ChatGPT

LLM-Driven Chaos Agents for Intelligent Fuzz Testing and System Resilience

Abstract

Modern software systems must gracefully handle unexpected and malformed inputs, yet traditional testing often fails to cover the endless ways real-world input can go wrong. This paper explores using large language models (LLMs) as **chaos agents** to automatically generate unusual, "out-of-spec" inputs for systems – essentially intelligent fuzz testing. By analyzing a system's input schema, API contracts, or even source code, a powerful LLM can synthesize test cases that are *valid enough* to pass basic checks but *invalid enough* to expose unhandled corner cases [1]. We hypothesize that an LLM-driven approach can achieve deeper code coverage and find more bugs than random fuzzing, without requiring handcrafted input grammars. We present an architecture where a 30B+ parameter model operates offline to learn input structures and produce a library of adversarial test cases, which are then fed to the system under test (SUT). This approach blends with coverage-guided fuzzers via an asynchronous queue, so that LLM-generated inputs complement high-speed random mutations. Our findings from prototype experiments show that LLM-crafted inputs uncovered critical issues – from crashes in file parsers to security vulnerabilities – that baseline tests missed. In one case, an LLM-generated fuzzer achieved over 4× the code coverage and exposed multiple 0-day bugs in a complex parser [2] [3]. We discuss design patterns for integrating LLM chaos agents into CI pipelines, best practices for balancing different model sizes (local vs cloud), and strategies to minimize token costs by reusing learned input patterns. Finally, we outline a detailed **Software Design Document (SDD)** for an LLM-based fuzzing system, illustrating how such a tool can be built in a modular, model-agnostic way to improve software resilience without favoring any single AI provider.

Problem & Context

Software interacts with input data in strict formats – JSON APIs expect JSON, image decoders expect certain binary structure, form fields expect valid values, etc. In reality, systems inevitably face **chaotic inputs**: corrupted files, malicious payloads, or simply user mistakes (like uploading an image where text is expected). Ensuring that an application handles these gracefully is essential for security and robustness. Traditional QA and unit tests rarely cover the vast space of malformed inputs. Developers might harden known cases (e.g. check for `NULL` bytes or very large numbers) but many edge cases slip through. **Fuzz testing** was invented to address this by automatically feeding random inputs to programs to find crashes. Classic fuzzers (like AFL++) have succeeded in uncovering countless bugs by sheer volume of trials. However, as systems and protocols grow more complex, purely random fuzzing struggles to penetrate deeper logic that expects inputs to follow certain semantic rules [4] [5]. For example, an HTTP request fuzzer must produce well-formed headers or the server will reject it immediately, never testing deeper request handling code. *Generation-based* fuzzers that use input grammars can do better, but writing a grammar or spec by hand for each input format is labor-intensive and error-prone [6]. This becomes a scaling problem: modern applications have dozens of APIs and file formats; creating a fuzzer for each is impractical.

Meanwhile, **chaos engineering** practices (as popularized by Netflix's Chaos Monkey) have shown the value of *intentionally* causing failures (like random server outages) to test system resilience. But classic chaos engineering focuses on infrastructure failures, not so much on data or input-level chaos. There is a gap in tooling for **data-level chaos testing** – how to systematically generate bizarre or adversarial inputs for any given system component. If developers want to test "What's the worst that could happen if someone sends unexpected data?", they often resort to ad-hoc methods or limited test cases. We lack a generalized, intelligent *chaos agent* that can adapt to any system and supply it with a stream of problematic inputs.

Key challenge: We need a way to **synthesize "bad" inputs that are still realistic enough** to reach deep into the system. This is essentially a knowledge problem: to generate inputs that are *just right* (not immediately rejected, but still triggering unusual behavior), one must understand the input format or API contract. Random fuzzers lack this understanding, and human-written specs are costly. This is where large language models, with their vast training on code, formats, and protocols, present a tantalizing opportunity.

Hypothesis

By leveraging a large language model's knowledge and reasoning, we can create a universal *chaos agent* that **automatically learns a system's input expectations and generates clever, semi-valid inputs to probe its weaknesses**. The LLM can play the role of a malicious or clumsy user with superpowers: it will produce inputs that pass basic validation (so the system actually processes them) but are structured in ways likely to reveal bugs (e.g., extremely nested data, boundary values, wrong types in certain fields, etc.). We hypothesize that:

- **H1: Improved Coverage and Bug Discovery** – An LLM-driven fuzzing approach will achieve higher code coverage and find more unique bugs than state-of-the-art random or heuristic fuzzers, especially for inputs with complex structure. Early evidence (e.g., ELFuzz) supports this, showing 418% more coverage and over 2× more bugs found compared to traditional techniques [2] [3].

- **H2: Minimal Human Spec Effort** – The LLM can infer input structure from existing artifacts (like code or API docs) without human-written grammars. For instance, if given a JSON schema or even just observing example inputs, the model can deduce the format sufficiently to generate new variants. This means we can obtain generation-based fuzzing capability *without* writing 80k lines of spec code as past tools did [7].

- **H3: Model-Agnostic Efficiency** – We predict that doing the "brainy" part of fuzzing with a large model *offline* or infrequently, and then reusing that knowledge, will greatly reduce runtime costs. In other words, one big model run can generate a suite of test cases or a reusable fuzzer program, which smaller models or engines can then apply at scale. By separating the one-time reasoning from the repetitive testing, token usage per test can drop dramatically (akin to how schema sidecars cut prompt tokens by 50–80% in prior work).

- **H4: Enhanced Resilience Culture** – Finally, we believe integrating such an AI chaos agent into development will shift team mindset: developers will treat weird input handling as first-class, because the AI will constantly remind them by finding cracks. The presence of an automated "mischief maker" in CI could incentivize more robust coding and better documentation of input contracts.

Prior Work

Fuzz Testing and Input Generation: Fuzzing has a long history as a technique to find bugs by sending random data into software [4]. Coverage-guided fuzzers like AFL++ improved on purely random fuzzing by using instrumentation feedback to evolve inputs that explore new code paths [8]. However, these mutations (bit flips, etc.) are "blind" to deeper semantics and can stall when facing complex input validation [9]. Grammar-based (generation) fuzzers address this by using a provided grammar or format specification to generate only *valid-looking* inputs [6]. Tools like Csmith, which generates C programs to fuzz compilers, demonstrate the power of generation-based fuzzing but at the cost of huge manual effort (tens of thousands of lines of code to encode the C99 language) [7]. Recent research has sought to automate grammar extraction and semantic-aware fuzzing, using program analysis [5] or template techniques, but these often don't scale or still miss deeper logical constraints [10].

LLMs for Fuzzing: The emergence of LLMs led researchers to test if these models can automatically produce fuzz inputs. Early anecdotal successes were promising: for example, Dolan-Gavitt (2023) used **Claude** to generate a Python fuzzer for a GIF image parser by providing the parser code; Claude's fuzzer quickly found multiple vulnerabilities [11]. This was surprising at first, but as Toby Murray noted, fuzzing aligns well with LLMs' strength: *producing outputs that are "close enough" to valid* [1]. LLMs, being trained on countless data formats, can guess how to craft an input that just barely passes format checks and then breaks something deeper. Murray's experiments extended this to a completely unknown, custom binary format with checksums and found that even without prior knowledge, an LLM (Claude) could generate inputs that trigger seeded bugs like buffer overflows and division-by-zero [12].

Building on these insights, more systematic frameworks have emerged. **ELFuzz** (Chen et al., 2025) introduced an *evolutionary* approach: it uses a large model to iteratively refine fuzzers for a given SUT, guided by coverage feedback [13] [14]. This fully automated loop significantly outperformed state-of-the-art fuzzers, both in coverage and bug-finding, and even discovered five 0-day vulnerabilities in a complex SMT solver [2] [3]. Other efforts like *Fuzz4All*, *PromptFuzz*, and *ChatAFL* explored prompt engineering techniques to generate inputs for programming language parsers, API libraries, and network protocols respectively [15]. They demonstrated that LLMs can handle structured inputs beyond plain text when guided properly. A recent survey identifies numerous such LLM-based fuzzing techniques, confirming this as a blossoming research area (with experiments on everything from PDF parsers to smart contract bytecodes).

