Remote Development Environments: a Review

Dr Martin McCaffery and Waldemar Kindler,
Think Ahead Technologies GmbH*

24th October 2025

Abstract

Modern software development is a challenging endeavour, with many conflicting priorities. Security is paramount, but developer experience is growing in importance. We compare three paradigms intended to address these particular challenges. Providing a laptop for each developer is a traditional approach, simple to understand but with unavoidable drawbacks. Virtual Desktop Infrastructure is a mature centralised solution which concentrates sensitive material in a datacentre but may be more limiting for developers. Remote Development Environments are a much newer paradigm, attempting to take the best of both other approaches by allowing a developer to code locally but run code remotely.

Keywords: RDE, VDI, DevEx.

Contents

*Think Ahead Technologies GmbH (https://think-ahead.tech/) is registered in Stuttgart, Germany, under HRB 794174.

1 Introduction

Modern software development is a complex endeavour, with many conflicting priorities and challenges, from code constraints to personnel changes and organisational expectations.

Many of these issues relate to the day-to-day work by developers. Inconsistencies and errors can be introduced to code in a range of ways, e.g. developers running varying versions of a tool, having missed onboarding steps, or running code on different systems from the deployment environment - among many other possibilities. Beyond this, security flaws can be introduced by these same inconveniences or at the points of connection between the coding environment and the organisation's version control system or deployment processes.

Quite a number of these issues can be mitigated by ensuring the coding environment is controlled by the organisation as a whole: this allows it to be subject to internal regulation, security teams and more. There are three broad ways of accomplishing this, each with their own trade-offs in terms of security and developer experience.

1.1 Three approaches

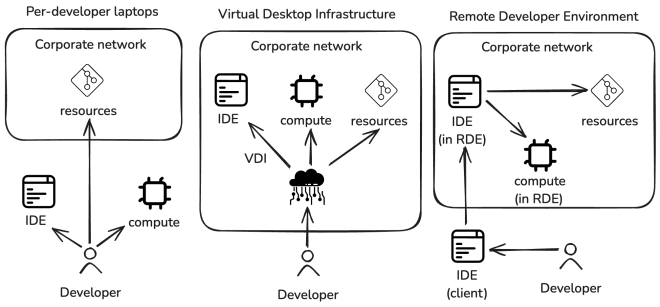


Figure 1: High-level comparison of three paradigms' architecture.

The first approach we will explore (Section 2) is to provide individual developers with **company laptops**, provided and set up by the organisation with any security features preconfigured. This has the benefit of developers themselves having wide control over what they do on the machine, and a generally comfortable development environment. The approach has a number of potentially severe security issues, however, along with the high logistical costs involved in procuring and providing laptops.

The second approach (Section 3) is to use **Virtual Desktop Infrastructure** (**VDI**, or **VDIs**). These involve full virtual machines being spun up within the organisation's infrastructure, and accessed as a remote desktop via a video feed. This is a relatively well-known solution, providing high levels of security by eliminating the need for (almost) any company information to be stored on developers' laptops. They are also easy to onboard and offboard, and have numerous other benefits. However,

the approach does not allow great specificity of configuration - e.g. supporting different versions of a language for different repositories - and the perpetual need for a video feed can lead to developer frustration, particularly with poor Internet connections.

The third approach (Section 4) is less well established, and is known (depending on context) as either **Remote Development Environments** (**RDEs**) or Cloud Development Environments. Rather than spinning up a full virtual desktop system, RDEs are smaller, more targeted, more ephemeral machines which contain development tools tailored specifically to the organisation's or team's context. More importantly, they can be accessed not by video feed but through an IDE directly, over `ssh`. This greatly reduces, and in some ways eliminates, the limitations of slow Internet, while retaining the security and other benefits of VDIs.

Each of these three approaches has benefits and downsides, and this document is not intended to provide a strong recommendation for any specific approach or tool. Instead, our goal is to provide sufficient context with which organisations can make decisions, taking into account the tradeoffs or the possibilities of combining approaches. We compare the solutions in Section 5.

1.2 Developer experience matters

While security is paramount in software development, an element which also has great influence but is occasionally overlooked is developer experience. This is a relatively novel area, based on the principle that development should be as simple as possible, facilitated rather than impeded by the tools developers use.

Developer Experience (or DevEx) is an area of growing interest, with a range of approaches and a growing consensus regarding its importance. In its formal introduction in 2012 it was described as "a concept that captures how developers think and feel about their activities within their working environments, with the assumption that an improvement of the developer experience has a positive impact on software development project outcomes" [9].

1.2.1 Satisfaction dividend

Since then, a range of research has investigated connections between productivity, satisfaction and numerous related factors. Not all of these use the term 'Developer Experience,' however: many simply explore links between productivity and emotional or other human-centred characteristics. The core hypothesis in many such studies is that satisfied developers are more productive.

Much research does indeed support the hypothesis, with methodology evolving over the years. Numerous studies have explored the relationship between developer emotions and productivity using a range of approaches, from commit logs and Jira issues to surveys and in-person observations [1]. Many of these have produced results which are either equivocal or not directly relevant to our situation, but some show clear and consistent correlations.

For example, an early small-scale study from 2013 used self-reported measures of productivity and of three different emotional attitudes during day-to-day work, and found a correlation for two of these [12]. The authors continued work in this vein, and found in 2017 that *low cognitive performance*, *low motivation*, *work withdrawal*, *low* ***productivity***, and *low code quality* stemmed from a negative experience of coding [13]. The following year, they again found comparable results in a larger but similar survey: poor experience led to *low* ***productivity***, *low code quality*, *lower motivation*, *work withdrawal*, *delay*, and *low focus* [14].

In 2019, an unrelated large-scale study using Microsoft as a case company explored a wide range of factors impacting both satisfaction and productivity. They found that for both junior and senior developers, the two elements were significantly correlated with each other [34]. Other factors which correlated with productivity included the engineering system and the work environment.

Another 2019 study explored a wide range of granular factors in three separate companies, again exploring relationships with productivity. Similarly to the previous studies, they found significant correlations between productivity and agreement with the following statements in the cohorts of at least one company: *I am enthusiastic about my job* (all three companies), *My job allows me to make*

decisions about what methods I use to complete my work (two companies) and *I use the best tools and practices to develop my software* (one company) [22].

1.2.2 DevEx formalisation

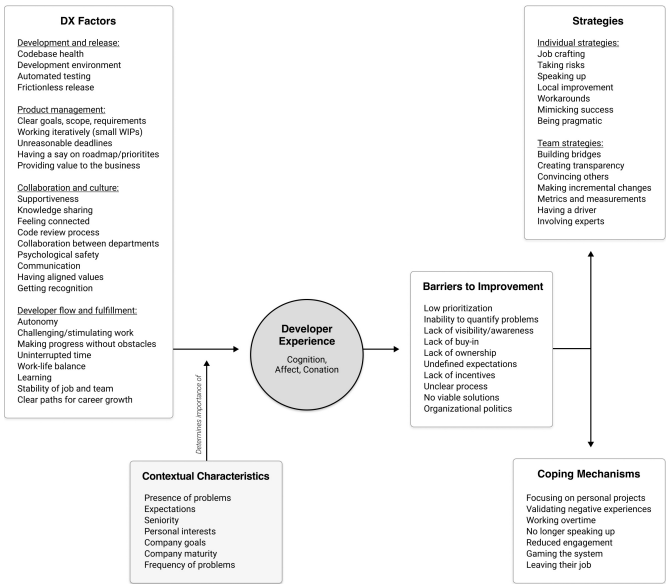


Figure 2: Summary of factors affecting Developer Experience, as proposed in *An actionable framework for understanding and improving developer experience* [15]

More recently, as DevEx as a concept has gained traction, work has taken place to formalise it using more objective characteristics than emotions.

A 2022 study performed detailed surveys with 21 developers and collated the results with a view to producing a broad "framework" through which to investigate Developer Experience (see Figure 2). As with the previously mentioned studies, they found that "Developer experience drives productivity, engagement and job satisfaction", while also providing more fine-grained recommendations for improving it within businesses. They emphasise that "Developer experience involves both organizational and technical challenges" and provide suggestions both on improving it directly and for how to approach the area in general [15].

A 2023 paper attempts to go beyond individual effects of DevEx to present three distinct "core dimensions of developer experience." These are: *flow state* (or *being in the zone*); *feedback loops* (delay in receiving feedback/validation for code, whether automated or manual); and *cognitive load* (how much developers must keep in mind at once). They also present something of a handbook on how to measure DevEx, with tips on survey design, suggestions for KPIs to use, and brief case studies such as at Pfizer [23].

A year later, the same team returned to present their own statistical investigation based on their defined approach. They explored the impacts at developer, team and organisation levels of changes to the three "dimensions" above, across a range of companies. They found most relationships significant to $p = 0.05$ or even $p = 0.001$ [10]. This suggests that helping developers enter and stay in "flow state" or (particularly clearly) reducing their cognitive load, though not necessarily providing faster feedback, is highly likely to improve performance at all levels of the organisation.

1.2.3 Corporate investigations

Academic research into Developer Experience is progressing, and has provided some useful insights and suggestions for firms to use. There is still, however, a lot which remains to be explored, not least because some of the most useful conclusions come from large-scale investigations in major organisations, which are themselves slow to set up and observe, and are often secretive about their results.

In the absence of formal academic studies, however, much useful information is provided publicly by companies. Some of this takes the form of formal reports, including the annual State of DevOps report showing in 2019 that useful, easy-to-use tools significantly improve productivity (see Figure 3).

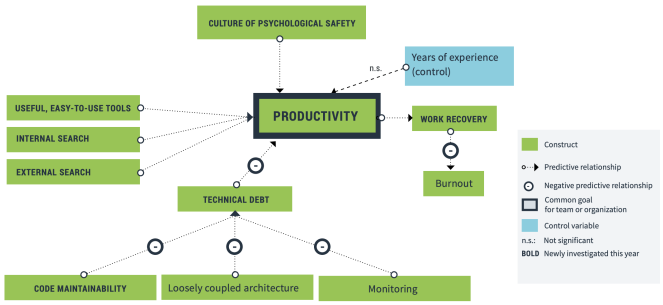


Figure 3: Research for the State of DevOps report has shown that useful, easy-to-use tools significantly increase productivity. [8]

Others are less formal, but potentially still useful. A 2024 Stack Overflow review ranked sources of frustration for developers, with tech debt and complexity of tools rating particularly highly [30]. An Atlassian publication from the same year reporting that 63% of their developers consider DevEx to be "Important" or "Very important" [3].

