechanisms ensure that **the system's behavior stays aligned with valid use cases and fails fast on invalid ones**. Users experience a more reliable product: fewer unpredictable crashes, fewer silent data corruptions, and quicker turnaround on fixes for odd cases (since those become tests quickly).

Finally, the combination of PDD and runtime type safety fosters a culture of quality and continuous improvement. It encourages developers to proactively think about edge cases (since writing a PDD test requires understanding the bug condition deeply) and about data contracts (since choosing or creating a Safe type requires understanding the domain limits). Instead of viewing tests as a chore or types as boilerplate, the team sees them as essential tools that enable faster development and safer deployments. The payoff is code that is not only **robust** (able to withstand misuse and change) but also easier to maintain. New features can be added with high confidence because the existing web of tests and type checks will immediately indicate if something is incompatible. As one might say, *the system is always telling you when you're about to step out of line*. This frees the technical leadership to focus on delivering value and evolving the product, rather than firefighting regressions or production issues.

When failures do occur, they are usually caught in pre-production and come with a test case to reproduce them, making root cause analysis straightforward.

Conclusion

In conclusion, Pass-Driven Development and OSBot-Utils' Type_Safe runtime primitives form a powerful one-two punch for engineering high-quality, resilient, and consumer-aligned software in dynamic codebases. PDD ensures that the journey of the code – every feature added, every bug found – is recorded in an ever-growing suite of passing tests that double as documentation and safeguards. It flips the traditional testing strategy to maximize coverage without jeopardizing continuous integration, creating a living record of system behavior and an active shield against regressions [8] [27]. Complementing that, runtime Type_Safe primitives bring the kind of strictness and guarantee to a dynamic runtime that one typically only expects from statically-typed, formally verified systems. They eliminate broad classes of errors by making invalid states unrepresentable and by enforcing security and validation rules by default [21] [30]. Together, these approaches lead to codebases that surface failures early, are self-testing and self-documenting, and can adapt to change with confidence.

The experience in projects like HTML Graph and MGraph-DB attests to the effectiveness of this combination. These systems handle complex, evolving data (from parsing HTML to managing knowledge graphs) and have benefitted from PDD and Type_Safe practices by maintaining stability and clarity even as they grow. Developers can refactor boldly – switching out data structures, adding new analysis features – supported by a safety net of tests and runtime checks that immediately flag any deviation. The software remains aligned with user needs because every odd scenario users encounter becomes fuel for an improved test suite and stronger data contracts. By embracing Pass-Driven Development and runtime type-safe primitives, technical leaders can foster an engineering environment where quality is baked in continuously. The result is software that not only meets its requirements but does so **reliably and safely**, even as those requirements evolve. This strategy ultimately delivers value to end consumers through fewer bugs in production, quicker turnaround on fixes, and systems that behave predictably in the face of the unexpected – a hallmark of truly resilient software.

---

[1] [2] [12] [13] [14] [15] [16] [17] [20] [21] [22] [23] [24] [25] [26] [30] the-business-case-for-type_safe__why-runtime-type-protection-matters.md
https://github.com/owasp-sbot/OSBot-Utils/blob/b76809ea6173cf8990dc69e5afb8b930c7e077fa/docs/type_safe/comparisons/the-business-case-for-type_safe__why-runtime-type-protection-matters.md

[3] [4] [5] [6] [7] [8] [9] [27] [28] [29] Start with passing tests (tdd for bugs), now (in 2024) with GenAI support
https://www.linkedin.com/pulse/start-passing-tests-tdd-bugs-now-2024-genai-support-dinis-cruz-pxnde

[10] Implementing Type-Safe Schemas in Performance Toolkit | Dinis Cruz posted on the topic | LinkedIn
https://www.linkedin.com/posts/diniscruz_strict-type-implementation-protocol-activity-7415717680055513088-Xv3G

[11] GitHub - owasp-sbot/OSBot-Utils: Project with multiple Util classes (to streamline development)
https://github.com/owasp-sbot/OSBot-Utils?trk=article-ssr-frontend-pulse_little-text-block

[18] [19] safe_str__overview_and_architecture.md
https://github.com/owasp-sbot/OSBot-Utils/blob/b76809ea6173cf8990dc69e5afb8b930c7e077fa/docs/type_safe/primitives/safe_str/safe_str__overview_and_architecture.md