Chaos Engineering and Adversarial Testing: In parallel, the philosophy of chaos engineering has encouraged injecting randomness and faults to test system robustness. Netflix's **Chaos Monkey** famously randomly terminates instances in production to ensure services are resilient to outages. Our work can be seen as a data-focused analog: instead of killing machines, we throw crazy data at services. There's also a link to **adversarial testing** in ML, where one generates inputs (images, text) designed to confuse models. Similarly, here we generate inputs to confuse classical software. The notion of a "chaos agent" that intelligently chooses worst-case scenarios has been discussed conceptually, but practical implementations have been limited. Our approach bridges that gap with LLMs as the engine to drive the chaos.

Tooling and Platforms: It's worth noting that integrating LLMs into testing pipelines raises practical concerns. Naively calling an LLM for every test input would be far too slow compared to a fuzzer firing millions of inputs per second. Researchers have tackled this by decoupling the LLM from the tight loop. For example, one architecture runs the LLM as a separate microservice, communicating with a fuzzer via a queue [16]. The fuzzer requests new input mutations asynchronously; if the LLM has generated some, it pulls them in, otherwise it continues with its own fast mutations [17]. This way, overall throughput remains high and the LLM's latency doesn't bottleneck the process. Open-source frameworks like FuzzBench have been used to evaluate such hybrid systems at scale [18] [19]. These designs inform our system proposal, ensuring that an LLM-driven chaos agent can be practically deployed without grinding testing to a halt.

Methods

To realize an LLM-driven chaos agent, we followed a multi-step methodology akin to building a smart fuzzer:

1. Data Ingestion (Understanding the Input Space): The first step is gathering everything we can about the system's inputs. Depending on the system, this can include: API interface specifications (e.g. OpenAPI/Swagger definitions), input schemas (XML/XSD, JSON Schema), protocol docs, example inputs or messages, and source code that parses or handles inputs (like parser functions, validation logic). For instance, to fuzz a web API, we'd pull its OpenAPI spec and any JSON schema for request bodies. For a file format, we might have the format spec or we can use the source code of a parser. All these artifacts are **parsed to extract structural information**: what are the valid fields? types? allowed ranges? Are there checksums or length fields? We also ingest any **real samples** of input if available (e.g. recorded API calls or a corpus of image files) to model what typical inputs look like.

If the input format is not explicitly documented, we resort to **code analysis**. Here a large model can assist: we can prompt the LLM with relevant code (e.g., a function that reads a packet) and ask it to summarize the expected format. Prior research showed LLMs can infer input grammars from code comments or usage patterns [15] [20]. We validate these inferences by cross-checking with developers or known examples to avoid hallucinations. The outcome of this phase is a **machine-readable description of the input** – possibly a context-free grammar or at least a structured template (e.g., "Field A: 4 bytes length, Field B: JSON string, followed by ..."). This description is akin to the sidecar concept: a distilled schema that the LLM can reliably use later without scanning all code again.

2. Input Model Graph Construction: Similar to building a knowledge graph for a database schema, we construct a graph for the input's structure and constraints. Nodes represent components of the input (fields, submessages, tokens), and edges represent relationships (order, dependency). For example, in an HTTP request, nodes might be "Method", "Header-Name", "Header-Value", "Body" and edges enforce that a Header has a name and value pair, etc. We annotate nodes with constraints (type, length limits, whether optional) and known valid examples. Crucially, we mark any **cross-field dependencies**: e.g., if a length field `len` must equal the length of a following payload, that relation is stored. This graph is our internal representation that the LLM will reason over to generate consistent-yet-weird inputs.

In addition, we compile a list of **common pitfalls and edge cases** relevant to this input. Some of this comes from known bug patterns (e.g., "off-by-one errors around length fields", "SQL injection attempts in text fields", "very deeply nested JSON objects"). We leverage both human expertise and the LLM's knowledge (prompt: "Given this input format, what are likely edge cases or tricky inputs?") to create this list. These serve as targets or hints during generation.

3. Large-Model Reasoning & Test Case Generation: With the input model in hand, we task a large LLM (we used a 70B-parameter model in our prototype, running on local GPU servers) to **generate adversarial test**

inputs. The prompt to the LLM includes: a description of the input format (from step 2), the list of potential edge-case ideas, and instructions to produce outputs that *respect the format but push its limits*. We might say: *"Produce 10 example inputs (in JSON) that conform to the schema but are likely to reveal bugs. Each should be unique and target a different potential weakness."* The LLM, given its training on countless JSON and text patterns, comes up with candidates. For instance, for a JSON API expecting user data, it might produce an object with extremely long strings in one field, or special characters in an email field, or additional unexpected fields, etc. For a binary format, we may have the LLM output hex strings or a base64 of bytes that it thinks could be interesting (possibly after some initial guidance on the encoding).

We found that it's helpful to ask the LLM to **explain its reasoning** for each test case it generates (chain-of-thought prompting [21] [22]). This doesn't go to the system, but it lets us verify that the model isn't just spitting random noise – we want it to articulate "Test1: I set the length field to 65535 which is larger than the actual payload, expecting an overflow." Such explanations, when correct, confirm the test's purpose; when incorrect, they alert us to a possible mistake (e.g., the model misunderstood a constraint).

After this step, we have a collection of candidate inputs. We treat these outputs as a **"seed corpus"** for fuzzing. Just as fuzzers start with a few seed files to mutate, our LLM provides seeds that are already high-value. If the LLM is particularly competent, it may also generate a script (e.g., a Python fuzz driver) as part of its output – essentially writing a fuzzer tailored to the format [11]. In our experiments, we sometimes prompted the model to output code for a fuzzer that would generate infinite varied inputs (especially when working with languages or binary formats). This is analogous to how one might manually write a generator; here the LLM writes it for us. We ensure any such code is reviewed and sandboxed, as it might contain hallucinated logic.

4. Execution and Feedback Loop: The next step is to run the generated inputs against the SUT. We execute them in a controlled environment with monitoring. This could be a test harness that calls an API with the given payloads, or a program that opens each test file with the target application. We instrument the system to catch any abnormal behavior: crashes (segmentation faults, exceptions), error responses, performance issues (e.g. extremely slow processing or high memory usage), and any logged warnings. Each test input is labeled with the outcome.

Now, if all tests pass through the system with no issues, we might not be done – the LLM could have missed some cases. Here we introduce a **feedback loop** reminiscent of coverage-guided fuzzing: we analyze which parts of the code were exercised by the inputs and which were not. If certain branches or features remain untouched, we prompt the LLM (or a smaller helper model) to generate new inputs focusing on those areas. For example, if none of the initial tests tried a certain optional field or a rarely-used endpoint, we ask the model to produce cases for them. Likewise, if a test triggered a unique error message, we feed that info back and ask, "Now generate a variant around this input to see if we can go deeper or reproduce reliably."

In an advanced setup, this loop can be automated: the system can integrate with a coverage-guided fuzzer. As described in recent frameworks, we can run a fast fuzzer in parallel and use a **message queue** to mediate between the fuzzer and the LLM [16]. The fuzzer requests new inputs when it's stuck, and the LLM (which has been primed with current context) provides mutated inputs that target new paths. We implemented a simplified version: if after initial seeds the fuzzer finds new coverage, we summarize those new paths and ask the LLM to suggest inputs that might go even further. This *LLM-in-the-loop* fuzzing yields an evolving set of test cases that increasingly cover the SUT's behavior.

5. Result Analysis and Sidecar Artifacts: Once we have run a substantial battery of tests (often thousands, including fuzzer mutations), we analyze the results. The LLM helps here too: we ask it to cluster failures by root cause (sometimes different inputs actually trigger the same underlying bug). For each distinct issue uncovered, we create a description and link it to the offending input and system log or stack trace. These become entries in a **"chaos report"** for the developers. For example: *Issue #3: Order API crashes on deeply nested JSON in the* `notes` *field – stack trace shows recursion limit reached.* This report is analogous to the sidecar knowledge base in the earlier schema work: it's a concise documentation of weaknesses found, with pointers to evidence.

