djinni.ai White Paper [1.3]

Maciej Chałapuk [1], Ben Peters [2]

ABSTRACT

This white paper outlines the research and development efforts behind djinni.ai's fully automated software development platform, which aims to transform traditional software creation through AI-powered automation. The project addresses the biggest technical hypothesis: whether a multi-agent system (MAS) using large language models (LLMs) can autonomously develop complex software systems through long-running processes while delivering high-quality output.

The R&D journey began with advanced prompting techniques, evolving into a proprietary MAS framework designed to handle the challenges of coding automation. Initial experiments identified error accumulation as a critical issue, which was mitigated by modeling software development as a series of conversations between AI agents. In June 2024, the first success was achieved with a system built entirely by the Coding Daemon, validating the feasibility of the approach.

Subsequent experiments tested various LLMs, with the claude-3-opus model achieving an 82% success rate in meeting software specifications. Metrics for correctness, efficiency, and cost-effectiveness were established, demonstrating up to a 30x reduction in development costs and a 24x increase in efficiency compared to traditional methods. Long term djinni.ai is aiming for five nines (99.999%) success rates.

The R&D efforts continue to focus on scaling the platform's capabilities to handle more complex systems, codifying more nuanced technical knowledge in advanced heuristics, and optimizing infrastructure to achieve convergence times under three minutes. Future work will incorporate open-source LLMs, custom hardware, and proprietary models to improve efficiency and reduce costs further.

1. BACKGROUND

1.1. LARGE LANGUAGE MODELS

In the first two decades of the 21st century, the aim of interactive software was to assist humans in their decision-making processes and optimize the time spent on tasks that did not require human cognitive power. Software developers focused on automating mundane tasks, aggregating information required for making decisions, and presenting it in ways optimized for human perception. The emphasis on "usability" observed over recent years highlights this paradigm.

The invention of large language models (LLMs) has changed the game when it comes to what is possible in software. The early applications of LLM technology have gained popularity faster than anything we've seen before. General-purpose AI assistants (ChatGPT) and AI coding assistants (Github Co-Pilot) are already being used by hundreds of millions. These new tools quickly became our favorites but they are just scratching the surface of what is truly possible.

The cognitive processes of making decisions, previously done only by human beings, can now be automated using general-purpose, artificial intelligence (AI) models. This development is both exciting and very concerning, given its potential to significantly destabilize the job market (Goldman Sachs estimates that AI could replace up to 300 million jobs [5]). It is essential for AI companies to establish and rigorously adhere to ethical principles that lead to development, deployment, and use of AI in a manner that is responsible, fair, and beneficial to society.

Autonomous decision-making represents the most disruptive capability to emerge in the software industry since its inception. Most of the software which was previously interactively used by human beings must be re-thought and re-designed, and many of these systems will be replaced by fully autonomous AI agents. The software industry stands on the cusp of a transformation which is both vast and inevitable. Many industries will be revolutionized, and the nature of human work redefined.

1.2. FULL SOFTWARE AUTOMATION

At djinni.ai, we define full software development automation as a mechanism that can create complex software systems by autonomously executing long-running processes without any human supervision. This level of automation has the potential to radically transform people's relationship with software, as it enables rapid software creation without any technical skills.

We see fully automated software development as one of the biggest challenges in the field of artificial intelligence, and one we are determined to tackle.

The following is a list of different types of knowledge required in the development of robust, commercial-grade software. A mechanism that fully automates software development must incorporate each of these knowledge types.

- **Hard Technical Knowledge** - The mechanism must know the syntactic rules and understand the semantic meaning of the most popular programming languages, know the documentation of software libraries and development tools, understand computer and network architecture, know commonly used file formats, network protocols, as well as various software technologies such as database systems and messaging tools.

- **Planning and Problem Solving** - The mechanism must implement various strategies for planning its work and executing its plan. The planning ability is especially needed when troubleshooting unforeseen problems.

- **Access to Resources** - The mechanism must have access to an integrated development environment as well as cloud resources. It must have the means to create software projects, manipulate their source code, and use software development tools like compilers, linters and testing frameworks. It must be able to create databases as well as virtual machines, and deploy created software onto those cloud resources.

