Why Geolog?

Martin Kleppmann

January 1, 1980

Abstract

The ARIA Safeguarded AI programme is developing a system called Geolog. Other documents explain the details of its design, but leave much of the motivation behind those design decisions implicit. This document offers an opinionated justification for why I believe Geolog is important. It incorporates insights from others in the SGAI programme, but it's my personal opinion, not a statement for the programme as a whole.

1 The national security case for formal verification

AI is rapidly advancing in many areas, but one particular area of concern is that AI agents are now able to independently find and exploit large numbers of high-severity security vulnerabilities in widely-used systems [Anthropic Frontier Red Team, 2026, Carlini et al., 2026, Ptacek, 2026]. As cybersecurity operations are now considered an indispensable element of military strength and deterrence [U.S. Department of Defense, 2023], we have to assume that these public reports are only the tip of the iceberg, and that nation states have secret research programmes that are stockpiling security vulnerabilities in software, hardware, and protocols for use in a potential future armed conflict. Just a few years ago, finding and exploiting vulnerabilities in computer systems was expensive [Slayton, 2017], but this is no longer true.

Cybersecurity generally favours the attacker, who has to find only one vulnerability to compromise a system, whereas the defender has to find and patch *all* vulnerabilities to make their system secure [Ryseff, 2017]. This fact, together with AI's rapidly growing capability for finding vulnerabilities, worries me a lot. The theory of *offense-defense balance* [Jervis, 1978], which is widely believed in international relations, says that when offensive technologies advance faster than defensive ones, this has a destabilising effect: a country that believes it has a temporary advantage in some offensive technology may decide to attack an opponent in order to exploit that advantage while it lasts. What matters is not so much whether this offensive advantage is real, but rather whether a potential aggressor *believes* it is sufficient to warrant starting a war [Ryseff, 2017].

In a time of increasing geopolitical tensions, with commentators warning that we may already be in the foothills of World War 3 [Smith, 2026], there is an

urgent need to stabilise the situation as much as possible to reduce the risk of a catastrophic escalation. (Personally I am less worried about the risk of a superhuman AI deciding to eradicate humanity, and more worried about humans deciding to attack other humans with the help of AI-enabled technology.) One way we can contribute to stabilisation is through advancing defensive technologies and making them widely available, so that a would-be aggressor is more likely to decide that it is not worth launching an attack in the first place.

In the cybersecurity domain, eliminating vulnerabilities is one of the best defensive measures. Some types of vulnerability can be avoided by using better tools: for example, using a memory-safe language like Rust instead of C/C++. However, many critical software systems like operating system kernels or JIT compilers for JavaScript/Wasm in web browsers inherently need to perform unsafe operations (interfacing with hardware devices, or generating machine code from untrusted input and executing it), which cannot be made safe simply by porting to Rust. Rust also provides only memory safety, but cannot rule out vulnerabilities due to logic bugs, such as incorrect access control checks.

A more comprehensive approach is to use formal verification, i.e. to prove that a program conforms to some specification under all possible executions. This specification can include memory safety properties, but also properties about the logical correctness of the program's security features, and any other properties that are considered important. These proofs are typically written in a specialised programming language and then checked by a tool such as Lean, Rocq, or Isabelle, which have a small, trusted proof-checking kernel, making it very unlikely that any bugs remain in the verified aspects of the program.

Until recently, formal verification was very expensive, requiring a large time investment from people with highly specialised skills to write the proofs even for small systems. For example, in 2009, the formally verified seL4 microkernel consisted of 8,700 lines of C code, but proving it correct required 20 person-years and 200,000 lines of Isabelle code – or 23 lines of proof and half a person-day for every single line of implementation [Klein et al., 2009].

Fortunately, AI is also getting very good at writing formal proofs; a lot of the current focus is on AI agents for writing proofs in Lean [de Moura, 2026, Achim et al., 2025, Bodnia et al., 2026, Mistral AI, 2026, Math, Inc., 2026, Chen et al., 2025, Ren et al., 2025]. This offers the prospect of formally verifying all of the software on which our critical infrastructure depends in the foreseeable future – an idea that would have been ludicrous a few years ago. And since formal proofs are automatically checked by the kernel, hallucinated proofs are rejected without any human involvement, making formal verification an excellent use case for our current powerful but imperfect LLMs [Kleppmann, 2025].