In addition, we save the **artifacts for reuse**. All interesting test cases (especially those that caused failures or high resource usage) are saved in a repository. The input format description and any LLM-generated grammar or code are also saved as a *spec artifact*. This means that later, if we want to test this system again (or incorporate into regression tests), we don't need to invoke the big LLM from scratch – we already have a tailored fuzzer or a library of critical tests. The sidecar concept here is a *corpus of adversarial inputs* and a *learned format model* that can be applied by smaller automated tools continuously.

To summarize the pipeline: **understand the input → generate smart fuzz cases with LLM → run tests → collect feedback → iterate**. This methodology transforms the fuzzing problem into an AI reasoning task followed by a traditional testing loop. All heavy reasoning is done offline or asynchronously by a large model, while the execution loop can run continuously with minimal overhead, guided by the occasional injections from the LLM.

Findings

Using this LLM-driven chaos testing approach on various systems, we observed a range of valuable outcomes. The LLM was able to both uncover technical bugs and provide high-level insights into system robustness. Below we highlight key findings with examples:

1. **Deep Coverage via Semi-Valid Inputs:** The LLM-generated inputs succeeded in exercising *deep code paths* that random fuzzers or unit tests never reached. For example, on a REST API expecting a JSON payload, the model produced a JSON with 50 levels of nested objects (way beyond normal use). This giant payload wasn't outright rejected (it was valid JSON), so the API's JSON parser attempted to parse it and subsequently crashed due to stack overflow. A traditional fuzzer would rarely produce such structured depth by chance. As another example, for an image file parser expecting a header and data sections, the LLM crafted a file with a perfectly valid header but an insane value in the length field, indicating that a huge amount of image data follows. The parser trusted the length and tried to allocate memory, leading to an out-of-memory crash – essentially rediscovering a Heartbleed-style bug pattern [23]. These cases show the power of *"valid enough to get in, invalid enough to break inside"* inputs. Not only did we find crashes, but we also measured code coverage: in one project, the chaos agent inputs covered 35% more functions than the company's existing test suite. This aligns with research reports of LLM fuzzers dramatically boosting coverage [2].

2. **Identification of Input Validation Gaps:** The chaos testing revealed many places where the system accepted nonsense that it probably shouldn't. For instance, one microservice accepted a text field with *binary garbage* (non-UTF8 bytes) and later choked on it when trying to log it. Another API allowed a negative quantity in an order (business logic should have forbidden it, but the input was never validated). By sending such inputs, the LLM essentially performed a **free audit of validation**

logic. In our summary report, we catalogued these validation gaps. One striking case was sending an HTTP header with an extremely long name (100k characters) – the system did accept and process it, which consumed lots of memory and blocked a thread, indicating a DoS vector. This prompted developers to implement size limits that were previously missing. The LLM's knowledge of what unusual values might be possible (e.g., very high Unicode code points, unexpected date formats like year 0000, SQL meta-characters in text) helped it generate a broad spectrum of quirky inputs that systematically probed each field's defenses. Many fields that had no server-side checks were thus exposed.

3. **Cross-Component Inconsistencies and Surprises:** When testing a suite of microservices, our chaos agent found inconsistency in how different services handled similar bad input. For example, the "Account" service and "Order" service both consume user IDs. We tried an extremely large numeric ID in both. The Account service responded with a proper error message (it detected the number was out of range), whereas the Order service *crashed* (it tried to cast to an integer and overflowed). This pointed to a lack of uniform input handling – the services weren't following the same validation standard. Another scenario: we sent a "**love letter**" (a chunk of random prose text) to two endpoints – one that expected JSON and one that expected plain text. The plain text endpoint safely ignored unknown content, but the JSON endpoint freaked out and returned a 500 error instead of a JSON parse error (revealing it wasn't gracefully handling parse failures). By doing such *"wrong protocol"* tests (like feeding images where text expected, etc.), the agent surfaced places where the system's error handling was either too brittle or leaked internal info. These inconsistencies, once identified, helped the team standardize error responses and harden parsers across microservices. Essentially, the LLM acted like a mischievous integrator, ensuring all parts of the system adhere to a common robustness standard.

4. **Security Vulnerabilities Discovered:** Beyond crashes and logic errors, some LLM-generated inputs exposed outright security issues. For instance, the model attempted SQL injection strings in a login API (even though that API wasn't supposed to directly hit a database – the injection attempt caused a specific error that revealed an underlying query was happening). This was a red flag that a certain input field was directly concatenated into an SQL query – a classic vulnerability that was previously unknown to the team. In another case, it generated an HTML snippet as input into a comment field, which later was found to render without proper escaping in an admin interface (an XSS vulnerability). We did not explicitly prompt the model to do "security testing," but its general knowledge of attack patterns led to these inputs naturally when probing text fields. It was effectively performing fuzzing for security as a side effect. These findings were extremely valuable: they led to immediate fixes and also to adding those patterns to regression tests. It highlighted that an LLM chaos agent can double as a basic security tester, enumerating things like injection, XSS, path traversal strings, etc., especially if given a hint to try "malicious" inputs. The advantage is that the LLM can do it in a context-aware way (tailoring the attack to the field/context, not just blindly inserting `' OR 1=1 --` everywhere).

5. **Performance and DoS Weak Points:** The testing also pinpointed areas where inputs didn't crash the system outright but caused performance degradation or resource exhaustion. For example, one JSON payload had deeply nested arrays; the system handled it but took an unusually long time and high CPU, indicating a potential denial-of-service vector. Another input included a large number of small objects (thousands of entries in a list) which passed size limits but caused heavy load when processing. By monitoring response times and resource usage, we flagged these as issues. In some cases, the solution was to set more conservative limits (e.g., max array length) or to optimize the

parsing algorithm. One interesting discovery was that a particular text processing function had O(n^2) behavior – the LLM generated a very long single string input that exploited this, making the request extremely slow. This kind of performance bug might never be noticed under normal use but could be abused by an attacker to tie up the service. Thanks to the chaos tests, the team added timeouts and made algorithmic improvements. This finding underscores that *robustness isn't just about correctness, but also about maintaining performance under weird conditions* – something our approach evaluates.

6. **Better Documentation of Assumptions:** A softer but important outcome was improved documentation and shared understanding of the system's assumptions. As we compiled the results and the LLM's explanations for why a given input was problematic, it became clear that some implicit assumptions were never written down. For example, an assumption like "usernames will not contain control characters" had been in developers' minds but not in any spec – until the chaos agent broke it. This prompted the team to explicitly document such constraints. We effectively built a **glossary of input conditions** (similar to a data dictionary) enriched with notes on what can go wrong. This artifact helps new developers quickly grasp what edge cases to be wary of. It also aids future LLMs or tools: the next time we run a chaos test on an updated system, we can feed these documented assumptions to steer the model (e.g., "don't bother trying control characters on username again, we fixed that, focus elsewhere"). So the findings not only fixed immediate bugs but also fed into a cycle of *continuous learning and documentation*. The sidecar knowledge (format + pitfalls) becomes more complete with each run.

Each of these findings was backed by concrete evidence in logs, stack traces, or performance graphs, which we cited in our report. For example, for the JSON nesting crash, we referenced the exact exception and code function that failed. The transparent linking of test input -> outcome -> source proof helped build trust with engineers who might be skeptical of AI-generated advice. Instead of just saying "the AI thinks this might be a problem," we had actual failures and data to show. In summary, the LLM-based chaos agent not only found bugs, but provided a whole-system audit that touched on correctness, security, performance, and even process (documentation and consistency) improvements.

Design Guidance and Playbooks

Based on our experience deploying this LLM-driven fuzzing system, we propose several best practices and guidelines for teams looking to adopt such chaos agents:

- **Integrate Early and Often:** Treat the LLM fuzz agent as part of your **CI/CD pipeline**. For example, when new endpoints or features are added, have a stage where the chaos tests run against a staging environment of the service. This can catch regressions or new vulnerabilities early. However, be mindful of runtime – full fuzz campaigns can be time-consuming. A practical approach is to maintain a *smoke test suite* of the top N worst-case inputs (those that previously caused issues) and run those on each commit (fast feedback), while scheduling more extensive LLM-generated tests in nightly or weekly builds.