- **Technical Meta-Knowledge** [35] - The mechanism must understand how to efficiently apply technical knowledge to implement commercial-grade systems, which are scalable, performant, and secure in their design, as well as robust and observable in their implementation. Meta-knowledge represents information about practical applications of formalized knowledge, a subtle nuance in skill that comes from experience, is difficult to codify, and is thus likely not found in the training data of today's LLMs. Technical meta-knowledge is one the most important terms used throughout this document.

To ease barriers of entry for users without a technical background, the mechanism should include a conversational UI through which users define the software they need. The conversational AI agent must be able to recognise the level of knowledge of the user and adjust the style of conversation based on it. It must be able to create a clear and unambiguous system specification, all while engaging in the conversation. For the process to be enjoyable, the agent must fill-in many of

the blanks instead of asking the user about every detail. The process must be swift and iterative, enabling the user to quickly move through many versions of their software, specifying more detail in every iteration.

Initial tests indicate that, when compared to traditional engineering teams, our fully automated software development platform will optimize cost of software creation by at least one order of magnitude (10x), and its time by at least two orders of magnitude (100x). With such a platform, users will be able to launch MVPs in days instead of months, allowing them to iterate rapidly, reduce operational costs, and get to a sustainable business model much quicker. We also expect that autonomous software development will positively impact the solopreneur trend, empowering individuals to create custom software for their businesses in a single day.

1.3. CODE AI MARKET

The software development industry is experiencing high growth. According to a report from Grand View Research[6], the global custom software development market was valued at $24 billion in 2021, with projections suggesting that by 2030 it will reach $150 billion. This exponential growth is being driven by the increasing demand for personalized software across various industries. The reported market projections should be seen as pessimistic as they do not account for the disruptive potential of emerging AI.

Amid this growth, the industry faces a critical challenge: an escalating shortage of skilled software developers. Korn Ferry reports that in the US could lose out on $162 billion of annual revenues unless it finds more high-tech workers[7]. This talent deficit is inflating development costs and compromising the quality of custom software solutions.

The advent of LLM technology has prompted an advance in the field of coding assistance which began with AI-based code completion (Github Co-Pilot) and AI-based code generation (ChatGPT). Those early use-cases enable software engineers to almost directly interact with language models and review the generated code in real-time. The experience is akin to having a pair-programmer, effectively doubling productivity according to a study done at Github[8].

Mid 2024, we saw the first AI coding agents developed by Cognition AI and Replit. Those tools push automation of software development one step further, enabling the creation and execution of a multi-step plan, consisting of different coding and maintenance operations. The work done by those agents must still be overseen by a software development professional but with the option, rather than the necessity, to interfere in the decision-making process.

These early coding agents combine hard technical knowledge with planning and problem solving, as well as access to resources. Since they lack technical meta- knowledge, they are far from being able to produce large software systems and commercial-grade code without any human intervention [33]. Oftentimes, these agents make common design mistakes or use resources in a sub-optimal way [34]. The user interfaces of those agents are integrated into software developer environments (IDEs) which can be very intimidating for non-technical users.

Most of the coding AI market produces tools that increase productivity of human software engineers. It is not clear whether any of the major players (Github, Cognition, Replit) will ever upgrade their services into full automation as this could be interpreted as being against the best interest of their userbase. Consequently, these enterprises face a classic innovator's dilemma-type problem. If any of the major players decide to take the step to full automation, they will have to carefully manage the change and execute it over an extended time period.

Current market conditions offer an opportunity for djinni.ai to be the first to target non-technical users with a fully automated software development tool. By implementing an AI-powered, long-running software development process and adding technical meta-knowledge on top of the current capabilities of software development agents, we will provide a platform for rapid creation of reliable custom software.

2. CONSIDERATIONS

2.1. LIMITED META-KNOWLEDGE IN LARGE LANGUAGE MODELS

Although LLMs have made tremendous strides in coding proficiency by being trained primarily on human-generated, open-source software, they are approaching the limits of what can be achieved using this data. Open-source projects offer extensive hard technical knowledge, such as syntax and semantics of programming languages, as well as common design patterns, but they fall short when it comes to technical meta-knowledge. This deeper understanding—how to build scalable, secure, and performant systems—cannot be fully learned from open-source code alone. Meta-knowledge is often implicit, built through years of experience in complex engineering environments, and LLMs have not yet demon-

strated the ability to acquire or apply this kind of knowledge in a consistent manner.

