Zero-Shot Learning - Episode 6: Maxim Fateev from Temporal

Distributed systems in disguise

Hosts: Nancy Wang, Chief Technology Officer, 1Password Jeff Malnick, Vice President and General Manager of Developer and AI, 1Password Guest: Maxim Fateev, Co-Founder and CTO, Temporal

---

Intro

**Nancy Wang:** Hey everyone, welcome to Zero-Shot Learning, a podcast about the reality of developing with AI from the people actually doing the work. I'm Nancy Wang and I'm the Chief Technology Officer at 1Password. Normally, I co-host this show with Dev Tagare, who's a Senior Director and Head of Engineering for Gemini Enterprise and Business at Google.

But today I'm joined by our very own Jeff Malnick, who is the Vice President and General Manager of Developer and AI at 1Password. Our guest is Maxim Fateev, who is the co-founder and CTO of Temporal. Before Temporal, Maxim was a tech lead at Amazon, where he helped break their monolith architecture into microservices and co-authored the AWS Simple Workflow Service.

He then went to Uber, where he co-created Cadence, an open-source workflow orchestration engine. Maxim is widely recognized as the pioneer of durable execution, which was a revolutionary shift in software development. Before, if a server were to crash mid-task, the application state would be lost and you had to build your own recovery logic or even start over.

Durable execution preserves the full execution state automatically, so complex tasks resume exactly where they left off. Maxim built Temporal to bring that property to the broader engineering community. Temporal is empowering global engineering teams to build highly reliable distributed systems.

Their open-source orchestration platform has become the backbone for AI development, serving as a foundational execution layer that allows complex, multi-step, autonomous AI agents to reason, persist, and run reliably without state loss. This is a super fun episode. Stick around after the interview for a demo of their chatbot agent generating code with different backends abstracted away - switching between Python and Docker, from local to remote environments. You really have to see this, so lock in. This is Zero-Shot Learning.

---

1. The Origins of Temporal: From Amazon to Durable Execution

**Nancy Wang:** Well. Hello everybody. Welcome to the Zero-Shot Learning podcast, where we have the pleasure of sitting down with some of the most cutting-edge builders in the AI world today.

**Jeff Malnick:** Hey, everyone. I'm Jeff. I'm the Vice President and GM of Developer and AI at 1Password. I've been at 1Password for about five months. Previously, I was leading engineering at HashiCorp for the Vault and Boundary products.

**Nancy Wang:** Awesome. We're joined here by Maxim Fateev today, the co-founder and CTO of Temporal. Welcome, Maxim. And I just want to say - today, as more enterprises are deploying agents in production, it's not just a simple one-task or chatbot function anymore. We're now expecting agents to do long-running workflows - things like, for example, even what we're building internally at 1Password, an AI SRE agent that needs to be able to understand state, store state, and do retries. And in that world, something that you've always talked about is that agents are bringing back distributed systems. Tell us more about how you got the inspiration to start Temporal - maybe drawing from your Amazon as well as Uber days.

**Maxim Fateev:** Yeah. I grew up at Amazon as an engineer and led quite a few projects. The most significant one - at least the initial big one - was a platform we called the Amazon Messaging Platform. It was a broker-based architecture, created around the beginning of 2000 - well before even Kafka and all these technologies were conceived. The entire backend of Amazon ran on that. And later, when it appeared that Simple Queue Service needed a storage engine, they took that and ran with it. At least for the first 10 or 15 years they were using that engine.

As a tech lead practically responsible for all of Amazon's messaging, I talked to every team, because Amazon was one of the first companies doing services at large scale. They had initially started with a monolith - kind of amazing, actually. You could run the whole Amazon website on your desktop. The catch was that just linking that binary took up to 45 minutes on a high-end developer desktop, and a full build was around 18 hours. So breaking the main line was a big deal - we had wiki pages dedicated to that, and it was taken seriously.

So we did what everyone is doing now: we moved to an event-driven architecture for the back end. As I was talking to all those teams, it quickly became clear that pub-sub is just not the right way to stitch together complex systems. Amazon always relied on orchestration, and initially had an orchestrator on top of Oracle using a kind of Petri-net-based approach. Then they needed another one, and my team decided to work on a new orchestrator. There were multiple attempts, and one ended up being the first public version of the Simple Workflow Service. I was deeply involved in that, and Samar - my co-founder who is now CEO of Temporal - was working with me on it.

Initially we were very focused on the backend part - how to do orchestration, how to store state. We tried all the traditional approaches: graph-based, and other permutations of that. At some point I just didn't like them because they're always very verbose and hard to program and troubleshoot. Then it took a few iterations to realize that you can actually just write code and preserve the full state of normal code execution - not some intermediate representation like an abstract syntax tree or a graph. That was a kind of big revelation. And that is how the idea we now call durable execution was born.