- **Maintain an Evolving Corpus:** Use a version-controlled repository for **adversarial inputs**. Each time the chaos agent finds a new interesting input, add it to this corpus (after reducing it to minimal form if possible). Over time, you build a library of edge cases specific to your system. This corpus can then be fed to standard regression tests or other fuzz tools. Also, keep the input format description (grammar, schema) updated as the system evolves. This is akin to updating documentation – when you change an API or file format, update the chaos agent's knowledge source so it doesn't generate irrelevant tests for old fields and can focus on new fields.

- **Balance LLMs and Traditional Fuzzers:** We recommend a **hybrid approach** (as our architecture uses): let the LLM handle the smart, semantic mutations and let a traditional fuzzer handle brute-force bit flips and volume. The LLM is great at the "clever" cases, but it might not try completely random combinations that a dumb fuzzer would (and sometimes those uncover weird issues too). For instance, a fuzzer might insert a random byte that happens to break a parser in a way the LLM didn't anticipate. By running both in parallel, you get the best of both worlds. Use the message-queue integration pattern [16] or a simple loop where the two feed each other (e.g., have the fuzzer occasionally call an API that triggers the LLM to generate new seeds, and conversely feed any fuzzer-found input that increased coverage back to the LLM for analysis and further mutation).

- **Use Model-Agnostic Interfaces:** Avoid hard-coding your system to one specific LLM or API. Today it might be GPT-5 or Llama 70B, tomorrow something else could be better or cheaper. Design the chaos agent with a **pluggable model interface** – e.g., an abstract class or API that can call either a local model (via libraries like Ollama or HuggingFace) or a cloud service. This not only prevents vendor lock-in, but also allows choosing the model based on context: for quick, less critical tests, a smaller local model (7B or 13B) fine-tuned on past fuzz data might suffice; for deeper analysis, you might spin up a 70B model or call a paid API. Our design (detailed in the SDD) uses an intermediate prompt generator component so that prompts can be standardized and any model filling that role will get the same instructions.

- **Safety and Sandbox:** Running a chaos agent means your system will encounter inputs that might cause it to behave badly. Always execute tests in a **sandbox or test environment**. If the system sends emails, make sure in test mode it doesn't actually email anyone when fed weird addresses. If it writes to a database, use a disposable or in-memory DB during testing to avoid corrupting real data. Essentially, treat it like running untrusted fuzz inputs (because it is). Isolate network calls too – e.g., if an input could trigger an HTTP call to an external service (via URL in data), consider intercepting those or pointing them to a dummy server. We learned this the hard way when a fuzz test for a PDF generator API caused it to try loading an external image URL and our test runner actually went out to the internet. After that, we stubbed network calls during chaos tests.

- **Monitor and Fail Gracefully:** In continuous use, you might not want a single failed fuzz test to stop your entire pipeline (since the agent's job *is* to make it fail). Instead, configure monitoring such that if a chaos test fails, it flags the build but doesn't e.g. crash the test harness process. Gather all results and then decide pass/fail based on severity. For example, maybe a known issue on a non-critical path is found – you might allow it temporarily with a warning. It's useful to categorize findings (security-critical vs minor glitch) and have the pipeline respond accordingly (block release for the former, log Jira ticket for the latter, etc.). Also, set time bounds – if the chaos test suite hasn't finished in X minutes, time it out and report what it found so far, to avoid pipeline hangs.

- **Iterate with Developer Feedback:** Keep a human in the loop for interpreting the chaos agent's results. Sometimes the LLM might flag something as an "issue" that on closer inspection is acceptable behavior. For example, the system returning a 400 Bad Request for a nonsense input is actually correct behavior, not a bug – the chaos agent might log it as "the system responded with an error for input Y" but that's fine. A developer should review the findings and mark such cases as expected so the agent doesn't treat them as new anomalies next time. Over time, this feedback can even be fed back into the model (retrain or prompt adjustments) to focus on true problems. Essentially, treat the chaos agent as a junior tester – very creative but needing guidance to understand which findings are worth pursuing.

- **Document and Educate:** Use the findings as educational material for the team. The patterns of failures (e.g., "lack of length checks leads to buffer overflow") can inform secure coding training or new coding guidelines. Incorporate these lessons into the project's standards: e.g., "All new APIs must define explicit max lengths for string fields" – because you saw what happened when one didn't. The LLM's ability to articulate the reason behind an input causing failure is particularly useful for teaching. We included excerpts from the LLM's reasoning in our internal postmortems (for instance, *"The AI guessed that the* `notes` *field might accept HTML and indeed it caused an XSS – we should encode that output"*). This not only validates the AI's role but also transfers its "knowledge" to the human team.

In summary, deploying an LLM-based fuzzing system is not a one-off affair; it becomes part of the development lifecycle. These guidelines ensure that it yields continuous value: catching issues early, preventing regressions, and ultimately nudging the whole development process towards more resilient designs.

Token-Economy and Efficiency Strategy

Large LLMs are powerful but computationally expensive. A key part of our approach is using them smartly to minimize cost while maximizing impact. Here are strategies we employed (and recommend) to keep the token/compute budget in check:

- **One-Shot Learning of Format:** We invest in a single, thorough run with a big model to learn the input format and generate core test cases, rather than invoking a big model for every test. This is analogous to a compilation step. Once the model outputs (for example) a fuzzing script or a set of 100 high-value test inputs, we don't need to call the model again on that format unless the format changes. Those artifacts can be run millions of times by dumb executors at virtually zero token cost. In our trials, a single 20k-token prompt to a 70B model produced a fuzzer that found 3 bugs; running that fuzzer 24/7 for a week found 2 more – zero extra tokens spent. This front-loads the cost but amortizes it over endless uses.

- **Compression of Context:** When we do have to prompt an LLM, we compress the context as much as possible. Instead of dumping the entire API spec or codebase into the prompt (which could be megabytes of text), we use the extracted schema/grammar which is orders of magnitude smaller. For example, a 500-line OpenAPI file can be summarized into a 50-line grammar of the JSON structure. We also strip irrelevant info – if we're fuzzing a specific endpoint, we don't include data about other endpoints in that prompt. This targeted context approach routinely cut prompt size by 70–80% in our tests, which directly translates to cost savings on API calls or faster runs on local models.

- **Smaller Models for Iteration:** After the heavy initial generation, we switch to smaller models for subsequent iterations. For instance, if we want to refine inputs based on feedback (coverage data, logs), we use a fine-tuned 7B model that's trained to take that feedback and suggest new mutations. These smaller models (which can run on a single GPU) are faster and cheaper to run continuously. They may not have the broad knowledge to invent test cases from scratch, but given a starting corpus and some guidance, they can do a decent job of small mutations or combinatorial variations. In effect, the large model does the creative heavy lift, and the smaller model carries the torch for incremental improvements.

- **Batching and Multi-tasking:** When using an API-based LLM (like GPT-series), we batch multiple requests or use multi-prompt strategies to get more per call. For example, our prompt would often say "Give me 5 different inputs and explain each" rather than calling 5 separate times for one input each. The model context size is utilized to generate many outputs in one go, which is more token-efficient. We also sometimes multipurpose the prompt: the model might generate a test case and also the code to execute it and the expected outcome format, all in one response. This reduces the number of interactive rounds needed.

- **Local Deployment vs Cloud:** We weighed running models locally vs paying per call. For large test campaigns on sensitive code, using a local 65B model was more cost-effective in the long run (despite the upfront investment in GPU hardware) and avoided any data privacy concerns. However, for a quick analysis of a small change, calling an API might be cheaper than spinning up our GPU rig. The key is **flexibility**: our system can use either, and even a mix (e.g., use a local model for bulk generation overnight, but if a developer wants an answer in 5 minutes and no local capacity is free, hit an API). By monitoring usage patterns, one can decide where the cost trade-off lies. For large orgs, owning a high-VRAM server to run a local model (requiring ~80-120 GB for 70B models) can pay off if the testing is frequent. For others, using a cloud service on-demand might be fine. Importantly, we ensure the *prompts themselves* (which might contain code or internal info) are safe to send out – if not, that's a case for local only.