To address this challenge, the solution must employ a set of rules which add meta-knowledge on top of hard technical knowledge contained in the language models. The solution must orchestrate how knowledge is applied in different language model invocations, involving only parts of meta-knowledge which match the currently executed software development processes.

2.2. LIMITED REASONING CAPABILITY OF LARGE LANGUAGE MODELS

While LLMs have shown exceptional proficiency in generating human-like text and writing code snippets, they often struggle with applying complex logical reasoning. This limitation becomes evident in scenarios that require the understanding and manipulation of intricate relationships, multi-step problem-solving, or abstract reasoning across large systems. LLMs are predominantly trained on vast amounts of human-generated data, which enables them to mimic patterns and structures in the code, but their reasoning abilities are still shallow compared to those of experienced human engineers.

To improve the LLM-based decision-making, complex tasks can be broken down into smaller, more manageable sub-decisions. Focusing on incremental steps rather than attempting to reason through the entire problem at once increases the probability of achieving an optimal or near-optimal outcome. Implementing such structured decision-making schemes significantly enhances the ability of LLMs to handle complex, multi-faceted software development tasks, leading to more robust and reliable solutions.

Historically, the reasoning capabilities of LLMs have seen considerable advancement with each new generation. The leap from GPT-3 to GPT-4 was especially significant, with GPT-4 demonstrating a much greater ability to handle complex tasks. Further refinement came with OpenAI's GPT-4o, which is specifically designed to enhance logical reasoning. Looking ahead, we expect to see more LLMs focusing on reasoning capabilities, with models designed explicitly to handle complex, multi-step decision-making processes, similar to OpenAI's o1 model. These developments suggest that LLMs will continue to improve in areas where they have traditionally struggled, reducing the need for building complex decision making schemes as workarounds for their lack of more advanced reasoning capabilities.

3. DJINNI.AI

The traditional approach to development of custom software is resource-intensive, risky, and often exclusive to larger enterprises with the capital to absorb the costs. Both high development expenses and technical complexity can pose a significant barrier to many small businesses, solopreneurs, and individuals. As a result, custom software—despite its clear advantages—remains inaccessible to many, reinforcing the digital divide and limiting innovation at the grassroots level.

This is where djinni.ai aims to offer a solution. By automating the entire software development process, djinni.ai seeks to democratize access to high-quality, custom software, eliminating the need for extensive technical skills or large budgets. By leveraging the latest advancements in AI, our platform is expected to reduce the inefficiencies and risks inherent in the traditional approach to software development. djinni.ai strives to empower individuals and small businesses to rapidly create custom software that is scalable, robust, and tailored to their specific needs.

The following section describes our long-term vision for djinni.ai software development platform.

3.1. FUNCTIONALITY

At the heart of djinni.ai's offering is our fully automated software development platform, designed to streamline the entire process from idea to deployment.

The journey begins by gathering the user's requirements through an intuitive conversation with a competent AI assistant. The conversational agent is sophisticated enough to adjust its interaction based on the user's level of technical knowledge, ensuring that both experienced developers and non-technical users alike can easily engage with the product. The AI assistant works to build a clear and unambiguous software specification, filling in gaps where necessary, allowing users to communicate their needs without getting bogged down in technical details.

Once the specifications are in place, the platform will seamlessly transition into coding and deployment, both of which are handled entirely by AI. The process is designed to be fast and efficient, without the need for any manual coding or configuration. Within minutes, new, high-quality software will be written, tested, deployed on cloud infrastructure, and ready to be used.

djinni.ai will automatically conduct acceptance testing, which verifies that the software meets the defined specifications and

that it functions as expected. Following the acceptance tests, the user will be invited to manually review and verify the final product, making any adjustments or providing feedback for further iterations. The platform will also offer autonomous site reliability engineering services (SRE) with AI-driven daemons continuously monitoring all deployed systems, detecting runtime issues, and resolving them in real-time, without any user intervention.

The entirety of the process, from gathering requirements to testing and deployment is expected to average at 30 minutes per iteration. This rapid turnaround will allow for swift revisions and adjustments, enabling users to refine their software with ease and efficiency.

4. PROOF OF CONCEPT

Djinni.ai's proof of concept (PoC) includes a conversational UI for project scoping (Chat UI) and a coding mechanism that implements a proprietary software development process, knowledge base of patterns and antipatterns, and decision making heuristics (Coding Daemon). We were able to extensively test our PoC using a specification of a simple software system. This section describes the scope of the experiment and its results in detail.

