XPRT.

Magazine Gold Edition / 2025

10 Years of Knowledge Sharing

Nº1

Xebia

We Are Xebia

Your Trusted Microsoft Solutions Partner

Microsoft Solutions Partner

GitHub Verified Partner

GITHUB PARTNER OF THE YEAR 2024
CHANNEL SECURITY

GITHUB PARTNER OF THE YEAR 2024
OVERALL CHANNEL PARTNER

Microsoft Solutions Partner
Infrastructure
Azure
Specialist
Infra and Database Migration

Microsoft Solutions Partner
Digital & App Innovation
Azure
Specialist
Kubernetes on Azure
DevOps with GitHub

Microsoft Solutions Partner
Data & AI
Azure
Specialist
Kubernetes on Azure
Infra and Database Migration

Microsoft Solutions Partner
Security
Specialist
Cloud Security

Xebia
xebia.com

Colophon

XPRT. Magazine N° 18/2025

Editorial Office
Xebia

This magazine was made by Xebia
René van Osnabrugge, Marcel de Vries, Emanuele Bartolesi, Wesley Cabus, Andreas Giesa, Olena Borzenko, Erwin Staal, Thomas Tomow, Michael van Rooijen, Jonathan David, Randy Pagels, Arjan van Bekkum

Contact
Xebia
Laapersveld 27
1213 VB Hilversum
The Netherlands
Call +31 35 538 19 21
xmsinfo@xebia.com
www.xebia.com

Photography
Studio OOM

Layout and Design
Studio OOM / www.studio-oom.nl

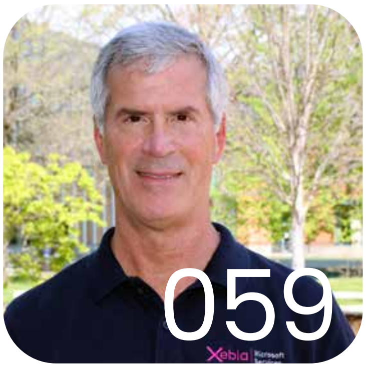
Translations
Mickey Gousset (GitHub)

© Xebia, All Right Reserved
Xebia recognizes knowledge exchange as prerequisite for innovation. When in need of support for sharing, please contact Xebia. All Trademarks are property of their respective owners.

Microsoft Solutions Partner

GitHub Verified Partner

If you prefer the digital version of this magazine, please scan the qr-code.



This issue of **XPRT.** magazine celebrates 10 years of knowledge sharing.

Intro

004 A Decade of the XPRT. Magazine

Knowledge Driven

005 Looking back on 10 years knowledge sharing

Power Through Platforms

010 Inspired by Canvas, Built With Code: How I Created My Own Art Tool

017 Infrastructure from Code

025 From Concept to Creation: Developing Intelligent AI Agents for Modern IT Solutions

032 Securely deploying to Azure using Octopus Deploy with Entra Workload Identity

037 Letting AI Help Make the World More Accessible - Analyzing Website Accessibility with Semantic Kernel and OmniParser

State-of-the-Art Software Development

045 .NET Aspire: A Game-Changer for Cloud-Native Development

049 Are you sure your JWT access tokens are really secure?

Smooth Delivery

055 Engagement Management your first-class ticket to value

Knowledge Driven

059 Startup Guide to Prompt Engineering using GitHub Copilot

063 AI Promptception - Iterating GitHub Copilot Prompts for Maximum Impact

A Decade of the XPRT. Magazine

It was 2015. I just entered the Xebia office as a freshman. The company was founded 3 months before, and the 6 others and I as a new joiner were gathering for our weekly knowledge sharing session on Tuesday. "What are we going to do?" was the question we asked each other.

Author René van Osnabrugge

"Techdays is coming up, and we have tons of sessions to present, so let's prepare that" said one. But that is not what we wanted, we wanted to make more impact than that. As a new kid on the block in the Dutch Microsoft market, we wanted more. One of my colleagues suggested: "Shall we make an old-fashioned magazine?. With all the content being online, this is maybe a refreshing thing to do!". "And then hand them out to all 3000 attendees of the conference in their goodie bag!", said another. "Yeah, and then we write articles about the sessions we do, so people have some background information, or are triggered to come to our sessions". And soon enough this came to fruition. With only 3 weeks to go before the event, we wrote the articles, designed the magazine, found a party that could do the lay-out, went to press and printed 5000 copies of the magazine. XPRT magazine No 1 was born. The night before the event we went to the venue with the whole crew to put the magazines in the goodie bags. At 3.30 AM we were done! Tired, but very proud, we went home for a few hours of sleep before delivering our sessions the next day!

That was the start of our magazine. Fast forward 10 years to this new shiny gold edition. Magazine 18. An anniversary edition in 2 parts, of which this is part one! A lot of things have changed. Compared to the first magazine we changed quite a bit in lay-out, photo's and graphic design. For a nice overview of covers, look at the article around 10 years of knowledge sharing to see the evolution of our Magazine, but also of our company's learning with knowledge sharing over the last 10 years. Although many things have changed over years one thing remains the same. The pure and honest, deep content of our own people. Not only from the Netherlands but from our company worldwide. The power of XPRT magazine is to be always from the hearts and minds of the people. They write about things they like! Things that they work with, figured out on one of our innovation days or about hobby projects in their spare time. But always with a focus on technology, and always on the bleeding edge of the next evolutionary wave.

This magazine's edition is not different in that respect. Again, we have articles talking about the next thing. Of course we are also heavily focused on AI. Randy Pagels has 2 articles to make better use of GitHub Copilot and how to improve your prompting to make better use of it and get better output. Thomas Tomow looks at the next evolution of AI Agents. He guides us through building an AI agent using Semantic Kernel, highlighting the shift from rule-based to learning-based and generative AI approaches. Jonathan David also uses tools like Semantic Kernel and OmniParser, and his article provides technical insights into using AI for compliance with accessibility standards. And that is not the only thing AI can do. It can be used for so much more. Olena Borzenko explores the concept of "Generative Chaos" and uses AI to generate pieces of art. But there is more than AI. There is still a lot going on in the infrastructure world as well. Erwin Staal discusses the evolution from traditional Infrastructure as Code (IaC) to "Infrastructure from Code," where infrastructure is generated directly from application logic. Emanuele Bartolesi introduces .NET Aspire, a Microsoft stack for building cloud-native applications and provides a step-by-step guide to getting started with it. As always, we have security in high regard. Wesley Cabus explains the differences between opaque tokens and JWTs for API authentication and authorization and Michael van Rooijen wrote about how to securely deploy applications to Azure using Octopus Deploy and Entra Workload Identity. And still, this is not all. Evolution does not only happen in technology. We also evolve in the way we run our projects. Andreas Giesa describes Xebia's approach to engagement management, focusing on delivering sustainable value in digital transformation projects and introduces the Xebia Engagement Management Framework.

I think we did it again. This new anniversary edition is packed with new technology and insights that are on the edge of innovation, I truly hope you enjoy the magazine and make sure you share this knowledge with others!

10 years of sharing knowledge

When we started Xpirit 10 years ago, we began with a mission and four company values. Our mission, "being an authority," says it all. We aim to be the best in our industry. Our company values support this mission: putting our people first, quality without compromise, customer intimacy, and, not least, sharing knowledge. And now, 10 years later, and being renamed to Xebia, this has not changed a bit! This article focuses on the last of these values because while sharing knowledge is easy to talk about, it is far more complex in practice. In an everchanging industry like ours, how do you keep things engaging? How do you stay up-to-date and relevant? And how do you ensure your people remain motivated?

Authors René van Osnabrugge and Marcel de Vries

It would be wonderful if the "Sharing Knowledge Recipe" could be laid out here in this article. After 10 years, we should have figured it out, right? But, unfortunately, it's not that simple. There isn't a single recipe, and if you think you've found one, it often falls apart. Almost like Schrödinger's cat. Schrödinger's cat is a famous thought experiment in quantum physics. Imagine a cat in a sealed box with a tiny bit of radioactive material. If it decays, it triggers a mechanism that kills the cat. If not, the cat lives. But until you open the box, the cat is both alive and dead at the same time. It's only when you observe it that one outcome becomes real. It's a strange idea, but it beautifully illustrates how uncertain and fluid things can be. Especially, as we've learned, when it comes to sharing knowledge. Once you observe it, it changes. So, rather than providing you with a recipe, a silver bullet, or a todo list to implement immediately, we want to share an overview of what we've done over the past 10 years. We'll highlight what worked and what didn't.

XPRT. Magazine N°6/2017
Reinventing Collaboration
endless possibilities
XPRT. Magazine N°7/2018
TECHORAMA SPECIAL EDITION
Future Tech, Test Adventures, Cloud Strategy and DevOps
XPRT. Magazine N°9/2019
TECHORAMA SPECIAL EDITION
The Rise of Software Complexity
developers business
XPRT. Magazine N°11/2021
XPRT. Magazine N°12/2022
Together we build an Engineering Culture
Launching Xpirit IoT: Smart & Connected Services
What's what with WebAssembly?
Shift left using Bicep
Customizing Codespaces
Xpirit as an IT Beehive

But keep this in mind: sharing knowledge is deeply cultural. What works for us may not necessarily work for others. Still, we hope you'll find some inspiration. Let's start where everything should begin, with the *why*!

Why Share Knowledge?
"In the land of the blind, the one-eyed man is king," or "knowledge is power." Sure, there's a bit of truth in both, but let's be honest, they're not exactly the best foundation for a modern, knowledge-based company. They're built on the idea that a select few hold exclusive knowledge, and that can work for a while... until those people leave. Then what? It's something many companies have faced. And even more importantly, what about everyone else who wasn't part of that inner circle?

In our industry, where change is constant and innovation never stands still, keeping up on your own simply isn't realistic. You need each other. And learning has to happen in both directions: broadly and deeply.

But the real reason we share knowledge goes deeper than that. It ties into what we believe. If you want to be an authority, you need to know a lot. And if you're surrounded by people who are passionate about their profession, they naturally want to share what they've learned. Sharing helps you grow as a person, but more importantly, it helps the group grow. Because we don't think in terms of individuals alone. We think in terms of teams, of people growing together.

That's why we believe so strongly in the principle of "you grow, we grow." It's something we actively support and nurture.

Sharing knowledge isn't just about picking up facts or tips from others. It's about fueling the growth of the group. And being part of that kind of collective growth gives people energy. It creates pride. It makes you thrive in your craft. And that, in turn, supports our mission of being an authority. Because we don't just share internally. We also share externally. And when people outside the company see a team that shares openly and knows their stuff, that's when you stand out. That's when you become a true authority.

Broad Knowledge
Broad knowledge is about exploring things you wouldn't find on your own. Everyone lives in a bubble, but fortunately, no two bubbles are identical. By engaging with others, you gain access to different ideas, interests, approaches, and perspectives. This interaction can spark innovation, inspire smart combinations, or reinforce existing ideas. Therefore, we don't limit our knowledge sharing sessions to technology or only Microsoft. We share knowledge about hobbies people have, like farming or retro gaming. We share knowledge about how you can grow as a human, what you have learned from your coaches, or how to be more

effective in conveying your perspective and convince people. The fact we share in so many areas makes it very diverse and engaging. You learn things you would never have imagined and while we share this with each other, you also make a better human connection.

Deep Knowledge

Deep knowledge, on the other hand, is about delving into a subject more thoroughly. You might already be an expert, but explaining a topic to others often leads to new insights. Think about explaining something to a friend that is not in your industry. It forces you to think differently and approach the topic from fresh angles. And the seemingly "simple" questions that are asked might uncover aspects you hadn't considered, giving you new paths to explore. Sharing knowledge, therefore, is not just about giving. It's also about receiving, even if it's not directly related to the topic at hand. It enhances you as a professional and as a person, whether you are the one sharing or the one learning. In an industry like ours, where innovation is relentless, a mechanism for sharing knowledge is essential to keep people engaged, curious, and up to date. Without it, you risk losing your edge and eventually becoming obsolete.

The How and What of Knowledge Sharing

Now, let's delve into the next topic: what is knowledge sharing, and how do you do it? What do you actually share? As mentioned earlier, knowledge is something you know. It can be deeply technical, a unique perspective, an insightful question, or even a combination of these elements. It could relate to a new technology, a book you've read, or the connection between the two. Ultimately, knowledge sharing always involves both giving and receiving information. There are countless ways to share knowledge, and it can be directed at different audiences. To understand this better, let's first make an important distinction between internal and external knowledge sharing and explore what we at Xebia have done in both areas. We'll start with external knowledge sharing, the most visible form of sharing for those who don't work with us (yet 😉).

External Knowledge Sharing

If you're familiar with us, you know we share knowledge worldwide. From national and international conferences to meetups, hackathons, and groups both big and small, many of our colleagues love sharing their stories with an audience. For us, this is crucial. It aligns with our mission to be an authority. We share what we know with others, and we're fortunate to have many people who are passionate about doing this. By supporting these individuals and giving them the time and freedom to speak at conferences, we achieve several critical goals. First, our people stay motivated to dive into new topics, improve themselves, and pursue our authority mission. Second, our presence at these events demonstrates to other companies that we have experts who truly know their stuff, which enhances our reputation. Third, individuals who share our ambitions and values recognize us as a potential employer. It's a win-win-win situation that makes everyone happier. Another significant way we share knowledge externally is through **writing**. Just like the writing of this article. The same principles apply as with speaking: when you need to explain what you know to others, it transforms into a completely different experience. Things you "just know" must be articulated more clearly, which sharpens your understanding. Writing remains a powerful way to convey deeper messages, enhanced with code samples, illustrations, and visuals. It's a refreshing alternative to the quick video snippets that dominate today's content landscape.

We also host **hackathons and workshops**. Take, for example, an event like the Global DevOps Experience[1]. An event around DevOps, Azure, GitHub and AI that can be hosted by communities all around the world. By creating this fully realized, out-of-the-box community event that others can host worldwide, we expand our reach significantly. The energy, excitement, and engagement generated internally while organizing these events are priceless. Just as importantly, we learn a great deal ourselves during the process and share that knowledge with one another. For instance, when planning such events, we aim to incorporate technologies we haven't used before so we can experiment, learn, and grow. Finally, another approach to external knowledge sharing, especially with our customers and prospects, is what we call **Pure Consultancy**. This involves sharing everything we know with clients, even if they haven't signed a contract yet. We believe that immediately offering help and discarding the old notion that "knowledge is power" fosters stronger relationships. And yes, if a customer finds what they need in a short conversation and doesn't require further business with us, we're perfectly okay with that. This open approach builds trust and sets the foundation for meaningful collaborations.

Internal Knowledge Sharing

While external knowledge sharing is highly visible and contributes significantly to our mission of becoming an authority, it is only possible because of the strong foundation of internal knowledge sharing. At Xebia, this culture is embedded in our DNA, starting right from the hiring process. During the hiring process, we discuss candidates' perspectives on knowledge sharing, their mission of authority, and their favorite topic, both professional and personal. This isn't limited to work-related subjects. Many of our team members share knowledge about hobbies like cooking or bouldering. Learning from diverse perspectives opens up a whole new spectrum of possibilities and fosters creativity. However, this culture doesn't develop automatically. Over the years, we've put in place several facilities, events, and ceremonies to nurture and sustain it.

Facilities for Knowledge Sharing

We believe sharing knowledge should be as seamless as possible. That's why every meeting room is equipped with whiteboards, sticky notes, and markers. Relaxed seating areas and phone booths are readily available, ensuring the environment supports open communication. In our culture, face-to-face discussions are prioritized. At any moment of the day. Because it is these 1 on 1 conversation that enable the sharing of tacit knowledge. Where "knowledge" includes anything you can explain, write down or teach, like facts or instructions, tacit knowledge is a subtler form of it: personal, experience-based and hard to express, often learned by doing rather than explaining. And the abundance of tools and accessible spaces are there to facilitate that and eliminate any barriers for discussions or brainstorms. And what is the result in our office? If someone is using a meeting room for a remote call, it's common practice to gently redirect them to a phone booth so the space can be used for collaboration.

Internal Events and Ceremonies

Over the years, we've developed several internal rituals to continuously improve and learn from each other:

1. Relive the Dream, Offsite, and Retrospectives

We begin each year with a "Relive the Dream" session, where we align ambitions, revisit personal and team goals, and discuss what's needed for success. At the year's end, we hold a retrospective to share lessons learned and celebrate our successes. Midway through the year, we organize a two-day offsite where the entire team gathers at an external location. These sessions focus on building connections, sharing personal stories, and developing both team and individual strengths. But you might wonder. How is this knowledge sharing? It absolutely is!
These sessions build trust and connection. Without trust and psychological safety, sharing knowledge becomes difficult. Sharing knowledge can feel vulnerable: What if someone knows better? What if a question arises you can't answer? Or what if a discussion leads to conflict? That's why creating a safe environment is crucial for fostering a knowledge-sharing culture.

2. XKE and XMS Tuesdays

While some companies organize quarterly knowledge-sharing sessions, we take it a step further and do it weekly.

- **XKE (Xebia Knowledge Exchange):** Every two weeks, we host this event. Some kind of mini-conference where colleagues from across Xebia propose sessions on various topics. Attendees can pick and choose sessions based on their interests, enabling a continuous exchange of ideas and expertise.
- **XMS Tuesdays:** On alternate Tuesdays, we narrow the focus to our Microsoft Service Line. These evenings are spent brainstorming, presenting sessions, mob programming, or simply engaging in informal discussions.

These weekly events ensure that learning and collaboration never stop.

Internal knowledge sharing is at the heart of everything we do. It creates a supportive environment, encourages curiosity, and ensures we continuously improve. Not just as individuals, but as a team and an organization. By fostering this culture, we remain at the forefront of innovation and set ourselves apart as a true authority in the industry.

3. Innovation Days

Every quarter, we dedicate a full day to **Innovation Days.** During this time, employees can pick and choose any project they like, as long as it's not for a customer and has an innovative aspect. The term "innovative" is broad.
It simply means something that's new or challenging for you personally. Typically, groups form around ideas or problems people want to explore but haven't had the time to address during customer projects. These days serve two key purposes: strengthening social connections with colleagues and experimenting with new, often untested ideas. Failure is common, but it's part of the process, and the lessons learned often spark further creativity and growth. And when you failed in your goals that day, that is not a reason not to share, it is the reason you do share, so others can learn from the failures and we collectively know more on how not to approach a certain problem or technology!

What Works and What Doesn't Work

Over the past 10 years, we've implemented numerous initiatives to support knowledge sharing. The practices described earlier in this article are the ones that continue to work well for us as a company. However, we've also experimented with approaches that didn't deliver the desired results and were eventually dropped. Here are a few examples:

What Didn't Work

Remote XKEs

During COVID, we attempted to replicate the in-office XKE (Xebia Knowledge Exchange) experience in a virtual format. It didn't resonate with people. We realized that the social aspect of these sessions is just as important as the content, and the lack of face-to-face interaction reduced the overall experience. When you are already a remote-first company, this is different. Knowledge sharing happens online, but moving in-person sessions to a remote only or hybrid sessions was very hard.

Focusing the Fire

We tried to concentrate our knowledge-sharing efforts each quarter around a limited number of topics.
This approach backfired, as people lost interest when they weren't enthusiastic about the chosen topics.
Allowing freedom of choice is essential for engagement.
One thing we learned is you cannot govern where people's

---

[1] https://globaldevexperience.com/

interests lie. And you are also missing out on ideas that you would not have thought of. This is one of those things that comes so naturally when you grow as a company, but we learned to embrace the abundance of ideas and topics to share knowledge about and learned not to govern them. Empower people to share whatever they want and you will see more energy and enthusiasm.

Themed Innovation Days

We organized innovation days centered on specific themes. While some people enjoyed these events, others ignored the themes and worked on projects they found more interesting. Once again, we learned that giving people autonomy yields better results. One of our big lessons here was that by theming your innovation days, you again try to govern innovation, while innovation comes from places you least expect it. By not freely letting it flow, you restrict innovation, and you restrict what you can collectively learn.

Themed Magazines

We also experimented with publishing magazines focused on a single theme. This, too, proved difficult, as contributors struggled with the restrictions. That's why we now encourage everyone to write about topics they are genuinely passionate about.

Summary

Over the last decade, we've built our company around the principle of knowledge sharing. By encouraging a strong knowledge-sharing culture and motivating all employees, both existing and new, to participate actively, we've established a core pillar for a successful engineering culture.

We've learned that flexibility, autonomy, and a focus on the social and collaborative aspects are key to sustaining this culture. Sharing knowledge remains essential to who we are, and we're committed to continuing and evolving these practices over the next 10 years and beyond.

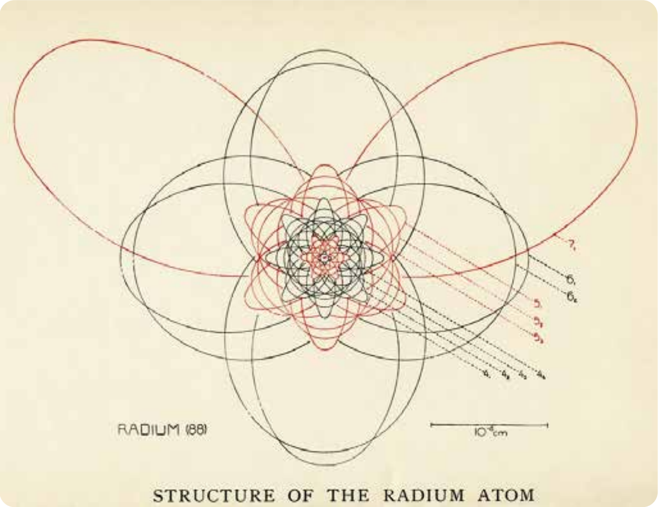
Inspired by Canvas, Built With Code: How I Created My Own Art Tool

It's not the first time I've leaned toward art and creativity in my articles and conference sessions—but this time, I took it a step further. For a very long time, I couldn't decide what I liked more: tech, science, or art. I'd read books like *A Brief History of Time* by Stephen Hawking, *The God Equation* by Michio Kaku, and others because complicated science was always on the edge of fantasy for me. At the same time, I'd write code and do techy things for work and never miss a chance to dive into something creative and artistic. Every now and then, I'd pick up my watercolors, promising myself to draw regularly, only to put them away again because of lack of time and the endless list of other things to do. (Very typical story, isn't it?) Still, I realized this pattern is just part of my creative rhythm. I always come back to art with new inspiration, then shift gears when life gets busy. And last summer was no exception.

Author Olena Borzenko

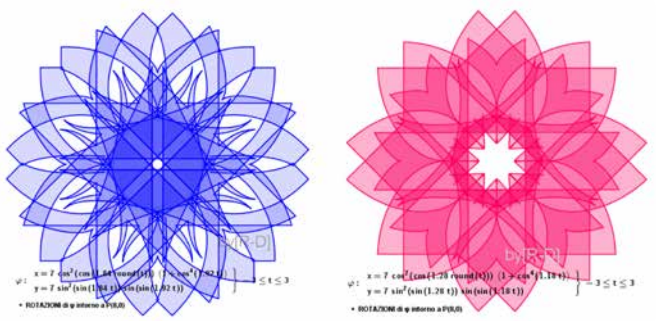
What Set Things in Motion

My surroundings in Ukraine have always included creative and artistic people. When I went home to visit my family and friends, I found myself in a very supportive and inspiring environment. It gave me the motivation I didn't know I needed. For a while, I'd been carrying around the idea of visualizing math equations or physical processes. I didn't know exactly what I wanted to create, but I spent a good amount of time researching different equations, reading articles about the intersection of math and art, and watching tons of videos about physics visualizations like particle spins, magnetic fields, and so on. One day, I came across an illustration of the Radium atom by Niels Bohr from his 1922 presentation, and I fell in love with its symmetry. The way the orbits formed a shape that looked like a flower or a star was fascinating.



Picture 1. Radium atom by Niels Bohr

I later found similar visuals on a site where the author had played with the parametric equations of a curve.


by[R-D]
by[R-D]
ROTAZIONI di φ intorno a P(0,0)
ROTAZIONI di φ intorno a P(0,0)

Picture 2. Life Through A Mathematician's Eyes blog. Parametric Equations article

I have to admit, I'm oddly drawn to these kinds of visualizations. So I knew, deep down, that I'd eventually start looking for this kind of geometry in my own artwork.

First Art works

Speaking of which, the summer of 2024 was when I created my first artwork on canvas. Everything about it felt new and unfamiliar. For the first time, I was working with modeling software, machine tools, real canvas, and acrylic paints. I'm really grateful to my partner for all the patience and support he gave me while teaching me how to use the tools and work with the CNC machine to bring my ideas to life. For context, a CNC (Computer Numerical Control) machine works by following digital instructions to move tools or pens across a surface, making it possible to cut, carve, or draw with precision.

My first few pieces were all about learning and exploring different styles, techniques, and colors. I was trying to find what felt right for me but I'm not gonna lie, I struggled quite a bit at the beginning. Creating sketches with modeling tools was especially tough, so I mostly stuck with primitive shapes.