The pervasive use of formally verified computer systems would strengthen the defensive side of cybersecurity, and thereby shift the offense-defense balance towards stability, reducing the risk of armed conflict. It would also reduce the risk of other harms created by security vulnerabilities, such as the WannaCry ransomware attack that seriously affected the NHS in 2017.

2 The case for a new proof assistant

To make this concrete: what would it take to prove that, for example, a version of the Linux kernel or the Firefox web browser is free from security vulnerabilities? Both of these systems consist of tens of millions of lines of code, making them several orders of magnitude larger than any formal verification project that has been attempted before. A proof of such scale cannot be accomplished by a single agent – it will definitely require many agents working in parallel on different aspects of the proof, for example verifying different modules in the code base, or trying different proof strategies on the same module in parallel to see which one works.

Coordinating between all those agents will require an infrastructure to keep track of which agent is working on what, which theorems have already been proved, which proof strategies have been attempted (including any dead ends, so that they don't get tried again), and so on. The Categorical Core of SGAI is that infrastructure, and Geolog is the language and data model of the Categorical Core. Think of Geolog as a database that is optimised for storing large programs and AI-generated proofs about those programs.

In fact, Geolog is a general-purpose database, and not specific to software verification. Building an automated researcher, as AI firms are now focussing on [Heaven, 2026], will require the same kind of infrastructure. However, I think that examining a concrete use case like "how could we make the Linux kernel provably secure?" helps bring clear justifications to design decisions that would otherwise remain abstract and vague. It also gives us a potential long-term vision to rally around, even if we are still years away from actually being able to verify a system as large as Linux.

AI agents writing formal proofs work best by rapidly iterating on small proof steps, and getting feedback from the proof checker after each step whether it is valid or not – much like the interaction that many proof assistants offer to human users, but with the AI providing the 'intuition' and guidance on which proof step to try next. At present, this can be done with Lean by using the Lean Language Server to expose an interactive LSP interface to the prover, and allowing an AI agent to call into this language server via MCP [Dressler, 2025].

That raises the question: why not simply use Lean, given that so much investment in AI-generated proofs is already focussing on Lean? A core hypothesis of the SGAI programme is that Lean will not scale to the size of proofs we are contemplating, because it is fundamentally designed for human-written proofs, and its support for AI-generated proofs is bolted on retroactively. By starting from a clean slate, we can design Geolog from the ground up to take advantage of the strengths of AI agents. While this is more work up front, we believe that it will unlock the ability to tackle more ambitious proof projects than will be possible in Lean.

Some of the key differences between Lean and Geolog are:

- Lean does not have any mechanism that allows multiple agents to collaborate on a proof, or to backtrack in case they hit a dead end in the proof

search. The best option is to put the Lean source files in a Git repository and let the agents make Git commits and inspect the Git history in the same way as a human would. But concurrent work on the same module would quickly lead to merge conflicts that are messy to resolve at the level of the textual source, and the proof structure has no way of referencing the Git commit history since Lean and Git are two separate systems. In contrast, Geolog is a database with built-in version control facilities, including principled ways of merging concurrent work from different agents. We are planning to allow the data in the database to reference its own version history, making it possible to record structured information about the meaning of the version history within the system itself. For example, the version history could encode the step-by-step evolution of a complex proof, and the database can record which proof rules, axioms, and lemmas were invoked at each step of the proof.

- The current integration between Lean and AI agents via MCP and LSP is baroque and slow, since the language server was designed for interaction with humans, not AIs. The faster we can make the feedback loop between AI and proof checker, the faster a correct proof can be found. With Geolog, we can bring the proof checker as close as possible to the AI model that is driving it: not just on the same machine, but potentially even running on the same GPU as the LLM, to enable extremely fast round-trip times.

- Lean uses text files as its representation, i.e. a linear sequence of symbols, because this is convenient for humans to enter on a keyboard and convenient to display on a two-dimensional screen for humans to read. But text is not amenable for reasoning: in order to prove something about a program, it first has to be parsed into a richer structure such as an abstract syntax tree (AST) or an intermediate representation (IR), and then manipulated at that level. AIs are not limited to keyboard input and they can work with multidimensional data better than humans can, and so Geolog skips the textual representation,[1] and allows the AI to interface directly with the AST or IR representation of programs and proofs.