**Jeff Malnick:** But you say that was like a principle or recurring architectural theme that kept coming up over time?

**Maxim Fateev:** It seems that when you have event-driven architecture, you still implement workflows. If you still have sequences of steps, you still need to execute operations based on whatever external outputs - so you have a flow. Not a fixed flow - it can be a super dynamic flow - but it is a flow. It's all about calling services in a certain order and reacting to events. And when you use event-driven architecture, you practically end up in a situation where this logic is kind of spread out everywhere and there are no real APIs.

Because what is the API of a service that consumes ten types of messages and produces 25 types of messages? The API is a state machine. Usually the state machine is not very well defined, so there's no API there.

---

2. Why Event-Driven Architecture Falls Short

**Nancy Wang:** That's actually a great segue to talk about what are some examples of coupling and cohesion. On the backend side, that's something our principal engineers talk about a lot when designing systems - which ones do you want loosely coupled? Maybe give us some examples.

**Maxim Fateev:** Okay. Loosely coupled is always the right goal where possible. But the problem with event-driven systems is that there is a big confusion between runtime and design time. They are absolutely loosely coupled at runtime - that's the whole idea. A synchronous system is so stateful that if anything fails, the whole system slows down, errors propagate, and so on. An event-driven system allows users to keep functioning even if other services are not functioning, because there are queues in the middle. So your systems are loosely coupled in production. That is very useful.

The problem is people make the assumption that they are also loosely coupled at design time - which is not true. The moment you change the format of a message in a system like that, you don't even know which services can be broken by your change. In a lot of cases, it's not even informative - maybe the order of the message changed or something else subtle. So these messages are flying around and you can change anything and things can break at any time.

Let me give you an analogy from another domain to make it more concrete. Imagine your engineer says, "our service is too tightly coupled - I want to make it more loosely coupled." So they go and change all variables to global variables. Why? Because now any new piece of code can just read those variables, and the other pieces of code don't need to be changed. But this is exactly the logic people use when they say "I can add a new service in the event-driven world." You can read those variables, but it doesn't mean that it's loosely coupled. Events are the global variables of distributed systems.

**Jeff Malnick:** One of the things I'm hearing you talk about is discovering these new workflows and figuring out how to get engineers to engage with distributed systems. It sounds like you were thinking deeply about developer experience - going from pub-sub architectures to a workflow-based system in code. Talk a little bit about how that thinking evolved.

---

3. Developer Experience and the Case for Durable Execution

**Maxim Fateev:** It's clear that we want orchestration for a lot of these systems. The problem always was that you either use your code and your tools, or you go into an orchestrator, which is usually some sort of state machine with some sort of domain-specific language. And unfortunately, it's not actually domain-specific - which is a huge confusion - because the whole definition of a domain-specific language means it's tied to a very specific domain. But workflows are a very general-purpose thing. Your infrastructure deployment is a workflow. Your agent is a workflow, even a super dynamic one. Your business transaction is a workflow. Your data pipeline is a workflow. Almost everything is a workflow.

So what happens is people end up trying to create a general-purpose programming language in JSON or XML or YAML or visual diagrams. And it always ends up as a bad contraption, because by definition there is a reason we have top-level programming languages. If JSON were better, we would just program the Linux kernel in JSON. But for some reason people say, "I want a JSON or DAG for workflows, but not for anything else."

The whole idea of durable execution is that you don't need any other language. You can just use Java, Python, TypeScript, whatever you normally use. And this is a little different from systems like Airflow, which says "workflows as code." Airflow code generates a DAG, and what actually executes is the DAG. For data pipelines, DAGs can be very reasonable - they're a domain-specific approach for that domain. But the moment people take Airflow and try to use something dynamic, they discover they cannot add a node at runtime. So they generate the DAG at runtime, and then it becomes a mess.

With durable execution, you can use Java directly and run normal code, and make that code durable. Durable means that this code cannot fail at all. You call an API, and the process crashes - the reply to that API call can come three days later. You resurrect that process in exactly the same state, still blocked on the same API call. Give it the result. It keeps running as if nothing happened.

**Nancy Wang:** So that's a checkpoint, right?

**Maxim Fateev:** It is checkpointed, but from a developer point of view it's not a checkpoint. From your point of view, your code just blocked for three days. Behind the scenes, how it's implemented - there are different ways. Checkpoint is not the only one. We actually don't use it.

**Nancy Wang:** Oh - tell us more about that.

**Maxim Fateev:** We use replay. We practically record the results of all external operations - side effects. And then when you need to resurrect that function, we just give it those results back. Of course it's deterministic and ends up in exactly the same state. We don't use OS-level memory checkpointing - you technically can checkpoint code, but you need to go pretty deep into the operating system memory. What we do is at the library level - for Java or Go, for example - we don't change the runtime at all, we don't own the runtime. It is just a normal library. And you cannot, at a library level, ask the JVM to snapshot something that is blocked on a synchronous API call.

**Nancy Wang:** It's actually a really clever way of solving it. So I spent about ten years designing backup software - a system that deals with a lot of what we just talked about: the ability to retry, checkpoint more on the logic side, knowing when to take a snapshot of an EBS volume, where to store it, when to hydrate it. In that system, I'm curious - drawing from your Amazon experience - what were you specifically looking to improve with Temporal?