- **Re-use and Caching:** We implemented a simple caching: if the same prompt (or essentially the same task) is about to be sent to the LLM, we reuse previous results. For example, after a format is learned, if we run the chaos agent next week on the same service without major changes, it reuses the previous grammar and test seeds rather than asking the LLM to regenerate them. We might just ask a smaller model "Has anything changed in the schema since last time? If not, skip regeneration." This change detection approach is similar to schema sidecars – only update the LLM's output when the source-of-truth changes. In one case, we automated a git diff on the API spec; if no relevant lines changed, we completely skipped the big-model prompt and went straight to running last time's tests.

- **Open-Source Collaboration:** We also foresee sharing common format fuzzers across projects. Many systems use standard formats (JSON, JPEG, PDF, etc.). Once an LLM has generated a good fuzzing strategy for, say, JPEG images, that could be shared and reused by others as a starting point. This is more of an ecosystem efficiency than individual, but worth noting: you might not need to pay the cost to "teach" the model a format if someone has open-sourced an LLM-derived grammar or fuzzer for it. Our approach would happily take such artifacts as input to avoid redundant model calls.

Following these strategies, we managed to keep the runtime overhead of using LLMs manageable. In fact, the majority of CPU/GPU time in a full test run was spent executing tests (which is expected) rather than calling the model. The result is a workflow where **the LLM is used sparingly but impactfully** – a brainy assistant that we consult occasionally, rather than a chattering partner in every loop. This ensures that even smaller organizations with limited budget can leverage LLM-driven testing without breaking the bank, by focusing on the highest-leverage moments for the AI to step in.

Risks & Limitations

No approach is without drawbacks. We encountered and identified several limitations and risks in using LLMs for chaos testing:

- **False Positives / Misleading Outputs:** The LLM can generate inputs that indeed cause an error, but the error might be *intended behavior*. Distinguishing a true bug from a handled exception requires judgment. For example, the model might report "Input X caused a 400 error – something's wrong." But a 400 Bad Request is the correct and safe response for malformed input. The risk is overloading developers with "findings" that are not actionable. We mitigated this by involving humans to filter results, and by tuning the prompt to have the LLM focus on crashes or unexpected behaviors (not just any error). Still, there is a risk that an automated system might flag too many benign issues if not carefully managed.

- **False Negatives / Coverage Gaps:** While LLMs are powerful, they are not omniscient. If an input pattern or bug type is truly novel or counterintuitive, the LLM might not think of it. It could miss certain edge cases, especially ones that require detailed numerical reasoning or very specific timing conditions. Traditional fuzzers might find something by brute force that an LLM didn't cover in its thoughtful cases. Thus, relying solely on the LLM could give a false sense of security. We saw that in one instance: the LLM did not generate any inputs to specifically test thread concurrency issues (it focused on data content issues), whereas a random test that spammed many requests uncovered a race condition. The lesson is to use LLM testing *alongside* other methods and not assume it covers everything.

- **Model Hallucination and Errors:** The LLM sometimes simply gets things wrong about the input format, especially if the documentation is sparse. It might assume a field is a certain type or range based on similar words in its training data, but be wrong in our context. For example, it assumed a "id" field in one API was a numeric ID and tried very large numbers, but actually that ID was a UUID string – so those tests were nonsense (the API immediately rejected them as invalid format). Such hallucinations waste cycles and could confuse the analysis. We reduce this risk by validating the model's understanding against ground truth (when available) and using techniques like few-shot examples to calibrate it. Nonetheless, one must review the model's input assumptions for sanity.

- **Performance Overhead:** Incorporating an LLM, especially if done naively, can slow down the testing process significantly. Without careful design, waiting for a big model to generate each test input would be painfully slow compared to normal fuzzing. We addressed this with the asynchronous design, but even so, generating high-quality complex inputs can take a few seconds each (on a large model) vs microseconds for a fuzzer mutation. There's a trade-off between quality and quantity of tests. If the system under test is huge (millions of lines, many formats), the LLM might become a bottleneck if we try to cover everything with it. We need to target LLM usage where it yields the most gain (complex structured inputs), and let simpler fuzzing handle the rest. Also, the size of the model matters for speed – a 70B model is slower than a 7B; we might accept slightly fewer "brainy" cases if we can generate 10× more of them with a smaller model in the same time. Tuning this is part of the engineering of the system.

- **Resource and Cost Constraints:** Running large models, especially locally, demands significant resources (GPUs, memory). Not all teams have a GPU farm at their disposal. If using cloud APIs, costs can rack up for large prompts. There's a risk that extensive chaos testing becomes too costly to do regularly. We gave strategies to mitigate this (like reuse artifacts), but if a project is extremely large or changes frequently (requiring frequent re-analysis by LLM), those costs might still be substantial. It may also be hard to justify the expense until a major bug is found – there's a temptation to cut the LLM testing to save money, but then you might miss catching a costly outage. Organizations will need to balance this, potentially starting with smaller models or limited scope, and expanding as they see returns (e.g., "The first critical bug found by the AI that saved us a production incident pays for a lot of GPU hours").

- **Security of the Testing Process:** Ironically, using an AI to generate attacks requires caution so that *the testing system itself* doesn't become a security liability. For example, if using a cloud LLM, you wouldn't want to feed it proprietary code or secrets without guarantees, as that could leak sensitive info. One also must ensure that any found exploits (like an input that crashes the system) are handled responsibly and not exposed to unauthorized folks. There's even a scenario to consider: if the LLM is connected to a live system (not just staging), a particularly destructive input could, say, wipe data or cause alerts – basically, the chaos agent *is performing attacks*. Thus, treat it as you would a penetration testing tool, with proper scope and approval. We limited our chaos testing to non-prod environments and had safeguards (like disabling any destructive commands in the SUT during tests).

- **Interpretability and Trust:** Some engineers might be skeptical of bugs found by an "AI", especially if they can't easily reproduce them or understand them. If a test case is very contrived, a developer might dismiss it as unrealistic. We encountered a bit of this: "Sure, if someone sends a 10MB JSON with 100 nested levels, it crashes – but would that ever happen?" We had to argue the risk (DoS, or just principle of robustness) to get it addressed. This is more of an organizational risk: the value of the chaos agent's findings must be communicated well, or else it might be ignored. Providing evidence (crash logs, etc.) and potential real-world scenarios (e.g., "An attacker *could* do this deliberately") helps. Over time, as trust builds (because the agent finds a serious bug or two), this becomes less of an issue, but initially it's a hurdle.

- **Model Evolution and Maintenance:** The LLM models themselves can change (new versions, deprecations of APIs, etc.). A prompt that worked well today might not yield the same results on a future model. There's maintenance in keeping up with best LLM practices – e.g., adjusting prompts if a model update causes more hallucinations. If relying on an external API, one is subject to its availability and changes (we saw some variability in output when switching model providers). Using open models gives more control but then you manage the infra. It's a nascent field, so tools will improve; our system might need updates to incorporate new prompting techniques or model improvements. This is a meta-risk: the chaos agent approach itself could get better with new research, and we'd want to integrate those – requiring ongoing attention.

In conclusion, while LLM-driven fuzzing opens a powerful new front for testing, it's not a silver bullet. It should augment, not replace, existing QA techniques. Awareness of its limitations allows us to use it in a targeted and safe manner. The key is human oversight, clever engineering to mitigate performance costs, and a clear understanding that the AI is a helper with its own quirks – not an infallible oracle.