Those differences motivate why the SGAI programme is starting afresh with Geolog, and incorporating lessons learnt from Lean and other proof assistants, rather than using the Lean system as it stands today.

[1] But LLMs are *language models*, I hear you cry, and language is inherently textual! Thus, does it really make sense to drop the textual representation?

Yes, in order for the LLM to interact with Geolog, it will probably output tool call requests as a stream of tokens, and the response from Geolog will go back into the LLM as tokens. However, following the principle of small proof steps, these requests and responses will be quite small. Rather than feeding an entire module of a program into the LLM as source code text turned into a token stream, it will be much more efficient for Geolog to store the code in a structured form, and to allow the LLM to interactively query it to learn exactly those things that it needs to know about the code. The textual representation is then only an incidental detail of the LLM/Geolog interface, and the structured representation stored by Geolog is the primary one.

3 The case for a new DBMS

In addition to being a proof assistant, Geolog is also a new kind of database crossed with a version control system – one that is optimised for storing a structured representation of programs and proofs about those programs. That raises the next question: building a new database is hard (folklore in the databases community says that it takes 10 years to build a new database), so can't we just use an existing one and save ourselves a lot of effort?

There is some related work: for example, Joern CPG [Yamaguchi et al.] and Fraunhofer AISEC CPG [Fraunhofer AISEC] define ways of mapping code structure onto a graph database such as Neo4j, and Galois MATE CPG [Galois, Inc.] stores the IR of a program in PostgreSQL. These tools are already widely used for finding security vulnerabilities through static program analysis. However, they are not designed to support proofs about programs, and they assume a static program rather than one that is being manipulated by agents.

In the context of AI agents collaborating on proofs, a server-centric database like Neo4j or PostgreSQL is not suitable, for the following reasons:

- In order to make the feedback loop for trying proof steps as fast as possible, we cannot have a network round-trip on every access to the database. It is essential that each agent can read and write its own local copy of the database, and replicate any writes asynchronously to other agents' replicas without blocking the AI's progress. In other words, we need a *local-first* approach [Kleppmann et al., 2019].

- In order to allow multiple agents to make progress in parallel without interfering with each other, we may want to let each work for a while on a private *branch* (in the version control sense), and decide later which (if any) branches to merge. In case one branch of a proof gets stuck, we want an agent to be able to go back to an older state and try branching off some previous version with a different proof strategy. These operations are supported by version control systems, but not by most databases.

A version control system like Git is closer to what we need in terms of supporting local reads/writes, branching, and reverting to old versions. But the downside of most version control systems is that they operate only on text (you can commit binary files to a repository, but the VCS doesn't provide any help with diffing or merging those files). As discussed previously, we need to operate on a structured data model with well-defined semantics for merging updates from different agents, which Git or similar systems do not provide. In fact, Git's merge semantics are poorly understood and surprising in various ways [Glodny, 2025].

The closest existing system to what we need is Automerge, an embedded database for local-first software with support for Git-like version control that uses CRDTs [Shapiro et al., 2011] to merge branches with well-defined semantics. However, Automerge's data model is JSON (with a few extensions), which is not well suited for representing programs and proofs.

Fortunately, the Automerge team is part of the SGAI programme. Our intention is to reuse the infrastructure we have already built for Automerge (physical data layout, versioning mechanisms, compression, replication protocols, authentication and access control, sync servers and clients, user interface tooling, etc.), and replace only the JSON data model layer with a new data model that is designed for programs and proofs. That way, we are starting the 10-year process of building a new database with a head start of several years, while at the same time having the freedom to build a system that is targeted towards the specific needs of the SGAI programme.

4 The case for a new data model

What data model should Geolog use? The aforementioned CPG projects demonstrate that it is possible to map code ASTs and IR to property graphs and the relational model. These general-purpose data models have the advantage that they are implemented by existing databases. However, they only capture a single version of a program, whereas for proofs we often need to describe a sequence of reasoning steps, each of which makes a statement about a program that follows from an earlier step. Modelling this with an existing relational or graph database quickly gets messy and inefficient. A new data model is needed.