---

4. What Temporal Improves: Scale and Developer Experience

**Maxim Fateev:** Our goal was to improve developer experience. When we started, the alternatives were: one, use traditional methods - pub-sub, databases, cron jobs, stitch all these things together. Or two, use traditional workflow engines, which had two problems. One is that developer experience was based on those configuration languages which stop working after a certain complexity. The probably well-known example is BPMN. There is a standard, and it has been around for 20 years. The reason is that the promise is actually very good - non-technical people can write these workflows via pictures, and everything magically runs. The result was that non-technical people actually cannot write them in a way that's production-executable. Developers have to deal with that. And the problem is that the picture only represents a sequence of steps, not data - variables, data mapping, inputs, outputs, state management, data structures. None of those visual languages represent any of that well.

**Jeff Malnick:** It's sort of like you still need to read between the lines as a systems engineer. You still need a lot of domain expertise.

**Maxim Fateev:** Yeah. And a lot of lower technical details. Most of them don't have strongly typed APIs - they have just a notion of a task. Even BPMN, if it calls a task, the task gets mapped back to a HashMap. Any task can update that HashMap. So it's practically a global variables map. Good luck figuring out which task updated a variable when something breaks - you're not going to see that in the diagram.

So practically going back to what we were trying to solve: developer experience and scale. We spent a lot of time on the backend service that scales practically infinitely and provides the capabilities required for that. But at the same time, we iterated on developer experience and tried to come up with the best approach. The first attempt was okay - which is why most people still don't know about Simple Workflow Service. It wasn't good enough to gain steam.

**Nancy Wang:** I do remember it from the AWS days. That's why I was so excited about Temporal.

**Maxim Fateev:** I don't think it's a secret, but when I was at AWS, Simple Workflow Service was one of the most widely used services internally within AWS by other services.

**Nancy Wang:** That is correct. We were a heavy consumer.

**Maxim Fateev:** And besides, it was very, very rudimentary. It was practically a proof of concept. It lacked a lot of features. Performance was abysmal because it had to replay the whole execution history on every step - so it was N-squared in performance. We knew how to fix that, but for various non-technical reasons the team went its own ways. I think the service is still frozen in the state we built it 15 years ago. And then my understanding is that AWS decided to go the DSL route and built Step Functions. I've already said what I think about those types of applications.

---

5. Agentic Workflows as a Distributed Systems Problem

**Nancy Wang:** So maybe then pivoting the conversation to agentic workflows - we firmly believe, with you, that everything becomes a systems problem at the end of the day. In the world of building agents and long-running workflows, walk us through how you think about when failure modes happen. Is it with the model or is it with the agent logic? And how do you discover that?

**Maxim Fateev:** I think you need to have a system that works end to end. But if you think about how all these applications started - and are still mostly in that mode, like Claude Code or Codex CLI - they're just in-memory programs making synchronous RPC calls. It kind of works if you have just one application running on your machine making those calls. But the moment you want a lot of them, or you want them to run longer, things change completely.

Think about it: you want them to complete. You don't want them to just go and die in the middle and lose everything. So you say "I want this agent to survive failures, even just process restarts." But then you call a tool, and the tool is not available. You need to retry. The moment you start retrying, a short call becomes a long call. You can decide how long you want to retry - 5 seconds or 5 hours - but conceptually, the moment you went from a short synchronous invocation to "I need this to complete," you've magically made it long-running. And then a lot of things start happening.

If you have 10,000 of those agents all hitting that tool at the same time, you'll overload it. So you need exponential backoff. But even with exponential backoff, many callers can still overload a service - so you need flow control. Once you have flow control, you need a queue. And then you need rate limiting. All of these things start adding up, even if you start with simple synchronous processes.

And then these processes can be long-running and need to recover their state. Snapshots alone aren't enough. Some agentic frameworks just provide snapshots, and people think that's enough - then they realize that if a process crashes, a snapshot doesn't help because you need to first discover that the process crashed. You need a heartbeat or some monitoring system. And if your process can handle not one agent but 10,000 agents, how do you find which agents should be recovered? All these things stack up, and I think people who started from "synchronous Python in-memory program" are learning the hard way that these are not simple problems.

**Nancy Wang:** Yeah. You built a swarm of agents as an example - for instance, to debug production databases. Can you talk about that, and given what Maxim just walked through, how do you know which agent failed and how do you retry?

---

6. Bootstrapping Agent Identity at Scale

**Jeff Malnick:** You know, my background and domain expertise have been a lot around identity and security. One of the interesting things with agent swarms, when we were trying to get this system off the ground, was: how do we actually bootstrap identity for a thousand agents that are all going to be working toward a single mission? That's not necessarily an easy problem to solve. You've got a thousand agents spinning up, they all want their identities enrolled, and they want to be given a token so they can go access tool calls through an MCP server.