To print a sketch with the CNC machine, you need to provide a file in a specific format called G-code. It's basically a list of instructions and coordinates that tells the machine how to move a pen across the canvas. To get to that point, I first had to create my design using software like 3ds Max, Inkscape, SketchUp, or others I haven't even explored yet. Then I had to convert the design into a 2D vector representation. Using tools like Aspire or Inkscape, I could build the drawing path and generate the G-code file. That G-code can then be used with the CNC machine but instead of using a rotary tool for engraving or carving, we used pens. So in a way, it became a kind of DIY plotter. The main difference is that while a traditional plotter is designed only for drawing, a CNC machine is much more powerful and flexible. It's typically used for cutting, milling, or engraving, but it can be adapted for more creative purposes. At our Xebia office in Hilversum, Netherlands, for example, we have a similar CNC machine that's used to water plants.

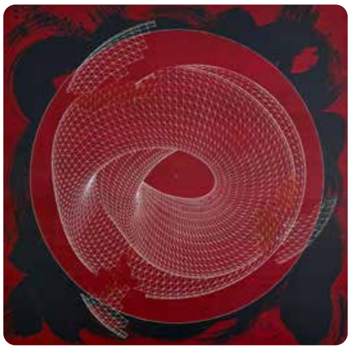
On the pictures below, you can see how some of my ideas started to take shape. I was combining 3D wireframe forms with loose acrylic brush strokes. The first piece I made was surprisingly large. [see picture 3]

Here's another example. This time, I chose a smaller canvas, but worked in the form of a diptych. With this piece, I discovered a really interesting effect, how paint strokes can add a sense of volume to the wireframe mesh. The intersections where textured brushstrokes meet the white lines are still my favorite parts. [see picture 4]

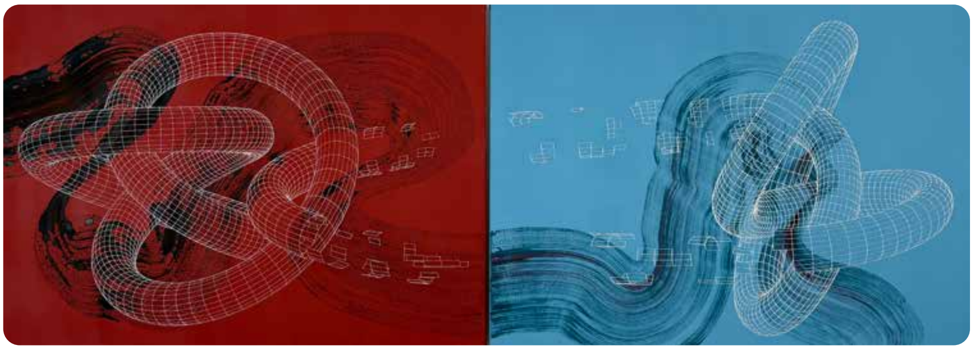
Finding the Right Shape

By this point, I had realized that I really enjoyed combining acrylic brushstrokes with wireframe mesh, so I knew I'd continue working in that style. But every sketch I'd made so far had been time-consuming and frustrating, mostly because I lacked the skills to work efficiently in 3ds Max. And still, none of the shapes felt quite right.

So I took a step back. Instead of tweaking segments and adding all kinds of scattered details manually, I went back to basics and explored some primitive forms, trying to avoid any heavy editing. That's when I stumbled upon the torus knot object. The torus knot is a general mathematical concept, a type of 3D curve that loops around a torus in a specific pattern, but many 3D editing tools, like 3ds Max, also include it as a ready-to-use object or modifier. It can be adjusted and transformed using different built-in modifiers and effects, and the results were unexpectedly awesome.



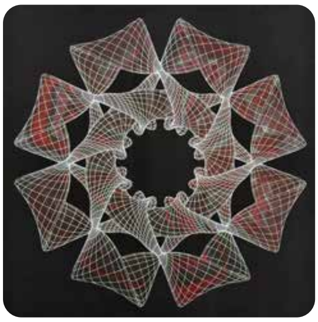
Picture 3. First artwork. Acrylic on canvas, 120x120 cm



Picture 4. Diptych. Acrylic on canvas, 80x55 cm

A closed shape, flowing curves, endless variations and each one brought a different kind of visual association to mind. That was it! Exactly what I'd been looking for. From that point on, I could finally shift my focus to painting, playing with patterns and textures, and actually enjoy the process — hands in paint, not just eyes on a screen.

At one of the conferences, I was really excited to show my work to a friend, and he described the paintings as "visually pleasing." I couldn't agree more and that comment stuck with me. I kept creating, and now I'm proud and happy to share some of that work with you here in this article. All pieces are acrylic on canvas, size 60x60 cm.


Picture 5. "Golem"

Picture 6. "Stardust"

Picture 7. "Salamander"

Picture 8. "Moss"

Picture 9. "Flux"

Picture 10. "Arachne"

What got me thinking was that, at some point, I started to feel limited by the shapes I was creating. I realized I had found a comfortable spot where I was only making small tweaks to the same base forms in each new sketch. It didn't seem like a big issue at first, but whenever I tried to create something more interesting, the results turned out messy and frustrating.

Since I wasn't eager to spend more time learning 3ds Max in depth, I knew it was time to take a different route: I was going to write some code to do all of this for me.

Another thing that bothered me was that sometimes I'd land on a cool shape but had no easy way to save or organize it. I wanted something like a catalog or gallery of all the interesting forms I'd discovered, so I could easily pick one for a next piece. Also too many manual steps, I thought. I knew I could automate many steps and build a place to store my sketches, and even create a browser-based playground instead of relying on heavy modeling tools.

Building a 3D Torus Knot in p5.js

I wouldn't be writing this article if I hadn't actually brought the idea to life.

While researching algorithmic art, I kept coming across parametric equations for different shapes including torus knot. I had already shown one of the examples I found online earlier [see pic. 2], and it got me thinking: if there's a formula for a shape, then I should be able to visualize it. If I could create a torus knot with lines, why not build it with a tube and then take it a step further by applying visual effects to introduce deformations like twists, lumps, and more?

Just for context, this is the parametric equation for a torus knot and it was my starting point. The entire visualization logic is built around it:

$$x(t) = (R + r \bullet \cos(qt)) \bullet \cos(pt)$$
$$y(t) = (R + r \bullet \cos(qt)) \bullet \sin(pt)$$
$$z(t) = r \bullet \sin(qt)$$

I decided to use the skills 'd built over the years to create a tool that would simplify the whole art creation process for me. I didn't want to automate everything—just enough to turn my hobby into a topic for technical articles and conference talks, so I could do both things I love at the same time (joking, not joking).

Since working with 3ds Max involved a lot of manual tweaking, I figured I could not only generate shapes programmatically but also eventually extract vector paths for printing. (Spoiler alert: that part turned out to be way harder than I thought. I'm still working on it, but I did manage to build a working torus knot gallery.)

I jumped straight into p5.js and spent a few days experimenting, trying to recreate the behavior I'd seen in 3ds Max. On the first day, I somehow managed to generate a perfect torus knot with a structured wireframe tube. Honestly, I didn't think I could pull it off, the math was tricky, and even with Copilot and ChatGPT it took a lot of effort to get the result I wanted.

And that was the easy part.

Over the next few days, I started adding visual effects one by one. There were moments when I thought the project was too complicated, and that I didn't know enough math or JavaScript to finish it. But after a lot of trial and error, and a few "crying and trying" sessions it finally worked the way I imagined.

I'm not going to share the entire implementation here, you can check it out on my GitHub, but this is the final version of the configuration object I use in my p5.js component to generate the visuals:

```
const defaultConfig = {
  // Number of times the knot wraps around
     (major direction)
  p: 3,
  // Number of twists (minor direction)
  q: 8,
  // Number of segments around the tube's cross-
     section
  ringDetail: 20,
  // Number of segments along the knot's path
  pathDetail: 400,
  // Radius of the tube (thickness of the wireframe)
  tubeRadius: 0.15,
  // Overall size of the knot
  knotRadius: 0.85,
  // Amplitude for adding waves along the shape
  waveAmplitude: 0.45,
  // Controls how stretched or "oval" the cross-
     section becomes
  eccentricity: 0.6,
  // Local twisting along the tube
  twistTurns: 20,
 // Overall twisting along the entire knot
  globalTwistTurns: 16.5,
  // Direction of the twist (1 or -1)
  twistDirection: -1,
  // Number of lumps (localized bulges) added
  lumps: 10,
  // Intensity/height of the lumps
  lumpHeight: 0.9,
  // Offset to shift where lumps appear
  lumpOffset: 0.0,
  // Whether to add "electricity" effect
     (chaotic perturbation)
  enableElectricity: false,
  // Strength of the electricity effect
  electricityStrength: 0.15,
  // Frequency (how often) the electricity effect
     happens
  electricityFreq: 2,
  bgColor: "rgb(15, 25, 45)",
  fillColor: "rgba(223, 103, 48, 0.75)",
  wireColor: "rgb(255, 255, 255)"
};
```

Each of these properties plays a role in calculating the final shape, and each one adds a unique visual effect. And if I were to simplify the whole implementation down to just a few lines, here's what it would look like:

```
 const sketch = (s: p5) => {
  const { bgColor } = config;

  let knotPoints: p5.Vector[] = [];
  let ringPoints: p5.Vector[][] = []

  s.setup = () => {
    s.createCanvas(width, height, s.WEBGL);
    s.ortho(-width / 2, width / 2, -height /
    2, height / 2, -1000, 1000);


    s.background(bgColor);
    s.noLoop()

    computeKnotPath();
    computeRingsWithParallelTransport()

    s.rotateX(s.HALF_PI * 2);
    s.scale(scale)

    drawDepthFill();
    drawWireframe();
}

  // ...and then a few hundred more lines of actual
  implementation
};
```

Building the Brains Around the Canvas

I wasn't even thinking about the rest of the application until I finished the torus knot implementation. Because honestly, if that part didn't work the way I wanted, there wouldn't have been much point in continuing. But once I finished playing around in the p5.js editor and realized I now had fully working visualization logic, it became a real project for me.

I didn't just want a tool that added randomization to shapes. I wanted something smarter—something that could also experiment with colors and help me discover interesting combinations. And right now, there's no better fit for such tasks than AI.

Let me explain how I imagined the tool working. I wanted to generate sketches automatically based on user input. Since I had been experimenting pretty randomly anyway, I didn't mind letting automation handle that part. I also wanted a gallery where I could see all the generated results and if I could search for similar sketches based on user input, that would be even better. Maybe in the future, I'll add functionality to adjust the generated visualizations directly in the UI, but that's a plan for later.

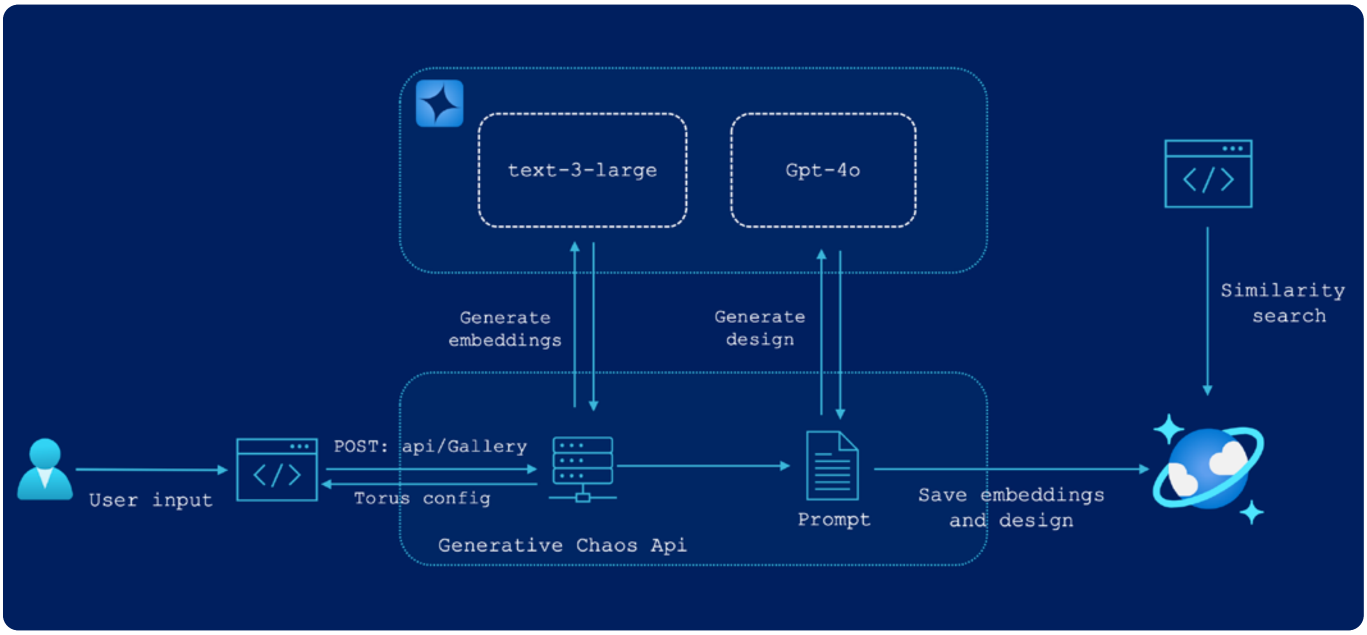
From the technology side, my plan was to use LLMs for the creative part to generate torus knot configuration objects and art descriptions based on user input. For storing the data and handling similarity searches, Azure Cosmos DB was a perfect fit, thanks to its support for vector embeddings and vector search. The previews of my sketches would be saved as images and stored in Azure Blob Storage. On demand, I could also generate 3D object files that could be opened in 3ds Max for further processing.

And because a picture is worth a thousand words, here's a diagram of how it all comes together:

The embeddings are important because they allow us to perform similarity searches across all stored records. My goal was to be able to find documents where the user input was similar, so I could easily compare the results.

I wanted to be able to find all documents where user input was similar so I can compare the results.

```
var embeddings = await _semanticKernelService.
GetEmbeddingsAsync(userInput);
```



Picture 11. High-level diagram of the application

Implementation details

I hope you now have a better idea of what I wanted to build and before I show you how it turned out, I'll walk you through a few key parts of the system, so you can get a sense of how everything works under the hood.

The first step happens in the UI: you type in some input. It can be anything, here are a couple of examples:

```
Example 1: Shape that looks like a start in the blue
color scheme

Example 2: Torus knot with the smooth curves and lots
of twists
```

You get the idea.

Once you click the **Generate** button, an API endpoint is called, and a few things happen. First, we generate embeddings using the text-3-large model. These embeddings are then stored in Azure Cosmos DB.

After that, the system generates both the art description and the torus knot configuration. Here's the `GenerateDesignConfigurationAsync` method, which runs right after the embeddings step.

```
public async Task<(string, TorusConfig)> Generate-
DesignConfigurationAsync(string userInput)
{
  var descriptionFunction = kernel.CreateFunction-
  FromPrompt(Prompts.DesignDescriptionPrompt-
  Template);
  var descriptionResult = await kernel.InvokeAsync
  (descriptionFunction, new KernelArguments {
  ["prompt"] userInput });

  var description = descriptionResult.ToString();
  var configFunction = kernel.CreateFunctionFrom-
  Prompt(Prompts.DesignConfigurationPromptTemplate);
  var configResult = await kernel.InvokeAsync
  (configFunction, new KernelArguments {
  ["$description"] = description });

  var artConfig = Prompts.DefaultConfig;

  var options = new JsonSerializerOptions {
  PropertyNameCaseInsensitive = true };
  var output = JsonSerializer.Deserialize<Torus-
  Config>(configResult.ToString(), options);
  artConfig = output ?? artConfig;

  return (description, artConfig);
}
```

I'm showing you this part of the code to make it clear that the art description is generated from the user input, and then the torus knot configuration is created based on that description, using more context to guide the config generation. Since user input can often be too short or too long, I wanted to first shape it with a specific prompt, so the description would provide a better foundation for generating the configuration object.

Here's what the first prompt looks like:

```
public const string DesignDescriptionPromptTemplate
= @"
  You're an assistant that transforms creative art
  prompts into vivid, artistic visual descriptions
  for generati  artwork based on torus knot
  geometry.
  The user provides a short input or concept, such
  as a mood, shape, emotion or other.
  You elaborate it into a short (1–3 sentence)
  artistic vision.
  The output should mention visual form (e.g.,
  structure, flow), style influences (if relevant),
  and color/emotion.
  Focus on describing the overall look and feel,
  not technical parameters. Input: {{$prompt}}";
```

The second prompt is much larger, so to save space, I'm just showing a part of it here.

```
public const string DesignConfigurationPrompt-
Template = @"You are an assistant that transforms
artistic visual descriptions into valid torus knot
configurations for generative art.
  The configuration will be used to render a 2D
  wireframe visualization of a 3D torus knot usithe
  p5.js library.
  Your job is to generate a balanced, visually
  pleasing configuration in strict JSON forma-
  following the schema and constraints below.
  Ensure all values are valid, visually harmonious,
  and within the allowed ranges.
  ---
  Color Guidance:
  Act as a color theory expert. Generate thoughtful
  and creative combinations.
  - fillColor should reflect the **dominant tone**
    from the input description.
  - bgColor must **contrast with the fillColor**
    to make the shape stand out.
  - wireColor must have **strong contrast with
    fillColor** for wireframe visibility.
  - Use rgb() or rgba() format as specified.
  ---
```

```
  // Here will be more additional rules...

  // And then output schema..."
```

The final step is saving all the information to Azure Cosmos DB.

```
var design = new Design(userInput, embeddings,
description, serialisedConfig);

await _cosmosDbService.InsertDesignAsync(design);
```

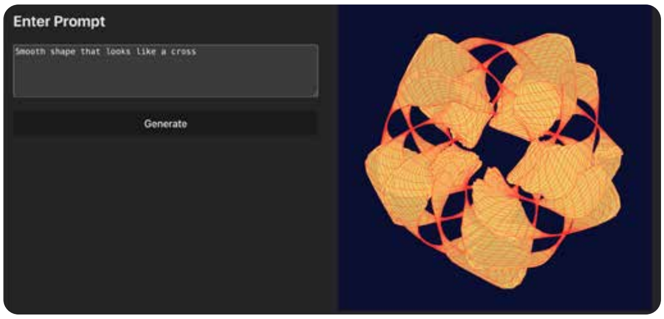
While all this data is being generated, I'm also sending another request to the API using the same user input. I generate embeddings again, and this time I run a similarity search to find related records. Here's how that query looks:

```
const string queryText = $"""
  SELECT Top 5
    c.id, c.userInput, c.generatedDescription,
    c.torusConfig, VectorDistance(c.vectors,
    @vectors) as similarityScore
  FROM c
  WHERE
    VectorDistance(c.vectors, @vectors) >
    @similarityScore
  ORDER BY
      VectorDistance(c.vectors, @vectors)
  """;

  var queryDef = new QueryDefinition(query:
  queryText)
      .WithParameter("@vectors", vectors)
      .WithParameter("@similarityScore", similarity-
      Score);

  using FeedIterator<Design> resultSet = _design-
  Container.GetItemQueryIterator<Design>(queryDef);
```

After the configuration is generated, I create the visualization and once the canvas is rendered, it's saved to Azure Blob Storage for future previews. Another cool thing about the visualization is that, since I'm building the torus knot in WebGL as a 3D shape, I can also generate files that can be opened in 3ds Max for further processing or refinement.

Final result

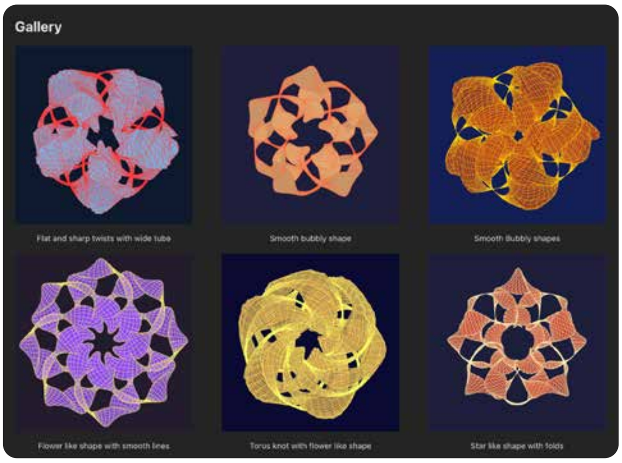
Once again, I included some parts of the implementation but you can find all source code on my GitHub if you want to have a closer look. As promissed here is a final result of how the gallery looks like:



Smooth shape that looks like a cross

Picture 12. Home page: text input for user prompt and the visualized result

I kept the UI as simple as possible, but in the future, I'd like to add some customization options so I can make minor adjustments directly in the interface. And here's another page view:


Gallery
Flat and sharp twists with wide tube
Smooth bubbly shape
Smooth Bubbly shapes
Flower like shape with smooth lines
Torus knot with flower like shape
Star like shape with folds

Picture 13. Gallery view showing previews of the generated results

There's still a lot I want to do and plenty of features I'd like to add, but even at this stage, I already find the tool useful for its intended purpose. I'm now getting ready to create more artworks, and this time, instead of manually sketching, I'll choose from one of the generated designs. I'll keep working on the tool, and hopefully next time, I'll write another article to share how it's evolved over time.

Infrastructure from Code: simplifying cloud deployments

Over the years, I've worked with several infrastructure as code (IaC) tools, and one challenge stands out: the effort required to bridge the gap between application development and infrastructure management. Infrastructure from Code addresses this issue by abstracting infrastructure directly from application logic, making development workflows more seamless and agile. In this article, we will discuss various tools that lead up to Infrastructure from Code. We will dive into what Infrastructure from Code entails and the issues it tries to solve.

Author Erwin Staal

What is Infrastructure from Code?

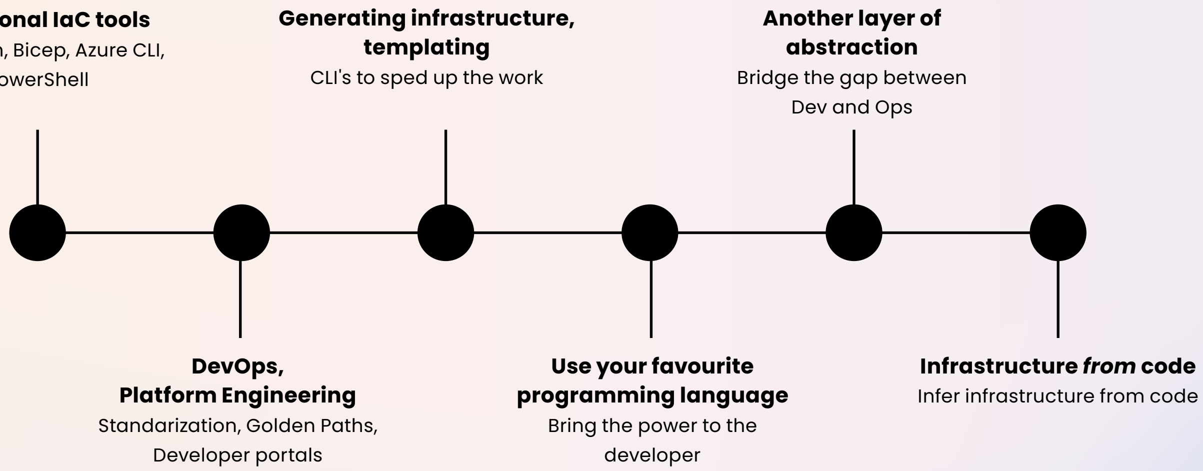
Infrastructure from Code allows you to generate and manage cloud infrastructure directly from application code. Unlike traditional IaC, which involves manually defining infrastructure with tools like Terraform or Bicep, Infrastructure from Code streamlines this process by inferring the required infrastructure from how the application is written. This shift enables developers to focus more on their code, with the confidence that their infrastructure will align seamlessly.

This is important because it provisions infrastructure faster, reducing delays during the development cycle. It ensures application and infrastructure changes are always in sync, which improves consistency across environments.
By reducing the need for developers to understand detailed infrastructure concepts, it simplifies workflows and enables teams to focus more on application development. But is Infrastructure from Code the right approach for you?

Before we dive into an Infrastructure from Code tool and see how that works, we first take a little tour through history to see how we got here and what changes in our industry are responsible for that. We start with looking at the traditional IaC tools and their challenges. Next, we will talk a little about how DevOps and Platform Engineering changed the way we look at these tools and their usage. We will see how tools that build abstractions on top of traditional IaC tools are becoming increasingly common. Finally, we will look at a tool that take this abstraction one step further by automatically provisioning infrastructure based on the application code.

A look back: traditional IaC

Traditional IaC has been very useful for managing infrastructure in a reproducible and predictable way. Tools like Terraform and Bicep make it easier to automate complex cloud environments by using declarative configuration languages.