In category theory it is common to reason using *diagrams*, in contrast to the reasoning based on textual syntax that dominates existing proof assistants. A diagram is not necessarily represented visually (although it can be); it is really a collection of objects that may be related to each other according to certain rules (different types of diagrams can set different rules). Data structures like graphs, trees, ASTs, IRs, string diagrams, petri nets, electronic circuits, etc. are all diagrams. Rather than picking one of these models, Geolog offers a generalised way for users to define their own types of diagrams by specifying their own rules for what types exist and how they can be related. As this is convenient for work in category theory, the database system based on Geolog is called the *Categorical Core*.

Geolog is based on the relational data model, which has proved extremely versatile and long-lasting, despite being over half a century old [Codd, 1970]. Almost any data structure, including diagrammatic representations of programs and proofs, can be represented in a relational way. However, the implementation of the relational model in SQL databases has limitations. In particular, SQL offers only very few types of constraints that an application can declare on a database: there are foreign key constraints, constraints on the values within a single row (check constraints), uniqueness constraints… and that's it. There is no way of specifying that the data must have the shape of, say, a linear linked list, or a tree, or a DAG.

In a server-centric database that is not a big problem, because consistency of the database can be enforced using serializable transactions: if the database starts off in a consistent state, and every transaction transforms the database from one consistent state to another consistent state, then by induction it will

remain consistent, even if the consistency property is not explicitly declared as a constraint. However, in a local-first system there are no serializable transactions (since that would require coordination between concurrent writes, slowing down parallel agents, and it would also rule out branching and merging workflows). It could then easily happen that two agents make updates that individually preserve consistency, but that violate consistency (e.g., breaking the linearity of the list or the acyclicity of the graph) when merged. Consequently, if we want branching/local-first and also want to ensure that the data has some well-formed structure, we need a more powerful way of declaring constraints than is offered by SQL.

This is the key thing that Geolog provides. Geolog is a dependently typed programming language that makes it possible to declare a relational schema with database constraints that are vastly more expressive than what SQL provides. As a user of Geolog you can write *theories* in this programming language, which are definitions of a database schema (data types that can be stored) and *axioms* (logical statements about the data in the database; the database will reject any writes that would violate an axiom). An *instance* of a theory is a dataset that conforms to the data types and axioms encoded in the theory.

You can think of a Geolog theory as a way of defining a custom data model for a database: if you define a theory of graphs, you have a graph database. But you can also define a theory for an IR to represent programs, and define constraints that ensure it is well-formed. For example, you can express that a sequence of instructions must appear in a linear order, and that the inputs to one instruction must be the outputs of instructions that appear earlier in that linear order. You can also express that every type of instruction must have a certain number of inputs, and check that every instruction has the correct number of inputs based on its type. These are just examples; Geolog is extremely general and flexible in what you can express.

It is even possible to implement a proof checker, analogous to Lean's kernel, using Geolog. This way, the database will only allow you to write data that is accompanied by a proof that demonstrates why it is true – for example, proving that one version of a program is semantically equivalent to another version by referencing an axiom or proof rule that allows a semantic-preserving rewrite of one version into the other version. Database constraint checking then becomes equivalent to proof checking, and any theorems stored in the database are guaranteed to be true (provided that the proof checker and its axioms are correct).

Now assume that two agents make concurrent updates to a database that individually satisfy the constraints and proof rules, but which violate a constraint when merged. In this case the database will detect a merge conflict, and ask the AI agent to resolve it. Due to the precise nature of how constraints are specified in Geolog, resolving this merge conflict is a very different matter from resolving a textual merge conflict in Git: in Geolog, the conflict specifies exactly which constraint is violated by which rows in which tables. The best way of resolving the conflict will depend on what the agents were trying to achieve; one option might be to simply discard one of the branches. In some situations, it might be possible to do a CRDT-style automatic merge that ensures all constraints are

satisfied in the merged version. But in general, resolving the merge conflict may not be deterministic, and may require the same kind of AI 'creativity' or 'intuition' that is needed to choose the next step of a proof.

Although the Geolog language is very powerful, it is also deliberately restricted compared to other dependently typed programming languages like Agda or Lean. It only allows the expression of constraints that can be represented in *geometric logic* (hence the name Geolog), a fragment of first-order logic that does not allow negation and restricts the use of implication and quantifiers. There are several reasons for this choice of logic; the main one is that it guarantees that constraints can be checked efficiently. In particular, if some new data is written to the database, the system only has to examine the new data that was written, and it doesn't have to go and re-check all of the constraints for all of the existing data – in other words, constraints can be checked *incrementally*. Or, phrased in the language of proofs: if you add new facts to the database, that cannot invalidate theorems that have already been proved, because the proof checker is guaranteed to be *logically monotonic*. In a large database with frequent writes, this property is essential for getting good performance.