The initial bootstrapping of that system - getting all those agents their identities and making sure they're all authorized to do the correct thing on behalf of some human or some workflow - was actually pretty challenging. What we decided was that they'd all spin up with their own ephemeral private keys. And we used a notion of a very loose affiliation through something called a verifiable credential to solve that problem. That reduced the performance implications and the round-tripping that would be required for a more federated infrastructure where they'd have to get identity through some other system.

We used workload attestation in that case to verify that the private key is being emitted from this specific system, this Kubernetes cluster - and we can say with a pretty good degree of confidence that this is a legitimate agent. So the policy we put around that station would be X.

I think in a lot of these systems today there are a lot of different ways that people are solving the problems around identity and security. And it's interesting hearing you talk about distributed systems problems in this space, because identity security is also a distributed systems problem. You have a lot of entities that you need to get information to - an identity document, at the end of the day - and you need to get it to that workload in a very secure fashion.

One recent approach I saw was through a client profile mechanism - you have a thousand different agents running behind one single client, and that client would be given the authorization to do something on behalf of all of them. That client could be a gateway. And I think the key point is: the level of productivity you get out of these agent swarms is very proportional to how long they can run without a human in the loop. So coming up with headless capabilities for reauthorization and authentication - or ways to notify a human when you do need to pull them in the loop - are some really interesting problems to solve right now.

---

7. Long-Running Workflows and Permanent State

**Nancy Wang:** And actually, that reminds me of one of the first conversations that you and I had, Maxim - in this world where agents are doing multi-step workflows and might even need to talk to other agents, you mentioned that some of these workflows could last as long as a year or a couple of months.

**Maxim Fateev:** They can be permanent.

**Nancy Wang:** Yeah.

**Maxim Fateev:** Like, if you think about something like Claude and these types of personal assistant agents - the idea is that they are running forever. But they certainly originate from a distributed systems point of view where that level of durability wasn't the design goal. And I know people usually don't want to be slowed down by building resiliency in from the start. That's why the message from our conference was that right now, teams have a choice. They either don't do resiliency, prioritize innovation, and hope that customers will tolerate the failures - or they invest time in reliability and risk falling behind. I had friends using these agents who said they needed Temporal because the agents were failing all the time and hanging.

And again, that's in a relatively contained environment, but I cannot imagine anyone putting that in real production at scale. The promise of Temporal is: move as fast as without resiliency, but get resiliency out of the box at the platform level. That is, I think, why we see so much adoption in the agentic space. People building startups just use Temporal. We don't have any specific AI framework, we don't have "AI" in our name - and we are probably one of the most widely used infrastructure companies in this space, obviously outside of the major labs themselves. Because we solve exactly the problem.

**Jeff Malnick:** It's pretty interesting. I want to say it was in February when Cursor unleashed their own version of an agent swarm to go build a web browser. Did you read about that?

**Maxim Fateev:** Yeah, I think I did. I know Cursor uses Temporal, but I don't remember the exact use case. It could be a different one.

**Jeff Malnick:** They were observing the agents coming to different conclusions about how to build this thing - thousands of agents essentially experimenting together. And you have to think architecturally: what does it take to get this working with a distributed system of essentially machine workloads? If you can teach a system to experiment on behalf of systems engineers, it

becomes very challenging to design for what that system might need next - because the experiment could result in a new escalation or access to a new system. That was one of the things that really intrigued me about the access management problem with these sorts of systems.

---

8. Retry Logic, Guardrails, and Error Handling

**Nancy Wang:** So maybe a question specific to agents - in traditional systems engineering, if it fails, you just keep retrying and you can put in logic about retry limits or windows. For agents, what's the most dangerous thing about retries in an agentic workflow?

**Maxim Fateev:** I think the big question is exactly at which point you stop. Normal retries, implemented properly with flow control and exponential backoff, are not that dangerous. Usually, humans decide how long the system should wait before it escalates - fails, contacts a human, or raises an alarm. Usually there's an expectation that you write your business logic and it doesn't contain a lot of error-handling code for the routine cases. You just say "retry forever," and then people use metrics and alarms, and they get paged when there are too many retries for too long. Then people try to resolve the root cause.

The question is: how do you deal with that for agents? Do you give the agent the ability to learn about this and decide how long to retry? And the most important thing is differentiating retrievable from non-retrievable errors. For agents, it's not always obvious when a tool fails with a retriable vs. non-retriable error - categorizing those things is not straightforward. I've seen people struggling with that.

And then when a retriable error happens, what does the agent do? Does it contact a human? Does it try a workaround? There's a whole big story around at which point humans should be in the loop - troubleshooting versus letting the agent try to find a workaround. And you know, agents are very persistent. They're so resourceful that sometimes you just need to have guardrails, because they will keep trying to solve problems in ways that are not very productive.

**Jeff Malnick:** Do you see them retrying a different way than you might have guessed they would retry? So the system you're building around it - maybe the workflow itself can't really be guided by the same deterministic workflows that we thought of in the past?