Traditional IaC tools
Terraform, Bicep, Azure CLI, PowerShell
DevOps, Platform Engineering
Standarization, Golden Paths, Developer portals
Generating infrastructure, templating
CLI's to sped up the work
Use your favourite programming language
Bring the power to the developer
Another layer of abstraction
Bridge the gap between Dev and Ops
Infrastructure *from* code
Infer infrastructure from code

These tools have several advantages:

- **Consistency:** Infrastructure can be defined in a consistent manner, reducing the risk of human error.
- **Scalability:** Infrastructure can be easily scaled up or down based on demand.
- **Collaboration:** Teams can work together on infrastructure code, making it easier to share knowledge and best practices.
- **Documentation:** Infrastructure code serves as documentation, making it easier to understand the architecture.
- **Testing:** Infrastructure code can be tested and validated before deployment, reducing the risk of errors in production.
- **Automation:** Eliminate manual errors and streamline provisioning.
- **Version Control:** Infrastructure definitions can be stored and tracked in repositories.
- **Reusability:** Modular setups allow for better management across regions, environments and teams.

Despite its advantages, traditional IaC still has some pain points:

- **Learning curve:** Tools like Terraform or Bicep require an in-depth understanding of their syntax. For example, newcomers to Terraform often struggle using more complex constructs like for_each or functions like flatten. These can be difficult for those unfamiliar with advanced HCL features. Handling state files in Terraform, whether stored locally or in remote backends like Azure storage, also adds complexity, especially when dealing with collaboration across multiple environments. Additionally, setting up and testing complex Terraform modules for scalability and reuse requires thorough understanding. Another problem is that learning just Terraform or Bicep is not enough. An engineer also needs to know about various other tools and concepts, such as Azure CLI, Azure DevOps, GitHub Actions, and CI/CD pipelines. This can be overwhelming for newcomers.
- **Synchronization overhead:** Keeping application and infrastructure changes aligned can slow down development. If an engineer alters application logic, updating the corresponding infrastructure code is often a separate and time-consuming process. Often, a separate team is needed to manage the infrastructure code, which can lead to wait time and communication overhead.
- **Deep understanding of infrastructure:** Engineers need to understand cloud concepts and services to define infrastructure correctly. This can be a barrier for those who are more focused on application development. For example, setting up a simple website on an Azure App Service requires knowledge of App Service Plans, networking, identity, and other Azure services.
- **Multi cloud support:** While tools like Terraform and Pulumi support multiple cloud providers, managing multi-cloud setups can be complex. Engineers need to understand the nuances of each cloud provider, which can be challenging. Also, I still talk to people who think that when they use Terraform, they can easily switch between cloud providers. This is not the case. While the syntax might be the same, the resources are not and most work needs to be re-done.
- **Complexity in advanced scenarios:** Advanced scenarios, such as cross-region failovers or integrating third-party tools, often require custom scripting or manual intervention. This can lead to increased complexity and potential errors.
- **Tooling and ecosystem:** The ecosystem around IaC tools can be fragmented. For example, while Terraform has a rich set of providers, not all cloud services are covered. This can lead to situations where engineers need to write custom providers or scripts to manage specific resources.

DevOps and Platform Engineering

The rise of DevOps and platform engineering has further complicated the landscape. As organizations adopt DevOps practices, the need for collaboration between development and operations teams has become more and more challenging, especially in large organisations. This shift has led to the emergence of new roles, such as platform engineers, who focus on building and maintaining the infrastructure that supports application development. Platform engineers are responsible for creating self-service platforms that enable developers to deploy and manage their applications without needing deep infrastructure knowledge. This shift has led to the development of tools and frameworks that abstract away the complexities of traditional IaC, allowing developers to focus on writing code rather than managing infrastructure.

Alternatives to traditional IaC

One benefit of tools like Bicep or Terraform is that they do give you full power over the infrastructure that needs to be provisioned. However, this power comes at a cost: the need to understand the underlying infrastructure and the tools themselves. Before we dive into Infrastructure from Code, let’s see how three other tools leverage above tools to make our lives easier. Since each of these tools gives us a higher level of abstracting, we can nicely plot them on a ladder. The higher we go, the less developers need to know about the underlying infrastructure and the tools used to provision it.

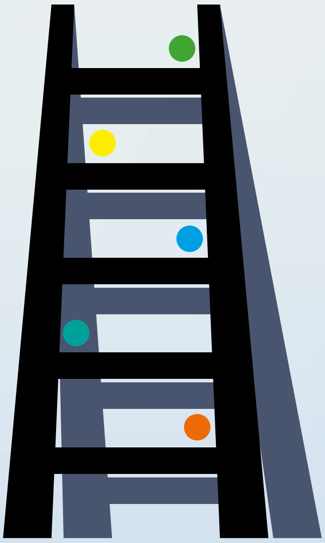
Azure Developer CLI (azd)

First, there is the Azure Developer CLI (azd). Azure azd is a tool that accelerates provisioning and deploying app resources on Azure. Engineers start by creating a project directory and running commands like azd init to initialize the application template. This scaffolds both the code and infrastructure. This might include a C# web application and it's needed infrastructure like Azure App Services, Function Apps, or Cosmos DB, storage, all based on the selected template. The underlying infrastructure is defined using either Bicep or Terraform. The default is Bicep, but you can switch to Terraform, if you prefer, which is still in beta.

Using azd up, developers can deploy the infrastructure and application in a single step, which simplifies the lifecycle of resource creation, application deployment, and environment setup. For example, if an engineer is deploying a web application, azd up handles tasks like creating the App Service Plan, provisioning a resource group, and applying appropriate configurations—all driven by the application's needs.

This approach solves problems such as the time-intensive coordination between development and DevOps teams, misaligned application requirements, and redundant scripting for repetitive deployments. Furthermore, Azure azd enables developers to maintain consistency across environments by defining everything as part of the project template. It also helps developers focus on application logic, abstracting the underlying infrastructure details which they might not have enough knowledge on.

However, Azure azd has limitations. It is primarily tailored to Azure-specific services, which means engineers cannot use it in multi-cloud strategies. Additionally, while it simplifies standard use cases, customizing infrastructure for advanced scenarios—like cross-region failovers or integrating third-party tools—often requires supplementing azd with traditional IaC tools like Terraform or ARM templates. I therefore see Azure azd more as a tool that simplifies the standard use cases, to quickly get started but not as a replacement for traditional IaC tools.



Nitric
IfC tools take another step in abstraction by generating infrastructure based on the applications logic

Radius
Abstracting the layer between dev and ops allows for developers to quickly launch new apps while operators can support them

Azure Developer CLI
Using a higher level of abstraction, tools can shorten the time to market for new apllications

Pulumi
Pulumi allows one to write IaC in their favorite tooling and offers a suite of additional features

Terraform, Bicep
Traditional IaC tools provide the groundwork for more abstract and advanced tooling

Pulumi

Pulumi is another tool that (mainly) uses Terraform under the hood to perform its provisioning. Pulumi allows developers to define infrastructure using familiar programming languages like C#, Python, TypeScript, or Go. This enables developers to use the same language for both application and infrastructure code, reducing context switching and enabling a more seamless development experience.

Pulumi solves a few challenges engineers normally run into with traditional IaC, like debugging the written code. It also enables developers to reuse existing programming constructs, such as loops and conditionals, which are familiar and expressive. They can also leverage the same concepts to share code between teams and projects, which can be beneficial for scaling infrastructure setups or setting compliant standards.

Despite its advantages, Pulumi does have limitations. It requires developers to have programming skills beyond standard IaC tool syntax, which might be a barrier. I therefor see Pulumi only useful for application developers that already have a deep understanding of a programming language. For those that don't, it might be better to stick with traditional IaC tools. One benefit of Azure azd is that it abstracts the underlying infrastructure details, which makes it easier for developers to get started. Pulumi, on the other hand, requires a deeper understanding of cloud infrastructure concepts like as was needed when using Terraform or Bicep.

```csharp
using Pulumi;
using Pulumi.AzureNative.Resources;
using Pulumi.AzureNative.Storage;
using Pulumi.AzureNative.Storage.Inputs;
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading;

return await Pulumi.Deployment.RunAsync(() =>
{
    while (!Debugger.IsAttached)
    {
        Thread.Sleep(100);
    }

    // Create an Azure Resource Group
    var resourceGroup = new ResourceGroup
    ("rg-pulumi-debug");

    // Create an Azure resource (Storage Account)
    var storageAccount = new StorageAccount
    ("stgpulumidebug", new StorageAccountArgs
    {
        ResourceGroupName = resourceGroup.Name,
        Sku = new SkuArgs
        {
            Name = SkuName.Standard_LRS
        },
        Kind = Kind.StorageV2
    });
});
```

Radius

Radius was originally created by Microsoft's incubation team but is now open source and a CNCF project. It is a cloud-native application platform designed to help not only developers but also the the infrastructure operators that support them. The goal is to make cloud-native application development more efficient and accessible, regardless of the cloud platform you land on. Radius provides this self-service platform that allows developers to focus on their core responsibilities. For some, that means coding microservices; for others, it means creating application infrastructure.

Cloud-native applications become more and more complex. The synergy between developers and IT operators therefor becomes more and more crucial. Radius facilitates collaboration between these two key teams. Here's how it works:

- **Defining Your Application:** Developers define their applications in Radius, including all the services, dependencies, and relationships between them, where they normally would do that in infrastructure or Kubernetes resources. This approach streamlines the development process, making it easier to manage complex applications.
- **Crafting Environments:** In parallel, operators define environments within Radius, encapsulating infrastructure templates, policies, and organizational requirements specific to their chosen platform, whether it's Azure, AWS, or self-hosted. This ensures that your applications can run smoothly on any cloud platform.

The code snippet below shows how to define a simple application in Radius. The code is written in Bicep, which is the underlying language used by Radius. The code defines a simple web application that uses a Redis store. The application is defined as a container, and the Redis store is defined as a datastore. Notice how we didn't specify detailed information about the underlying infrastructure. This is all handled by Radius.

```bicep
extension radius

@description('The Radius Application ID. Injected
automatically by the rad CLI.')
param application string

@description('The environment ID of your Radius
Application. Set automatically by the rad CLI.')
param environment string

resource demo 'Applications.Core/containers@2023-
10-01-preview' = {
  name: 'demo'
  properties: {
    application: application
    container: {
      image: 'ghcr.io/radius-project/samples/
      demo:latest'
      ports: {
        web: {
          containerPort: 3000
        }
      }
    }
    connections: {
      redis: {
        source: db.id
      }
    }
  }
}

resource db 'Applications.Datastores/redisCaches@
2023-10-01-preview' = {
  name: 'db'
  properties: {
    application: application
    environment: environment
  }
}
```

Running the command 'rad up' will deploy the application to your local cluster—for example, using K3D. The translation between the types defined in above example and what infrastructure is actually run is done by what we call 'Recipes' in Radius. So, when you run 'rad up', Radius will look at the application and environment definitions, and then use the recipes to deploy the application to the local cluster. When deploying to e.g. Azure, Radius will use the Azure recipes to deploy the application to Azure. The image below shows how Radius uses the recipes to deploy the application to Azure.

Infrastructure from Code

Using any of the tools above, you still need to define the infrastructure. Its either done directly through e.g. Terraform or by using a tool that abstracts the infrastructure like Azure azd. Infrastructure from Code takes this abstraction one step further by automatically provisioning infrastructure based on the application code. This approach is particularly powerful for serverless and event-driven applications, where infrastructure components, like APIs, message queues, or databases, are tightly coupled to code.

There are various tools that implement this concept in one way or another. Let's compare some of these tools at a high level before we dig into a few to see them in action.
When picking a tool, there are multiple factors we can look at. For example, what cloud platforms can they target, what programming languages do they support and what IaC language do they use under the hood. Here's a table of the tools and the supported cloud platforms.

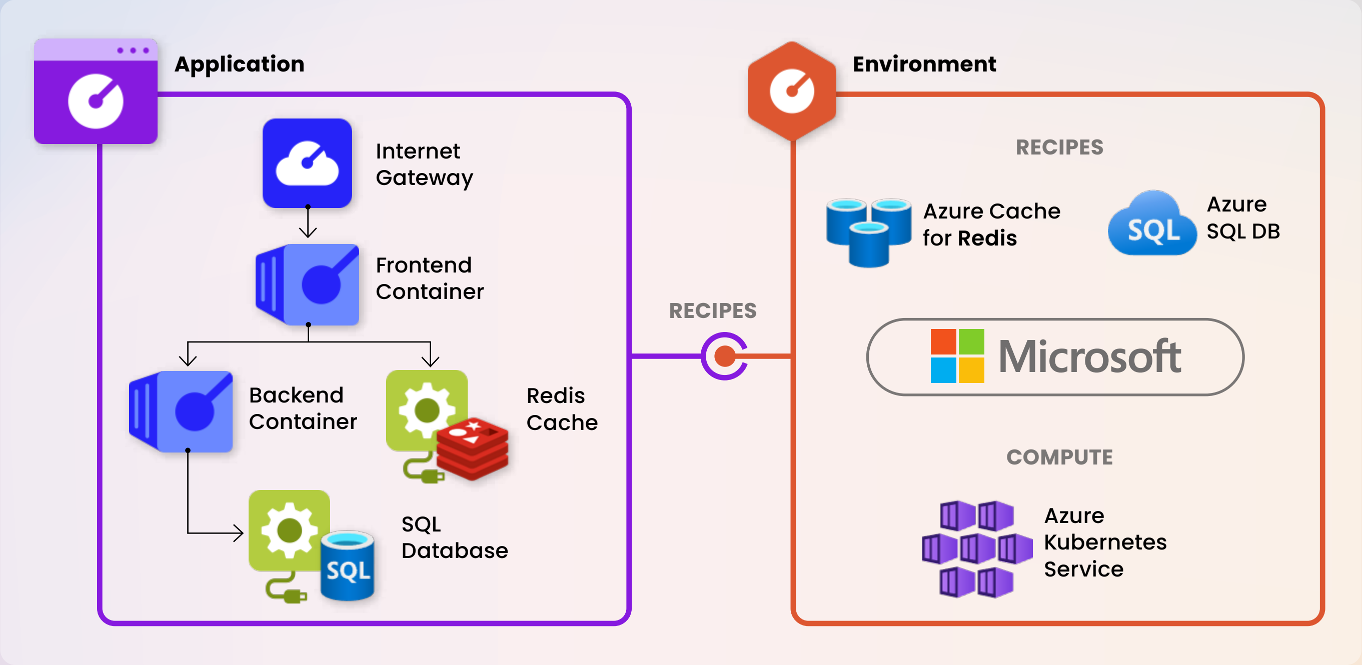
Tool	Azure	AWS	GCP
Nitric	X	X	X
Klotho		X	
Ampt		X	
Encore		X	X

Supported programming languages:

Tool	.Net	Java	Python	Go	TypeScript	Node.js
Nitric	X*	X	X	X	X	
Klotho			X	X		
Ampt					X	X
Encore				X	X	X

Underlying IaC language:

Tool	Terraform	Pulumi	Other
Nitric	X	X	X
Klotho			X
Ampt			X
Encore			X



After reviewing this list, I decided to dig into Nitric. This tool allows me to target Azure and write my code in a language that I know (sort of 😉), TypeScript. There is (very) experimental support for C#, so I decided not to take that route.

Nitric

Nitric offers a framework for building backends. It's a declarative framework with common components like APIs, queues, and databases. Nitric abstracts the underlying infrastructure, allowing developers to focus on building applications but also allowing developers to interact with these resources. It becomes easy to swap underlying services as they are pluggable.

A simple API endpoint in Nitric looks like this:

```
import { api } from '@nitric/sdk'

const main = api('main')

main.get('/hello/:name', async ({ req, res }) => {
  const { name } = req.params
  ctx.res.body = `Hello ${name}`
})
```

The above code is called a service in Nitric. It's the entry point for the application. You can define one or multiple services. By importing the SDK and creating an instance of the api type, you're instructing Nitric to expose an API. When we later run Nitric and deploy to Azure, this will create an Azure Container App to run the application on and will create an API Management instance to expose and test the API.

In a config file, like the one below, you tell Nictric where to look for the services and how to run them.

```
name: example
services:
  - match: services/*.ts
    start: npm run dev:services $SERVICE_PATH
```

Then we have the concept of resources. These are the underlying infrastructure components that the services uses. In the example below, I'm using a storage resource.

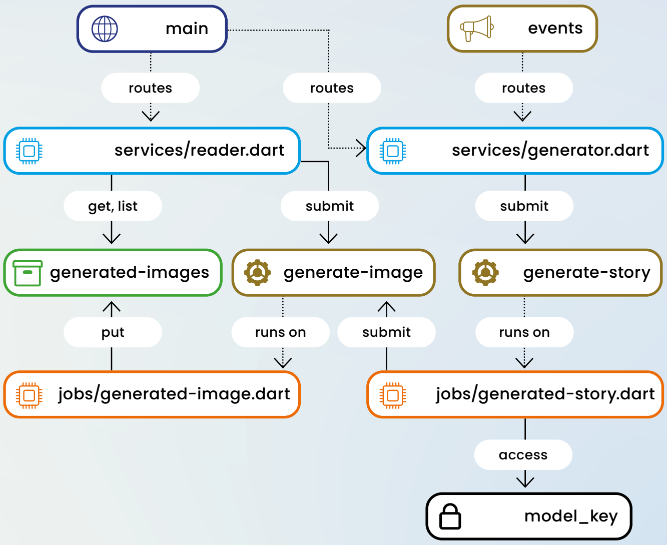
```
import { bucket } from '@nitric/sdk'

const uploads = bucket('uploads').allow('read')

const files = await uploads.files()

files.forEach((file) => {
  console.log(file.name)
})
```

The above bucket will be deployed in Azure as a Storage Account. The bucket is then used to store files in blobs and the code above lists all the files in the bucket. See how easy it is to define your requirements without writing a single line of infrastructure code?

But how does Nitric know how to deploy the resources? This is done in a nitric..yaml file. In this file, you define the provider to be used. In the example below, I'm using the Pulumi provider to target Azure.

```
provider: nitric/azure@1.1.1
region: northeurope
```

With that in place, it's time to run our app. You can do that locally by running 'nitric start' from a terminal. All services will then be emulated locally using containers. It also provides a Dashboard UI to interact with the resources.



When you now run 'nitric up', your application will be deployed to the cloud. In my case, I see various resources like an API Management instance, Azure Storage Account, and a Function App being created. The API Management instance is used to expose the API endpoints. The Azure Storage Account is used to store files in blobs. The Function App is used to run the application code. All without us defining a single line of infrastructure.

When and why to use Infrastructure from Code

Now we've seen an example of how we can levarage the power of IfC using Nitric, let's talk about where I see it fit. Infrastructure from Code is best suited for situations where developer velocity and simplicity matter more than fine-grained infrastructure control. It shines particularly in the following contexts:

Startups and MVPs

Startups often need to ship quickly without dedicating time to infrastructure design. IfC tools allow them to:

- Focus purely on writing product code.

- Automatically deploy infrastructure without needing much support from a platform team.
- Iterate fast and pivot without worrying about cloud plumbing.

Event-driven applications

Many IfC tools are designed with cloud-native, event-driven patterns in mind (e.g., HTTP routes, queues, functions). They're ideal for:

- APIs triggered by HTTP or events.
- Background workers, cron jobs.
- Serverless use cases that glue services together.

Internal tools and prototypes

For internal apps that solve operational problems or automate workflows:

- Simplicity and speed often outweigh the need for enterprise-grade infra design.
- IfC lets devs roll out tools without engaging infra/platform teams.

Developer-driven teams

In teams where developers own the full lifecycle (build → deploy → monitor), IfC reduces the need to learn Terraform, Bicep, or YAML.

- It supports the "you build it, you run it" model.
- Lower barrier to cloud deployment = more empowered devs.

Considerations and future outlook

A tool like Nitric looks really promising and makes creating a new application really easy. The tool, and other IfC tools as well, don't really feel mature yet. I got quite a few errors along the way. Most of them were quickly fixed in a new version but still. The support for various cloud resources is also quite small. While I really do see this new approach working in the future, it also introduces some additional considerations:

- **Abstraction overhead:** Too much abstraction might reduce control.
- **Cloud or vendor lock-In:** Tools can sometimes be tightly bound to specific providers or vendor SDKs, in the case of Nitric.
- **Complex scenarios:** Advanced customizations might still rely on traditional IaC techniques.

Infrastructure from Code offers a transformative approach to cloud development. By focusing on application-driven workflows and abstracting the details of infrastructure setup, developers can move faster and reduce operational complexity. Tools like Nitric can really speed up your development processes but are not for everyone. Any of the other mentioned tools can already be a significat step up compared to using traditinal IaC.

Below you will find a table that lists the tools mentioned in this post and summarizes their properties like abstraction level, language used, and cloud support. This should help you to pick the right tool for your needs.

Tool	Infra Defined In	Abstraction Level	Language Used	Cloud Support	Best For
Terraform	Declarative (HCL)	Low	HCL	Multi-cloud	Infra teams comfortable with code
Bicep	Declarative (Bicep)	Low	Bicep	Azure	Azure-focused infra teams
Pulumi	Imperative (Code)	Medium	TypeScript, Python, Go, C#	Multi-cloud	Developers preferring real languages
azd	Templates + IaC	Medium	CLI (Bicep/Terraform under the hood)	Azure	Quick app bootstrapping on Azure
Radius	Recipes + IaC	High	Config + Bicep	Azure, AWS, Google, Kubernetes	Platform teams enabling developers
Nitric	In Application Code	Highest	TypeScript, Python, Go	Multi-cloud	Serverless and event-driven apps

From Concept to Creation: Developing Intelligent AI Agents for Modern IT Solutions

Theory - What you should know!

Imagine having an intelligent system that independently analyzes problems, makes decisions, and continuously improves - exactly the capability provided by modern AI agents. This next generation of automation not only surpasses traditional workflows but opens entirely new avenues in IT operations, support, and beyond.

Author **Thomas Tomow**

In this article, we will explore how AI agents work, what technologies power them, and how to build your own AI agent with Semantic Kernel. Step by step, we'll create an intelligent system that interacts with Large Language Models (LLMs) and flexibly approaches problems with self-developed solutions. Ready to dive into the world of AI agents? Then, let's get started!

Artificial intelligence has made great strides. Unlike traditional systems with fixed rules, modern AI agents adapt dynamically, learn from experience, and make decisions independently. These systems solve complex tasks by learning from interactions.

In this article, we will examine the core concepts behind AI agents, trace their evolution from traditional workflow engines to adaptive, autonomous systems, and guide you step by step through building your own AI agent using the Semantic Kernel.

Timeline and development

Early 2000s: Traditional workflow engines

In the early 2000s, automation was largely dominated by workflow engines such as Nintex, Windows Workflow Foundation, IBM WebSphere Process Server, and BizTalk Server. These tools enabled companies to automate business processes without in-depth programming for the first time.

Features:

- Workflow engines were characterized by rigid, predefined logic and decision trees.
- Developers implemented automation using if-then-else conditions and rule-based configurations.
- The focus was on repeatable, structured tasks with clearly defined results.
- Systems were not very adaptable and required extensive reconfigurations during process changes.

While these tools increased efficiency in structured environments, they were limited in processing unstructured data and adapting to dynamic business needs.

The Mid-2010s: The Rise of AI and Machine Learning

While rule-based systems dominated in the early 2000s, a new era of data-driven approaches began in the 2010s. With the addition of machine learning, the focus shifted to more automated decision-making. This led to more flexible and intelligent automation solutions based on patterns and predictions.

Important Developments:

- **Adoption of predictive analytics:** Systems began to use data patterns to make predictions and reduce manual intervention. (SAS Advanced Analytics, IBM SPSS Modeler, RapidMiner)
- **The transition from rule-based to learning-based models:** AI applications analyzed historical data and adapted to changing conditions (TensorFlow (2015), Microsoft Azure Machine Learning (2015), H2O.ai)
- **Early applications:** fraud detection, recommender systems, and process optimization. (PayPal Fraud Detection, Amazon Recommendation Engine, Google RankBrain (2015))

These changes laid the foundation for more intelligent automation, but AI was still primarily used to complement rule-based workflows rather than as a standalone decision-maker. (UiPath, Automation Anywhere, Blue Prism)

Early 2020s: Emergence of Generative AI

At the end of the 2020s, generative AI emerged with transformer models – a special machine learning architecture – and its most prominent model type, GPT, demonstrated remarkable abilities in understanding and generating natural language. Initially, results were modest, but rapidly evolved to highly convincing outputs. With the ability to create human-like texts and provide contextual responses, AI was no longer seen merely as a supportive tool but as a decision-making authority in its own right.

Key features:

- AI systems became more context-aware and could understand and generate human-like texts.
- Generative AI sparked an interest in intelligent automation that went beyond traditional workflows.
- Applications expanded to include chatbots, virtual assistants, and AI-powered content creation.

These advances have laid the foundation for AI agents – autonomous systems that make decisions, learn from interactions, and continuously improve.

Present: AI Agents and Agent-Based Systems

With the continued advances in AI, we are now on the cusp of agent-based technology – autonomous entities that sense their environment, make decisions, and act to achieve specific goals.

Description of AI agents

Unlike traditional workflows, AI agents can process **unstructured data** and make complex decisions.
More advanced agents, can also **learn from interactions**, by continuously improving their performance and decisions. Those AI agents are **goal-oriented** and dynamically adapt their behavior to new circumstances. They use advanced techniques such as **reinforcement learning**, **self-supervised learning**, and **deep neural networks** to optimize their decision-making ability. An agent framework by BMW[1] or a solution like MADDOX from Peerox GmbH in Germany are examples of self-learning and adaptive systems.