Other corporate publications include opinion pieces by companies either trying to influence DevEx practices – such as Thoughtworks, a consultancy [37] – or those selling tools intended to improve DevEx, like the Remote Development Environment software Okteto [27, 25].

1.2.4 Opinions

Overall, we consider DevEx to be an up-and-coming field which has demonstrated its importance: it provides some specific guidance and some broader trends which can be used to improve development organisations. However, given its relative youth, it is also not yet clear exactly how the relationship between DevEx improvements and productivity is affected by different organisational structures.

It is possible, for example, that while many simple changes to working environments may improve DevEx and consequently productivity, further changes may not have the same effect. As described in Figure 4, changes to developers' working environments may end up being overbearing, unnecessary, or have other unintended consequences which actually harm productivity more than they encourage it.

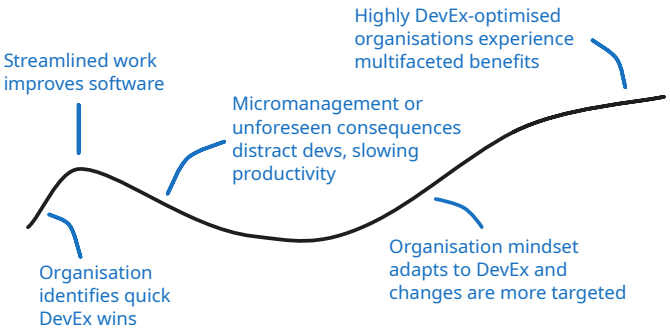


Figure 4: Curve describing possible inconsistent productivity gains from DevEx improvements.

Such a situation could occur as a consequence of management attempting to enact change without truly understanding developers' needs, such as by applying changes which have helped other organisations but are inappropriate for theirs due to a difference in scale or specialisation. In this case, issues can be mitigated if the organisation learns from these mistakes and calibrates future changes to be more in-line with the real pain points and characteristics of its developers.

We emphasise, however, that this complex evolution of DevEx is merely a possibility. Well-targeted

improvements are still likely to lead to powerful benefit in numerous areas, and even in the above scenarios we suggest it would still be worth pursuing DevEx. Working toward a more mature development experience is likely to bring short-term benefits, and will encourage ever greater software excellence as its practices become familiar.

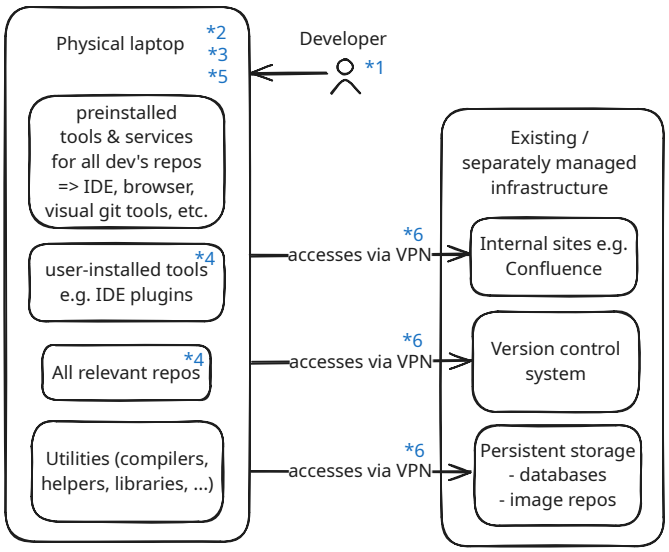
2 Solution: per-developer laptops

The first solution to the requirements we have discussed is to provide all developers with company laptops. Such machines can be provided by the organisation directly. They can be preconfigured according to a strict set of security requirements, allowing a reasonable level of control over developers' activities while still providing a flexible space to work. This allows a high level of homogeneity even if developers themselves are spread across a range of organisations, e.g. contractors.

Within the development space, a few essential security requirements must be considered. Most basic is that developers must of course access tools within the company's infrastructure for e.g. knowledge sharing (Confluence) or resource access (git repositories, docker images, etc.) These should not be exposed on the open Internet, so a networking solution, such as a VPN, is required. Such a tool can be pre-configured on each laptop, ensuring that only officially approved machines can access the corporate network.

More generally, unchecked software installations can pose significant security risks. To mitigate these, it is strongly recommended not to allow developers administrative access to the machine, as this could lead to malware being installed. Instead, users should either rely on preinstalled software for their day-to-day work, or – in case specific additional software is truly required – an administration team should first validate the software and then actually install it.

Furthermore, to limit the danger posed by theft, all laptops should be locked down: hard drives must be encrypted, and access should only be possible with a secure password. Laptops should also automatically lock themselves on inactivity, and users should be trained to lock them manually whenever they are unattended.



Notes on laptop paradigm
*1 Developer must learn laptop OS and preconfigured software
*2 Hard drive should be encrypted
*3 Developer should not have admin access on laptop
*4 Developer must set up some configuration manually
e.g. install plugins, download repos
*5 All information cached on laptop, vulnerable to wide range of malware
*6 Single layer of defence against most attacks

Figure 5: Architectural overview of per-developer laptops

2.1 Advantages

2.1.1 Familiarity

The first and most obvious advantage of providing a laptop to each developer is its familiarity. Developers are used to managing their own machine, and both they and peripheral staff are likely to generally understand the processes involved in obtaining, running and maintaining them. Laptops can be selected or configured to support any accessibility, language, security or other needs the organisation considers important. For contractors, the laptops provide a clear separation between their direct employers and the organisation, with each using different machines.

Similarly, within the organisation, the staff required to support this solution is fairly standard. A team must ensure the laptops are configured securely, while others might manage procurement and the logistics of providing machines to new (remote) joiners. The skills for such teams are relatively commonplace, with few unique complications that would make hiring or training unduly difficult.

One word of warning, however: while the concept of setting up a laptop for corporate users is well-known, developers represent a rather unusual use case. Many employees may be content with a stable suite including word processors, email and the like, but developers need both a much larger suite, and one which evolves rapidly. Specialist tools may be needed by one team or individual only, while the changing landscape of development may require frequent security reviews or installation processes. This may lead to further issues if admins are too permissive (constantly spending time installing varied tools, potentially with security flaws) or too restrictive (preventing developers using tools they need in order to be productive).

2.1.2 Longevity

Once a laptop is with its developer, one further strong advantage is that little maintenance work is required on the organisation's part before the developer leaves. Any urgent software updates can potentially be pushed by a central team using mobile device management software from third-party vendors like Microsoft's Intune [21]; beyond this, the developer themselves, in concert with their team, is responsible for ensuring any tools and software is up-to-date. In the best case, this can instil a sense of being trusted to set up one's own developer space and work as one likes within it.

2.1.3 Availability

Further benefits stem from the laptops' independence from organisational infrastructure, relating to reliability. Development work can continue under a wide range of adverse circumstances. In case of any outage or downtime – planned or unplanned – within the organisation's network, developers can continue their work locally: only resources in private datacenters, such as CI/CD systems or image stores, will be unavailable. This can be critical in the (hopefully extremely rare) circumstances where hotfixes are urgently required to fix an issue. In absolute worst-case scenarios such as destructive hacks, laptops can also represent backups of resources like git repositories.

Indeed, even without any issue with the organisation's infrastructure, laptops' independence can be beneficial for a developer if they are temporarily without access to the Internet, such as while travelling, when visiting a physical site where connectivity is restricted, or simply during periods of unreliability of Internet at remote employees' home offices. This allows work to continue with fewer interruptions, potentially leading to more reliable delivery.

2.2 Disadvantages

Despite the benefits, there are a lot of drawbacks of providing per-developer laptops. Most of these relate to security limitations, but a few more directly affect day-to-day coding.

2.2.1 Compatibility

Some of the (relatively) less impactful of the latter relate to compatibility. A laptop is a single device, and one which is costly to provide – while end users of any software products will inevitably use a range of devices. These will often have wildly varying operating systems and capabilities, making any

testing that takes place on a single developer machine inherently incomplete. This can be mitigated using remote machines, or occasionally with emulation, but these have their own significant downsides: the former requires significant bespoke digital infrastructure, while the latter can be inaccurate or slow on standard laptop hardware.

Even when a laptop is at a given moment ideally suited for the software it is developing, this may not always remain the case. Over time, different projects may use different versions of a tool, library or language, while others – particularly legacy tools common in larger enterprises and critical infrastructure – may lag behind. This can cause headaches for developers trying to work with multiple such projects together, particularly in cases where urgent fixes require a return to a less well-maintained project alongside day-to-day work on a more modern one.

Furthermore, upgrades can cause their own issues. A newer version of software, for example, may not support legacy hardware, or in the other direction a physical upgrade – like a new chipset, as happened with MacOS recently [33] – may introduce widespread compatibility issues. Such incompatibilities are irregular issues: most will have only limited impact, but they represent an ongoing risk of disruption.

2.2.2 Setup costs

Relative to software incompatibilities, the logistical requirements of per-developer laptops are likely to be a more consistent pain point, whenever new developers or contractors must be onboarded. All developers - even remote ones - would be required to pass by a central depository to collect and deposit machines, or a costly delivery system must be set up. Even a close-to-optimal process would likely involve delays and wasted person-hours, not to mention the cost of the system and the further cost of the laptops themselves.

Once a laptop has arrived with its user, the developer must then get used to the new machine. This may involve installing or setting up credentials for tools they use, providing a further one-time slowdown. Past this point, developers will need access to an IDE. Providing a single, homogeneous and free IDE to all developers – such as Visual Studio Code, used by 73.6% of developers anyway [31] – is likely to inconvenience some who either are used to, or would simply be more efficient with, a different tool. Alternatively, providing a selection of options – where some options are likely to require license fees – runs the risk of either needing a complicated installation and license procurement/management process, or the more expensive choice of wastefully paying for licenses for all employees.

2.2.3 Security

Furthermore, once the laptops have arrived with the developers, the more serious security worries come to the fore. As mentioned, we assume that administrator permissions are not provided, as these would allow developers to install arbitrary software with unpredictable potential security holes. We also assume laptop hard drives are encrypted to mitigate theft. Nonetheless, even with strictly controlled software, no administration privileges and locked-down hardware, several issues can arise.

One tranche of issues relates to network connections. With a full suite of company software, source code and documentation on the laptop, an attacker gaining access to the machine has many avenues for causing further trouble. Attackers may not even need to access the physical device, as developers working from insecure environments - be these at home or in public environments like coffee shops - may unknowingly open them to remote access.