4.1. BIGGEST TECHNICAL HYPOTHESIS

When djinni.ai was founded (September 2023), we were uncertain whether it was possible to effectively use LLM technology for the complex, long-running task of coding an entire software system. It wasn't clear whether an LLM-based long-running process would reach its stop condition. We identified this as the biggest technical hypothesis connected with djinni.ai.

After extensively researching the field of AI, we identified multi-agent systems (MAS) as one of the best paradigms to use for implementing djinni.ai. We expanded our biggest technical hypothesis to include that an LLM-based multi-agent system (MAS), when executing a long-running software development process, would consistently reach its stop condition while delivering high-quality output.

4.2. ADDITIONAL HYPOTHESES

We aim to reduce the costs of software development for our customers by an order of magnitude (10x). In order to achieve profit margins close to 90%, the cost of the development process must be reduced by at least two orders of magnitude (100x). We also hypothesized that automating the whole development process will decrease the time needed to produce software by at least two orders of magnitude (100x).

When djinni.ai was founded, GPT-4 was the leading LLM. While it showed great potential, it was uncertain whether its reasoning abilities could handle fully automated software development processes. With new versions of large language models consistently showing improved performance, we hypothesized that upgrading the underlying model would enhance the decision-making and planning abilities to the levels required for automated coding.

We aim to introduce a failover mechanism that automatically switches the underlying LLM used by the Coding Daemon to a different provider; removing a single point of failure from the djinni.ai platform and reducing the risk of depending on language models provided by a single third-party. Given this we hypothesized that outputs of flagship LLMs of different providers would be similar enough for us to build a well-defined abstraction layer for seamlessly changing LLM providers.

4.3. RESEARCH AND DEVELOPMENT

Our first attempts at designing a fully automated software development process (September, 2023) were based on advanced prompting techniques (tree-of-thought, re-act) but turned out to be extremely complex. This led us to start looking for a different approach, one that would better manage the complexity of the task by introducing multiple levels of abstraction.

We estimated that a fully automated software development process would contain a chain of several hundreds of LLM calls. Our experiences with other LLM-based systems (ChatGPT) suggested that the biggest problem to solve in such a long-running process would be the accumulation of errors between different LLM calls. Since it is impossible to prevent LLMs from making errors, we needed a solution that would minimize the number of errors propagated through the process.

After researching the space of AI, we concluded that modeling processes using conversations between AI agents would have the potential to greatly simplify our approach to automated coding and increase the quality of the output by stopping error propagation. Seeing the potential to solve both of our problems—complexity management and error accumulation—we decided to build our own, commercial-grade framework for writing multi-agent systems (MAS) and developed it over the last quarter of 2023.

In January, 2024, we started working on the Coding Daemon using LLMs from OpenAI. The daemon was designed to be a multi-agent system that automates the coding process, using a specification of a software system as its input. We focused on implementing a multi-phase coding scheme, including requirements analysis, source code design, implementation, testing, and integration.

June 11th, 2024 marked a pivotal moment in our R&D efforts, as we witnessed the first implementation of a software system, exactly matching its input specification, created by our Coding Daemon. That same month, we designed correctness metrics for the generated code and added support for the language models provided by Anthropic. In the first week of July, we extensively tested our solution using LLMs of multiple providers and measured their correctness, efficiency, and cost-effectiveness for the first time.

4.4. CORRECTNESS METRICS

As part of our R&D efforts, we developed a way to measure correctness of the software systems produced by djinni.ai's Coding Daemon. Each run of the Coding Daemon ends with one of four states, which translates to one of four correctness metrics.

- **Failure**: Occurs when the coding machine is unable to complete the software creation process. The frequency of incomplete runs is referred to as Failure Rate.

- **Convergence**: The state where the process of creating the software system finished without an error. A converged run doesn't guarantee that the produced software meets the input specification. The frequency of completed runs is referred to as Convergence Rate. High Convergence Rate is crucial for ensuring the system can handle long-running, complex processes with consistency.

- **Success**: The state where the process finished, the output code matches input specification, and the code works. All successful runs are also converged. The frequency of runs resulting in code matching the specs is referred to as Success Rate. It is the primary metric for correctness of the systems created by the Coding Daemon.