AI agents represent a paradigm shift — from **static automation** to adaptive, **self-improving systems** that can operate in dynamic, real-world environments (see Figure 1 below).

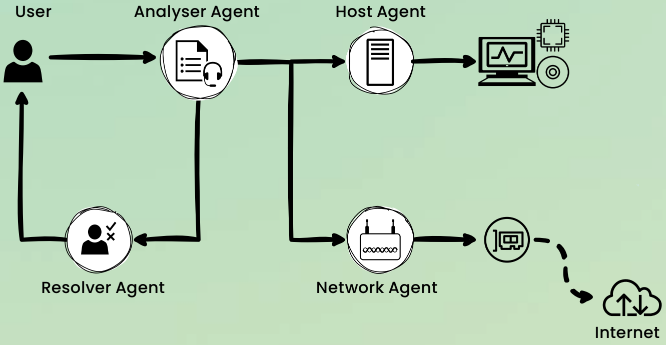


Figure 1. AI Ops Agents – A group of AI agents that address problems on the local system

Although the definition of "agent" may vary, agent-based systems (often called agentic or multi-agent systems) empirically demonstrate scalability by orchestrating specialized groups of AI agents (e.g., for network analysis) within a decentralized, modular framework — allowing additional agents to be integrated seamlessly without a complete redesign of the overall system. (see Figure 2).

For example, while a single AI agent might only answer the question, "Why is my network so slow?", an agentic system leverages the capabilities of entire groups of AI agents (covering network, application monitoring, ticketing systems, etc.) to solve more complex problems.

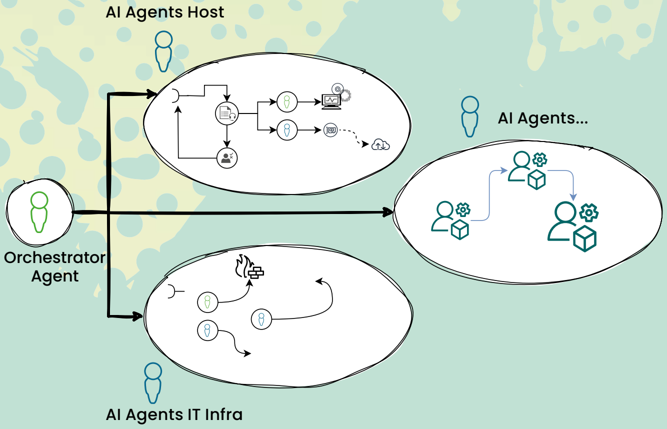


Figure 2. Agentic System

Essential AI Technologies for AI Agents

AI agents leverage key artificial intelligence technologies to analyze data, understand speech, and process visual information. Machine learning, a subset of artificial

---

[1] https://arxiv.org/abs/2406.20041

intelligence, enables AI agents to recognize data patterns and make predictions or decisions based on accumulated experience. This gives rise to the following specializations:

- **Neural Networks and Deep Learning**
  Neural networks inspired by brain structures help AI agents understand complex data. Deep learning leverages multi-layered networks for tasks such as image and speech recognition.
- **Natural Language Processing (NLP)**
  NLP enables communication in human language, which is essential for chatbots, virtual assistants, and automated systems. It underpins large language models – research in this domain has led to their creation. Examples include sentiment detection, intent recognition, or entity detection applied either within or on top of LLMs.
- **Computer Vision**
  AI agents interpret visual data and make decisions that are useful in robotics, quality control, and face recognition.
- **Reinforcement Learning**
  AI agents learn optimal decision-making through reward feedback systems, particularly in dynamic environments.
- **Transformer Models and LLMs**
  Transformer models, a neural network architecture, enable efficient text processing and generation. They form the basis for LLMs, which are trained on vast amounts of textual data, enabling advanced language processing, comprehension, and generation.

Practice - Create Your Own AI Agent!

Now that we have explored the evolution and core concepts of AI agents – from rule-based systems to autonomous, self-improving entities – it's time to put theory into action.

In this practical section, we will build a working AI agent using Microsoft's Semantic Kernel. This framework acts as a bridge between your application and powerful Large Language Models (LLMs), making it easy to integrate intelligent behavior into your software.

Step by step, we will walk through setting up a .NET project, integrating Semantic Kernel, and creating a team of specialized agents that can analyze and resolve IT issues autonomously. Whether you're building a smart assistant, an AI Ops tool, or just experimenting with agent-based systems, this guide will give you the foundation you need to get started.

Let's dive in and create your first intelligent AI agent.

Brief Introduction to Semantic Kernel

Semantic Kernel is a framework that acts as middleware between applications and LLMs. It provides an abstraction layer that allows developers to easily integrate AI capabilities into their software. Thanks to its modular structure, different AI models and plugins can be flexibly combined.

Key features:

- **Easy integration:** Integrates seamlessly with existing .NET projects.
- **Modularity:** Allows flexible adaptation of AI components to specific requirements.
- **Scalability:** Supports deployment in large environments and distributed systems.

Developers can use Semantic Kernel to create powerful AI agents and extend their applications with intelligent features.

Several frameworks exist for developing AI agents, including LangChain/Graph, AutoGen / AG2, and Semantic Kernel. In this article, we will use Semantic Kernel due to its flexible and modular approach for integrating large language models (LLMs) into applications. At the time of writing, the Agent Framework within Semantic Kernel is still in preview but is under active development by both Microsoft and the open-source community – ensuring continuous improvements and long-term support.

Semantic Kernel is particularly well-suited for building LLM-based AI agents because of its strong focus on language model orchestration and plugin extensibility. It supports multiple programming languages, including C#, Python, and Java, making it accessible to a wide range of developers.

Step-by-Step Guide: Integrating the Semantic Kernel with the Agent Framework

We begin by setting up a new project, adding the necessary dependencies, and configuring access to an LLM. By the end of this section, you will have a working base for your AI agent.

In the following steps, we will:

- Create a new .NET Project
- Install the Semantic Kernel NuGet packages
- Build a simple AI agent
- Evolve it into a team of agents

Preconditions:

- **Technical knowledge:** Experience with C#, .NET and Terminal commands
- **Access to an LLM, such as:**
  - an (Azure) OpenAI GPT model
  - a locally hosted Ollama model

Start with an example: "AI Ops"

We are planning an analysis system for AI Ops that uses generative AI to automate the resolution of IT tickets.

The goal is for AI agents to analyze and solve problem descriptions. These agents will need access to the network, CPU, and memory. Our team will consist of an analysis agent and an evaluation agent. The system should be capable of independently finding problems and proposing expert-level solutions (see Figure 1 AI Ops Agents).

Create a New .NET Project:

```
dotnet new console -o MyAIAgents
cd MyAIAgents
```

Installing Semantic Kernel

To work with Semantic Kernel Framework, add the following Nuget packages:

```
dotnet add package ...
• Microsoft.SemanticKernel
• Microsoft.SemanticKernel.Agents.abstractions ↩
  --version 1.35.0-alpha
• Microsoft.SemanticKernel.Agenten.core ↩
  --version 1.35.0-alpha
• Microsoft.SemanticKernel.agenten.openai ↩
  --version 1.35.0-alpha
```

> ⓘ **NOTE**
>
> Some of these packages are still in preview, and version numbers are subject to change. Please refer to the official Semantic Kernel documentation if you are using a newer version.

Access to an LLM

To create an AI agent, you need access to an LLM. We mentioned three common options here:

- **Azure OpenAI:** Create an Azure OpenAI service in the Azure portal and note the API key.
- **OpenAI API:** Register on the OpenAI website and generate an API key.
- **Local models:** Use Ollama for a locally hosted model.

For this project, we use the Azure OpenAI Service. OpenAI's API requires only minor changes in the code, as both approaches are implemented almost identically in Semantic Kernel.

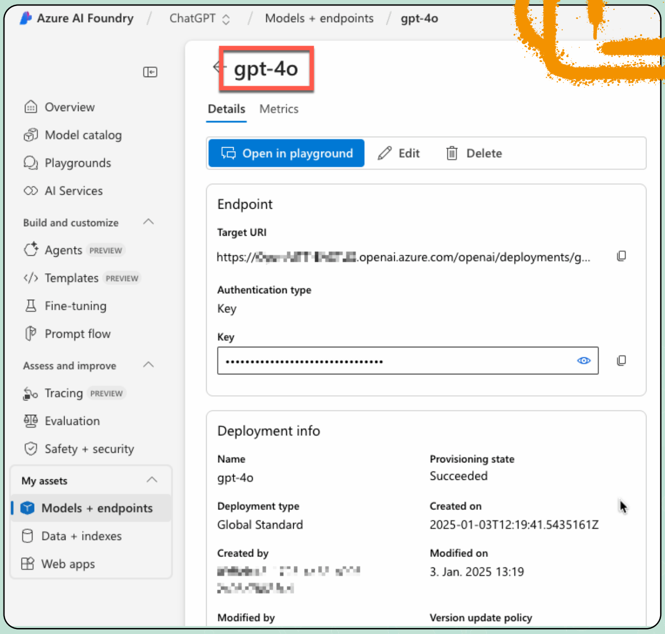
Set Environment Variables

We typically store configuration details such as API keys in environment variables. This approach simplifies moving the project to different environments, including Docker. (A version using standard .NET settings is also available in our GitHub repository.)

Create a file called `.env` in the project directory and add:

```
AZURE_OPENAI_API_KEY=<your API Key>
AZURE_OPENAI_ENDPOINT=<your endpoint>
AZURE_OPENAI_MODEL_ID=<your deployment model>
```

> ⓘ **NOTE**
>
> The deployment model is the name of the model you've deployed to Azure OpenAI.



gpt-4o

Endpoint

Target URI
https://[redacted].openai.azure.com/openai/deployments/g...

Authentication type
Key

Key
••••••••••••••••••••••••••••••••

Deployment info

Name gpt-4o	Provisioning state Succeeded
Deployment type Global Standard	Created on 2025-01-03T12:19:41.5435161Z
Created by [redacted]	Modified on 3. Jan. 2025 13:19
Modified by	Version update policy

Figure 3. Model deployment name in Azure Foundry

Install the `DotNetEnv` package to load the environment variables in your code:

```
dotnet add package DotNetEnv
```

> ⓘ **NOTE**
>
> Also create a `.gitignore` file and add .env to avoid storing sensitive information in your Git repository.

After adding Semantic Kernel to our .NET project, we can create a simple AI agent that interacts with an LLM. Let's start writing code.

Initializing Semantic Kernel

We next adjust Program.cs by:

- Building the Kernel
- Setting up Semantic Kernel for connection to an LLM

Program.cs

```
class Program
{
    static async Task Main(string[] args)
    {
        //Alternative to appsettings.json..:
        DotNetEnv.Env.Load();  //load the Env Vars

        //Initializing the kernel
        var builder = Kernel.CreateBuilder();
        Builder.AddAzureOpenAIChatCompletion(
        Environment.GetEnvironmentVariable
        ("AZURE_OPENAI_MODEL_ID"),
        Environment.GetEnvironmentVariable
        ("AZURE_OPENAI_ENDPOINT"),
        Environment.GetEnvironmentVariable
        ("AZURE_OPENAI_API_KEY"));
        var kernel = builder.Build();

        //Creating a clone to provide a network
        capability
        var networkKernel = kernel.Clone();
        var pluginNetwork = KernelPluginFactory.
        CreateFromType<NetworkMonitor>();
        networkKernel.Plugins.Add(pluginNetwork);

        //Second kernel with host monitoring
        capability
        var hostMetricsKernel = kernel.Clone();
        var pluginHost = KernelPluginFactory.
        CreateFromType<HostMetrics>();
        hostMetricsKernel.Plugins.Add(pluginHost);
        // Logging and housekeeping omitted for
        brevity
    }
}
```

The method `DotNetEnv.Env.Load()` loads environment variables from the .env file and makes them available in the program, which is especially useful for managing API keys without embedding them in code. First, the `KernelBuilder` is created. With `AddAzureOpenAIChatCompletion`, the kernel is configured to interact with the appropriate LLM. For those opting for OpenAI's API, use `builder.AddOpenAIChatCompletion` instead.

We clone the kernel to add specialized functionality through plugins, such as network checks or host metrics. Cloning allows for a separation of responsibilities and prevents agents from calling functions they should have no access to. An agent can only use the plugins present in the kernel to perform its intended tasks. In our example, we have an agent that can perform network checks:

```
KernelPluginFactory.CreateFromType<NetworkMonitor>();
networkKernel.Plugins.Add(pluginNetwork);
```

Below is an excerpt from the plugin class to illustrate its implementation.

NetworkMonitor class

```
[KernelFunction()]
[Description("Returns information about the network
adapters")]
public static List<string> adapterInfos()
{
    List<string> adapterInfo = new List<string>();
    foreach (NetworkInterface ni in NetworkInterface.
    GetAllNetworkInterfaces())
    {
        string info = $"Name: {ni.Name},
        type: {ni.NetworkInterfaceType},
        status: {ni.OperationalStatus}";
        adapterInfos.Add(info);
    }
    return adapterInfos;
}
```

Semantic Kernel leverages method decorators, such as **`KernelFunction`** and **`Description`**, to expose functions to the LLM while preventing direct system access. In this way, when the LLM requires information on network adapters, it simply triggers the adapterInfos() function, which securely returns the data.

Our first agent

Let's define our first AI agent – a network analyst.

```
ChatCompletionAgent agentNetworkChecker = new()
{
    Name = nameNetworkChecker,
    Instructions =
    """
    **Role:** You are a Network Checker with over
    10 years of experience. You specialize in
    diagnosing network connectivity issues.

    <… (shortened) The prompt defines the role of
    the agent and provides instructions on how to
    analyze network problems>
    """,
    Kernel = networkKernel,
    Arguments = new KernelArguments(new AzureOpenAI-
    PromptExecutionSettings()
        {FunctionChoiceBehavior = FunctionChoice-
        Behavior.Auto() }
            ),
    LoggerFactory = loggerFactory
};
```

Each agent is defined by a name, a prompt (instructions), and a kernel. The prompt instructs the LLM regarding its role, and the arguments allow it to decide whether to call a plugin function or produce a direct response. In this case, the agent is configured with network-specific capabilities via the network kernel.

Next, create a ChatHistory object to manage the conversation.

```
ChatHistory chat = [];

//Add the user message
chat.Add(new ChatMessageContent(AuthorRole.User,
"<user input>"));

//Generate the agent response(s)
await foreach (ChatMessageContent response in
agentNetworkChecker.InvokeAsync(chat))
{
  //Process the answer(s)...
}
```

An Agent System

While a single AI agent is useful, their full potential is realized when multiple agents work together. In our system, various specialized agents collaborate to handle complex tasks.

In addition to the network agent, you can define agents for CPU/storage checks, analysis, and resolution:

```
ChatCompletionAgent agentNetworkChecker = ... ;
//checks the network
ChatCompletionAgent agentCommonChecker = ... ;
//checks CPU, storage, ...
ChatCompletionAgent agentAnalyst = ... ;
//Analyzes agent results
ChatCompletionAgent agentResolver = ... ;
//evaluates the analysis result
```

To coordinate these agents effectively, we define an **`AgentGroupChat`** structure. This setup controls which agent speaks next and when the analysis ends, utilizing a **`SelectionStrategy`** and a **`TerminationStrategy`**.

```
AgentGroupChat groupChat =
new(agentAnalyst, agentNetworkChecker, agentResolver,
agentCommonChecker)
{
    ExecutionSettings = new()
    {
        SelectionStrategy = new KernelFunction-
        SelectionStrategy(selectionFct, kernel)
        {
            InitialAgent = agentAnalyst,
            HistoryReducer = new ChatHistory-
            TruncationReducer(1),
            HistoryVariableName = "lastmessage",
            return the response from the current
            agent/or its name
            ResultParser = (result) =>{
                var selection = result. GetValue
                <string>() ?? agentAnalyst.Name;
                return selection;
            }
        },
        TerminationStrategy = new KernelFunction-
        TerminationStrategy( terminationFct, kernel)
        {
            agents = new[] { agentResolver },
            HistoryVariableName = "lastmessage",
            HistoryReducer = new ChatHistory-
            TruncationReducer(1),last message
            MaximumIterations = 10,
            // maximum 10 rounds
            ResultParser = (recall) => {
                //check agent response for presence
                of the word "YES"
                var result = recall.GetValue-
                <string>();
                var containesYes = result?.ToLower().
                Contains("yes");
                return containsYes ?? false;
            }
        }
    }
};
```

Our `SelectionStrategy` determines which agent responds next based on the previous message; the `TerminationStrategy` defines when to end the dialogue, such as when a "YES" response is detected or after a maximum number of rounds.

Following we write a `SelectionStrategy` template, orchestrating agents by their subsequent responses.

```
KernelFunction selectionFct = AgentGroupChat.
CreatePromptFunctionForStrategy(
$$$"""
**Task:** Determine which agent should act next ba-
sed on the last response. Respond with **only** the
agent's name from the list below. Do not add any ex-
planations or extra text.

**Agents:**

- {{{nameAnalyst}}}
- {{{nameNetworkChecker}}}
- {{{nameCommonChecker}}}
- {{{nameResolver}}}

**Selection Rules:**

- **If the last response is from the user:** Choose
  **{{{nameAnalyst}}} **.
- **If the last response is from {{{nameAnalyst}}}
  and they are requesting network details:** Choose
  **{{{nameNetworkChecker}}}**.
- **If the last response is from {{{nameAnalyst}}}
  and they are requesting non-network system
  checks:** Choose **{{{nameCommonChecker}}}**.
- **If the last response is from {{{nameAnalyst}}}
  and they have provided an analysis report:** Choose
  **{{{nameResolver}}}**.
- **If the last response is from {{{nameNetwork-
  Checker}}} or {{{nameCommonChecker}}}:** Choose
  **{{{{nameAnalyst}}} }**.
- **Never select the same agent who provided the
  last response.**

**Last Response:**

{{$lastmessage}}
""",
safeParameterNames: "lastmessage"
);
```

This template clearly outlines which agent should respond based on the content of the last message. The placeholders like `{{{nameAnalyst}}}` are replaced at runtime with the actual agent names. The variable `lastmessage` is made available via the `HistoryVariableName` property, ensuring only the relevant message is used for agent selection.

The `TerminationStrategy` is managed similarly to the `SelectionStrategy`, determining the chat ending.

Finally, we initiate the group conversation by adding a user message and processing responses asynchronously.

```
groupChat.AddChatMessage(new ChatMessageContent
(AuthorRole.User, <User Input Message>));
await foreach (ChatMessageContent response in
groupChat. InvokeAsync())
{
    Console.WriteLine($"{response. AuthorName.
    ToUpperInvariant()}:\n{response. Content}");
}
```

This setup allows us to embed our AI agents in support chatbots, automated incident management systems, or internal analysis pipelines. For a complete, interactive implementation, please refer to our GitHub repository[2].

Conclusion

AI agents are more than just a technological innovation – they transform the way we approach automation and decision-making. With Semantic Kernel, we can build intelligent systems that autonomously search for solutions to complex problems. As development in this area continues to progress rapidly, new models, enhanced orchestration mechanisms, and optimized agent interactions will further expand the realm of possibilities.

To explore the full implementation – including an interactive loop that allows you to chat directly with your AI agents – please visit our GitHub repository[2]. This is an excellent starting point for experimenting, extending, and building our own intelligent agent systems.

---

[2] https://github.com/totosan/tutorial-aiagents

Securely deploying to Azure using Octopus Deploy with Entra Workload Identity

In this article, we'll explore how Octopus Deploy (I'll use Octopus from now on) can be used to deploy applications to Azure securely. Before diving into the technical details, we should define what 'secure' means in this context.

Author Michael van Rooijen

With 'secure', we mean that we want to:

- Apply the least privilege principle. We don't want to provide an account/credential with more permissions than necessary.
- Protect Octopus properly so that accounts cannot be compromised easily.
- Use secrets with a short validity period. The shorter, the better. The longer a secret is valid, the more time someone with ill will has to discover the secret, and the longer a compromised secret can be abused. Or, even better:
- Don't use any secrets. By not using secrets, we remove a potential attack vector because there are no more secrets that could inadvertently be leaked or stolen.

Octopus concepts and their relation to Azure

Because we'll use some Octopus-specific terminology in the rest of this article, we should define what these mean:

Environments in Octopus represent the deployment pipeline's different stages, such as development, testing, and production. These environments can be mapped to, for example, Azure Resource Groups or Azure Subscriptions, allowing for organised and isolated deployments across different stages.

Projects in Octopus encapsulate an application's deployment process, defining the steps and configurations required for a release.

Variables play an important role in Octopus by allowing the definition and management of configuration settings across deployment processes. They can also store sensitive information like API keys, ensuring these values are securely managed and only accessible when needed. Variables can be scoped to different levels, such as environments, deployment targets, or tenants, providing flexibility in managing configurations. They are used in the various steps of the deployment process to provide values necessary at application deploy time or runtime or to decide which steps to run in the process.

Octopus authentication

Another important area in Octopus, certainly when discussing security, is authentication. Authentication is the process of verifying the identity of a user, device, or system before granting access to resources or performing actions. Authentication typically involves providing credentials or proof of identity, which the system verifies against its records. In Octopus, authentication can be broadly grouped into Users and Accounts. In my head, I also categorise them as 'Internal Accounts' and 'External Accounts', reflecting the account's purpose.

- **Users / Internal Accounts:** Users perform actions in Octopus. They are defined at the Octopus system level as user or service accounts. User accounts can use the Octopus Web Portal and the Octopus API and authenticate with a username and password, Active Directory credentials, or an Octopus API key. Service accounts are API-only accounts that should be used for automated services that integrate with Octopus and can only authenticate with an Octopus API key.
- **(External) Accounts:** Accounts are used for Octopus to authenticate itself during deployments. Octopus supports a variety of account and credential types. Examples of account types are 'cloud' accounts (Azure, AWS and Google Cloud), Username/Password and SSH key. An Azure Account (it's actually called an 'Azure Subscription' in Octopus) can be configured with three types of credentials: 1. Service Principal with a client secret 2. OpenID Connect 3. Management Certificate. For this article, we will only talk about OpenID Connect.

OpenID Connect

In Octopus, using OpenID Connect with Azure means we'll use Entra ID Federated Identity Credentials. Federated identity credentials are a relatively new type of credential that enables workload identity federation for software workloads. Workload identity federation allows you to access Microsoft Entra-protected resources without managing secrets.

A trust relationship is created between an external identity provider (IdP, in this case, Octopus) and an app in Microsoft Entra ID by configuring a federated identity credential. The federated identity credential indicates which token from the external IdP your application can trust. After creating a trust relationship, Octopus can exchange trusted ID tokens for access tokens from Microsoft Entra ID. Octopus then uses that access token to access the Microsoft Entra-protected resources to which the Entra app has access. This process eliminates the maintenance burden of manually managing credentials and eliminates the risk of leaking secrets or having certificates expire. The underlying tokens exchanged between both systems are short-lived, which means that even if a token is compromised, the window of opportunity for an attacker is limited.

Configuring an Azure account with OpenID Connect credentials

To configure an Azure account with OpenID Connect, the following properties will have to be provided:

- **Client/application ID:** This is the unique identifier generated for the application when it was created in Entra ID. Entra ID uses it to identify the application during the authentication process.
- **Tenant ID:** The Tenant ID is the unique identifier of the Azure tenant the application is part of.
- **Subscription ID:** An Azure account must be tied to an Azure subscription in Octopus, identified by its unique identifier, even though this is not strictly necessary from the Azure perspective.
- **Audience claim:** This is put into the aud claim of the ID token sent to Entra ID. Microsoft and Octopus recommend using the value 'api://AzureADTokenExchange'.
- **Subject claim:** This is put into the sub claim of the ID token sent to Entra ID. The value can be configured to be a combination of the name of the account, environment or project (and a few other options). The value can differ based on where/when the account is used in deployments/runbooks or health checks.

Examples of subjects:

- **From a deployment**
  - Space (name=default) & account (name=azureaccount) => `space:default:account:azureaccount`
  - Environment (name=uat) & project (name=website) & type => `project:website:environment:uat:type:deployment`
- **From account test** (this is used when testing out the account, through the website or API)
  - Space (name=default) & account (name=azureaccount) => `space:default:account:azureaccount:type:test`

Programmatically creating the account

An account in Octopus can be created via the Octopus Web Portal or programmatically using the Octopus API. In our examples, we'll use the Octopus API to create an Azure account with OpenID Connect credentials, calling the API from PowerShell.