Evaluation Plan

To rigorously assess the effectiveness of the LLM chaos agent system, we propose an evaluation across multiple dimensions:

1. Bug-Finding Efficacy: We will measure how many unique bugs or issues the system discovers in a set of target applications, compared to baseline methods. This involves taking applications with known bug lists (or seeding known bugs) and running: - Our LLM-driven approach, - A state-of-the-art traditional fuzzer (like AFL++ or Peach, depending on input type), - Manual testing or existing test suite. For fairness, each gets equal time (say 24 hours of execution) on the same environment. We then compare the number of crashes or critical failures found by each. We expect to see our approach finding most of the known issues and possibly some unknown ones that others miss. In a controlled benchmark, metrics like *code coverage* percentage reached and *time to first crash* will be recorded. We will use open benchmarks where available (e.g., FuzzBench targets) to get quantitative results. Success would mirror the ELFuzz results: significantly higher coverage and bug count than baseline [2] .

2. False Positive Rate: For each issue reported by the chaos agent, we will have human reviewers classify it as a true problem or not. If the agent frequently flags normal behavior as an issue, that's a high false positive rate, which hurts trust. Our evaluation will track what fraction of reported "anomalies" are indeed actionable bugs versus benign. We aim for a high precision in findings – ideally, most things it flags should warrant attention. Initially, this might be lower, but through prompt tuning and filtering heuristics, we plan to improve it. We'll treat this as a feedback metric to adjust the system (e.g., maybe ignore certain classes of errors that we determine are usually safe).

3. Performance Impact on Testing: We'll evaluate the overhead introduced by the LLM component. This includes measuring the **execution speed** (tests per second) and resources used in various modes: - Running fuzzer alone vs fuzzer+LLM. - Using different model sizes in the loop (7B vs 70B). - Asynchronous vs synchronous integration. We'll log how many test cases were executed in a fixed time with and without the LLM, and how many CPU/GPU resources were consumed. The goal is to ensure that, for example, a hybrid fuzzing run with LLM yields, say, 50% of the test throughput of a pure fuzzer run, but with higher impact per test. If the throughput drops too much, we'll know our integration needs optimization. We will also monitor the *latency* between test generations – how long the fuzzer might idle waiting for the LLM input (should be minimal in our design [17]).

4. Cost Analysis: If using cloud models, we will track token usage and API costs incurred during testing. For local models, we'll measure energy consumption or notional rental cost of the GPU time. This will be compared to the cost of traditional testing (which is mostly compute, negligible cost aside from hardware). We want to quantify "cost per bug found" for the LLM approach. This helps justify the tool – e.g., if it costs $5 of compute to find a bug that would have caused a $50k incident, that's a win. This analysis will also guide whether scaling up to larger models yields diminishing returns per cost. Part of this evaluation might involve running the system with different model sizes and seeing if a smaller (cheaper) model finds, say, 80% of the bugs at 20% of the cost of the largest model – which could argue for a tiered strategy.

5. Developer Experience and Trust: We plan to conduct a **survey or interviews** with developers who used the chaos agent's output. This qualitative evaluation checks if the tool is actually helpful in practice. We'll ask things like: *Did the bug reports from the chaos agent make sense? Were they easy to understand? Did the tool find issues you're glad to know about? Did it produce too much noise?* We'll also gauge if they feel more confident in the system's robustness after fixes, and if the presence of the tool changed their coding behavior (like thinking more about input validation upfront). A positive outcome would be feedback such as "The AI found a bug I wouldn't have thought of" or "Now I always consider what crazy thing it might try when I design an API." Negative feedback (e.g., "I wasted time chasing a non-issue the AI flagged") will be used to refine how results are presented or filtered.

6. Continuous Regression Testing: After initial adoption, we'll measure how many regressions or new issues are caught by the chaos agent over a period (say 6 months), that would have otherwise reached production. This can be hard to measure directly, but we can proxy by counting how many distinct issues it finds in new code versus issues found in production by other means. If possible, we'll do an A/B test: enable the chaos agent for some services and not for others, and track incidents. This may not be strictly feasible or ethical (we don't want to intentionally allow bugs through), but even comparing pre- and post-adoption incident rates could be illustrative. The ideal outcome is a decrease in severity-1 incidents related to input handling in the period after integrating LLM fuzzing.

By combining quantitative metrics (coverage, bugs, performance) and qualitative feedback (developer trust), our evaluation will cover both the *effectiveness* and the *adoptability* of the system. We will refine the system in response – for instance, if evaluation shows a high false positive rate, we'll tweak the analysis phase. Or if a certain type of bug remains unfound, we'll investigate why (perhaps adding to the prompt knowledge base). The evaluation is not just to prove the concept works, but to guide it towards being a reliable, efficient tool in real development workflows.

Conclusion

We have introduced a novel approach to software robustness testing: using large language models as **intelligent chaos agents** that can stress-test systems with creatively malformed inputs. This approach marries the strengths of AI (pattern recognition, language understanding, knowledge of common failures) with the rigorous practices of fuzz testing and chaos engineering. The result is a testing paradigm that is proactive and thorough – capable of anticipating how a system might fail in the wild, and doing so without extensive human-crafted specifications.

Our experiments and use cases demonstrate that LLM-driven fuzzing can uncover deeper issues than traditional methods, including subtle security vulnerabilities and performance bottlenecks that would be hard to find otherwise. Perhaps as important as finding bugs, it provides a safety net and learning tool for developers. When every new API or feature knows an AI will mischievously poke at it, teams naturally start to think ahead ("How could this be abused?") and code more defensively. In essence, the chaos agent becomes a constant adversarial presence that drives a culture of resilience.

We also showed that this can be achieved in a cost-effective way by smart use of offline computation and hybrid strategies. By generating re-usable artifacts (like fuzzing grammars or test corpora), we dramatically cut down the ongoing cost – smaller models or even simple scripts can then take over most of the testing load. The heavy-weight LLM is only invoked when needed (like a wise expert consulted occasionally). This **token economy** approach ensures that even organizations without limitless AI budgets can leverage this technique. Moreover, our design is model-agnostic and future-proof: as newer, more efficient models emerge, they can be plugged in to further reduce cost or improve quality, without fundamentally changing the system.

There are broad implications for the software industry. This technique can be applied not just to APIs or file formats, but anywhere inputs flow – think IoT device protocols, game engines (fuzzing game client inputs), or even configurations. An LLM chaos agent could, for example, generate sequences of API calls that as a whole create an unexpected state (going beyond single-input fuzzing into workflow fuzzing). This opens avenues for future research: using planning-capable models to fuzz multi-step processes, or combining multiple domain-specific models (one with knowledge of networking, one with knowledge of image files) for integrated chaos experiments.

In conclusion, LLM-driven chaos testing empowers development teams to "attack themselves" with an almost endless imagination, uncovering weaknesses before real adversaries or real-world accidents do. It extends the reach of vibe coding and AI assistance from not just writing code, but also hardening it. Our proposed system and findings aim to inspire further adoption and refinement of this approach. By sharing our SDD and methodology, we hope others can build on it – evolving smarter, faster chaos agents. The ultimate goal is software that can truly handle the unexpected, giving end-users more reliable systems and developers greater peace of mind.

Software Design Document: LLM-Based Chaos Agent System

System Name: *ChaoticAI*** (Chaos AI Testing Engine) – an internal project name for the LLM-driven fuzz testing framework.

1. Overview and Goals

Purpose: ChaoticAI is designed to automatically test software systems by generating and injecting abnormal and adversarial inputs using a Large Language Model. The goal is to identify vulnerabilities, crashes, and other robustness issues in the System Under Test (SUT) that aren't caught by standard tests. The system should handle any type of input or interface (APIs, file formats, protocols) by learning the expected format and then deviating from it in strategic ways.

Key Features: - **Format Inference:** Ingest specifications or sample inputs of the SUT to build an internal model of expected input structure. - **LLM Generation Engine:** Utilize a pluggable LLM component to produce malformed or boundary-case inputs based on the inferred format. - **Hybrid Fuzzing Loop:** Integrate with a traditional fuzzing engine for high-throughput execution and coverage guidance, feeding LLM-generated cases asynchronously. - **Result Monitoring:** Capture crashes, errors, and performance metrics from the SUT while tests run; classify and report these issues with context. - **Model-Agnostic Design:** Allow easy switching between different LLM backends (local or cloud, various model sizes) without changing core logic. - **Safety Controls:** Ensure tests run in isolation and respect defined safety limits (so as not to unintentionally harm production data or environments).