**Maxim Fateev:** Yeah. I think guardrails in general for agentic systems are not a solved problem in any absolute way. Agents can be very inventive - this is how they find exploits. You put them in a jail and say "get out," and they find a way.

**Nancy Wang:** Or in our world, even if you don't give credentials to an agent, I've seen coding agents actually reach into running processes to find what they need.

**Jeff Malnick:** Yeah. There was a really fun example one of my colleagues shared. He had a coding agent - he was telling it to send an email, so it needed to log in to Gmail. He had second-factor authentication set up. He got the text message and assumed the agent wasn't going to get in. But the agent - which had access to the local file system - reverse-searched the file system, found the recovery codes for the MFA buried in two years of downloads in the downloads folder, and just logged in anyway.

**Maxim Fateev:** Exactly. Temporal certainly doesn't help you there in terms of making it smarter - we allow your agent not to die, but we're not making it smarter. There is certainly a lot of research to happen in that area.

**Jeff Malnick:** And the agents - if they run for longer, they can be more persistent and come up with more. Not always a good thing.

**Maxim Fateev:** It's an interesting research area, and exactly why people fear them. If they become too capable, they can bypass all our security and do whatever. But I think "smart" is actually the wrong way to even think about them. The right framing is: their capabilities to achieve certain results are there and they are growing. The persistency and resourcefulness are there. But they don't understand why they're doing things - they don't understand when to stop because something doesn't make sense. That's the key missing element.

---

9. Hooks Over Skills: Enforcing Hard Boundaries

**Maxim Fateev:** One thing I found with coding agents is that I cannot trust skills - they ignore them left and right. I almost always end up doing hooks, for practically anything I want the agent to actually do. A hook is a hard boundary: a script or something that will stop a variation in a way that the agent can't route around. If there's something I don't want the agent to do, I don't put it in a skill. I create a hook, if at all possible.

**Nancy Wang:** That's actually a really good insight. We've developed essentially both skills and hooks. For example: don't leave any credentials lying around in plain text, don't save any credentials to your local disk. I'd be very curious to see how our users adopt that in the wild.

**Maxim Fateev:** Yeah, absolutely. I think one thing I like about the recent notion of plugins is that you can decide what goes in the plugin list - hooks, skills, whatever. And you can change it over time. People do need to think about this. You just update the plugin and get better functionality. We certainly started from just simple skills, and we see a lot of adoption of skills because Temporal code is a little bit different from standard code - there are certain rules, and skills help agents follow them. People can run pretty complex applications. But the next step is a lot of validation: hooks, limiters, static checkers, and so on.

---

10. Complex Applications: Distributed Harnesses and Sandboxes

**Nancy Wang:** So tell us more about the demo you wanted to show our audience today.

**Maxim Fateev:** What I really want to show is the recent release of the OpenAI Agents SDK sandbox integration. What I want to demonstrate is how easy the setup is to manage sandboxes, because they already provide integrations with sandbox providers. That means Temporal-powered agents are automatically pauseable. Practically, you don't need to run any compute when the agent is waiting - you can have millions of them running in parallel. Even if you're on a small cloud, millions of agents in parallel consume no compute resources while they're waiting. The moment there is a new event, a timer, or a reply from a long-running operation, the agent wakes up and decides what to do. And if the agent says it needs to run something that requires a sandbox, it will execute that code in the sandbox. And then there's the whole sandbox lifecycle management - if you haven't used a sandbox for some period of time, it will shut down automatically.

**Nancy Wang:** That's something that maps well to our work as well. We've been doing integrations with Daytona for their sandbox environments. The future of enterprise productivity with AI and agents is probably in the sandbox environment.

**Jeff Malnick:** The thing that's interesting here - when you mentioned local agents accessing your full file system - one of the primary things we've been working on is how do you secure the system that's on your local machine? A lot of people end up giving file system access to AI agents running on the CLI. It's essentially handing a perfect stranger your laptop and saying "have at it." And what's scary is that a lot of endpoints store cleartext credentials. When you give an AI agent that runs on the CLI direct access to that file system, it can suddenly access everything in the downloads folder, your dot directories, all your configuration.

So coming up with ways to sandbox that agent where it's running locally is really important. And up in the cloud, the process of getting secure credential information to an agent harness in a secure fashion is a pretty challenging thing. The harness itself, in some cases, you can trust to consume a credential and not pass it off to the agent's context. Other harnesses - you

might not be able to trust that they won't exfiltrate it. So the only way to control for that is to run it in an environment where you have complete control over what the agent has access to, and just assume that things that enter that environment are potentially insecure.

That's why the work we're doing with Daytona in particular is compelling - it allows us to take just a simple reference to a credential in a 1Password vault and resolve that inside that sandbox in a completely end-to-end encrypted, zero-knowledge way.

**Nancy Wang:** Yeah. And that might be a way for us to also work together - when an agent wants to spin up another sandbox and needs credentials within that sandbox to access certain systems, we could have the 1Password broker there as well.