- **Perfection**: The state where all architectural decisions were correct and the Coding Daemon did not produce any dead code. All perfect runs are also successful. The frequency of optimal code creation is referred to as Perfect Rate.

4.5. EXPERIMENT

The specification used for testing described a very simple software system for tracking expenditures. The specification contained only two features—adding an expenditure and calculation of total spending—and didn't specify any non-functional requirements beyond the programming language. All the runs of the Coding Daemon included in the experiment used the same input specification.

The experiment included 11 runs of the Coding Daemon using the claude-3-opus model, 5 runs using the gpt-4o model, and 5 runs using the gpt-4-turbo model. The software systems resulting from each run of the Coding Daemon were assigned to one of four ending states (failure, convergence, success, perfection) after a careful evaluation done by a qualified software development professional.

4.6. OBSERVATIONS

4.6.1. CORRECTNESS

As illustrated in Figure 10, large differences in success rate of the Coding Daemon can be observed when changing the underlying language model. We found that running the daemon on claude-3-opus model results in 82% success rate which is the highest out of all the tested models.

A run was considered failed if a critical error was raised during its execution. Runs which did not fail were considered convergent. Convergent runs which produced a software system matching the specification with integration tests covering its whole functionality were considered successful. Successful runs which produced no dead code and exactly followed the meta-knowledge included in the Coding Daemon were considered perfect.

Model	Failure Rate	Convergence Rate	Success Rate	Perfect Rate
claude-3-opus	18 %	82 %	82 %	0 %
gpt-4o	20 %	80 %	40 %	0 %
gpt-4-turbo	100 %	0 %	0 %	0 %

Figure 10: Correctness metrics

With the success rate of 82%, if we run the same job simultaneously on three instances of the Coding Daemon, the probability of getting at least one success will be 99.4%. We interpret those numbers as reaching high predictability of the coding process. Our findings validate our biggest technical hypothesis, which significantly reduces the techno-

logical risk associated with the development of the djinni.ai platform, and effectively gives us our PoC.

What's important to point out is that the perfect rate of 0% is consistent with human performance. Human developers often make suboptimal decisions, some of them conscious, some not, accumulation of which is referred to as technical debt. Our analysis suggests that by adding more decision-making schemes to the Coding Daemon, we'll be able to increase perfect rates to very high levels, completely outperforming human developers.

Since the Coding Daemon used in the experiment contains very little meta-knowledge, the observed levels of correctness should be seen as very promising. Long-term, we'll be aiming at 99.999% (five nines) target success rate of the Coding Daemon, as well as 99% (two nines) target perfect rate.

4.6.2. EFFICIENCY

The efficiency metric we use for the Coding Daemon is convergence time—the time between the Coding Daemon receiving a request containing a system specification and fully implementing a software system described by said input specification. Since the Coding Daemon has spent most of its time waiting for responses from the LLMs, the average time per LLM call and the average time per input/output token were approximated by dividing the convergence time by the average number of LLM calls in a convergent run.

As illustrated in Figure 11, the convergence time for the claude-3-opus model averages a little above one hour. Assuming that an average software developer could create the same code (~750 lines) in 24 hours, we have observed a 24x increase in efficiency compared to human developers. Although the assumption of 100x better efficiency was not confirmed by this experiment, we see this assumption as safe.

Our observations suggest that extended LLM response times are due to rate limits, rather than our provider's efficiency limits. In our assessment, custom rate limits have the potential to increase the efficiency of the Coding Demon by at least an order of magnitude (10x). We expect to achieve a similar increase in efficiency by running open source language models on infrastructure under our full control. In our assessment, possible optimizations of token consumption in the Coding Daemon will provide additional 5x efficiency improvement.

Model	AVG Convergence Time	AVG Time / LLM Call	AVG Time / IO Token
claude-3-opus	64 min	15.42 sec	18 millisec
gpt-4o	38 min	10.45 sec	17 millisec

Figure 11: Efficiency metrics

Several well-funded startups are currently developing ASIC (Application-Specific Integrated Circuit) chips, which are projected to be up to 20x faster than traditional GPUs for specific AI workloads. Utilizing these chips could lead to additional 20x improvement in efficiency, drastically reducing our computation times. Leaders in this space include Groq [30], SambaNova [31], and Cerebras [32], whose chips are specifically designed to accelerate AI inference.