A basic step to limit such network-based attacks is of course to ensure any communication with sensitive resources - e.g. company repositories - goes through a VPN. This eliminates a huge number of issues, but is not very finely configurable. The fact that the VPN itself will need to be accessed by developers in arbitrary locations raises two issues. First, it is difficult to prevent bad actors from accessing its endpoints and exploiting any security flaw in the VPN's own authentication. Secondly, once within a VPN, it is not convenient to further restrict access to different endpoints: in most cases, any employee can access any tool, even if they have no business need for it. This violates the principle of least privilege, leading to a large blast radius for any compromised account.

One final issue, particularly for highly regulated industries, is auditability. When each laptop is largely independent, communicating with organisational infrastructure only when it needs to, there is

no natural way to track individuals' work. This may be an issue when trying to trace unauthorised activity, or more broadly for regulatory compliance.

3 Solution: Virtual Desktop Infrastructure (VDIs)

Virtual Desktop Infrastructure (VDI, or VDIs) represent a well-established approach to solving the problems described above. Rather than a physical machine being provided to each developer, they involve a virtual machine being created in a datacenter. The developer can connect to this machine via a video link e.g. through the RDP protocol on Windows, seeing and interacting with its desktop as if accessing it locally. It can be used more or less identically to a physically accessible device, including (pre)installing software, connecting to other services over the network, and running terminals or visual software.

VDIs have existed for many years, and are used within many organisations of different scales and purposes. A wide range of specific products exist within the category, some tailored to specific use cases while others are broader. Note that a comprehensive survey of all products has not been carried out within this research project. This is partly for time reasons, and partly because many other resources exist which provide context and recommendations on the subject of VDIs. What follows is instead a high-level exploration of the pros and cons of the paradigm, plus a limited investigation of specific tools for context.

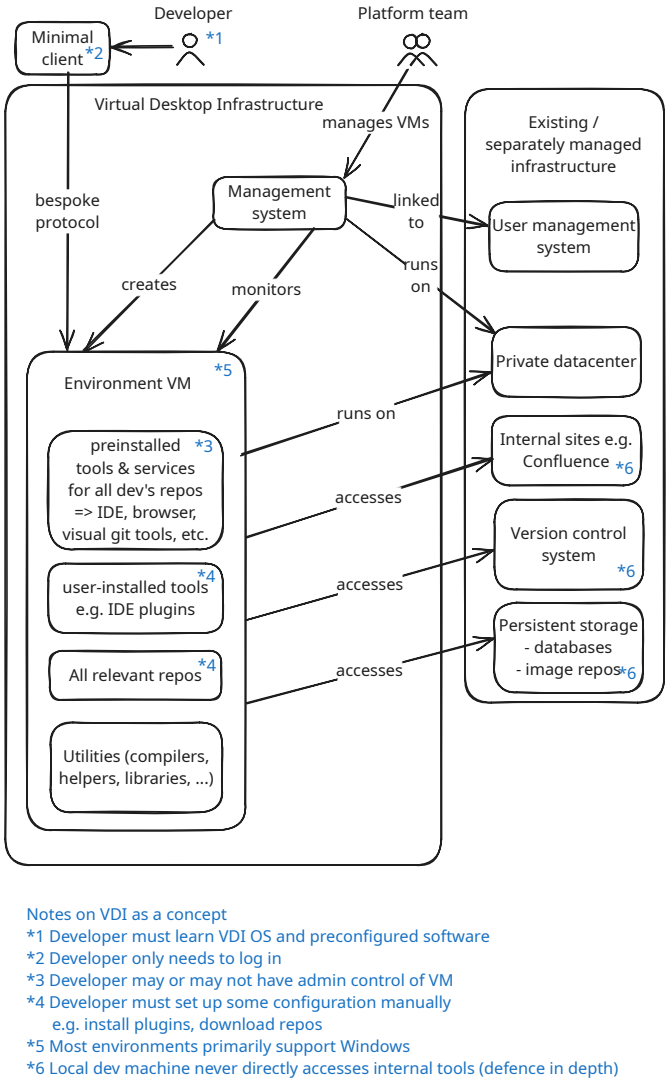


Figure 6: Architectural overview of virtual desktop infrastructure

3.1 Advantages

Virtual Desktop Infrastructure has powerful benefits, and relatively limited downsides when compared with per-developer laptops. The most evident advantages lie within the key domain of security, but reduced and predictable logistics provide other powerful reasons to consider them.

For developers themselves, VDIs provide the powerful benefit of being accessible from a wide range of devices. Any laptop, desktop computer, or potentially even tablet device can be used as an end device, as all these need to support is a connection to the VDI. Thus, hard drive space and processing power is largely irrelevant. This can be convenient when a project requires high levels of compute which would be impractically expensive to provide to each developer. It is also very useful when dealing with a large number of contractors, each with their own in-house devices, as there is little need for these to be standardised: they must simply have the VDI client software installed.

More situational benefits can also occur from the system being hosted within the organisation's network. Network proximity can be beneficial if a project requires many different processes to work together, with high levels of data transfer - e.g. between a test system and a heavy-duty database, or many microservices.

For the organisation hosting a VDI, another enormous benefit is the ease of setting up a new virtual environment. Rather than providing a physical device which must be purchased or stockpiled, new users can be set up with the simple click of a button. This both guarantees consistency and saves potentially significant time for technical staff.

3.1.1 Security

Arguably more important than these practical considerations, however, is the impact VDIs have on security. The chief benefit is simple: VDIs move (almost) all security questions from the developer's machine to the VDI, which is hosted in (conceptually) one single environment. This means that development security can be regulated uniformly within that environment. These can extend to the VDI itself being hosted within the organisation's own datacenter, and potentially on the same network (via VPN) as its users.

Effectively, this means that the development process can be subject to similar levels of security as any other software product, from internal personnel tools like Confluence to client-facing services. A security team can assess the needs of different groups, can restrict access in a targeted manner, and can apply policies across the board. This can be done in a more calibrated manner than a VPN, potentially allowing access to some tools from only some VDI instances' more-predictable IP addresses, rather than broadly allowing all corporate tools to anyone accessing the VPN. This would limit the blast radius of any attack.

The implication of these changes is that any loss of the developer's device has relatively minimal impact. Rather than losing control of a range of intellectual property and credentials, any loss is initially simply of an access point. Attackers must then take advantage of the client in ways limited by that client's security – which could, for added confidence, itself require a VPN.

Furthermore, even if access is gained, it can be revoked promptly by an administrator – likely in a single operation. As such, the impact of a wide range of attacks, particularly involving device theft, is primarily just to the individual developer. Even replacement of the device is a cost that may not fall on the host organisation, if the developer is employed as a contractor.

3.2 Disadvantages

While VDIs have powerful benefits in terms of security, they are not perfect. While for the most part their impact on security is enormously positive, they have one potentially significant flaw: connection. All communication with the VDI - thus, any development at all - must be authenticated by the system itself. This can be made secure in many ways, and is likely to be no weaker than any other single connection point, but their all-encompassing nature makes them a prime target for attacks.

If an attacker does manage to hack this step - by social engineering of credentials, man-in-the-middle attacks, or any other approach - they then have access to the full development environment and

potentially (depending on configuration) the broader corporate network. This is roughly equivalent to the attacker stealing a physical laptop with all software on it, except for a few factors: the potential network access; the possible lack of need to unlock the machine once accessed; and the possibility of such an attack taking place from anywhere in the world without the need for a physical theft.

Note that this security flaw can be mitigated in various ways, such as with multi-factor authentication, and extensive monitoring to detect anomalous access so credentials can be swiftly revoked. However, without such additional processes, hacks are realistic: the RDP protocol, used by numerous VDI tools, has numerous known flaws [36]. Alternative approaches are available, but RDP is used in a wide range of common tools [2, 35] so care should be taken with selecting and implementing a VDI.

One other debatable flaw is that VDIs involve a constant cost for the infrastructure they require. To a certain extent this is true for any solution, except per-developer laptops which have their own non-trivial costs. Depending on factors like the power of the relevant machines and the specific infrastructure setup, the cost of VDIs could in practice be less than alternatives - particularly if costs are saved by automatically spinning down the machines during 'off hours' like nights and weekends.

3.2.1 Developer experience

Separately from security, the main area where VDIs have potentially severe limitations is that of developer experience (see Section 1.2). The most obvious element here is that as a remote system, all development must take place over an Internet connection. More than this, because the connection is over a video stream, development can be quite intensive in its bandwidth use, and requires a high quality connection. This precludes situations where Internet is entirely unavailable, like most air travel, but can also impact more common situations like train travel, work from public spaces like libraries, or even many standard work-from-home setups.

In these situations where the Internet connection is poor, the impact can be jarring. Keystroke delay, between pressing a key and it appearing on screen, can cause great frustration in itself, while even short cut-outs of Internet can interrupt concentration. Distracted developers are prone to human error in a wide variety of ways, from missing potential bugs / edge cases to changing working patterns to optimise for low keystrokes. Loss of Internet also leads more directly to slowdown, as developers are unable to work for periods of time.

Other issues with developer experience on VDIs come as possible consequences of their security benefits. Strict control over the setup of the software in a VDI may lead to inconvenience when developers are restricted from installing all tools they might desire. Similarly, locked-down network setups may restrict access to websites developers find handy, from Stack Overflow and Medium.com to more targeted tools. These can potentially be accessed from the developer's end device, but then copying and pasting from the host machine to the VDI may be either restricted (again, for security reasons) or simply clunky.

Furthermore, while VDIs can be set up automatically with a range of software, they are still quite broad-purpose systems on which a developer may be expected to work on different tools across different repositories for different projects. This means that they may waste resources on tools not being used at any given moment, or introduce confusion through multiple versions of a language - say, node.js - being needed in different contexts. Such issues are normal in the world of per-developer laptops too, and are manageable, but eliminating them might nonetheless be desirable if such a thing were possible.

3.3 Industry trend

Virtual Desktop Infrastructure as described thus far represents a broad range of products in the market. Many of these are long-standing, mature offerings which present well-known features and little risk; even newer ones are operating in an established niche. We will not discuss individual VDI software in this report, as there are many resources online which explore the strengths and weaknesses of different offerings in greater depth than would make sense here.

However, one trend is worth noting. While the age of the VDI market suggests stability, in recent years at least two major vendors have changed significantly. Citrix, an industry leader, was taken private in 2022 and merged [11] with another software provider, Tibco, after several reports of underwhelming share prices. This has then led to significant redundancies and an updated product vision [19].