**Maxim Fateev:** Absolutely. We should never pass these tokens through the sandbox. I remember from Amazon - there's a way to access AWS resources without ever passing credentials. Something like service-linked roles.

**Nancy Wang:** Like service-linked roles, yes.

**Maxim Fateev:** So I think we need something like that for the industry everywhere. Ideally some sort of standard. And you guys should just plug in in a standard way. I'm not sure MCP is the right mechanism - MCP is about different things. We need something more like an invocation-level credential injection standard.

**Nancy Wang:** I think the MCP spec kind of just avoided that topic altogether.

**Maxim Fateev:** Yeah. We probably need to extend something for that, because it looks like we're converging more and more on CLI in the near term. We decided to invest in CLI much more than MCP server because the CLI already provides a lot of capabilities - let's just work better with it. But that also means our client needs to be authenticated. If there were a standard API that said "I am running inside a sandbox - here's what I'm going to do, and here's how to inject credentials" - that would be awesome.

**Jeff Malnick:** Yeah. There are some interesting ways people are solving this. One approach we've been using is a shell extension - you can build a wrapper around an executable in the shell that allows you to securely transport a 1Password vault item to that executable. The interesting thing with MCP and MCP servers is that because it's essentially a networking protocol, you can build a gateway around it. You don't have to actually transport the credential to the environment the agent is running in - you just have to worry about transporting it to the MCP gateway that the agent is communicating through. And that gateway doesn't have to be running in the cloud or remotely. You can have it running locally on top of the sandboxing. That way you're free of the concern "did the developer code this to consume a credential securely?" - you can leave all the security guarantees to the sandbox and the networking layer, and make that developer experience pretty transparent.

---

11. What Would You Build? Nexus RPC

**Nancy Wang:** So I usually want to close on our favorite question we like to ask guests: if you could build anything you want over the next month, what would that be? It could be at Temporal, but it could also be something else.

**Maxim Fateev:** At Temporal, there's one thing I just want to finish. We call it Nexus RPC. The idea is that event-driven systems don't have real interfaces - but you still need long-running operations. And you can't just say "use Temporal workflows" as the generic way to solve this problem for everyone.

Can we have a standard RPC protocol for long-running operations? Because if I come to ten developers and say "I want to implement an operation that takes three hours - how do you model that?" - you'll get ten different solutions: webhooks, notifications, queues, whatever. Can we just standardize on one way to invoke it, get notification, learn about its status, cancel it? And then we can do protocol bindings - in HTTP you can implement it using webhooks, or long polling, or gRPC, or Kafka. We model it in a standard way.

And obviously it absolutely fits tool calling. Because if there's already a standard RPC protocol for long-running operations, you just say "MCP has pluggable transports" - this is just another transport. And now MCP magically supports long-running tools. That would probably be the project I'd want everyone to adopt today. We're already building it at Temporal - we're practically at v15 and it already works within our ecosystem. But what I'd love is to speed up getting to the point where we can talk to everyone and make everyone adopt that.

**Jeff Malnick:** So that's like a system with built-in retries?

**Maxim Fateev:** That's right - retries are built in. Long-running operations without guarantees of delivery make no sense. So the whole idea of this protocol would be that it's not only operation delivery, it guarantees the delivery of the operation. Internally it will use queues and webhooks - we're building on top of existing technologies - but we just elevate the level of abstraction from queues to a long-running RPC.

**Jeff Malnick:** You know, what's interesting is that I'm seeing this in the security domain and you're seeing it in the distributed systems domain - in both worlds we are essentially building on top of, and in many cases adapting, the existing tools and protocols we have to make these systems work. And then thinking about new ways to extend those existing protocols to make them function in this new world of long-running autonomous workflows.

**Nancy Wang:** Well, I'd be excited to see that in real life. And I feel like we could certainly continue this conversation for many, many hours. But I just want to say thank you so much, Maxim, for joining us in the studio today.

**Maxim Fateev:** Thank you.

---

Part 2: Live Demo - Durable Agents with Temporal and OpenAI Agents SDK

The following is a live demo session recorded with Maxim Fateev, demonstrating Temporal's integration with the OpenAI Agents SDK and cloud sandbox environments.

**Nancy Wang:** Hey, Maxim, thanks so much for joining us. We spoke a lot during the live interview about the work you've been doing at Temporal as co-founder and CTO, and so we wanted to welcome you back to demo something super exciting - a recent launch from your team. Do you mind giving us a quick intro on what you're about to show today?

**Maxim Fateev:** So I wanted to show you a pretty simple sample of a chatbot that invokes tools and executes those tools - especially shell commands - in a sandbox. As I mentioned, sandboxes have become part of the application. A harness is not just an application that invokes an LLM and tools - it also has to provision infrastructure, because a sandbox is practically infrastructure and you're mixing them in one place. OpenAI's Agents SDK recently added sandbox integration and it's already integrated with the OpenAI SDK, which made it durable because the OpenAI Agents SDK is just a Python program running in memory. But with Temporal, it becomes a durable workflow - which means that it preserves all the state. Any part of the application can fail at any time. The process can crash, the machine can go down, but the actual durable event loop stays durable, so you don't lose data. And it also provisions sandboxes to execute dangerous operations. That's what I wanted to demonstrate.