Long-term, we aim for convergence time measured on the "Spending Tracker" specification to be less than 3 minutes.

4.6.3. COST-EFFECTIVENESS

Figure 12 illustrates differences in the cost of running the Coding Daemon using various language models. Since we did not observe any convergent runs using the gpt-4-turbo model, it is not included in the table.

Model	AVG Input Tokens / Run	AVG Output Tokens / Run	AVG Cost / Run
claude-3-opus	1,923,775	117,889	$37.70
gpt-4o	1,301,249	22,387	$21.20

Figure 12: Cost-effectiveness metrics

The cost of convergence when using the claude-3-opus model averaged at $37.70. Assuming the average annual salary of software developer to be $100,000 ($384.61 per day) and that it would take them 3 workdays (24 hours) to implement the same system (~750 lines of code), we have observed a 30x decrease in cost when compared to traditional software development. The assumption of a 100x decrease in cost was not confirmed by this experiment but the observed levels would already result in good profit margins (50%), should our assumptions about pricing be achievable.

We assess that running open source models on infrastructure under our control has the potential to reduce costs by one order of magnitude (10x). Additionally, optimizations in token consumption of the Coding Daemon will provide additional 5x cost reduction.

The total cost of the experiment amounts to $613.67. To reduce the cost of the experiment, we decided to focus on

testing the language model which manifested the highest correctness rates, which resulted in an uneven number of runs between different models. We aim to track the correctness and time metrics across all internal releases of the Coding Daemon. We expect the cost of tracking those metrics to increase significantly as we add new, more complex specifications to the benchmark.

4.6.4. MISCELLANEOUS

Changing the provider of the underlying language model did not require additional changes to the Coding Daemon which validated one of our additional hypotheses.

Our findings suggest that newer versions of LLMs perform better not only at simple tasks and in prompting technique benchmarks but also in complex, long-running processes containing hundreds of LLM calls.

4.7. CONCLUSION

Through this PoC, we achieved an 82% success rate with the claude-3-opus model, a milestone that significantly reduces the technical risks associated with developing a fully autonomous software development platform. While there remains a gap between the current efficiency and cost-effectiveness metrics and our long-term goals, we are confident that these will be bridged as we continue to optimize the platform's architecture, introduce proprietary expert models, and gain better control over infrastructure.

Looking ahead, our focus will be on extending the scope of the Coding Daemon to handle more complex software systems while increasing its ability to make optimal decisions by incorporating deeper technical meta-knowledge. We anticipate significant improvements in both efficiency and cost reduction as we introduce token consumption optimizations, open-source LLMs, and ASIC chips optimized for AI inference.

Our findings strongly suggest that a multi-agent system powered by large language models can autonomously and consistently generate fully functioning software systems. Our biggest technical hypothesis has been validated.

ACKNOWLEDGEMENTS

Many thanks to Tomasz Fortuna for providing consulting in many technical subjects mentioned in this document and for just the best attitude towards helping people.

Our profound gratitude to Bogusław Kluge for pointing us in a very good direction in the latter stages of the PoC.

Special thanks to Filip Victor for motivating the creation of this whitepaper and for its review.

REFERENCES

[1] Maciej Chałapuk <maciej@djinni.ai>
[2] Ben Peters <ben@djinni.ai

[5] BBC: AI could replace equivalent of 300 million jobs
https://www.bbc.com/news/technology-65102150

[6] Custom Software Development Market Size Report
https://www.grandviewresearch.com/industry-analysis/custom-software-development-market-report

[7] KornFerry: The $8.5 Trillion Talent Shortage
https://www.kornferry.com/insights/this-week-in-leadership/talent-crunch-future-of-work

[8] Github: Copilot's impact on productivity and happiness
https://github.blog/news-insights/research/research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[14] Github Copilot - The most widely adopted AI dev tool
https://github.com/features/copilot

[17] Replit: Build software faster
https://replit.com/

[30] Groq is Fast AI Inference
https://groq.com/

[31] SambaNova: The World's Fastest AI Inference
https://sambanova.ai/

[32] Cerebras Inference: 20x Faster than GPUs
https://cerebras.ai/

[35] University of Michigan: Meta-Knowledge
https://web.archive.org/web/20051119163215/http://ai.eecs.umich.edu/cogarch0/common/prop/metaknow.html