The following JSON payload will need to be sent to the API. Most of the properties have been described above. In addition to those, a name and the subject keys to be used for deployments/runbooks, health checks, and the account test need to be provided.

```
$octopusApiPayload = @{
    Name                      = 'NameOfTheAccountInOctopus'
    AccountType               = 'AzureOidc'
    SubscriptionNumber        = '1f111d11-11d1-1d11-bcc1-1da111111111'
    ClientId                  = '2f222d22-22d2-2d22-bcc2-2da222222222'
    TenantId                  = '3f333d33-33d3-3d33-bcc3-3da333333333'
    Audience                  = 'api://AzureADTokenExchange'
    DeploymentSubjectKeys     = @('space', 'account')
    HealthCheckSubjectKeys    = @('space', 'account')
    AccountTestSubjectKeys    = @('space', 'account')
} | ConvertTo-Json
```

To call the Octopus API, you'll need to provide the API key and the Content-Type as HTTP request headers. The API key can be for a User Account or a Service Account. This account needs to have the permissions to create, update, and delete accounts. The 'Environment Manager' User Role in Octopus covers these permissions.

```
$headers = @{
    'X-Octopus-ApiKey'  = 'API-1234567890abcdefghijklmn'
    'Content-Type'      = 'application/json; charset=utf-8'
}
```

To create a new account, an HTTP POST request needs to be sent to the Octopus API `/accounts` endpoint:

```
Invoke-RestMethod -Uri 'https://youroctopusfqdn/
apiaccounts' -Method Post -Headers $headers -Body
$octopusApiPayload
```

To update an existing account the same payload can be sent, but using an HTTP PUT request instead of a POST. You'll also need to know the accountId of the account that you want to update. I use the `/accounts/all` endpoint for this purpose. This returns all accounts in the Octopus instance, which I then filter on the name.

```
# retrieve existing accounts
$existingAccounts = Invoke-RestMethod -Uri
'https://youroctopusfqdn/api/accounts/all' -Method
Get -Headers $headers
# get the correct account by filtering on account
name, which is unique
$existingAccount = $existingAccounts | Where-Object
{ $_.Name -eq 'NameOfTheAccountInOctopus' }

# update the existing account
Invoke-RestMethod -Uri "https://youroctopusfqdn/api/
accounts/$($existingAccount.Id)" -Method Put -Headers
$headers -Body $octopusApiPayload
```

That was the first of the two parts to setting up the trust relationship between Octopus and Entra ID. Next, we'll configure Entra ID side.

Creating federated credentials in Entra ID

In Entra ID, federated credentials can be created for two types of principals: Service Principals and User-Assigned Managed Identities (I'll call it a UMID from now on). A UMID is a special type of Service Principal because its lifetime is tied to a resource in the Azure Control Plane, whereas a Service Principal only lives in Entra ID. In the context I've been using Octopus with Entra federated credentials, I've chosen to use Service Principals to create the Azure Infrastructure needed for my application and the UMIDs I've chosen to use.
The UMIDs I then primarily use for data plane operations in Azure, for example, a database deployment using SQL package.

Next, I'll show how to create federated credentials for both types of principals because there are slight differences in how you create them.

Using a Service Principal with federated credentials

To create federated credentials for a Service Principal, a few properties will need to be provided:

- **Issuer:** As mentioned in the Octopus docs, your Octopus instance will need to have two URLs anonymously exposed to the Internet: `https://server-host/.well-known/openid-configuration` and `https://server-host/.well-known/jwks`, where '`server-host`' is the Internet-resolvable FQDN of your Octopus instance. Entra uses these endpoints to look up the relevant OpenID Connect information of Octopus. The value for '`issuer`' should be the Internet-resolvable FQDN of your Octopus instance.
- **Subject:** This should be set to the same 'subject' value you've configured your Octopus account with.
- **Audiences:** As mentioned in the previous section, the value for this should be `api://AzureADTokenExchange`
- **Name:** Name of your choosing, identifying the credential.
- **Description:** Description of the credential.

We'll use the Azure CLI. The relevant command requires sending a JSON payload to create the Federated Credential. We'll first define the payload, again using PowerShell:

```
$parameters = @{
    name        = 'NameOfTheFederatedCredential'
    issuer      = 'NameOfTheFederatedCredential'
    subject     = 'project:website:environment:uat:
                  type:deployment'
    description = ''
    audiences   = @('api://AzureADTokenExchange')
} | ConvertTo-Json
```

Because I was having issues with correctly quoting the JSON string on the command line, I passed in the JSON payload from a temporary file.

```
$tempFile = [System.IO.Path]::GetTempFileName()
$parameters | Out-File -FilePath $tempFile
-Encoding utf8
az ad app federated-credential create
--id '4f444d44-44d4-4d44-bcc4-4da444444444'
--parameters @$tempFile -o none
```

As you can see, you will also need to pass in the Object ID of the Service Principal for which you are creating the Federated Credential.

Using a UMID with federated credentials

As you would expect, to create a Federated Credential for a UMID, you must also provide the `issuer`, `subject` and `audiences`. Additionally, you'll need to provide the UMID you are creating the credential for, which is identified using `identity-name`, `subscription` and `resource-group`.

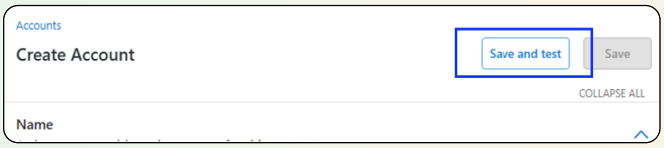
```
az identity federated-credential create --identity-
name 'NameOfYourUMID' --resource-group 'Resource-
GroupName' -n 'NameOfTheFederatedCredential'
--issuer 'NameOfTheFederatedCredential' --subject
'project:database:environment:uat:type:deployment'
--audiences 'api://AzureADTokenExchange' --subscription
'5f555d55-55d5-5d55-bcc5-5da555555555'
```

Testing the Octopus Azure account

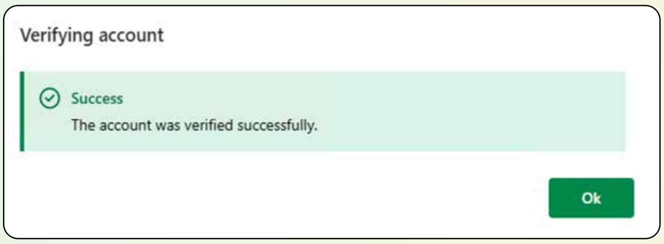
Now that we've configured Octopus and Azure, we can test if Octopus can use the federated credential properly. There are two ways of doing that: Manually through the Octopus Web Portal and through the Octopus API.

Testing in the Octopus portal

In the Octopus portal, on the account creation/update page, there's a button which has Octopus verify if the account you've configured works. The subject claim you've configured for 'account test' is used in this case. Octopus will exchange an ID token for an Azure access token and, with that access token, will try to retrieve the subscription info for the subscription ID you've configured. This means that the Azure principal you've configured will at least need read permissions on the subscription in question for this test to succeed.



If the test succeeds, Octopus will report success:


Verifying account

Success
The account was verified successfully.

Testing through the Octopus API

We can use the Octopus API to create a task. This task will then have Octopus run the account verification, after which we'll use the task's state to verify if the account test was successful. First, we'll define the JSON payload for such a task:

```
$jsonPayload = @{
    "Name"          = "TestAccount"
    "Description" = "Test Azure account"
    "Arguments"   = @{
        "AccountId" = 'Accounts-123'
    }
} | ConvertTo-Json
```

As you can see, we need to pass in the account's Octopus ID. With this JSON payload, we can create the task:

```
$task = Invoke-RestMethod -Uri
"https://youroctopusfqdn/api/tasks" -Method
Post -Headers $headers -Body $jsonPayload
```

Using the State property of the response object we can verify if the test was successful.

```
if ($task.State -eq "Success") {
    Write-Host "The Octopus Account can successfully
    connect to Azure"
}
```

Be aware that often the task state returned will be 'Queued'. You'll have to wait some time (mostly a few seconds) and retrieve the status again to see if the test was successful. This can be done as follows:

```
$taskStatus = Invoke-RestMethod -Uri
"https://youroctopusfqdn/api/tasks/$($task.Id)"
-Method Get -Headers $headers
```

Summary

In this article we've used the (relatively new) Entra ID workload identity federation to allow Octopus to authenticate against Entra ID without using a secret and retrieve an access token which can be used to execute permitted actions in Azure.

Letting AI Help Make the World More Accessible - Analyzing Website Accessibility with Semantic Kernel and OmniParser

By July 2025, digital accessibility will no longer be a goal … it's a requirement[1]. With AI and cutting-edge technologies, we can simplify the journey toward inclusivity and legal compliance.

Author **Jonathan David**

This piece explores both the *why* behind accessibility and the how of using AI to help. We'll delve into a technical proof-of-concept, discussing the architecture and specific tools like Semantic Kernel and OmniParser. Expect a moderate level of technical detail suitable for developers and AI practitioners looking into practical applications.

---

[1] https://web.archive.org/web/20240812205927/https://ec.europa.eu/social/main.jsp?catId=1202&intPageId=5581

The Importance of Accessibility

A moral imperative and a legal obligation - accessibility has become a fundamental requirement for digital products and services. According to Eurostat, 101 million or 1 in 4 adults in the European Union have a disability[2]. This number is expected to rise as the population ages. These individuals often encounter barriers that hinder equal participation in education, employment, and everyday life. At the heart of the European Accessibility Act (EAA) and Germany's Barrierefreiheitsstärkungsgesetz (BFSG) is the commitment to eliminate these barriers in the digital space. These aren't just abstract values, either, as they come with enforceable deadlines.

Why Accessibility Design is Critical Today

Statistical evidence highlights why designing accessible websites is essential and not just for compliance, but to create meaningful digital experiences and also a competitive advantage. For example:

- 70% of users with disabilities will abandon a website if it isn't accessible[3].
- More than 2.2 billion people worldwide have visual impairments, which means over 27% of the global population can benefit from using screen readers to navigate the web[4].
- Businesses could unlock a market of $13 trillion by embracing disability inclusion, with web accessibility being a crucial component of this strategy[5].
- Digital products that fully comply with WCAG 2.0 are expected to have a 50% higher market performance than their competitors[6].
- Businesses with non-compliant digital platforms risk legal challenges, fines, and reputational harm. In 2024, in the US alone, there were over 4,000 lawsuits related to non-compliance[7].

There are many more statistics[8] that underscore the importance of accessibility, but the bottom line is clear: accessibility is a strategic imperative.

The Role of AI in Accessible Development

AI offers the potential to automate, streamline, and scale the analysis of websites against international accessibility guidelines like WCAG (Web Content Accessibility Guidelines)[9]. By leveraging machine learning and natural language processing models, AI-powered tools can examine web content for typical accessibility issues such as:

- Missing or insufficient alt text for images.
- Improper semantic structure in HTML or ARIA (Accessible Rich Internet Applications) tags.
- Inaccessible user journeys, such as broken keyboard-only navigation paths.
- Form fields without proper types, labels, or instructions.

In addition to analyzing the content, AI can interpret visual elements such as layout, color contrast, and text size to identify potential issues that could impact users with visual or cognitive impairments. Using AI tools (e.g. Microsoft's OmniParser[10]) we can visually extract and analyze the elements of a website and with this assist the language models in understanding the context of the content.

Analyzing Accessibility

As part of our quarterly innovation days at Xebia, my colleagues and I came up with the idea to explore how AI could help us analyze the accessibility of a website as of today. The idea was to leverage modern tools and frameworks to identify accessibility issues and provide actionable insights. The goal was not just to detect problems, but to demonstrate how the findings from multiple agents could be combined into a comprehensive accessibility report. At the end of that day we had a working **proof of concept** that could analyze a website for accessibility issues.

Overview of the Solution

The solution we built is centered around two major components, where the analysis part takes the spotlight (we'll dive deeper into the details later):

- **Capturing the website content and visuals.** Using a headless browser, we obtain the fully rendered HTML source code and a screenshot of the website.
- **Analyzing the captured results.** This involves using Semantic Kernel's process framework to orchestrate the analysis workflow with multiple specialized AI agents.

For the Proof of Concept, we selected a set of analyzers to focus on specific WCAG criteria that demonstrate the potential of this system. The POC includes the following analyzers:

- **HTML Content Analysis:**
  - Alt text analysis.
  - Semantic code validation.
  - Keyboard navigation checks.
  - Form validation.
- **Visual Analysis:**
  - Color contrast analysis.
  - Font size and readability evaluation.

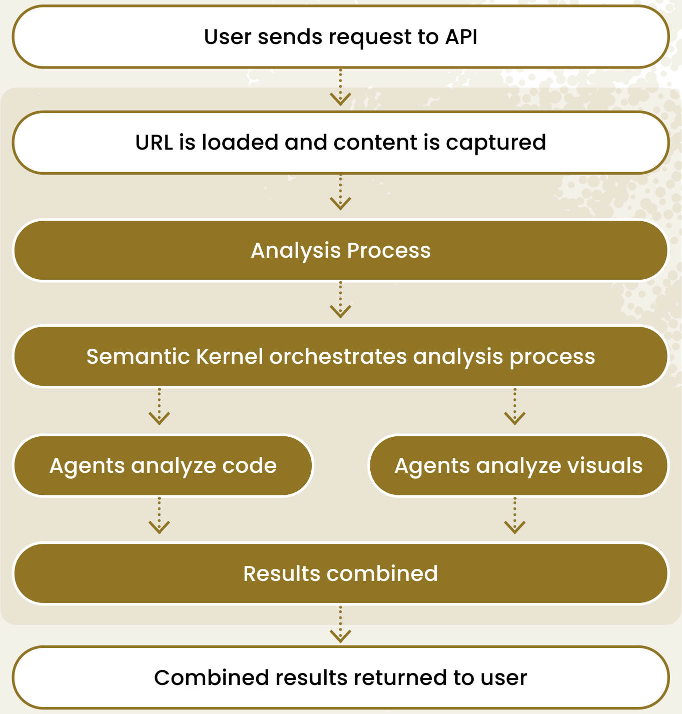
---

[2] https://www.consilium.europa.eu/en/infographics/disability-eu-facts-figures/
[3] https://reciteme.com/us/
[4] https://devblogs.microsoft.com/xamarin/the-journey-to-accessible-apps-screen-readers/
[5] https://www.forbes.com/consent/ketch/?toURL=https://www.forbes.com/councils/forbestechcouncil/2022/05/03/unlock-business-potential-by-improving-accessibility-in-the-workplace/
[6] https://www.gartner.com/en/documents/3986300
[7] https://www.accessibility.works/blog/ada-lawsuit-trends-statistics-2024-summary/
[8] https://accessiblyapp.com/blog/web-accessibility-statistics/
[9] https://www.w3.org/WAI/standards-guidelines/wcag/
[10] https://github.com/microsoft/OmniParser

Instead of building a complete testing suite to cover every WCAG criteria, we intentionally scoped the POC to include these six analyzers. This allowed us to test the concept and validate the architecture without overwhelming scope or complexity.

To house the solution, we implemented it as a Minimal API[11] using .NET 9. Minimal APIs in .NET provide a lightweight framework for quickly developing APIs with just the core essentials.

From a high-level perspective, the process looks like this:


User sends request to API
URL is loaded and content is captured
Analysis Process
Semantic Kernel orchestrates analysis process
Agents analyze code
Agents analyze visuals
Results combined
Combined results returned to user

A simple HTTP request to the URL of the website is not enough to retrieve the website's content or take an accurate screenshot. Websites today often rely heavily on client-side JavaScript for rendering dynamic elements such as buttons, menus, or even text content. To ensure the page is rendered completely as intended, we need to simulate a user environment via a headless browser.

For this Proof of Concept, we used the Playwright library[12]. Playwright loads the website in a headless browser, giving JavaScript the chance to execute and ensuring a fully rendered page. Once rendered, the HTML source is captured along with a full-page screenshot.

> We opted to use the largest breakpoint (desktop view) to capture the screenshot, as it typically provides the most comprehensive view of a website's content. In a real-world scenario, capturing at additional breakpoints - such as mobile or tablet views - would be advisable to audit the website's accessibility across multiple devices.

Handing over to Semantic Kernel

Once we have the content and the screenshot, we pass them to Semantic Kernel[13], specifically using its Process Framework[14], which efficiently manages complex workflows by organizing tasks into modular, reusable steps.

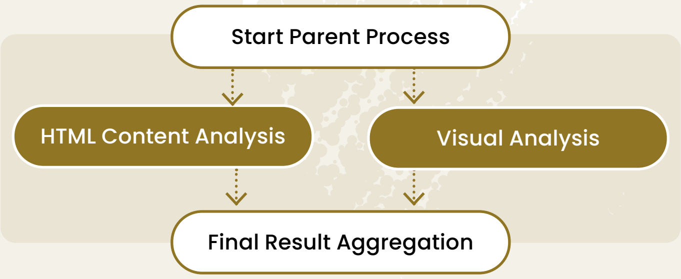
The Process Framework lets us define workflows where each step performs a specific task. This helps break our accessibility analysis into logical components - processing content, running specialized evaluations, and compiling results. Each part can be developed and improved independently, making the codebase more maintainable. It also simplifies integrating different AI capabilities to assess specific aspects of accessibility.

In our Proof of Concept, the Process Framework orchestrates our multi-step analysis. Steps are assigned to specialized AI agents focused on specific aspects of website accessibility - analyzing HTML structure, evaluating semantic elements, or checking visual aspects like text contrast and font sizes. This modular design keeps the system flexible and ready to support additional checks in the future.

We considered Semantic Kernel's Group Chat[15] capabilities but found the Process Framework more suitable for our use case. While Group Chat is geared toward real-time collaboration, the Process Framework is better for orchestrating complex, repeatable workflows.

The Process' Implementation

To ensure the workflow is organized, efficient, and scalable, we split the accessibility analysis into two distinct sub-processes: HTML Content Analysis and Visual Analysis. These sub-processes operate independently and are orchestrated by a parent process, which combines their results into a unified accessibility report.



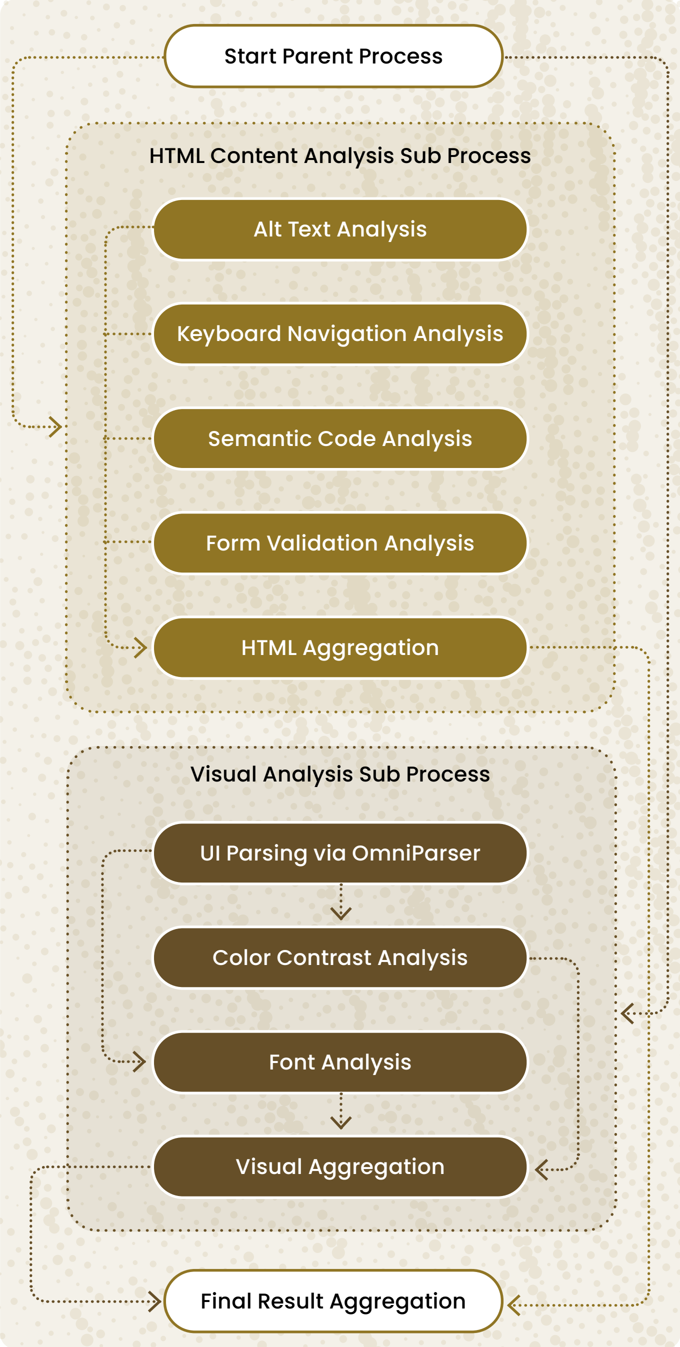
Each sub-process executes a series of specialized steps to analyze its corresponding aspect of accessibility. Tasks within a sub-process, such as **alt text detection** or **color contrast evaluation**, are executed in parallel whenever possible. This reduces the overall time required to analyze each aspect. Similarly, the HTML Content Analysis and Visual Analysis sub-processes themselves run in parallel, further optimizing performance and ensuring comprehensive results are generated quickly.

---

[11] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/overview?view=aspnetcore-9.0
[12] https://playwright.dev
[13] https://github.com/microsoft/semantic-kernel
[14] https://learn.microsoft.com/en-us/semantic-kernel/frameworks/process/process-framework
[15] https://learn.microsoft.com/en-us/semantic-kernel/frameworks/agent/agent-chat?pivots=programming-language-csharp

Within each process, aggregation steps collect and consolidate the outputs of the individual analysis tasks. For example, the HTML Content Analysis sub-process gathers results from tasks like **keyboard navigation checks** and **form validation**, while the Visual Analysis sub-process processes results from **font size** and **color contrast analyses**. These aggregated results are then passed back to the parent process, which merges them into a final accessibility report.

This modular and parallelized design ensures flexibility, scalability, and speed. By dividing the workflow into clearly defined processes with parallel execution, we are able to reduce processing time, reuse steps, and expand or adjust the analysis framework without disrupting the overall system.



Managing Parallel Execution

To manage parallel execution effectively, each process includes an **aggregation step** that consolidates the results of all the previous steps. These aggregation steps are essential for synchronizing asynchronous workflows, ensuring that no task is left incomplete before moving on to the next phase.

Each step in the process emits an event when it completes. These events are routed to specific event handlers in the aggregation step. The aggregation step maintains a state object with flags (booleans) to track the completion of each step. When an event updates the state, the handler checks whether all flags are set to true. Once all tasks in the sub-process are complete, the aggregation step emits its own event to signal the parent process or trigger the next stage.

By using this design, tasks can run in parallel without blocking, and the pipeline remains organized while ensuring that each step's results are accounted for.

HTML Content Analysis

The steps in the HTML Content Analysis sub-process focus on analyzing the HTML source of the website. Each analysis step is implemented independently, orchestrated by Semantic Kernel, and focuses on a particular aspect of accessibility. While the specific goals of each step vary, the structure of the steps is largely identical. Each step uses a specific system prompt tailored to its task and leverages state management and event-driven communication to collaborate with the rest of the analysis process.

At their core, the analyzer steps:

1. Start with a system prompt for the Large Language Model (LLM), providing **detailed instructions** about the **evaluation criteria** and the **expected result format**.
2. Manage step-specific state (e.g., chat history) to ensure context is preserved across interactions.
3. In a **Kernel Function**, process incoming data (e.g., HTML content from the webpage) and send it using Semantic Kernel's ChatCompletionService[16].
4. Send results "back" using an `EmitEventAsync` call. An event is emitted that the process knows how to handle.

A prompt example for the Alt Text Analysis step might look like this:

```
You are an accessibility expert. Your role is to
analyze HTML code.
Analyze the HTML_CODE for WCAG 2.1 Level AA
accessibility issues, only focusing on:

Alt Text for Images:
1. Meaningful Images Without Alt Text
  - Identify images that convey meaningful content
    but do not have alt text
```

---

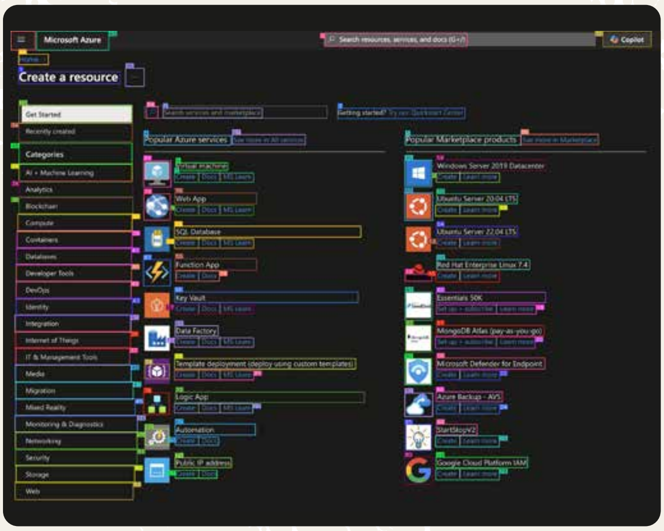
[16] https://learn.microsoft.com/en-us/semantic-kernel/concepts/ai-services/chat-completion/?tabs=csharp-AzureOpenAI%2Cpython-AzureOpenAI%2Cjava-AzureOpenAI&pivots=programming-language-csharp

Xebia Microsoft Services
→ AI

```
  - Ensure that all images with a functional or
    informational purpose have descriptive alt text
  - Check that alt text is concise, clear, and
    provides context

2. Decorative Images with Alt Text
  - Check that purely decorative images (such as
    those styled with CSS or providing no information)
    do not have alt text or have an empty alt=""
    attribute
  - Make sure that images used for visual styling
    (e.g., icons, borders) do not have misleading
    alt text
```

Visual Analysis using OmniParser v2

The Visual Analysis sub-process operates similarly to the HTML Content Analysis sub-process, using independent steps for specific tasks such as **color contrast analysis** and **font size evaluation**. However, it also includes an additional crucial step: **UI Parsing**, which uses OmniParser v2[17] to extract structured visual and positional data from the screenshot generated via Playwright.