These restrictions on the dependent types that can be used in Geolog do not affect the semantic power of the proofs you can write inside a Geolog instance. The restrictions are only on the logic that is used in the definition of database constraints (i.e., the implementation of the proof checker), not on the logic of the proofs that are stored within the database. In fact, Geolog says nothing about what logic is used for the proofs within the database – it only provides the database, and the proof rules are defined by the authors of Geolog theories. But with the appropriate theories, anything that can be proved in Lean can also be proved in Geolog.

Also, the restriction on negation applies only to the language for defining constraints, not to queries in general. It is possible to run arbitrary queries on a database snapshot (an instance of a theory at some point in time) – such queries are allowed to use negation, and they could be written in an existing relational query language such as SQL or Datalog. However, nonmonotonic queries would probably not be incremental: that is, if the database changes, you would have to re-run the query to refresh the results.

I have motivated Geolog in the context of reasoning about traditional programs and their IR, but the language is flexible enough to represent a wide variety of different models of computation and reason about them. It could represent spreadsheets, which are essentially a graph of cells containing formulas that can reference other cells. It could represent string diagrams, petri nets, probabilistic programming languages, circuits, and all sorts of other diagrammatic models. Other creators within the SGAI programme are developing techniques for cyber-physical modelling that could eventually be implemented in Geolog, enabling a much broader range of scientific modelling, engineering, and verification, from microelectronics to supply-chain management.

An exciting aspect of Geolog theories is the way they compose. For example, say you have a theory that represents a particular version of a particular program. You can then define a higher-level theory that allows you to

store many versions of the same program – for example, the result of individual rewrites and optimisation passes that happen in a compiler – and for each transformation from one version to another, it stores enough information that a proof checker can verify semantic equivalence (which rewrite rule was applied to which elements of the program). Each individual version of a program can then reference only things within the same version, but the higher-level theory can make references across versions. This process can be repeated, making ever higher-level abstractions out of lower-level ones. I don't know of any existing database that has a comparably powerful data model.

5 Research roadmap for the physical database

So far, the justifications for the design decisions behind Geolog are mostly hypotheses that are yet to be experimentally validated. Does AI really need parallelisation and human-style division of labour, or can enough be achieved by a single agent by making it really fast? How much do version-control workflows and diagrammatic reasoning actually affect agent performance in writing proofs? We believe that these things will be important, but we should aim to test our hypotheses empirically as soon as possible. Careful experiments will be needed, such as ablation studies that measure the effectiveness of different approaches to proofs, and quantify the impact of different factors.

However, before we can run such experiments, we need an implementation that is sufficiently complete that it can be used for non-trivial example proofs. Our team is therefore pushing towards having working software as soon as possible, even if it is incomplete and slow, so that we can start putting it to use on realistic problems and learn from those applications. I strongly believe that *effective system design requires insights drawn from serious contexts of use* [Matuschak, 2023]. The people developing the theory and the people developing the system need to be in constant close collaboration with the people who are trying to put both to serious use in a realistic application. It is important for these applications to be realistic and demanding, not just toy examples.

The implementation of Geolog is still at an early stage, but it is taking shape. Others have already written a compiler that can parse Geolog theories, typecheck them, and lower them into an IR that consists of relational table definitions and constraints in the form of geometric sequents (very roughly: one conjunctive query implies another conjunctive query). It's not yet complete – for example, it does not yet support computing a least fixed point, such as the transitive closure of a graph (aka an initial model, to use the category theory term) – but the software exists and is sufficiently complete that we can start playing with it. Others are also thinking about how to efficiently check constraints, a problem that is related to incremental Datalog query evaluation.

However, although you can define a Geolog theory for a graph, which will ensure that the data in the database is always a well-formed graph, that does not ensure the resulting system has the *performance characteristics* of a graph database. Getting good performance requires carefully designing the physical

data representation (e.g. indexes), and that requires detailed knowledge about the workload that the database will have to run – i.e., what the size and distribution of the data looks like, and what types of queries will be run frequently. Thus, even though Geolog as a data model is very general-purpose, we need to look at a couple of specific example use cases and datasets in order to be able to start making design decisions about the physical data layout. This is what our team is thinking about now.