In a move more targeted to VDIs, VMWare, another industry leader in virtualisation, has been in flux for some years. It was sold in 2022 to Broadcom, who made aggressive pricing changes which significantly changed its approach to its customers for better or worse [38]. VMWare's Horizons VDI, among other tools, was then sold on in 2024, resulting in a landscape described as risky [28].

These transactions do not change the fact that both Citrix and VMWare's VDI products are well-established market leaders and viable choices when considering using such a tool. There are also many other tools unaffected by these changes. However, the fact that large-scale changes are taking place around some of the most-used tools in the sector could suggest a greater degree of flux and uncertainty than the age of the field would indicate. VDIs are a powerful tool to solve the problems discussed in this report, but their sector is equally not devoid of risk.

4 Solution: Remote Development Environments (RDEs)

While per-developer laptops represent an extremely common paradigm, and Virtual Development Infrastructure is an established alternative used in a wide range of situations, the third paradigm we will explore is much less well-known. Remote Development Environments are a comparatively new approach to the questions of secure, effective development, and are well worth exploring.

4.1 Concept

What is a Remote Development Environment (RDE)? Commonly also referred to as Cloud Development Environments, these describe a range of systems and ecosystems in which development can take place. Primarily cloud-based systems, their crucial difference relative to VDIs is that not *all* aspects of development take place on the remote system. Instead, a developer can connect an IDE, running on their own end device, to a personal instance of an RDE; any execution or compilation of the code then takes place there.

Note that as an RDE's primary interface is an IDE, they generally do not have their own graphical user interface. All communication instead takes place over `ssh`, or through web interfaces like remote terminals and web-browser IDEs. This does not prevent visual access to e.g. web apps being developed: these can be viewed - and in some cases even shared - via workspace- and port-specific URLs, or accessed locally via port forwarding.

RDEs represent something of a hybrid between the paradigms of per-developer laptops and VDIs: rather than all coding taking place on the end device, or all on the remote system, RDEs attempt to represent the 'best of both worlds.' They are primarily marketed as tools for increasing developer productivity, but they also have the potential to provide comparable security to VDIs.

RDEs are a much younger technology than VDIs, with pioneers in the field dating from 2014 [16] and 2016 [7]. Many younger competitors, however, are relatively immature or represent approaches we would not recommend for security-related reasons.

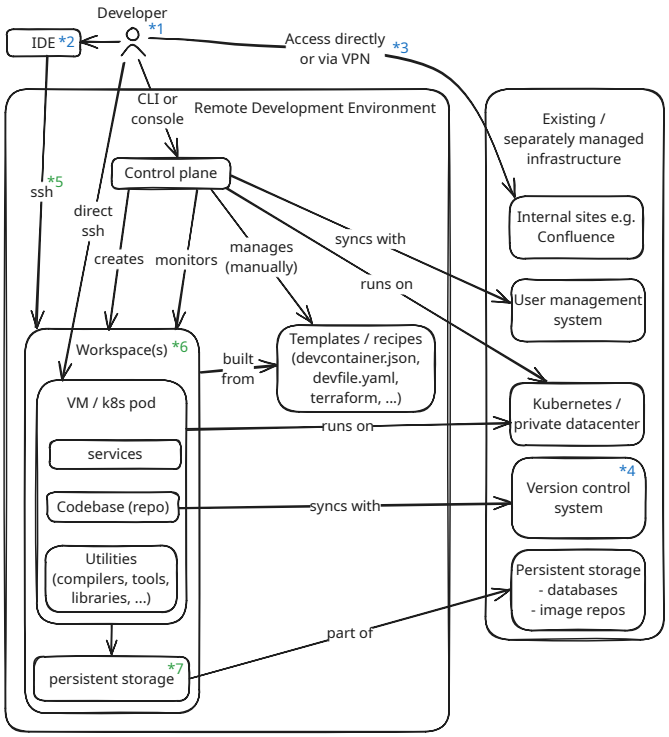
4.2 Advantages & Disadvantages

As a (relatively) new technology, RDEs are far from standardised. As such, their pros and cons inevitably vary depending on the exact choice of product. Nonetheless, many solutions, share some characteristics worth discussing.

4.2.1 Security

The most important of these relate, of course, to security. Here, we see similar benefits to VDIs (see Section 3.1.1), albeit with some limits. To summarise: almost all code execution and network activity takes place within the RDE, allowing strict control of network access, along with much lower risk of theft of intellectual property from end-device loss when compared to per-developer laptops.

In an absolutely maximum-security system, code too would be stored exclusively remotely. However, the developer must of course have access to their code: in such a setup, showing the text on their screen would require either a VDI-style video feed, or a limited `ssh`-based system like `vim`. In practice,



Notes on RDEs as a concept
*1 Developer has minimal range of tools they must learn
*2 IDE can be on local machine, or
(most tools support) via browser / web IDE entirely within RDE
*3 Visual tools/resources must be accessed separately from RDE
*4 Local developer machine never directly accesses VCS (defence in depth)

Notes on specific RDEs
*5 Okteto has code on local machine and syncs with remote env over ssh
⟹ allows local-only development
⟹ somewhat reduces security
*6 Some tools host workspaces on the developer's machine (i.e. not actually "remote")
*7 Depends on RDE product

Figure 7: Architectural overview of generic remote development environments

most RDEs implement some form of caching, such as storing open files on the developer's machine (whether as files or in memory), communicating them to the server on every change, and clearing them from the cache when closed.

Such a caching system has powerful developer-experience benefits, as discussed below, but it does represent a limited security risk. Any attack which accesses the machine itself could potentially claim any cached files, or even collect a whole repository by watching as files are opened during normal development. Furthermore, malicious developers can relatively easily send data to third parties by copying raw text, something much more difficult to do – though not impossible – via a graphical link or through terminal editors like `vim`. (Note that even with a VDI, caches could lead to security weaknesses if an attacker obtains session tokens or other details of an active connection: no system is perfect.)

Note that some tools adopt a less security-focused approach. The least secure we have observed is to have the full codebase stored on the developer's laptop, synced by the RDE daemon to the remote device where it executes in a consistent environment [26]. This has the potential to provide great benefits in terms of the speed of development - with the company claiming a velocity increase of 50% for a major client coupled with a 50% reduction in bug incidents [24] - but leaves the codebase at the mercy of the laptop's own security measures.

RDEs represent above all a *targeted* approach to development, optimising for the most prevalent operations such as coding and execution of code. This has a range of benefits, but one downside is that some daily activities do not fall within their scope. These include internal documentation tools such as Confluence, and access to web frontends for tools like Harbour or Grafana. Such tools must be managed by a separate system, likely a VPN, potentially leading to some of the network-access risks described in Section 2.2. This also requires more administrative overhead, and potential software licensing costs, than simply providing the VPN alone along with per-developer laptops.

4.2.2 Developer convenience

While security is paramount for many modern enterprises, and this can exclude many RDE offerings from consideration, this does not invalidate the approach as a whole. With products boasting stronger security practices and structures, a number of powerful benefits can be seen.

The first relates to the caching discussed above. With code files stored temporarily on the developer's machine while they are being worked on, frustrations from keystroke delay are eliminated. Internet connectivity is needed to execute any changes or open new files, but during brief moments of down-time on intermittent connections, the developer can nonetheless continue working on any open files. Furthermore, the communication primarily being text-based over `ssh` means that even slow Internet can potentially be functional, with the RDE neither imposing high usage nor demanding high responsiveness.

Another benefit likely to be greatly appreciated by developers is that with many RDE tools, they have the flexibility to choose any IDE that they choose, at least within the VS Code or JetBrains ecosystems which represent a large fraction of the market [31]. This eliminates questions potentially posed by VDIs or strictly controlled developer laptops over software installation.

With the other tools, administration privileges may be required to install software, costing time and effort, and an organisation may be tempted to support one IDE over others for simplicity of software suite or to reduce licencing costs. With an RDE, the choice of IDE is entirely up to the developer, with any licencing costs falling to their immediate employer (which may not be the hosting organisation). This allows developers to work in the way they consider most productive.

4.2.3 Configuration

RDEs' flexibility is not limited to allowing different developers to use their own IDEs. Additionally, RDEs generally support tailored configuration even across repositories. By committing files containing standardised definitions (see Section 4.4) of dependencies required for a repository, teams can control precisely which dependencies are preconfigured for their developers. This may involve specific versions of a programming language, or separate binaries needed for compilation. It can even include modules

for different IDEs. These configurations can then be updated simply by modifying the configuration file within a repository, allowing teams a high level of ownership of their code.

This preconfiguration allows developers to feel at home the moment they start coding on a new repository, greatly reducing onboarding time without sacrificing understanding [29, 4]. It also creates consistency between developers on a team without demanding uniformity across all projects. While a VDI could be set up with preinstalled tools, these would likely be uniform across all areas of the business, including projects with very different aims, histories, and technical makeup.

Specifying configuration in the repository thus allows a level of consistency which arguably maximises coding productivity across varied teams. As a helpful side effect, it also provides an auditable view of teams' preferred setups, creating potential ways to review development practices and easily share good practices. It also creates minor efficiencies by not requiring all tools to be included in all projects: users who do not need extensive tools can have simpler, slightly faster-loading environments which contain only what they need.

The automation of RDE setup, coupled with their relatively minimal scope when compared with VDIs, also has another benefit: speed of creation. Generally, the process of creating a workspace is similar to creation of a docker container: it involves a single large download – which may be cached – and supports a number of subsequent installation steps if desired. The speed this allows means workspaces can be treated somewhat ephemerally, terminated and recreated with minimal cost. Note that some elements within workspaces, such as downloaded repositories or home-directory `.rc` files, can be retained according to the same per-repository configuration as the rest of the RDE.

Workspaces' ephemerality is a powerful tool, lowering the barrier to good practice approaches which other paradigms do not encourage. For example, regular upgrades of entire environments, for example to apply security patches, can be applied with minimal inconvenience to developers who neither need to wait long for their system to come back online, nor need to reconfigure their settings. Similarly, project changes - such as upgrading versions of a programming language - can be applied without any fear of old installations persisting. Cost savings from systems being disabled during downtime (evenings, weekends) also apply, similarly to VDIs.

The goal of per-repository configuration is to empower teams to set up their environments as best suits them. However, this is not purely a positive thing, as it introduces the potential of security flaws if, for example, a team member accidentally (or even maliciously) installs malware, or the team fails to update a tool with a critical vulnerability. This is a difficult problem to solve without also negating the key benefit of empowerment.