**Nancy Wang:** So would you say this is a core separation of compute and the other processes? Maybe walk us through why you've designed it that way.

**Maxim Fateev:** A few things. Sandboxes are not cheap, especially if you have a very large number of agents running in parallel. But you have to have them - otherwise you wouldn't have appropriate security. The idea is that your sandbox is active only when you need to run dangerous commands. For example, when your agent is waiting for user input - which can be pretty long, like 20 hours - you don't want a sandbox running for 20 hours waiting for that. You don't want to be paying for a separate machine for every agent.

So the idea is that the agent loop can run in a separate process. You can have a lot of them running in parallel with no overhead while they're passively waiting. You can have billions of agents running in parallel using a very small number of processes, and only start sandboxes when they need to execute specific commands. That's the idea. It also allows you to add durability more easily, because you can separate out the durability of the agent loop and the durability of the file system and sandbox - they become separate concerns.

**Nancy Wang:** That makes a lot of sense. We'd love to see it in action.

**Maxim Fateev:** So here I have a simple coding agent demo. Let's ask it to generate some code.

Let's say regenerate hello world in Python.

It's pretty primitive - it has some basic shell tools - but it demonstrates the idea. Okay, we can find it here on my local machine. And you see, it's a pretty simple Python program. But let's do something interesting. Let's switch to a different backend. So currently we're running locally on my machine. Now I want to run this in Docker.

I'll say okay, change to "hello docker."

And it will start running this command. And we can bring up the Docker dashboard. You see - it just started a container inside Docker to execute this command. And then it shuts down the container after executing. And it printed "Hello Docker" when it was executed.

By the way, if you look at the original place where we did that, it still says "hello world" - because when we were switching to this Docker environment, we took a snapshot of the local development environment file system and saved it. Then we reconstructed it in the new Docker image, made the change, and executed it. So the change was only made locally in the Docker image. The local environment is still running on my machine.

**Nancy Wang:** So would you say the key value capture here is that you have a consistent runtime, whether you're in Docker containers or local? And even when you say "change the backend," it's a consistent runtime experience for the end user?

**Maxim Fateev:** Yes, it's a consistent runtime. Also, the user really doesn't need to think about where this agent is running. Let me show you something.

**Nancy Wang:** That brings up the question - now that you're abstracting away different backends and environments: how are you thinking about observability? When you have an agent hopping around different environments and processes, how would an end user think about debugging or replaying certain agent actions?

**Maxim Fateev:** Okay - I'm bringing up the Temporal UI window. This is actually a direct answer. Because Temporal logs everything that goes on into your agent and then your application. In this case, you can see the whole session is recorded. For example, it was using the local sandbox - you can see all the commands that went into that sandbox. And then it had the Docker sandbox - it actually said "okay, we need to start the session," so it shows the input to the commands, which snapshot to use, and all the other information about the Docker image. And it's all recorded.

And you can look at that. This is the message I sent: "change to hello docker." And it started to execute Docker commands. And then you see the last command was "delete docker image" because we finished the command. Everything is recorded.

Let me show you something else. I'm going to kill the process - this is the process that runs the workflow, that actually runs the agent loop. And if I try to ask something - let's say "change back to hello" - you see, it got my message, but nothing is happening, because the process that actually executes this logic is down.

But the moment I bring it back, it will continue execution from exactly the same point, as if nothing happened. Because the full state of the agent is persisted inside of Temporal. It will just continue executing. And this is the power of Temporal - your infrastructure can fail at any time. But if you look at the code of this agent, it doesn't contain any crash recovery logic. It is the same OpenAI Agents SDK code written without any crash recovery in mind. What comes out of the box with all these features is because it's using Temporal. So it's not only about visibility - it's not just for human consumption. The recorded history is actually used to recover state.

**Nancy Wang:** Got it. So you've essentially moved the entire debugging story into Temporal and away from the sandbox or whatever environment it happens to be running in.

**Maxim Fateev:** Exactly. The sandbox becomes just the place where you execute certain commands. The actual full state, debugging, scalability, human-in-the-loop - all of these things become part of the Temporal story. Also, this UI is talking to the Temporal server - it doesn't execute any commands directly. So you can kill it and reconnect any time.

**Nancy Wang:** So the Temporal server essentially becomes a control plane.

**Maxim Fateev:** More like a coordinator. The code still runs on your premises. But let me show you one more thing - I can actually switch this to Daytona, which is a cloud provider for sandboxes.

Now if I say hello - you see, a Daytona sandbox was created. And it's now executing this command in Daytona. And it took the snapshot we took from my Docker - so you can see the session continues. It's still the same workflow, still my command. Now it's executing Daytona commands. Daytona sandbox session starts and stops are recorded. And now it's done. The agent can wait for my input for the next few days, and no resources are used - especially if the UI is disconnected.