Example of OmniParser v2 results showing the parsed elements of the Azure portal website.

Role of OmniParser

OmniParser, as described in the Microsoft Research blog[18], is a trained AI model. More specifically, it is a deep learning-based computer vision model designed to understand and parse visual elements of graphical user interfaces (GUIs). OmniParser is fundamentally a standalone machine learning model that employs computer vision to interpret visual structures and extract meaningful content.

By bridging visual elements with semantic context, OmniParser transforms previously unstructured interface data into structured, machine-readable formats, enabling deeper insight and analysis.

In the context of our accessibility solution, OmniParser is used in the **UI Parsing** step of the Visual Analysis workflow. OmniParser processes the screenshot captured via Playwright and returns structured JSON data. This output provides:

- **Key:** A unique identifier for each element, matches the number on the parsed screenshot.
- **Type:** The category of the element (e.g., "text," "image").
- **Bounding Box:** Visual coordinates of the element on the page ([x1, y1, x2, y2]).
- **Interactivity:** Whether the element is actionable (e.g., a clickable button or link).
- **Content:** The visible text or description associated with the element (e.g., button labels or menu text).

For example, the OmniParser-generated JSON might look like this:

```
[
  {
    "icon 0": {
      "type": "text",
      "bbox": [0.45530304312705994,0.004352557007223368,
      0.6613636612892151,0.031556040048599243],
      "interactivity": false,
      "content": "O Search resources, services, and
      docs (G+/)"
    }
  },
  {
    "icon 1": {
      "type": "text",
      "bbox": [0.011363636702299118,0.08487486094236374,
      0.15833333134651184,0.10990206897258759],
      "interactivity": false,
      "content": "Create a resource"
    }
  },
]
```

Structured Data and LLM

The structured data generated by OmniParser, along with the screenshot captured by Playwright, forms the foundation for the visual analysis. By including the structured data in the LLM's prompt alongside the screenshot, we aim to give the LLM additional contextual information about the website's visual elements, such as their content, placement, and interactivity.

While we hypothesize that this dual-input approach enhances the LLM's ability to evaluate accessibility issues, this has not yet been conclusively tested. In theory, the structured data could provide explicit spatial relationships and content details, helping the LLM analyze visual elements more comprehensively. Combined with the screenshot, these inputs might allow the LLM to cross-reference visual and semantic contexts to identify potential issues with layout, contrast, or interactivity.

---

[17] https://www.microsoft.com/en-us/research/articles/omniparser-v2-turning-any-llm-into-a-computer-use-agent/

[16] https://www.microsoft.com/en-us/research/articles/omniparser-for-pure-vision-based-gui-agent/

Working with the Results

To ensure that the output from each analysis step is structured and easy to process, we include a system message in every prompt that defines the expected output format. This approach standardizes the results from each analyzer, making it easy to deserialize the data.

Once all analysis steps and their sub-processes are complete, the parent process aggregates the results from both the HTML Content Analysis and Visual Analysis sub-processes. The final output is a structured JSON object that includes:

```
{
  "analysis": [
    {
      "id": "2.1.1", // WCAG success criterion
      reference, e.g., "1.1.1"
      "error": "Code.Keyboard.Focusable", // unique
      error code, 'Code' if html was analyzed,
      'Visual' if screenshot was analyzed
      "category": "Keyboard Accessibility", // category
       of the agent performing the analysis
      "description": "Element is not keyboard-
      focusable.", // Very brief description of the
      issue, acts as the title
      "detail": "All interactive elements must be
      accessible via keyboard navigation.",
      // Description of the issue
      "location": "Line 10, Column 20: <span id=\
      "foo\">Bar</span>" // Location of the issue
      in the code, or a description where in the
      screenshot if a screenshot is provided
    }
  ]
}
```

To help illustrate how the results appear in practice, here's an example of the JSON output generated by the analysis process:

```
{
  "analysis": [
    {
      "id": "1.1.1",
      "error": "Code.Image.MissingAlt",
      "category": "Alt Text for Images",
      "description": "Meaningful image lacks alt
      text.",
      "detail": "The image at <img src=\"og-image.png\">
      is meaningful but does not have alt text.
      Add descriptive alt text.",
      "location": "Element in meta tag with attribute:
      property='og:image'",
      "information": []
    },
    {
      "id": "2.4.3",
      "error": "Code.Keyboard.TabOrder",
      "category": "Keyboard Accessibility",
      "description": "Incorrect Tab Order value used.",
      "detail": "`tabindex=\"-1\"` on non-modal
      elements disrupts logical navigation as it
      removes them from the tab order.",
      "location": "Line 10: <span tabindex=\"-1\">
      </span>",
      "information": []
    },
    {
      "id": "3.1.2",
      "error": "Forms.Missing.Label",
      "category": "Form Accessibility",
      "description": "Missing form labels.",
      "detail": "The 'Search' button lacks a proper
      label or an aria-label to describe its purpose.",
      "location": "Line containing `<button aria-label=
      'Search'>`.",
      "information": []
    },
    {
      "id": "1.3.1",
      "error": "Forms.Placeholder.Misuse",
      "category": "Form Accessibility",
      "description": "Placeholder as label.",
      "detail": "The 'Search' button uses the place-
      holder 'Search' as the only means of labeling,
      which is an incorrect accessibility practice.",
      "location": "Line containing `<span class=
      'DocSearch-Button-Placeholder'>Search</span>`.",
      "information": []
    },
         ...
  ],
  "summary": {
    "failed": 12
  }
}
```

The list of analysis results is returned to the API function called by the user and sent back as a structured JSON response. Our Proof of Concept ends here, and we did not implement a UI to visualize the results. However, the JSON output can be easily used in a front-end framework or reporting tool to display the findings.

Conclusion

This POC demonstrated the potential of combining Semantic Kernel, OpenAI models, and tools like OmniParser to automate website accessibility analysis. The solution achieved solid results as a foundational starting point, particularly in analyzing HTML structure and visual elements for WCAG compliance.

We relied on Semantic Kernel to handle the orchestration of the analysis workflow. Its process framework simplified the implementation of the multi-step process by enabling state management, event handling, and coordination between agents. This modular design made it easy to develop and expand the system without introducing unnecessary complexity.

The OpenAI models performed well in certain areas, such as diagnosing alt text issues and semantic code validation, but they were not without limitations. Token limits presented challenges when sending larger sections of the HTML source to multiple agents, requiring careful adjustments to avoid truncating important content. To control costs, we transitioned from GPT-4 to GPT-3.5 during development. Both GPT-4 and GPT-3.5 occasionally produced hallucinations, such as generating non-existent HTML code. These issues show the need for additional layers of validation when using language models for tasks that require high accuracy.

There were also some unexpected challenges with integrating OmniParser. The instance hosted on Hugging Face[19] returned errors during testing, which required us to self-host OmniParser to proceed.

Despite these challenges, the results showed that AI can - at this stage - uncover a range of accessibility issues and provide insights that are actionable for developers. The structured JSON outputs from our system lay the groundwork for integration into frontend tools or reporting platforms.

What Could Be Improved

While the Demo showed promising results, there are several areas that could be refined for future iterations, including:

Specialized Models for Accessibility Analysis

Currently, **general-purpose** language models like GPT-3.5 and GPT-4 are being used for analysis. While these models performed well, developing or fine-tuning specialized models focused specifically on accessibility issues could improve accuracy and reduce the chances of hallucination. A model trained on accessibility-specific datasets would also be better equipped to handle nuances in WCAG compliance.

Optimized Token Usage

Token limits were one of the key challenges we encountered when processing larger sections of source code in parallel. This limitation required careful adjustments to ensure no critical information was truncated before analysis. One way to address this issue might be to minify the HTML code prior to sending it to the models, reducing token usage. However, further testing would be needed to verify that this doesn't degrade detection accuracy, especially for scenarios where formatting or element relationships play an important role.

Another solution is to explore Large Language Models with larger context windows. For example, newer models like Gemini 2.0 Pro[20] offer a context window of up to 2 million tokens, far exceeding the limits of GPT-3.5 or GPT-4.

In addition, Azure OpenAI Service provides flexibility in managing token usage. Token rate limits can be adjusted via Azure AI Foundry for the selected LLM deployment, allowing developers to fine-tune the settings to better suit specific workloads. By tweaking these settings, it's possible to mitigate some of the practical limits imposed during input processing and optimize workflows for more complex, large-scale analysis.

Discovering Playwright MCP

While writing this article, I came across Microsoft's Playwright MCP repository[21], which provides LLMs with the ability to analyze and interact with web pages using Playwright.

MCP (Model Context Protocol) acts as a standardized way for AI applications to connect with data sources and tools. Anthropic describes MCP as a "USB-C for AI applications"[22], simplifying how LLMs exchange context with external systems. *This would be a whole article on its own.*

Playwright MCP allows LLMs to navigate web pages, fill out forms, click buttons, and extract data. A key feature is its ability to use the accessibility tree of a webpage, which represents UI elements as seen by assistive technologies. This lets the tool analyze UI interfaces without requiring the LLM to process raw HTML directly. Playwright MCP also supports screenshot-based analysis through a visual mode, making it a flexible option for different types of workflows.

Given its ability to interact with the accessibility tree, Playwright MCP could complement OmniParser as part of future iterations. By providing a direct, standards-compliant mechanism to evaluate UI elements, it reduces reliance on standalone visual parsing tools or reliance on HTML source code, potentially simplifying the architecture while improving results. *Definitely worth checking out!*

Final Thoughts

With the July 2025 deadline for compliance with accessibility standards in the EU fast approaching, the need to prioritize digital accessibility has never been greater. AI-driven tools, such as the ones explored in this POC, can play a growing role in supporting developers and accessibility experts by identifying potential issues and providing actionable insights. While AI is not a replacement for static analysis tools or human expertise, it can complement existing efforts by detecting areas that might otherwise be overlooked and speeding up the auditing process. Despite its current limitations, the progress achieved here demonstrates how AI can meaningfully contribute to accessibility analysis today - and its role is only set to grow as we refine these tools further.

---

[19] https://huggingface.co/microsoft/OmniParser-v2.0

[20] https://explodingtopics.com/blog/list-of-llms#context-windows-and-knowledge-boundaries

[21] https://github.com/microsoft/playwright-mcp

[22] https://docs.anthropic.com/en/docs/agents-and-tools/mcp

.NET Aspire: A Game-Changer for Cloud-Native Development

Nowdays developers are increasingly tasked with building distributed applications that are scalable, resilient, and cloud-native. However, orchestrating multiple services, managing configurations can be daunting and time-consuming.

Author Emanuele Bartolesi

Designed with developers in mind, .NET Aspire simplifies the process of building, deploying, and managing distributed applications by providing a cohesive set of tools, templates, and integrations.

Whether you're developing microservices, integrating with various cloud services, or aiming for a more efficient development workflow, .NET Aspire offers a unified approach to tackle these challenges head-on.

What is .NET Aspire?

.NET Aspire is an opinionated, cloud-ready application stack introduced by Microsoft to streamline the development of cloud-native applications. It provides developers with a cohesive set of tools, templates, and integrations designed to simplify the complexities associated with building distributed systems.

At its core, .NET Aspire aims to enhance the developer experience by offering:

- **Dev-Time Orchestration:** Facilitates the running and connecting of multi-project applications and their dependencies in local development environments.
- **Integrations:** Provides NuGet packages for commonly used services, such as Redis or PostgreSQL, ensuring consistent and seamless connections within your application.
- **Tooling:** Includes project templates and tooling experiences for Visual Studio, Visual Studio Code, and the .NET CLI to assist in creating and interacting with .NET Aspire projects.

By addressing common challenges in modern app development—like managing multiple services, handling configurations, and ensuring observability—.NET Aspire empowers developers to focus more on writing code and less on infrastructure concerns.

Real-World Experience: Transitioning from Docker Compose to .NET Aspire

A lot of people starts to transisioning from using Docker Compose to .NET Aspire for orchestrating a multi-service application. Initially, Docker Compose managed services like PostgreSQL and RabbitMQ. However, integrating .NET Aspire's AppHost project streamlined the orchestration process.

With AppHost, service dependencies and configurations were defined directly in code, eliminating the need for separate YAML files. This approach not only reduced complexity but also enhanced the development experience by providing a unified view of the application's architecture.

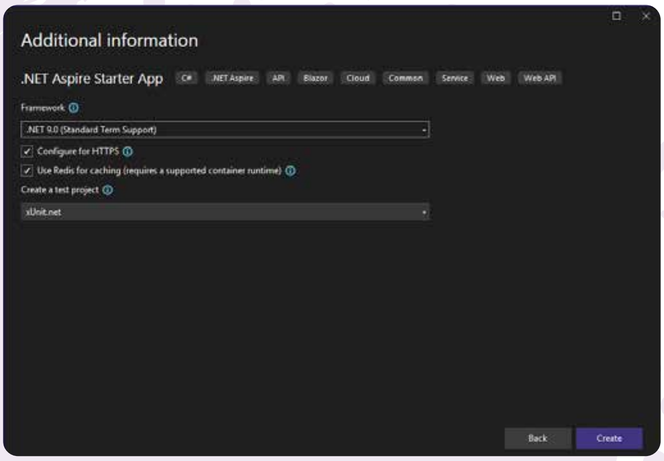
Moreover, .NET Aspire's built-in observability features, such as OpenTelemetry integration, offered deeper insights into application performance, facilitating quicker debugging and optimization.

Step-by-Step: Getting Started with .NET Aspire

Creating a New .NET Aspire Project You can initiate a .NET Aspire project using Visual Studio, Visual Studio Code, JetBrains Rider or the .NET CLI. In this article, we will focus on using Visual Studio to create a new .NET Aspire project.

Using Visual Studio

- Navigate to File > New > Project.
- In the dialog window, search for "Aspire" and select the **.NET Aspire Starter App** template.
- On the "Configure your new project" screen:
  - Enter a project name, such as AspireSample.
  - Specify the location for your project.
- On the "Additional Information" screen:
  - Ensure that .NET 9.0 is selected as the target framework.
  - Check Use Redis for caching if you want to include Redis integration.
  - Optionally, select Create a tests project to include a test project in your solution.
- Click Create to generate the solution.



Additional information

.NET Aspire Starter App

Framework: .NET 9.0 (Standard Term Support)

☑ Configure for HTTPS

☑ Use Redis for caching (requires a supported container runtime)

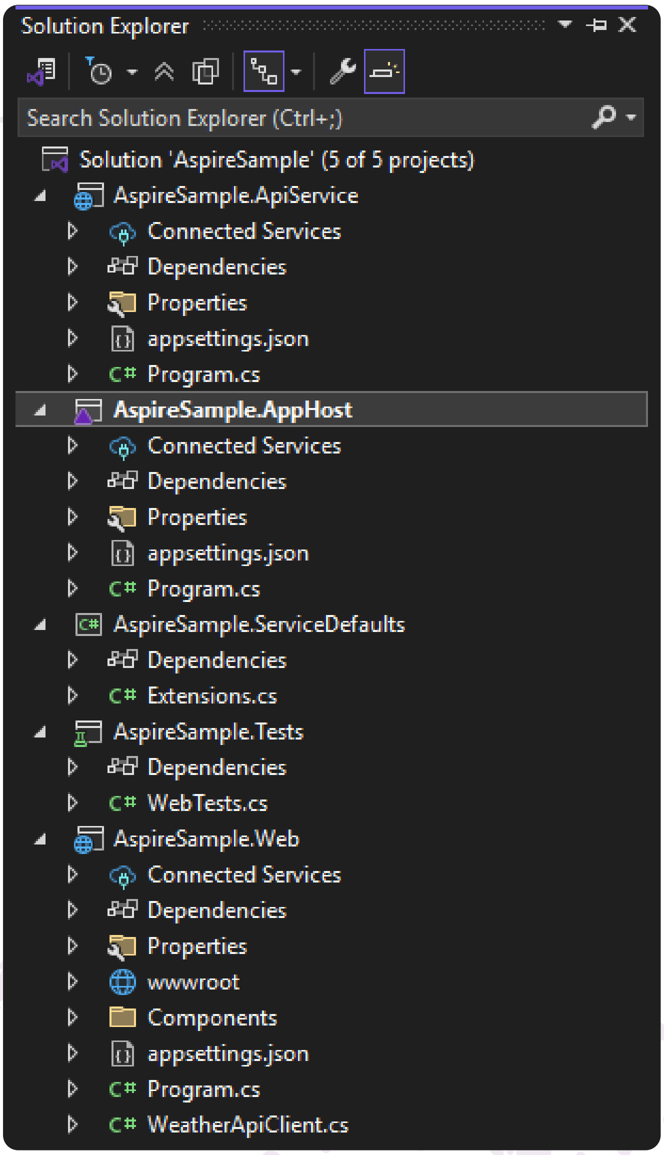
Create a test project: xUnit.net

Visual Studio will create a solution structured for .NET Aspire, including projects like AppHost, ServiceDefaults, and sample services.

Exploring the Solution Structure

The generated solution will include the following projects:

- **AppHost:** The main entry point for your application, responsible for orchestrating services and managing configurations.
- **ServiceDefaults:** Contains default configurations and settings for your services.
- **ApiService:** A sample API service demonstrating how to implement a service using .NET Aspire.
- **Tests:** A test project for writing unit and integration tests for your services.
- **WebApp:** A sample web application that interacts with the API service.



```
Solution 'AspireSample' (5 of 5 projects)
  AspireSample.ApiService
    Connected Services
    Dependencies
    Properties
    appsettings.json
    Program.cs
  AspireSample.AppHost
    Connected Services
    Dependencies
    Properties
    appsettings.json
    Program.cs
  AspireSample.ServiceDefaults
    Dependencies
    Extensions.cs
  AspireSample.Tests
    Dependencies
    WebTests.cs
  AspireSample.Web
    Connected Services
    Dependencies
    Properties
    wwwroot
    Components
    appsettings.json
    Program.cs
    WeatherApiClient.cs
```

The AppHost project is where you define your application's architecture, including service dependencies and configurations. The ServiceDefaults project provides default settings that can be overridden in individual services. The ApiService and WebApp projects serve as examples of how to implement and consume services within the .NET Aspire framework.

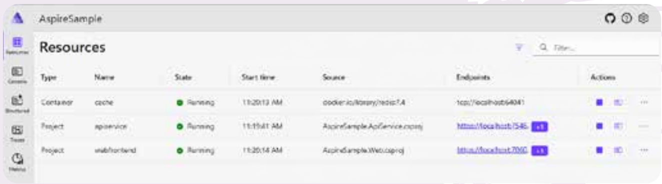
```
var builder = DistributedApplication.CreateBuilder
(args);

var cache = builder.AddRedis("cache");

var apiService = builder.AddProject<Projects.
AspireSample_ApiService>("apiservice");

builder.AddProject<Projects.AspireSample_Web>
("webfrontend")
     .WithExternalHttpEndpoints()
     .WithReference(cache)
     .WaitFor(cache)
     .WithReference(apiService)
     .WaitFor(apiService);

builder.Build().Run();
```

MAURITIUS
OPILOT
AI POWERED

As you can see, the code above is a simple example of how to define service dependencies and configurations in the AppHost project. The `AddRedis` method adds Redis as a caching service, while `AddProject` registers the ApiService and WebApp projects. The `WithReference` and `WaitFor` methods establish dependencies between services, ensuring that they are started in the correct order.

Running the Application

To run the application, simply press F5 in Visual Studio or use the .NET CLI to execute the project. The AppHost project will start all registered services, and you can access the web application through your browser.



Type	Name	State	Start time	Source	Endpoints	Actions
Container	cache	Running	11:20:13 AM	docker.io/library/redis:7.4	tcp://localhost:64041	■ ···
Project	apiservice	Running	11:19:41 AM	AspireSample.ApiService.csproj	https://localhost:7546 +1	■ ···
Project	webfrontend	Running	11:20:14 AM	AspireSample.Web.csproj	https://localhost:7060 +1	■ ···

Benefits of Using .NET Aspire

1. **Simplified Orchestration:** .NET Aspire eliminates the need for complex YAML configurations by allowing developers to define service dependencies and configurations directly in code. This approach reduces complexity and enhances the developer experience.
2. **Built-in Observability:** With integrated observability features, .NET Aspire provides insights into application performance, making it easier to identify and resolve issues. This is particularly beneficial for distributed applications where monitoring multiple services can be challenging.
3. **Unified Development Experience:** By providing a cohesive set of tools, templates, and integrations, .NET Aspire streamlines the development process, allowing developers to focus on writing code rather than managing infrastructure.
4. **Community-Driven:** As an open-source project, .NET Aspire benefits from contributions from a vibrant community of developers. This collaborative approach fosters innovation and ensures that the framework remains relevant in the ever-evolving landscape of cloud-native development.
5. **Future-Proofing:** .NET Aspire is designed to adapt to new trends and technologies, ensuring that developers can build applications that are future-proof and ready for the challenges of tomorrow.
6. **Cross-Platform Support:** Built on the .NET platform, .NET Aspire can run on various operating systems, including Windows, Linux, and macOS. This cross-platform support allows developers to choose the environment that best suits their needs.
7. **Rapid Development:** With its extensive libraries and tools, .NET Aspire accelerates the development process, enabling developers to build and deploy applications faster than ever before. This rapid development cycle allows organizations to respond quickly to market demands and stay ahead of the competition.
8. **Robust Testing Framework:** .NET Aspire includes a robust testing framework that allows developers to write and execute tests easily. This ensures that applications are thoroughly tested before deployment, reducing the risk of bugs and issues in production.
9. **Comprehensive Monitoring and Logging:** The framework provides built-in monitoring and logging capabilities, allowing developers to track application performance and identify issues in real-time. This visibility is crucial for maintaining the health of cloud-native applications. ✤

Conclusion

.NET Aspire is a game-changer for cloud-native development, providing developers with the tools and resources they need to build, deploy, and manage applications in the cloud. With its focus on performance, security, and developer experience, .NET Aspire is poised to become the go-to framework for organizations looking to embrace the power of cloud computing. Whether you are a seasoned developer or just starting your journey, .NET Aspire has something to offer everyone.

Resources
- Documentation https://learn.microsoft.com/en-us/dotnet/aspire/
- YouTube Playlist https://www.youtube.com/playlist?list=PLdo4fOcmZ0oWTWWbWXqhn2w8NM3sQ_qDz

Are you sure your JWT access tokens are really secure?

When you're building a Web API, there's a high possibility that you're relying on JWT access tokens when client applications or services interact with your API in order to perform authentication and authorization. The two most common types of access tokens are opaque (or reference) tokens and JSON Web Tokens (aka JWTs).

Author **Wesley Cabus**

These two types of tokens both have their pros and cons:

Opaque token	JWT
+ Better privacy	- JWT can contain sensitive data
+ Easier to revoke	- Lifetime of a JWT is encoded within
+ Very small in size, since it's just a reference	+/- Can be kept small, but can also easily grow large when including lots of claims
- Additional server requests are needed to check the token's validity and to retrieve user data	+ Self-contained
- There's no real defining standard	+ Standardized

In this article, I'll focus on JWT tokens, since this type of token is used more frequently.

What's inside a JWT, or JSON Web Token?

A JSON Web Token consists of three parts, each part separated by a dot:

```
eyJhbGciOi...VCJ9 . eyJzdW....MDgxMTMzMX0 . GvHaTrJuFo...s15Ss
     header       .        payload        .     signature
```

The first two parts are Base64URL-encoded JSON objects, representing the token’s header information (called a JOSE header) and the data or payload of the token. The third section contains a Base64URL-encoded array of bytes, representing the signature of the header and payload combined.

> **ⓘ NOTE**
> Note that the signature part is not required! Of course, it is highly recommended to sign tokens when constructing a JWT for security purposes. Next to signed JWTs using the JWS (JSON Web Signature standard), you can also create encrypted tokens using the JWE (JSON Web Encryption) standard. JWE tokens, however, will not be further discussed in this article.

1. The JOSE Header

The header of a JWT typically includes the following three properties when creating signed tokens:

- `typ`: the media type of the token. Sometimes omitted, this value is typically set to **JWT** to indicate that this is a JSON Web Token. For access tokens, you can also see the value **at+jwt** to further specify that the JSON Web Token is an access token.