Environments can of course be configured centrally, by a platform team, according to an organisation-wide image (or small group thereof) containing all software teams are expected to need. This has the minor benefit of images being widely reused, potentially increasing caching and thus startup speed. However, requiring a centralised team to make any changes risks slowing down some teams, providing unnecessary bloatware to others, and enforcing a potentially unhelpful uniformity across teams. This could, for example, inconvenience a team not yet ready to upgrade to the next major version of a language or tool, just because a different team wishes to use a new feature.

An alternative approach, smoother for teams but potentially more risky, would be to regularly audit all configuration files across all repositories. Tools like Snyk or OWASP's DependencyTrack validate either base configuration or resulting docker images, and compare them with up-to-date vulnerability lists. Their use should be coupled with formalised processes ensuring any revealed vulnerabilities are dealt with. Ideally, such processes would be run by teams, but in case of severe vulnerabilities the platform or security team's input may be required.

4.2.4 More double-edged swords

We have seen that in both security and developer empowerment, RDEs have powerful benefits which inherently come with their own potential downsides or limitations. In practice, such double-edged swords are not restricted to those two domains.

The simplest limitation RDEs impose on themselves is that they are targeted to a specific task, namely development. This means they do not cover some everyday activities, notably those involving graphical

user interfaces. For example, database access that isn't supported within the user's IDE must be done on the command-line, while separate visual git tools like GitKraken or SourceTree become unwieldy. Many such operations can be covered by modern IDEs, but exceptions will always exist. In any case, forcing users to use their IDEs for such situations limits their options, contradicting the goal of allowing them to work however they wish.

For visual tools which must be used, including internal websites like Confluence, the picture gets more complicated. Or rather, as the RDE does not have a graphical interface, it arguably reverts to the situation of per-developer laptops: a VPN is required. There are likely non-trivial numbers of such tools in any organisation, effectively meaning a VPN will always be needed alongside an RDE. The scope of such a VPN can be restricted, precisely because it does not need to cover developer sandboxes, internal databases and the like, but a handful of the attack vectors from Section 2.2 are still likely to apply.

Another benefit with a potential downside is the ease with which RDEs can be spun down when not in use. This saves costs, but means that accessing an RDE after a period of inactivity may require it to be redeployed. The time required for this can be minimised by streamlining images and configuration, but there is always likely to be *some* inconvenience to developers. At peak times like Monday morning, when many developers may be starting their workspaces at once, this could also lead to excessive load and further slowdown.

And of course, the key difference between per-developer laptops and any form of remote environment has its obvious downside: RDEs require an Internet connection. They have much lower needs relative to VDIs, as discussed in Section 4.2.2, but working entirely offline is simply not supported. This could be relevant when working on-site around physical infrastructure. Similarly, very high latency or intermittent internet is still likely to cause frustration and inconvenience to developers.

Finally, it is impossible to discuss a remote system without acknowledging that the infrastructure required for such a thing is non-trivial. Platform and security teams will require a high level of expertise to manage the clusters on which RDEs will be run, and to ensure capacity is available but not wasted. Much of the day-to-day functioning of an RDE should ultimately be automated, but setting up such automation is likely to be costly, and ongoing maintenance will always be required.

4.3 Configuration approaches

Remote Development Environments can be configured in a number of ways. We will explore a few, in increasing order of complexity.

4.3.1 Universal image(s)

The simplest approach is for each RDE to be created as a generic environment from a base docker image containing all software the organisation expects its developers to need. This simplifies administrative effort in configuration, and may be sufficient for some smaller organisations. It also allows for simplicity of storage, with only one (or a handful of versions of an) image to be accessible at any given time. The image itself, moreover, can be (relatively) easily checked for security flaws, and any flaws can be mitigated with a single change.

However, the one-size-fits-all approach has a few clear downsides. First, the image required for such a breadth of tooling is likely to be very large. This is liable to slow down setup times, while also costing resources in terms of storage – not to mention any background processes needed for some situations which are simply wasted processing in others. Beyond this, any teams using a variety of versions of a tool or language in different repositories will run into clashes.

Furthermore, and arguably even more inconvenient: any updates to such versions must pass through the team responsible for the image, potentially leading to significant inconvenience if that team is slow or rejects a change. This can be particularly inconvenient when legacy tooling requires pinned, deprecated versions of a tool; in an ideal world these would not be necessary, but often in practice long-lived systems can easily end up relying on such dependencies.

A large image, and centralised control of it, can furthermore cripple the ability to effectively test such

images. Features are likely to be added on behalf of different teams, to be used in ways which are likely not to be followed closely by the team managing the image. When a change is made, the only way to be confident a new version is issue-free is then arguably to have every team run their workloads on it and see if it breaks. If it does break, multiple teams are blocked or must revert back to previous versions, slowing down development either way.

Note that an alternative to a single universal image would be to have a limited selection available, e.g. one per business unit or per team. This may mitigate some of the limitations of a single image, but does not remove – and indeed, depending on the number of images, may exacerbate – the inconvenience of a centralised team needing to control updates.

4.3.2 Per-repository configuration

The issues with universal images relate to their separation between the consumers of an image (a team responsible for a project) and its creators (a central platform team). This separation can, however, be removed, if an RDE's configuration is placed under the control of the individual teams: in configuration files stored within their repositories. This allows small, targeted RDEs to be spun up for such repositories, with everything needed but nothing unnecessary.

There are several benefits to storage configuration in a repository. Beyond simply removing the potential blockers of an external team, such configuration can be much more detailed than would be practical for a universal image. More than simply specifying generic tooling and company-wide preferences, it can allow individual teams to choose their own ways of working, from IDE setup and plugin choice to preferred package managers and deployment orchestrators.

This can give a team better cohesion, particularly when working on varied projects and approaches. Testing also becomes much easier, as teams know how their own tools are to be used – and any errors they make are likely to have a blast radius no larger than the team itself.

The primary downside with such configuration is that it gives teams a form of indirect admin control over their RDEs. If a new tool is desired, it can be added to the configuration and an RDE will be built with it – even if that tool is insecure, deprecated or otherwise problematic. Such concerns can be mitigated by automated checks or notifications whenever a configuration is updated, so a central team can validate such changes. However, a balance must be struck with this, as too much or too-strict validation could represent a de facto return to dependency on that external team. We believe a balance can in practice be found which renders per-repository configuration the most desirable approach.

4.4 Configuration standards

Configuration for an RDE – whether universal or per-repository – can take several forms. The simplest is a Dockerfile: this allows specifying a wide range of details, from an operating system to a language, dependencies, and even individual codebases if required. This allows for a flexible, powerful configuration for a single machine. If more than one container is required, docker-compose is able to define networking and other relationships within the docker system.

Docker on its own, however, is a very generic tool, primarily used to create specific, complete machines for targeted purposes like deploying a specific application. Development environments are somewhat different, with some common elements – like IDEs and their plugins – and others which vary even between 'identical' environments, such as per-user home directories or codebases. To streamline such features, multiple standards have been created; the most common are devcontainer.json and devfile.yaml [32].

One goal of such standards is to allow configuration tailored for the unique paradigm of RDEs. This also benefits the testing, and thus the reliability, of such environments. Development environments, when fully built, contain a mix of technology, configuration and project-specific code; by specifying these in clearly separated sections of a single configuration, it becomes (relatively) easy to test each in isolation and thus gain confidence that the numerous possible combinations are reliable.

The devcontainer.json standard is intended to allow generic definition of containers across different environments, building on (and referencing) docker images. Its single configuration file can specify

different dependencies to be installed on development environments, CI or testing environments, and production. A community-led endeavour initiated by Microsoft, it has been adopted by a wide range of tools.

The devfile.yaml approach is similarly community-led and open-source, but built around a more centralised set of templates. Where devcontainer configurations can be copied from existing deployments or built from scratch, devfiles can also be pulled from common repositories, both public (maintained at least in part by Red Hat) or private.

In practice, these two standards have similar capabilities and can be used in very related situations. Due to the active involvement of Microsoft in devcontainer.json, it has a high level of integration with their popular Visual Studio Code IDE, which may be more or less of a draw than devfile's repository system, depending on an organisation's preferences and toolset.

It is worth noting that not every aspect of an RDE is supported by these standards. They all allow configuration of individual worker nodes – e.g. Kubernetes pods, or VMs – but none supports the specification of underlying machines, Kubernetes clusters, and other low-level details. Generally RDE tools attempt to provide their own configuration for this, with greater or lesser levels of abstraction depending on the product.

4.5 Testimonials

Exploring the high-level features of RDE offerings is a valuable activity, as is exploring the theoretical bounds of the paradigms. Both, however, are best when complemented with real-world experience of the paradigm or tools. During this report, only some tools have been explored firsthand, within testing environments – details are below – but other companies have had their own journeys with the paradigm. Some, such as Meta [20], merely mention that their developers often run their code through a combination of remote servers and IDE plugins, while others describe their journeys in more detail.

Discord, for example, has been experimenting with remote development since 2020, when they began using Coder [33]. Their blog post describes some pros and cons of the approach as they see them, and mentions a range of conclusions from different points in the migration process. They focused on a small-scale deployment initially, with "champions" chosen to test new functionality, before scaling up to a final fixed switch-over date. Their deployment was accelerated when the latest Mac chipset was released, disrupting their on-device workflows.

Discord found Kubernetes-based containers to be surprisingly insufficient for a development process involving tools needing privileged access. This is a conclusion also discussed in depth by GitPod, a provider [39]. Discord found the latest version of Coder – with a number of changes in networking and container use [33] – to be much more performant. They also speak of the importance of documentation and other processes to ensure developers are not inconvenienced while adapting to the new system.

Unlike Discord, LinkedIn chose to implement their RDE entirely in-house [17]. Their "RDevs" are accessible over SSH from developers' machines and configured "with all the necessary tools and packages" for a given project. They emphasise the benefits not just of the higher compute power of cloud machines, but of the time saved when dependencies can be downloaded within the same internal corporate network. They report four-fold or even much greater improvements in build time and dependency retrieval relative to local development.

Shopify has similarly built a bespoke system, by adapting their local system "Spin" to be cloud-native [18]. This involved a lengthy process of building out multiple solutions, iteratively improving the overall experience through relatively unspecialised VMs, to Kubernetes pods with Dockerfile-based configuration, and finally a customised Linux distribution. Each step was taken alongside regular consultation with developers, becoming progressively more cloud-native on the way. One central challenge was keeping the implementation simple enough to help rather than confuse developers, with intermediate solutions requiring detailed knowledge of their system's internal processes to use smoothly.