5.1 Physical data layout

As an example application, we are developing a Geolog theory for the *static single assignment* (SSA) data structure of MLIR, the multi-level intermediate representation used internally by various compilers. We chose it because it is a general-purpose structural representation of programs that is widely used in practice, which means that it should be easy to get lots of example data (programs) and transformations of that example data (such as rewrites applied by compiler optimisation passes). It is complex enough to test the limits of Geolog, while also being simple enough that implementing it should not be a huge project. But MLIR is just one example, and we would also be happy to see some contrasting examples so that we don't overfit to MLIR. We could look at how Datalog is used for program analysis (since that also requires a relational model representing a program), and maybe even try linking up Geolog with Soufflé, an in-memory Datalog engine often used for program analysis, to see whether we can support the same kinds of reasoning with Geolog.

Once we have an example Geolog theory and some sizeable instances (example datasets) of that theory, we will start working on the physical data representation. Our current hypothesis is that a columnar representation [Stonebraker et al., 2005, Abadi et al., 2013] will probably work well. Automerge already contains a good columnar storage engine in Rust, which we are factoring out into a separate library called Hexane and planning to use for Geolog. Unlike Parquet, Apache Arrow, and many other columnar formats designed for read-only analytics, Hexane also supports efficient updates (inserting values into the middle of a column), which is a key requirement for the Categorical Core. Automerge's new compressed on-disk storage format (Sedimentree), replication protocol (Subduction), and authentication and access control system (Keyhive) are further existing components that we plan to use.

Even with these existing components we will still have to make a lot of design decisions about the physical data layout. For example, the order in which rows are stored in a table makes no semantic difference to Geolog, but it makes a huge difference in terms of the effectiveness of the columnar compression and the performance of different types of queries. Indexes will be required for fast random access, and they will need to be carefully designed too. Perhaps we need optimised physical representations of specific structures, such as roaring bitmaps for sets of integers. How we represent version history will also be crucial: Automerge's OpSet is optimised for text and JSON editing, but Geolog has a more monotonic data model where facts are mostly added, and deletions are

often represented as explicit tombstones at the theory level.

When Geolog stores a large proof (e.g., that the compiled machine code of a program is semantically equivalent to its source code), this will likely take the form of a large diagram (e.g., the MLIR representation of the program) that goes through a long series of transformation steps (e.g., the rewrites applied by the compiler). At each step, most of the diagram remains unchanged, with only a few elements being changed by the rewrite. A naive representation might store a whole separate copy of the diagram for every step, but this would quickly become very inefficient with large diagrams. In order to scale to proofs with a large number of reasoning steps, we will have to map this logical data layout onto a different physical data layout that is semantically equivalent to having separate copies, while offering much better performance characteristics.

This is, in essence, what Automerge already does: it stores the full editing history of a JSON document in a way that scales to a large number of editing steps, each of which makes just a small change to the file (often a single keystroke, in the case of text). It does this by storing just what changed from one version to the next in the form of *operations*, and it can reconstruct exactly what the document looked like at any point in the editing history by applying a visibility filter over the set of operations. However, Automerge is hard-coded to a specific set of operations that are relevant for JSON. We need to generalise this mechanism to allow versioning of any Geolog theory in a way that avoids physically duplicating all the parts of the diagram that have not changed from one version to another, while allowing a higher-level theory to reference elements across different versions.

Maybe we can leverage content addressed storage to detect parts of an instance that are identical, or maybe we can borrow some ideas from persistent data structures [Okasaki, 1999], which use structure-sharing to reduce the overhead of making references to objects immutable. More generally, it is likely that authors of Geolog theories will need to give some sort of hints to the storage engine to help it choose the most suitable physical data layout. In SQL, this is done by declaring indexes and materialised views; what the Geolog equivalent will look like is not yet clear.

5.2 Scaling to large proofs

The hypothetical formalisation of a large open source project like the Linux kernel would likely also be an open, collaborative project. That would allow agents from many different organisations to participate and have their proofs accepted if they can be verified by others. In such a setting, agents may not all be trustworthy, and we want to avoid a dependency on any one trusted party if possible, so that the proof can be a community-run project. Thus, we are designing the Categorical Core to be able to operate in a peer-to-peer fashion, with data replicated directly between participating agents who authenticate each other using cryptographic keypairs. Sync servers can optionally be used, but they are not special or trusted – they only relay data from one agent to another and provide a backup. Subduction and Keyhive already work this way.