- `alg:` the cryptographic algorithm used to sign the token. Examples are **HS256** for tokens signed using a symmetric HMAC SHA-256 algorithm, **RS256** for tokens signed with an assymetric algorithm using a private/public RSA key pair and SHA-256, etc. You can find a complete list in RFC 7518.
- `kid:` the key ID of the cryptographic key used to sign the JWT. When validating a JWT signature, the kid value is used to look up the correct public key information, in case multiple JSON Web Keys (JWKs) are available to sign or validate JSON Web Tokens. For example, you can see the current list of available JWKs[1] for demo.duendesoftware.com.
  A JWT generated by demo.duendesoftware.com would have `1CE1DCBAC07A41376725E3B725E75EC4` for its kid value.

```
{
    "keys": [
        {
            "kty": "RSA",
            "use": "sig",
            "kid": "1CE1DCBAC07A41376725E3B725E75EC4",
            "e": "AQAB",
            "n": "uIeg9avR-Pxg4rVR2qC2p7gpiQHD3kt
            WRT7k4_ZxRZpd964WMYAccAibf5Y3AcdHv6PWUU
            egtkwv1hAHomhPE-KOCE8uS0wsA9z8xPwO2TSm
            7AIIUz22Jym6Lj6l-o5Tdf3jpr2j7F4Ad40hQPA
            caS28NJm0ipA4blROMSe62KosNA5EdSLCGjX6Y
            qPlEislKZc3nWzf2VisvMvvFa9hEcul6FVAYG
            lnHm9N47-DphV-g7loK93WLpfNilhZDg6CF5
            pwamRVyXpH9TVqi_ZIMBsFCFnKujX54hPv-dqN
            CkAPwNNlFQ5Tr47X4l2GdgIrbVbyJ701x0Fcu
            BFxQwnlsw",
            "alg": "RS256"
        }
    ]
}
```

If you want to read more about the header of a JWT, the best resource available is RFC 7515, section 4.

2. The payload of a JSON Web Token

The payload of a JWT is a JSON object containing claims. Some claim types are registered and frequently used, or even mandatory, but you can include as many custom claims as you wish. Just remember that every claim adds data to your token and increases the token's size. Tokens can become too large to be delivered to client applications depending on the delivery mechanism!

Additionally, claims can contain sensitive data. Access tokens typically don't include too much user-specific information to prevent leaking details about the application's users.

The most commonly used claim types are:

- `iss:` the issuer of the JWT. This claim typically points to the service that issues your tokens. In OAuth 2.0 flows, you can tpyically use this claim's value to find the discovery document at the URI `/.well-known/openid-configuration`.
- `sub:` the subject of the JWT. The value indicates the user or application who can access your API resource server.
- `aud:` the audience, or intended recipients for the JWT. The audience value points to your API, to indicate that the JWT was meant to be used to gain access to your API.
- `exp:` the expiration time. When this timestamp passes, the JWT is no longer valid for use.
- `nbf:` the not-before time. The JWT only becomes valid for use after this timestamp.
- `iat:` the issued-at time. This is the timestamp when the JWT was originally created, and can be used to determine the age of the JWT.
- `jti:` a unique identifier for the JWT, which can be used to prevent replay attacks.

Validating JSON Web Tokens

It would be pretty bad if your API would just accept any access token when that token contains the correct audience claim values, so we ensure that our API will only accept tokens which satisfy a few rules:

1. The `aud` claim contains (only) values that we expect for our API.
2. The current date/time falls between the `nbf` and `exp` values. Some leeway can be granted using a clock skew, to accommodate for time drifting between different servers.
3. The `iss`, or issuer of the token, is a known service which our API trusts.
4. The `alg`, or signature algorithm, is on our allowed list of cryptographic algorithms.
5. The signature can be successfully validated using the `kid` and the corresponding public key material of the issuer.

Luckily, there are plenty of open-source libraries out there for a variety of frameworks and programming languages, which you can use to validate JSON Web Tokens. You can find a list of JWT libraries at JWT.io[2].

But...

What if this open-source library contains a vulnerability? Or when an update breaks one of the critical validation paths? Or if some of the library's methods work every-so-slightly different when validating tokens, causing some invalid tokens to suddenly become valid?

Trust, but verify: test your JWT validation!

The easiest method to add some trust into your API, is by writing tests to validate that various JSON Web Tokens, both valid and invalid, are being properly validated and respectively accepted or rejected by your API. Every time you then update the JWT validation library, you can automatically rerun your tests to verify that the validation rules still work as expected.

---

[1] https://demo.duendesoftware.com/.well-known/openid-configuration/jwks
[2] https://jwt.io/libraries

Wondering what could go wrong with access tokens? Well, let's give some examples of tokens that may try to fool the validation logic.

Algorithm? What algorithm?

The `alg` property in the token's header, again, indicates which cryptographic algorithm is used to sign the token. But this property can also be set to `none`, to indicate that the JWT is not signed at all.

Which means that the following token is a valid JWT:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiIxMjM0I
iwibmFtZSI6Ildlc2xleSIsImlhdCI6MTczMDgxMTMzMX0.

// The JWT above equals the following:
{
  "alg": "none",
  "typ": "JWT"
}
.
{
  "sub": "1234",
  "name": "Wesley",
  "iat": 1730811331
}
.
// no signature
```

This poses a problem, exactly because this JWT is completely valid, as far as RFC 7519, section 6 is concerned. Luckily, most validation libraries will allow you to specify which signature algorithms your API allows, and most will prevent `"alg": "none"` from being used by default.

There are some people out there, however, who don't play nice and think outside the box. These people may try to authenticate against your API using a JWT where the `alg` property is set to `nONe`, or `some-random-value-here`. This could be enough to fool a poorly written JWT validation library, and yes, some libraries have accepted `"alg": "nONe"` as a valid token for a long time…

Writing an integration test to catch these invalid or unsupported signature algorithms can be very easy:

```
public class SignatureAlgorithmTests(TargetApiWeb-
ApplicationFactory factory) : JwtGuardTestBase
(factory)
{
    [Theory(DisplayName = "When a token uses an
    unsupported signature algorithm, the API should
    return a 401 Unauthorized response.")]
    [InlineData("none")]
    [InlineData("nOnE")]
    internal async Task Accessing_AuthorizedUrl_
    Is_Unauthorized_For_Unsupported_Signature_
    Algorithms(string signatureAlgorithm)
    {
        // Arrange
        var jwt = await GetJwtAsync(signature-
        Algorithm);
        Client!.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Bearer", jwt);

        // Act
        var response = await Client.GetAsync(Test-
        Settings.CurrentTestSettings.TargetUrl);

        // Assert
        Assert.Equal(HttpStatusCode.Unauthorized,
        response.StatusCode);
    }

    private Task<string> GetJwtAsync(string
    signatureAlgorithm)
    {
        // Use a JwtBuilder instance to build an
        access token
        return Factory.CreateJwtBuilder()
            .WithSignatureAlgorithm(signature-
            Algorithm)
            .BuildAsync();
    }
}

public class JwtBuilder
{
    // Most other properties and methods are omitted
    for brevity...

    public Microsoft.IdentityModel.Tokens.Signing-
    Credentials? SigningCredentials { get; private
    set; }
    public string? SignatureAlgorithm { get; private
    set; }

    public JwtBuilder WithSignatureAlgorithm(string
    signatureAlgorithm)
    {
        SignatureAlgorithm = signatureAlgorithm;
        SigningCredentials = null;

        return this;
    }

    public async Task<string> BuildAsync()
    {
        if (!string.IsNullOrEmpty(SignatureAlgorithm)
        &&
            (string.Equals(SecurityAlgorithms.None,
            SignatureAlgorithm, StringComparison.
            OrdinalIgnoreCase) ||
            !TestSettings.KnownSecurityAlgorithms.
            Contains(SignatureAlgorithm)))
        {
            // Either using "none" (case-insensitive)
            or an unknown algorithm. TestSettings.
            KnownSecurityAlgorithms
            // contains a list of known security
            algorithms.

            // Return an unsigned token.
            return BuildJwtHeader().Base64UrlEncode()
            + "." + BuildJwtPayload().Base64UrlEncode()
            + ".";
        }

        // default logic which signs and returns the
        token goes here...
    }
}
```

With a bit of modification, the same test logic can also be used to test that validly signed tokens are rejected if they're signed using an algorithm which your API doesn't want to use, like HS256 for example.

But wait, there's even more shenanigans...

When creating a JWT header, you have the option to use some additional "special" properties next to the `alg`, `kid` and `typ` properties we already discussed earlier. These special properties allow you to specify where the JWT validation logic should retrieve public key information from when validating the token's signature. For someone with malicious intent, this is absolutely wonderful, because this allows a potential attacker to create self-signed JWTs!

These are the special properties:

- `jku:` a URL pointing to a set of JSON Web Keys. While this URL could in theory point to the authority or the issuer of the token, it's not a requirement! Which means that everyone can create a JWT token, host public JWK material and add a reference URL to the token header.
- `jwk:` using this property, a token includes the public JSON Web Key in its header, to allow for full self-validation of the signature. This is very dangerous! Because this means that an attacker can craft a self-signed JSON Web Token and simply include the public JWK in the token's header!
- `x5u:` a URL pointing to a X.509 certificate or certificate chain, which can be used to validate the signature by hosting the public certificate (chain) online. Just like the `jku` property, the URL could in theory live on the same server as the authority or issuer, but this doesn't need to be the case.
- `x5c:`an X.509 certificate or certificate chain, which can be used to self-validate the signature, just like the `jwk` property. Again, very dangerous!

Testing these very specific methods to bypass our API's security is a bit more challenging, since we need to generate valid signature key material and find a way to host it (for the `jku` and `x5u` test scenarios) externally. But in pseudocode, this is how you would write the tests:

```
public class ExternalSignatureTests : Integration-
TestBase
{
    [Fact]
    public async Task RejectExternallySignedToken()
    {
        // Arrange
        var jwt = GetJwt("jwk");
        Client!.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Bearer", jwt);

        // Act
        var response = await Client.GetAsync("/secure-
        api-endpoint");

        // Assert
        Assert.Equal(HttpStatusCode.Unauthorized,
        response.StatusCode);
    }

    private string GetJwt(string testCase)
    {
        var signatureAlgorithm = "ES256"
        var jwtBuilder = Factory.CreateJwtBuilder()
            .WithSignatureAlgorithm(signature-
            Algorithm);

        var header = jwtBuilder.BuildJwtHeader();
        var payload = jwtBuilder.BuildJwtPayload();

        var encodedPayload = payload.Base64UrlEncode();

        var headerAndPayload = "";
        var signature = "";

        switch (testCase)
        {
            case "jwk":
                signature = InjectJsonWebKey(signature-
                Algorithm, header, encodedPayload, out
                headerAndPayload);
               break;

            // other test cases go here...

            default:
                return jwtBuilder.BuildAsync().
                GetAwaiter().GetResult();
        }

        return headerAndPayload + "." + signature;
    }

    private string InjectJsonWebKey(string signature-
    Algorithm, JwtHeader header, string encoded-
    Payload, out string headerAndPayload)
    {
        var securityKey = SecurityKeyBuilder.Create-
        SecurityKey(signatureAlgorithm);
        var jsonWebKey = JsonWebKeyConverter.Convert-
        FromSecurityKey(securityKey);
        jsonWebKey.Alg = signatureAlgorithm;
        jsonWebKey.Use = "sig";
        header["jwk"] = jsonWebKey.ToDictionary();
        header["kid"] = jsonWebKey.KeyId;

        return SignAndReturnJwt(header, encoded-
        Payload, signatureAlgorithm, securityKey,
        out headerAndPayload);
    }

    private string SignAndReturnJwt(JwtHeader header,
    string encodedPayload, string signatureAlgorithm,
    SecurityKey securityKey, out string headerAnd-
    Payload)
    {
        headerAndPayload = header.Base64UrlEncode() +
        "." + encodedPayload;

        var asciiBytes = Encoding.ASCII.GetBytes
        (headerAndPayload);
        var signatureProvider = CryptoProviderFactory.
        Default.CreateForSigning(securityKey, signature-
        Algorithm);
        try
        {
            var signatureBytes = signatureProvider.
            Sign(asciiBytes);
            return Base64UrlEncoder.Encode(signature-
            Bytes);
        }
        finally
        {
            CryptoProviderFactory.Default.Release-
            SignatureProvider(signatureProvider);
        }
    }
}
```

Writing these test cases can be very tedious and pose some challenges by themselves. But what if I told you that there's a solution for that?

JWT Guard to the rescue!

After attending an API security workshop, I got the idea to write some of these test cases for my own API's, but quickly found that I was repeating myself. And I don't like repeating myself. And I don't like repeating myself. So I started working on extracting the test cases in a separate project, and made these tests configurable to easily apply them to the several API projects.

The end result? JWT Guard: a test project template for .NET, allowing you to easily add a JWT test project to your existing API project by running these two simple commands:

```
# You only need to do this once per computer
dotnet new install JWTGuard.Template

# This command runs the JWT Guard template and creates
a new JWT test project:
dotnet new jwt-guard --apiProject <relative-path-to-
web-api-project>
```

After adding the new test project, all you need to do is configuring the TestSettings class in the new test project. The easiest way to configure these settings, is by overriding the defaults, for example:

```
public readonly struct TestSettings
{
    /// <summary>
    /// Static constructor for the <see cref="Test-
    Settings"/> struct.
    /// </summary>
    static TestSettings()
    {
        // Override the default test settings here
        CurrentTestSettings = DefaultTestSettings with
        {
            DefaultAudience = "my-secured-api",
            TargetUrl = "/your-secure-api-endpoint"
        };
    }

    // ... the rest remains as-is.
}
```

Right now, JWT Guard adds the following integration tests to your API project:

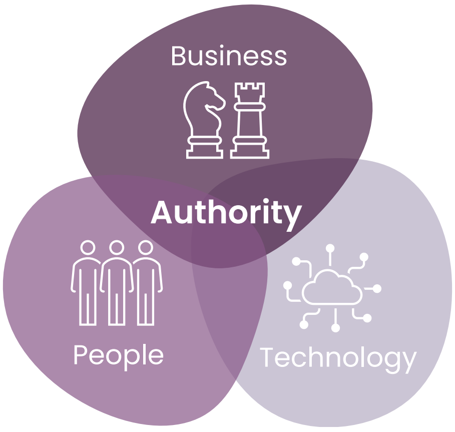
- Audience tests (`aud` claim): verifies that tokens only contain allowed audiences and are being rejected when a disallowed audience is present in the token.
- Issuer tests (`iss` claim): verifies that tokens only contain allowed issuers and are being rejected when a disallowed issuer is present in the token.
- JWT type tests (`typ` claim): verifies that tokens only use allowed values for the typ claim and are being rejected when a disallowed type is being used in the token's header.
- Signature algorithm tests (`alg` claim): verifies that tokens using disallowed signature algorithms are being rejected.
- External signature tests (`jwk`, `jku`, `x5c` and `x5u` claims): verifies that self-signed tokens or tokens signed with external key material are being rejected.

Want to know more? Then visit https://jwtguard.net for the full documentation. Happy testing!

Engagement Management – your first class ticket to value

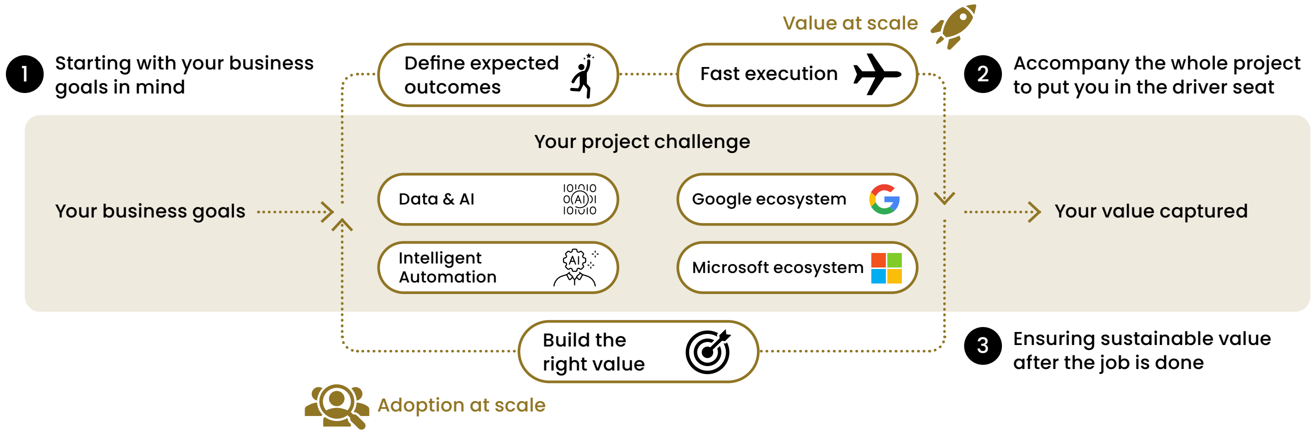
In the rapidly evolving landscape of digital transformation, companies face continuous pressure to deliver projects faster, more efficiently, and with sustainable value. At Xebia, we position ourselves as authorities like domain experts or high-class developers and specialists in solving the most challenging customer problems. Our origin and core strength lie in addressing complex technological challenges for our clients. However, customers need more than just technological expertise; they require solutions that scale effectively and are readily adoptable within their digital transitions. Therefore, Engagement Management at Xebia is not merely a methodology; it's our commitment to creating sustainable, scalable value for every customer engagement.

Author **Andreas Giesa**



Fundamentals & Definitions
Engagement Management refers to a structured approach to planning, executing, and controlling projects, fostering deep engagement among customers and internal teams. Technology alone doesn't guarantee success, so our approach integrates technical challenges with business objectives and considers the human factors critical for adoption and scaling.

Our framework, the Xebia Engagement Management Framework, ensures this balanced approach is consistently applied across all engagements, combining domain expertise with methodologies like PRINCE2 Agile, Prosci, and Lean Change, enriched by decades of transformation experience. It establishes standardized processes that ensure consistency, predictability, and reliability, thus allowing each engagement to leverage proven practices tailored to unique client needs.



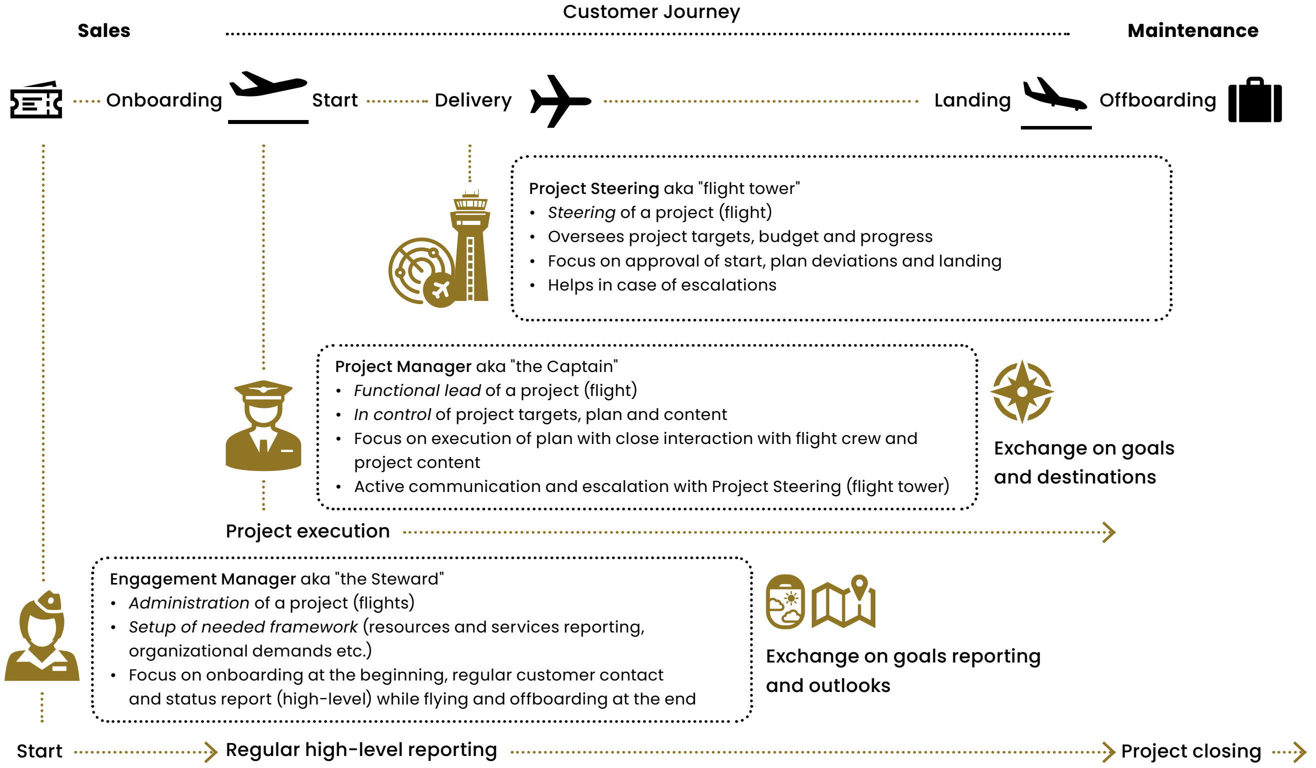
The Role of Engagement Managers

Engagement Managers at Xebia utilize the Xebia Engagement Management Framework to clearly define project scopes and outcomes, manage resources efficiently, facilitate transparent communication, proactively identify and mitigate risks, and uphold consistent quality standards. Their role is crucial in ensuring projects align with strategic business goals, deliver measurable value, and engage every stakeholder effectively, including managing the organizational dynamics that arise during transformation.

Engagement Managers act as facilitators between technical teams and business stakeholders, translating complex technical challenges into clear, strategic conversations. By building this bridge, they ensure mutual understanding, fostering trust and collaboration throughout the project lifecycle.

Engagement Managers ensure transparent reporting, forward-looking planning, and maintain customer control over project goals, phases, and budgets. Clear communication and actionable insights empower stakeholders, technical and non-technical alike, to effectively monitor progress, make informed decisions, and achieve measurable outcomes. While focusing on project deliverables, Engagement Managers also recognize the human impact of change to ensure new solutions are adopted effectively and deliver lasting benefits. By abstracting complexity to simplify stakeholder understanding, they facilitate confident decision-making and realistic planning.

Flight Metaphor: Pilot, Steward and Flight Tower



Project Steering aka "flight tower"
- *Steering* of a project (flight)
- Oversees project targets, budget and progress
- Focus on approval of start, plan deviations and landing
- Helps in case of escalations

Project Manager aka "the Captain"
- *Functional lead* of a project (flight)
- *In control* of project targets, plan and content
- Focus on execution of plan with close interaction with flight crew and project content
- Active communication and escalation with Project Steering (flight tower)

Engagement Manager aka "the Steward"
- *Administration* of a project (flights)
- *Setup of needed framework* (resources and services reporting, organizational demands etc.)
- Focus on onboarding at the beginning, regular customer contact and status report (high-level) while flying and offboarding at the end

To illustrate the role of Engagement Managers, consider the metaphor of a flight journey. The Engagement Manager acts as the steward, guiding passengers through every step of their journey. They take care of the passengers' preferences, manage their luggage – which symbolizes past experiences and lessons learned – and clearly communicate safety procedures, analogous to proactive risk management in a project context. Like stewards updating passengers on flight progress, Engagement Managers provide regular updates and keep stakeholders informed and reassured.

The pilot represents the Project Manager, responsible for navigating the plane and ensuring the flight reaches its destination safely and efficiently. The pilot focuses on operational execution, detailed planning, and technical direction, managing day-to-day project activities.

Overseeing the entire flight is the control tower – the project sponsor – who defines the destination, approves course changes, and ensures the project remains aligned with the business's strategic goals.

Importantly, these are roles rather than fixed positions – any consultant can fulfil these responsibilities. The critical aspect is ensuring that individuals clearly understand and diligently carry out their respective tasks, apply the best practices defined within the Xebia Engagement Management Framework, and collaborate closely. Together, the steward (Engagement Manager) and pilot (Project Manager) ensure a seamless journey, keeping stakeholders confident, informed, and empowered to make critical decisions effectively (specially the Project Sonsor in the Flight Fower).

Use Cases & Best Practices

Our framework applies proven best practices, combining agile principles with real-world transformation insights. Critical success factors include transparent communication, clear documentation and tracking of milestones, strong client collaboration, and integrated tools for real-time insights. Rigorous adherence to continuous improvement empowers teams to adapt rapidly and effectively to emerging challenges and opportunities.

Practical Example: A Typical Client Project
Consider a mid-sized organization undertaking a cloud migration:

- **Kick-off, Scoping and Alignment:** Initial workshops clearly define project objectives, KPIs, and outcomes using our framework elements, ensuring alignment across all stakeholders. This step also ensures that the right business goals are met, correctly measured and confirms that the offered scope remains aligned with these goals.
- **Value Driven Development:** Projects are executed using agile frameworks – using Scrum, Kanban, Scrumban, or tailored hybrids – with stand-ups, feedback loops, and open communication channels to maintain alignment and agility. These practices enable the team to respond swiftly to feedback and changing requirements. Additionally, asynchronous and modern communication channels and tools are set up from the beginning to ensure seamless collaboration across all participants, regardless of time zones or roles.
- **Risk Management and Transparency through Reports:** Proactive risk assessments and transparent communication ensure stakeholders are well-informed and in control, preventing surprises and fostering trust. At a minimum, checklists are used to cover all relevant topics and to ensure that nothing essential is missed throughout the engagement. Frequent reporting is established to maintain budget control and to document important decisions transparently throughout the project lifecycle.
- **Adoption and Scaling:** Post-implementation training, monitoring, and measurement ensure sustainable adoption and continued value creation, embedding solutions deeply into the client's operational fabric. At the end of the engagement, structured feedback is gathered to ensure no gaps remain after project completion.

Structured Implementation Recommendations

Implementing our Engagement Management framework involves:

- Clearly defining roles and responsibilities across project teams and stakeholders to ensure accountability and transparency.
- Choosing suitable methodologies and tools (e.g., Jira, Azure DevOps, collaborative platforms) that facilitate real-time tracking, transparency, and effective collaboration.
- Engaging stakeholders continuously through workshops, retrospectives, and regular feedback sessions to ensure alignment and continuous learning.
- Promoting a culture of transparency, accountability, and continuous learning, reinforcing a proactive mindset focused on collective success.
- Training teams extensively in agile methodologies, lean governance, and change management practices, equipping them to handle complex projects effectively.

Benefits of Structured Engagement Management

Effective engagement management brings multiple benefits:

- **Sustainable Value:** Ensuring alignment with strategic long-term goals and embedding solutions into client operations.
- **Employee Satisfaction:** Clearly defined roles, responsibilities, and transparent processes significantly reduce burnout and enhance team motivation and productivity.
- **Customer Success:** Consistent, high-quality outcomes enhance customer trust, satisfaction, and long-term relationships, creating opportunities for growth and repeat engagements.
- **Risk Reduction:** Early identification and proactive management of risks substantially lower potential negative impacts, securing predictable outcomes.

Conclusion & Outlook
Mastering Engagement Management is essential for navigating today's complex digital environments. By embedding our structured framework into every engagement, from small to large, Xebia ensures that projects are technically sound and strategically aligned, effectively solving customer problems, driving user adoption, and ensuring measurable outcomes.

Our holistic approach – uniting technology, business goals, and human factors – ensures that each engagement carries the distinctive Xebia signature: expert solutions, measurable value, and lasting adoption.

Startup Guide to Prompt Engineering Using GitHub Copilot

Prompt engineering is an essential skill for developers seeking to maximize the utility of GitHub Copilot. By structuring effective prompts, developers can receive accurate, context-aware suggestions that save time and enhance productivity.

Author **Randy Pagels**

This article explores foundational and advanced techniques to harness GitHub Copilot for writing code, debugging, and testing in C#, a language widely used in software development.

Core Concepts in Prompt Engineering

Prompt engineering strategies include:

- **One-shot prompting.** Providing a single example to guide the model.
- **Single-shot prompting.** Similar to one-shot, but often used interchangeably.
- **Chain-of-thought prompting.** Breaking down the problem into smaller steps.

One-Shot Prompting

One-shot prompting involves crafting a single, concise instruction. This method is best for straightforward tasks. It allows GitHub Copilot to generate code based on minimal input, making it efficient for simple functions or commonly known programming patterns.

Example `Reverse a string.`
Prompt `"Create a C# function to reverse a string."`
Output