Overall, the companies with experience in the paradigm appear to agree that it is complex. Developing such a system in-house runs into a number of common challenges, such as onboarding developers without inconveniencing them, and the technological limitations of using standard Kubernetes pods to manage something with as broad requirements as a development environment. Using off-the-shelf

tools can be much more straightforward, but immature tools – such as Coder before its v2 redesign – can introduce their own complexities or limitations.

On the other hand, each case study appears to be pleased with the approach overall, with each different solution or step on the journey leading to benefits – often powerful ones – for their users. In short: RDEs, particularly immature ones, represent a risk as they can involve unforeseen challenges – but the benefits of a functioning one can be enormous.

4.6 Sample Requirements

Should an organisation decide that they wish to procure an RDE for their developers, the next question becomes: how to select a specific product? Many companies can simply explore the available options and select one, but for some organisations a public tender is required. The exact criteria that must go into such a tender will vary depending on the organisation, but many elements will be common to many contexts.

Thus, to facilitate the process of creating such tenders, we have produced a list of sample requirements for RDE products. These can be found in Appendix A. Our hope is that these requirements may serve either as the basis for future public tenders, or as inspiration for more informal selection processes within less regulated organisations.

5 Solution comparison

We have presented three solutions to the problem of secure, streamlined development. Each has its own intrinsic benefits and issues, but the most relevant question is how they compare to each other. Are the downsides of one greater than those of another? In what ways could they be combined?

To explore these questions, we will consider the implications of each paradigm within a range of scenarios. These scenarios are not intended to represent all aspects of development, nor to favour one paradigm over another, but rather to illustrate and collect the widely varying impacts the choice of development environment can have.

After presenting these scenarios as objectively as possible, the rest of this section will present the authors' subjective take on the trends they reveal. This is intended to both provide suggestions for how to make decisions, along with enough information to disagree with those suggestions if desired.

5.1 Situational comparisons

Note that only notable effects are included in this list; for example, the cost in effort of spinning up a VDI or RDE is considered negligible. Similarly, elements with similar effort across all paradigms – such as creating accounts for new users – are omitted.

5.1.1 Hiring new developers

Laptop All candidates are familiar with laptops

VDI Some candidates may not like VDIs

RDE RDEs may be too new and confusing for some

5.1.2 Hiring new admins

Laptop Security and logistics are long-established fields

VDI Cloud engineering is a newer but mature field

RDE Relatively few engineers are familiar with RDEs

5.1.3 Onboarding new user

Laptop Laptop must be bought, set up, collected, configured, provided with code

VDI Instance must be configured, provided with code

RDE Repository must be configured. Individual RDE setup is then trivial

5.1.4 Offboarding / termination

Laptop Laptop must be returned & reset

VDI Machines can be shut down centrally

RDE Workspaces can be shut down centrally

5.1.5 User tries to install new software (dev not admin)

Laptop Physical access may be needed to approve

VDI Centralised hosting simplifies admin team input

RDE End-device software has minimal impact on security

5.1.6 User tries to install new software (dev is admin)

Laptop Limited safeguards against user-installed malware

VDI VM in controlled environment, reducing risk but increasing impact from attacker gaining access

RDE End-device software has minimal impact on security

5.1.7 End-device theft

Laptop Data may be stolen if drive not encrypted. Full onboarding must be repeated, with cost to organisation

VDI Developer can switch to any other reasonable device using VM client; cost to immediate employer only

RDE Developer can switch to any other reasonable device with just IDE; cost to immediate employer only

5.1.8 Device left unattended & unlocked

Laptop Passer-by could install malware or steal IP directly

VDI Passer-by could install malware or steal IP by copying from VDI to host

RDE Difficult to steal/affect more than currently open repo

5.1.9 Credentials stolen

Attacker:

Laptop needs physical device, then has broad access (incl. write permission) to code and potentially VPN

VDI can access code and software without physical presence. VM can be given more granular access than VPN allows, limiting blast radius

RDE can access code without physical presence. Workspace may be limited to single repo, limiting blast radius

5.1.10 Tracing a hack

Laptop Little knowledge of who was working when

VDI Easy to track who was connected when

RDE Easy to track projects each dev is working on

5.1.11 Downtime (weekends, nights)

Laptop No effect

VDI May be shut down / restarted automatically

RDE Workspaces can be shut down centrally

5.1.12 After downtime

Laptop *N/A*

VDI Machine restarted; state persists

RDE New RDE spun up with selected data persisted

5.1.13 Copying from Confluence

Laptop Straightforward, via standard VPN

VDI Straightforward, via VDI

RDE VPN needed separately from RDE

5.1.14 Copying from Stack Overflow

Laptop Straightforward

VDI VM's browser likely slower to use than host's. Ease of copying from host to VM depends on VDI product

RDE Straightforward

5.1.15 Hotfix needed on long-dormant legacy project

assuming any relevant configuration was set up on previous use.

Laptop Need to install legacy tools. Potential version clashes. Potential confusion from lost institutional knowledge.

VDI VDI snapshot can be reloaded. Not standard practice; on long-lived VDI, same as above.

RDE Configuration can be read from dormant repo: barely different from working on modern project.

5.1.16 Mismatching tool versions between active projects

Laptop Requires manual solving

VDI Requires manual solving

RDE Simple: use different RDE instances for each project

5.1.17 Manual software testing for different architectures

Laptop Requires separate VM(s)

VDI Multiple VMs can be spun up for a single user, but not standard practice

RDE Simple: just change RDE config within repository

5.1.18 DevOps coding, deploying to company infrastructure

Laptop Potential to restrict deployment to users on VPN: somewhat secure

VDI Potential to locate all VDIs in same network as all deploy targets: secure

RDE Potential to locate RDEs in same networks as relevant deploy targets: very secure

5.1.19 Long-term maintenance

Laptop Little on day-to-day basis; infrequent expensive laptop replacements

VDI Full highly-available cloud infrastructure needed

RDE Full cloud infrastructure needed, plus (e.g.) VPN

5.1.20 Long-term risk profile

Laptop Security risk landscape is always evolving

VDI Mature field, but currently evolving its business models

RDE Relatively young field could evolve in unpredictable ways

5.2 High-level comparisons

5.2.1 Local vs. remote coding

When broadly comparing solutions for efficient, secure development, one of the primary high-level questions to ask is: should we go remote at all? Using remote infrastructure would offer the potential for consistency, control and efficiency, but may come at a cost in terms of upheaval, maintenance and a more centralised risk.

We consider this question one of the simpler ones to answer. One of the important questions in software development today is security, and on this front both remote approaches offer benefits that laptops simply cannot. As discussed in Section 2.2.3, having a laptop for each developer creates a range of flaws – from networking requirements to vulnerability to theft, and the relative difficulty of deep auditing.

Adding to the security worries, the logistics of laptops are likely to be more costly and inconvenient than those of remote systems. This is less clear-cut, however, as remote systems must be supported through extensive, sophisticated infrastructure which requires expertise and constant maintenance. We expect (but cannot guarantee) that in the long term these costs will be lower than those of procuring, provisioning and providing laptops.

Even if we assume the cost and effort of laptops are equivalent to those of either remote system, however, two broad factors still lead us to prefer the latter approach. One is the structure of that cost and effort. With laptops being inherently (relatively) decentralised, changes are likely to involve inconvenience to all developers using the machines. Any changes to software or orchestration are likely to involve familiarisation with a new tool/process at best, or physical reconfiguration of machines at worst.

On the other hand, with a remote system, more changes can be enacted behind the scenes. Upgrades to basic features like operating systems, networking configuration or software dependencies can be managed by a central platform team and rolled out during off-peak hours with (in theory) minimal impact to individual developers. This centralisation does come with the corresponding risk that any major *problems* with the system are likely to be felt by all developers at once. This is far from a trivial risk, but can be mitigated with good practice in a way that laptops' shortcomings cannot.

Beyond potentially limiting the impact of changes, this centralisation has numerous benefits stemming from the flexibility it offers. Rather than a physical machine which for financial reasons should be used for as long as possible, remote systems can be altered, upgraded and terminated at will: they can become 'cattle' rather than 'pets', serving a purpose without imposing their individual needs and headaches in return.

- What if a new project requires much more compute than a laptop offers? Simply spin up a new VM or RDE using the existing systems.
- What to do when a new developer joins the company? Simply click a button and set everything up as usual.
- Our existing environments aren't compatible with the latest dependency versions! Simply tweak your configuration and spin up new environments.

Many of these points only apply in specific scenarios, or are better suited to one or other remote solution, but the sheer number of different scenarios – both foreseeable and unforeseeable – where change may be needed leads us to prefer these approaches.

5.2.2 VDIs vs. RDEs

After choosing whether to use a remote environment at all, comparing the two remote options is a more challenging endeavour. Each has powerful strengths, and many of their weaknesses are similar. Both offer a powerful combination of consistency and flexibility. Both also require major investments in infrastructure to support a technology all developers will need to use, along with potential learning at many levels around how to secure, maintain and access such infrastructure.

We consider two major elements to be key to the decision between the two paradigms. The first is somewhat nebulous: RDEs are simply newer technology, much less widespread and understood. This poses risks for any organisation adopting them, risks which would reduce over the coming years as others experiment and expand our collective knowledge of how to best leverage the technology. It is possible that an organisation might stumble into pitfalls which could not reasonably have been predicted, or might get locked into a product which ends up being far from the best in class. With VDIs, on the other hand, the landscape is relatively easy to explore and predict.

The flipside of this risk, however, is the potential of a new technology to provide improvements older tools cannot offer. Some of these benefits are predictable, and are explored in Chapter 4. Others are likely to emerge as the technology grows, and more products introduce novel ways to leverage the base concept of an RDE in ways that VDIs cannot offer. Whether this potential to be part of a new movement within the development space is worth the risk is unquantifiable, and depends primarily on the attitudes of individual decision-makers.

In pure practical terms, also, the two approaches differ noticeably. VDIs, with there full-featured operating systems, are likely to require a large, but potentially consistent, amount of compute power, leading to a high cost for infrastructure and energy. RDEs, on the other hand, are likely to require significantly less compute – although their needs may vary dramatically depending on the projects currently under development. The relative novelty of RDEs may also lead to more logistical effort in setting them up.