If an agent misbehaves and writes invalid data that violates the axioms of a Geolog theory, other agents must reject that invalid data. This will give us *Byzantine fault tolerance* (BFT) – robustness against potentially malicious agents. I have developed a technique for doing this in prior work [Kleppmann, 2022]; the hash graph that this approach depends on is already implemented as part of Automerge. (Automerge as a whole is not yet fully BFT, but I believe that making the Categorical Core BFT is feasible.)

Colleagues at our department in Cambridge maintain formal semantics of the instruction set architecture (ISA) of several CPU vendors, including for multicore concurrency [Armstrong et al., 2024]. To test out Geolog's reasoning capabilities on real software, it might be interesting to import a minimal subset of such a model into Geolog, and to combine it with some critical assembly code (say, the implementation of a locking primitive in the Linux kernel) to see if we can prove the code correct. This would likely be a substantial project, but it would be a prerequisite on the path towards verifying much larger software systems.

As described so far, anyone who wants to check that a Geolog proof is correct needs to be given a *witness*, i.e. a sequence of proof steps that they can validate against the axioms of the appropriate Geolog theory. That is potentially a lot of data. A slightly wild but intriguing idea would be to use *succinct non-interactive argument of knowledge* (SNARK) algorithms to prove cryptographically that a witness for proving a particular statement exists, without having to send the witness itself. There is a lot of recent work on using SNARKs for scaling blockchains, and new algorithms have reduced their computational cost significantly, making them a potentially viable technique for software verification. Some SNARKs are zero-knowledge, although that property is less relevant in this context; what matters more is that they are succinct (i.e., the proof is much smaller than the witness). Verifying SNARK proofs is very efficient: for example, if we had a proof that the Linux kernel was free from vulnerabilities, it would be conceivable that a computer could check the proof as part of its boot process, so that it would refuse to boot an insecure kernel. Unlike current software signing, which only verifies that the software was signed by a particular organisation, this would actually allow users of the software to ensure that they are running secure software.

6 Conclusions

This document started by arguing for the importance of using AI for large-scale formal verification of widely-used computing systems. Whether this is sufficient to significantly reduce the risk of World War 3 is perhaps debatable, but it is hopefully uncontroversial that it would be very beneficial to society if we can drastically reduce the number of security vulnerabilities that currently afflict our critical infrastructure.

Until recently, formal verification was too expensive to contemplate at large scale, but AI is drastically changing the economics of proof. Geolog and the

Categorical Core form a system that is designed to maximise the chances of AI agents successfully verifying the correctness of large systems by enabling many agents to collaborate effectively. I believe that we have an opportunity to harness AI as a force for good by building this foundation and then using it for ambitious verification projects.

References

Daniel J Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013. doi:10.1561/1900000024. URL http://cs-www.cs.yale.edu/homes/dna/papers/abadi-column-stores.pdf.

Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: IMO-level automated theorem proving, October 2025. URL https://arxiv.org/abs/2510.01346.

Anthropic Frontier Red Team. Partnering with Mozilla to improve Firefox's security, March 2026. URL https://www.anthropic.com/news/mozilla-firefox-security.

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Shaked Flur, Kathryn E. Gray, Robert Norton-Wright, Christopher Pulte, and Peter Sewell. The Sail instruction-set semantics specification language, 2024. URL https://alasdair.github.io/.

Eve Bodnia, Yann LeCun, Michael Freedman, Vlad Isenbaev, and Patrick Hillmann. Logical intelligence, 2026. URL https://logicalintelligence.com/.

Nicholas Carlini, Newton Cheng, Keane Lucas, Michael Moore, Milad Nasr, Vinay Prabhushankar, Winnie Xiao Evyatar Ben Asher, Hakeem Angulu, Jackie Bow, Keir Bradwell, Ben Buchanan, Daniel Freeman, Alex Gaynor, Xinyang Ge, Logan Graham, Hasnain Lakhani, Matt McNiece, Adnan Pirzada, Sophia Porter, Andreas Terzis, and Kevin Troy. Assessing claude mythos preview's cybersecurity capabilities, April 2026. URL https://red.anthropic.com/2026/mythos-preview/.

Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang

Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. Seed-Prover: Deep and broad reasoning for automated theorem proving, August 2025. URL https://arxiv.org/abs/2507.23726.

Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970. doi:10.1145/362384.362685.

Leonardo de Moura. When AI writes the world's software, who verifies it?, February 2026. URL https://leodemoura.github.io/blog/2026/02/28/when-ai-writes-the-worlds-software.html.

Oliver Dressler. Lean LSP MCP: Tools for agentic interaction with the Lean theorem prover, March 2025. URL https://github.com/oOo0oOo/lean-lsp-mcp.

Fraunhofer AISEC. Code property graph. URL https://fraunhofer-aisec.github.io/cpg/.

Galois, Inc. The code property graph. URL https://galoisinc.github.io/MATE/cpg.html.

Niels Glodny. Analyzing and evaluating the behavior of Git diff and merge. Master's thesis, Ludwig-Maximilians-Universität München, July 2025. URL https://arxiv.org/abs/2507.22071.

Will Douglas Heaven. OpenAI is throwing everything into building a fully automated researcher. *MIT Technology Review*, March 2026. URL https://www.technologyreview.com/2026/03/20/1134438/openai-is-throwing-everything-into-building-a-fully-automated-researcher/.

Robert Jervis. Cooperation under the security dilemma. *World Politics*, 30(2):167–214, January 1978. doi:10.2307/2009958.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *22nd ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

Martin Kleppmann. Making CRDTs Byzantine fault tolerant. In *9th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC, pages 8–15. ACM, April 2022. doi:10.1145/3517209.3524042.

Martin Kleppmann. Prediction: AI will make formal verification go mainstream, December 2025. URL https://martin.kleppmann.com/2025/12/08/ai-formal-verification.html.

Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward!, page 154–178. ACM, October 2019. doi:10.1145/3359591.3359737.

Math, Inc. OpenGauss: an open source, state of the art autoformalization harness, 2026. URL https://www.math.inc/opengauss.

Andy Matuschak. Effective system design requires insights drawn from serious contexts of use, July 2023. URL https://notes.andymatuschak.org/Effective_system_design_requires_insights_drawn_from_serious_contexts_of_use.

Mistral AI. Leanstral: Open-source foundation for trustworthy vibe-coding, March 2026. URL https://mistral.ai/news/leanstral.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

Thomas Ptacek. Vulnerability research is cooked, March 2026. URL https://sockpuppet.org/blog/2026/03/30/vulnerability-research-is-cooked/.

Z.Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z.F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition, July 2025. URL https://arxiv.org/abs/2504.21801.

James D. Ryseff. The maliciously formed packets of August: Cyberwarfare and the offense-defense balance. Technical report, Center for Strategic & International Studies, September 2017. URL https://csis-website-prod.s3.amazonaws.com/s3fs-public/170907_Ryseff_Cyberwarfare_And_the_Offense_Defense_Balance.pdf?wmiLQuqdILwMEO5YnxfQJY1IA4Ytbp_2=.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS, pages 386–400. Springer, 2011. doi:10.1007/978-3-642-24550-3.

Rebecca Slayton. What is the cyber offense-defense balance? conceptions, causes, and assessment available. *International Security*, 41(3):72–109, January 2017. doi:10.1162/ISEC_a_00267.

Noah Smith. Are we in the foothills of World War 3?, March 2026. URL https://www.noahpinion.blog/p/are-we-in-the-foothills-of-world.

Michael Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth O'Neil, Patrick O'Neil, Alexander Rasin, Nga Tran, and Stanley Zdonik.

C-Store: A column-oriented DBMS. In *31st International Conference on Very Large Data Bases*, VLDB, pages 553–564, 2005. URL http://www.vldb2005.org/program/paper/thu/p553-stonebraker.pdf.

U.S. Department of Defense. 2023 cyber strategy, 2023. URL https://media.defense.gov/2023/Sep/12/2003299076/-1/-1/1/2023_DOD_Cyber_Strategy_Summary.pdf.

Fabian Yamaguchi, Markus Lottmann, Niko Schmidt, Michael Pollmeier, Suchakra Sharma, and Claudiu-Vlad Ursache. Code property graph specification 1.1. URL https://cpg.joern.io/.