```
using System;

public class StringManipulation
{
    public static string ReverseString(string input)
    {
        char[] charArray = input.ToCharArray();
        Array.Reverse(charArray);
        return new string(charArray);
    }
}
```

A clear instruction produces a simple and effective solution.

Single-Shot Prompting

Here, you provide an example input and desired output to guide Copilot. This technique helps Copilot understand the expected structure and format, making it particularly useful for tasks that require precise output patterns or specific conditions.

Example `Filter even numbers from a list.`

Prompt `"Given a list of integers, return a new list containing only even numbers. Example: Input: [1, 2, 3, 4] Output: [2, 4]."`

Output

```
using System;
using System.Collections.Generic;
using System.Linq;

public class NumberFilter
{
    public static List<int> FilterEvenNumbers
    (List<int> numbers)
    {
        return numbers.Where(n => n % 2 == 0).ToList();
    }
}
```

This approach ensures Copilot understands your expectations.

Chain of Thought Prompting

For complex tasks, break the problem into logical steps and guide Copilot progressively. This method helps Copilot generate more structured and accurate responses by incrementally refining the solution. It is particularly effective for algorithms, multi-step workflows, and scenarios that require sequential decision-making.

Example `Process user data from a CSV.`

Prompt `"Write a C# program to read a CSV file, filter rows based on a condition, and output the results."`

Output

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

class CSVProcessor
{
    public static List<string[]> ReadCSV(string
    filePath)
    {
        return File.ReadAllLines(filePath).Select(line
        => line.Split(',')).ToList();
    }

    public static List<string[]> FilterRows(List
    <string[]> data)
    {
        return data.Where(row => int.Parse(row[1]) >
        50).ToList();
    }
}
```

Clearly distinguish the prompt from the output.

Zero Shot Chain of Thought Prompting

Zero-shot CoT Prompting essentially involves adding "Let's think step by step" to the original prompt.

Example `Process user data from a CSV.`

Prompt `"Determine whether a number is prime. Let's think step by step."`

Output

```
using System;

public class PrimeChecker
{
    public static bool IsPrime(int number)
    {
        if (number < 2) return false;
        for (int i = 2; i <= Math.Sqrt(number); i++)
        {
            if (number % i == 0) return false;
        }
        return true;
    }
}
```

Breaking the solution into steps improves accuracy.

> **NOTE**
>
> Read more about Chain-of-Thought prompting here: https://www.promptingguide.ai/techniques/cot

Writing Effective Prompts

Clear prompts are the foundation of accurate outputs. Ambiguity leads to irrelevant or incomplete code. A well-structured prompt ensures that GitHub Copilot understands the developer's intent, leading to more precise and useful code suggestions. The more specific and descriptive the prompt, the better the results, as it reduces the need for unnecessary iterations and refinements.

Examples of Effective vs. Ineffective Prompts

Ineffective: `"Create a function."`
- **Effective:** `"Write a C# method that accepts a list of integers and returns the sum of even numbers."`

Ineffective: `"Sort a list."`
- **Effective:** `"Write a C# method that accepts a list of strings and returns the list sorted in alphabetical order."`

Ineffective: `"Generate a report."`
- **Effective:** `"Write a C# method that generates a sales report for a given month, including total sales, average sales per day, and the highest sales day."`

These examples demonstrate how providing specific details in your prompt can lead to more accurate and useful code suggestions from GitHub Copilot.

Multi-step Prompts

Progressively guide Copilot with incremental instructions.

Prompt 1 "Create a C# web API project."

Output 1

```
// Create a new ASP.NET Core Web API project using
   the .NET CLI
// Command: dotnet new webapi -n UserManagementAPI
```

Prompt 2 "Add a UsersController to manage users."

Output 2

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

[ApiController]
[Route("api/users")]
public class UsersController : ControllerBase
{
    private static List<string> users = new
    List<string> { "Alice", "Bob", "Charlie" };

    [HttpGet]
    public IActionResult GetUsers()
    {
        return Ok(users);
    }
}
```

Prompt 3 "Add a method to add a new user to the list."

Output 3

```
[HttpPost]
public IActionResult AddUser([FromBody] string user)
{
    users.Add(user);
    return Ok();
}
```

Prompt 4 "Add a method to delete a user from the list."

Output 4

```
[HttpDelete("{name}")]
public IActionResult DeleteUser(string name)
{
    users.Remove(name);
    return Ok();
}
```

Advanced Techniques

Here are more advanced examples using Copilot for performance optimization.

Prompt "Optimize the following C# method to reduce memory usage and improve performance."

Initial Code

```
public static List<int> GetSquareNumbers(List<int>
numbers)
{
    List<int> squares = new List<int>();
    foreach (var num in numbers)
    {
        squares.Add(num * num);
    }

    return squares;
}
```

Optimized Code Using LINQ

```
public static List<int> GetSquareNumbers(List<int>
numbers) => numbers.Select(n => n * n).ToList();
```

Using LINQ improves performance by reducing iterations.

Testing and Debugging

Testing and debugging are crucial in software development, and GitHub Copilot simplifies these tasks. With well-structured prompts, developers can generate unit tests, detect bugs, and refine code efficiently. Copilot automates test creation and suggests improvements, enhancing code reliability while reducing manual effort. It also helps troubleshoot runtime errors by offering fixes and explaining common mistakes. Leveraging Copilot for testing and debugging accelerates development and improves software quality by identifying issues early.

Example Instead of excessive code comments, use Copilot Chat for a more interactive debugging session.

Prompt "Why is my C# function returning incorrect results?"

Output

- "Let's check for edge cases. Are you handling empty lists correctly?"
- "Consider adding input validation before processing."

This interactive approach is more practical than embedding lengthy comments inside the code.

Generating Unit Tests

Copilot's ability to generate unit tests helps developers validate their code quickly.

Prompt "Write unit tests for a method that checks if a number is prime."

Output

```
using System;
using Xunit;

public class PrimeCheckerTests
{
    [Fact]
    public void Test_PrimeNumbers()
    {
        Assert.True(PrimeChecker.IsPrime(5));
        Assert.True(PrimeChecker.IsPrime(11));
    }

    [Fact]
    public void Test_NonPrimeNumbers()
    {
        Assert.False(PrimeChecker.IsPrime(4));
        Assert.False(PrimeChecker.IsPrime(10));
    }
}
```

Unit tests ensure correctness and catch regressions early.

Debugging Assistance

GitHub Copilot can assist in identifying and fixing bugs by analyzing existing code and suggesting improvements.

Example `Fixing a concurrency issue in a multi-threaded application.`
Prompt `"Fix the race condition in this C# multithreading example."`
Buggy Code

```
using System;
using System.Threading;

class Counter
{
    private int count = 0;

    public void Increment()
    {
        count++;
    }

    public int GetCount() => count;
}
```

Copilot Suggested Fix (Using lock for thread safety)

```
using System;
using System.Threading;

class Counter
{
    private int count = 0;
    private readonly object lockObject = new object();

    public void Increment()
    {
        lock (lockObject)
        {
            count++;
        }
    }

    public int GetCount() => count;
}
```

Using lock ensures only one thread can modify count at a time, preventing race conditions.

Real-World Testing Scenarios

Generate mock data for testing APIs or database operations effortlessly. Ensure the prompt is specific about test data.

Prompt `"Generate mock test data for a C# user registration system."`
Output

```
using System;
using System.Collections.Generic;

public class MockUserData
{
    public static List<(string name, int age)>
    GenerateUsers(int count)
    {
        var users = new List<(string, int)>();
        var rand = new Random();

        for (int i = 0; i < count; i++)
        {
            users.Add(($"User{i}", rand.Next(18, 70)));
        }
        return users;
    }
}
```

By specifying the user registration system, Copilot provides relevant mock data.

By leveraging these techniques, GitHub Copilot becomes a powerful tool not just for writing code but for ensuring its quality and reliability.

Summary

I hope you found this article helpful in kick starting prompt engineering with GitHub Copilot. Whether you're refining simple functions or tackling complex workflows, the right prompts can make all the difference. Keep experimenting, iterating, and optimizing your prompts for the best results. Best of luck on your coding journey!

Here are the key takeaways:

- Start with clear, specific, structured prompts for better responses and straightforward tasks.
- Leverage context, including comments and multi-step guidance, for complex workflows.
- Use Chain of Thought prompting and iterative refinement for handling complex challenges.
- Utilize Copilot Chat for debugging instead of excessive inline comments.
- Optimize performance using advanced techniques and structured refactoring.
- Incorporate testing and refactoring into your workflow to maximize Copilot's utility.

AI Promptception – Iterating GitHub Copilot Prompts for Maximum Impact

GitHub Copilot is an impressive AI coding assistant, but it's only as good as the prompts you give it. The difference between a useful output and a frustrating mess often comes down to how well you structure your request.

Author **Randy Pagels**

This article explores **how to improve prompt engineering for GitHub Copilot** using **real-world examples**, focusing on Playwright test automation.

User Story: The Frustration of Incomplete Tests

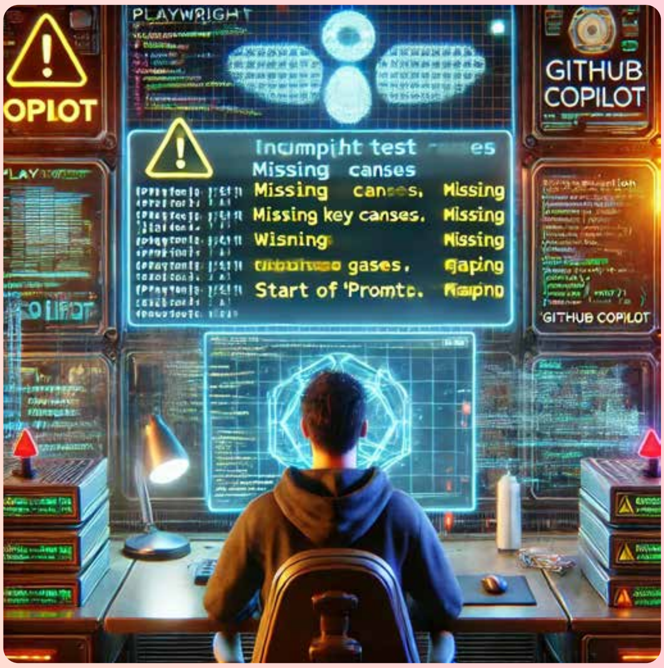
Imagine you're working on a critical project with tight deadlines. You've just finished building a React component, `PlaneList.tsx`, and now it's time to write tests using Playwright. Instead of manually writing each test, you decide to save time and use **GitHub Copilot**.

You type a simple prompt:

```
"Create the remaining tests for
#file:PlaneList.tsx based on
#file:PlaneList.spec.tsx."
```

The results? **Incomplete, generic tests** that miss key scenarios like **API failures, edge cases, and user interactions**. Frustrated, you realize that the problem isn't GitHub Copilot—it's the prompt.

This is where **Promptception** begins: using GitHub Copilot to **write better prompts for GitHub Copilot**, turning frustration into a powerful feedback loop for maximum impact.

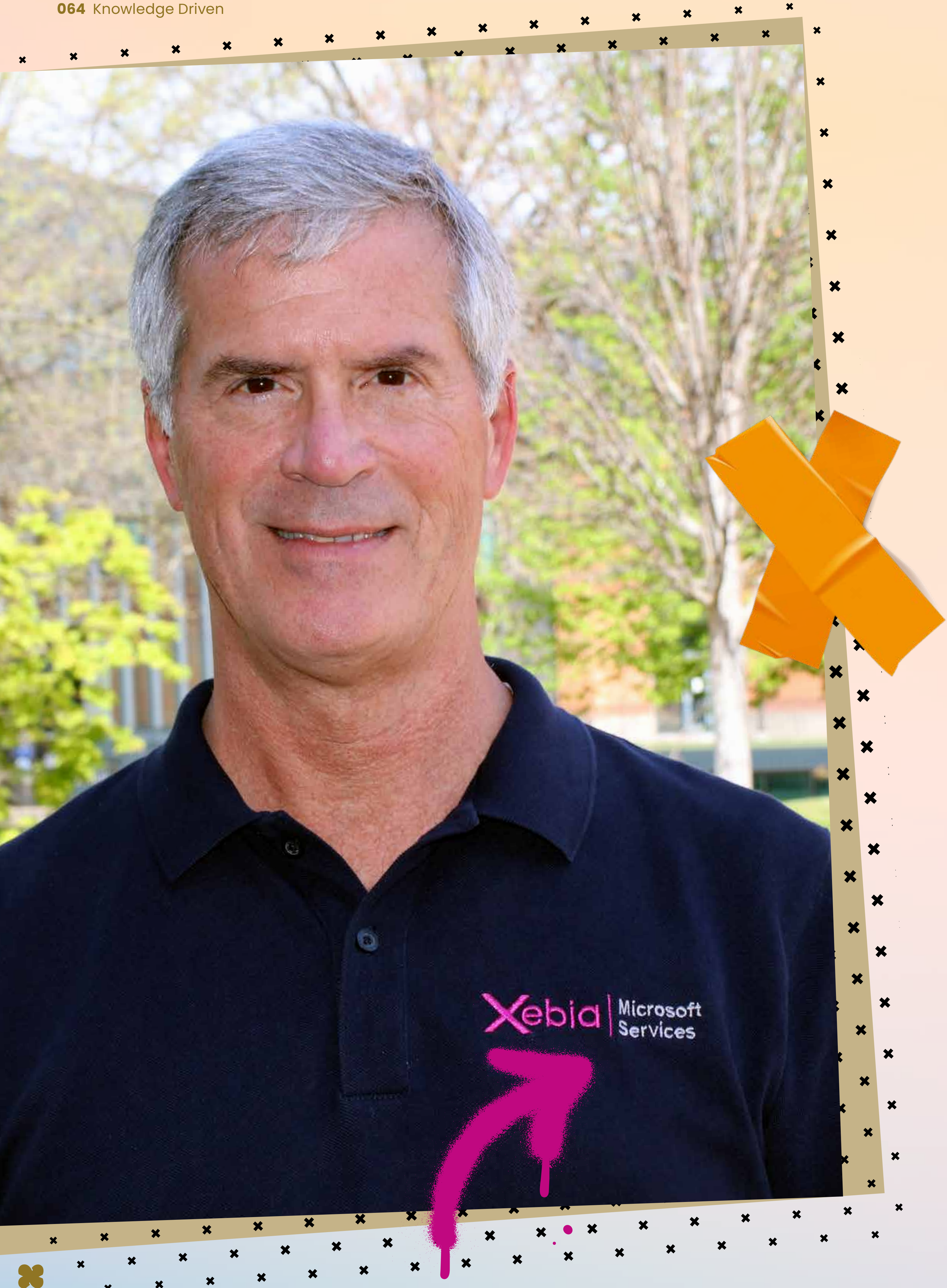


User Story on a Kanban Board

Example:

Title: Write Playwright Tests for `PlaneList.tsx`

- **As a** developer **I want to** generate comprehensive Playwright tests for `PlaneList.tsx` using GitHub Copilot **so that** I can ensure robust coverage for UI rendering, user interactions, and API edge cases.


Xebia | Microsoft Services

Acceptance Criteria:

- ✓ Tests cover all major UI components and their states
- ✓ Includes edge cases like empty lists, large datasets, and API failures
- ✓ Follows Playwright best practices with proper assertions and API mocking
- ✓ Accessible tests with basic a11y checks

The Initial Prompt

I needed to generate Playwright tests for a React component. I started with a straightforward request:

```
"Create the remaining tests for
#file:PlaneList.tsx based on
#file:PlaneList.spec.tsx."
```

At first, this seemed reasonable, but the results were **incomplete and generic.** GitHub Copilot lacked the right **context and direction**, leading to tests that:

- ✗ Didn't fully align with Playwright's structure
- ✗ Missed key testing scenarios
- ✗ Contained redundant or irrelevant assertions

This prompted a closer look at what was missing.

Prompt Review

After analyzing the output, I found **three key weaknesses** in my original prompt:

- ✓ **No mention of Playwright** – GitHub Copilot supports multiple frameworks. Without specifying Playwright, it wasn't clear what syntax to use.
- ✓ **Unclear test coverage requirements** – The generated tests lacked **important cases** like **empty lists, filtering, and API failures.**
- ✓ **No mention of best practices** – Playwright tests involve **event handling, API mocking, and accessibility checks.** Without guidance, Copilot didn't generate them.

At this point, I needed to **refine my prompt** to guide Copilot toward a better solution.

How I Used GitHub Copilot to Improve My Prompt

Sometimes, the best way to **write a better prompt** is to use **GitHub Copilot itself**. Since Copilot is trained on a vast amount of **coding patterns and best practices**, it can **help structure a well-formed prompt** that follows clear engineering principles.

Step 1: Started with a Simple Inquiry

I first asked Copilot:

```
"What are the key elements of a good AI prompt
for generating Playwright tests?"
```

Copilot responded with an outline of best practices, including:

- **Be specific about the testing framework.**
- **Clearly define expected test scenarios.**
- **Use structured bullet points for clarity.**

This confirmed that my original prompt was **too vague** and lacked the **details necessary for an accurate output.**

GitHub
Operation
Improving AI-powered code suggestions
Git Hub
GitHub Copilot in Dependency Improvements
Improvements & refinements
Improvements & refinments

Step 2: Asked for a Better Prompt Structure

To get more specific guidance, I refined my request:

```
"How can I improve this prompt: 'Create the remaining
tests for PlaneList.tsx based on PlaneList.spec.tsx'?"
```

Copilot's response **suggested adding**:

- ✓ The **test framework** (Playwright)
- ✓ The **type of tests** (end-to-end, component tests, UI interactions)
- ✓ A **list of required test cases**

Step 3: Iterating with Refinements

Now that I had a clearer structure, I asked Copilot to **expand on missing elements**:

- ✓ "Make sure it includes accessibility and performance checks."
- ✓ "Reword it to be clearer but still concise."

Result: I ended up with a **precise, structured, and effective prompt** that GitHub Copilot could **easily understand and execute.**

Prompt Refinement

Using Copilot's feedback, I restructured my prompt:

```
"Create the remaining end-to-end (E2E) and component
tests for #file:PlaneList.tsx using Playwright in
#file:PlaneList.spec.tsx. Ensure broad test coverage,
including:
• UI Rendering: Verify layout and default states
• User Interactions: Test clicks, filtering, and list
  updates
• Edge Cases: Handle empty lists, large datasets, and
  API failures
• Performance & Accessibility: Ensure fast rendering
  and basic a11y checks
• Mocking APIs & Network Requests: Simulate responses
  for dynamic content"
```

This small adjustment had a **huge impact**. GitHub Copilot **generated Playwright tests** that:

✓ Correctly used **Playwright's test methods**

✓ Included **real-world scenarios**

✓ Followed **best practices**

Breaking Down the Anatomy of a Good Prompt

A great prompt follows a clear structure:

- **Clarity** – State exactly what you need. Avoid vague or open-ended requests.
- **Specificity** – Mention the **framework** (Playwright), the **file** (`PlaneList.tsx`), and the **test type** (E2E, component tests).
- **Intent** – Clearly define the **goal** (e.g., ensure broad test coverage, mock APIs, etc.).
- **Structure** – Use bullet points or numbered lists to break down different test scenarios.

Iterating for Perfection

Even with a refined prompt, GitHub Copilot might need **small adjustments**. Instead of rewriting everything, **follow-up prompts** help fine-tune the output.

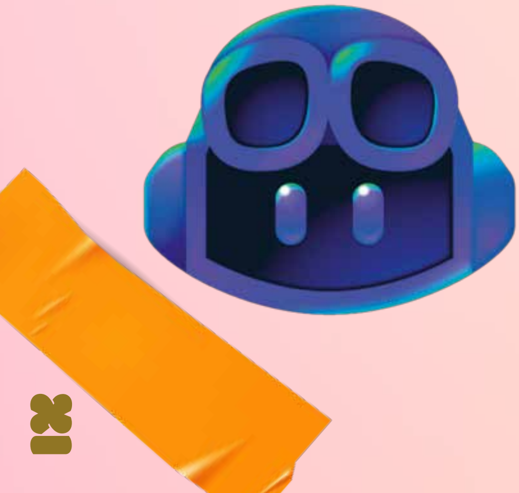
Example:

✓ "Refactor tests to use `test.use()` for shared setup."

✓ "Optimize API mocking to improve test speed."

✓ "Add accessibility assertions for buttons and inputs."

These **micro-adjustments** allow **GitHub Copilot to polish the output** without requiring a complete rewrite.



Avoiding Common Pitfalls

Even experienced developers make mistakes when prompting. Here are some **pitfalls to avoid**:

🚫 **Too vague:** "Write tests for this file."
  ✓ **Better:** "Write Playwright E2E tests for `#file:PlaneList.tsx`, covering filtering, UI rendering, and API handling."

🚫 **Too broad:** "Generate all tests needed."
  ✓ **Better:** "Generate tests for missing scenarios, including empty lists, large datasets, and API failures."

🚫 **Too rigid:** "Write a test exactly like this one."
  ✓ **Better:** "Write a test that follows the same structure but covers user interactions."

Why Prompt Engineering Matters for Developers

GitHub Copilot is more than just an autocomplete tool. Used correctly, it **accelerates development, improves test coverage, and reduces manual effort**.

However, it's not magic—**it needs guidance.** Prompt engineering is the skill that separates a **good developer** from a **great one** when using AI-assisted coding.

Final Thoughts & Best Practices

Here's a **quick reference guide** for writing better prompts:

✓ **Be specific** – Mention the **framework, file, and test type.**

✓ **Use lists** – Break down test cases instead of lumping them together.

✓ **Refine iteratively** – Use follow-up prompts to fine-tune the output.

✓ **Avoid vague requests** – Clearly state what's missing or needs improvement.

With these techniques, you can **turn GitHub Copilot into a precision tool** for generating **cleaner, more reliable code.**

Try GitHub Copilot for refining your prompts today and see the difference!

Creating
Digital Leaders
in the AI Era

Ready for something different? Join us.

Let's explore the opportunities together, no strings attached. Let's meet.

https://xebia.com/careers/

Xebia

Creating Digital Leaders in the AI Era

If you prefer the digital
version of this magazine,
please scan the qr-code.

xebia.com