The other main aspect where VDIs and RDEs differ is in their levels of centralisation. Both of course involve their infrastructure being managed centrally, but from developers' perspectives they look quite different. A VDI is much closer to a remote equivalent of a simple laptop, providing the consistency of environment and 'traditional' flexibility of having an operating system governing every aspect of their work. RDEs, on the other hand, provide more granular control over the environment, allowing easy targeted changes while omitting many elements – most notably a GUI – which developers may be familiar with.

The specific benefits of both levels of centralisation are discussed in the two paradigms' respective sections, but at a high level they represent different mindsets around software development. Should it involve a developer in their controlled environment, coding in the well-understood style of everyone around them? Alternatively, should they be in an environment they have built, in collaboration with their team, which allows flexibility but makes oversight more complicated?

5.2.3 Combining paradigms

Choosing between physical laptops, VDIs and RDEs involves weighing numerous factors against each other. Rather than choosing a single approach, however, one could attempt to mitigate the downside

of any individual one by combining several. This can often result in a combination with few of the benefits of any individual option and potentially additional unique downsides, but some combinations can be helpful.

Combining **company-issued laptops with a VDI tool** may be the simplest potential combination, as the two tools offer the most similar experiences. Each represents a standard operating-system environment, which the user is expected to use for an extended period of time, allowing anything not requiring administrative permissions. The primary differences from the developer's perspective are the need for an Internet connection, and the possibility for the environment to be reset or altered remotely.

Given the similarities, there are not a huge number of situations which would require use of one environment over the other. As such, there is little meaningful benefit for the developer from the combination. For the organisation, on the other hand, it allows some developers to move to a new technology while others remain on the existing system.

The price of this flexibility, however, is high: the full logistical processes (and costs) for providing laptops are needed, alongside the fully-fledged datacenter infrastructure needed for the remote environment. In addition, attackers may choose either environment as a target, effectively maximising the security risks.

A similar argument applies to the combination of **company-issued laptops with RDEs**: both approaches must be fully supported and defended. This makes the combination difficult to recommend, but it has some benefits which stem from the limits of each individual approach. Laptops, as discussed earlier, are fixed in their compute power and architecture; on the other hand, RDEs do not necessarily support third-party GUI tooling such as video-call software or database management.

As such, a developer with access to both could run software on any supported architecture, with as much processing power as the private datacenter allows, while the organisation can be secure in the knowledge that any non-coding work is still strictly controlled. This reduces the risk of malware to close to zero, at the expense of limiting some of the developer's freedom of choice of tooling.

Ultimately this decision should be decided according to the decision-makers' level of risk-avoidance. We believe that the coding security provided by RDEs is sufficient even for highly regulated sectors, on the condition that some level of auditing is done to ensure end devices used by contractors are within reasonable parameters. On the other hand, it may be justifiable to simply take absolutely all measures possible to reduce development risk, no matter the cost or limitations that entails.

Another way to achieve the lowest possible security risk would be to combine a **VDI and an RDE**. This would allow developers to use any end laptop they desire, with the only stipulation being that they must install the VDI client and any VPN it requires. Once thus connected to the corporate network, they would have access to a fully-provisioned VDI machine. The RDE would further allow testing in varied environments and using tools configured by the team for each project.

This approach would thus theoretically allow maximum flexibility along with maximum security, and may not require a significantly increased infrastructure approach as both tools are remote and could run on the same datacenter. However, this view is somewhat simplistic, as numerous aspects of the configuration might quickly become onerous.

First, the different compute requirements of running RDEs alongside VDIs would significantly increase both the cost and the complexity of the infrastructure. Second, the experience for the developer would be, at best, equivalent to simply using the VDI alone. At worst, they would need to log onto a VPN, then a VDI, then spin up an RDE, before being able to run their code. This would then provide functionality equivalent to the laptop-and-RDE combination discussed above, without the benefit of being able to continue coding without an active Internet connection.

Overall, of the three options for combining approaches, we consider only laptops and RDEs to be genuinely viable. Even this combination depends on the decision-makers' level of risk tolerance, based on the limited security questions discussed in Section 4.2.1. The other two possibilities, we feel, add significant complexity and restrict developers unduly, without providing commensurate benefits. (Combining all three technologies would then add further complexity and even less relative value.)

5.2.4 Inevitable interactions

While formal, intentional combinations of different paradigms have their own upsides and downsides, it is difficult to avoid some such interactions. These come about from fundamental aspects of the software engineering process, or of individual business domains. These do not affect the merits or otherwise of individual paradigms, but may marginally affect the costs of implementation.

Firstly, it is evident that every developer requires a laptop to work on their code. This does not necessarily have to be fully development-ready as per Section 2 if a remote paradigm is used, but some form of machine will be required. Thus, the logistical process of procuring and providing them must exist in almost any organisation, and should not fully be considered a cost of the choice of per-developer development laptops.

On the other hand, when using an RDE or VDI the laptop can have minimal specs, reducing costs and potentially lengthening their useful lifetimes. Furthermore, when a remote paradigm is used then third-party contractors can connect directly from their own devices with a 'bring your own machine' approach, further reducing costs.

Another tool which may be required in any paradigm is a VPN. With per-developer laptops, one is required for all interactions between the developer and any protected or internal resources; however, for RDEs it is equally necessary for a more limited range of tools. Even with VDIs, it may be helpful to use a VPN to secure the client, adding a powerful second layer of protection to the primary attack vector.

6 Conclusion

Software development in the modern era is a complex endeavour. Security is a primary concern, while developer experience is gaining increasing relevance. Writing code is merely one part of a process involving planning, testing and deployment, with each step offering unique challenges and potential solutions.

Development is, however, a very significant element of this process, and changes to coding processes are likely to have broad impact within the organisation making them: changes in practice, mindset and the relationship between parts of the organisation. As such, any solution is also likely to impact the other steps in the software pipeline, and should be chosen with care. This is particularly true within the area of security, where weaknesses in one step can negate good practices elsewhere.

This report has focused on the impacts to development itself of three distinct paradigms. Each is able to address security, developer experience and much more. Our goal is to provide sufficient information with which to make decisions between the three paradigms.

Our priority within this has been objectivity, exploring trade-offs and implications within the development space. Impact beyond development is not within the scope of this report, largely because this highly depends on the individual characteristics of the organisation in question – but the authors urge decision-makers to think of any such implications.

Within the three paradigms, per-developer laptops are a traditional, well-understood method of providing compute to developers. They involve high logistical costs and potentially greater security risks than remote solutions, but they do not require as powerful a private cloud and are resilient in the face of intermittent Internet connectivity.

Virtual Desktop Infrastructure provides many of the benefits of per-developer laptops, providing a somewhat long-lived complete space for developers to work. They offer high security and are relatively well-known within the industry, making them a relatively easily justified choice – apart from the potential developer experience questions, and their inherent requirement that developers be connected to high-speed Internet during all development work.

Remote Development Environments are a newer approach, providing security, simplicity, consistency and a smooth developer experience across projects in ways the other paradigms cannot match. Some elements of a developer's working day are not supported by RDEs, however – notably visual internal tools like Confluence – and their novelty makes their infrastructure maintenance more challenging.

6.1 Further work

This report contains an overview of each of our three paradigms, presenting the aspects which appear most important. However, when discussing three dramatically differing approaches to software development, there are inevitably limits to the analysis possible. A continuing research project, or later works in the area, would have the potential to provide broader insights.

The main area in which this report is limited is its investigation of individual tools. While a few were set up and experimented with, any in-depth investigation was out of the scope of this research. Additionally, in a novel and fast-growing field, point-in-time evaluations would be severely limited in their potential to meaningfully inform.

Beyond RDEs, we have not performed a detailed review of VDI tools, or of individual providers or processes for per-developer laptops. VDI tools, as stated in Section 3.3, are well-understood: numerous comparisons and details reviews exist elsewhere. On the other hand, per-developer laptops are more challenging to explore as there is an extremely wide range of possibilities for their selection, security and logistical elements.

Finally, as stated above, this report focuses purely on development. The choice between the three paradigms we present is far-reaching, and could be seen as representing different approaches an organisation can choose to the broad questions of software delivery. We encourage any reader to consider all aspects of their software delivery process – including those far outside the scope of this targeted report – when making decisions about the future of their organisation.

References

[1] Mohammed R Anany, Heba MW Hussein and Sherif G Aly. 'A survey on the influence of developer emotions on software coding productivity'. In: *International Journal of Social and Humanistic Computing* 3.3-4 (2020), pp. 216–244.

[2] Ann Ming Samborski and Gabriel Bauman and Athanasios Filippidis and Mike Borkenstein. *RDP without the risk: Cloudflare's browser-based solution for secure third-party access*. 21st Mar. 2025. URL: https://blog.cloudflare.com/browser-based-rdp/ (visited on 11/04/2025).

[3] Atlassian. *State of Developer Experience Report 2024*. Tech. rep. Atlassian, 2024.

[4] Frederick P Brooks Jr. *The mythical man-month (anniversary ed.)* Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] Bundesamt für Sicherheit in der Informationstechnik. *BSI TR-03185 Secure Software Lifecycle*. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03185/BSI-TR-03185.html. Nov. 2024.

[6] Bundesamt für Sicherheit in der Informationstechnik. *ISO 27001 Zertifizierung auf Basis von IT-Grundschutz)*. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Standards-und-Zertifizierung/Zertifizierung-und-Anerkennung/Zertifizierung-von-Managementsystemen/ISO-27001-Basis-IT-Grundschutz/iso-27001-basis-it-grundschutz_node.html (visited on 15/03/2025).

[7] Coder Technologies, Inc. *About - Coder*. URL: https://coder.com/about (visited on 05/02/2025).

[8] DORA. *Accelerate State of DevOps 2019*. Tech. rep. DevOps Research & Assessment, 2019.

[9] Fabian Fagerholm and Jürgen Münch. 'Developer experience: Concept and definition'. In: *2012 international conference on software and system process (ICSSP)*. IEEE. 2012, pp. 73–77.

[10] Nicole Forsgren et al. 'DevEx in Action: A study of its tangible impacts'. In: *ACM Queue* 21.6 (Feb. 2024). URL: https://www.microsoft.com/en-us/research/publication/devex-in-action-a-study-of-its-tangible-impacts/.

[11] Antone Gonsalves. *Thousands of Citrix, Tibco employees laid off following merger — TechTarget*. 11th Jan. 2023. URL: https://www.techtarget.com/searchenterprisedesktop/news/252529104/Thousands-of-Citrix-Tibco-employees-laid-off-following-merger (visited on 03/04/2025).