Let me also show what happens when things fail. Let's shut down our worker - which hosts this code - and simulate a failure by putting in an invalid Daytona API key, and restarting with that misconfigured worker. And then I'll ask it something.

You see - it's throwing "failed: invalid credentials." This is an error happening. But if you go to the Temporal dashboard, you will immediately see: "Daytona sandbox client resuming: failing because invalid credentials." The whole stack trace of the error is there, and it's being retried using exponential backoff. You have full visibility.

Let's say this was an intermittent problem. We'll restart the worker with correct credentials. And on the next retry - you see it went through. The sandbox is running. You can control the timeouts and retry policies. But the basic idea is: the process that hosted this code died. It was misconfigured when we restarted it. But your session keeps going. And that is the basic idea - Temporal allows you to write agents that are durable, that don't fail, that can run for a very long time, that don't consume resources when they're waiting. And with the OpenAI Agents SDK sandbox integration, we execute dangerous commands in sandboxes only when needed.

**Nancy Wang:** Wow, this is super cool. As more enterprises deploy agents in production, the runtime is going to change. I'd be curious to understand - what about scenarios where you have them running in parallel, or cases where there are flaky tools as well?

**Maxim Fateev:** So it will be exactly the same, because all external API calls - LLM calls, tool calls, sandbox management operations - are modeled as what we call activities, and they are retried by default. The only work you need to do as a developer is to differentiate retrievable from non-retrievable errors. For example, we treated invalid credentials as a retriable error in

this demo. But some errors are not retrievable. And it's interesting because in the context of long-running agents, what's retrievable vs. non-retrievable is different from traditional applications.

In a normal application, if you make a request and get "bad credentials," all you can do is fail - your credentials will not be fixed in 5 seconds. But if you have something running for a day, you absolutely can provide a fix, and many errors that are usually non-retriable become retriable. But some still aren't - like if you're making a money transfer and the account ID is wrong, you probably don't want to keep retrying that forever.

**Nancy Wang:** I guess the first part is: what happens when you have different environments running in parallel?

**Maxim Fateev:** So the worker here is just a process that hosts agent code, and you can have a lot of them running in parallel. You can kill any of them at any time, and states will be migrated automatically to another machine. That comes out of the box.

There are also other scenarios - when you have agents belonging to different groups that need to communicate, or tools belonging to different teams. Essentially, think about a service-oriented architecture. The problem with current service architectures is that they're not friendly to long-running requests. If you have a well-defined service interface with operations and timeouts, and then you say "this operation can take three hours" - you can't have a synchronous interface for that. You're forced into the event-driven paradigm. Temporal fixes this out of the box by allowing you to define long-running operations that map very naturally to tool calling. And sub-agents can be modeled as distributed durable workflows. You can absolutely run a lot of them in parallel - the scalability is practically unlimited.

**Nancy Wang:** Well, that's the agent swarm use case I'm seeing across DevOps and SRE use cases. One maybe final concept - idempotency. Especially around long-running workflows with retries. How do you avoid side effects when idempotency is an important design consideration?

**Maxim Fateev:** In any distributed system, the reality is that you have to make external APIs idempotent, because if you're calling a tool, there's always a probability that the tool will complete execution and then the acknowledgment and result will be lost in transit - for example due to a network issue or process crash. Which means the actual operations you're calling must be idempotent. Temporal has some support for non-idempotent operations in the form of "at most once" execution - you can call an operation and say "I will execute this operation once, and return an error if I can't guarantee exactly once." But one problem is that if you get an error, you don't know if the operation succeeded or not. So you need to run some other operation to validate if it succeeded. You can mimic idempotency at a higher level. But generally, every operation that calls external APIs - those APIs need to be idempotent. I don't think there's a good way around that.

**Nancy Wang:** Living proof that even in the world of agents, we still have to remember how to build scalable distributed systems.

**Maxim Fateev:** Yes. 100%. I think agent systems are becoming more and more like real distributed systems. From local simple single-process programs, agent swarms and sandbox management are becoming very large distributed systems. And this is a great, important moment - while we were experimenting with local Python programs, yes, you didn't need Temporal. But right now, once you start scaling those systems and adding resiliency to them, you need durable execution to run in production.

**Nancy Wang:** Couldn't agree more. Maybe one word of advice for customers who are going to be trying this out in their production systems?

**Maxim Fateev:** Do it iteratively. Temporal is not the easiest way to write your code. But make sure that you don't only focus on the AI and LLM part. Think about the distributed systems problems, because the moment you deploy these things in production, you are dealing with a large-scale distributed systems problem. And if you use Temporal, we can take on most of the burden of those complexities.

**Nancy Wang:** Great advice, Maxim. Well, thanks so much for coming to the studio again and walking us through this demo. I can't wait to try it myself.

**Maxim Fateev:** Absolutely. Thank you.

---