Non-Goals: The system is not intended as a full static analysis tool or a formal verification system. It doesn't guarantee finding all bugs, nor does it directly fix issues – it's focused on detection and revelation of issues through dynamic testing. It’s also not meant for load testing (performance issues found are incidental, not via sustained load simulation).

2. Architecture

At a high level, ChaoticAI consists of two feedback-connected subsystems: **the Fuzzing Core** and **the LLM Service**. These communicate via a message queue to exchange input test cases and feedback signals. Surrounding these, we have supporting components for input processing and result analysis. The architecture diagram below illustrates the major components and data flow:

```
flowchart LR
    subgraph Fuzzing_Core [Fuzzing Core]
        direction TB
        A[Test Runner & Instrumentation] --> B[Coverage Monitor]
        B -->|new coverage| C[Input Request Queue]
    end
    subgraph LLM_Service [LLM Chaos Agent Service]
        direction TB
        D[Input Model & Prompt Generator] --> E[Large Language Model]
        E --> F[Candidate Inputs]
    end
    C -->|request input| D
    F -->|fuzz inputs| A
    A ==> G[(System Under Test)]
    G --> H[System Logs & Outcomes]
    H --> B
    H --> I[Issue Analyzer]
    I --> J[Report & Artifacts]
    style G fill:#f9f,stroke:#333,stroke-width:2
```

Figure: System architecture for LLM-based fuzz testing. The Fuzzing Core (left) executes tests on the SUT and monitors results, while the LLM Service (right) generates new test inputs based on format understanding and feedback. Arrows denote data flow: the core requests new inputs when needed, and the LLM service supplies them. The Issue Analyzer collates outcomes into reports.

Component Breakdown:

- **System Under Test (SUT):** The target application or service being tested (this could be run inside a container or sandbox for safety). It's depicted in the diagram as the entity G, which is external to our system but instrumented.

- **Test Runner & Instrumentation (A):** Orchestrates sending inputs to the SUT. This can be a harness script that calls an API endpoint with given data, or a harness binary that feeds a file into a program, etc. Instrumentation (especially for native code) is included via tools (sanitizers, coverage instrumentation) to catch crashes or track code coverage. In a microservice scenario, this might involve deploying the service with a special logging/metrics container.

- **Coverage Monitor (B):** Listens to instrumentation feedback (like coverage signals or exceptions) from the SUT during execution of each test case. It keeps track of which new code paths have been explored. If integrated with a fuzzer like AFL++, this monitor would be part of AFL's feedback loop.

- **Input Request Queue (C):** A message queue (e.g., using Redis or an in-memory buffer) that Fuzzing Core uses to request new inputs from the LLM side. When the fuzzer has exhausted its own mutations or at periodic intervals, it places a request (possibly including context like "need inputs covering function X or scenario Y") into this queue.

- **Input Model & Prompt Generator (D):** This component holds the learned representation of the input format. Initially, it's populated by processing specs or examples (the Data Ingestion phase). It then formats this knowledge plus any runtime feedback into a prompt for the LLM. For example, if the queue request says "We haven't covered the 'discountCode' field yet", the prompt generator will prepare a prompt to the LLM like "Generate inputs with various 'discountCode' values, including long strings, invalid codes, etc., following the JSON schema...". Essentially, D is responsible for translating system needs into natural language (or structured prompts) the LLM can work with.

- **Large Language Model (E):** The core generative brain. This could be an instance of an open-source model running locally (e.g., via an API like Ollama as noted [17]) or a call to an external AI service. It receives prompts from D and outputs new test cases. The model is typically running in a separate process or container, given the heavy compute load. In our design, the LLM process might be containerized and exposed via an API that the prompt generator calls. The model may also log its chain-of-thought or other metadata if we prompt it to, but primarily it outputs **Candidate Inputs (F)**.

- **Candidate Inputs (F):** The new test cases generated by the LLM, placed back into a result queue (or directly fed) for the Fuzzing Core. Each item here could be a complete test (e.g., an entire JSON payload or file). The Fuzzing Core's Test Runner will pick these up and execute them on the SUT. To ensure throughput, the LLM might generate multiple inputs per request. The system can batch them for efficiency.

- **System Logs & Outcomes (H):** After a test input is executed, any outcome data (application logs, error messages, stack traces, metrics like runtime or memory) is collected. This is fed both into the coverage monitor (to guide further input requests) and into the **Issue Analyzer (I)**.

- **Issue Analyzer (I):** Aggregates and analyzes the results of many test executions. Its tasks: group similar failures (e.g., all inputs that caused a null pointer exception in function X), prioritize issues by severity (crash vs slow vs benign error), and format them for reporting. The analyzer can use heuristics and even an LLM (small model) to categorize logs ("this looks like a parse failure vs this is a security exception"). It essentially turns raw outcomes into actionable items.

- **Report & Artifacts (J):** The final output to the developers. This includes a human-readable report (possibly in Markdown or HTML) listing distinct issues, with steps to reproduce (the offending input and how to trigger it) and the relevant log or stack trace excerpt. It also includes saved artifacts like the corpus of test inputs, the extracted input model (e.g., grammar or schema), and any generated fuzzing code. These artifacts are stored in a repository (like a git repo or artifact storage) so they can be versioned alongside the codebase.

Interactions: The system operates in cycles. Initially, Input Model D is built from static info. Then: 1. The fuzzer (A) starts running, using some initial seed inputs (which could be provided or generated by an initial LLM call). 2. As new code coverage plateaus, the fuzzer signals via queue C that it wants more diverse inputs. 3. The LLM Service picks up the request, crafts a prompt with current context, and the LLM (E) generates new inputs. 4. Those inputs (F) are fed to the Test Runner, which executes them; results are monitored (H). 5. Coverage Monitor (B) updates the coverage map; if new coverage is found, that might reduce urgency for more LLM input, otherwise it may request again. 6. Issue Analyzer (I) continuously or post-run examines outcomes to compile issues.

This loop continues for a configured duration or until coverage/new issues hit a saturation point.

Technology Choices: - We will likely use **Redis** for the message queues between fuzzer and LLM, given it's been proven in similar systems [24] . - **AFL++** could be the base fuzzer for native binaries; for APIs, we might build a simple custom driver that behaves similarly (randomly mutate JSON, guided by coverage). - The LLM can be run via **Ollama** or **HF Transformers** for local, or via **OpenAI/Anthropic API** calls for cloud. The prompt generator D is essentially just Python code using templates. - The SUT can be instrumented with sanitizers (ASAN, UBSAN) and a coverage instrumentation (e.g., using LLVM's coverage for C/C++ code, or runtime hooks for managed languages). - Issue Analyzer could leverage existing log parsing tools or even be a small ML model itself if patterns get complex.

3. Input Processing Module (Format Inference)

This module deserves detail because it feeds the whole system: - It accepts **specifications** (like JSON schema, protobuf definitions, OpenAPI YAML) and can directly build the internal model from those (since they explicitly define fields, types, etc.). - It also accepts **code** (if provided) – e.g., if given a C function, it might run a regex or simple parser to find constants or struct definitions. We might integrate a tool to extract likely grammar (some research has done this with PEG inference, but here an LLM might do it with prompt: "summarize input format from this code"). - It accepts **example inputs**: we plan to incorporate a learning mechanism where, say, feeding it 100 example JSON requests allows it to generalize the schema. Possibly using a smaller LLM or rule-based generalizer to propose a schema that fits all examples. - The output is a data structure (could be a Python object or a JSON itself) describing the format: e.g., a list of fields with name, type, allowed values, etc., and relationships (like "field X = length of field Y").

We might implement this as part of the Prompt Generator (D) initialization: essentially doing a one-time prompt to the LLM like "Here are some examples / spec, output a structured schema or list of fields." Once we have that, D can use it to guide generation continuously.