[12] Daniel Graziotin, Xiaofeng Wang and Pekka Abrahamsson. 'Are happy developers more productive? The correlation of affective states of software developers and their self-assessed productivity'. In: *Product-Focused Software Process Improvement: 14th International Conference, PROFES 2013, Paphos, Cyprus, June 12-14, 2013. Proceedings 14*. Springer. 2013, pp. 50–64.

[13] Daniel Graziotin et al. 'Consequences of unhappiness while developing software'. In: *2017 IEEE/ACM 2nd International Workshop on Emotion Awareness in Software Engineering (SEmotion)*. IEEE. 2017, pp. 42–47.

[14] Daniel Graziotin et al. 'What happens when software developers are (un) happy'. In: *Journal of Systems and Software* 140 (2018), pp. 32–47.

[15] Michaela Greiler, Margaret-Anne Storey and Abi Noda. 'An actionable framework for understanding and improving developer experience'. In: *IEEE Transactions on Software Engineering* 49.4 (2022), pp. 1411–1425.

[16] Tyler Jewell. *Eclipse Che: Microservices für Eclipse*. 23rd June 2015. URL: https://entwickler.de/eclipse/eclipse-che-microservices-fur-eclipse.

[17] Shivani Pai Kasturi. *Building in the cloud with remote development*. 4th Nov. 2021. URL: https://www.linkedin.com/blog/engineering/cloud-computing/building-in-the-cloud-with-remote-development (visited on 03/04/2025).

[18] Don Kelly. *The Journey to Cloud Development: How Shopify Went All-in on Spin - Shopify*. 9th June 2022. URL: https://shopify.engineering/shopifys-cloud-development-journey (visited on 03/04/2025).

[19] Adam Lotz. *Citrix's Growth and Transformation with Cloud Software Group - Citrix Blogs*. 6th Sept. 2023. URL: https://www.citrix.com/blogs/2023/09/06/one-year-of-citrix-as-part-of-cloud-software-group/ (visited on 03/04/2025).

[20] Joel Marcey. *Facebook and Microsoft Partnering on Remote Development*. 19th Nov. 2019. URL: https://developers.facebook.com/blog/post/2019/11/19/facebook-microsoft-partnering-remote-development/ (visited on 06/02/2025).

[21] Microsoft. *Microsoft Intune—Endpoint Management — Microsoft Security*. URL: https://www.microsoft.com/en/security/business/microsoft-intune (visited on 09/04/2025).

[22] Emerson Murphy-Hill et al. 'What predicts software developers' productivity?' In: *IEEE Transactions on Software Engineering* 47.3 (2019), pp. 582–594.

[23] Abi Noda et al. 'DevEx: What Actually Drives Productivity: The developer-centric approach to measuring and improving productivity'. In: *Queue* 21.2 (May 2023), pp. 35–53. ISSN: 1542-7730. DOI: 10.1145/3595878. URL: https://doi.org/10.1145/3595878.

[24] Okteto Inc. *How Okteto Helped monday.com Accelerate Development Velocity by 50%*. URL: https://www.okteto.com/case-study-monday-2024/ (visited on 19/02/2025).

[25] Okteto Inc. *Improving Development Velocity by Building a Better DevX*. 17th Nov. 2023. URL: https://www.okteto.com/blog/development-velocity/ (visited on 20/02/2025).

[26] Okteto Inc. *Welcome to Okteto! — Okteto Documentation*. URL: https://www.okteto.com/docs/ (visited on 06/02/2025).

[27] Okteto Inc. *Why Development Experience Matters*. 27th Sept. 2022. URL: https://www.okteto.com/blog/why-development-experience-matters/ (visited on 20/02/2025).

[28] Simon Sharwood. *Gartner warns Omnissa represents risk for VMware EUC users • The Register*. URL: https://www.theregister.com/2024/08/28/gartner_omnissa_vmware_euc_strategy/ (visited on 03/04/2025).

[29] Susan Elliott Sim and Richard C Holt. 'The ramp-up problem in software projects: A case study of how software immigrants naturalize'. In: *Proceedings of the 20th international conference on Software engineering*. IEEE. 1998, pp. 361–370.

[30] Stack Exchange Inc. *Professional Developers — 2024 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2024/professional-developers#developer-experience-frustration (visited on 27/02/2025).

[31] Stack Exchange Inc. *Technology — 2024 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment (visited on 10/03/2025).

[32] Starflows OG. *Devfile & devcontainer vs. Dockerfile & Docker-Compose*. 1st Feb. 2024. URL: https://cloudomation.com/en/cloudomation-blog/devfile-devcontainer-vs-dockerfile-docker-compose/ (visited on 11/03/2025).

[33] Denbeigh Stevens. *How Discord Moved Engineering to Cloud Development Environments*. URL: https://discord.com/blog/how-discord-moved-engineering-to-cloud-development-environments (visited on 05/02/2025).

[34] Margaret-Anne Storey et al. 'Towards a theory of software developer job satisfaction and perceived productivity'. In: *IEEE Transactions on Software Engineering* 47.10 (2019), pp. 2125–2142.

[35] The Hacker News. *A Few More Reasons Why RDP is Insecure (Surprise!)* 20th June 2023. URL: https://thehackernews.com/2023/07/a-few-more-reasons-why-rdp-is-insecure.html (visited on 11/04/2025).

[36] The Mitre Corporation. *CVE - Search Results*. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rdp+windows (visited on 11/04/2025).

[37] Jacob Bo Tiedemann and Tanja Bach. *Five proven approaches for a better Developer Experience in your organisation*. 13th Oct. 2021. URL: https://www.thoughtworks.com/en-us/insights/blog/experience-design/approaches-for-a-better-developer-experience (visited on 25/03/2025).

[38] Dave Vellante. *Broadcom's VMware strategy is winning despite market friction - SiliconANGLE*. 13th Nov. 2024. URL: https://siliconangle.com/2024/11/13/broadcoms-vmware-strategy-winning-despite-market-friction-2/ (visited on 03/04/2025).

[39] Christian Weichel and Alejandro de Brito Fontes. *We're leaving Kubernetes - Blog*. URL: https://www.gitpod.io/blog/we-are-leaving-kubernetes (visited on 11/04/2025).

A Sample Requirements

This Appendix presents a set of requirements for remote development environment products, to be modified as appropriate before publication in public tenders. For more context, see Section 4.6, or Section 4 more generally.

A.1 Functional requirements

A.1.1 Access to secondary tools

Confidential tools – such as GitHub and Grafana – MUST be accessible with any solution.

It SHOULD be possible to set up a new user on the system using an automated process which can be validated as secure. Where relevant/possible, any ephemeral infrastructure SHOULD also be configurable automatically.

A.1.2 IDE support

MUST include a range of IDEs: Visual Studio Code and JetBrains IDEs at minimum, and Visual Studio on Windows if possible, as these are the most-used IDEs according to a recent Stack Overflow survey [31].

A.1.3 Support for arbitrary languages & packages

MUST support at least: Java, Dotnet, Go, Javascript / Typescript, Python, Terraform / HCL, bash.

MUST allow any relevant form of dependency management that stems from a trustworthy source citebsi-secure-software-lifecycle, including screened mirrors of third-party repositories and company-internal repositories.

Any third-party dependencies MUST be scanned for vulnerabilities [5].

Viewing and hot reloading a web app during development SHOULD be possible.

To ensure smooth access to legacy tech, it MAY be possible to spin up systems using insecure/deprecated versions of tools. Any such insecure versions SHOULD, however, be strictly restricted to pre-production environments except for extremely rare cases (e.g. urgent hotfixes) and SHOULD require a manual approval process.

A.1.4 Helper tools

An unlimited range of helper tools, e.g. database managers, visual git tools SHOULD be supported.

Docker MUST be usable where needed, e.g. for integration tests.

A.1.5 Single sign-on

Any authentication MUST take place using one of the organisation's central single sign-on mechanisms.

There MUST NOT be any way to access a development environment except via such authentication.

A.1.6 Pair programming

SHOULD be straightforward, between two users working remotely.

A.1.7 Terminal access

MUST be available to any development environment.

A.1.8 Programmatic definition

A configuration-as-code system SHOULD allow setting up new environments.

A.1.9 Access to version control system

MUST be possible via both CLI and GUI, with read and write access, supporting Git as a minimum.

A.1.10 Access to deployment & status monitors

MUST be possible, including CI/CD system's GUI and any other deployment monitors.

A.1.11 Network access

Development-stage network resources (e.g. APIs, databases, Kubernetes clusters, etc.) MUST be accessible.

Network resources in other stages (e.g. production) generally MUST NOT be accessible. A manual approval step SHOULD be available to allow access in exceptional circumstances.

A.1.12 Role-based access control

Users' privileges MUST be controllable using granular roles.

A.1.13 Local testing

Any tests or other processes run in the CI pipeline SHOULD be executable on the development system.

A.2 Non-functional requirements

A.2.1 Auditability

Development environments SHOULD provide ways to verify their integrity and to trace any changes made [5].

Changes (e.g. within version control) SHOULD be marked with their author in a manner that can be cryptographically guaranteed to trace back to the user's environment. Any keys used for such validation MUST be stored and accessed securely. If possible, this SHOULD be provably traceable back to the physical human, though many systems will not support this [5].

A.2.2 Availability

The new system SHOULD function at any time of day; with a maximum of 5 minutes of startup time after any period of inactivity-related downtime; and should be available 99.99% of the time.

A.2.3 Hardened access

Any user-accessible endpoints MUST be hardened and secured [5].

A.2.4 Hosted privately

Third-party cloud providers SHOULD NOT be needed to run any aspect of the solution.

Deployment SHOULD be possible to an environment without access to the Internet. Even authentication SHOULD take place via a VPN.

A.2.5 Scalable

There SHOULD NOT be a discernible difference between the system supporting one developer or fifty.

A.2.6 Rapid onboarding

T new user SHOULD be able to contribute to an existing repository within one hour.

A.2.7 Restricted infrastructure

Access to any underlying infrastructure MUST be strictly restricted, with only dedicated maintenance teams allowed under clearly defined, exceptional circumstances. Such access MUST be audited to the same standard as that of end users.

A.3 Commercial requirements

A.3.1 Predictable performance

Any commercial product SHOULD include 24/7 Enterprise-grade support, along with clear SLAs.

A.3.2 Predictable pricing

Third-party product usage SHOULD be charged based on easy calculated metrics such as number of users.

A.3.3 Certified providers

Any third-party provider MUST be certified with ISO27001 [6].