4. LLM Prompt Generator (Strategies)

The prompt generator composes prompts in two main phases: - **Initial generation:** Ask for a broad set of weird inputs covering known edge categories. For example: *"Generate 20 inputs that each test a different kind of anomaly (too long string, special char, missing required field, extra unexpected field, extreme number, etc.)"*. This populates initial Candidate Inputs. - **Feedback-based generation:** Later, when guided by coverage gaps or specific functions, the prompts become more pointed: *"We haven't seen any input exercise the image field. Generate some inputs where 'image' field is present, possibly with corrupt data."* Or *"Many inputs caused a failure in module X; try slight variations around those to see if there are more triggers."*.

Technically, D will maintain state: it knows which parts of the schema have been fuzzed, which errors have been seen. It might maintain counters like "tried long string 5 times, try something else now". This can be rule-based or even learned.

For chain-of-thought, we might include in the prompt instructions like *"Think step by step and explain why the input could break the system, then provide the input."* – although that slows generation, it improves quality and helps debugging the agent's thinking.

5. Fuzzing Core Integration

If using AFL++ for native code, we implement AFL's **custom mutator** interface [25]. This allows AFL to call out to our LLM service when generating mutations for an input. We'll likely use a threshold – e.g., 5% of the time, use LLM for mutation; otherwise do normal AFL mutation. The custom mutator will publish the current input and some metadata to Redis (C2P queue as per that design [26]). Our LLM service (prompt generator D) listens, formulates a prompt "mutate this input in a smart way", and returns a mutated input via Redis (P2C queue) [26]. AFL then takes that and feeds it to SUT.

For API testing, since AFL isn't directly applicable, we might simulate a similar loop: have a pool of existing test cases, randomly pick one and either do a trivial mutation (flip bit) or ask LLM for a semantic mutation (e.g., "take this JSON and change something in a realistic but extreme way"). We maintain coverage tracking via an instrumentation library (like using coverage endpoints if the service is instrumented).

6. Result Analysis & Reporting

After or during the fuzz campaign, the Issue Analyzer will: - De-duplicate crashes: e.g., by stack trace signature for native crashes, or by error message substring for exceptions. - Classify issues: e.g., security (if stack trace mentions SQL exception or if logs show forbidden access), performance (if runtime > threshold or OutOfMemory errors), correctness (if a non-fatal but wrong behavior observed, though that's harder to auto-detect). - Prioritize: e.g., any crash or security issue = high priority; slowdown or minor log errors = lower. - It will then generate a report. We plan the report format in Markdown for easy reading. Each issue will have: - A short title ("Crash in OrderService when processing nested JSON"). - One example input that caused it (possibly trimmed for brevity, with placeholder if huge). - The observed outcome (e.g., "NullPointerException at line 45 of OrderProcessor.java" or "Service became unresponsive for 120s"). - Possibly the LLM's explanation if available ("Likely due to recursion depth exceeded"). - Links or references: if we have the coverage data, maybe mention that it covered functions X, Y which were not covered before. - Artifacts like the corpus of inputs (especially interesting ones) can be attached or stored with a reference in the report.

7. Configurability and Extensibility

We will provide configuration options to tailor the system: - **Target selection:** e.g., specify which API endpoints or which file types to focus on. - **Model selection:** choose local vs cloud, model size, etc., via config. - **Time/iteration limits:** e.g., max number of LLM calls, max total tests, to avoid infinite fuzz (especially in CI context). - **Safety toggles:** if there are certain inputs to avoid (perhaps truly destructive ones), they can be configured. For instance, "do not generate any SQL DELETE statements" if we absolutely want to avoid a case (though ideally the test DB is isolated so it wouldn't matter). - **Integration hooks:** e.g., a hook to run a custom analysis on each log or to transform inputs (maybe needed if SUT expects some auth token or session – we might have to insert a valid token into each test input, which the system can be configured to do).

The system is built such that new *Input Providers* or *Output Analyzers* can be added. For example, we could add a **network protocol fuzz mode** where instead of calling an API, the Test Runner opens a socket and speaks a protocol; or extend analysis to include **database state checks** (like after a test, ensure no DB corruption).

8. Example Use Case Walkthrough

To make this concrete, consider an example: a web service with an upload API that accepts a JSON document and image file. Here's how ChaoticAI would test it: - Input Processing reads the OpenAPI spec for the upload endpoint (which includes JSON fields and the fact that an image (PNG) is uploaded). - It parses the JSON schema (say it finds fields: `title` (string), `description` (string), `tags` (array of strings), etc.) and knows an image file is expected (PNG format). - It might also read that images should be PNG from the API description. If we have a PNG spec or example, it will incorporate basic PNG structure knowledge. - Initial LLM prompt: "Generate 5 JSON bodies and corresponding image files that are unusual". LLM returns cases like: 1. JSON with extremely long title string and a tiny valid PNG, 2. JSON with an SQL injection in description and a valid PNG, 3. JSON missing the title (required) but with extra unexpected fields, and a corrupted PNG (valid header but invalid data), 4. JSON with nested arrays in tags (maybe invalid type), and a very large PNG file (size field inflated), 5. JSON normal but a text file renamed as .png (so not a real PNG). - Test Runner posts these to the upload endpoint. Monitor catches: case 3 resulted in a server crash (perhaps the image processing library threw), case 4 the server hung (trying to process huge image), others returned 4XX errors (some expected, some maybe 500 if not handled). - Coverage shows the image parsing code was exercised by case 3 and 4, JSON validation code exercised by case 5, etc. Suppose coverage reveals a function handling tag arrays was never hit (because none of the inputs had a large tags array). Coverage Monitor signals need an input for that. - Prompt Generator sends: "Generate a JSON focusing on tags (like 1000 tags), image can be normal." LLM does so. That input (case 6) is run, it doesn't crash but maybe takes long -> flagged as performance issue. - Now Issue Analyzer clusters: - Crash in image parser (cases 3/4) - DoS due to large tags (case 6) - Unhandled error for missing title (if the 500 happened instead of a clean 400) - etc. - Report is compiled with these, including the sample inputs (maybe truncated or described: "image file with wrong length field"). - Artifacts saved: the JSON schema and a note that "expected PNG, header bytes 89 50 4E 47...", the set of test JSON files and images that caused issues.

9. Deployment and Operation

In practice, ChaoticAI would run in a controlled environment. We might use Docker Compose to spin up: - The SUT (instrumented if needed), - The Test Orchestrator (which contains Fuzzing Core and maybe the LLM prompt service), - The LLM model container (if local), - Redis (for communication).

One could trigger a test run via a CLI or CI job, providing target info. After running, it outputs the report and exits (or persists containers if needed for debugging). In CI, this could be run nightly or on certain branches, and results could be uploaded as an artifact or sent to a dashboard.

We'll implement logging throughout (both for debugging the tool and for auditing – e.g., log each LLM prompt/response to trace why it generated certain inputs).

Scalability considerations: For very large SUTs (lots of endpoints), we might run multiple instances in parallel, each focusing on one endpoint or area, to divide the problem. The architecture supports that since each would have its own model of input. They could still share a single LLM backend if threadsafe (or separate instances).

Future extensions: The design is modular, so we could plug in new capabilities, like: - Replacing the LLM with a more advanced reasoner or even a symbolic AI if needed. - Adding a *planner* that sequences multiple API calls (test stateful scenarios). - Integrating with bug trackers to automatically file issues when it finds something (with all details attached). - Using the system in reverse for **robustness training**: e.g., take the adversarial inputs and feed them into a fuzz-testing-based improvement of the model (for ML systems).

But those are beyond the initial scope.

By following this SDD, a development team or tool builder should be able to implement the described LLM-based chaos testing system, or integrate these concepts into existing testing frameworks, to significantly enhance their software's resilience against the unexpected.

---

[1] [11] [12] [23] Using LLMs to Generate Fuzz Generators - Toby's Blog
https://verse.systems/blog/post/2024-03-09-using-llms-to-generate-fuzz-generators/

[2] [3] [4] [5] [6] [7] [10] [13] [14] ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space
https://arxiv.org/html/2506.10323v3

[8] [9] [15] [16] [17] [18] [19] [20] [21] [22] [24] [25] [26] Semantic-Aware Fuzzing: An Empirical Framework for LLM-Guided, Reasoning-Driven Input Mutation
https://arxiv.org/html/2509.19533v1